



RENDU DU PROJET GRILLE ET CLUSTER

Master 1 informatique (CILS)
Conception et Intelligence des Logiciels et Systèmes

Travail

Partie Programmation

Partie Administration et clustering

Réalisé par :

Massinissa OUHEB Nabil BOUHAR Hatim ISMGH

Enseignant :

M. Patrice Lucas

2020

Table des matières

I	Programmation parallèle	2
I.1	Introduction	2
I.2	Fractale de Mandelbrot	3
I.2.1	Analyse du code	5
I.3	Programmation séquentielle	5
I.3.1	Compilation et execution	5
I.3.2	Les résultats de itertab	6
I.3.3	Tests et paramétrage	6
I.4	La librairie <i>pthread</i>	8
I.4.1	Implémentation d'une solution avec la librairie <i>pthread</i>	8
I.4.2	Compilation	10
I.4.3	Tests et paramétrage	10
I.5	<i>OPENMP</i>	10
I.5.1	Compilation	10
I.5.2	Implémentations	11
I.5.3	Tests et paramétrage	11
I.6	Message Passing Interface (MPI)	12
I.6.1	Configuration et compilation	12
	Configuration du fichier hosts	12
	Installation de MPICH	12
	Compilation	13
I.6.2	Implémentation	13
I.6.3	Tests et paramétrage	15
I.7	Statistiques	15
II	Administration système	17
II.1	Installation du système de base CentOS	17
II.1.1	Mise en place du réseau local	18
II.2	Mise en place d'un environnement minimal d'administration distante	18
II.2.1	Résolution de nom	18
II.2.2	Configuration SSH	19
II.2.3	Configuration d'un export NFS	20
II.2.4	Installation de l'outil « pdsh »	21
II.3	Outils de déploiements de cluster : SystemImager	22
II.3.1	Installation des packages systemimager	23
II.3.2	Prise d'une image golden	23
	Préparation du client à prise d'une image	23
	Récupération et configuration de l'image sur le serveur	23

II.3.3	Préparation du serveur pour déployer	24
	Installation d'un serveur dhcp et tftp	24
	Configuration de la chaine de boot pour systemima- ger : serveur dhcp et tftp	24
II.3.4	. Déploiement du noeud client	25
	Démarrage des services de déploiement	25
	Déploiement du client Cluster2	26
III	Gestion des ressources	28
III.1	SLURM	28
III.1.1	Définitions et quelques commandes de base	28
III.1.2	MUNGE	29
III.1.3	Fonctionnement	29
III.1.4	Installation et configuration	30
	Installation d'un environnement de compilation et de génération de RPM	30
	Installation de Munge	30
	Configuration et lancement de MUNGE	31
	installation de SLURM	31
	Utilisation de SLURM	32
IV	Système de fichiers parallèles	35
IV.1	LUSTRE	35
IV.1.1	Architecture LUSTRE	35
IV.1.2	Installation des paquets nécessaires	36
	Cluster0 (MDT + MGS)	36
	Cluster1 (OSS)	36
	Cluster2 (Client)	36
IV.1.3	Création des espaces nécessaires nécessaires	36
IV.1.4	Installation de la configuration Lustre	37
	Installation du serveur de méta-données et activation (Cluster0)	37
	Installation d'un serveur de donnée et activations(Cluster1)	37
	Montage sur le noeud client (Cluster2)	38
IV.1.5	Exploration des fonctionnalités annexes	38
	Création d'un deuxième OST (Cluster1)	38
	Gestion des politiques des stripping	39
	Gestion des quotas	40
	Bibliographie	43

Table des figures

I.1	Fractale de Mandelbrot	3
-----	----------------------------------	---

I.2	time	6
I.3	visualisation de itertab avec GNUPLLOT	6
I.4	La visualisation de itertab pour chaque test	7
I.5	le temps d'exécution en fonction de la résolution	15
II.1	IP adresses	18
II.2	Test de connectivité ICMP	19
II.3	Test de connectivité SSH	19
II.4	Vérification du montage nfs	21
II.5	Test pdsh	22
II.6	Test pdcp	22
II.7	Configuration DHCP	25
II.8	Recupération de l'adresse MAC de cluster2	25
II.9	Rendre le boot réseau prioritaire	26
II.10	Grub qui affiche l'image système	27
III.1	Description générale de l'architecture de SLURM	29
III.2	Fonctionnement de SLURM	30
III.3	Tansfert de la clé munge	31
III.4	Test sinfo	33
III.5	Test sinfo	33
III.6	Test de lancement de taches et manipulation	34
IV.1	Architecture du LUSTER	35

Introduction

Dans le but de s'initier à l'univers du HPC et du clustering ce projet sera divisé en deux chapitres, dans le premier chapitre, on abordera la programmation parallèle avec le langage C sur un exemple de calcul donné qui est la fractale de Mandelbrot, pour cela nous allons exploiter le parallélisme avec trois manières différentes :

- a. En utilisant les threads Posix de la librairie PThread.
- b. En utilisant OpenMP (Open Multi-Processing) en memoire partagée.
- c. En utilisant la programmation par passage de message à l'aide de la librairie MPI en mémoire distribuée.

puis évaluer les performances des differents programmes.

Dans le second chapitre nous allons étudier le monde du clustering et nous apprendrons à installer et à utiliser quelques outils permettant l'administration et la gestion des ressources dans un cluster.

Chapitre I

Programmation parallèle

I.1 Introduction

Les temps de calcul en séquentiel sont souvent trop élevés car le programme est exécuté par un et un seul processus. De nos jours, la quasi-totalité des machines possèdent des architectures parallèles (multi-cœurs). Cependant avoir un processeur multi-cœurs n'est pas suffisant pour optimiser le calcul, il faut aussi accompagner ce parallélisme hardware par un parallélisme software. Et pour ce faire, il faut savoir créer des programmes parallèles qui utilisent les ressources de la machine de manière optimale.

Avant d'entamer ce chapitre nous allons d'abord présenter la configuration de la machine sur laquelle les opérations qui suivront vont se dérouler. En effet les programmes seront exécutés sur une machine avec un OS **Linux (Debian 10.2)**.

Connaitre les caractéristiques du processeur sur Linux avec la commande suivante :

```
lscpu
```

```
Architecture:           x86_64
CPU op-mode(s):         32-bit, 64-bit
Byte Order:             Little Endian
Address sizes:          36 bits physical, 48 bits virtual
CPU(s):                 4
On-line CPU(s) list:    0-3
Thread(s) per core:     2
Core(s) per socket:     2
Socket(s):              1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  69
Model name:             Intel(R) Core(TM) i3-4030H CPU @ 1.90GHz
Stepping:               1
CPU MHz:                798.166
CPU max MHz:            1800.0000
BogoMIPS:               3791.18
Virtualization:         VT-x
RAM:                    4 Gb
```

Essentiellement nous avons juste besoin de savoir :

- Socket(s) : 1, donc un seul noeud processeur
- Core(s) per socket : 2, le cpu contient 2 coeurs physiques.
- Thread(s) per core : 2, chaque coeur contient 2 coeurs logique donc au total 4 coeurs logique.
- CPU MHz : 798.166, c'est la fréquence de chaque coeur physique en méga Hertz.

I.2 Fractale de Mandelbrot

Parmi les problèmes mathématiques qui engendrent une charge de calculs importante sur le processeur, nous avons l'ensemble de Mandelbrot. Ce dernier nous permettra la réalisation d'une étude comparative entre les différentes solutions proposées.

La fractale de Mandelbrot est constituée d'un ensemble des points $M(x, y)$ dont le nombre complexe $C = x + yi$ en est l'affixe, tel que la suite définie par :

$$\begin{cases} Z_0 &= 0 \\ Z_{n+1} &= Z_n^2 + C \end{cases}$$

Le point $M(x, y)$ lorsque $|Z_n|$ deviens supérieure strictement à 2 partir d'une itération numéro n .

La génération de cette fractale nécessite beaucoup de calculs mathématiques, notamment pour une résolution et un nombre d'itérations (nombre d'itérations maximal pour le test de la convergence) élevés.

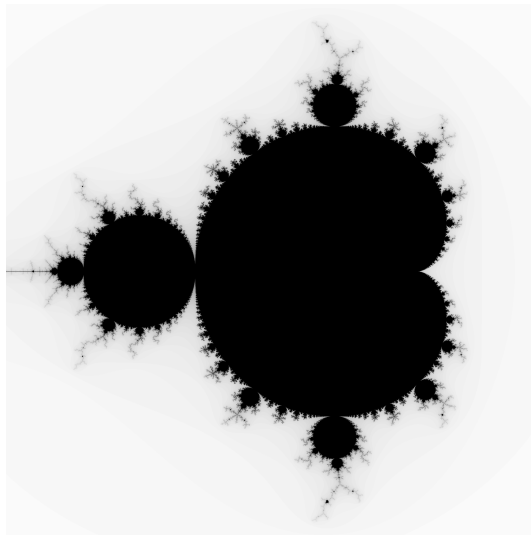


FIGURE I.1 – Fractale de Mandelbrot

Le code C fournit en dessous permet l'interprétation de l'ensemble de Mandelbrot, et avant de passer à la section suivante nous allons analyser et comprendre le fonctionnement général de ce programme.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5  #include <math.h>
6
7  #define OUTFILE "mandelbrot.out"
8
9  double XMIN = -2;
10 double YMIN = -2;
11 double XMAX = 2;
12 double YMAX = 2;
13 double RESOLUTION = 0.01;
14 int NITERMAX = 400;
15
16
17 int main(int argc, char* argv[]){
18
19     int* itertab; //valeurs de n lorsque Zn diverge
20     int nbpixelx;
21     int nbpixely;
22     int xpixel = 0, ypixel = 0;
23     FILE * file;
24
25     //calcul du nombre de pixel
26     nbpixelx = ceil((XMAX-XMIN)/RESOLUTION);
27     nbpixely = ceil((YMAX-YMIN)/RESOLUTION);
28
29     //allocation du tableau de pixel
30     if ((itertab = malloc(sizeof(int)*nbpixelx*nbpixely))==NULL)
31     {
32         printf("allocation error");
33         return 1;
34     }
35
36     //calcul des points
37     for (xpixel = 0; xpixel < nbpixelx; xpixel++){
38         for (ypixel = 0; ypixel < nbpixely; ypixel++){
39
40             double xinit = XMIN + xpixel * RESOLUTION;
41             double yinit = YMIN + ypixel * RESOLUTION;
42             double x = xinit;
43             double y = yinit;
44             int iter = 0;
45
46             for (iter = 0; iter < NITERMAX; iter++){
47                 double prevy = y;
48                 double prevx = x;
49                 if ((x*x + y*y) > 4){
50                     break;
51                 }
52                 x = prevx*prevx - prevy*prevy + xinit;
53                 y = 2*prevx*prevy + yinit;
54             }
55
56             itertab[xpixel*nbpixely+ypixel] = iter;
57         }
58     }
59
60     //output resultat gnuplot
61     if ((file=fopen(OUTFILE, "w"))==NULL){
62         printf("file open error");
63         return 1;
64     }
65     for (xpixel = 0; xpixel < nbpixelx; xpixel++){
66         for (ypixel = 0; ypixel < nbpixely; ypixel++){
67             double x = XMIN + xpixel * RESOLUTION;
68             double y = YMIN + ypixel * RESOLUTION;

```



