Parallélisation de EO

Caner Candan
caner.candan@thalesgroup.com

27 janvier 2011

Table des matières

1 Introduction

Nous utilisons le framework EO [1] pour développer nos algorithmes évolutionnaires [2]. Le sujet consiste à implémenter, au framework EO, un parallèlisme à mémoire partagée en utilisant OpenMP.

2 Préambule

Avant de commencer il est important de préciser que les EA [3] travaillent sur une population d'individus aussi appelé échantillon. Un individu étant représenté par un point dans l'échantillon, la complexité d'un problème est définit par le nombre de dimensions pour chaque individu.

Le problème est représenté par la fonction objectif qui prend en paramètre un individu (un point dans l'échantillon) et évalue toutes ses dimensions pour en déduire la qualité [4]. La qualité est le critère de comparaison d'un point dans son échantillon.

3 Identifier les ressources les plus utilisées

Un test de profiling a été executé afin d'identifier les ressources les plus utilisées dans le framework EO. Le test nous a permi d'identifier une fonction qui est utilisée par une grande majorité des opérateurs [5] EO. Il s'agit de la fonction "apply". Elle prend en paramètres une population d'individus et un opérateur. Cette fonction va itérer sur tous les individus de la population et appliquer l'opérateur. Il peut être intéressant d'optimiser cette fonction. Nous allons nous limiter à transformer le code sequentiel en parallèle.

4 Pseudo-code de la fonction apply

L'algorithme 1 prend en paramètre une population ainsi qu'un opérateur à appliquer à chaque individu.

```
   Données : P ∈ K_n, F ∈ Operateur
   Résultat : B' ∈ K_n
1  début
2  |  pour i ← 0 à n faire
3  |  |  F(P(i))
4  fin
```

Algorithme 1 : La fonction apply

5 La fonction en parallèle

L'algorithme 2 transforme la fonction "apply" en parallèle en utilisant le modèle PRAM CREW [6] et $O(n)$ processeurs pour parcourir tous les individus de la population.

1. Evolving Object, `http://eodev.sf.net`
2. Algorithme évolutionnaire : `http://fr.wikipedia.org/wiki/Algorithme_evolutionnaire`
3. Algorithmes évolutionnaires
4. Fitness en anglais
5. Opérateurs de sélection et variation
6. Concurantial Read Exclusive Write

Données : $P \in K_n, F \in Operateur$
Résultat : $B' \in K_n$
1 **début**
2 | *parallèle*
3 | **pour** $i \leftarrow 0$ **à** n **faire**
4 | | $F(P(i))$
5 **fin**

Algorithme 2 : La fonction omp_apply

6 Speed-up

Après avoir crée la fonction alternative employant le parallèlisme à mémoire partagée, appelé "omp_apply", nous allons étudier une solution de mesure du speed-up [7].

L'équation, en figure 1, présente une méthode de mesure du speed-up et est implémentée dans l'algorithme 3.

$$Mesure\ du\ Speedup = r \sum_{k=0,l=0}^{P,D} S_{p_{kl}}$$

FIGURE 1 – Mesure du Speedup

Une description des paramètres est disponible dans la figure 2.

Paramètres	**Description**
p	la taille minimum de la population
$popStep$	le pas d'iteration de la population
P	la taille maximum de la population
d	la taille minimum de la dimension
$dimStep$	le pas d'iteration de la dimension
D	la taille maximum de la dimension
r	le nombre d'exécution pour chaque combinaison de p et d

FIGURE 2 – Description des paramètres utilisés

7 Mesures

En prenant en compte les paramètres décrits précédement, nous allons lancer les tests sur deux architectures matérielles différentes présenté en figure 3.

Pour visualiser l'évolution du speed-up, nous utilisons un outil de génération de graphiques [9], avec les données produits par les tests.

