
# Rapport d'Optimisation Métaheuristique

Fabien GRONDIN, Ambre MAURICE, Ugo STELLA

ING3 IAB

Mars 2023

# Table des matières

# 1 Introduction

Le Cloud computing est une technologie indispensable de nos jours. Elle permet de stocker ou accéder à des données et applications, de n'importe où. De plus, elle réduit les coûts car l'infrastructure physique est mutualisée ; les utilisateurs peuvent se concentrer sur leurs applications et non sur la mise en place et la maintenance de l'infrastructure sous-jacente. L'utilisateur paie la location de l'infrastructure comme un service et non pas comme un investissement initial. Les serveurs Cloud sont centralisés dans des data centers ; ce qui peut créer de la latence en fonction de la localisation de l'utilisateur par rapport aux data centers. C'est pourquoi, le Cloud fonctionne en collaboration avec le Fog computing, afin de faire fonctionner les applications des utilisateurs. Le Fog computing désigne des machines dispersées un peu partout et proches de l'utilisateur ; ce qui réduit le temps de traitement.

Pour que les applications dans le Cloud fonctionnent de façon optimale, il est important de s'intéresser à la répartition des tâches sur les différentes machines. Ainsi, dans ce projet, nous tentons d'optimiser la répartition des tâches sur des machines virtuelles, ayant des caractéristiques propres afin de minimiser la latence, le coût d'exécution et de maximiser la fiabilité.

Tout d'abord, nous décrirons le problème sous forme de modèle mathématique. Puis, nous expliquerons l'approche NSGA-II, utilisée pour résoudre le problème. Enfin, nous analyserons nos expérimentations et les résultats obtenus.

# 2 Modèle mathématique

Dans ce modèle, nous supposons que nous avons $n$ tâches à répartir sur $m$ machines virtuelles $V_M$. Nous ne prendrons pas en compte l'ordre d'exécution des tâches.

L'ensemble des tâches sera noté $T = \{T_1, T_2, ..., T_n\}$
Chaque tâche $T_i$ aura un nombre d'instructions $I(T_i)$.

L'ensemble des machines virtuelles sera noté $V_M = \{V_{M_1}, V_{M_2}, ..., V_{M_m}\}$
Chaque machine virtuelle $V_{M_j}$ aura :

- une vitesse de calcul (computing rate) $C_r(V_{M_j})$ en milliers d'instructions par seconde.
- un coût d'utilisation par seconde $C_u(V_{M_j})$ en euros.
- un taux d'échec de la machine virtuelle $\lambda_j$.
- un temps de transfert entre la VM et la machine qui envoie les tâches : $T_{net}(V_{M_j})$ en secondes.

Une solution sera représentée de la façon suivante : $S = \{T_1^{j_1}, T_2^{j_2}, ..., T_n^{j_n}\}$ avec les $T_i^j$ indiquant que la tâche $i$ est effectuée par le serveur $j$. (Ici, on note $j_1, j_2, ...j_n$ pour montrer que les serveurs sont différents)

**Variables de décision :** les variables de décision sont les $T_i^j$ indiquant que la tâche $i$ est effectuée par le serveur $j$.

$$X = \{T_1^{j_1}, T_2^{j_2}, ..., T_n^{j_n}\}$$

**Paramètres :** les paramètres sont les éléments décrits plus haut : $I(T_i)$, $C_r(V_{M_j})$, $C_u(V_{M_j})$, $\lambda_j$, $T_{net}(V_{M_j})$

On pourra, à partir de ces paramètres, calculer $T_{ex}$, le temps d'exécution d'une tâche sur une VM. $T_{ex}$ nous permettra de calculer les 3 éléments à optimiser :

— le coût d'exécution $C_{ex}(X)$
— la fiabilité $F(X)$ qui correspond à la probabilité que les processus s'exécutent sans problème
— la latence $L(X)$ qui est la somme du temps d'exécution des instructions et du temps de transfert

Ci-dessous, les calculs :

$$T_{ex}(T_i^j) = \frac{I(T_i)}{C_r(V_{M_j})}$$

$$C_{ex}(X) = \sum_{i=1}^{n} T_{ex}(T_i^j) \times C_u(V_{M_j})$$

$$F(X) = Exp^{-\sum_{i=1}^{n} T_{ex}(T_i^j) \times \lambda_j}$$

$$L(X) = \sum_{i=1}^{n} (T_{ex}(T_i^j) + T_{net}(V_{M_j}))$$

**Fonctions objectif :** pour les variables de décision $X = \{T_1^{j_1}, T_2^{j_2}, ..., T_n^{j_n}\}$ nous souhaitons :

$$Minimiser : C_{ex}(X)$$
$$Maximiser : F(X) \Leftrightarrow Minimiser : -F(X)$$
$$Minimiser : L(X)$$

**Contraintes :**

$$\forall i \in [\![1, n]\!], \exists j \in [\![1, m]\!] \; tel \; que \; T_i^j \in X$$
$$I(T_i) > 0, \forall \, T_i$$
$$C_r(V_{M_j}) > 0, \forall \, V_{M_j}$$
$$C_u(V_{M_j}) > 0, \forall \, V_{M_j}$$
$$\lambda_j > 0, \forall \, V_{M_j}$$

