Optimisation évolutionnaire parallèle

Fabien Teytaud

To cite this version:

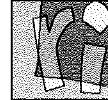
Fabien Teytaud. Optimisation évolutionnaire parallèle. Algorithme et structure de données [cs.DS]. 2008. dumas-00350003

HAL Id: dumas-00350003
https://dumas.ccsd.cnrs.fr/dumas-00350003

Submitted on 5 Jan 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



STAGE DE MASTER 2 RECHERCHE EN INFORMATIQUE

Optimisation évolutionnaire parallèle

Auteur :
Fabien TEYTAUD

Maître de stage :
Dr. Anne AUGER

Organisme d'accueil :
Équipe projet Tao, INRIA Saclay,
Université de Paris Sud, LRI

Secrétariat - tél : 01 69 15 75 18 Fax : 01 69 15 42 72
courrier électronique : Jacques.Laurent@lri.fr
12 mars – 11 septembre 2008

Contents

Contents i

1 Introduction 1

2 État de l'art de l'optimisation évolutionnaire 2
- 2.1 Les différentes approches 2
- 2.2 Maître-esclave 2
- 2.3 Approches Fine-grained 3
- 2.4 Algorithmes à populations multiples 4
- 2.5 Les algorithmes hybrides 4

3 Étude d'algorithmes séquentiels sur la fonction sphère 7
- 3.1 Pas optimal (σ^*) 8
- 3.2 Self-Adaptation 9
- 3.3 Cumulative Step-size Adaptation 10
- 3.4 Règle des $\frac{1}{5}$ 12
- 3.5 Synthèse 12

4 Étude d'algorithmes asynchrones sur la fonction sphère 14
- 4.1 Pas optimal (σ^*) 14
- 4.2 Cumulative Step-size Adaptation 15
- 4.3 Règle des $\frac{1}{5}$ 16
- 4.4 Délai 16

5 Conclusion et perspectives 20

Références 22

Résumé

Dans ce rapport nous nous intéressons à la parallélisation d'algorithmes évolutionnaires afin d'optimiser des fonctions objectifs coûteuses. Nous travaillons sur la fonction sphère, avec des algorithmes non élitistes de type $(1, \lambda)$. Nous constatons qu'un des inconvénients est qu'à partir d'une certaine taille de population l'algorithme admet une perte d'efficacité. Nous proposons une alternative qui correspond au modèle $(1, 1 * \tau)$. Nous comparons les deux modèles pour différentes tailles de population, puis simulons une parallélisation asynchrone pour le modèle $(1, 1 * \tau)$.

Mots clefs

optimisation, stratégies d'évolution, taux de convergence, parallélisme.

CHAPITRE 1

Introduction

Le but de l'optimisation est de trouver la meilleure solution pour un critère donné à partir d'un espace de recherche. L'espace de recherche contient l'ensemble des solutions possibles, alors que le critère (également appelé fonction objectif) permet d'évaluer une solution.
Formellement, soit $F : \mathbb{R}^d \rightarrow \mathbb{R}$, il s'agit de trouver

$$x^* \in \Omega \text{ tel que } x^* = ArgMin(F).$$

dans le cas d'une minimisation.
Il existe plusieurs types d'algorithmes pour résoudre un problème d'optimisation. On peut séparer ces algorithmes en deux groupes. Tout d'abord on trouve les méthodes déterministes, dans lesquelles on utilise la dérivée, comme par exemple les algorithmes de gradient.
Par ailleurs, on trouve les méthodes approchées, comme par exemple les méthodes utilisant des heuristiques ou les méthodes stochastiques.
Les algorithmes évolutionnaires sont des méthodes stochastiques inspirées par la théorie de l'évolution. Dans un algorithme évolutionnaire nous manipulons une population de solutions. Le principe Darwinien est de faire en sorte que les individus les plus adaptés survivent et se reproduisent. La pression de l'environnement est simulée par la fonction objectif. La survie est caractérisée par l'opérateur de sélection (sélection des meilleurs individus au sens de la fonction objectif) et la reproduction est donnée par l'opérateur de mutation. Il s'agit de méthodes stochastiques car au moins un des opérateurs (en général la mutation) utilise des processus aléatoires.
Les méthodes évolutionnaires sont efficaces dans les cas où la fonction objectif est bruitée, non dérivable ou alors multimodale, c'est à dire une fonction ayant plusieurs optima locaux.

Dans ce stage, nous nous intéressons à l'utilisation d'algorithmes évolutionnaires parallèles pour optimiser des fonctions objectifs coûteuses. Pour cela, nous pouvons nous demander quel est le speed-up de l'algorithme $(1, 1 * \tau) - ES$ pour différentes dimensions.
Afin de répondre à cette question nous commençons par un rapide état de l'art sur les algorithmes évolutionnaires parallèles, puis nous décrivons les différents algorithmes séquentiels classiques utilisés dans ce stage. Nous voyons, ensuite, un nouvel algorithme et nous le comparons à l'algorithme classique. Enfin nous simulons une parallélisation asynchrone sur ce nouvel algorithme.

CHAPITRE 2

État de l'art de l'optimisation évolutionnaire

2.1 Les différentes approches

Les algorithmes génétiques permettent de trouver une solution acceptable dans une durée de temps raisonnable en particulier pour des problèmes avec beaucoup de paramètres. Seulement, les problèmes deviennent de plus en plus complexes, et les durées de plus en plus courtes, il faut donc rendre ces algorithmes plus rapides. Une des solutions consistent à utiliser une implémentation parallèle.
L'ensemble des algorithmes évolutionnaires parallèles peut être divisé en plusieurs parties. Nous avons choisi le classement établi par [4]. Nous nous appuierons également sur les algorithmes donnés par Tomassini [10] et [11].