7. $S_p = \frac{T_1^*}{T_p}$: `http://en.wikipedia.org/wiki/Speedup`
9. Utilisation de matplotlib en python pour générer des boites à moustache

```
   Données : p, P, popStep, d, D, dimStep, r ∈ N
 1 début
 2     pour k ← p à P faire
 3         pour l ← d à D faire
 4             pour m ← 0 à r faire
 5                 T_s ← 0
 6                 T_p ← 0
 7                 début
 8                     t_1 ← omp_get_wtime()
 9                     ... code sequentiel avec k et l exécuté m fois ...
10                     apply( ... )
11                     t_2 ← omp_get_wtime()
12                     T_s ← t_2 − t_1
13                 fin
14                 début
15                     t_1 ← omp_get_wtime()
16                     ... code parallèle avec k et l exécuté m fois ...
17                     omp_apply( ... )
18                     t_2 ← omp_get_wtime()
19                     T_p ← t_2 − t_1
20                 fin
21                 ... on conserve le speed-up T_s / T_p pour k et l ...
22 fin
```

Algorithme 3 : La fonction de mesure du speedup

Processeur	**Nombre de coeurs**	**Fréquence**	**Cache L1**
Intel Centrino vPro	2	2.40GHz	3072KB
Intel Core i7	8 (hyperthreading [8])	2.67GHz	8192KB

FIGURE 3 – Architectures matérielles

7.1 Fonctions objectifs

Il est important de simuler toutes les compléxités de problèmes que l'on peut être ammené à résoudre. Pour nos tests, le choix a été orienté vers deux fonctions objectifs de complexité différente présenté en figure 4.

Fonction objectifs	**Complexité en temps**	**Algorithme**
L'algorithme du Sphere	$O(1)$	$Sphere = \sum_{k=0}^{n} individu_k$
Problème à temps variable quelconque	$O(n)$	usleep$(U(0,1) * 10)$

FIGURE 4 – Fonctions objectifs

7.2 Benchmark

Pour faciliter et automatiser les tests, une liste de mesures a été élaboré avec les paramètres décrits précédement et un script contenant l'ensemble des tests à executer a été crée. La liste des mesures est présentée en figure 5.

Mesure	Description
1	mesure pour toutes les combinaisons de P et D
2	mesure pour $P \in [1, 101[$ avec $D = 1000$
3	mesure pour $P \in [1, 1001[$ avec $popStep = 10$ et $D = 1000$
4	mesure pour $D \in [1, 101[$ avec $P = 1000$
5	mesure pour $D \in [1, 1001[$ avec $dimStep = 10$ et $P = 1000$

FIGURE 5 – Liste des mesures

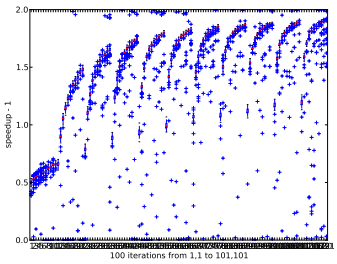
7.3 Dynamicité

Parmi les optimisations possibles en OpenMP, il existe deux types de planification, la planification statique, utilisé par défaut, divise le nombre de tâches à traiter à tous les processus disponibles et la planification dynamique maintient une file de tâches traitée au fur et à mesure par les processus disponibles. Nous évaluons la dynamicité par le rapport entre une mesure de speedup en mode statique et une mesure de speedup en mode dynamique [10].

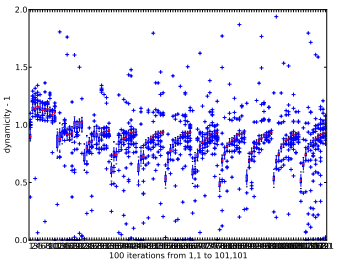
7.4 Résultats en $O(1)$

Le benchmark a été executé dans un premier temps pour le problème du Sphere qui se résoud en temps constant. Les résultats de chaque mesure sont numérotés d'après le tableau des mesures décrit précédement.

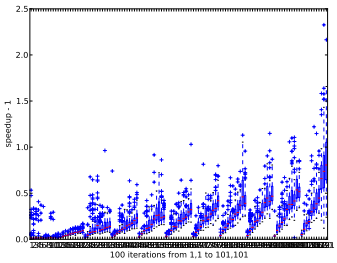
Les mesures sont présentées en fonction des processeurs utilisés et disponibles en figure 6, 7, 8, 9 et 10.



