

ST7 :  
SIMULATION HAUTE PERFORMANCE  
POUR LA RÉDUCTION D'EMPREINTE

---

# OPTIMISATION PAR COLONIES DE FOURMIS

---

Simon Maréchal  
Matthieu Oberon  
Emile Prost  
Carlos Santos García  
Paul Saurou

7 avril 2021

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Iso3dfd . . . . .	3
1.2	Objectifs . . . . .	3
1.3	Optimisation par colonie de fourmis (ACO) . . . . .	3
<b>2</b>	<b>Modélisation du problème</b>	<b>4</b>
2.1	Arbre parcouru . . . . .	4
2.2	Matrice de phéromones . . . . .	4
2.3	Coûts . . . . .	6
2.4	Stratégies des fourmis . . . . .	6
2.4.1	Stratégie classique . . . . .	6
2.4.2	Stratégie élitiste . . . . .	7
2.4.3	Stratégie Max-Min . . . . .	7
2.4.4	Stratégie des fourmis folles . . . . .	7
2.4.5	Stratégie de différenciation des coûts . . . . .	7
<b>3</b>	<b>Parallélisation des méthodes d’optimisation</b>	<b>8</b>
3.1	Algorithme à communication collective . . . . .	8
3.2	Algorithme à communication point à point . . . . .	9
3.3	Déploiement sur le cluster . . . . .	10
<b>4</b>	<b>Difficultés rencontrées</b>	<b>11</b>
4.1	Appropriation de sbatch . . . . .	11
4.2	Multithreading . . . . .	11
4.3	Parallélisation de iso3dfd . . . . .	12
4.4	Incertitudes . . . . .	12
<b>5</b>	<b>Résultats préliminaires</b>	<b>13</b>
5.1	Données brutes . . . . .	13
5.2	Comparaison de la communication collective et de la communication point à point . . . . .	14
5.3	Speed up . . . . .	16
5.4	Multithreading . . . . .	17
<b>6</b>	<b>Expérimentations avec les stratégies et les hyper-paramètres</b>	<b>18</b>
6.1	Colonie de fourmis classique . . . . .	18
6.2	Système élitiste . . . . .	20
6.3	Système de fourmis folles . . . . .	22
6.4	Initialisation de la matrice de phéromones . . . . .	24
6.5	Comparaison des stratégies . . . . .	24
<b>7</b>	<b>Conclusion</b>	<b>26</b>

# 1 Introduction

## 1.1 Iso3dfd

Les méthodes aux différences finies sont des outils particulièrement adaptés pour la résolution d'équations différentielles [1]. Cette méthode, facile à implémenter et permettant un ajustement facile de la précision souhaitée, est néanmoins gourmande en ressources CPU.

Le programme iso3dfd développé par Intel permet l'implémentation de la résolution par différences finies 3D de l'équation de propagation d'ondes sismiques  $\frac{\partial^2 p}{\partial t^2} = c^2 \nabla^2 p$ . D'un point de vue plus pragmatique, l'exécutable iso3dfd prend en arguments huit variables : les dimensions du domaine modélisé ( $n_1, n_2, n_3$ ), le nombre de threads sur lesquels l'exécutable est lancé, le nombre d'itérations et les paramètres de cache blocking (cbx, cby et cbz).

## 1.2 Objectifs

La finalité de ce projet est d'utiliser des techniques de colonies de fourmis pour l'optimisation du déploiement du programme iso3dfd. Pour cela, on pourra jouer sur les tailles des structures de données mises en jeu lors de la résolution de l'équation. On pourra aussi modifier les options de compilation du code dans le but de déterminer la meilleure stratégie de déploiement du code.

Pour évaluer les différentes stratégies nous nous focaliserons sur un critère tel que le temps de calcul, la vitesse de calcul ou la consommation énergétique. On utilisera pour cela les ressources du cluster à notre disposition, afin de paralléliser la recherche de ce point de fonctionnement optimal.

## 1.3 Optimisation par colonie de fourmis (ACO)

Les algorithmes de colonies de fourmis sont une classe d'algorithmes d'optimisation méta-heuristique calquée sur le fonctionnement des fourmilières. Les fourmis sont des insectes sociaux qui communiquent à l'aide de phéromones. Tous comme les fourmis réelles, nos fourmis artificielles se déplacent dans un espace de paramètres, et déposent des phéromones au cours de leur parcours afin de mettre en évidence un chemin optimal. Le cadre de ces méthodes limite cependant leur application aux problèmes modélisables sous forme de graphes, comme le voyageur de commerce. [2].

L'intégralité de notre code est accessible sur le gitlab ViaRézo à l'adresse suivante :

<https://gitlab.viarezo.fr/2019marechals/st7-intel>.

## 2 Modélisation du problème

### 2.1 Arbre parcouru

On modélise le problème par un arbre à  $1 + 6$  couches (voir figure 1). La racine représente un état initial, depuis lequel toutes les fourmis partent. Les couches 1 à 6 représentent respectivement les paramètres  $n1$ ,  $n2$ ,  $n3$ ,  $cbx$ ,  $cby$ ,  $cbz$  à optimiser. Le poids d'une branche entre ces noeuds représentent alors la quantité de phéromones disposées sur le chemin entre ces deux paramètres. Plus la quantité de phéromones est importante, plus il y a de chances qu'une fourmi emprunte ce chemin. Étant limités par la puissance du cluster, nous ne modéliserons pas des tailles  $n_1$ ,  $n_2$  et  $n_3$  supérieures à 512.

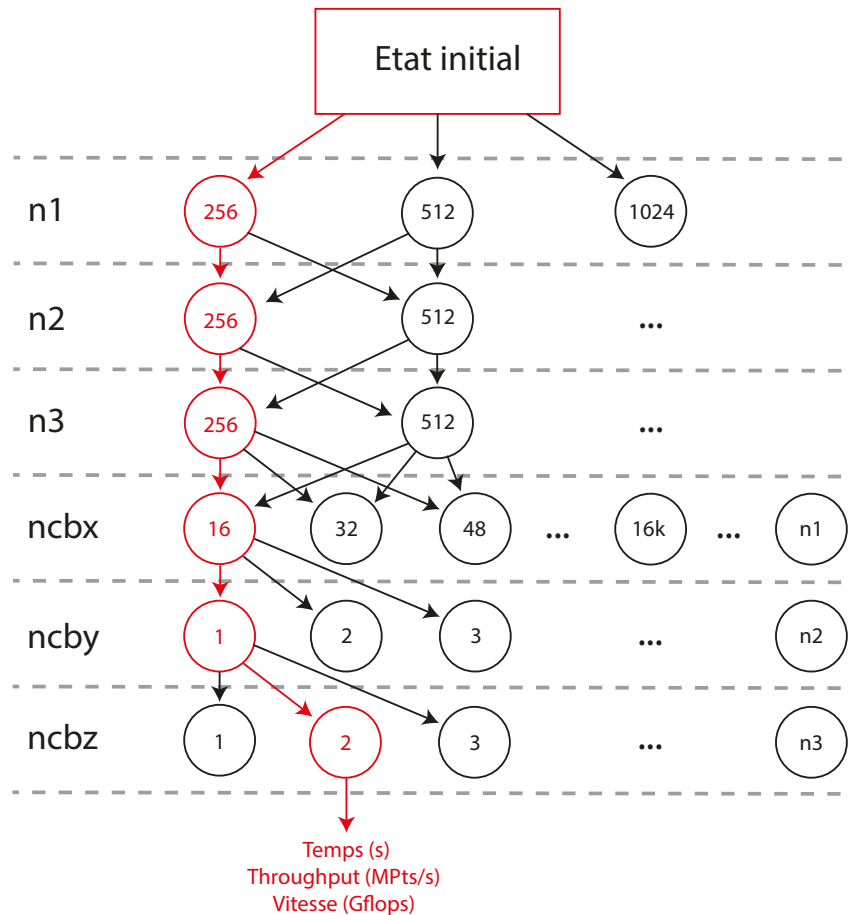


FIGURE 1 – Arbre modélisant notre problème

### 2.2 Matrice de phéromones

La matrice de phéromones représente la matrice d'adjacence de notre graphe. Sa taille est fixe tout au long de l'exécution de l'algorithme ACO.

$$\tau = \begin{pmatrix} 0 & \underbrace{\tau_{1 \rightarrow n1}}_{1 \times 2} & & & & & & \\ 0 & & \underbrace{\tau_{n1 \rightarrow n2}}_{2 \times 2} & & & & 0 & \\ 0 & & & \underbrace{\tau_{n2 \rightarrow n3}}_{2 \times 2} & & & & \\ 0 & & & & \underbrace{\tau_{n3 \rightarrow ncbx}}_{2 \times 32} & & & \\ 0 & & 0 & & & \underbrace{\tau_{ncbx \rightarrow ncby}}_{32 \times 512} & & \\ 0 & & & & & & \underbrace{\tau_{ncby \rightarrow ncbz}}_{512 \times 512} & \\ 0 & & & & & & & \end{pmatrix}$$

La première colonne de 0 est inutile d'un point de vue théorique, mais est utile dans notre implémentation, car elle simplifie énormément la gestion des indices : les indices des tableaux dans Python commencent à 0, alors que les paramètres ont comme valeur minimale 1.

Les phéromones laissées par des fourmis lors du parcours du graphe permettent aux générations futures de choisir les combinaisons de paramètres les plus prometteuses. Pour cela, la quantité de phéromones sur une arête du graphe est mise à jour par l'ensemble des fourmis ayant pris cette arête et dépend des performances obtenues par chacune d'elles : plus le gain est important, plus les fourmis laissent des phéromones pour inciter leurs camarades à prendre ce chemin. Au contraire, une arête prise par aucune fourmi verra ses phéromones s'évaporer peu à peu, suite au manque d'intérêt des fourmis pour ce chemin. À chaque fin de génération, la matrice  $\tau$  doit donc être mise à jour. L'équation pour mettre à jour les phéromones de l'arête  $\tau_{i,j}$  prise par un ensemble  $K$  de fourmis, où le coût associé au chemin pris par chacun d'elles est noté  $c_i$  est :

$$\tau_{i,j} \leftarrow (1 - \rho)\tau_{i,j} + \sum_{k \in K} Qc_k$$

Le paramètre  $Q$  est la quantité fixe de phéromones que les fourmis laissent, fixé dans notre implémentation à  $Q = 0.1$ . Le paramètre  $\rho$ , appelé coefficient d'évaporation des phéromones, a aussi été fixé dans un premier temps à  $\rho = 0.1$ . Cette équation de mise à jour des phéromones est susceptible d'être modifiée lors de l'utilisation de stratégies différentes pour l'exploration du graphe, mais reste le pilier de notre algorithme.