```
69         fprintf(file,"%f %f %d \n",
70             x,y,itertab[xpixel*nbpxely+ypixel]);
71     }
72     fprintf(file,"\n");
73 }
74 fclose(file);
75
76 return 0;
77 }
```

I.2.1 Analyse du code

- Pour déterminer l'ensemble de Mandelbrot, un calcul est effectué sur chaque pixel.
- Le programme vérifie pour chaque point sa divergence.
- La résolution **RESOLUTION** est un paramètre qui permet de déterminer le nombre de points à calculer.
- On considérera qu'une suite diverge lorsqu'on itère le calcul jusqu'à un certain seuil fixé par l'utilisateur NITERMAX ou lorsque $|Z_n| > 2$.
- Le tableau **itertab** contient le nombre d'itération de chaque point avant sa divergence et on va utiliser le logiciel **gnuplot** qui va nous générer une image qui représente le tableau pour faciliter la visualisation des résultats, La couleur d'un pixel est définie par le nombre d'itérations réaliser avant la divergence.

I.3 Programmation séquentielle

Dans cette section on va exécuter le code définit dans la section précédente et on va étudier le temps d'exécution pour différents paramétrage de résolution et de nombre d'itération maximale.

I.3.1 Compilation et execution

Sous linux on peut compiler les programmes écrits en C en utilisant le compilateur GCC.

```
gcc préambule_séquentiel.c -o préambule_séquentiel -lm
```

On a utiliser la commande **TIME** de Linux pour mesurer le temps d'exécution du programme, *real* est le temps réel écoulé depuis l'exécution de la commande jusqu'à l'affichage du résultat et *user*, *sys* retournes le temps pris par le CPU pour exécuter le programme en *user-mode* et *kernel-mode* or le programme s'exécute en *user-mode* alors on va se baser sur *user* pour mesurer le temps d'exécution.

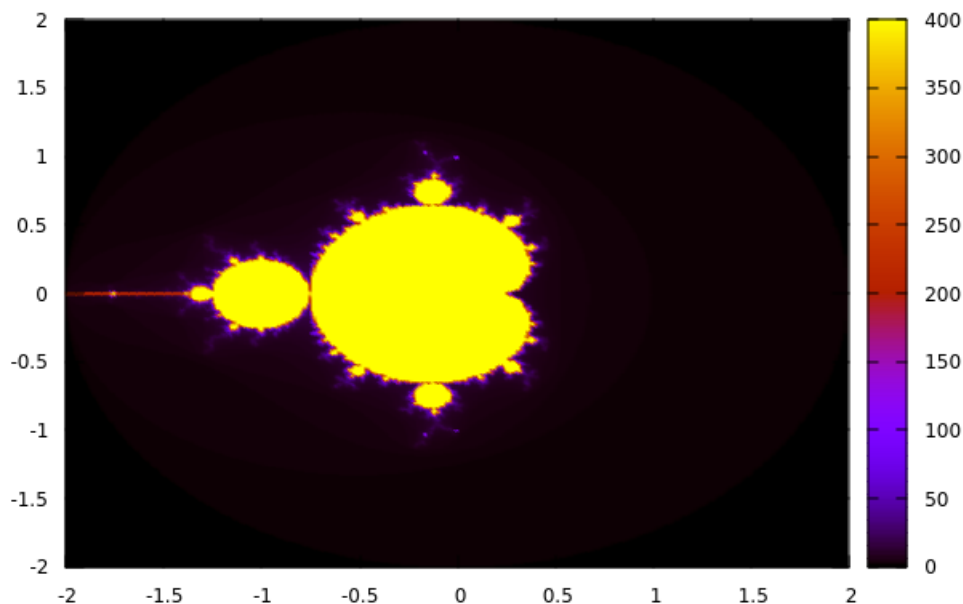
```
time
```

```
hatim@debian:~/Desktop/4-Grigra/td-1/code$ time ./préambule_séquentiel  
  
real    0m0.241s  
user    0m0.241s  
sys      0m0.000s
```

FIGURE I.2 – time

I.3.2 Les résultats de `itertab`

Après visualisation du tableau `itertab` en utilisant *GNUPLLOT* on a obtenue l'image suivante :

FIGURE I.3 – visualisation de `itertab` avec *GNUPLLOT*

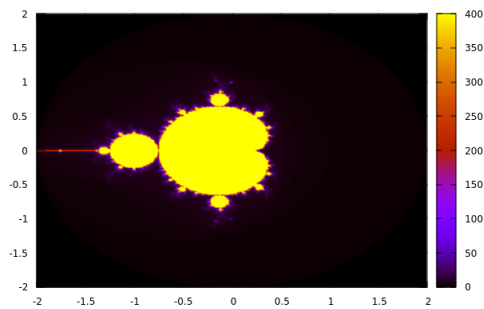
Les pixels qui ont une couleur jaune ont divergée à partir de l'itération numéro 400 par contre qui ont une couleur noir ont divergé dès les premières itérations comme c'est affiché dans le barème à droite de l'image. Cette image va nous aider à vérifier notre code lorsqu'on passe à la partie de la parallélisation en comparant l'image qu'on a obtenue avec cette image.

I.3.3 Tests et paramétrage

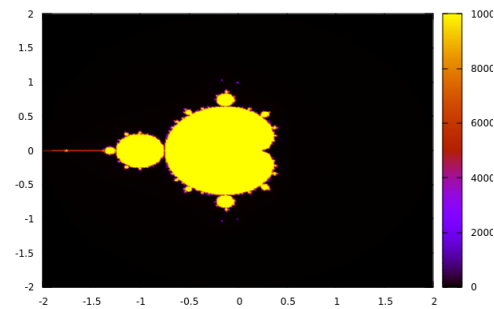
Après l'exécution du programme avec différents paramètres pour **RESOLUTION** et **NITERMAX** on a obtenu les résultats affichées dans le tableau suivant :

Tests	Paramètres		Le temps d'exécution	
n	RESOLUTION	NITERMAX	<i>real</i>	<i>user</i>
A	0.01	400	0.241s	0.241s
B	0.01	10000	2.186s	2.156s
C	0.001	2000	54.434s	53.598s
D	0.0006	2000	2m 29.558s	2m 27.839s

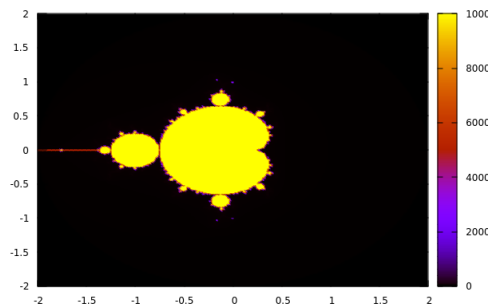
On peut remarquer d'après le tableau que le temps d'exécution du programme augmente en fonction des paramètres, car si on augmente le nombre maximale d'itération alors le nombre d'itération des points représentés par les pixels jaune va aussi augmenter car il n'ont pas divergé à partir de l'itération numéro 400, ainsi que lorsqu'on a diminuer la résolution le temps d'exécution a augmenté car on va calculer la divergence d'un ensemble de points plus grand.



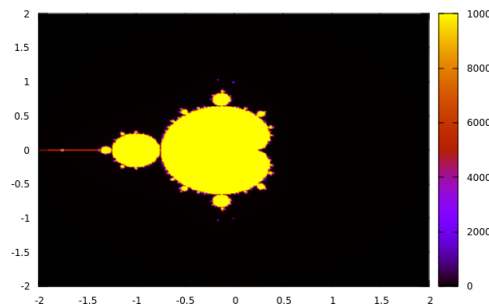
(A)



(B)



(C)



(D)

FIGURE I.4 – La visualisation de *itertab* pour chaque test

On peut constater d'après la figure I.4 que l'image qu'on obtient va pas changer beaucoup en fonction des paramètres, on peut aussi remarquer que l'image est symétrique par rapport à l'axe des abscisses, donc il suffit de calculer la divergence des points qui en une valeur d'ordonnée supérieure ou

égale à 0, mais on va pas utiliser cette méthode mathématique pour optimiser le temps d'exécution du programme car notre but est d'optimiser le temps d'exécution en parallélisant les instructions.

I.4 La librairie *pthread*

La bibliothèque *pthread* définit des fonctions, des structures et des constantes qui vont permettre de paralléliser l'exécution des instructions en créant des *threads*.

La norme POSIX fournit un ensemble de primitives permettant de réaliser des processus légers (Threads). Un thread est une portion de code (fonction) qui se déroule en parallèle au thread principal (main) et qui partage son contexte (espace mémoire). Les threads permettent d'effectuer plusieurs tâches en parallèle au sein d'un même processus. L'avantage des threads par rapport aux processus c'est leur temps de création : environ 1 ms contre 50 ms pour un processus (fork).

I.4.1 Implémentation d'une solution avec la librairie *pthread*

Nous avons utilisé dans cette partie deux types de parallélisations : Selon l'axe des abscisses puis selon l'axe des ordonnées. Pour assurer la portabilité et l'optimalité de la solution proposée, nous avons choisi de donner le nombre de threads à créer comme paramètre au programme à l'exécution.

On a utilisé la structure de données suivante pour le passage d'arguments au thread :