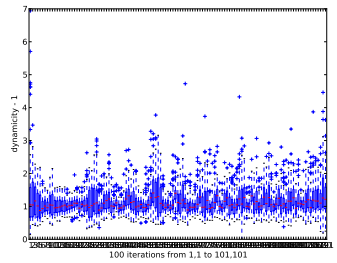
(a) speedup sur 2 coeurs



(b) dynamicité sur 2 coeurs



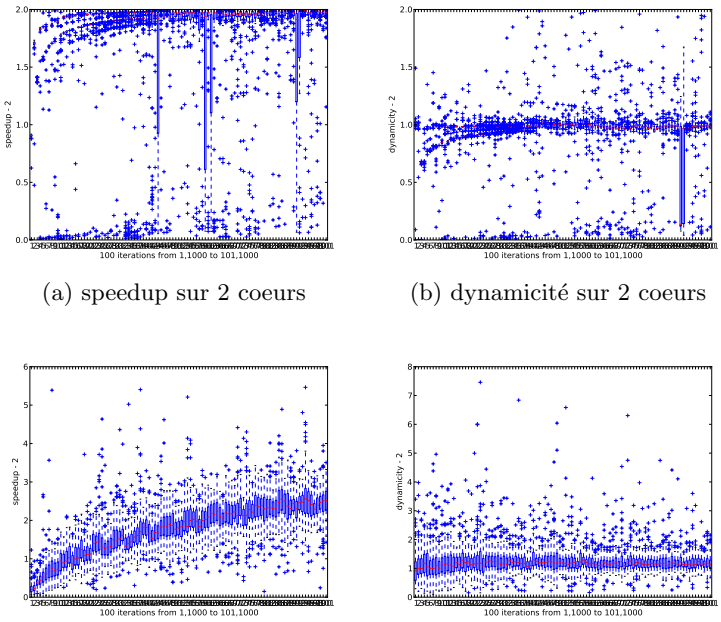
(c) speedup sur 8 coeurs



(d) dynamicité sur 8 coeurs

FIGURE 6 – Mesure 1 en $O(1)$ sur 2 et 8 coeurs

10. Dynamicité : $D_p = \frac{S_p}{S_p^d}$



(a) speedup sur 2 coeurs (b) dynamicité sur 2 coeurs

(c) speedup sur 8 coeurs (d) dynamicité sur 8 coeurs

FIGURE 7 – Mesure 2 en $O(1)$ sur 2 et 8 coeurs

7.5 Résultats en $O(n)$

Dans un second temps, le benchmark est executé pour le problème qui se résoud en temps variable. Les résultats de chaque mesure sont numérotés d'après le tableau des mesures décrit précédement en figure 5.

Il est important d'ajouter que compte tenu de la résolution du problème en $O(n)$ et non plus en fonction de la dimension, celle ci perd de son importance. Ainsi les mesures 1, 4 et 5 ne sont plus nessaire à mesurer. Seules les mesures 2 et 3 seront présentées ci-dessous.

Un algorithme séquentiel executé sur un processeur à 8 coeurs en comparaison avec un processeur à 2 coeurs est plus performant sur des petits échantillons. Nous avons donc choisit de définir les bornes des paramètres, décrits en figure 5, comme multiple du nombre de coeurs disponible.

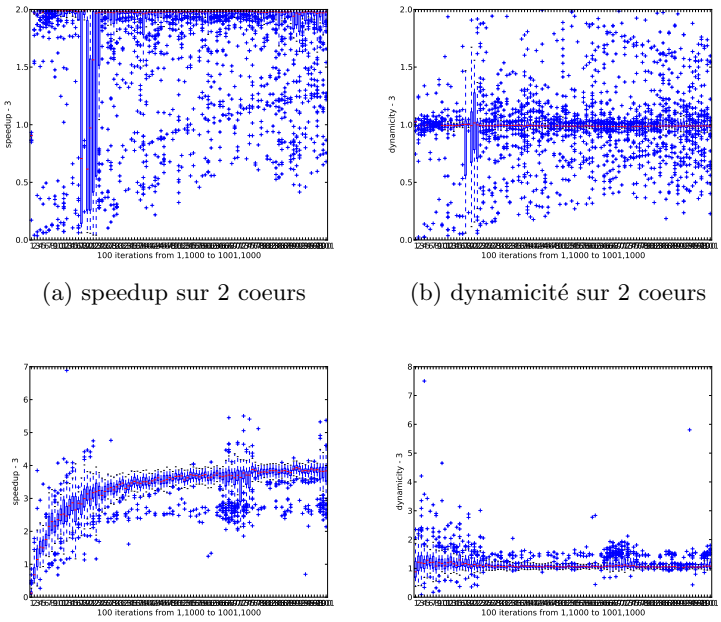
Les mesures sont présentées en fonction des processeurs utilisés et disponibles en figure 11 et 12.