Pour les raisons citées ci-dessus, la taille maximale des paramètres  $n_i$  est 512,  $\tau$  a alors pour taille  $551 \times 1063$ . Dans ce cas, le taux de remplissage de la matrice de phéromones est proche de 47%, ce qui rend l'utilisation de matrices creuses peu pertinente : en effet, d'après nos vérifications, ces dernières deviennent intéressantes pour des calculs matriciels dès lors que le taux de remplissage des matrices est inférieur à 3%. Mais

comme  $\tau$  nous permet uniquement de stocker des valeurs de phéromones posées sur les branches de l'arbre, nous n'aurons pas à faire de calcul matriciel avec. Pour ce qui est de l'espace de stockage utilisé par la matrice, une *sparse matrix* du module *scipy* nécessite 3 tableaux pour stocker les valeurs (numéro de ligne et de colonne à stocker en plus pour chaque valeur), ce qui rend son utilisation plus gourmande en espace de stockage qu'une matrice numpy dès lors que le taux de remplissage de la matrice est supérieur à 33%. L'utilisation de matrices denses paraît alors adéquat.[3]

## 2.3 Coûts

Une fois le chemin choisi, le programme `iso3dfd` est lancé avec les paramètres des noeuds rencontrés sur le chemin, via la commande

```
$ iso3dfd_dev13_cpu_avx512.exe n1 n2 n3 8 100 cbx cby cbz
```

Ce programme renvoie trois informations : le temps d'exécution (*Time*, en secondes), la vitesse d'exécution (*Throughput*, en millions de points par seconde) ainsi que le nombre d'opérations par secondes (*Flops*, en milliards d'opération par seconde) :

```
time:          7.66 sec
throughput:    180.57 MPoints/s
flops:         11.01 GFlops
```

Pour notre étude, nous allons considérer le *throughput* comme gain, que nous allons donc chercher à maximiser.

## 2.4 Stratégies des fourmis

Pour se déplacer dans l'espace des paramètres, les fourmis choisissent aléatoirement entre tous les chemins disponibles. En se déplaçant elles déposent des phéromones qui augmentent la probabilité que le chemin soit choisi par une des fourmis lors de l'itération suivante de l'algorithme. La quantité de phéromones déposée par chaque fourmi dépend du gain de son parcours dans l'espace des paramètres d'une part et d'une stratégie choisie d'autre part. On peut imaginer une infinité de stratégies différentes ; nous avons choisi d'étudier ici celles d'entre elles qui nous paraissaient les plus intéressantes, que ce soient des solutions naïves plutôt classiques ou des stratégies plus performantes ou bien adaptées au problème.

### 2.4.1 Stratégie classique

Dans cette stratégie, toutes les fourmis laissent des phéromones dans leur parcours du graphe. La quantité de phéromones déposée dépend linéairement du gain du parcours emprunté.

Cette stratégie est la plus simple possible et permet de comparer la pertinence des autres stratégies pour notre problème.

### 2.4.2 Stratégie élitiste

Cette stratégie, publiée en 1996 dans [4], fonctionne presque comme la stratégie classique. La seule différence réside dans le fait qu'à chaque itération la meilleure fourmi laisse deux fois plus de phéromones sur le chemin qu'elle a emprunté.

Cette stratégie permet une convergence plus rapide en récompensant plus fortement le meilleur trajet.

### 2.4.3 Stratégie Max-Min

La stratégie Max-Min, parue en [5], ne récompense que la meilleure fourmi de chaque génération. Toutefois, la quantité de phéromones est bornée : l'évanescence ne peut faire diminuer la quantité de phéromones en dessous du seuil minimal et même si beaucoup de fourmis très performantes empruntent le même chemin, sa quantité de phéromones ne dépassera pas le seuil maximal. Elle est présentée dans la littérature scientifique [6] comme l'une des stratégies aux meilleures performances grâce à la place prépondérante de l'exploration assurée par la borne minimale. Cependant, ceci est fait au détriment de la vitesse de convergence de l'algorithme, qui explore des solutions souvent peu optimales.

Cette stratégie permet de continuer l'exploration de nouvelles pistes et d'éviter de rester bloqué dans un minimum local de la fonction de coût.

### 2.4.4 Stratégie des fourmis folles

Cette stratégie, inspirée des stratégies  $\epsilon$ -greedy en apprentissage par renforcement, et similaire à celle présentée comme Ant Colony System dans [7], consiste à introduire des fourmis folles dans la colonie. Chaque fourmi choisit un chemin prometteur grâce aux phéromones déposées à l'itération précédente avec une probabilité de  $1 - \epsilon$ , et a une probabilité  $\epsilon$  d'ignorer complètement les phéromones dans son choix de parcours pour partir explorer d'éventuelles solutions. Elle va alors choisir des paramètres aléatoirement et potentiellement explorer des chemins qu'elle aurait eu une chance infinitésimale d'explorer.

Cette stratégie permet de conserver une composante d'exploration dans l'algorithme sans l'empêcher de converger vers une solution. Toutefois, elle augmente le temps d'exécution de l'algorithme d'optimisation puisqu'elle freine la convergence. Cette stratégie peut se combiner à d'autres stratégies et oblige à trouver un compromis entre exploration de chemins aléatoires et réduction du temps de calcul en jouant sur la proportion de fourmis folles, choisie en tant qu'hyper-paramètre.

### 2.4.5 Stratégie de différenciation des coûts

Cette stratégie consiste à changer la façon dont les phéromones sont déposées. Elles ne sont plus déposées linéairement en fonction du coût mais suivant une fonction dépendant du coût du chemin emprunté. Lorsque nous avons testé cette stratégie, nous

avons sélectionné une sigmoïde pour augmenter l'influence des chemins performants par rapport aux chemins peu performants. Cela permet d'accélérer la convergence de l'algorithme sans empêcher l'exploration d'autres pistes puisque tous les chemins empruntés se verront récompensés en phéromones.

Cette stratégie s'inscrit dans le même esprit que la stratégie élitiste mais va plus loin car elle s'applique à tous les chemins empruntés et pas seulement au plus performant. Elle peut se combiner aux autres stratégies.

### **3 Parallélisation des méthodes d'optimisation**

L'algorithme des colonies de fourmis nécessite de calculer de nombreuses fois le coût des solutions trouvées (autant qu'on a de fourmis). Il est donc intéressant de le paralléliser, principalement au niveau du calcul du coût qui est l'étape la plus longue. On distribue alors les fourmis sur plusieurs processus où sont calculés le gain de chaque chemin suivi par ces fourmis. Cette parallélisation nécessite cependant qu'à la fin de chaque génération, les processus communiquent entre eux afin de mettre à jour la matrice de phéromones.

#### **3.1 Algorithme à communication collective**

La première implémentation se base sur des communications collectives.

Sur chaque processus, nous faisons parcourir dans le graphe une partie des fourmis du batch global puis nous calculons les gains associés. Nous transmettons alors tous les chemins et les coûts au processus 0 avec un 'gather' afin de mettre à jour la matrice de phéromones. Cette dernière est finalement renvoyée à tous les processus à l'aide d'un 'broadcast'. Le schéma en figure 2 représente la situation décrite ci-dessus.

Nous avons ensuite essayé une deuxième méthode de parallélisation car cette méthode nécessite de communiquer à chaque itération la matrice de phéromones qui peut être très grande si l'on augmente la taille du problème.



## PARALLELISATION PAR BROADCAST

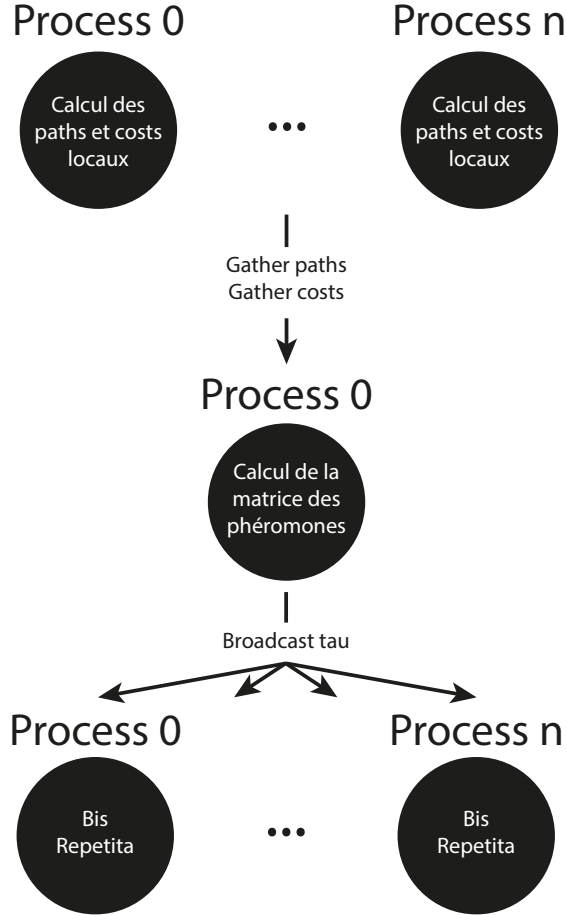


FIGURE 2 – Parallélisation par Broadcast

### 3.2 Algorithme à communication point à point

La deuxième implémentation de la parallélisation se base sur des communications point-à-point, sous la forme d'un anneau. Chaque processus calcule les chemins et les coûts associés à ses fourmis puis met à jour localement la matrice de phéromones avec ses fourmis. Ses fourmis sont ensuite envoyées au processus d'après et il reçoit les fourmis du processus d'avant, qu'il utilise pour à nouveau mettre à jour la matrice de phéromones. On répète cela jusqu'à ce que tous les processus aient traité toutes les fourmis. Cette méthode limite la taille des données échangées car à chaque communication chaque processus n'envoie que  $(n_{\text{processus}} - 1) \times n_{\text{fourmis locales}} \simeq n_{\text{total de fourmis}}$  valeurs, ce qui est bien plus faible que la taille de la matrice de phéromones. Le schéma en figure 3 représente l'anneau décrit ci-dessus.