2.2 Maître-esclave

Les algorithmes synchrones

Il existe plusieurs approches d'algorithmes parallèles. En général, l'opération la plus coûteuse dans un algorithme génétique est l'étape d'évaluation d'un individu. Dans cette optique, une première approche serait donc de faire calculer la fitness de chaque individu simultanément, en les envoyant à chaque processeur. Il s'agit dans ce cas de l'approche synchrone. Il y a deux problèmes à cette méthode. Le premier est qu'il faut que le temps d'évaluation de chaque individu soit à peu près le même. Le second est qu'il faut que le nombre d'individus soit un multiple du nombre de processeurs. Un certain nombre de processeurs sera en attente si l'une de ces deux conditions n'est pas remplies. Dans le cas d'une parallélisation synchrone idéale il faudrait avoir un processeur par individu, c'est à dire qu'il faudrait autant de processeurs que d'individus présents dans la population.

Les algorithmes asynchrones

L'approche asynchrone est également une approche de type maître-esclave, avec une population globale. Elle diffère cependant de la méthode synchrone dans la distribution des individus. En effet, dans ce cas, dès qu'un processeur est libre, un individu y est envoyé pour évaluation. On peut donc remarquer qu'avec cette méthode, le temps d'attente des processeurs est minimal, et les deux conditions précédentes sont résolues; cependant le séparation entre les différentes générations est moins nette.

Il est inutile de chercher à paralléliser les opérations de mutation et de croisement car dans la plupart des cas, ce sont des opérations peu coûteuses. Dans le cas où l'ensemble des opérateurs serait parallélisé, le nombre de communications engendrées serait beaucoup plus important.

Pour éviter d'avoir des coûts de communications trop importants, certains travaux, comme par exemple [1] ont utiliser des systèmes avec une mémoire partagée. Dans ce cas, la mémoire du maître est partagée avec tous les esclaves.

[5] a montré que les méthodes basées sur des procédés de passage de messages ont de meilleurs performances que les systèmes à mémoire partagée. Travaux confirmé par [6], qui compare

loa 1: Master-slave algorithm

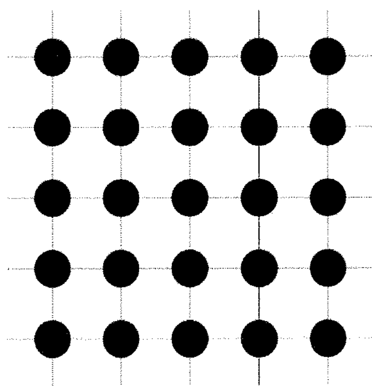
1: Produce an initial population of individuals.
2: **do in parallel**
3: evaluate the fitness of all individuals
4: **end parallel do**
5: **tantque** termination condition not met **faire**
6: select fitter individuals for reproduction
7: produce new individuals
8: mutate some individuals
9: **do in parallel**
10: evaluate the fitness of all individuals
11: **end parallel do**
12: generate a new population by inserting some new good
13: individuals and by discarding some old bad individuals
14: **fin tantque**

également les performances entre les deux procédés, en veillant à utiliser les mêmes paramètres pour les deux procédés.

Les approches maîtres-esclaves ne sont donc pas les plus difficiles à implémenter, et sont, en général, efficaces si la fonction d'évaluation est chère. Un aspect apprécié est, notamment pour le cas synchrone, que le comportement soit proche des algorithmes génétiques séquentiels. La théorie sur ces derniers peut donc en général s'appliquer.

2.3 Approches Fine-grained

Une troisième méthode utilisant une population globale est l'approche *fine-grained.* Cette approche est appropriée pour les architectures massivement parallèles. La population est distribuée spatialement, et les individus n'effectuent les opérations de mutation et de croisement seulement dans leur voisinage. L'idéal est d'avoir un individu par processeur.



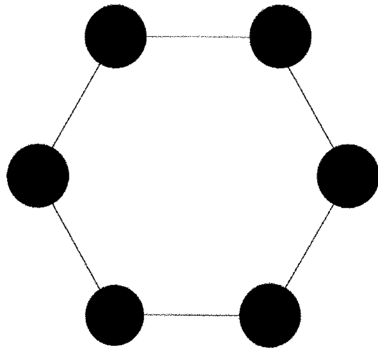
Ce modèle peut être représenté par une grille, dans laquelle chaque noeud représente un individu. La communication entre les individus se fait spatialement.

loa 2: Fine-grained algorithm

1: **pour** each grid point **in parallel faire**
2: generate a random individual
3: **fin pour**
4: **tantque** termination condition not met **faire**
5: **pour** each grid point k **in parallel faire**
6: evaluate individual in k
7: select a neighboring individual q
8: produce offspring from k and q
9: assign one of the offspring to k
10: mutate k with probability p_m
11: **fin pour**
12: **fin tantque**

2.4 Algorithmes à populations multiples

Cette méthode diffère par rapport aux précédents par le fait que nous n'avons plus une population globale mais un ensemble de sous populations. Les approches avec des populations multiples sont assez populaires, elles sont souvent comparées à la co-évolution en biologie. Chaque population évolue indépendamment, mais afin d'éviter une convergence trop rapide, il est nécessaire de faire des migrations entre les dèmes. Ces migrations sont contrôlées par le nombre d'individus choisi pour chaque migration mais également par la fréquence de ces migrations.
On peut parfois trouver ces approches sous le nom d'*algorithme génétique distribué*, *modèle en Ile* ou encore *coarse grained*



Un modèle où chaque noeud représente une sous-population. Les arêtes représentent les communications entre ces sous-populations

2.5 Les algorithmes hybrides

Une dernière méthode permettant la parallélisation d'algorithme est d'effectuer une hybridation entre les différentes approches précédentes. On combine, en général, au niveau supérieur un algorithme multi-dème et au niveau inférieur un algorithme avec une population globale.

Ces types de systèmes sont également appelés *algorithmes hybrides*.

loa 3: Multi-demes algorithm

1: initialize P subpopulations of size N each
2: generation number ← 1
3: **tantque** termination condition not met **faire**
4: **pour** each subpopulation **in parallel faire**
5: evaluate and select individuals by fitness {frequency est le nombre de génération avant un échange}
6: **si** generation number mod frequency = 0 **alors**
7: send K<N best individuals to a neighboring subpopulation
8: replace K individuals from a neighboring population
9: replace K individuals in the subpopulation
10: **finsi**
11: produce new individuals
12: mutate individuals
13: **fin pour**
14: generation number ← generation number + 1
15: **fin tantque**

[4] nous donne trois exemples, représentés par les 3 figures suivantes.