# 3 Approche de résolution NSGA-II

Afin de résoudre notre problème multi-objectif, nous avons décidé d'utiliser le Non-dominated Sorting Genetic Algorithm (NSGA-II ou NSGA2), qui nous semblait adapté. C'est un algorithme génétique avec des fonctions de sélection modifiées pour la reproduction et la survie. Dans cette partie, nous allons décrire les étapes les plus importantes de cette algorithme. Ci-dessous, le schéma de l'ordre des étapes :

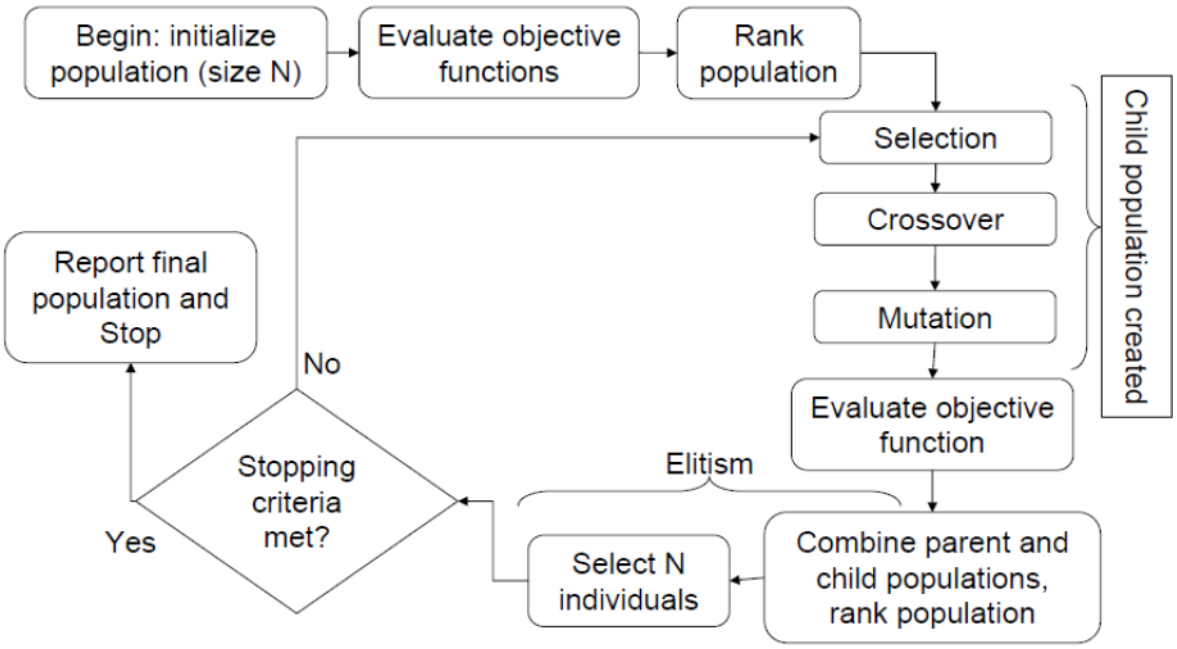



FIGURE 1 – Schéma de l'ordre des étapes du NSGA-II

Tout d'abord, nous initialisons la population de taille N et nous faisons une première évaluation de la fonction objectif. Puis :

- Nous commençons par appliquer les opérations de crossover et de mutation, afin de générer de nouveaux individus.
- Nous devons ensuite combiner les solutions parents et enfants ; ce qui donne 2N solutions par reproduction ; ces enfants vont chacun avoir une mutation dans leurs gènes.
- Nous allons, après la reproduction, donner un rang aux individus de la population pour les classer.
- Nous revenons à N solutions. Pour cela nous sélectionnons celles qui sont les plus éparpillées et surtout de rang plus élevé. L'éparpillement permet de garder une bonne répartition et le tri par rang permet de rapprocher nos solution de la frontière de Pareto.
- On recommence jusqu'à ce que le critère d'arrêt soit atteint. Dans notre cas, le critère d'arrêt est un nombre d'itérations.

## 3.1 Première génération des individus et évaluation des fonctions objectif

Les individus correspondent à l'association d'une tâche et d'une machine virtuelle. Dans le code, ce sont des instances de la classe "IndividualClass". Cette classe possède comme attributs : le coût d'exécution, la fiabilité, la latence, le rang et la crowding distance. Ces deux derniers ne sont pas définis lors de la création d'un individu, mais les autres attributs le sont. Ainsi, l'étape d'évaluation des fonctions objectif est effectuée, pour chaque individu, à sa création. Ci-dessous, le code de la classe :

```python
#Class for individual
class IndividualClass:
    def __init__(self,individual):
        self.individual = individual
        self.CostEx = Cex(individual)
        self.Reliability = F(individual)
        self.Latency = L(individual)
        self.Rank = None
        self.Crowding_distance=None

    def setRank(self, rank):
        self.Rank = rank

    def show(self):
        string="["
        for DV in self.individual:
            string+='{'+str(DV.TaskId)+','+str(DV.VMId)+'}'
        string+="]"
        print(string)
```

Lors de la création de la population de base, des instances de la classe "IndividualClass" sont créées aléatoirement, en associant les tâches à des machines virtuelles.

## 3.2 Sélection, crossover et mutation

Les étapes de sélection, crossover et de mutation permettent de sélectionner des parents et de créer de nouveaux individus à partir des parents.