```
1 struct args {
2     int xdebut,ydebut,xfin,yfin ;
3     int * itertab ;
4 };
5 typedef struct args args ;
```

Et la fonction suivante pour le corps du thread :

```
1 void * calcul_point (void * arg){
2     args * param ;
3     param = (args*) arg ;
4     int xdebut = param->xdebut;
5     int ydebut = param->ydebut ;
6     int xfin = param->xfin ;
7     int yfin = param->yfin ;
8     for(int xpixel=xdebut;xpixel<xfin;xpixel++)
9         for(int ypixel=ydebut;ypixel<yfin;ypixel++) {
10             double xinit = XMIN + xpixel * RESOLUTION;
11             double yinit = YMIN + ypixel * RESOLUTION;
12             double x=xinit;
13             double y=yinit;
14             int iter=0;
15             for(iter=0;iter<NITERMAX;iter++) {
16                 double prevy=y,prevx=x;
17                 if( (x*x + y*y) > 4 )
18                     break;
```

```

19         x = prevx*prevx - prevy*prevy + xinit;
20         y = 2*prevx*prevy + yinit;
21     }
22     itertab[xpixel*nbpixelx+ypixel]=iter;
23 }
24 pthread_exit(NULL);

```

Comme indiqué auparavant, nous avons réalisé deux types de parallélisation

PARALLÉLISATION SELON L'AXE DES ABSCISSES :

Ici la variable « nbthread » représente le nombre de threads à créer, elle est donnée en paramètre au programme lors de l'exécution. Par exemple :

```
1 time ./mandelbrot_parallele_y -n
```

pour une exécution avec « n » threads. L'axe des abscisses est divisé en « n » parties égales, chacune est calculée par un thread. L'axe des ordonnées quant à lui n'est pas divisé et est le même pour chaque thread.

```

1  int nbthread = atoi(argv[1]);
2  int dep = 0;
3  int pas = ceil(nbpixelx/nbthread);
4  for (t=0;t<nbthread;t++){
5      arg[t].xdebut=dep ;
6      arg[t].xfin=dep+pas;
7      dep=arg[t].xfin ;
8      arg[t].ydebut=0 ;
9      // pas de parallélisation selon l'axe des y
10     arg[t].yfin=nbpixelx;
11     rc = pthread_create(&threads[t], NULL,calcul_point ,
12         (void *)&arg[t]);
13     if (rc) exit(-1);
14 }

```

PARALLÉLISATION SELON L'AXE DES ORDONNÉES : Ici la variable « nbthread » représente le nombre de thread à créer, elle est donnée en paramètre au programme lors de l'exécution. Par exemple :

```
time ./mandelbrot_parallele_x -m
```

pour une exécution avec « m » threads. L'axe des ordonnées est divisé en « m » parties égales, chacune est calculée par un thread. L'axe des abscisses quant à lui n'est pas divisé et est le même pour chaque thread.

```

1  int nbthread = atoi(argv[1]);
2  int dep = 0;
3  int pas = ceil(nbpixelx/nbthread);
4  for (t=0;t<nbthread;t++){
5      arg[t].ydebut=dep ;
6      arg[t].yfin=dep+pas;
7      dep=arg[t].yfin ;
8      arg[t].xdebut=0 ;
9      arg[t].xfin=nbpixelx;
10     printf("creation du thread : %d \n",t);
11     rc = pthread_create(&threads[t], NULL,calcul_point ,

```

```

12         (void *)&arg[t]);
13         if (rc) exit(-1);
14     }

```

TERMINAISON DES THREADS : Pour que les threads se terminent correctement il faut que le programme principal les attende. Les threads dépendent du processus qui les a créés et se terminent si ce dernier finit son exécution.

```

1  for(t=0; t<nbthread; t++){
2      void * ret_ptr;
3      pthread_join( threads[t], &ret_ptr );
4  }

```

I.4.2 Compilation

Pour utiliser la bibliothèque *pthread* il faudra inclure le fichier *pthread* et ajouter l'option *-lpthread*.

```
gcc thread_posix.c -lpthread -o thread_posix -lm
```

I.4.3 Tests et paramétrage

Tests <i>n</i>	Paramètres			Le temps d'exécution	
	THREADS	RESOLUTION	NITERMAX	<i>real</i>	<i>user</i>
<i>A</i> ₂	2	0.01	400	0.253s	0.275s
<i>A</i> ₄	4	0.01	400	0.185s	0.226s
<i>B</i> ₂	2	0.01	10000	1.611s	2.164s
<i>B</i> ₄	4	0.01	10000	1.472s	2.280s
<i>C</i> ₂	2	0.001	2000	42.524s	53.449s
<i>C</i> ₄	4	0.001	2000	39.809s	54.902s
<i>D</i> ₂	2	0.0006	2000	1m 59.404s	2m 29.099s
<i>D</i> ₄	4	0.0006	2000	1m 49.518s	2m 34.072s

I.5 OPENMP

La directive **OPENMP** offre une interface de programmation pour paralléliser le calcul sur une mémoire partagées. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement. *OPENMP* utilise les *threads* comme *pthread* mais il facilite la parallélisation et permet de réaliser la même tâche avec moins de lignes de codes.

I.5.1 Compilation

Pour compiler le code on ajoute l'option *-fopenmp* pour obliger le compilateur à exécuter les directives et dans le cas d'utilisation d'une fonction de l'interface il faudra inclure la bibliothèque dans **omp** le programme.

```
gcc -fopenmp openmp.c -o openmp
```

Pour modifier le nombre de *threads* qu'on veut créer, on utilise la variable d'environnement *OMP_NUM_THREADS*.

```
set OMP_NUM_THREADS= <nombre de threads>
```

Ou bien :

```
export OMP_NUM_THREADS= <nombre de threads>
```

I.5.2 Implémentations

On va utiliser le même raisonnement de la section précédente, le calcul sera partagé en fonction du nombre des *threads* créés et selon l'axe des abscisses.

```

1  /* initialisation (ne va pas changer)*/
2  /*calcul des points*/
3  #pragma omp parallel
4  { // fonction de thread
5      int xpixel, ypixel;
6      #pragma omp parallel for
7      for(xpixel=0; xpixel<nbpixelx; xpixel++)
8          for(ypixel=0; ypixel<nbpixely; ypixel++) {
9              double xinit = XMIN + xpixel * RESOLUTION;
10             double yinit = YMIN + ypixel * RESOLUTION;
11             double x=xinit;
12             double y=yinit;
13             int iter=0;
14             for(iter=0; iter<NITERMAX; iter++) {
15                 double prevy=y, prevx=x;
16                 if( (x*x + y*y) > 4 )
17                     break;
18                 x= prevx*prevx - prevy*prevy + xinit;
19                 y= 2*prevx*prevy + yinit;
20             }
21             itertab[xpixel*nbpixely+ypixel]=iter;
22             itertab[xpixel*nbpixely-ypixel+nbpixely]=iter;
23         }
24  /*output des resultats compatible gnuplot (ne va pas changer)*/

```

On peut remarquer qu'on a ajouté juste 4 lignes de code au programme séquentiel pour arriver aux résultats qu'on a pu obtenir en utilisant la bibliothèque *pthread*.

I.5.3 Tests et paramétrage

On peut remarquer d'après le tableau que le temps d'exécution stagne ou plutôt augmente lorsqu'on dépasse le nombre des coeurs physique de la machine, donc on peut pas optimiser le temps d'exécution encore plus car le nombre de *threads* que la machine peut exécuter au même temps est limité au nombre de ses coeurs physique qui est 2 dans notre cas.

Tests	Paramètres			Le temps d'exécution	
<i>n</i>	THREADS	RESOLUTION	NITERMAX	<i>real</i>	<i>user</i>
A_2	2	0.01	400	0.218s	0.307s
A_4	4	0.01	400	0.252s	0.605s
B_2	2	0.01	10000	2.162s	4.188s
B_4	4	0.01	10000	2.689s	10.134s
C_2	2	0.001	2000	54.040s	1m 34.476s
C_4	4	0.001	2000	1m 4.621s	3m 35.010s
D_2	2	0.0006	2000	2m 01.185s	4m 24.609s
D_4	4	0.0006	2000	3m 6.592s	9m 53.087s

I.6 Message Passing Interface (MPI)

On a vu dans les deux sectionnes précédentes comment paralléliser le code en utilisant les *threads* qui partagent la mémoire, par contre la plupart des super-calculateurs sont des machines à mémoire distribuée, donc ça sera intéressant d'exécuter un programme sur des systèmes à mémoire distribuée. **MPI** est une bibliothèque permettant de coordonner des processus en utilisant le paradigme de l'échange de messages. On va utiliser cette bibliothèque pour paralléliser le calcul de la **FRACTALE DE MANDELBROT** sur des **clusters** (noeuds) d'ordinateurs hétérogènes à mémoire distribuée.

I.6.1 Configuration et compilation

Malgré que **MPI** est très puissant par rapport au autres outils de parallélisation car les applications qu'on peut réaliser sont beaucoup plus intéressantes, mais la complexité de sa configuration augmente en fonction du nombre de noeuds.

Nous avons décidé d'attendre la fin du déploiement de notre cluster sur VMWare pour pouvoir continuer cette partie, puisque l'accès aux machines du département n'est plus possible [2].

Configuration du fichier hosts

```
$ cat /etc/hosts
127.0.0.1 localhost
192.168.10.1 cluster0 #maitre
192.168.10.2 cluster1 #esclave
192.168.10.3 cluster2 #esclave
192.168.10.4 cluster3 #esclave
#cluster3 est une doublure pour cluster2 même methode de
#déploiement
```

Installation de MPICH

```
cd /nfs_export_cluster_dir
curl http://www.mpich.org/static/downloads/3.1.4
/mpich-3.1.4.tar.gz
```



```
tar -xvf mpich-3.1.4.tar.gz #decompression
cd mpich-3.1.4
./configure prefix=/nfs_export_cluster_dir
/MPITCH/disablefortran
#####
make
make install
```

Ajouter dans chacun des 4 noeuds les deux variables d'environnement :

```
nano $HOME/.bahsrc
#####
export PATH=/nfs_export_cluster_dir(OU nfs_cluster_dir)
/MPITCH/bin:$PATH
export LD_LIBRARY_PATH="/export_nfs_dir/MPITCH/lib:
$LD_LIBRARY_PATH"
source ~/.bashrc
```

Compilation

Après avoir installé et configuré **MPICH** nous allons procéder à la compilation de notre code.

Exécuter la commande suivante dans le dossier code partagée entre les noeuds qui contient le programme *mpi.c*.