Tout d'abord 2.1. nous avons un algorithme à population multiple à chaque niveau.

Dans le dernier exemple, 2.3, on a un algorithme à population multiple au niveau supérieur et un algorithme fine-grained à l'étage inférieur.

Pour l'exemple 2.2 nous avons encore un algorithme à population multiple au niveau supérieur mais chaque noeud est un algorithme maître-escalve.

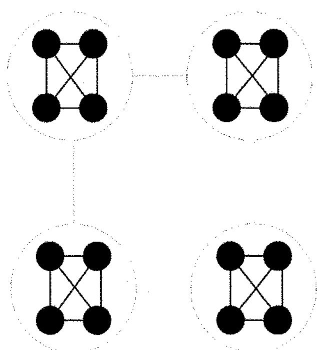


Figure 2.1: Un multi-demes GA au deux niveaux

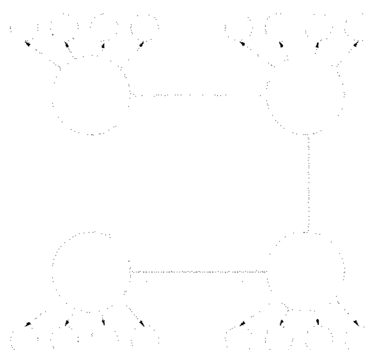


Figure 2.2: Un multi-demes GA où chaque noeud est un GA maître/esclave

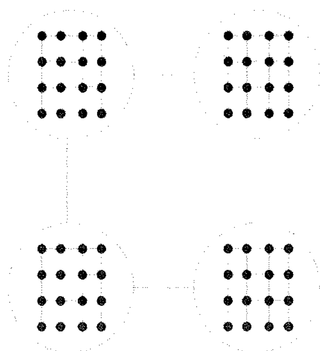


Figure 2.3: Un multi-demes GA et un fine-grained GA

CHAPITRE 3

Étude d'algorithmes séquentiels sur la fonction sphère

Dans cette section nous allons analyser les principaux algorithmes séquentiels sur une certaine fonction objectif. Nous avons choisi d'utiliser comme fonction objectif la fonction sphère, définie comme ceci $f := (x_1, ..., x_d) \in \mathbb{R}^d \to ||x||^2 = \sum_{i=1}^{d} x_i^2$, d étant la dimension.
Une stratégie d'évolution est un algorithme évolutionnaire dans le cas continu, les individus sont représentés par des vecteurs réels. Pour ce projet, nous avons utilisé un algorithme de type (1,λ)-ES. Dans cette stratégie d'évolution, on a une solution potentielle X_n, appelée parent. À partir de ce parent, et pour chaque génération n, sont générées stochastiquement λ nouvelles solutions, appelées enfants. Pour $i = 1 \ldots \lambda$

$$X_n^i = X_n + \sigma_n N^i(0, Id)$$

avec $\sigma_n \in \mathbb{R}^+$ correspondant au pas de la mutation, et $N^i(0, Id)_{1 \leq i \leq \lambda}$ étant λ tirages indépendants de variable aléatoire gaussienne de moyenne 0, et de matrice de covariance identité.

Le meilleur enfant est sélectionné pour devenir le prochain parent X_{n+1}. Dans le cas de la minimisation de la fonction sphère, cette sélection peut être définie comme ceci

$$X_{n+1} = \arg\min\{\|X_n^1\|^2, \ldots, \|X_n^\lambda\|^2\}$$

Un point important est la mise à jour du pas σ_n. Elle est différente d'un algorithme à l'autre, et c'est donc à partir de cette spécification que nous allons séparer les algorithmes que nous présentons. Nous allons décrire les 4 principaux algorithmes utilisés. Le premier utilise le fait que nous connaissons la distance à l'optimum sur la fonction sphère. Les trois autres méthodes sont dites adaptatives, c'est à dire que le réglage du pas va s'obtenir de manière automatique. Il est important de comprendre pourquoi le pas doit être adaptatif. La Figure 3.1 montre l'évolution de la fonction objectif que l'on a pendant le déroulement d'un algorithme évolutionnaire, c'est à dire en fonction des itérations, pour une dimension 3 et une taille de population de 5.

Intéressons nous tout d'abord au pas constant (ligne pleine), i.e. $\sigma_n = \sigma$, nous avons choisi $\sigma = 0.1$. Jusqu'à l'itération 1200 environ, nous ne pouvons constater qu'une amélioration minime. En effet, dans ce cas le pas est trop petit par rapport à la distance à l'optimum, c'est à dire que la perturbation des parents est très légère et, par conséquent, on ne se déplace jusqu'à l'optimum que lentement.
À partir de l'itération 1300, la distance à l'optimum devient trop petite par rapport au pas. Lorsque le pas est trop grand, la probabilité de succès, i.e., d'avoir un enfant meilleur que le parent, devient très faible, ce qui explique pourquoi dans cette troisième phase la fonction fitness n'est pas seulement décroissante, mais qu'une oscillation apparaît.
La courbe en pointillés, représente une méthode ou le pas est mis à jour à chaque itération. Une première chose à constater est que la fonction fitness est logarithmiquement linéaire. On appelle taux de convergence le coefficient directeur de cette droite, on le note $c(\lambda, d)$. Formellement, le taux de convergence est donc égale à

$$c(\lambda, d) = \lim_{n \to \infty} \frac{1}{n} (\ln \|X_n\| - \ln \|X_0\|) \quad (3.1)$$

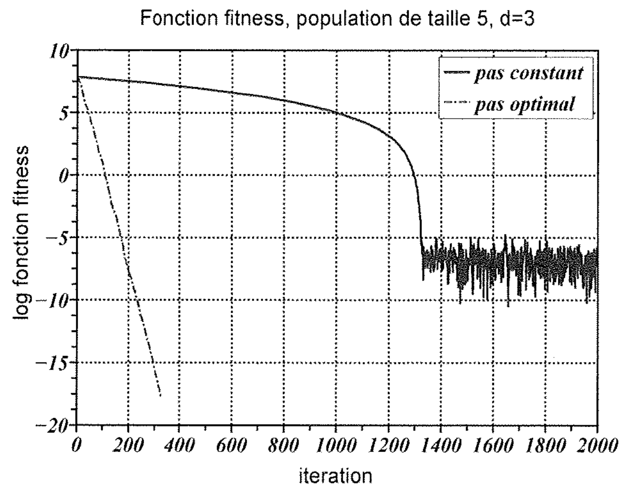


Figure 3.1: La courbe pleine représente la fonction objectif obtenue avec un pas de mutation constant pour l'exécution globale de l'algorithme. La courbe en pointillé correspond à la fonction fitness en fonction du nombre de génération si le pas se met à jour à chaque itération. La taille de la population est de 5 individus.