Pour sélectionner les parents de la reproduction, nous parcourons tous les individus $i$ de la population et les reproduisons avec l'individu $(i+1)\%N$. Une fois les parents sélectionnés, nous avons décidé que l'enfant prendrait aléatoirement 1 gène de chaque parent. Puis, une mutation aléatoire intervient sur l'un des gènes de l'individu créé.

## 3.3 Classement de la population

Une solution ou individu $x_1$ domine un individu $x_2$, s'il est meilleur sur tous les critères.

Les solutions Pareto-optimales ne sont dominées par aucune autre solution et sont situées sur la frontière de Pareto.

Pour déterminer le rang des individus, on va récupérer toutes les solutions non dominées entre elles et leur donner le rang 1. Puis, on retire les solutions

de rang 1 de la recherche et on recommence pour trouver les solutions de rang 2. On répète ces opérations, jusqu'à ce que tous les individus aient un rang.

## 3.4 Élitisme : combinaison des parents et sélection d'individus

A la fin des opérations de crossover et de mutation, nous avons 2N individus, mais nous devons sélectionner seulement N individus.

Pour revenir à N solutions, on va tout d'abord récupérer les individus dans les premiers rangs. Puis, on utilise la distance de Crowding pour compléter la population.

La distance de Crowding est utilisée pour déterminer la distance entre une solution et son voisinage (hypervolume). En effet, ici, on veut récupérer les solutions les plus éloignées entre elles sur la frontière de Pareto.

Pour calculer la distance de Crowding :
- On initialise les distances à 0 pour toutes les solutions
- On trie les solutions pour l'objectif 1
- La première et la dernière solution prennent la distance infinie. Pour les autres, on a $dist(i) = dist(i) + f1(i+1) - f1(i-1)$ avec $f_j$ le score pour l'objectif $j$.
- On recommence le tri et les étapes pour le prochain objectif (quand on commence à s'intéresser au deuxième objectif, les $dist(i)$ auront déjà des valeurs différentes de 0).

## 3.5 Explication de certains passages du code

Tout comme les individus, les tâches, les machines virtuelles et les variables de décisions sont des classes. Cela permet de très facilement charger des données dans ces objets et d'accéder à l'ensemble de leurs attributs très simplement. La classe TaskClass possède l'attribut nbInstruction, la classe VMClass elle contient ComputerRate, UseCost, FailureRate et NetworkTime. DVClass est la classe d'un gène, ou d'une variable de décision, et va donc associer une machine virtuelle à une tâche. Les attributs des classes filles sont rappelés dans cette classe afin d'y accéder plus facilement.

```
####Classes

#Class for task
class TaskClass:
    def __init__(self,Id,nbInstruction):
        self.Id=Id
        self.nbInstruction=nbInstruction

#Class for virtual machine
class VMClass:
    def __init__(self,Id,ComputerRate,UseCost,FailureRate,NetworkTime):
        self.Id=Id
        self.ComputerRate=ComputerRate
        self.UseCost=UseCost
        self.FailureRate=FailureRate
        self.NetworkTime=NetworkTime

#Class for decision variable
class DVClass:
    def __init__(self,Task, VM):
        self.Task=Task
        self.VM=VM
        self.TaskId=Task.Id
        self.nbInstruction=Task.nbInstruction
        self.VMId=VM.Id
        self.ComputerRate=VM.ComputerRate
        self.UseCost=VM.UseCost
        self.FailureRate=VM.FailureRate
        self.NetworkTime=VM.NetworkTime
```

Les paramètres sont définies par des fonctions. Elles dépendent des variables de décision, donc des tâches et des machines virtuelles. Elles sont la traduction algorithmique des fonctions du modèle mathématique. On notera que le Coût d'utilisation est divisé par 60 pour faire la conversion de seconde en minute. Ces fonctions sont directement appelés lors de l'initialisation des classes (voir l'objet IndividualClass plus haut)

```
#T_ex(Tij) or Execution Time
def Tex(T):
    return(T.nbInstruction/T.ComputerRate)

#C_ex(X) or Execution Cost
#The division by 60 is to convert useCost(min) into useCost(sec)
def Cex(X):
    res=0
    for DV in X:
        res+=Tex(DV)*DV.UseCost/60
    return(res)

#F(X) or Reliability
def F(X):
    res=0
    for DV in X:
        res+=(Tex(DV)*DV.FailureRate)
    return (exp(-1*res))

#L(X) or Latency
def L(X):
    res=0
    for DV in X:
        res+= Tex(DV)+DV.NetworkTime
    return(res)
```

En ce qui concerne les croisements d'individus, nous sélectionnons la moitié du nombre de variable de décision au hasard dans le génome d'un des individus à croiser. Nous sélectionnons l'ensemble inverse chez le deuxième parent. Nous créons un nouveau individu à partir de ces deux moitiés de génome. Nous générons enfin une mutation aléatoire à chaque naissance d'un nouveau individu, ce qui va nous permettre d'explorer les différentes solutions possibles.

```
#crosses randomely genes of two parents to make a baby
#one gene is randomely muteted in the process
def breeding(mom,dad,Tasks,VMs):
    nT=len(Tasks)
    nVM=len(VMs)
    random_genes=sample(range(nT),k=nT//2)
    random_mutation=randint(0,nT-1)
    baby=[]
    for i in range(nT):
        if (i in random_genes):
            baby.append(mom.individual[i])
        else:
            baby.append(dad.individual[i])
        if (i==random_mutation):
            baby[i]=DVClass(Tasks[i],VMs[randint(0,nVM-1)])
    return IndividualClass(baby)
```