## PARALLELISATION PAR ANNEAUX

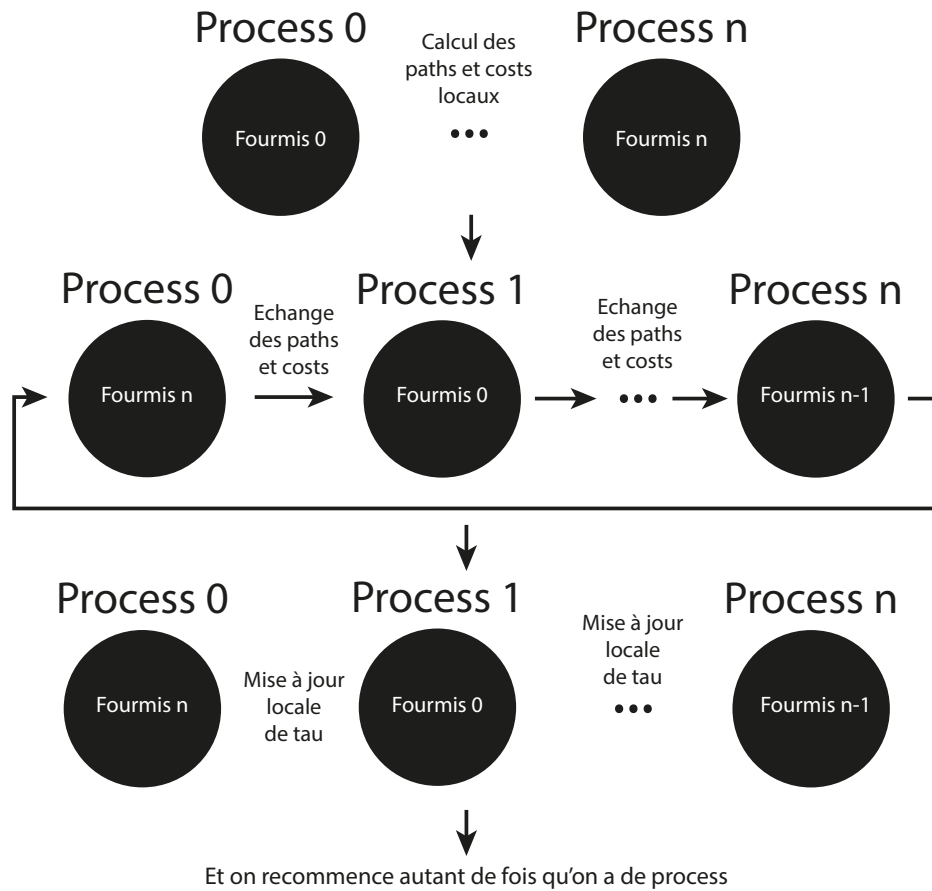


FIGURE 3 – Parallélisation par Communication Point à Point

### 3.3 Déploiement sur le cluster

Nous avons essayé deux déploiements différents pour exécuter notre algorithme parallélisé : soit en mettant 1 processus par coeur, soit en en mettant 1 par socket (cf. figure 4).

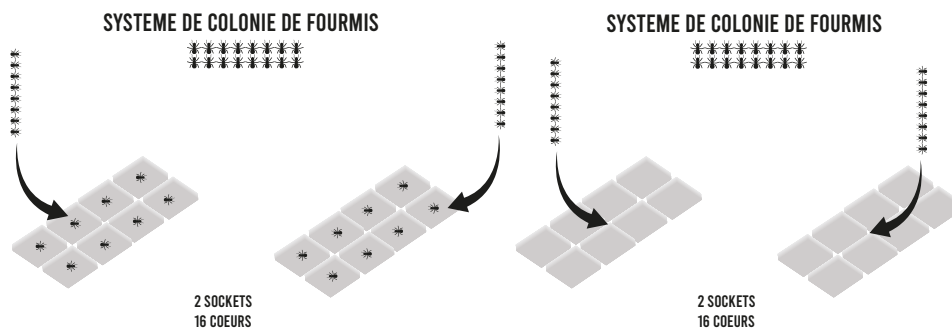


FIGURE 4 – À gauche "1 processus par core" et à droite "1 processus par socket"

Nous avons décidé de faire chaque lancement d'iso3dfd sur un socket entier en parallélisant sur 8 threads. Cela permet d'utiliser tous les coeurs physiques d'un socket. Il en découle que chaque processus est lancé sur un socket entier. Comme nous pouvons utiliser un maximum de huit noeuds par batch, la configuration utilisée pour tous les résultats de ce rapport, sauf mention explicite du contraire, est la suivante :

- Parallélisation de notre algorithme d'optimisation sur huit noeuds
- Un process par socket (donc un total de 16 process)

## 4 Difficultés rencontrées

### 4.1 Appropriation de sbatch

Nous avons eu quelques difficultés au départ pour nous approprier sbatch. En effet, nous utilisions les mauvaises partitions et le mauvais `-qos` (`-qos=max16` au lieu de `-qos=max8`) pour sbatch. Cela eut pour conséquence de soit nous retourner une erreur, soit de mettre le batch en "QOSMaxNodePerUserLimit". M. Vialle nous a alors indiqué la manière correcte d'utiliser sbatch et ces problèmes disparurent.

### 4.2 Multithreading

Comme cela est expliqué plus haut, l'exécutable iso3dfd peut être déployé en parallèle. Cette parallélisation est déterminée par le quatrième paramètre fourni à l'exécutable lorsqu'il est appelé. Or au début du projet, en testant directement en ligne de commande différents paramètres (1, 4 et 8 threads), le code lançait des processus uniquement sur quatre threads, quelque soit le paramètre que nous lui fournissions. En effet, en utilisant la commande `top -iH` nous vîmes les différents threads du programme (voir 5). Quand nous avons communiqué le problème à M. Thierry, ce dernier nous a fourni une version du code plus récente pour laquelle le multithreading prend en compte le paramètre passé en argument dans la commande. Ainsi, nous pouvons lancer iso3dfd sur 8 threads (comme cela est montré dans la figure 5).

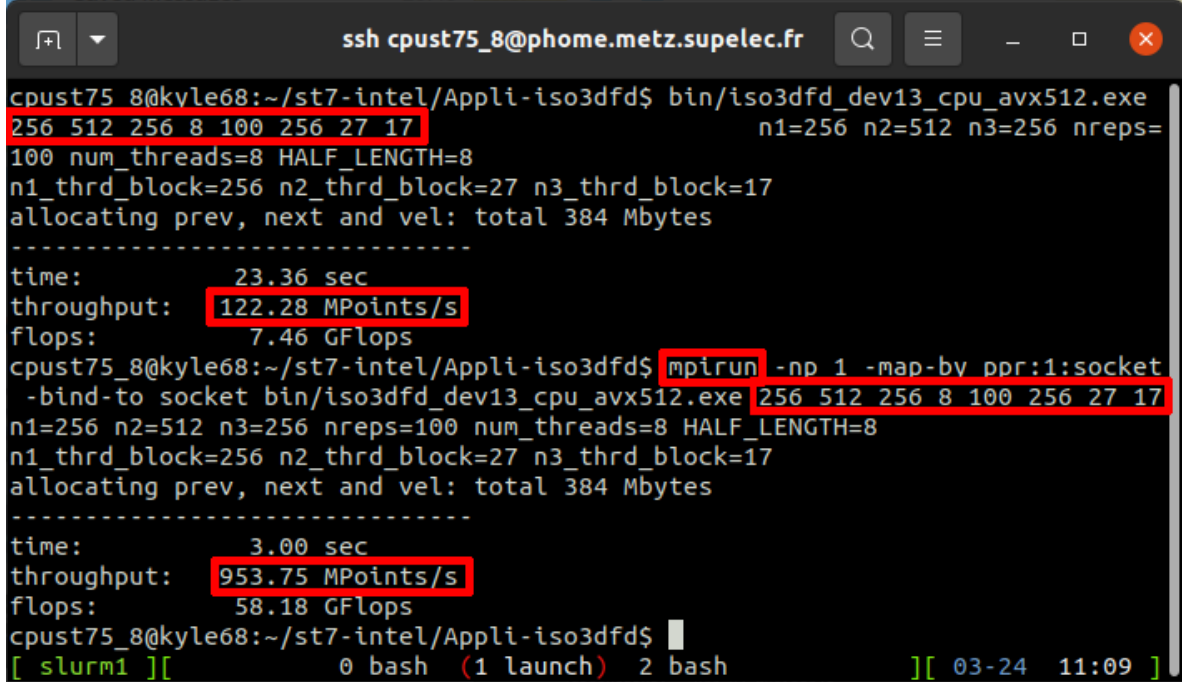
```
top - 16:31:24 up 18 days, 5:46, 1 user, load average: 4,48, 1,39, 0,50
Threads: 861 total, 9 en cours, 671 en veille, 0 arrêté, 0 zombie
%Cpu(s): 5,7 ut, 0,6 sy, 0,0 ni, 93,7 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem : 65420940 total, 46145280 libr, 13773284 util, 5502376 tamp/cache
KiB Éch: 33554428 total, 33554428 libr, 0 util, 50986328 dispo Mem
```

PID	UTIL.	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TEMPS+	COM.
20167	cpust75+	20	0	12,531g	0,012t	4328	R	25,1	19,2	0:20.58	iso3dfd_dev13_c
20170	cpust75+	20	0	12,531g	0,012t	4328	R	25,1	19,2	0:11.37	iso3dfd_dev13_c
20172	cpust75+	20	0	12,531g	0,012t	4328	R	25,1	19,2	0:11.37	iso3dfd_dev13_c
20173	cpust75+	20	0	12,531g	0,012t	4328	R	25,1	19,2	0:11.37	iso3dfd_dev13_c
20174	cpust75+	20	0	12,531g	0,012t	4328	R	25,1	19,2	0:11.37	iso3dfd_dev13_c
20171	cpust75+	20	0	12,531g	0,012t	4328	R	24,8	19,2	0:11.37	iso3dfd_dev13_c
20175	cpust75+	20	0	12,531g	0,012t	4328	R	24,8	19,2	0:11.37	iso3dfd_dev13_c
20176	cpust75+	20	0	12,531g	0,012t	4328	R	24,8	19,2	0:11.36	iso3dfd_dev13_c
20781	cpust75+	20	0	36016	4660	3188	R	1,0	0,0	0:00.32	top

FIGURE 5 – Processus exécutés lorsque iso3dfd est exécuté sur 8 threads.

### 4.3 Parallélisation de iso3dfd

Une autre difficulté majeure que nous avons rencontrée concerne la compréhension du fonctionnement et du déploiement de iso3dfd. En effet, en lançant via la ligne de commande iso3dfd seul et en le lançant via mpirun, les performances mesurées sont très différentes (voir figure 6). L'origine de ce phénomène étant assez floue (problème de compatibilité hardware et software), il nous a été conseillé de lancer iso3dfd avec mpirun pour obtenir les meilleurs résultats possibles, vu que le problème n'aurait pas pu être résolu avant la fin du projet.



```
ssh cpust75_8@phome.metz.supelec.fr
cpust75_8@kyle68:~/st7-intel/Appli-iso3dfd$ bin/iso3dfd_dev13_cpu_avx512.exe
256 512 256 8 100 256 27 17 n1=256 n2=512 n3=256 nreps=
100 num_threads=8 HALF_LENGTH=8
n1_thrd_block=256 n2_thrd_block=27 n3_thrd_block=17
allocating prev, next and vel: total 384 Mbytes
-----
time: 23.36 sec
throughput: 122.28 MPoints/s
flops: 7.46 GFlops
cpust75_8@kyle68:~/st7-intel/Appli-iso3dfd$ mpirun -np 1 -map-by ppr:1:socket
-bind-to socket bin/iso3dfd_dev13_cpu_avx512.exe 256 512 256 8 100 256 27 17
n1=256 n2=512 n3=256 nreps=100 num_threads=8 HALF_LENGTH=8
n1_thrd_block=256 n2_thrd_block=27 n3_thrd_block=17
allocating prev, next and vel: total 384 Mbytes
-----
time: 3.00 sec
throughput: 953.75 MPoints/s
flops: 58.18 GFlops
cpust75_8@kyle68:~/st7-intel/Appli-iso3dfd$
[ slurm1 ][ 0 bash (1 launch) 2 bash ][ 03-24 11:09 ]
```

FIGURE 6 – Différence de performances de iso3dfd selon si on le lance avec mpirun ou seul.