3.1 Pas optimal (σ^*)

Une première méthode est d'utiliser le fait que l'on connaît la distance à l'optimum pour la fonction sphère. Cet algorithme nous donne le taux de convergence optimal pour les stratégies d'évolution [2].

On va chercher le σ^* tel que $\sigma_n = \sigma^* \|X_n\|$ ait le meilleur taux de convergence. La Figure 3.2 nous permet de voir comment cette valeur est calculée. On discrétise σ^* et on calcule le taux de convergence pour chacune de ces valeurs. La valeur de σ^*_{opt} correspond au taux de convergence le plus petit, c'est à dire lorsque l'algorithme trouve une solution en un nombre de générations le plus faible.

L'algorithme 4 correspond à cette méthode.

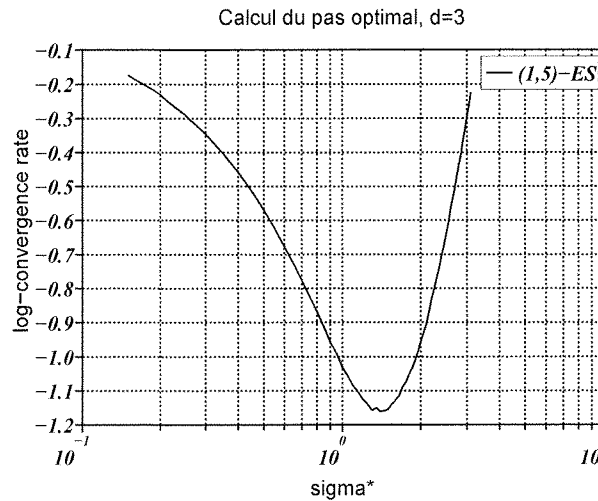


Figure 3.2: Calcul du σ^*_{opt}. Nous avons le taux de convergence en fonction de σ^*. Pour cet exemple, la taille de la population est 5 et la dimension 3. La valeur σ^*_{opt} correspond au taux de convergence le plus petit.

loa 4: sigma*

Entrées: $\sigma \neq 0$
1: P $\leftarrow$ init()
2: **tantque** termination condition not met **faire**
3: **pour** $i \leftarrow 1$ to λ **faire**
4: $X' \leftarrow X + \sigma^* N(0, Id)$
5: $X.caluleFitness()$
6: P'.ajoute(X);
7: **fin pour**
8: $X \leftarrow$ min(P')
9: $\sigma \leftarrow \sigma^* \|X\|$
10: P $\leftarrow$ X
11: **fin tantque**

3.2 Self-Adaptation

Introduit par Rechenberg [8] et Schwefel [9], l'idée est de faire également muter les paramètres de mutation, en partant du principe qu'un mauvais paramètre de mutation ne donnera pas de bons enfants. Le paramètre de mutation pour la génération suivante est celui qui a donné le meilleur enfant.
La mutation de σ_n est effectué à la ligne 4 de l'algorithme 2. La notation $X.\sigma$ est utilisée pour signaler que nous avons un pas spécifique pour chaque individu.

loa 5: SA

1: P $\leftarrow$ init()
2: **tantque** termination condition not met **faire**
3: **pour** $i \leftarrow 1$ to λ **faire**
4: $X.\sigma' \leftarrow \sigma \exp(\frac{N(0,1)}{\sqrt{d}})$
5: X $\leftarrow X + X.\sigma' N(0, Id))$
6: X.calculeFitness()
7: P'.add(X);
8: **fin pour**
9: X $\leftarrow \arg\min(P')$
10: $\sigma \leftarrow X.\sigma'$
11: P $\leftarrow$ X
12: **fin tantque**

3.3 Cumulative Step-size Adaptation

Créé par Hansen et Ostermeier en 2001 [7], cette méthode consiste à regarder le chemin suivi par l'algorithme, et le comparer au chemin que l'on aurait si la fonction fitness était aléatoire.
Si le chemin suivi par l'algorithme est supérieur à la longueur du chemin dans le cas aléatoire, alors on va augmenter le pas, dans le cas contraire nous allons le diminuer.

La formule de calcul du chemin est

$$p_\sigma^{n+1} = (1 - c_\sigma)p_\sigma^n + \sqrt{(c_\sigma(2 - c_\sigma)}\frac{(X_{n+1} - X_n)}{\sigma_n}$$

En choisissant $c_\sigma \in \ ]0,1[$, le facteur $(1 - c_\sigma)$ permet de donner moins d'importance aux pas effectués aux itérations précédentes.

Justification du facteur $\sqrt{(c_\sigma(2 - c_\sigma)}$.

On a $X_{n+1} = X_n + \sigma_n N(0, Id)$.

Si la fonction fitness est aléatoire, la sélection n'a pas d'effet donc $\frac{X_{n+1} - X_n}{\sigma_n} \sim N(0, Id)$.

Si $p_\sigma^n \sim N(0, Id)$ alors

$$p_\sigma^{n+1} = (1 - c_\sigma) \underbrace{p_\sigma^n}_{\text{N(0,Id)}} + \sqrt{(c_\sigma(2 - c_\sigma)} \underbrace{\frac{(X_{n+1} - X_n)}{\sigma_n}}_{\text{N(0,Id)}}$$

Soient $\alpha = (1 - c_\sigma)$ et $\beta = \sqrt{(c_\sigma(2 - c_\sigma)}$

Pour chaque composante, on a $\alpha N + \beta N = N(0, \alpha^2) + N(0, \beta^2)$.