Crowding_distance_sorting est une très grosse fonction, mais elle est composée de la même sous fonction qui se répète plusieurs fois pour chaque paramètre à optimiser. On commence par initialiser le vecteur des distances à zéro avec infinie en tête et en queue. Ensuite, pour chaque paramètre :

- Nous trions la population par le paramètre P à tester
- Nous calculons les valeurs maximum et minimum $P_(max)$ et $P_(min)$ de ces paramètres
- Nous calculons $distance[i] = \frac{P(X[i-1])-P(X[i+1])}{P\_(max)-P\_(min)}$
- Nous ajoutons la distance de crowding à tout les individus
- Nous répétons jusqu'au dernier paramètre

```python
#calculate the crowding distance for each individual
def crowding_distance_sorting(sub_group, removeParameter=False):
    n=len(sub_group)
    #initialising the crowding distances to zero
    for individual in sub_group:
        individual.Crowding_distance = 0
    #initialising the distance list to list of 0 and infinity for first and last distance
    distances=[0 for i in range(n)]
    distances[0]=float("inf")
    distances[-1]=float("inf")
    #we iterate for each parameters
    if (not removeParameter=="CostEx"):
        #we sort the population by the parameter CostEx
        sub_group = sorted(sub_group, key=lambda individual: individual.CostEx)
        f_min = sub_group[0].CostEx
        f_max = sub_group[-1].CostEx
        #this prevents dividing by zero
        if (f_max != f_min):
            #we calculate the crowding distance for each individual for the parameter CostEx
            for i in range(1,n-1):
                distances[i] += (sub_group[i+1].CostEx - sub_group[i-1].CostEx) / (f_max - f_min)
        #we assign the crowding distance to the individuals
        for i in range(n):
            sub_group[i].Crowding_distance = sub_group[i].Crowding_distance+distances[i]
        distances=[0 for i in range(n)]
        distances[0]=float("inf")
        distances[-1]=float("inf")
    #we repeat for Reliablity
    if (not removeParameter=="Reliability"):
        sub_group = sorted(sub_group, key=lambda individual: individual.Reliability)
        f_min = sub_group[0].Reliability
        f_max = sub_group[-1].Reliability
        if (f_max != f_min):
            for i in range(1,n-1):
                distances[i] += (sub_group[i+1].Reliability - sub_group[i-1].Reliability) / (f_max - f_min)
        for i in range(n):
            sub_group[i].Crowding_distance = sub_group[i].Crowding_distance+distances[i]
        distances=[0 for i in range(n)]
        distances[0]=float("inf")
        distances[-1]=float("inf")
    #We repeat for Latancy
    if (not removeParameter=="Latency"):
        sub_group = sorted(sub_group, key=lambda individual: individual.Latency)
        f_min = sub_group[0].Latency
        f_max = sub_group[-1].Latency
        if (f_max != f_min):
            for i in range(1,n-1):
                distances[i] += (sub_group[i+1].Latency - sub_group[i-1].Latency) / (f_max - f_min)
        for i in range(n):
            sub_group[i].Crowding_distance = sub_group[i].Crowding_distance+distances[i]
        distances=[0 for i in range(n)]
        distances[0]=float("inf")
        distances[-1]=float("inf")
    #we now sort the population by crowding distance order
    sub_group = sorted(sub_group, key=lambda individual: individual.Crowding_distance, reverse=True)
    return sub_group
```

Ceci est la fonction pour créer les sous-groupes regroupés par domination. Elle fonctionne en trouvant la frontière de Pareto d'un ensemble. Puis en retirant cette frontière pour la mettre dans un sous groupe d'éléments à qui on donne un même rang, on calcule la nouvelle frontière de Pareto et ainsi de suite jusqu'à parcourir l'ensemble de la population.

```
#Create subgroups sorting individual by ranks
#an individual is in a rank i if he is not dominated by any indivual in a rank lower
#and if there is at least one individual in the rank i-1 dominating him
def create_subgroups(population,removeParameter=False):
    #we need a copy of the population so poping individual during the
    #domination check algorithm does not falsify the results
    deepcopyPop=deepcopy(population)
    subgroups=[]
    #until every one as been ranked
    while(len(population)>0):
        subgroup=[]
        #for each individual
        for i in reversed(range(len(population))):
            boolean=True
            Xi=population[i]
            #we check every other individual in the popultion
            for j in range (len(population)):
                Xj=population[j]
                #and check for domination
                if (removeParameter=="CostEx"):
                    if (Xi.Reliability < Xj.Reliability and Xi.Latency > Xj.Latency):
                        boolean=False
                        break
                elif (removeParameter=="Reliability"):
                    if (Xi.CostEx > Xj.CostEx and Xi.Latency > Xj.Latency):
                        boolean=False
                        break
                elif (removeParameter=="Latency"):
                    if (Xi.CostEx > Xj.CostEx and Xi.Reliability < Xj.Reliability):
                        boolean=False
                        break
                else:
                    if (Xi.CostEx > Xj.CostEx and Xi.Reliability < Xj.Reliability and Xi.Latency > Xj.Latency):
                        boolean=False
                        break
            #if no individual dominates this one
            if (boolean):
                #we add the individual to the sublist and we pop it from the copy of population
                subgroup.append(deepcopyPop.pop(i))
        subgroups.append(subgroup)
        #we update population by removing the freshly ranked individuals
        population=deepcopy(deepcopyPop)
    return(subgroups)
```

