Machines parralèles

L3 informatique – 2016

Université Paris 8

Initiation à l'algorithme génétique et tentatives de parallélisation

Cardinal Dorian

I - Introduction

Les algorithmes dit « génétiques » sont des algorithmes permettant la résolution de problèmes d'optimisation s'appuyant sur une imitation des mécanismes d'évolution de la nature tels que les mutations, la sélection naturelle, ect...
L'idée générale est de générer une population d'individus divers, dont chacun d'eux représente une solution à un problème donné. Une suite d'opérations sur cette population nous mène une génération suivante d'individus supposée représenter une meilleure solution au problème que les individus de la génération précédente. Ainsi, de génération en génération, il est possible de s'approcher d'une solution optimale. L'algorithme prend fin lorsque nous obtenons une solution satisfaisante ou en ayant définit un nombre de génération maximum.

Le but de ce projet est d'implémenter une version simpliste d'un algorithme génétique et d'explorer des pistes concernant les possibilités de parallélisation de celui-ci.

II – Construction du programme séquentiel

II . I – La boucle principale

L'objectif de ce programme ne sera pas de résoudre un problème donné mais de reproduire le fonctionnement d'un algorithme génétique

Commençons par identifier les principales étapes d'un algorithme génétique, puis essayons d'en déterminer l'architecture du programme ainsi que les fonctions nécessaires.
La première chose dont nous avons besoin est de fixer la représentation d'un individu et d'une population. Dans un algorithme génétique classique, un individu est représenté avec des chromosomes, chiffrés avec un certain nombre de gènes, pouvant être interprété comme un caractère ou un bit. Ici nous feront simple : un individus est une suite continue d'informations, chaque information étant stocké sur un bit. Nous utiliserons le type long long int pour stocker les bits, l'individu sera alors un tableau de long long int, que nous appellerons bitarray. La population sera alors un tableau de bitarray. En outre nous aurons besoin de stocker le nombre de bits codant un individus (fixe), le nombre de long long int permettant de les stocker (de manière à pas le recalculer à chaque fois) et du nombre d'individus dans la population (fixe également).

Notre structure contenant l'ensemble des ces informations :

```
typedef unsigned long long int uli;
typedef struct problem{
      int nb_bits;
      int size_bitarray;
      int population_size;
      uli **bitarrays;
}problem;
problem p;
```

Notre première fonction servira à générer notre population de base

```
void gen(int nb_bits, int pop_size)
```

Les opérations pour passer à la génération suivante d'individus peuvent être nombreuses. Nous en implémenterons que les plus importantes.

Dans l'ordre, les opérations à effectuer sont :

- L'évaluation : elle permet de mettre une valeur à un individus, de façon à pouvoir les comparer.
- La sélection : elle permet de désigner quels seront les individus qui seront accéderont à la reproduction et ceux qui seront remplacés.
- La mutation : elle permet de sortir d'une « impasse » en faisant apparaître des nouvelles solutions.

La boucle principale ressemblera à cela :

```
int c_gen = 0;
gen();
while(1)
{
      eval_pop();
      if(!select_pop() || c_gen++ < MAX_GEN)
            break ;
      mutate();
}
```

La fonction de sélection nous renvoie le score du meilleur individus, celui-ci s'approchant de 0 lorsque la solution est optimisée. Logiquement on pourrait attribuer ce rôle à la fonction d'évaluation, mais elle n'a pas vocation à chercher le « meilleur » individus, alors que la fonction de sélection pourra le faire de façon simple, car les individus seront triés.

La question que nous devrions à présent nous poser est la suivante : Comment trier les individus et comment représenter leur score ?

Nous auront besoin de deux autres tableau, chacun de la taille de la population : un tableau contenant les scores et un tableau contenant les indices des individus.

En effet, il est beaucoup plus coûteux de trier une liste contenant des listes (**bitarray) que de trier une liste contenant des numéros d'indices. De plus, si nous trions la populations dirrectement dans le tableau dans lequel elle est stockée, nous devrons alors appliquer le même ordre pour le tableau des scores.