### 4.4 Incertitudes

Nous nous sommes rapidement rendu compte de l'incertitude assez grande qu'il existe sur les métriques renvoyées par iso3dfd (le throughput, les GFlops et le temps). En exécutant plusieurs fois iso3dfd avec  $n_1 = 256$ ,  $n_2 = 512$ ,  $n_3 = 512$ ,  $n_{\text{threads}} = 8$ ,  $\text{reps} = 100$ ,  $\text{cbx} = 256$ ,  $\text{cby} = 9$ ,  $\text{cbz} = 232$ , nous mesurons la moyenne et l'écart type du throughput et des GFlops. Nous obtenons alors :  $\text{throughput} = 1070 \pm 60 \text{ MPoints/s}$  et  $\text{GFlops} = 66 \pm 5 \text{ GFlops}$ . L'intervalle d'incertitude est donc de l'ordre de 10% de la valeur moyenne. Cela explique pourquoi certains résultats ne semblent pas toujours consistants.

Ces incertitudes sont liées, d'une part, au bruit inhérent des machines sur lesquelles divers processus sont lancés en continu, mais aussi à l'hétérogénéité du cluster, où chaque noeud travaille à des vitesses différentes. Cette dernière remarque est illustrée

en figure 7.

```
Path followed at iteration 299 on process 0 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -718.73
Path followed at iteration 299 on process 1 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -754.07
Path followed at iteration 299 on process 2 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -748.86
Path followed at iteration 299 on process 3 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -762.38
Path followed at iteration 299 on process 4 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -802.74
Path followed at iteration 299 on process 5 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -804.59
Path followed at iteration 299 on process 6 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -717.34
Path followed at iteration 299 on process 7 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -746.17
Path followed at iteration 299 on process 8 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -803.93
Path followed at iteration 299 on process 9 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -786.55
Path followed at iteration 299 on process 10 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -1011.32
Path followed at iteration 299 on process 11 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -710.92
Path followed at iteration 299 on process 12 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -744.75
Path followed at iteration 299 on process 13 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -773.93
Path followed at iteration 299 on process 14 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -804.47
Path followed at iteration 299 on process 15 by ant 1 : [512, 512, 256, 496, 17, 30] with cost equal to -828.93
```

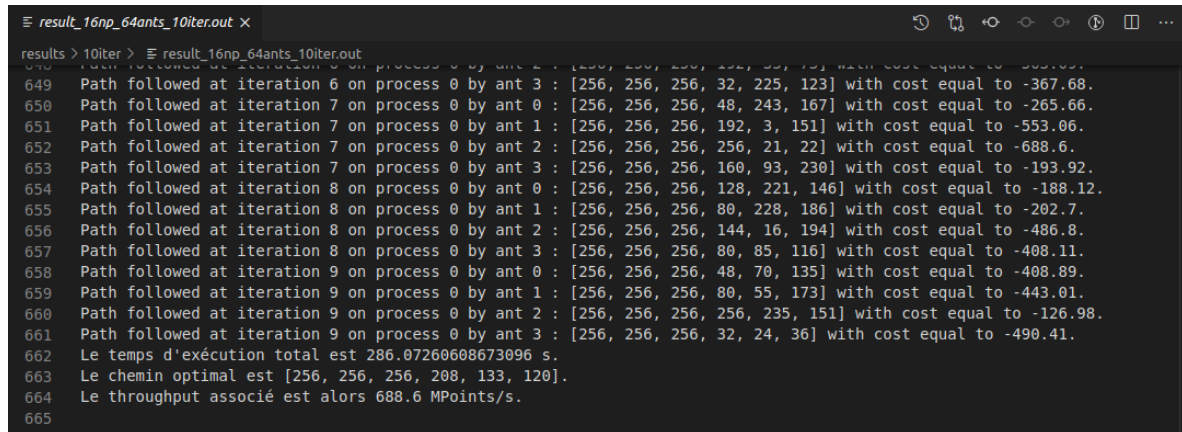
FIGURE 7 – Différentes performances sur différents processus. Le même chemin est calculé à l'itération 299 mais les différents coûts associés sont très différents.

## 5 Résultats préliminaires

### 5.1 Données brutes

Les résultats de notre code sont retournés sous forme de deux fichiers textes (voir figure 8) : un .txt contenant le meilleur coût à la fin de chaque itération et un .out contenant la liste de tous les chemins suivis par toutes les fourmis ainsi que leurs coûts associés. Le coût optimal et le chemin associé est aussi affiché dans le .out. Des scripts python sont alors utilisés pour en extraire les données.

Notre code python utilise de l'aléatoire sur chaque process. Nous n'avons pas fixé la seed utilisée par la bibliothèque random, donc les batches ne sont pas reproductibles. Si nous avions fixés les seeds, il aurait fallu en choisir une différente par process pour éviter que les process soient juste des doublons les uns des autres dans leurs choix de chemins pour les fourmis.



```
result_16np_64ants_10iter.out
results > 10iter > result_16np_64ants_10iter.out
649 Path followed at iteration 6 on process 0 by ant 3 : [256, 256, 256, 32, 225, 123] with cost equal to -367.68.
650 Path followed at iteration 7 on process 0 by ant 0 : [256, 256, 256, 48, 243, 167] with cost equal to -265.66.
651 Path followed at iteration 7 on process 0 by ant 1 : [256, 256, 256, 192, 3, 151] with cost equal to -553.06.
652 Path followed at iteration 7 on process 0 by ant 2 : [256, 256, 256, 256, 21, 22] with cost equal to -688.6.
653 Path followed at iteration 7 on process 0 by ant 3 : [256, 256, 256, 160, 93, 230] with cost equal to -193.92.
654 Path followed at iteration 8 on process 0 by ant 0 : [256, 256, 256, 128, 221, 146] with cost equal to -188.12.
655 Path followed at iteration 8 on process 0 by ant 1 : [256, 256, 256, 80, 228, 186] with cost equal to -202.7.
656 Path followed at iteration 8 on process 0 by ant 2 : [256, 256, 256, 144, 16, 194] with cost equal to -486.8.
657 Path followed at iteration 8 on process 0 by ant 3 : [256, 256, 256, 80, 85, 116] with cost equal to -408.11.
658 Path followed at iteration 9 on process 0 by ant 0 : [256, 256, 256, 48, 70, 135] with cost equal to -408.89.
659 Path followed at iteration 9 on process 0 by ant 1 : [256, 256, 256, 80, 55, 173] with cost equal to -443.01.
660 Path followed at iteration 9 on process 0 by ant 2 : [256, 256, 256, 256, 235, 151] with cost equal to -126.98.
661 Path followed at iteration 9 on process 0 by ant 3 : [256, 256, 256, 32, 24, 36] with cost equal to -490.41.
662 Le temps d'exécution total est 286.07260608673096 s.
663 Le chemin optimal est [256, 256, 256, 208, 133, 120].
664 Le throughput associé est alors 688.6 MPoints/s.
665
```

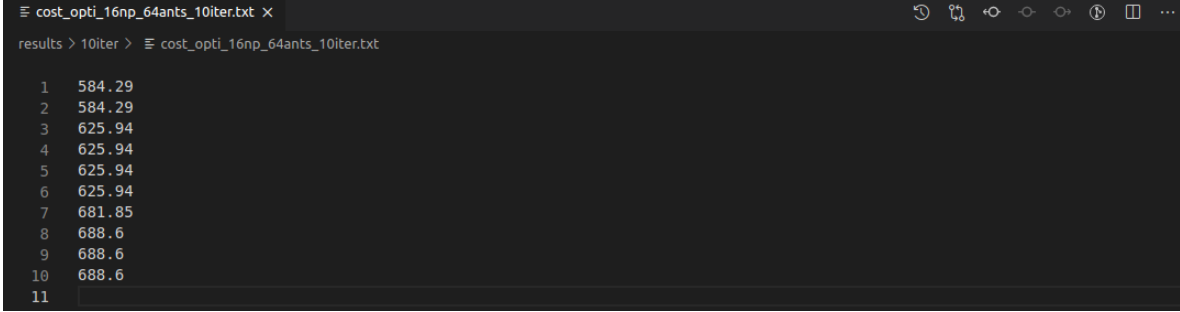


FIGURE 8 – Exemples de données retournées par notre programme

## 5.2 Comparaison de la communication collective et de la communication point à point

Comme cela a été précisé, il est nécessaire à la fin de chaque génération de fourmis que les différents processus communiquent entre eux afin de mettre à jour la matrice de phéromones. Deux méthodes sont envisageables : la communication collective (avec un Gather et un Broadcast, méthode qui sera par conséquent dénommée méthode GatherBroadcast) et la communication point à point sous forme d'anneau (avec des Sendrecv\_replace, méthode qui sera par conséquent dénommée méthode SendRcv). La solution initiale qui consiste à diffuser la matrice de phéromones mise à jour depuis le processus 0 vers tous les processus (Broadcast) peut représenter une barrière à l'extensibilité. Il nous a alors été conseillé d'implémenter un anneau (voir section 3 pour plus d'informations). Nous estimons intéressante la comparaison de la durée de communication pour ces deux méthodes. En effet, nous avons vu en cours que la méthode choisie pour le message passing et la qualité du réseau peuvent représenter des facteurs limitant important à l'extensibilité. La méthode GatherBcast risque d'être plus longue pour un nombre important de processus car elle fait intervenir des communications hors du même noeud, communications qui sont particulièrement lentes. Au contraire, pour un nombre faible de processus, on peut s'attendre à ce que l'anneau soit plus lent.

Pour réaliser cette comparaison, des barrières MPI sont ajoutées pour isoler la section du code qui met à jour la matrice de phéromones [8]. Après chaque barrière une mesure du temps est effectuée et la différence est affichée. Les mesures sont toutes effectuées pour 10 générations de 64 fourmis par processus, d'abord sur 1 processus puis sur 16 processus. Les autres paramètres étant ceux "par défaut", c'est-à-dire ceux qui sont par défaut renseignés dans le code sur gitlab. Ces paramètres sont ( $n_i = 512$ ,  $\alpha = 1$ ,  $nb\_threads = 8$ ,  $reps = 100$ ,  $\rho = 0.1$  et  $Q = 0.1$ ). Les résultats obtenus sont présentés sur la figure 9.