```
$ mpicc mpi.c -lm -o mpi
```

Pour l'exécution il faudra spécifier le nombre processus à créer par l'option *-np* et le nombre des noeuds à utiliser par l'option *-host*.

```
$ mpirun -np 2 --host cluster0,cluster1 ./mpi
#ou
$ mpirun -np 4 --host cluster0,cluster1,cluster2,cluster3 ./mpi
```

I.6.2 Implémentation

On va découper le travail avec la même manière, celons l'axe des abscisses et en fonctionne du nombre des processus. Chaque noeud va calculer la divergence d'un ensemble des points, l'enregistrer dans le tableau ITER-TABP et l'envoyer au noeud maître qui va faire la restitution dans ITERTAB.

```
1  #include <stdio.h>
2  #include "mpi.h"
3  #include <stdlib.h>
4  #include <math.h>
5  #include <string.h>
6  #include <errno.h>
7  #define OUTFILE "mandelbrot.out"
8  double XMIN=-2;
9  double YMIN=-2;
10 double XMAX=2;
11 double YMAX=2;
12 double RESOLUTION=0.01;
13 int NITERMAX=400;
14 int main( int argc, char **argv )
15 {
16     /*l'identifieur de chaque processus et la taille
17     de tous les processus*/
```

```

18     int rank, size;
19     // le fichier de sortie
20     FILE * file;
21     MPI_Init( &argc, &argv );
22     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
23     MPI_Comm_size( MPI_COMM_WORLD, &size );
24     /*calcul du nombre de pixel*/
25     int nbpixelx = ceil((XMAX - XMIN) / RESOLUTION);
26     int nbpixely = ceil((YMAX - YMIN) / RESOLUTION);
27     /*découpage de l'intervale des abscises en des
28     sous intervalles */
29     int etape = nbpixelx/size;
30     //le tableau que doit remplir chaque processus
31     int *itertabp = malloc(sizeof(int)*nbpixely*etape);
32     /* le calcule de la fractale */
33     for(int xpixel= 0; xpixel<etape; xpixel++)
34         for(int ypixel=0;ypixel<=nbpixely;ypixel++) {
35             double xinit = XMIN+(xpixel+rank*etape)*RESOLUTION;
36             double yinit = YMIN + ypixel * RESOLUTION;
37             double x=xinit;
38             double y=yinit;
39             int iter=0;
40             for(iter=0;iter<NITERMAX;iter++) {
41                 double prevy=y,prevx=x;
42                 if( (x*x + y*y) > 4 )
43                     break;
44                 x= prevx*prevx - prevy*prevy + xinit;
45                 y= 2*prevx*prevy + yinit;
46             }
47             itertabp[xpixel*nbpixely+ypixel]=iter;}
48     /*le tableau qui contient le resultat finale après
49     rassembler tous les tableaux */
50     int *itertab;
51     if(rank==0) itertab=malloc(sizeof(int)*nbpixelx*nbpixely);
52     else itertab = NULL;
53
54     // rassembleage des résultats en utilisant mpi_gather
55     MPI_Gather(itertabp, nbpixely*etape, MPI_INT, itertab,
56     nbpixely*etape, MPI_INT, 0, MPI_COMM_WORLD);
57
58     if(rank == 0)
59     { //remplissage du fichier de sortie par le processus root
60         /*output des resultats compatible gnuplot*/
61         if( (file=fopen(OUTFILE,"w")) == NULL ) {
62             printf("Erreur à l'ouverture du fichier de sortie");
63             printf(": errno %d (%s) .\n",errno,strerror(errno));
64             return EXIT_FAILURE;
65         }
66         int xpixel=0,ypixel=0;
67         for(xpixel=0;xpixel<nbpixelx;xpixel++) {
68             for(ypixel=0;ypixel<nbpixely;ypixel++) {
69                 double x = XMIN + xpixel * RESOLUTION;
70                 double y = YMIN + ypixel * RESOLUTION;
71                 fprintf(file,"%f %f %d\n", x, y,
72                 itertab[xpixel*nbpixely+ypixel]);
73             }
74             fprintf(file,"\n");
75         }
76         fclose(file);
77     }
78     MPI_Finalize();

```

```

79     return 0;
80 }

```

I.6.3 Tests et paramétrage

Pour les tests on va fixer le nombre des procès aux nombre des noeuds du système pour garder le même modèle de test.

Tests	Paramètres			Le temps d'exécution	
n	NOEUDS	RESOLUTION	NITERMAX	<i>real</i>	<i>user</i>
A_2	2	0.01	400	1.014s	0.402s
A_4	4	0.01	400	1.036s	0.637s
B_2	2	0.01	10000	2.587s	3.531s
B_4	4	0.01	10000	2.347s	5.886s
C_2	2	0.001	2000	46.402s	1m 18.727s
C_4	4	0.001	2000	42.669s	2m 7.571s
D_2	2	0.0006	2000	2m 11.100s	3m 44.333s
D_4	4	0.0006	2000	1m55.868s	5m53.417s

I.7 Statistiques

On va visualiser les tests en utilisant des courbes qui représente le temps d'exécution en fonctionne de la résolution pour synthétiser l'efficacité des outils de parallélisation.

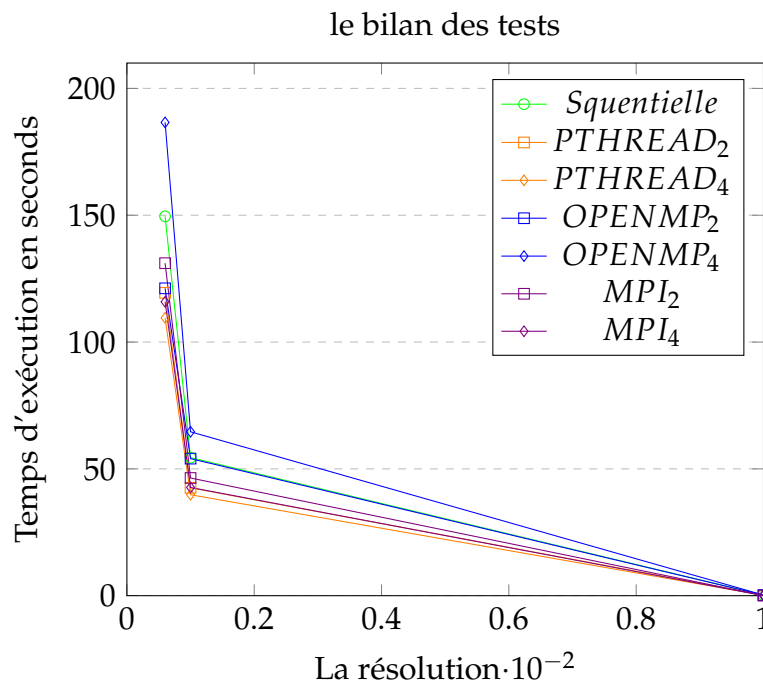


FIGURE I.5 – le temps d'exécution en fonction de la résolution

le sens d'évolution est de droite à gauche plus on diminue la résolution, le temps nécessaire pour terminer va augmenter.

On peut remarquer que la meilleure optimisation par rapport au programme séquentielle est obtenue en utilisant Pthread ensuite c'est MPI, donc on peut conclure que malgré la simplicité du directive OPENMP le travail qui doit être réaliser le compilateur pour l'interprétation est beaucoup plus que le travail qui doit être réaliser lorsqu'on travaille avec Pthread, par conséquence le temps d'exécution augmente. En sachant que le temps d'interprétation reste le même si on diminue beaucoup la résolution et on augmente le nombre de threads, le temps qui sera pris pour terminer en utilisant OPENMP va être sûrement trop inférieur au temps qui sera pris par le programme séquentielle mais vu qu'on dispose d'une machine qui n'est pas puissante, on peut pas réaliser ce test.

Chapitre II

Administration système

Dans cette partie d'administration nous allons voir l'installation d'un cluster avec un environnement système minimal. Ce cluster sera constitué de trois nœuds :

- Nœud serveur d'administration (Cluster0)
- Nœud client de référence (Cluster1)
- Nœud client déployé (Cluster2)

Nous verrons aussi l'installation de quelques outils nécessaires à l'administration d'un cluster, et également comment déployer une image système sur le nœud Cluster2 en utilisant l'outil SystemImager.

II.1 Installation du système de base CentOS

Nous avons configuré 3 machines virtuelles sous VMware Workstation 15, tout en veillant à ce que l'espace disque utilisé pour le nœud Cluster0 soit plus important que celui utilisé par les deux autres nœuds. Cela dans le but de pouvoir contenir l'image système Cluster1.

Cluster0 :

Le Cluster0 est une machine virtuelle installé avec la configuration VMWare suivante :

- Custom (advanced)
- Guest OS Installation : Installer disc image file (iso)
- Distribution Red Hat Linux
- Processor Configuration : 2 processors
- RAM : 2 Go
- Network Type : host-only (nous avons configuré notre VMWare de telle sorte que le réseau host_only ne contienne plus de service DHCP)
- IO Controller Type : LSI-Logic
- Disc Type : SCSI
- Disc Capacity : 30 Go.

Cluster1 :

Le Cluster1 est installé comme **minimal installation groupe** avec la configuration VMWare suivante :

- Distribution Red Hat Linux
- Processor Configuration : 1 processors
- RAM : 1 Go
- Network Type : host-only
- IO Controller Type : LSI-Logic
- Disc Type : SCSI
- Disc Capacity : 10 Go.

II.1.1 Mise en place du réseau local

Les machine virtuelles *host_only* (*VMnet1*) sont configuré avec le réseau 192.168.10.0/24 avec la fonction DHCP désactivée.

- Cluster0 192.168.10.1
- Cluster1 192.168.10.2
- Cluster2 192.168.10.3

Nous allons donc configurer les adresses IP du cluster statiquement avec la commande :