7.6 Conclusion

Pour un processeur à 2 coeurs, selon la complexité du problème utilisée, nous pouvons observer par la mesure du speedup que les ressources sont utilisées au maximum [11] pour de grandes tailles d'échantillon. Les petites tailles d'échantillon restant dominées par une execution sequentielle.

Pour un processeur à 8 coeurs, en hyperthreading, notre première observation consiste à montrer les limites de l'hyperthreading, en effet nous utilisons pas plus de 4 coeurs. Et pour rejoindre ce qui a été dit sur 2 coeurs, un plus grand nombre de petites tailles d'échantillon

11. cpu-bound



(a) speedup sur 2 coeurs (b) dynamicité sur 2 coeurs

(c) speedup sur 8 coeurs (d) dynamicité sur 8 coeurs

FIGURE 8 – Mesure 3 en $O(1)$ sur 2 et 8 coeurs

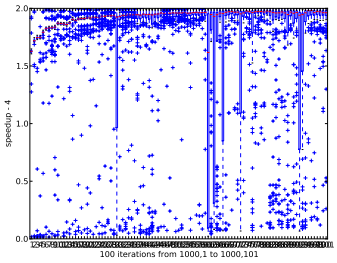
reste dominée par une execution sequentielle.

La différence entre les deux complexités de problème s'observe dans la mesure de la dynamicité. En effet pour un problème en $O(1)$, la planification statique en OpenMP permet d'avoir une meilleur efficacité de travail tandis que en $O(n)$, la planification dynamique apporte quelque amélioration.

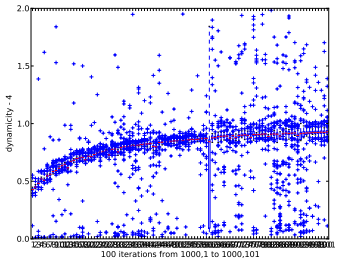
Pour résumé nous avons comparé par nos mesures les 2 types de planification de tâches en OpenMP, les 2 compléxités de problèmes que nous serons ammené à parallèliser et finalement sur 2 types de processeurs différents.

Un point que je n'ai pas eu le temps d'éclaircir et que j'invite à regarder est l'utilisation de l'auto parallèlisation intégerer au compilateur GCC [12]. Il s'agit d'un niveau d'optimisation qui consiste à verifier la non dépendance des données dans les boucles et les parallèliser.

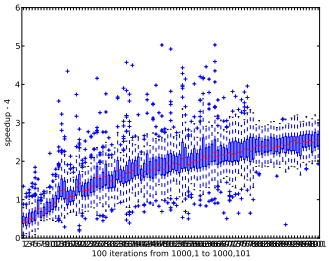
12. Automatic Parallelization : `http://gcc.gnu.org/wiki/openmp`



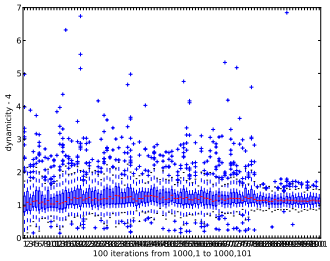
(a) speedup sur 2 coeurs



(b) dynamicité sur 2 coeurs

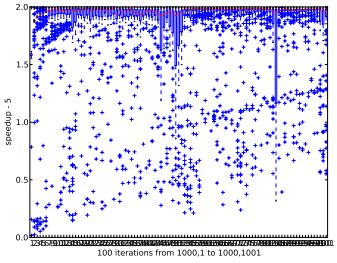


(c) speedup sur 8 coeurs

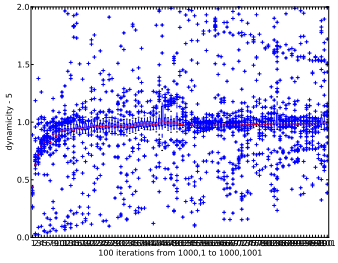


(d) dynamicité sur 8 coeurs

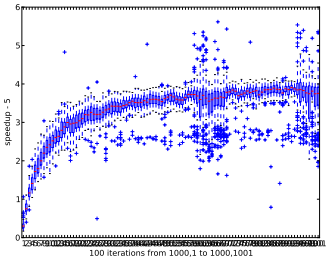
FIGURE 9 – Mesure 4 en $O(1)$ sur 2 et 8 coeurs