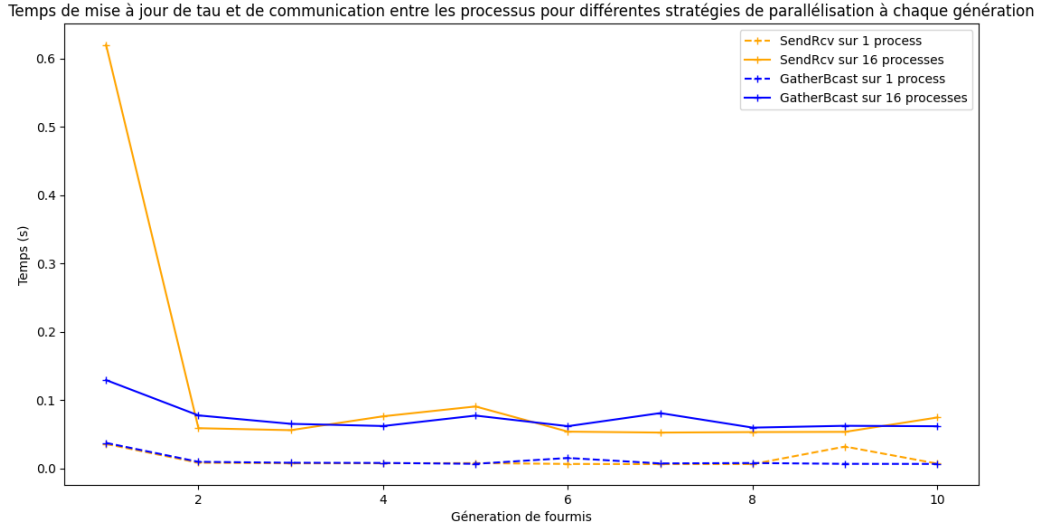


FIGURE 9 – Comparaison de la parallélisation GatherBroadcast avec la parallélisation SendReceive (en anneau).

On constate que pour les deux méthodes, quand le programme n'est pas parallélisé (c'est-à-dire qu'il est lancé sur un seul processus), la durée de mise à jour de la matrice de phéromones est la même (de l'ordre de 0.01s). Cela semble cohérent puisqu'il n'y a alors pas de transmission de message.

Lorsque nous lançons le programme sur 16 processus (c'est-à-dire sur huit noeuds), nous constatons que le temps de mise à jour de la matrice de phéromones prend plus de temps (de l'ordre de 0.05s). De plus la méthode de parallélisation (communication point à point ou communication collective) ne semble pas importer.

Nous constatons cependant que pour toutes les méthodes, lors de la première génération de fourmis, le temps de mise à jour de la matrice de phéromones prend plus de temps qu'aux autres générations. Cela peut avoir deux origines :

1. L'initialisation de matrices et objets intermédiaires à la première génération. Cela explique que même lorsque le programme n'est pas parallélisé, la première génération de fourmis met plus de temps à mettre à jour la matrice de phéromones.
2. L'établissement de la connexion entre les différents processus. On constate aussi que la durée de mise à jour de la matrice de phéromones à la première génération pour la parallélisation en anneau est beaucoup plus longue que toutes les autres initialisations. Cela explique pourquoi la solution en anneau est dix fois plus longue à la première génération pour mettre à jour la matrice de phéromones.

Pour la suite, puisque les méthodes SendRcv et GatherBcast ont des performances équivalentes, les résultats présentés seront uniquement issus de la méthode SendRcv.



### 5.3 Speed up

Un paramètre important pour juger la qualité d'un programme parallèle est le speed-up : à une taille de problème fixée, les temps d'exécution sont mesurés pour des déploiements sur un nombre croissant de processus. Pour un programme parallèle idéal, on peut s'attendre à ce que  $T(P) = T(1)/P$  avec  $T(P)$  la durée d'exécution du programme parallélisé en  $P$  Processus. Cette équation se réécrit  $\log(T(N)) = \log(T(1)) - \log(P)$ , ainsi sur un graphique en échelle logarithmique la courbe idéale est une droite décroissante.

Dans notre cas, la taille du problème est déterminée par la taille des colonies de fourmis (nombre de générations et nombre de fourmis) et les dimensions  $n_i$ . Les mesures ont été effectuées pour des colonies de 10 générations de 64 fourmis sur 1, 2, 4, 8 et 16 processus avec  $(\forall i = 1, 2, 3), n_i = 256$ . Les autres paramètres sont ceux "par défaut" ( $\alpha = 1$ , nb\_threads = 8, reps = 100,  $\rho = 0.1$  et  $Q = 0.1$ ). Le déploiement de chaque processus est toujours le même : 1 processus par socket et iso3dfd lancé sur 8 threads. Les résultats sont ceux présentés dans la figure 10.

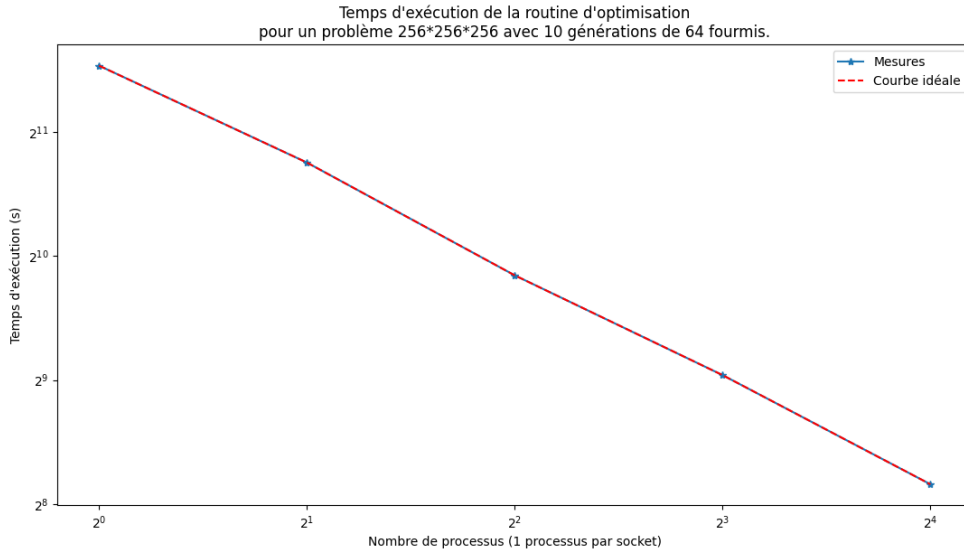


FIGURE 10 – Temps d'exécution de la routine d'optimisation pour un problème 256\*256\*256 avec 10 générations de 64 fourmis.

On constate sur cette figure que les mesures coïncident parfaitement avec la courbe idéale. On aurait pu s'attendre à ce que les mesures expérimentales soient moins bonnes que la théorie. En effet les communications inter-processus et la partie non parallélisable d'un code ont tendance à limiter le speed-up. Or nous avons vu (cf. la section 5.2) que la transmission de messages est négligeable devant toutes les autres durées (calcul de la nouvelle matrice de phéromones et exécution de iso3dfd). De plus, la partie non parallélisable du code est elle aussi négligeable (initialisation des hyperparamètres et de l'objet MPI comm). Pour la taille du problème donnée, le speed-up est donc idéal (au moins jusqu'à 16 processus).



## 5.4 Multithreading

Une des variables à définir en lançant iso3dfd est le nombre de threads que le programme va utiliser. Il est important de noter que nous avons étudié ce paramètre à la fin de notre projet, durant tout nos essais nous avons utilisé 8 threads (1 par coeur physique).

Afin de vérifier la pertinence de notre choix nous lançons iso3dfd dans la configuration suivante :  $n1 = 512$ ,  $n2 = 256$ ,  $n3 = 512$ ,  $reps = 100$ ,  $ncbx = 128$ ,  $ncby = 20$ ,  $ncbz = 9$ . Nous lançons six fois le programme pour chaque valeur du nombre de threads (variant entre 4 et 24). On obtient alors la figure 11.

Throughput moyen pour l'exécution d'iso3dfd à nombre de threads variable

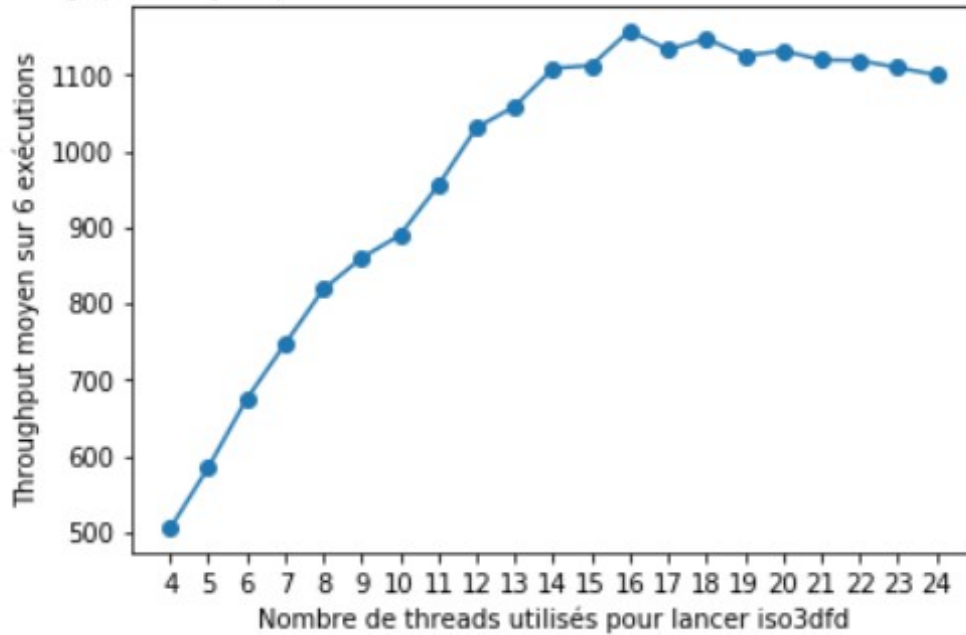


FIGURE 11 – Throughput moyen d'iso3dfd selon le nombre de threads utilisés.

Ces mesures ont été observées suite à l'exécution d'iso3dfd sous mpirun à l'aide d'un subprocess. Il est apparu que lorsque nous utilisons subprocess pour lancer un mpirun sur un socket, nos processus envahissent le socket restant de la machine utilisée. Cela explique pourquoi les performances continuent d'augmenter entre 8 et 16 threads. Au delà de 16 threads, nous commençons à utiliser des coeurs logiques ce qui diminue nos performances.