Nous ajoutons alors deux éléments à notre structure :

```
int *distances;
int *sorted_indexes;
```

Nous trierons alors notre population grâce à la fonction standard qsort :

```
qsort(p.sorted_indexes, p.population_size, sizeof(int), compare);
```

```
int compare(const void *arg_a, const void *arg_b)
{
      int *i1 = (int *)arg_a;
      int *i2 = (int *)arg_b;
      return p.distances[*i1] > p.distances[*i2];
}
```

De cette façons, nous pouvons accéder au « meilleur » individus de la population ou bien obtenir son score de cette façon :

```
p.bitarrays[p.sorted_indexes[0]];
p.distances[p.sorted_indexes [0]];
```

La fonction des tri se basant sur l'évaluation des individus, elle est à placer juste après celle-ci :

```
while(1)
{
	eval_pop();
	qsort(p.sorted_indexes, p.population_size, sizeof(int), compare);
	if(!select_pop() || c_gen++ < MAX_GEN)
		break ;
	mutate();
}
```

Cet algorithme pourrait s'appliquer à n'importe quel problème d'optimisation, la seule différence serait la méthode d'évaluation. Cela est comparable à l'heuristique d'une intelligence artificielle.

II . I – La fonction d'évaluation

Rappelons nous que l'objectif n'est pas de résoudre un problème, mais d'évaluer les performances de l'algorithme.

Il est tout à fait possible de s'imaginer un individus modèle représentant la solution que l'on souhaite atteindre, et d'évaluer l'algorithme sur sa capacité à se rapprocher de cet individu. De ce fait, plus le nombre de bits pas individus est grand, plus l'évaluation d'un individu sera longue.

Il est alors important de se fixer comme règle de base que :

- L'individu modèle (que nous appellerons solution) ne doit être accessible que pour la fonction d'évaluation.
- Le reste de l'algorithme fonctionnera de la même façon peu importe la fonction d'évaluation.

```
void eval_pop()
{
	for(int i = 0; i < p.population_size; i++)
	{
		p.distances[i] = 0;
		for(int j = 0; j < p.size_bitarray; j++)
		{
			uli diff = p.bitarrays[i][j] ^ p.solution[j];
			for(int k = 0; k < (j == p.size_bitarray - 1 ? p.nb_bits %
64 : 64); k++)
			{
				if( (diff & (0x1UL << (k) )) != 0 )
					p.distances[i]++;
			}
		}
	}
}
```

La fonction parcourt l'ensemble des individus. Pour chaque individus, elle parcourt tous ses bits et les comparent directement avec la solution.

Pour comparer un individus avec la solution, on effectue un xor et on compte le nombre de bits qui sont à 1. L'opération est à effectuer sur l'ensemble des long long int de l'individus.

A noter qu'il est possible de faire beaucoup plus rapide pour conter le nombre de bits à 1 de diff. Cependant, il ne faut pas que la fonction d'évaluation soit trop rapide pour garder un maximum de réalisme par rapport à une réelle fonction d'évaluation.

II . II – La fonction de sélection

Il y a de très nombreuses possibilités en matière de sélection. Il est possible de faire varier la taille de la population, et il y a de nombreuses méthodes existantes permettant de sélectionner (sélection par roulette, sélection par tournoi, ect…) et de faire se reproduire les individus.
En général, la fonction de sélection est compliquée à paramétrer et doit s'adapter à la manière dont est codé un individu.

Ici, la sélection est assez minimaliste : 50 % des individus les plus faible sont remplacés par un mélange aléatoire des 10 % des individus les plus fort.
La taille de la population reste donc fixe.

Il est important de préciser que cette méthode de sélection ne marche que parce que les bits constituant l'information sont indépendant les uns des autres. Si l'information était encodée par groupe de bits, cela n'aurait aucun sens de ne sélectionner qu'un seul bit de ce groupe.

```
int select_pop()
{
      int nb = p.population_size / 10;
      int nb2 = p.population_size / 2;
      for(int i = 0; i < nb2; i++)
      {
            for(int j = 0; j < p.nb_bits; j++)
            {
                  if(p.bitarrays[p.sorted_indexes[rand_r(&p.seed) % nb]][j / 64]
& (0x01UL << (j % 64)))
                        p.bitarrays[p.sorted_indexes[p.population_size - i - 1]]
[j / 64] |= (0x01UL << (j % 64));
                  else
                        p.bitarrays[p.sorted_indexes[p.population_size - i - 1]]
[j / 64] &= ~(0x01UL << (j % 64));
            }
      }
      return p.distances[p.sorted_indexes[0]];
}
```

II . III – La fonction de mutation

Sans celle-ci, l'algorithme se retrouve rapidement bloqué car il est assez rare qu'un mélange aléatoire d'un groupe d'individus (même s'il s'agit des 10% meilleurs) produise des individus encore meilleur que ceux choisit pour faire ce mélange.