Nous allons maintenant terminer par la fonction qui servira de fonction principale pour exécuter l'ensemble des sous-fonctions expliquer précédemment. Nous commençons par lire notre base de données, puis nous créons la première population aléatoirement. Puis jusqu'au critère d'arrêt qui est la fin de notre boucle for, nous enchaînons la suite de fonction suivante :

— Nous mélangeons la population avant de faire les croisements.
— Nous faisons la reproduction.
— Nous assignons un rang à chaque individu.
— Nous trions les individus par rang puis par crowding distance.
— Nous faisons la sélection naturelle .
— Nous affichons le graphe (à certaines itérations seulement).

```
def NSGA_II(population_size=200,nb_generations=100, removeParameter=False, show_convergence=False):
    #reads the files' data and load them in classes
    Tasks,VMs=generateTasksAndVMs()
    #generates a random generation of individuals for initialisation
    population=generate_random_population(population_size,Tasks,VMs)
    #displays some datas
    display_means(population)
    #start of the simulation
    for i in range(nb_generations):
        #regularly prints the steps of progression of the algorithm
        if (i%10==0):
            print("étape "+str(i))
        #randomly shuffles the population before reproduction
        shuffle(population)
        #reproduction of the population
        population=reproduction(population,Tasks,VMs)
        #creates the sub groups by rank
        rankedPop=create_subgroups(population,removeParameter)
        #assignes a rank to each individual
        rankedPop=giveRank(rankedPop)
        #sorts the population by crowding distance
        rankedPop=sort_sub_group(rankedPop,removeParameter)
        #selection of featest individuals
        population=natural_selection(population_size, rankedPop)
        #generateq a graph for some of the steps
        if (i==0 or i==nb_generations-1 or show_convergence and(i==2 or i==4 or i==6 or i==10)):
            savefileName="_generation_n-"+str(i)
            if removeParameter:
                savefileName= "SavedPlots/2D_Plot_"+removeParameter+"_ignored"+savefileName
            else:
                savefileName= "SavedPlots/3D_Plot"+savefileName
            show_Domination_Plot(population,removeParameter,save=savefileName)
            print(savefileName)
    #displays some data at the end of the simulation
    print("average of final generation parameters:")
```

En réalisant le projet nous souhaitions faciliter la manipulation des algorithmes en ajoutant plusieurs options à notre fonction principale :

— La possibilité de choisir le nombre d'individus dans la population, 200 par défaut.
— La possibilité de choisir le nombre de générations, 100 par défaut.

— La possibilité de retirer une variable de décision afin d'afficher un graphe en 2D, il suffit d'entrer en paramètre removeParameter="CostEx""Reliability" ou "Latency". Par défaut ou si mauvaise input la fonction affichera un graphe en 3D.
— La possibilité d'afficher les graphes de plus des premières générations afin de mieux pouvoir observer la convergence vers la frontière de Paleto avec show_convergence=True. Par défaut à False.

# 4 Expérimentations et résultats

## 4.1 Environnement de simulation et paramètres

Pour représenter notre modèle de répartition des tâches, nous avons utilisé un environnement IaaS Cloud (Infrastructure as a Service), à l'aide des données des machines virtuelles fournies par Mme Yassa.

Ainsi, nous disposons dans un premier temps de trois machines virtuelles avec des attributs de fréquence d'échec, de coût, de vitesse de calcul ainsi que de délai réseau différents comme montré ci-dessous :

| FailureRate | UseCost | ComputerI | NetworkTime |
|---|---|---|---|
| 0.0000001 | 0.92 | 5400 | 0.123 |
| 0.0000002 | 0.57 | 4800 | 0.453 |
| 0.000001 | 1.14 | 8500 | 0.914 |

Nous disposons également de 15 tâches virtuelles définies, avec un grand nombre d'instructions variable :

| nbInstruction |
|---|
| 300000 |
| 600000 |
| 600000 |
| 900000 |
| 300000 |
| 150000 |
| 150000 |
| 300000 |
| 300000 |
| 300000 |
| 600000 |
| 300000 |
| 600000 |
| 150000 |
| 300000 |

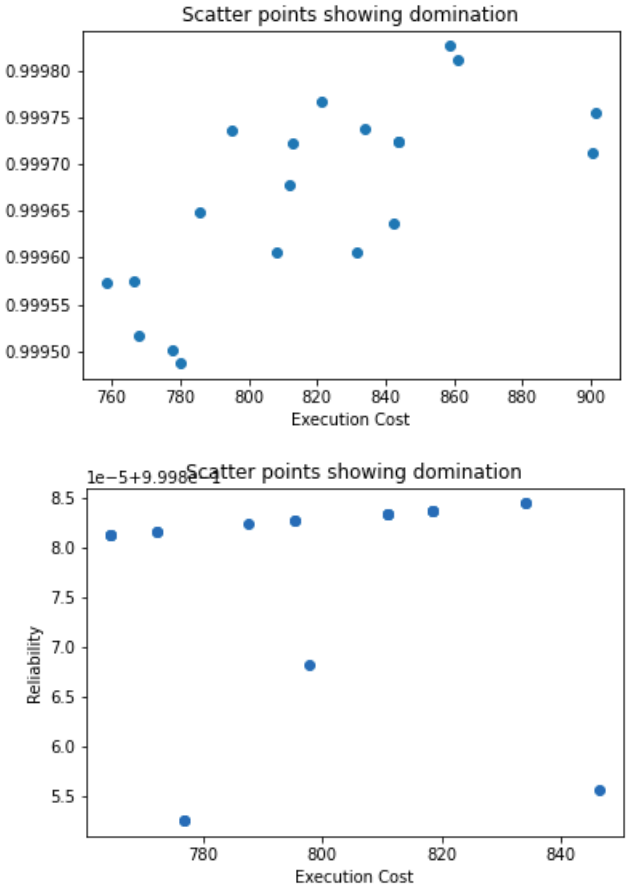
En plus des informations de l'environnement données précédemment, nous avons défini certains paramètres de la métaheuristique, nécessaires au calcul de l'optimisation à faire. Ainsi, nous avons :