Or, la somme de deux variables gaussiennes indépendantes est aussi une variable gaussienne. Soient X_1 et X_2, deux variables aléatoires indépendantes suivant respectivement $N(0, \alpha^2)$ et $N(0, \beta^2)$ alors $X_1 + X_2$ suit $N(0, \alpha^2 + \beta^2)$.

Or, on peut remarquer que :

$$\alpha^2 + \beta^2 = (1 - c_\sigma)^2 + c_\sigma(2 - c_\sigma) = 1 - 2c_\sigma + c_\sigma^2 + 2c_\sigma - c_\sigma^2 = 1$$

Par conséquent, pour chaque composante on a $p_\sigma^{n+1} \sim N(0, 1)$

Le facteur $\sqrt{(c_\sigma(2 - c_\sigma)}$ permet d'avoir le fait qu'avec une fonction fitness aléatoire $p_\sigma^{n+1} \sim N(0, Id)$ si $p_\sigma^n \sim N(0, Id)$.

La mise à jour du pas est $\sigma_{n+1} = \sigma_n \ \exp((\frac{c_\sigma}{d_\sigma})(\frac{||p^{n+1}||}{E(||N(0,Id)||)} - 1))$

Nous pouvons constater que si $||p_\sigma^{n+1}|| > E(||N(0,Id)||)$ alors $\frac{||p^{n+1}||}{E(||N(0,Id)||)} - 1 > 0$, donc $\exp(((\frac{c_\sigma}{d_\sigma})(\frac{||p^{n+1}||}{E(||N(0,Id)||)} - 1))) > 1$ le pas est ainsi augmenté. A l'inverse, si $||p_\sigma^{n+1}|| < E(||N(0,Id)||)$ alors le pas est diminué.

L'algorithme est présenté en Algorithme 6

loa 6: CSA

1: P ← init()
2: **tantque** termination condition not met **faire**
3: **pour** $i \leftarrow 1$ to λ **faire**
4: X ← $X + \sigma * N(0, Id))$
5: X.calculeFitness()
6: P'.add(X):
7: $p \leftarrow (1 - c_\sigma)p + \sqrt{c_\sigma(2 - c_\sigma)}(\frac{X_{n-1} - X_n}{\sigma_n})$
8: **fin pour**
9: X ← min(P')
10: $\sigma \leftarrow \sigma \exp(\frac{c_\sigma}{d_\sigma}(\frac{||p_\sigma||}{E(||N(0,Id)||)}))$
11: P ← X
12: **fin tantque**

3.4 Règle des $\frac{1}{5}$

La règle des $\frac{1}{5}$ a été introduite par Rechenberg et date de 1973 [8].
L'idée de cette règle est d'augmenter le pas de la mutation si la probabilité de succès est supérieure à $\frac{1}{5}$, le diminuer sinon.
La valeur $\frac{1}{5}$ est une approximation de la probabilité de succès sur la fonction sphère, nous allons essayer de calculer une approximation plus fine.

Calcul de la probabilité de succès

Soit $e_1 = (1, 0 \dots 0)$ (En particulier, on a $\|e_1\| = 1$).
Pour $i = 1 \dots N$, on définit

$$X_i = e_1 + \sigma^*_{opt} N(0, Id)$$

avec σ^*_{opt} le pas de la mutation associé au meilleur taux de convergence, pour le λ correspondant.
Nous avons vu comment le calculer dans la Section 3.1.
Ensuite, on va compter le nombre d'individus générés dont la norme est inférieure à 1 ($\|e_1\| = 1$), et le diviser par N.
Nous obtenons donc, sur une moyenne de N tirages la probabilité qu'un individu soit meilleur que l'individu qui l'a généré, c'est à dire que sa norme soit inférieure. On appelle cette probabilité probabilité de succès.

Algorithme

loa 7: oneFifth

1: P ← init()
2: **tantque** termination condition not met **faire**
3: **pour** $i \leftarrow 1$ to λ **faire**
4: X ← $X + \sigma N(0, Id))$
5: X.calculeFitness()
6: P'.add(X);
7: **fin pour**
8: X ← $\arg\min(P')$
9: $\sigma \leftarrow \sigma \exp(\frac{p - p_{target}}{1 - p_{target}})$
10: *where* $p = \frac{\#successfulOffsprings}{\lambda}$
11: *and* $p_{target} = \frac{1}{5 + \frac{\sqrt{N}}{2}}$
12: P ← X
13: **fin tantque**

3.5 Synthèse

La Figure 3.3 représente une comparaison du taux de convergence pour les algorithmes vus précédemment.
Il s'agit de la mesure du taux de convergence, en fonction de la taille de la population, λ, par rapport au nombre d'évaluations.
Il est important de voir pourquoi nous avons choisi d'effectuer nos mesures en regardant le nombre d'évaluations et non le nombre d'itérations.

En général, l'étape d'évaluation est l'opération la plus coûteuse dans un algorithme évolutionnaire, il est donc intéressant de minimiser le nombre d'évaluations que l'on fait, c'est à dire limiter la taille de la population.

Lors de l'équation 1, nous avons défini le taux de convergence. Nous pouvons de la même façon définir le taux de convergence en tenant compte en plus du nombre d'évaluations que l'on va effectuer lors de l'algorithme.

$$\frac{c(\lambda, d)}{\lambda} = \lim_{n \to \infty} \frac{1}{n * \lambda} (\ln \|X_n\| - \ln \|X_0\|) \tag{3.2}$$

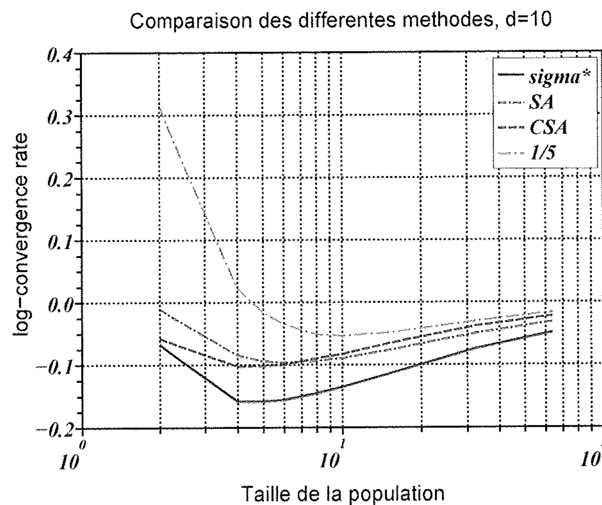


Figure 3.3: Comparaison du taux de convergence pour les différentes méthodes en fonction de λ. En ordonnée, nous avons le taux de convergence, et en abscisse la taille de la population.