```
nano /etc/sysconfig/network-scripts/ifcfg-eth0
```

```
DEVICE=eth0
BOOTPROTO=static
HWADDR=00:0C:29:7A:69:2A
ONBOOT=yes
IPADDR=192.168.10.1
NETMASK=255.255.255.0
NETWORK=192.168.10.0
BROADCAST=192.168.10.255
```

(A) Cluster0

```
DEVICE=eth0
BOOTPROTO=static
HWADDR=00:0C:29:7C:69:7B
ONBOOT=yes
IPADDR=192.168.10.2
NETMASK=255.255.255.0
NETWORK=192.168.10.0
BROADCAST=192.168.10.255
```

(B) Cluster1

FIGURE II.1 – IP adresses

II.2 Mise en place d'un environnement minimal d'administration distante

II.2.1 Résolution de nom

Pour faciliter la manipulation du cluster sans s'encombrer avec les des adresse ipv4, nous alons utiliser la resolution de noms.

```
nano /etc/hosts
```

```
#Ecrire les resolutions de noms
192.168.10.1 cluster0
192.168.10.2 cluster1
192.168.10.3 cluster2
```

```
[root@cluster0 massinissa]# ping cluster1
PING cluster1 (192.168.10.2) 56(84) bytes of data.
64 bytes from cluster1 (192.168.10.2): icmp_seq=1 ttl=64 time=1.32 ms
64 bytes from cluster1 (192.168.10.2): icmp_seq=2 ttl=64 time=0.665 ms
64 bytes from cluster1 (192.168.10.2): icmp_seq=3 ttl=64 time=1.24 ms
64 bytes from cluster1 (192.168.10.2): icmp_seq=4 ttl=64 time=0.650 ms
64 bytes from cluster1 (192.168.10.2): icmp_seq=5 ttl=64 time=1.25 ms
64 bytes from cluster1 (192.168.10.2): icmp_seq=6 ttl=64 time=1.23 ms
^C
--- cluster1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5009ms
rtt min/avg/max/mdev = 0.650/1.060/1.320/0.288 ms
```

FIGURE II.2 – Test de connectivité ICMP

II.2.2 Configuration SSH

SSH est à la fois un protocole de communication sécurisé, il permet de se connecter sur des machines distantes de façon sécurisée soit avec mot de passe soit avec un couple de clé (public, private). Dans le cadre de notre projet, nous allons permettre à Cluster0 de se connecter en SSH (sans mot de passe) sur les autres parties du Cluster.

Générer les clés RSA sur le client (Cluster0)

```
mkdir /root/.ssh
cd /root/.ssh
ssh-keygen -t rsa
```

```
ls -l
total 8
-rw----- . 1 root root 1679 Apr 30 21:30 mykey #private key
-rw-r--r-- . 1 root root 395 Apr 30 21:30 mykey.pub #public key
```

Nous avons généré les clé privée et public *mykey* et *mykey.pub* sans mot de passe. Ensuite, Nous allons transférer la clé publique vers Cluster1 qui va la mettre dans **authorized_keys** pour permettre la connexion.

```
ssh-copy-id -i /root/.ssh/mykey.pub root@cluster1
ssh-add mykey #permettre la connexion sans mot de passe
```

```
[root@cluster0 .ssh]# ssh root@cluster1
Last login: Thu Apr 30 16:23:31 2020 from 192.168.10.1
[root@cluster1 ~]# echo $HOSTNAME
cluster1
[root@cluster1 ~]#
```

FIGURE II.3 – Test de connectivité SSH

II.2.3 Configuration d'un export NFS

Network File System (NFS), littéralement système de fichiers en réseau, est un protocole qui permet à un nœud d'accéder à des fichiers distants. Nous allons configurer un export NFS du répertoire partagé «export_nfs_dir» du nœud d'admin Cluster0 pour l'ensemble des nœuds clients Cluster1 et Cluster2.

1. création du répertoire exporté sur le serveur Cluster0

```
mkdir /nfs_export_cluster_dir
```

2. configuration de /etc/exports sur cluster0

```
cat /etc/exports
/nfs_export_cluster_dir cluster1(rw, sync, no_root_squash,
                                no_subtree_check)
/nfs_export_cluster_dir cluster2(rw, sync, no_root_squash,
                                no_subtree_check)
```

Où :

- **rw** : permet la lecture et l'écriture sur un partage pour l'hôte défini.
- **async** : cette option permet d'améliorer les performances en dépit d'un risque de perte de données en cas d'un redémarrage non propre. Car elle permet de répondre aux requêtes avant que les changements effectués par la requête aient été appliqués sur l'unité de stockage, ce qui est impropre au protocole NFS [3].
- **no_root_squash** : spécifie que le root du nœud sur lequel le répertoire est monté a les droits de root sur le répertoire.
- **no_subtree_check** : Cette option neutralise la vérification de sous-répertoires, ce qui a des subtiles implications au niveau de la sécurité, mais peut améliorer la fiabilité dans certains cas [3].

3. Activation du service nfs sur cluster0

```
service nfs start
systemctl status nfs # vérifie qu'il est bien activé
```

4. Création sur les clients du répertoire de montage

```
mkdir nfs_cluster_dir
```

5. Montage sur les noeuds clients

```
#on desactive le firewall SELinux sur Cluster0 avec
systemctl disable firewalld
## on revien vers Cluster1
nano /etc/fstab
cluster0:/nfs_export_cluster_dir /nfs_cluster_dir nfs
                                defaults 0 0
##
mount -t nfs cluster0:/nfs_export_cluster_dir
                                /nfs_cluster_dir
# on verifi avec cette ligne de commande
```



```
df -h
```

```
[root@cluster1 /]# df -h
Filesystem                Size      Used Avail Use% Mounted on
devtmpfs                   484M        0   484M   0% /dev
tmpfs                      496M        0   496M   0% /dev/shm
tmpfs                      496M    7.2M   489M   2% /run
tmpfs                      496M        0   496M   0% /sys/fs/cgroup
/dev/mapper/centos-root    13G     1.7G    11G  14% /
/dev/sda1                  1014M    158M    857M  16% /boot
tmpfs                      100M        0   100M   0% /run/user/0
cluster0:/nfs_export_cluster_dir 27G     4.3G    23G  16% /nfs_cluster_dir
```

FIGURE II.4 – Vérification du montage nfs

II.2.4 Installation de l'outil « pdsh »

Parallel Distributed Shell est un outil qui nous permettra d'exécuter des commandes sur des nœuds clients distants, d'une manière parallèle et performante. Son utilité apparaît lors de l'administration d'un cluster contenant un nombre important de nœuds, où l'exécution séquentielle d'une même commande manuellement serait ardue.

Dans ce qui suit, nous allons installer via les outils dans l'image iso «pdsh et pdsh_rcmd_ssh» sur le nœud d'admin Cluster0 pour lancer des commandes parallèles sur les deux autres nœuds Cluster1 et Cluster2.

```
rpm -ivh PDSH_2_18_1_I386.RPM
rpm -ivh PDSH_RCMD_SSH_2_18_1_I386.RPM
```

Explorer les différentes commandes

:

pdsh : permet l'exécution des commandes en parallèle sur nos machines cluster[1-2] .

pdcp : permet de copier un fichier de Cluster0 vers les autres nœuds du Cluster.

dshbak : c'est une option qui met en forme les output et les afficher dans le terminal.

-c : synthétise l'affichage.

Voir les figures suivantes :

```
[root@cluster0 massinissa]# pdsh -w cluster[1-2] vmstat|dshbak -c
-----
cluster1
-----
procs -----memory----- ---swap-- ----io---- -system-- -----cpu-----
r b  swpd  free  buff  cache  si  so   bi   bo   in  cs us sy id wa st
3  0      0 696960  2116 145740  0  0   814   37  320  906  6 13 81  0  0
-----
cluster2
-----
procs -----memory----- ---swap-- ----io---- -system-- -----cpu-----
r b  swpd  free  buff  cache  si  so   bi   bo   in  cs us sy id wa st
2  0      0 684564  2116 148984  0  0   684   35  299  763  5 12 82  0  0
-----
```

FIGURE II.5 – Test pdsh

```
[root@cluster0 ~]# nano test.txt
[root@cluster0 ~]# pdsh -w cluster1 ls /root |dshbak -c
-----
cluster1
-----
anaconda-ks.cfg
[root@cluster0 ~]# pdcp -w cluster1 test.txt /root
[root@cluster0 ~]# pdsh -w cluster1 ls /root |dshbak -c
-----
cluster1
-----
anaconda-ks.cfg
test.txt
[root@cluster0 ~]#
```

FIGURE II.6 – Test pdcp

II.3 Outils de déploiements de cluster : SystemImager

SystemImager est un logiciel qui automatise les installations Linux, la distribution de logiciels et le déploiement de production.

SystemImager facilite les installations automatisées (clones), la distribution de logiciels, la distribution de contenu ou de données, les changements de configuration et les mises à jour du système d'exploitation sur votre réseau de machines Linux.

Il peut également être utilisé pour garantir des déploiements de production en toute sécurité. En enregistrant votre image de production actuelle avant la mise à jour vers votre nouvelle image de production, vous disposez d'un mécanisme d'urgence hautement fiable [1].

Nous allons utiliser SystemImager pour créer l'image système de Cluster1 « Golden client », la récupérer sur Cluster0 « Serveur de déploiement », la déployer et l'installer sur Cluster2.

II.3.1 Installation des packages systemimager

1. installation des dépendances pour Cluster[0-1] :