## 6 Expérimentations avec les stratégies et les hyper-paramètres

### 6.1 Colonie de fourmis classique

Nous avons lancé plusieurs batchs afin de tester différentes tailles de colonies. Nous sommes arrivés à la conclusion qu'au vu de la limite de temps imposée par sbatch (4h30), nous pouvons raisonnablement lancer notre programme avec au total 800 exécutions de iso3dfd par processus, soit 800 fourmis par processus. Cet ordre de grandeur nous permet de dimensionner la colonie en fonction du nombre de processus que l'on veut lancer. Cette règle dépend cependant des dimensions  $n_i$  utilisés par iso3dfd. En effet, plus les dimensions sont grandes, plus l'exécution est longue. La règle des 800 fourmis par processus est donc valable pour des dimensions  $n_i$  inférieures ou égales à 512.

Un exemple de configuration acceptable est l'exécution du programme sur 16 processus (nous rappelons que 1 processus = 1 socket dans notre cas) avec 400 générations de 32 fourmis. On a alors  $\frac{32}{16} \times 400 = 800$  fourmis par processus.

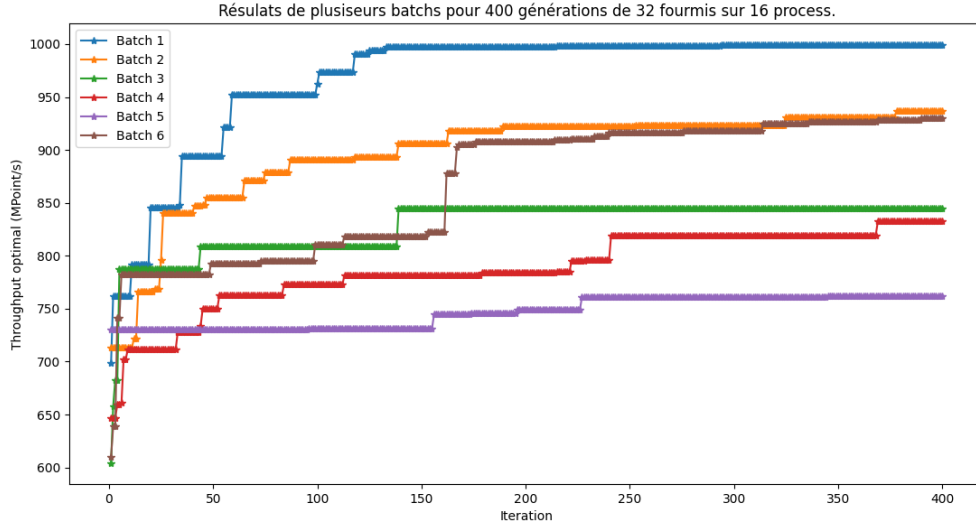


FIGURE 12 – Évolution du meilleur throughput au cours des générations de fourmis. Calculs réalisés pour  $n_1, n_2, n_3 \leq 512$  et 400 générations de 32 fourmis sur 16 processus. Attention, l'échelle des ordonnées ne commence pas à 0, l'écart relatif peut donc sembler plus important qu'il ne l'est réellement.

Une série de six batchs identiques a été lancée afin d'évaluer la stabilité de notre algorithme d'optimisation pour la stratégie de colonie de fourmis classique (résultats de la figure 12). Les paramètres de colonie sont 400 générations de 32 fourmis, les autres paramètres étant ceux "par défaut" :  $n_i = 512$ ,  $\alpha = 1$ ,  $nb\_threads = 8$ ,  $reps = 100$ ,  $\rho = 0.1$  et  $Q = 0.1$ .

On constate que pour les paramètres choisis, l'algorithme donne des résultats bien différents. Le throughput optimal varie entre 760 MPoint/s et 1000 Mpoint/s. Ces résultats correspondent, en terme de GFlops à des valeurs entre 55 GFlops et 70 GFlops. Les valeurs en GFlops sont obtenues après avoir relancé iso3dfd seul avec les chemins optimaux retournés par notre algorithme d'optimisation, les GFlops et throughput sont donc un peu meilleurs puisque iso3dfd est lancé seul, sans programme python qui consomme des ressources (cf. 1). Nous rappelons ici que les valeurs de throughput et de GFlops ont une incertitude assez grande (cf. section 4.4). Cela explique qu'un chemin retourné par ACO apparemment meilleur qu'un autre, ne l'est pas nécessairement. C'est le cas dans le tableau 1, où le chemin optimal du batch 1 est apparemment meilleur que ceux des autres batchs, mais est en réalité équivalent à ceux des batchs 2 à 4.

Les paramètres actuels ne permettent pas d'atteindre en 400 itérations le même optimal, ou au minimum des solutions équivalentes. Il faudrait donc changer les paramètres pour favoriser, au moins dans un premier temps la diversification, i.e. la découverte de nouveaux chemins. Une solution simple pour cela consiste à augmenter les phéromones posées initialement dans  $\tau$ . En effet, les mesures effectuées dans la figure 12 l'ont été pour une matrice de phéromones  $\tau$  initialisée uniformément. Tous les éléments non nuls sont initialisés à  $\tau_0 = 100 \times Q$ . Cette valeur initiale avait été choisie parce que le passage d'une fourmi sur un chemin ajoute  $Q \times \text{throughput}$  phéromones, ce qui en ordre de grandeur donne  $100 \times Q$ . Ainsi, en choisissant une valeur suffisamment grande, les fourmis ne se restreignent pas aux chemins initialement parcourus. Mais grâce à l'évaporation, les chemins non parcourus sont alors oubliés petit à petit. Cependant, comme nous l'avons constaté, les chemins parcourus ne sont pas assez divers. Nous testons cette hypothèse avec  $\tau_0 = 300 \times Q$  dans la section 6.4.

En analysant les résultats avec  $\tau_0 = 100 \times Q$  de manière plus précise (cf. tableau 1), on constate que même si les chemins optimaux sont tous différents, des similarités se dégagent. Comme cela nous a été indiqué par l'intervenant d'Intel, les throughputs sont maximaux lorsque les données sont sous forme de "frites". En effet, la dimension  $n_1$  semble toujours valoir 256 (la valeur minimale), la dimension  $n_2$  vaut 512 (la valeur maximale), cbx prend des valeurs élevées et enfin, cby et cbz prennent des valeurs plutôt faibles.

batch	$n_1$	$n_2$	$n_3$	cbx	cby	cbz	throughput ACO (MPoint/s)	throughput obtenu seul (MPoint/s)	GFlops obtenu seul
1	256	512	256	256	27	17	1000	1068	65
2	256	512	512	240	35	60	936	1062	66
3	256	512	512	224	7	28	844	1087	65
4	256	512	512	176	9	35	833	1074	66
5	256	512	512	80	10	10	762	974	59
6	512	512	256	256	25	9	930	895	55

TABLE 1 – Paramètres optimaux et throughputs associés retournés par les six batchs

Enfin, cette série de batch a aussi mis au jour un autre problème avec les résultats de notre algorithme. En effet, en analysant les chemins suivis lors des dernières itérations (voir figure 13), on constate que ces chemins ne sont pas ceux qui donnent les meilleurs throughputs. En effet, mise à part le batch 1, tous les batchs ne convergent pas même après 400 itérations. De plus, on voit sur cette figure que les chemins les plus suivis ne sont pas nécessairement ceux qui donnent le meilleur throughput (c'est le cas des batchs 3 et 5 notamment). Une solution pour régler ce problème consisterait simplement à augmenter le nombre d'itérations, malheureusement cela implique de dépasser la limite de temps autorisée par batch. Une autre solution consiste à tester le système de colonie de fourmis élitiste ou alors un Ranked Ant System. En effet, la (ou les) meilleure(s) fourmis déposent alors plus de phéromones que les fourmis moins performantes. On convergerait ainsi normalement plus vite et vers des solutions meilleures. Il faudrait cependant veiller à ce que cela ne pénalise pas trop les autres fourmis : on risquerait en intensifiant l'algorithme à outrance de ne pas explorer suffisamment de chemins et de rater des solutions bien meilleures.

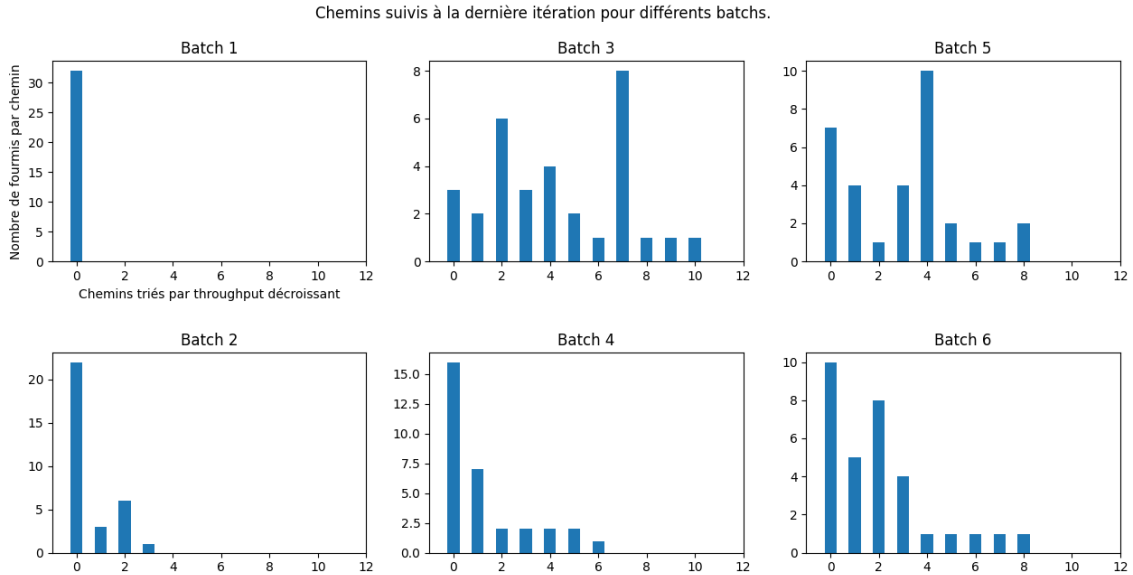


FIGURE 13 – Chemins suivis à la dernière itération pour six batchs avec  $n_1, n_2, n_3 \leq 512$  et 400 générations de 32 fourmis sur 16 processus

## 6.2 Système élitiste

Après avoir expérimenté avec les colonies de fourmis classique, nous avons testé le système de fourmis élitistes (voir la section 2.4.2 pour plus d'informations). L'objectif de cette méthode est de favoriser les chemins qui permettent d'obtenir de meilleurs gains, ainsi nous espérons converger plus rapidement vers un chemin (dans la limite des 400 itérations due au temps d'exécution limité). Nous nous attendons cependant à avoir de moins chemins puisque cette stratégie est une stratégie d'intensification : on explore moins de chemins pour converger plus vite. Les résultats sont fournis dans les

figures 14 et 15.