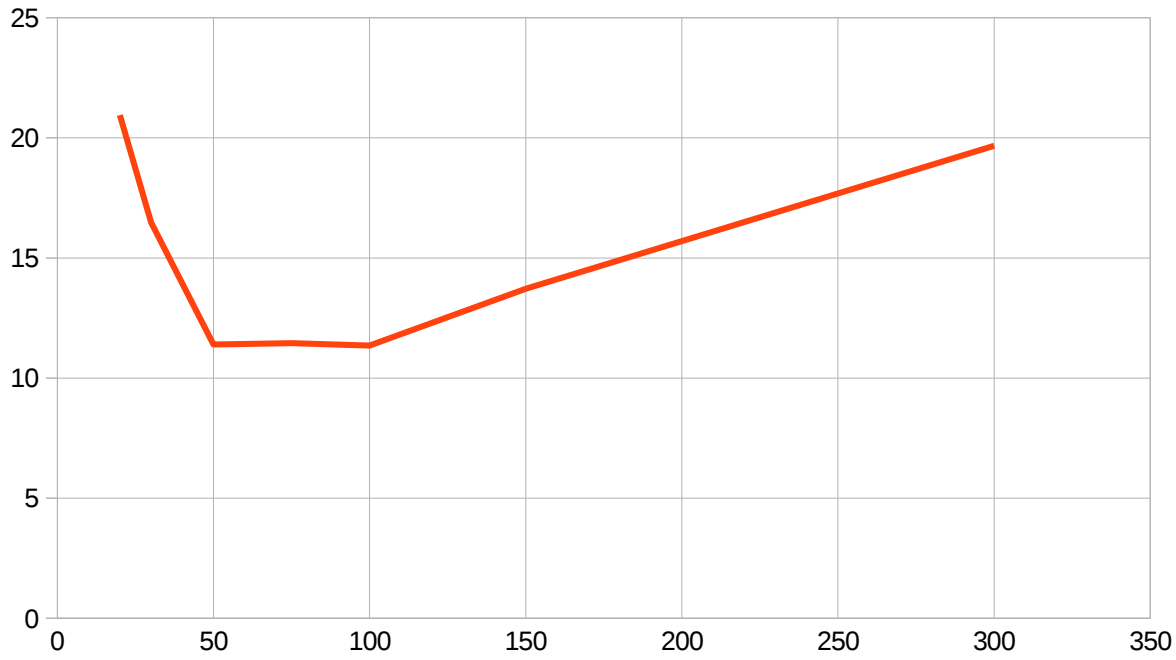
La mutation se contente de parcourir l'ensemble des individus – 1, et d'inverser aléatoirement un des bits.

Il est possible de faire intervenir une probabilité de mutation, ainsi que d'inverser plusieurs bits. Cependant, après plusieurs tests, ce paramétrage semble être le plus approprié ici .
La mutation n'affecte pas le meilleur élément car il faut s'assurer que celui-ci ne puisse pas se dégrader d'une génération à l'autre. D'après mes tests, il est inutile de sauvegarder plus d'un individu, et la sauvegarde du meilleur individu améliore significativement l'algorithme.

```
void mutate()
{
	for(int i = 1; i < p.population_size; i++)
	{
		int k = rand_r(&p.seed) % p.nb_bits;
		p.bitarrays[p.sorted_indexes[i]][k / 64] ^= (0x1UL << (k % 64));
	}
}
```

II . IV – Population optimale

Ce graph illustre l'évolution du temps de calcul (axe y, en secondes) en fonction du nombre d'individus dans la population.
Cette évolution varie fortement en fonction du paramétrage de l'algorithme.



Avec notre programme de base, sans parallélisation, la population optimisée semble se situer entre 50 et 100.

III - Parallélisation

L'objectif ici est de proposer des méthodes permettant d'améliorer le temps de calcul en séparant les calculs entre différentes unités.