CHAPITRE 4

Étude d'algorithmes asynchrones sur la fonction sphère

4.1 Pas optimal (σ^*)

Dans ce modèle nous partons du principe que $\lambda = 1$ mais que chaque individu de la population a un âge. À chaque génération, nous ne créons qu'un seul nouvel individu mais la sélection se fait parmi tous les enfants qui n'ont pas atteint l'âge limite τ.
Nous noterons ce procédé $(1, 1 * \tau)$.

L'algorithme correspondant est l'algorithme 8

loa 8: sigma*, (1,1*τ)

```
1: P ← init()
2: tantque termination condition not met faire
3:   X' ← X + σ*N(0, Id))
4:   X.calculFitness()
5:   P.ajoute(X)
6:   P.incrementeAge
7:   pour tout X in P faire
8:     si X.age = τ alors
9:       P.supprime(X)
10:    finsi
11:  fin pour
12:  X ← arg min(P)
13:  σ ← σ||X||
14:  P ← X
15: fin tantque
```

Il nous faut connaître le σ^*_{opt} pour cette méthode.

La Figure 4.1 correspond aux tracés des différentes valeurs du taux de convergence pour différents σ. Ceci nous permet tout d'abord de connaître le valeur de $\sigma*$ dans le cas de la méthode $(1, 1 * \tau)$, mais on peut également constater qu'à partir d'un λ suffisamment grand, on obtient aucune amélioration du taux de convergence.

La Figure 4.2 nous permet de voir la différence entre les taux de convergence entre les deux procédés pour différentes tailles de population.

Nous pouvons constater qu'à partir d'un τ suffisamment grand, le taux de convergence tend vers un algorithme $(1 + 1)$. Dans cet algorithme, le principe est de ne générer qu'un individu par génération, le remplacement est alors effectué entre le parent et l'enfant généré. Cet algorithme est donc élitiste, c'est à dire que l'on garde toujours la meilleure solution. Il est optimal sur la fonction sphère parmi les algorithmes $(1 + \lambda)$ et $(1, \lambda)$ utilisant des mutations gaussiennes.

L'idée est maintenant de comparer l'efficacité entre un algorithme adaptatif suivant le modèle $(1, 1 * \tau)$ et la méthode utilisant le pas optimal.

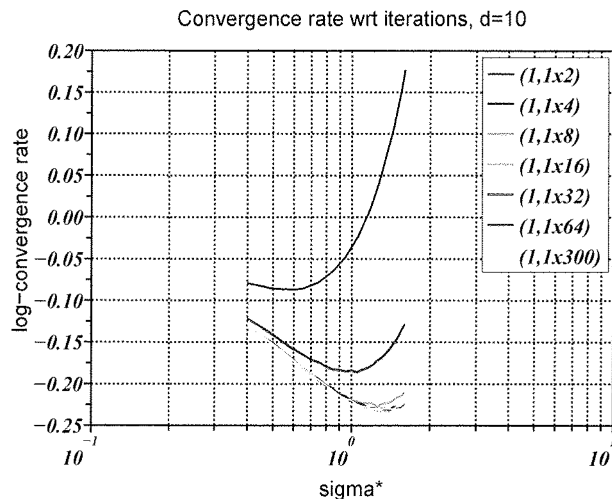


Figure 4.1: Taux de convergence normalisé en fonction de σ^*, pour une dimension égale à 10. On mesure pour différentes tailles de population le taux de convergence afin de déterminer σ^*_{opt}. À partir de $\lambda = 16$ plus aucune amélioration n'est constatée, effet que l'on peut également voir sur la Figure 4.2.

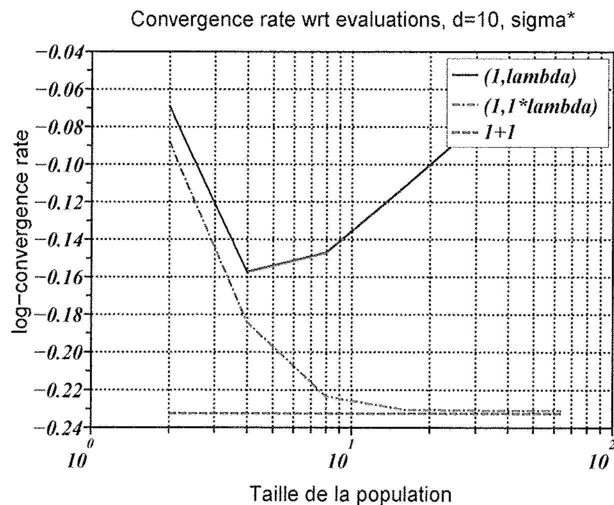


Figure 4.2: Comparaison du $(1, \lambda)$ et $(1, 1 * \tau)$. On regarde le taux de convergence en fonction de la population. On note que le taux de convergence correspondant au $(1, 1 * \lambda)$ tend vers un algorithme $(1 + 1)$ lorsque la taille de la population devient grande.

4.2 Cumulative Step-size Adaptation

Dans la stratégie d'évolution $(1, \lambda)$ la mise à jour du pas est, comme nous l'avons déjà vu, $\sigma_{n+1} = \sigma_n \exp((\frac{c_\sigma}{d_\sigma})(\frac{||p^{n-1}||}{\beta_d} - 1))$, avec $\beta_d = E(||N(0, Id)||)$.

Pour le modèle $(1, 1 * \tau)$, cela ne sera plus le cas. En effet, on cherche à comparer la longueur du vecteur p_σ^n à la longueur qu'aurait le pas si la fonction fitness était aléatoire. Nous devons donc recalculer la longueur de p_σ^n dans le cas d'une fonction fitness aléatoire pour le modèle $(1, 1 * \tau)$.