```
rpm -ivh
PERL_APPCONFIG_1_66_1_EL5_R.RPM
PERL_XML_PARSER_2_34_6_1_2_.RPM
PERL_XML_SIMPLE_2_14_4_FC6_.RPM
```

2. installation des packages en communs pour Cluster[0-1] :

```
rpm -ivh
SYSTEMCONFIGURATOR_2_2_11_0.RPM
SYSTEMIMAGER_COMMON_4_1_6_1.RPM
```

3. installation des packages spécifiques au client et au serveur :

```
# Pour le client Cluster1 on install
rpm -ivh
SYSTEMIMAGER_CLIENT_4_1_6_1.RPM
SYSTEMIMAGER_I386INITRD_TEM.RPM
# Pour le serveur Cluster0 on install
rpm -ivh
SYSTEMIMAGER_SERVER_4_1_6_1.RPM
SYSTEMIMAGER_I386BOOT_STAND.RPM
```

II.3.2 Prise d'une image golden

Préparation du client à prise d'une image

Dans la machine cluster1 on execute la commande suivante :

```
si_prepareclient --server cluster0
```

Permet de créer une image système du Cluster1, dont le nom est « initrd.img » qui sera positionnée dans le répertoire /etc/systemimager/boot , et autorise sa récupération par Cluster0. Elle lance pour ce faire, un **daemon rsync**

Récupération et configuration de l'image sur le serveur

Sur le Cluster0 on va récupérer l'image du Cluster1 avec :

```
si_getimage --golden-client cluster1 --image imgCluster1
--post-install reboot
```

On accepte de lancer la commande **si_clusterconfig -e** , qui permet de modifier le fichier `"/etc/systemimager/cluster.xml"`, lequel décrit la topologie des clients et les informations du serveur, il est utilisé par toutes les commandes pour identifier les groupes des clients et leurs images associées. On ajoute un nouveau groupe :

```
<group>
  <name>VMCluster</name>
  <priority>20</priority>
  <image>imgCluster1</image>
  <override>imgCluster1</override>
  <node>cluster1,cluster2</node>
</group>
```

II.3.3 Préparation du serveur pour déployer

Installation d'un serveur dhcp et tftp

Pour transférer l'image système "imgCluster1" de Cluster0 vers Cluster2. Le Cluster2 doit posséder une adresse IP afin que Cluster0 puisse le contacter. Et installer sur le Cluster0 un serveur de fichiers. Pour attribuer une configuration IP au Cluster2, nous installons un serveur DHCP sur Cluster0 et pour le transfert de fichiers, on installe un serveur TFTP sur Cluster0.

```
rpm -ivh
xinetd-2.3.14-10.el5.i386.rpm
DHCP_3_0_5_18_EL5_I386.RPM
TFTP_0_42_3_1_EL5_CENTOS_I3.RPM
TFTP_SERVER_0_42_3_1_EL5_CE.RPM
```

Configuration de la chaine de boot pour systemimager : serveur dhcp et tftp

La configuration des services de boot dhcp et tftp pour systemimager est automatisée par l'exécution des commandes suivantes : «**si_mkbootserver**» et «**si_mkdhcpserver**».

Mais avant cela on démarre d'abord le service "xinetd" qui permet de démarrer **tftp-server**

```
service xinetd start
si_mkbootserver
si_mkdhcpserver
## editer dhcpd.conf
nano /etc/dhcpd.conf
```

Il est nécessaire de retoucher la configuration du serveur dhcp pour rajouter explicitement les noeuds clients.

```
host cluster2{
option host-name "cluster2";
hardware ethernet 00:0C:29:C7:2C:3A:3C;
fixed-address 192.168.10.3;
}
```

FIGURE II.7 – Configuration DHCP

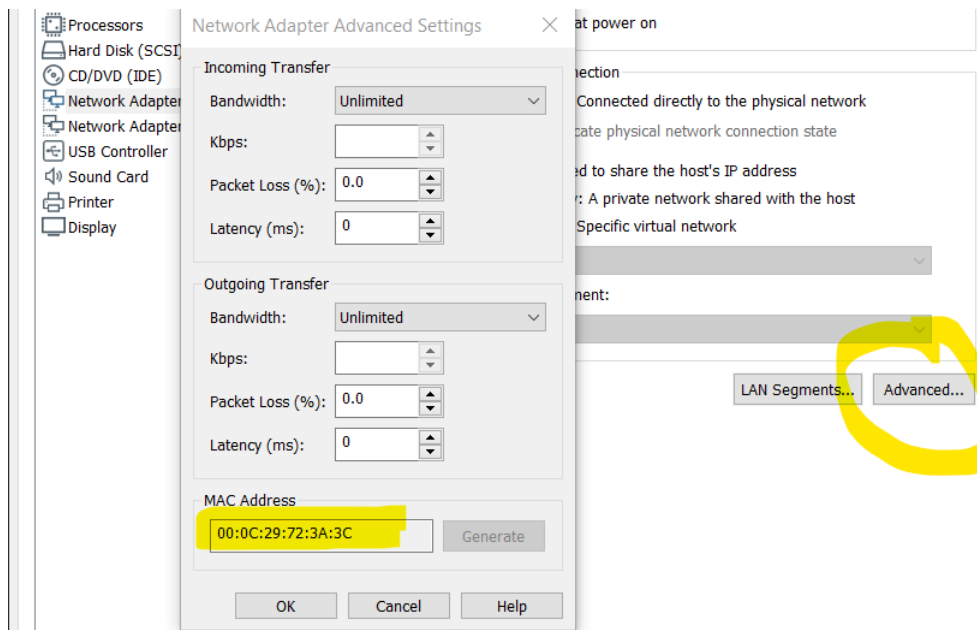


FIGURE II.8 – Recupération de l'adresse MAC de cluster2

Puis on redemmare le service via :

```
service dhcpd restart
```

II.3.4 . Déploiement du noeud client

Démarrage des services de déploiement

1. On démarre serveur rsync qui prend charge le déploiement de l'image système, lequel est contrôlé par le service "systemimager-server-rsyncd"

```
service systemimager-server-rsyncd start
```

2. On démarre le service "systemimager-server-netbootmond" qui contrôle le boot et l'installation des clients. Ce service modifie la configuration de boot des clients lorsqu'il détecte la fin de l'installation. Nous l'utilisons pour forcer les clients à rebooter en local et non plus en réseau. Pour cela, nous avons eu à éditer le fichier `/etc/systemimager/systemimager.conf` puis redemarrer le service :