— le nombre d'itérations effectuées : 100,
— la taille de la population : 200,
— la probabilité d'apparition d'une mutation : 100% (chaque enfant né a un gène aléatoire qui mute)

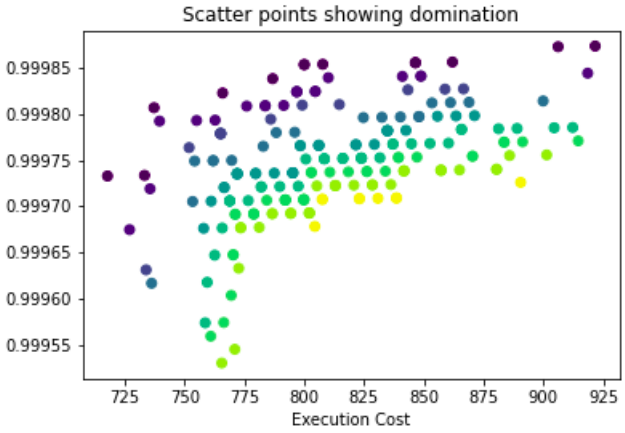
## 4.2 Tests effectués

Dans un premier temps, nous avons construit l'algorithme évolutif ; puis nous avons ajouté le calcul des rangs, avant de représenter graphiquement ce que nous obtenions pour vérifier que notre proposition était bonne.
L'algorithme tourne en 45 secondes en le laissant jusqu'à la dernière itération.

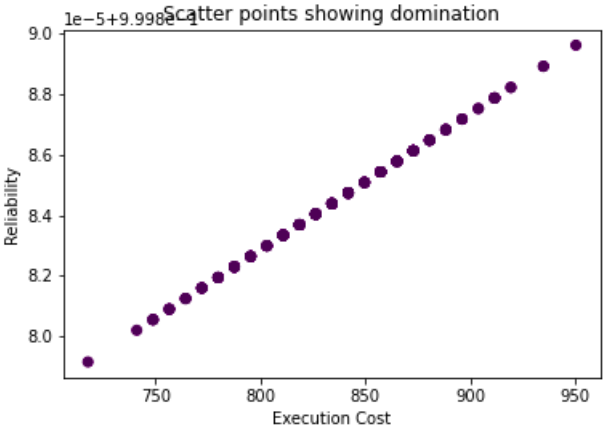
Il s'est avéré que que ça ne nous donnait que des graphes en 2 dimensions, au lieu de graphes en 3 dimensions selon nos 3 paramètres à optimiser, ce qui provenait du modèle de données. Nous avons également eu des gènes mutants qui semblent ne pas avoir passé la sélection naturelle, comme le montre les points "anormaux" des graphes ci-dessous :



Nous avons, ci-dessous, la représentation des rangs à la génération 1, allant du plus petit à 1 au niveau du violet pour les plus dominants, jusqu'aux rangs les plus élevés, qui sont de plus en plus clairs. Nous pouvons constater que le rang 1 est cohérent , étant donné que la fiabilité est à maximiser, et le coût d'exécution, à minimiser :

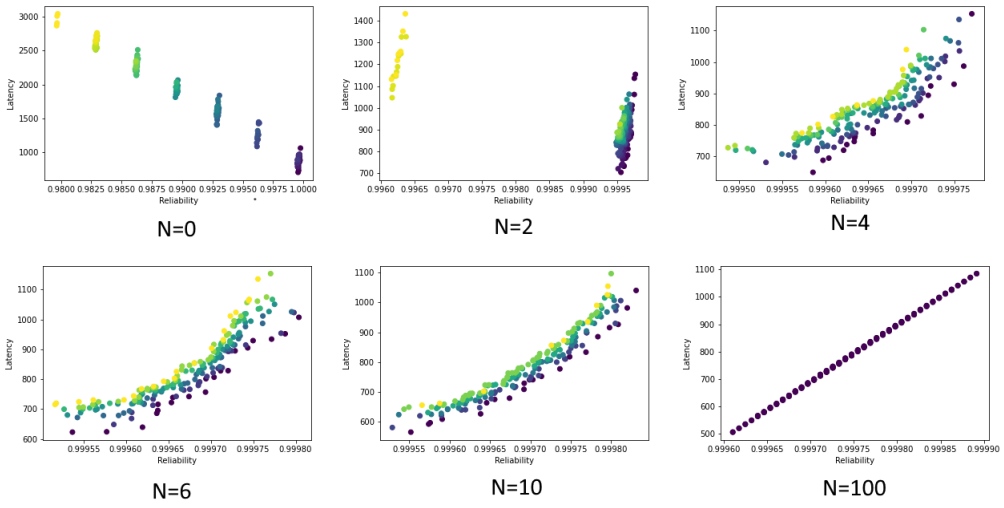



Après plusieurs itération de notre algorithme génétique, on observe bien une convergence de nos variable de décision vers un espace bordure. Cette bordure est la bordure de Pareto, et c'est donc ici que l'on trouve les solutions qui minimisent le mieux nos fonctions objectifs.