Nous devons déterminer la valeur de β_d pour laquelle le pas reste constant durant le déroulement d'un algorithme avec une fonction fitness aléatoire. Si σ_{iter} correspond à la valeur du pas lors de la dernière itération, et σ_{init} correspond à la valeur du pas initial alors on va prendre la valeur de β_d tel que $\frac{(\ln(\sigma_{iter})-\ln(\sigma_{init}))}{iter} = 0$.

4.3 Règle des $\frac{1}{5}$

Cette règle s'applique particulièrement bien au $(1, 1 * \tau)$.
En effet, le seul paramètre à fixer dans ce cas est la probabilité de succès, or λ est constant donc la probabilité de succès l'est également. Il est donc facile d'adapter un algorithme $(1, \lambda)$ avec la règle du $\frac{1}{5}$ en un $(1, 1 * \tau)$ avec la même règle.

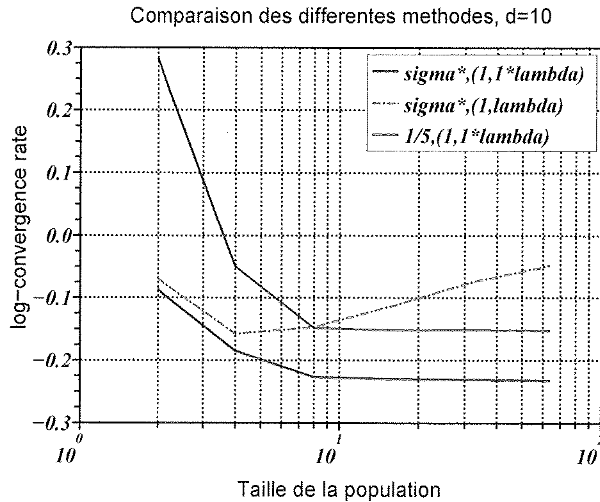


Figure 4.3: Taux de convergence en fonction de λ. On mesure le taux de convergence en fonction de la taille de la population.

Sur la Figure 4.3 nous comparons le modèle $(1, 1 * \tau)$ avec du $(1, \lambda)$. Nous pouvons constater qu'en utilisant la règle des $\frac{1}{5}$ et avec un τ suffisamment grand on obtient un taux de convergence optimal presque équivalent à une stratégie d'évolution $(1, \lambda)$ utilisant le pas optimal. La probabilité de succès choisie est celle obtenue après calculs, il s'agit de la même probabilité que dans le cas $(1 + 1) - ES$, à savoir 0.27 pour une dimension égale à 10.

4.4 Délai

Nous simulons dans cette partie une parallélisation asynchrone du modèle $(1, 1 * \tau)$. Dans ce modèle, le choix du parent se fait normalement parmi les τ enfants disponibles. Pour simuler un délai, nous allons faire la sélection parmi les $\tau - P$ enfants. La valeur P nous permet de spécifier le nombre d'enfants étant en attente d'évaluation dans le cas d'une parallélisation asynchrone. La Figure 4.4 explique ce principe.

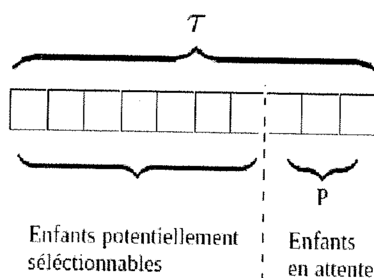


Figure 4.4: Fonctionnement du délai

Dans un premier cas de figure (Figure 4.5) nous avons choisi une valeur de τ assez importante ($\tau = 100$). Nous mesurons ensuite le taux de convergence en fonction du nombre de processeurs P. Pour cet exemple, l'algorithme choisi est scale-invariant. Le taux de convergence est mesuré par rapport au nombre d'itérations. Pour simuler une parallélisation asynchrone, nous multiplions le taux de convergence par le nombre de processeur. Dans ce cas, nous ne tenons pas compte, par contre, du coût des communications résultant. Nous pouvons constater une amélioration lorsque le nombre de processeurs augmente. Cette amélioration est logarithmique jusqu'à 70 processeurs environ.

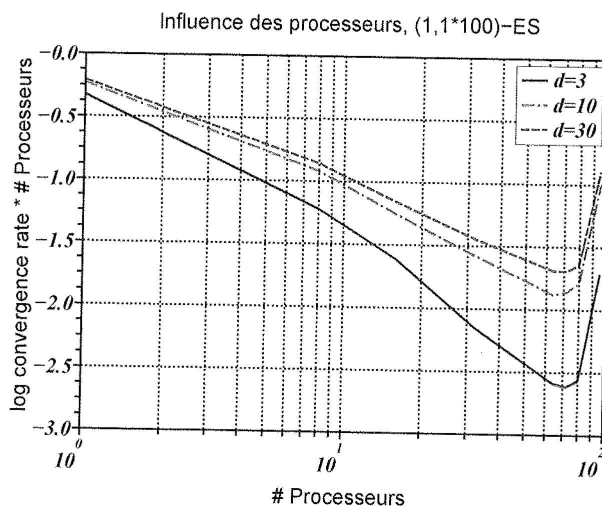


Figure 4.5: On mesure le taux de convergence en fonction du nombre de processeurs, pour différentes dimensions. On peut noter une amélioration sous linéaire en fonction du nombre de processeurs.

Nous avons vu que τ représentait le nombre d'enfants sélectionnables plus le nombre d'enfants en attente. Sur la Figure 4.6, nous fixons le nombre d'enfants sélectionnables et nous faisons varier le nombre d'enfants en attente. Nous avons donc une stratégie $(1, 1 * (32 + P))$, et nous faisons varier P pour mesurer l'influence du délai, là encore en utilisant l'algorithme scale-invariant.