```

service systemimager-server-netbootmond start
## modifier cette ligne dans le fichier
nano /etc/systemimager/systemimager.conf
    NET_BOOT_DEFAULT=local
# redemmarage
service systemimager-server-netbootmond restart

```

Déploiement du client Cluster2

Afin que le noeud client "Cluster2" n'ait pas une adresse IP autre que celle spécifiée précédemment, il est nécessaire de désactiver le serveur DHCP de la VMware.

Puis on configure la machine virtuelle Cluster2 de sorte qu'elle boot en réseau et non pas en local. Pour accéder au BIOS virtuel, il faut :

- Faire un clic droit sur la machine client
- On va vers l'option **Power**
- Simple clic sur **Power On to Firmware**

Une fois dans le Menu on va dans la section **BOOT**, on fait remonter le démarrage correspondant au réseau *host-only* en première priorité.

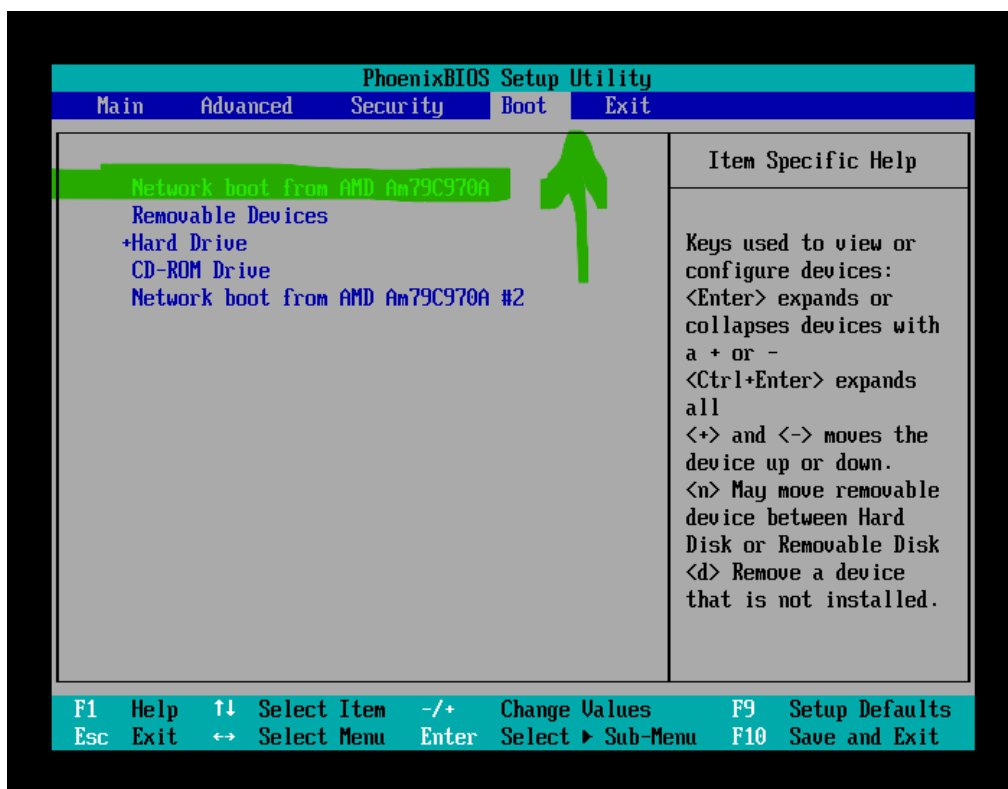


FIGURE II.9 – Rendre le boot réseau prioritaire

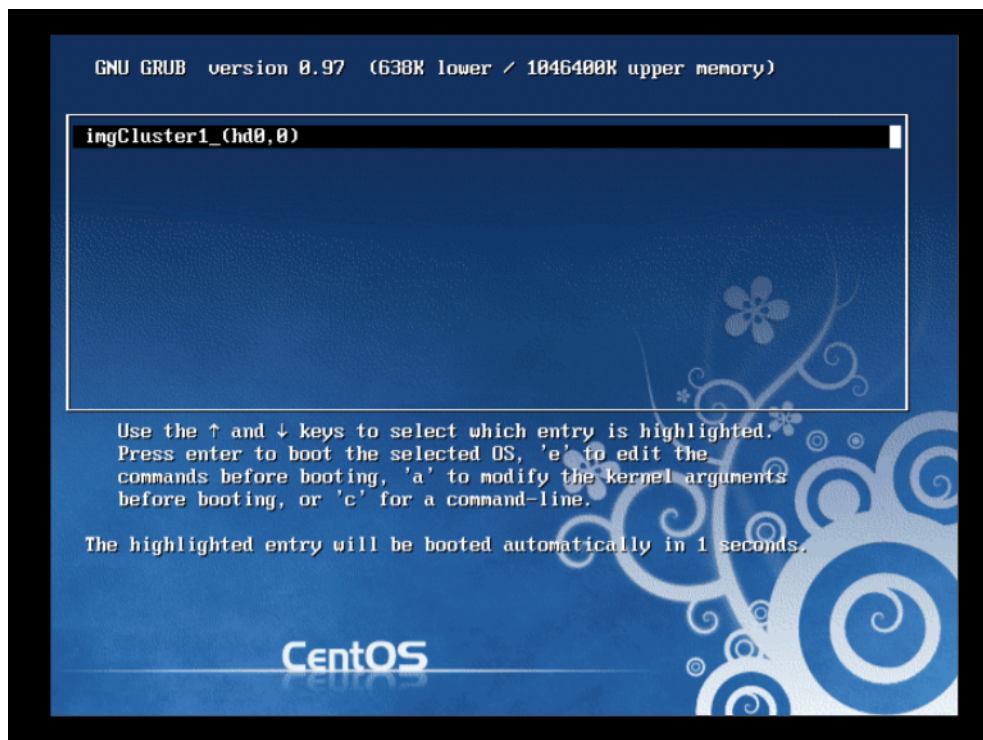


FIGURE II.10 – Grub qui affiche l'image système

Chapitre III

Gestion des ressources

III.1 SLURM

III.1.1 Définitions et quelques commandes de base

SLURM est un job Scheduler. Son rôle est de maintenir une image aussi précise que possible de l'état d'utilisation des ressources sur le cluster. Il se base sur cette image pour placer les jobs des utilisateurs en fonction des ressources libres. Il assure l'ordonnancement et la planification des travaux.

La soumission des jobs à SLURM se fait soit en mode « interactif » soit en mode « batch » :

- En mode interactif, l'utilisateur aura accès à un Shell interactif qui se situe sur le premier nœud réservé pour la tâche en question.
- En mode batch, il faut spécifier un script lors de la soumission qui sera exécuté sur le(s) nœud(s) réservé(s).

Les options de SLURM permettent dans ces deux modes de définir des paramètres tels que : la partition, la durée du job, le nombre de nœuds, de cœurs, la quantité de mémoire, le nom du job, les noms des fichiers de sortie, etc...

Lors de la soumission, SLURM indique le numéro d'identification du job qui permettra de le suivre et d'interagir avec lui.

On peut interagir avec le gestionnaire de ressources grâce aux commandes suivantes :

- **srun <paramètres>** : exécution d'une tâche en mode interactif. Elle prend en paramètre les différentes options, comme le nombre de nœuds, le nombre de tâches, le nombre de processeurs par tâche, ...etc
- **sbatch <fichier_batch>** : soumission d'un job dans une file d'attente (partitions dans SLURM). Il prend en paramètre un fichier script Shell qui sera exécuté par la suite.
- **sinfo** : pour l'interrogation des files d'attente. Elle permet d'afficher l'ensemble du cluster, le nombre et la liste de nœuds, l'état, ...etc
- **squeue** : interrogation des jobs. Elle permet de voir l'état du cluster, les différents nœuds, l'ID et l'état des tâches, le temps d'exécution consommé, ...etc
- **scontrol<paramètres>** : elle permet de mettre à jour l'état des nœuds et d'autres informations sur le cluster.
- **scancel** : Pour la suppression d'un job

— **sprio** : priorités relatives entre les jobs en attente.

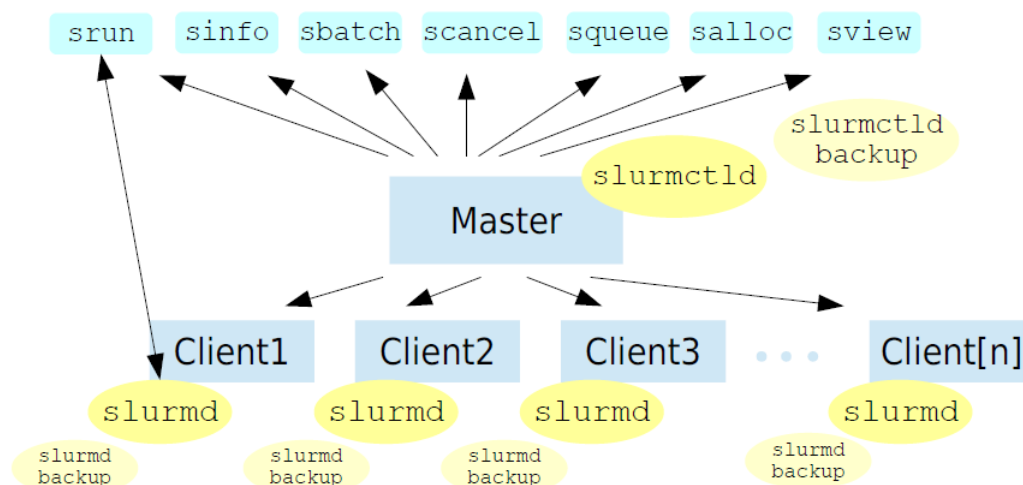


FIGURE III.1 – Description générale de l'architecture de SLURM

III.1.2 MUNGE

MUNGE est un service d'authentification pour la création et la validation des informations d'identification qui identifie l'utilisateur à l'origine d'un message. En effet, chaque message a une section « auth_info » générée par l'émetteur et validée par le récepteur qui contient les UID/GID de l'émetteur. Il permet d'authentifier par le biais d'une clé partagée les échanges entre 2 services fournis par SLURM : SLURMCTLD et SLURMD.

- SLURMCTLD est un service actif sur le nœud d'administration (Cluster0). Il connaît la topologie du cluster et l'état global d'utilisation des ressources.
- SLURMD est un service actif sur les nœuds de calcul (Cluster1 et Cluster2). Il est chargé de lancer des jobs sur un nœud de calcul, y gère la consommation de ressources.

III.1.3 Fonctionnement

- L'utilisateur se connecte au cluster et utilise pour cela la commande « srun » pour le mode interactif ou « sbatch » pour le mode batch, puis envoie un job sur le cluster.
- Une connexion à un service « SLURMCTLD » en exécution sur le nœud d'administration est établie. Ce service connaît la topologie du cluster ainsi que l'état d'utilisation des ressources.
- Si les ressources demandées par l'utilisateur sont libres, SLURMCTLD demande au(x) service(s) SLURMD de(s) nœud(s) de calcul(s) réservé(s) d'exécuter le job.

- Le service SLURMD accepte la requête du service SLURMCTLD et crée avec un « fork » un processus SLURMSTEPD qui va gérer la consommation de ressources du job ainsi que ses entrées / sorties.
- Le job est un processus fils de SLURMSTEPD exécuté sous l'identité de l'utilisateur.

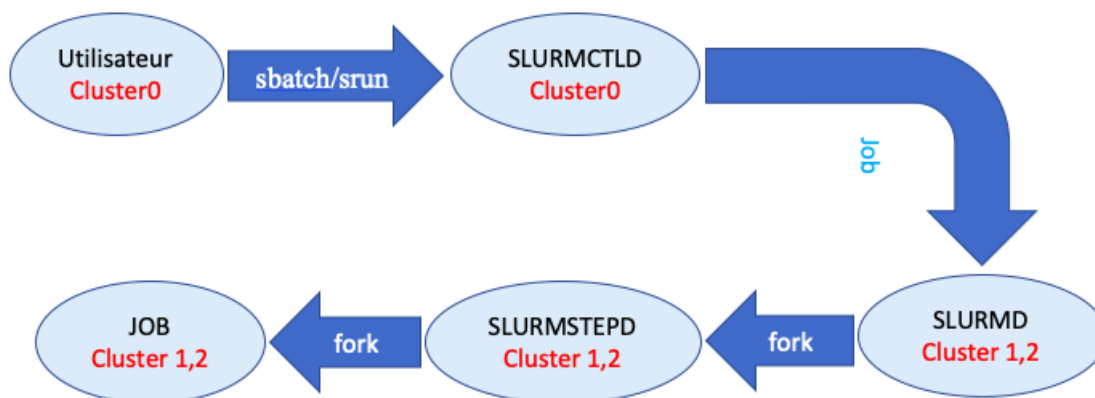


FIGURE III.2 – Fonctionnement de SLURM

III.1.4 Installation et configuration

Installation d'un environnement de compilation et de génération de RPM

Nous allons commencer par mettre en place une plateforme de compilation et de génération de RPM sur le Cluster0, en installant les packages :

```
rpm -ivh --nodeps --force
ELFUTILS_0_137_3_EL5_I386.RPM
ELFUTILS_LIBS_0_137_3_EL5_I.RPM
RPM_BUILD_4_4_2_3_9_EL5_I38.RPM
```

Et les packages : **Gcc, Glibcdevel, Glibcheaders, Kernelheader, Libgomp** pour la compilation.