III . I – Avec OpenMP

La parallélisation la plus simple consiste à répartir les principales boucles (évaluation, sélection, mutation) entre les différentes unités. Il est également possible de trier la population en parallèle, mais le gain risque d'être faible, étant donné que la taille de la population restera faible.

Avec OpenMP, les unités sont des threads, ce qui signifie que la mémoire est partagée. Au démarrage du processus, seul le thread principal est actif et c'est grâce à des directives de précompilation que l'on parvient à séparer l'exécution d'une boucle dans plusieurs threads.
L'avantage des threads, c'est que la mémoire est partagée. Cela implique que chaque thread a accès à la population générée par le thread principal. Il n'y a donc pas besoin de regrouper les informations ou de faire communiquer les threads : chaque boucle est découpée, chaque thread s'occupe d'une partie de la boucle.

Par exemple avec la mutation :

```
void mutate(problem *p)
{
      #pragma omp parallel for
      {
            for(int i = 1; i < p->population_size; i++)
            {
                  int k = rand_r(&p.seed) % p->nb_bits;
                  p->bitarrays[p->sorted_indexes[i]][k / 64] ^= (0x1UL << (k %
64));
            }
      }
}
```

Il y a cependant un problème lié à l'utilisation de l'aléatoire : Si tous les threads utilisent le même seed, il arrive que la synchronisation du seed soit faite trop tardivement et que deux threads produisent le même aléatoire. Résultat : une faible diversité dans la population et un temps d'exécution allongé.

Pour résoudre ce problème, il faut premièrement s'assurer que chaque thread utilise son propre seed (seed local), le seed local s'appuiera sur le rang du thread, mais aussi sur un seed dit global, qui devra évoluer à chaque appel de la fonction rand, pour s'assurer que les boucles produisent un résultat différent à chaque génération.

```
void mutate(problem *p)
{
      #pragma omp parallel private(local_seed)
      {
            local_seed = global_seed ^= omp_get_thread_num();
            #pragma omp for
                  for(int i = 1; i < p->population_size; i++)
                  {
                        global_seed++;
                        int k = rand_r(&global_seed) % p->nb_bits;
                        p->bitarrays[p->sorted_indexes[i]][k / 64] ^= (0x1UL <<
(k % 64));
                  }
      }
}
```

ici, local_seed et global_seed sont des variables globales. Chaque thread a son propre local_seed qu'il fixe en fonction de son rang et de global_seed.

La même logique est appliquée à chaque étapes de l'algorithme.

Un seul défaut subsiste encore : les résultats ne sont pas reproductibles. En effet, chaque thread a un seed local qui dépend du seed global et le seed global n'est toujours pas synchronisé entre les threads. De plus, l'ordre dans lequel démarrent les thread est indéfini. Quand un thread démarre, les threads précédents peuvent avoir incrémenté le seed global de 1, 2, 3, voir pas du tout. Il est même possible que tous les threads démarrent avec le même seed global. Pour résoudre ce problème il faudrait forcer les threads à se synchroniser et l'on perdrait efficacité.
Les temps d'exécution peuvent alors varier d'environ 20%.

A noter qu'il est possible de forcer le nombre de threads, mais j'ai préféré ne rien changer car ce nombre est déjà définit en fonction du nombre de coeurs logiques dont je dispose (4 ici, grâce à l'hyper threading).

III . I – Avec MPI

Avec MPI, au démarrage du programme, plusieurs processus sont créés. Le code est le même, la seule différence entre les processus étant leur rang.

Étant donné que nos individus sont sous forme de pointeur, il n'est absolument pas rentable de séparer les boucles entre les processus : il faudrait faire nb_bits/64 communications pour envoyer un individu d'un processus à un autre, regrouper tous les individus pour les trier puis les redistribuer prendrait un temps inconsidérable.

Une autre approche est alors possible : on a vu que l'aléatoire est un facteur qui joue beaucoup dans notre algorithme. Si chaque processus démarre avec un seed unique, chaque processus fera tourner un algorithme génétique qui n'aura pas la même efficacité que son processus voisin. On pourra alors identifier quel est le processus le plus efficace, et celui-ci peut alors transmettre ses meilleurs individus aux autres processus.

La seule étape que nous aurons alors besoin de modifier est la sélection.