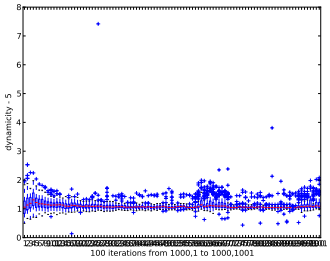
(a) speedup sur 2 coeurs



(b) dynamicité sur 2 coeurs

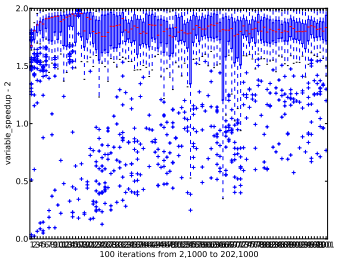


(c) speedup sur 8 coeurs

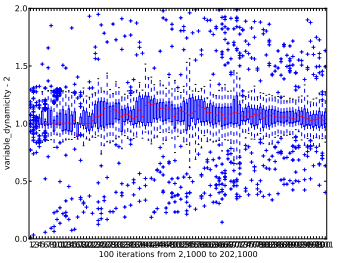


(d) dynamicité sur 8 coeurs

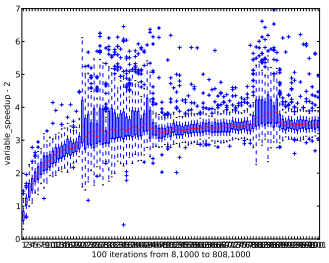
FIGURE 10 – Mesure 5 en $O(1)$ sur 2 et 8 coeurs



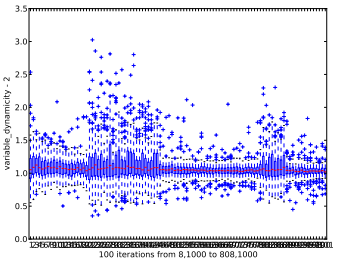
(a) speedup sur 2 coeurs



(b) dynamicité sur 2 coeurs

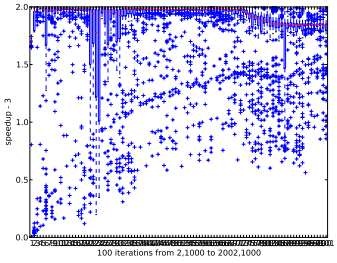


(c) speedup sur 8 coeurs

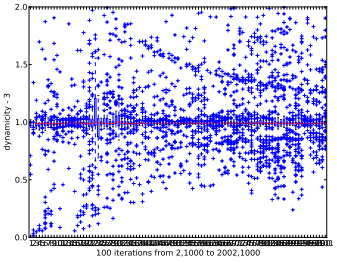


(d) dynamicité sur 8 coeurs

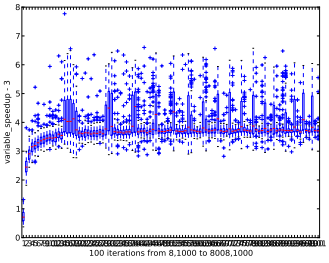
FIGURE 11 – Mesure 2 en $O(n)$ sur 2 et 8 coeurs



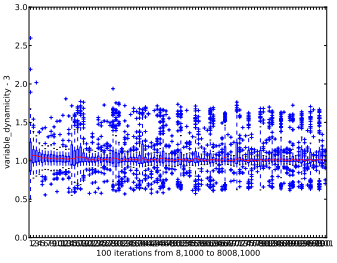
(a) speedup sur 2 coeurs



(b) dynamicité sur 2 coeurs



(c) speedup sur 8 coeurs



(d) dynamicité sur 8 coeurs

FIGURE 12 – Mesure 3 en $O(n)$ sur 2 et 8 coeurs