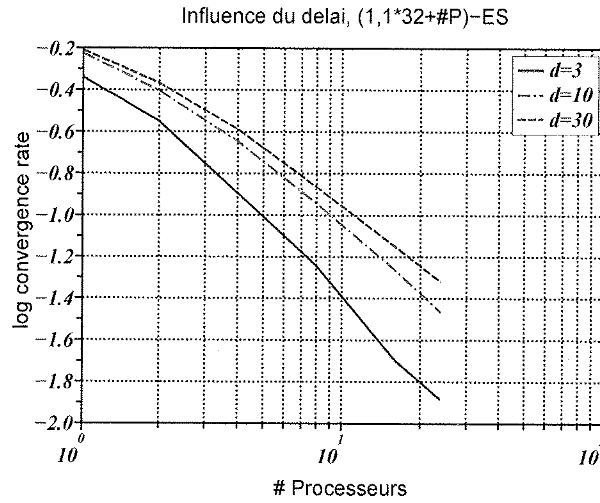


Figure 4.6: On mesure le taux de convergence en fonction du nombre de processeurs, pour différentes dimensions.

En suivant toujours la même méthodologie nous effectuons une simulation de notre modèle en utilisant une règle réaliste pour la mise à jour du pas, la règle des $1/5$, Figure 4.7.
La taille de la population sélectionnable est de 200 individus, et nous simulons jusqu'à 160 processeurs. Nous notons que nous avons là encore un speed-up logarithmique jusqu'à 128 processeurs pour le modèle $(1, 1 * \tau)$. Pour la stratégie $(1, \lambda)$ nous avons un speed-up logarithmique pour une dimension petite seulement.

Pour mesurer le gain pour le modèle $(1, \lambda)$ il suffit de faire en sorte que le nombre de processeurs soit égal à la taille de la population. Pour notre simulation, cela revient donc à calculer le taux de convergence par rapport au nombre d'itérations.

Lorsque λ devient grand l'approximation de la probabilité de succès devient plus importante, ce qui explique que les courbes de la statégie $(1, \lambda)$ ne soient pas monotones.

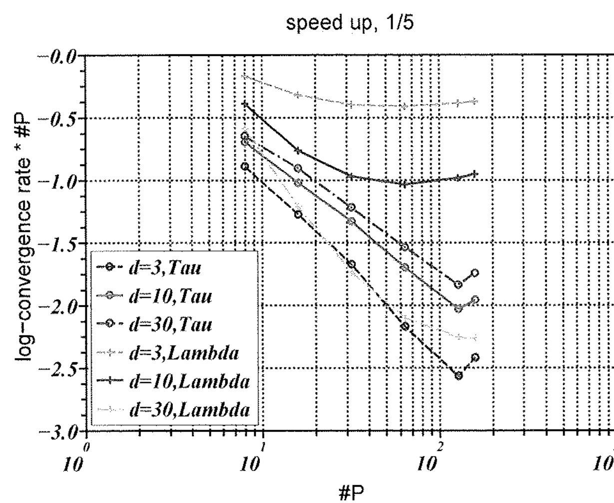


Figure 4.7: On mesure le taux de convergence en fonction du nombre de processeurs, pour les dimensions 3, 10 et 30. Nous pouvons constater que le speed-up est logarithmiquement linéaire jusqu'à 128 processeurs pour la stratégie $(1, 1 * \tau)$.

CHAPITRE 5

Conclusion et perspectives

Nous avons décrit jusqu'ici les travaux menés dans le cadre de l'optimisation évolutionnaire continue. Nous avons présenté la stratégie d'évolution classique $(1, \lambda)$, en utilisant et comparant plusieurs règles de mise à jour. Nous avons également effectué des travaux sur l'amélioration de certaines constantes se trouvant dans les règles de mise à jour, travaux que nous n'avons pas pu vous décrire dans ce rapport.

La seconde partie du travail a été d'utiliser un nouveau modèle $(1, 1 * \tau)$, et de l'utiliser avec les mêmes règles que précédemment. La difficulté a été de revoir la presque totalité des paramètres utilisés dans ces règles, afin de les adapter à ce modèle. Nous avons enfin commencé à simuler une parallélisation asynchrone sur le modèle $(1, 1 * \tau)$. Nous obtenons pour ce modèle un speed-up logarithmique. Pour le modèle $(1, \lambda)$, Beyer annonce un speed-up logarithmique également [3].

La première perspective est évidemment d'effectuer une parallélisation réelle en gardant les mêmes paramètres que lors des simulations. Il est intéressant de voir également le modèle $(1, 1 * \tau)$ sur une fonction plus complexe que la fonction sphère, par exemple une fonction multimodale ou bruitée.

Il serait intéressant de poursuivre ces mêmes travaux sur l'algorithme CMA-ES (*Covariance Matrix Adaptation Evolution Strategy*) qui devient l'un des algorithmes les plus importants dans le domaine.

Remerciements

Je tiens à remercier particulièrement Anne Auger pour son encadrement, son aide et sa disponibilité. Je tiens également à remercier Nikolaus Hansen pour ses précieux conseils.

Je voudrais remercier Michèle Sebag et Marc Schoenauer pour m'avoir accueilli dans l'équipe.

Un grand merci à l'équipe Tao, ainsi que l'ensemble des doctorants et des stagiaires pour leur accueil et leur amabilité.

Références

[1] D. Abramson and J. Abela. A parallel genetic algorithm for solving the school timetabling problem. Technical report, 1991.

[2] Anne Auger and Nikolaus Hansen. Reconsidering the progress rate theory for evolution strategies in finite dimensions. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 445–452, New York, NY, USA, 2006. ACM.

[3] Hans-Georg Beyer. *The Theory of Evolution Strategies.* Natural Computing Series. Springer, Heidelberg, 2001.

[4] E. Cantú-Paz. A survey of parallel genetic algorithms, 1997.

[5] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, San Jose, California, 1994.

[6] G. Digalakis and K.G. Margaritis. Parallel evolutionary algorithms on message-passing clusters.

[7] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[8] I. Rechenberg. Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. *Frommann-Holzboog*, 1973.

[9] H.-P. Schwefel. Adaptive mechanismen in der biologischen evolution und ihr einflub auf die evolutionsgeschwindigkeit. technical report, technical univer- sity of berlin. abschlubbericht zum dfg-vorhaben re 215/2. 1974.

[10] M. Tomassini. Parallel and distributed evolutionary algorithms, 1999.

[11] Marco Tomassini. Evolutionary algorithms. In Eduardo Sanchez and Marco Tomassini, editors, *Towards Evolvable Hardware; The Evolutionary Engineering Approach*, pages 19–47, Berlin, 1996. Springer.