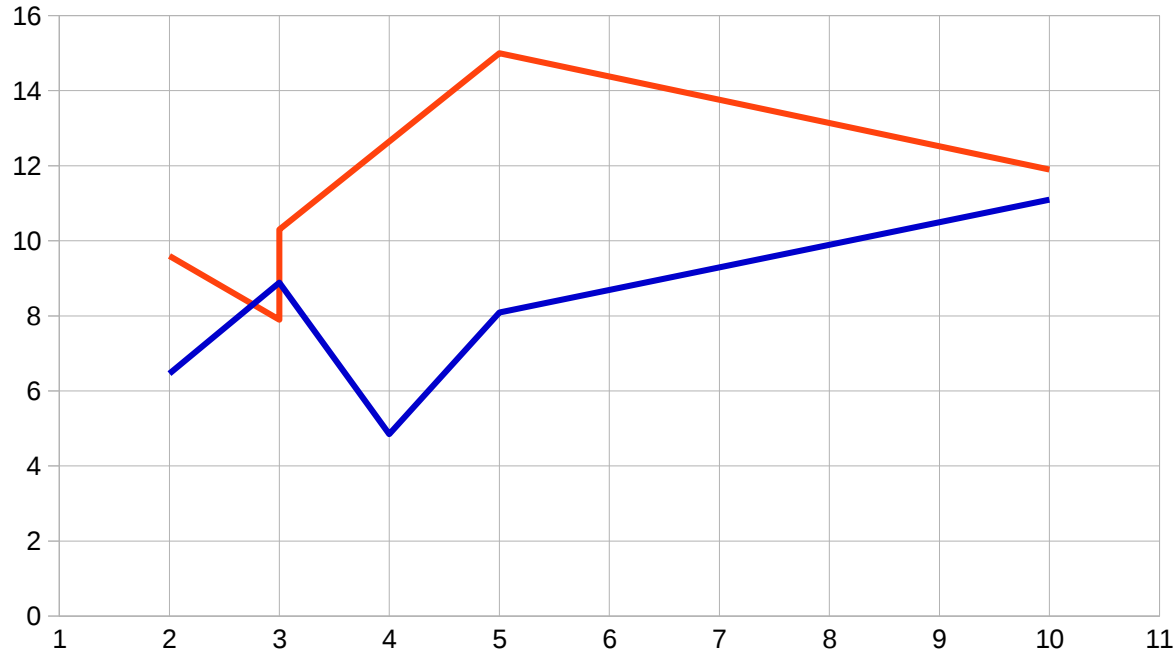
```
int best = p->nb_bits;
int best_i = 0;
int bests[p->msize];
MPI_Allgather(&p.distances[p.sorted_indexes[0]], 1, MPI_INT, bests, 1, MPI_INT,
MCW);
for(int i = 0; i < p->msize; i++)
{
      if(bests[i] < best)
      {
            best_i = i;
            best = bests[i];
      }
}
```

Ce code permet de savoir quel est le processus disposant du meilleur individu.
Le meilleur score du meilleur individu de chaque processus étant p.distances[p.sorted_indexes[0]], la directive Allgather permet que chaque processus aient un tableau (dont la taille sera le nombre de processus totaux) contenant les meilleurs scores des processus, dans le bon ordre.
Puis, chaque processus fait la même opération lui permettant de connaître le rang du processus détenant le meilleur individu.

```
MPI_Bcast(p->bitarrays[p->sorted_indexes[0]], p->size_bitarray,
MPI_UNSIGNED_LONG, best_i, MCW);
```

Une fois que l'on connaît le processus dont il faut prendre le meilleur individu (et que ce numéro est le même pour chaque processus), la directive Bcast permet au processus dont best_i correspond à son rang d'envoyer son meilleur individu à tous les autres.

Le graph suivant nous montre l'évolution du temps de calcul en fonction du nombre de processus, pour 10000 bits et une population de 100 (orange) et 50 (bleu).