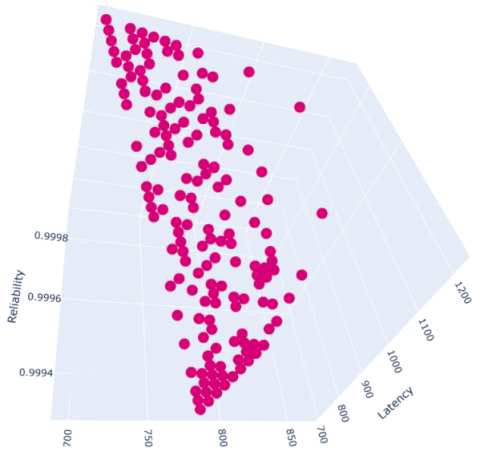



Nous avons enfin ajouter la fonction de crowding distance sorting, qui a permis d'éviter les extremums locaux même si dans notre modèle il ne semble

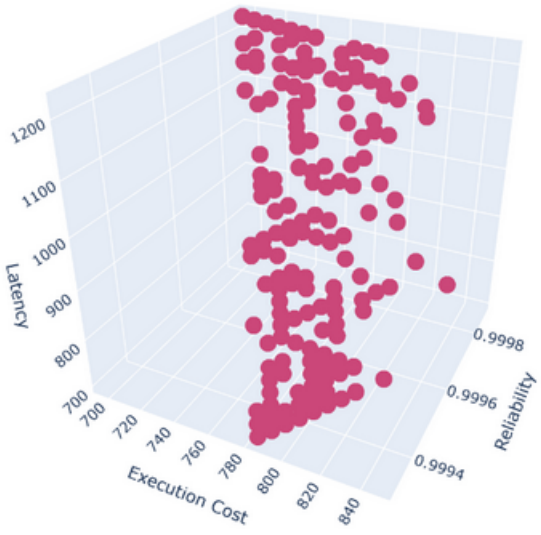
pas en avoir, et ainsi finir la recherche des solutions pareto-optimales à l'aide de NSG-II Nous pouvons observer que l'algorithme converge bien également avec 2 autres choix de critères : ici la Latency ajoutée et le Coût d'exécution, n correspondant au nombre d'itérations de l'algorithme.




Nous avons pu également représenter les points en 3 dimensions selon les objectifs, mais ils sont tous de rang 1, étant sur un même plan. On expliquera l'origine de ce problème page suivante. Il semblerait que l'ensemble des solutions possibles avec nos trois machines sont pratiquement toutes optimales à quelques exceptions près, qui n'ont pas été gardées après la 1ère sélection.

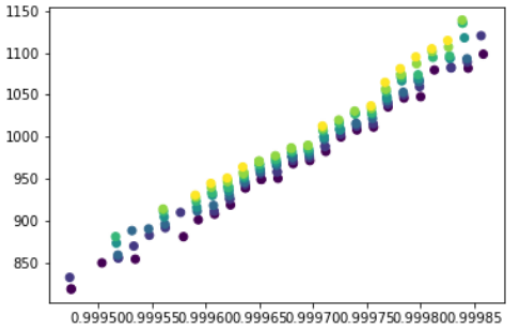


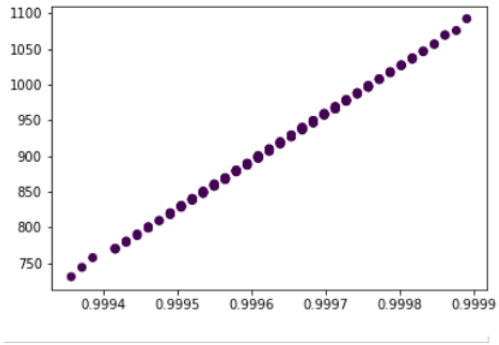

Nous obtenons ainsi notre ensemble Pareto-optimum de nos 3 objectifs à optimiser, qui est sous la forme d'un plan :




En appliquant l'algorithme pour la latence et la fiabilité pour comprendre la forme de plan, nous constatons que les deux sont quasiment corrélées ; ce qui explique qu'à l'itération zéro les solutions sont très proches de former une droite. Cela explique que dès la première itération l'ensemble des solutions

se trouve sur le plan Pareto-optimal.