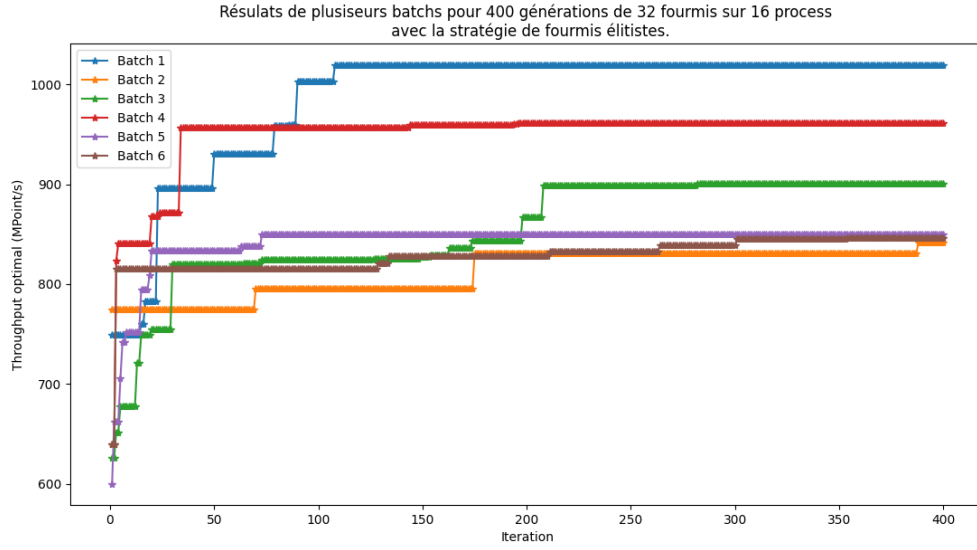


FIGURE 14 – Évolution du meilleur throughput au cours des générations de fourmis. Calculs réalisés avec la stratégie élitiste pour  $n_1, n_2, n_3 \leq 512$  et 400 générations de 32 fourmis sur 16 processus.

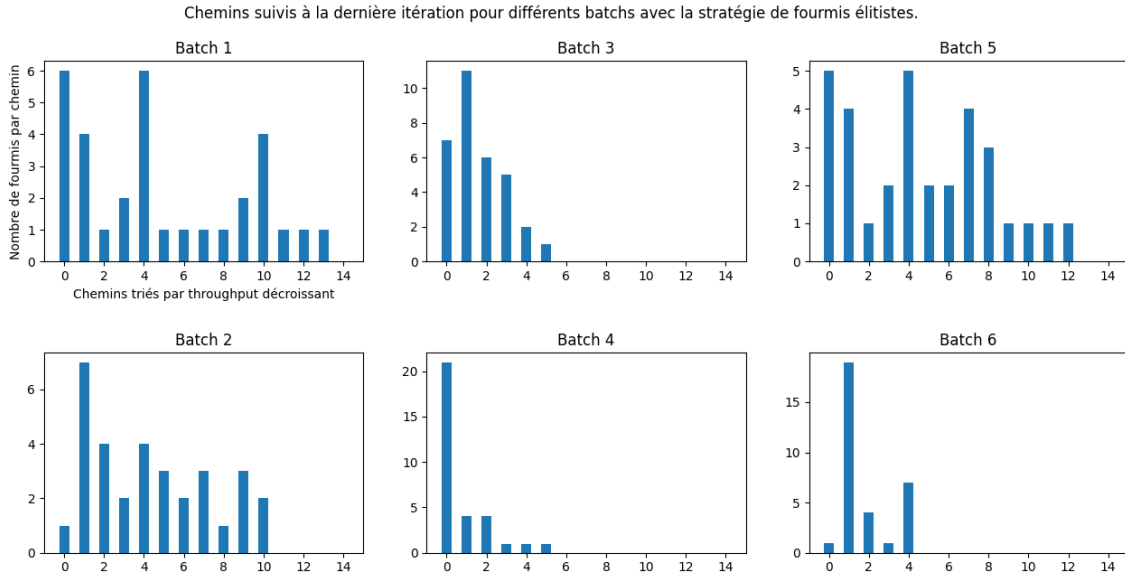


FIGURE 15 – Chemins suivis à la dernière itération avec une stratégie élitiste pour six batches avec  $n_1, n_2, n_3 \leq 512$  et 400 générations de 32 fourmis sur 16 processus.

Les résultats obtenus ne sont alors pas du tout ceux escomptés. En effet, les chemins optimaux fournis par la stratégie élitiste semblent globalement un peu meilleurs que ceux fournis par la stratégie normale lorsque l'on compare les figures 12 et 14. La stratégie élitiste fournit en moyenne au bout de 400 itérations un throughput optimal de

875 Mpoint/s contre 850 pour la stratégie classique. Au vu de la variation des résultats obtenus entre différents batchs identiques, cette différence peut être largement expliquée par l'aléatoire inhérent à l'algorithme des fourmis. De plus, à la dernière itération, l'algorithme n'a toujours pas convergé (voir figure 15). Comme précédemment, le chemin suivi par la majorité des fourmis à la dernière itération n'est pas nécessairement celui qui donne le throughput optimal.

L'utilisation de la stratégie élitiste ne représente donc pas un avantage considérable par rapport à la stratégie de base. Nous pensons que cette stratégie n'est peut-être pas adaptée à notre cas à cause de la taille des colonies. En effet dans la stratégie élitiste, à chaque génération (32 fourmis), la meilleure fourmi ajoute deux fois plus de phéromones. Cela ne représente cependant qu'une contribution mineure par rapport au 31 autres fourmis. Pour réellement améliorer la vitesse de convergence de l'algorithme, il faudrait certainement que la meilleure fourmi ajoute beaucoup plus de phéromones (trois fois ou cinq fois plus?). Dans ce cas, on peut alors espérer qu'un véritable phénomène d'intensification se produise. Pour ne pas converger vers une solution trop mauvaise, une stratégie hybride peut alors être employée : on commence avec une stratégie normale et au bout d'un certain temps (à déterminer empiriquement), on passe à la méthode élitiste.

### 6.3 Système de fourmis folles

L'utilisation de cette stratégie assure l'exploration constante des chemins pendant tout le long de l'algorithme, en évitant ainsi des phénomènes de blocage dans un optimum local qui empêche l'exploration de chemins potentiellement meilleurs. Néanmoins, cette stratégie nous oblige à lancer moins de générations que pour les autres heuristiques, puisque les fourmis folles ont plus de chances de tomber sur des chemins peu intéressants et de ralentir donc l'exécution de notre programme. Les résultats obtenus sont présentés en figure 16.

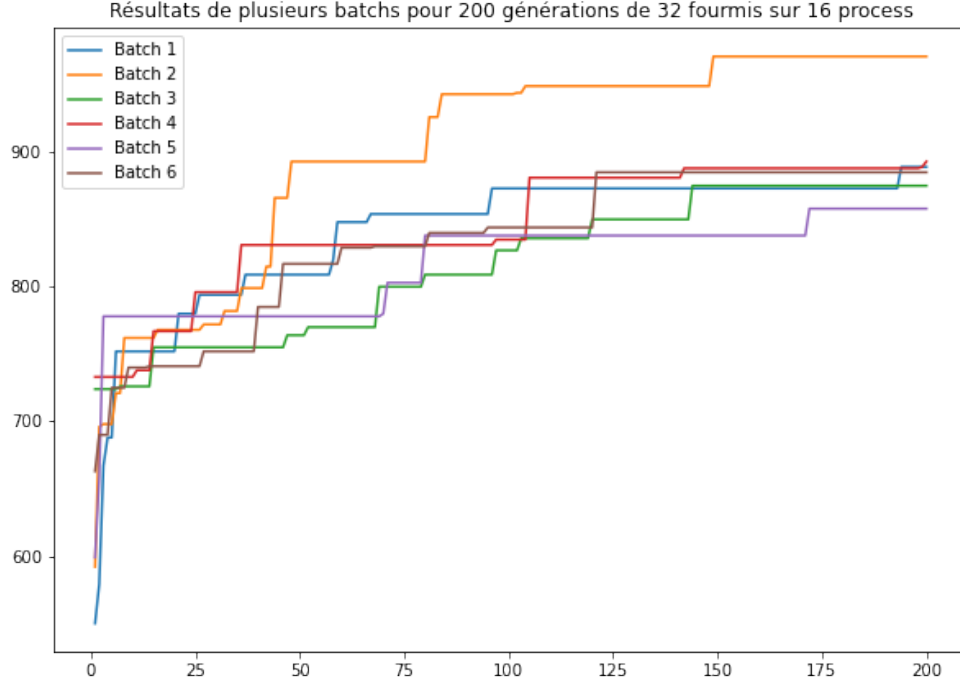


FIGURE 16 – Évolution du meilleur throughput au cours des générations de fourmis suivant la stratégie des fourmis folles. Calculs réalisés pour  $n_1, n_2, n_3 \leq 512$  et 200 générations de 32 fourmis sur 16 processus.

L’exploration rapide de nombreux chemins dès l’initialisation permet, comme on l’observe sur la figure 16, d’éviter que la colonie s’engouffre dans des chemins médiocres dès les premières itérations de l’algorithme. Les résultats sont plus consistants que pour la méthode classique : l’amélioration des scores est en moyenne plus rapide que pour le système classique. Néanmoins, lorsque des bons scores sont trouvés, l’exploration de chemins peu intéressants ralentit de manière conséquente l’exécution de notre algorithme. C’est pour quoi nous nous sommes limités à 200 générations par batch afin de ne pas dépasser le temps d’exécution maximal sur le cluster.

Pour finir, nous avons donc réfléchi à des possibles améliorations de cet algorithme, en rendant le paramètre d’exploration  $\epsilon$  variable, i.e. en utilisation une suite de réels  $(\epsilon_n)_{n \in \mathbb{N}}$  : nous imaginons que lors des premières itérations, une valeur  $\epsilon_n$  grande assure une exploration forte de notre espace, mais que cette valeur diminue pour rester pratiquement nulle lorsque les quelques centaines de générations sont atteintes. Ceci permettrait de trouver un équilibre entre l’exploration de nouveau chemins et la convergence de l’algorithme.