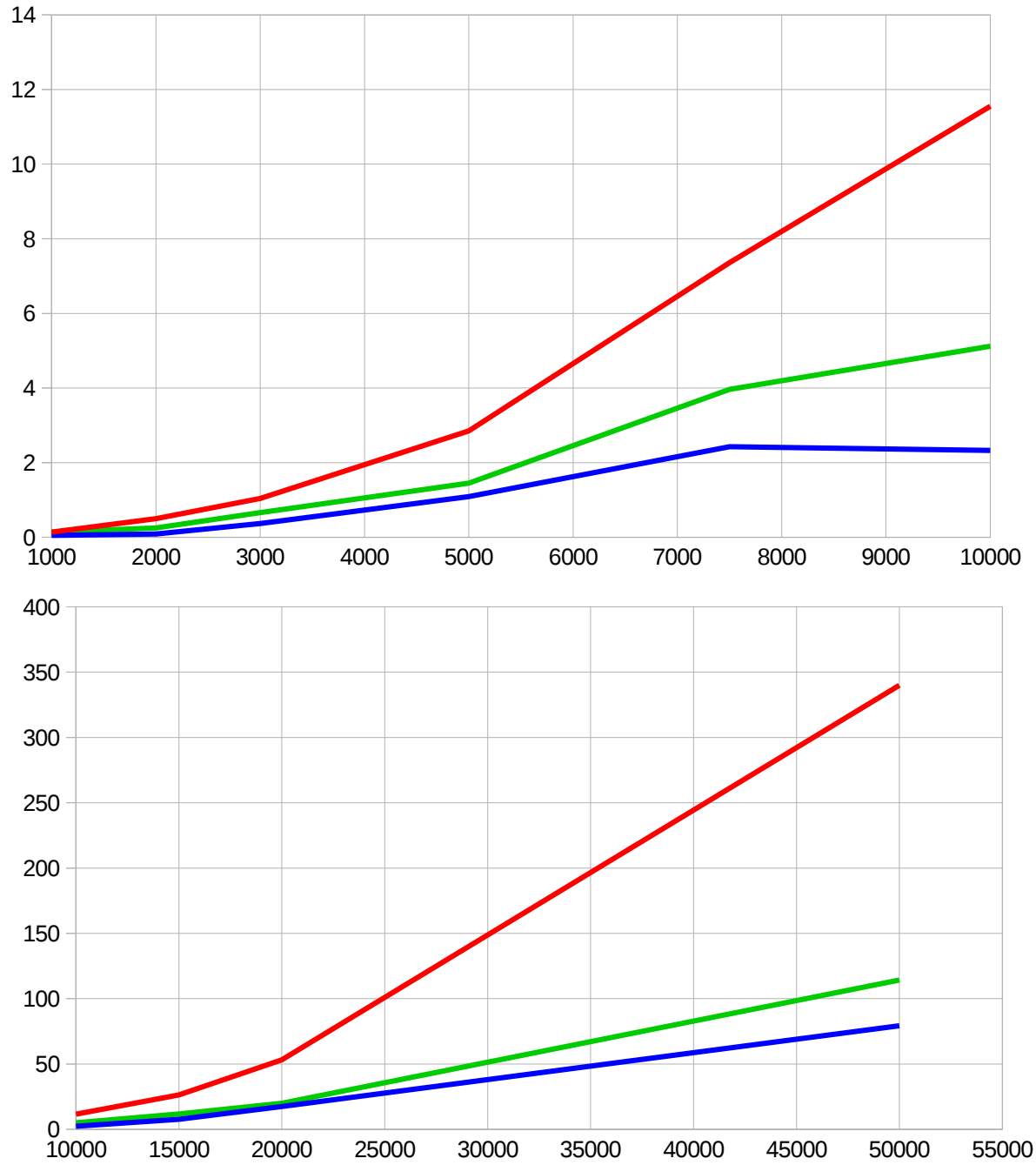


Avec MPI, il semble préférable d'utiliser une population de 50 par processus, avec 4 processus.

IV – Résultat et conclusion

Maintenant que nous avons nos 3 programmes et que nous connaissons leur paramétrage optimal, comparons-les.

Le graph ci-dessous montre l'évolution du temps de calcul en fonction du nombre de bits.
En rouge, le programme séquentiel, avec une population de 100.
En vert, le programme parallèle avec MPI, avec une population de 50 par processus et 4 processus.
En bleu, le programme avec OpenMP, avec une population de 100 et 4 threads.



Le programme MPI semble être le plus rapide peu importe la configuration. Cependant, il pourrait être intéressant de voir les performances du programme MPI si on avait fait en sorte que chaque processus ait ses propres paramètres.