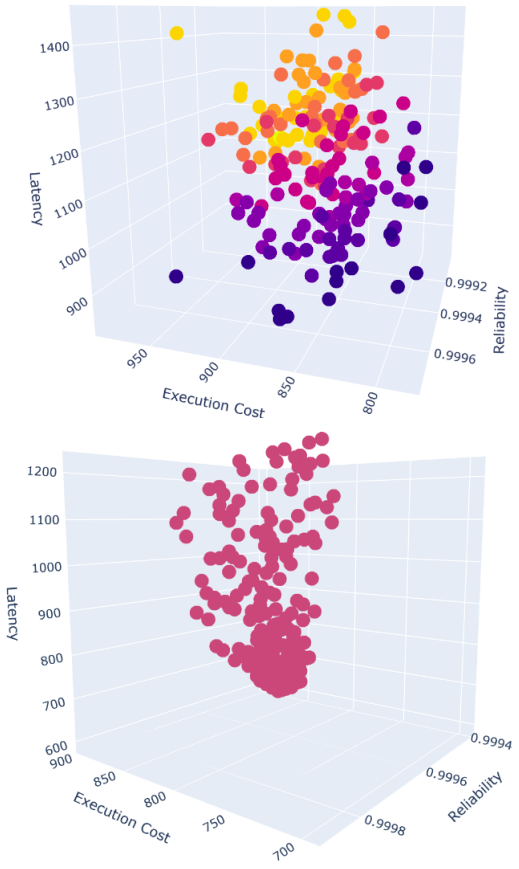


Pour tenter d'obtenir de meilleurs résultats, plus visuels, nous avons décidé d'essayer d'ajouter 2 nouvelles machines virtuelles à notre environnement, à partir de valeurs réalistes. L'ensemble des VMs est donc configuré ainsi :

| FailureRate | UseCost | ComputerRate | Latency |
|---|---|---|---|
| 1E-07 | 0.92 | 5400 | 0.123 |
| 1E-07 | 0.57 | 4800 | 0.453 |
| 1E-06 | 1.14 | 8500 | 0.914 |
| 1E-06 | 1.5 | 10000 | 0.516 |
| 1E-06 | 0.4 | 2000 | 0.1 |

Nous obtenons ainsi plus de diversité dans les solutions possibles, ce qui est mis en évidence par un plus grand nombre de rangs représentés avec des couleurs différentes, et toujours une convergence vers les solutions Pareto-optimales :



Dans le code, nous affichons aussi quelques solutions. Par exemple, la meilleure solution par rapport au coût d'exécution, par rapport à la latence ou par rapport à la fiabilité.

```
average of final generation parameters:
10.593808449074075 0.9997387953176373 826.7209022222228
exemples of some of the fittest individual:
[{0,4}{1,4}{2,4}{3,4}{4,4}{5,4}{6,4}{7,4}{8,4}{9,4}{10,4}{11,4}{12,4}{13,4}{14,4}]
5.443750000000001 0.9996100760401144 505.6799999999999
[{0,4}{1,4}{2,0}{3,4}{4,1}{5,0}{6,1}{7,1}{8,0}{9,0}{10,4}{11,4}{12,1}{13,4}{14,0}]
10.009143518518519 0.9997232327515676 834.0045555555554
[{0,0}{1,0}{2,4}{3,4}{4,4}{5,0}{6,1}{7,4}{8,1}{9,4}{10,1}{11,4}{12,4}{13,0}{14,1}]
9.15011574074074 0.9997015723161802 789.2602222222221

best solution for CostEx is :
[{0,4}{1,4}{2,4}{3,4}{4,4}{5,4}{6,4}{7,4}{8,4}{9,4}{10,4}{11,4}{12,4}{13,4}{14,4}]
5.443750000000001 0.9996100760401144 505.6799999999999
best solution for Latency is :
[{0,4}{1,4}{2,4}{3,4}{4,4}{5,4}{6,4}{7,4}{8,4}{9,4}{10,4}{11,4}{12,4}{13,4}{14,4}]
5.443750000000001 0.9996100760401144 505.6799999999999
best solution for Reliability is :
[{0,0}{1,0}{2,0}{3,0}{4,0}{5,0}{6,4}{7,0}{8,0}{9,1}{10,0}{11,0}{12,0}{13,0}{14,0}]
16.066666666666666 0.9998775075028187 1078.2640000000001
```

Une solution est un ensemble de couples (tâche,machine virtuelle). En dessous, nous affichons le coût, la fiabilité et la latence.

# 5 Conclusion

Pour conclure, nous avons eu l'occasion d'expérimenter de nombreuses simulations en faisant varier les paramètres minimiser ainsi que le nombre de machines virtuelles
Nous avons pu voir qu'augmenter le nombre de machines virtuelles pouvaient apporter une plus grande diversité de solutions et qu'un faible nombre, avec nos valeurs initiales pouvait apporter une diversité si faible que l'ensemble des combinaisons possible est solution.

Nous avons également constaté que avoir des variables corrélées dans un problème d'optimisation méta-heuristique pouvait grandement impacter l'efficacité de l'algorithme NSGA-II. Étant donné que l'on cherche à se rapprocher de la frontière de Pareto et non à trouver un ensemble de solutions qui serait un compromis, l'algorithme est peu intéressant dans un scénario où les solutions se trouvent déjà sur cette frontière.

Ce modèle optimisé de répartition des tâches peut permettre une bonne allocation des ressources, notamment dans les environnements Cloud qui le nécessitent principalement ; ce qui concerne beaucoup de domaines différents comme la santé, la bio-ingénierie ou encore les finances.

Il faudrait néanmoins le comparer à d'autres méthodes d'optimisation métaheuristique pour définir la méthode la plus efficace, selon les cas de gestion, car NSGA-II ne nous donne pas de solution compromis qui minimise le plus possible l'ensemble des variables.