## 6.4 Initialisation de la matrice de phéromones

Comme cela a été proposé dans la section 6.1, nous testons une initialisation différente de la matrice de phéromones  $\tau$ . Dans les batchs précédents  $\tau$  est initialisée de façon à ce que toutes les arêtes aient une probabilité égale d'être choisies. La valeur initiale de phéromones est alors fixée à  $\tau_0 = 100 \times Q = 10$ . Nous constatons cependant que cette valeur est trop faible, ce qui limite l'exploration de nouveaux chemins (à cause de l'évaporation trop rapide). En choisissant la valeur initiale plus élevée ( $\tau_0 = 300 \times Q$ ) avec la stratégie de colonie de fourmis basique, nous obtenons les résultats de la figure 17.

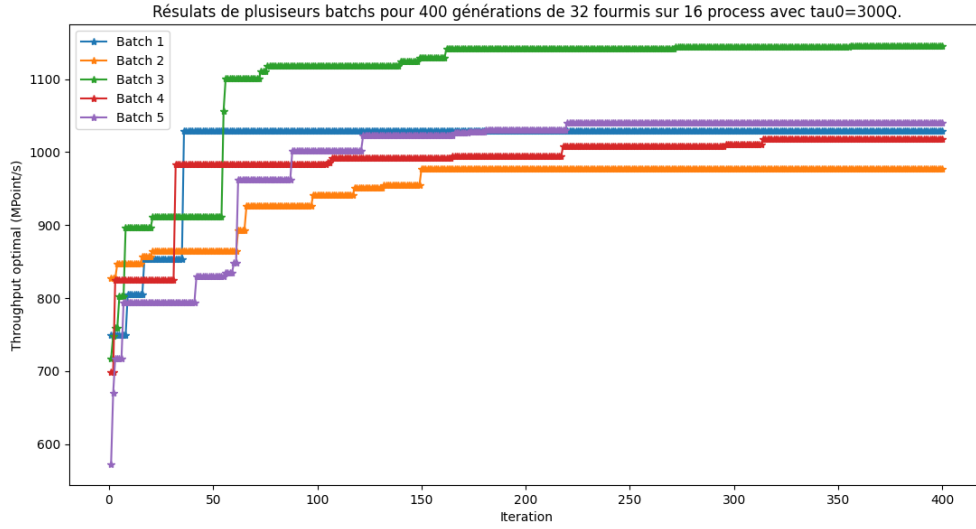


FIGURE 17 – Évolution du meilleur throughput au cours des générations de fourmis avec  $\tau_0 = 300 \times Q$ . Calculs réalisés pour  $n_1, n_2, n_3 \leq 512$  et 400 générations de 32 fourmis sur 16 processus.

En comparant les figures 12 et 17, on constate que nos prédictions sont vérifiées. En effet, les throughputs sont globalement bien plus élevés au bout de 400 générations avec une quantité initiale de phéromones plus importante.

## 6.5 Comparaison des stratégies

La figure 18 illustre les différentes vitesses de convergence des stratégies. Elle montre bien que l'utilisation de méthodes plus sophistiquées (à  $\tau_0$  fixé) permet d'obtenir de meilleurs résultats plus vite et de façon plus consistante. Il serait intéressant de comparer les performances des différentes méthodes sur un plus grand nombre d'itérations, mais la limite de temps d'utilisation du cluster ayant été presque atteinte, cette étude n'a pas pu être achevée. Toutefois, il est clair qu'une initialisation astucieuse des hyperparamètres tels que  $\tau_0$  permet d'obtenir des solutions bien meilleures.



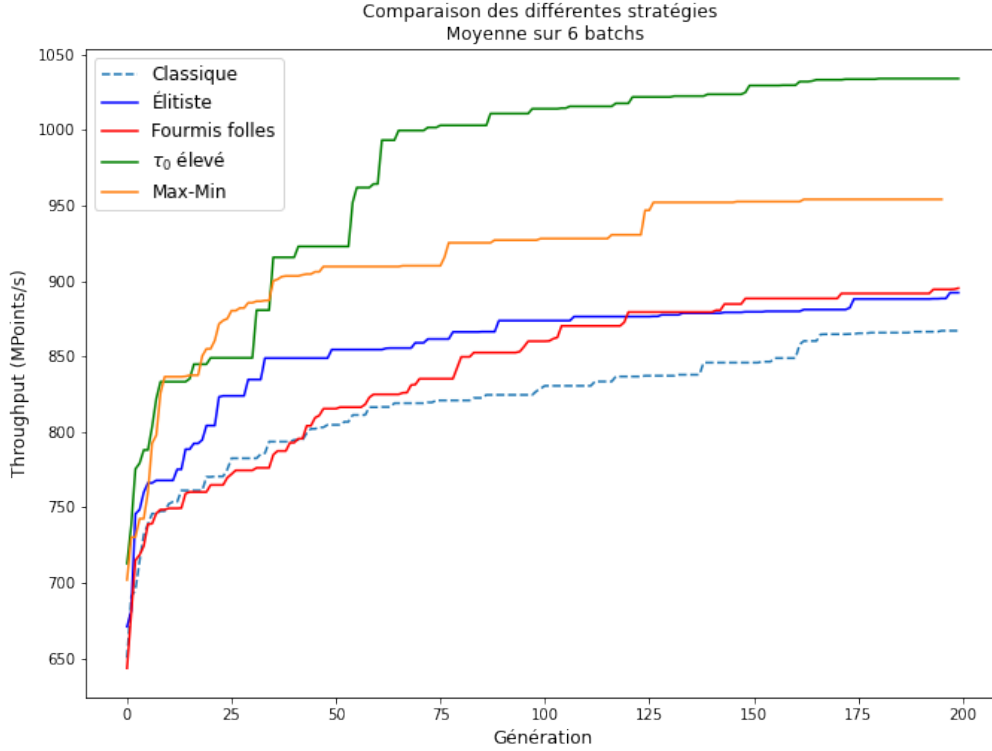


FIGURE 18 – Comparaison moyennée des performances initiales des différentes stratégies. Calculs réalisés pour  $n_1, n_2, n_3 \leq 512$  et 200 générations de 32 fourmis sur 16 processus.

Le tableau 2 recueille les solutions optimales trouvées par nos différentes stratégies ainsi que la comparaison avec un set de paramètre quelconque (donné en exemple au début du projet).

Stratégie	$n_1$	$n_2$	$n_3$	cbx	cby	cbz	throughput ACO (MPoint/s)	throughput seul (MPoint/s)	GFlops seul
Classique	256	512	256	256	27	17	999	1087	66
Élitiste	256	512	512	224	3	34	1019	1043	64
Folles	256	512	512	256	9	232	1138	1131	69
Max-Min	512	512	256	496	17	30	1024	1118	68
Baseline	256	256	256	32	32	32		545	33

TABLE 2 – Meilleurs résultats obtenus par stratégie sur 300 itérations, lancés sur 8 threads. Les chemins ont été lancés individuellement via la commande mpirun et comparés aux résultats annoncés par ACO.

Nous constatons d’abord que notre algorithme d’optimisation permet en effet de trouver des sets de paramètres bien meilleurs. En effet, nous doublons quasiment le throughput et les GFlops entre la baseline et les solutions optimales retournées par notre algorithme. Ces résultats confirment d’une part les prédictions d’Intel : les solutions optimales sont

obtenues pour des caches ayant une dimension  $cbx$  proche de  $n_1$  et des dimensions  $cby$  et  $cbz$  plus petites. D'autre part, on remarque comme précédemment une légère amélioration des résultats lorsque `iso3dfd` est lancé via la commande `mpirun` plutôt que par le biais de notre algorithme python : lorsque le programme ACO est lancé, Python et `iso3dfd` se partagent les ressources, ce qui tend à diminuer légèrement les performances.

## 7 Conclusion

Les différentes stratégies étudiées ont des performances variables : les méthodes plus complexes que nous avons choisies sont plus performantes que la stratégie classique. Cependant, toutes les stratégies convergent vers des paramètres similaires : le cache a une structure de "frite" (une des dimension est bien plus grande que les deux autres).

Pour un lancement sur 8 threads, on atteint les 1100 MPoints/s (ce qui correspond à 66 GFlops) mais nous avons vu a posteriori qu'il est plus performant d'utiliser 16 threads, ce qui permet d'atteindre 1600 MPoint/s et 100 GFlops.

Ces performances peuvent être améliorées en faisant une étude plus poussée des hyperparamètres : nous avons déjà vu qu'en augmentant les phéromones initialement présentes sur le graphe, nous obtenons de meilleures performances. En effet, en lançant la meilleure solution trouvée avec 8 threads on obtient 1145 MPoint/s et en utilisant 16 threads on obtient 1788 MPoint/s et 109 GFlops. Il serait donc intéressant de chercher à améliorer ces paramètres.

Au cours de notre travail, nous avons réalisé que notre modélisation n'est pas adaptée à l'utilisation d'heuristiques permettant d'élaguer des branches en évaluant le chemin à mi-parcours. En effet nous ne pouvons pas évaluer un score au milieu d'un chemin puisque nous devons choisir tous nos paramètres pour faire tourner `iso3dfd` et avoir le coût. Nous perdons ainsi un avantage important des algorithmes de colonie de fourmis qui permettent d'accélérer la recherche de manière intelligente.

## Références

- [1] O Pironneau B LUCQUIN. *Introduction au Calcul Scientifique*. John Wiley & Sons, 1997.
- [2] Robert E. Bixby DAVID L. APPLEGATE. « The traveling Salesman Problem : A Computational Study ». In : *Princeton University Press* (2006).
- [3] The SciPy COMMUNITY. *Sparse matrices (scipy.sparse) documentation*. 2021. URL : <https://docs.scipy.org/doc/scipy/reference/sparse.html>.
- [4] M. DORIGO, V. MANIEZZO et A. COLONI. « Ant system : optimization by a colony of cooperating agents ». In : *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1 (1996), p. 29-41. DOI : [10.1109/3477.484436](https://doi.org/10.1109/3477.484436).
- [5] Thomas STÜTZLE et Holger H. HOOS. « MAX-MIN Ant System ». In : *Future Generation Computer Systems* 16.8 (2000), p. 889-914. ISSN : 0167-739X. DOI : [https://doi.org/10.1016/S0167-739X\(00\)00043-1](https://doi.org/10.1016/S0167-739X(00)00043-1). URL : <https://www.sciencedirect.com/science/article/pii/S0167739X00000431>.
- [6] Sergio ALONSO et al. « La Metaheurística de Optimización Basada en Colonias de Hormigas : Modelos y Nuevos Enfoques ». In : (jan. 2003).
- [7] M. DORIGO et L. M. GAMBARDELLA. « Ant colony system : a cooperative learning approach to the traveling salesman problem ». In : *IEEE Transactions on Evolutionary Computation* 1.1 (1997), p. 53-66. DOI : [10.1109/4235.585892](https://doi.org/10.1109/4235.585892).
- [8] Open MPI. *MPI documentation*. 2020. URL : <https://www.open-mpi.org/doc/current/>.