```
rpm -ivh
GCC_4_1_2_44_EL5_I386.RPM
GLIBC_DEVEL_2_5_34_I386.RPM
GLIBC_HEADERS_2_5_34_I386.RPM
KERNEL_HEADERS_2_6_18_128_E.RPM
LIBGOMP_4_3_2_7_EL5_I386.RPM
```

Installation de Munge

Afin de sécuriser les communications intra-cluster, d'authentifier chaque utilisateur et de vérifier l'authenticité de chaque message nous aurons besoin d'installer le plugin MUNGE.

Après l'installation des prérequis prérequis à la compilation de MUNGE sur tous les noeuds, On compile avec :

```
rpmbuild --rebuild munge0.5.81.src.rpm
```

Nous allons procéder à l'installation de MUNGE comme suite avec les fichiers générés :

- Sur Cluster0, on installe les RPMs (**munge,munge-libs, munge-devel**).
- Sur Cluster1 et Cluster2, on installe uniquement **munge, munge-libs**. Car le RPM « **munge-devel** » sert uniquement à la compilation de SLURM et pas pour le fonctionnement du produit.

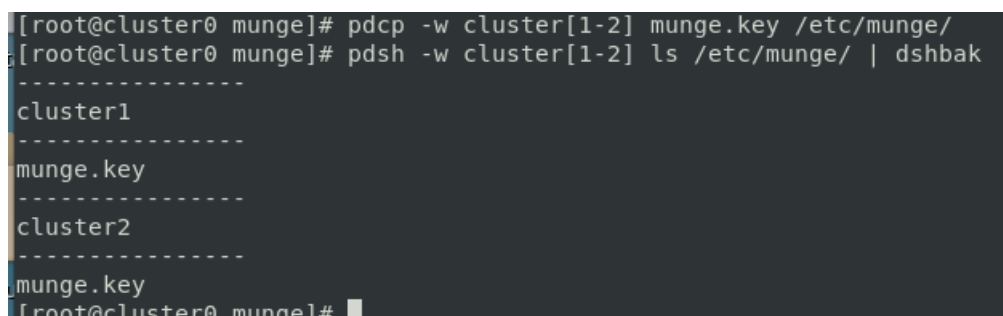
Configuration et lancement de MUNGE

1. Après avoir généré et installé les RPM de MUNGE, on a généré une clé aléatoire de 1024 blocs d'un octet chacun en exécutant sur le Cluster0 la commande suivante :

```
dd if=/dev/urandom of=/etc/munge/munge.key bs=1 count=1024
```

2. Copier la Clé du Cluster0 avers Cluster1 et Cluster2 :

```
pdcp -w Cluster[1-2] munge.key /etc/munge  
pdsh -w Cluster[1-2] ls /etc/munge | dshbak
```



```
[root@cluster0 munge]# pdcp -w cluster[1-2] munge.key /etc/munge/  
[root@cluster0 munge]# pdsh -w cluster[1-2] ls /etc/munge/ | dshbak  
-----  
cluster1  
-----  
munge.key  
-----  
cluster2  
-----  
munge.key  
[root@cluster0 munge]#
```

FIGURE III.3 – Tansfert de la clé munge

3. Démarrer le service MUNGE sur tous les nœuds : cluster0, cluster1 et cluster2 en exécutant la commande suivante sur les 3 nœuds :

```
service munge start
```

installation de SLURM

1. Nous allons commencer par l'installation des RPM rérequis à la compilation de SLURM : **Readline-devel, Pam-devel et Libtermcap-devel** avec la commande vue précédemment.
2. Ensuite nous allon procédé à la compilation et la génération des RPM binaires installables à partir de l'archive source de SLURM. Sur le Cluster0, on exécute la ligne de commande suivante :

```
rpmbuild -ta /usr/src/redhat/SOURCES/slurm-2.0.8.tar.bz2
```

-ta : Cette option est utilisée lorsque la source de la commande rpm-build est une archive TAR.

3. Ensuite, sur tous les nœuds du Cluster, on installe les trois RPM générés à l'étape précédente (**Slurm, Slurm-plugins et Slurm-munge**).
4. Pour la configuration de SLURM, on va créer dans le répertoire **/etc/slurm/** et un fichier qu'on nommera **slurm.conf**. On copiera dans le contenu du fichier **/etc/slurm/slurm.conf.exemple** avec la commande suivante :

```
cp /etc/slurm/slurm.conf.exemple /etc/slurm/slurm.conf
```

et on va adapter son le contenu en modifiant son contenu comme suit :

```
ClusterName=Cluster
ControlMachine=Cluster0
SlurmUser=root
SlurmctldPort=6817
SlurmdPort=6818
AuthType=auth/munge
StateSaveLocation=/tmp
SlurmdSpoolDir=/tmp/slurmd
SlurmdTimeout=300
SchedulerType=sched/backfill
FastSchedule=1
NodeName=Cluster[1-2] Procs=1 State=UNKNOWN
PartitionName=VMCluster Nodes=Cluster[1-2] Default=yes
MaxTime=INFINITE State=UP
```

5. Ensuite, on va synchroniser les configurations sur les trois nœuds. Pour cela, on va copier le fichier de configuration **/etc/slurm/slurm.conf** sur les deux autres nœuds avec la commande suivante :

```
pdcp -w Cluster[1-2] /etc/slurm/slurm.conf /etc/slurm/
```

6. Et enfin, on lance le service SLURM sur l'ensemble des nœuds du cluster.

```
Service slurm start
```

Utilisation de SLURM

Dans cette partie, nous allons exécuter un ensemble de commandes afin de tester notre solution et de mettre en pratique les commandes décrites plus haut.

Informations sur le Cluster

Pour afficher des informations sur les partitions et les différents nœuds du Cluster on utilisera la commande suivante :

```
sinfo
```

```
[root@cluster0 ~]# sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE  NODELIST
compute*   up    infinite     2  down*  Cluster[1-2]
[root@cluster0 ~]#
```

FIGURE III.4 – Test sinfo

On peut aussi changer l'état des clusters de DOWN vers un état PRET (idle)

```
scontrol update nodename=Cluster[1-2] state=idle
```

```
[root@cluster0 ~]# sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE  NODELIST
compute*   up    infinite     2   idle  Cluster[1-2]
[root@cluster0 ~]#
```

FIGURE III.5 – Test sinfo

Creation d'un job

Un travail ou job se présente généralement sous la forme d'un script Shell. Il peut également s'agir d'une simple commande. Il se compose de deux parties :

- La partie demande de ressources ou il faut spécifier le nom de la tâche, le nombre de nœuds, le nombre de cœurs, la mémoire vive nécessaire, le nombre de tâches à exécuter, le temps nécessaire à l'exécution du travail, le fichier de sortie ...etc

Ces informations permettent d'optimiser l'allocation de ressources pour vos travaux.

- Une autre partie pour les tâches : ou on liste les tâches à exécuter.

La manière typique de créer un travail consiste à écrire un script Shell (par exemple un script bash) dont les commentaires, s'ils sont préfixés avec SBATCH, sont interprétés par SLURM comme des paramètres décrivant les demandes de ressources.

Le script lui-même constitue une étape du job. Les autres étapes sont créées avec la commande «srun».

Le script suivant, enregistré dans un fichier « test.sh », demande un processeur pendant 10 minutes, ainsi que 100 Mo de RAM, dans la file d'attente par défaut. Une fois démarré, le travail exécutera la première étape du job «srun hostname», qui lancera la commande UNIX «hostname» sur le nœud sur lequel la CPU demandée a été allouée. Ensuite, une deuxième étape de travail démarre la commande «sleep».

```
#!/bin/bash
#
#SBATCH --job-name=test
#SBATCH --output=res.txt
#
#SBATCH --ntasks=1
#SBATCH --time=10:00
#SBATCH --mem-per-cpu=100
srun hostname
srun sleep 60
```

Soumission d'une tâche

Comme on a déjà vu précédemment, il existe deux façons de le faire soit avec « srun » ou avec « sbatch ». Ces deux commandes ont un fonctionnement similaire à la différence que « srun » ne rend la main qu'à la fin de l'exécution du travail alors que « sbatch » rend immédiatement la main après avoir soumis notre travail à la ferme de calcul.

Dans notre cas, une fois le script écrit, on va le soumettre à SLURM via la commande « sbatch » qui, en cas de succès, répond avec l'ID attribué à la tâche.

```
sbatch test.sh
```

Informations sur les travaux

Après sa soumission, le travail entre dans la file d'attente à l'état « PENDING ». Une fois que les ressources sont disponibles et que la tâche a la priorité la plus élevée, une allocation est créée pour elle et elle passe à l'état « RUNNING ». Si le travail se termine correctement, il passe à l'état « COMPLETE », sinon, il passe à l'état « FAILED ».

Pour obtenir l'état des différentes tâches on utilise la commande « squeue ».

```
sbatch: Submitted batch job 25
[root@cluster0 ~]# squeue
  JOBID PARTITION  NAME     USER  ST       TIME  NODES NODELIST(REASON)
   25    compute    test     root   R        0:03      2 cluster[1-2]
[root@cluster0 ~]# scancel 25
```

FIGURE III.6 – Test de lancement de tâches et manipulation

On peut aussi obtenir des informations en temps quasi réel sur le programme en cours d'exécution avec la commande « sstat ». Pour annuler/arrêter un travail il suffit d'exécuter la commande « scancel »

```
sstat -j <jobid>
scancel <jobid>
```

Chapitre IV

Système de fichiers parallèles

IV.1 LUSTRE

En 2013 le chercheur Jean Michel a réaliser une simulation cosmologique sur Curie, 5PB de données ont été produit lors de la simulation, on constate donc que les données sont le coeur des centres de calcules et donc on a besoin d'un systèmes de gestion de données, plus précisément on a besoin que plusieurs noeuds peuvent écrire dans des places différents du même fichier au même temps alors que d'autres peuvent le lire.

LUSTRE nous permet gérer l'accès a un système de fichiers en Cluster, il a été conçu pour assurer la scalabilité et la performance, il emploie une architecture client/serveur et il est utilisées par les plus grands super-ordinateurs classées par **TOP500**.

Dans cette partie on va installer et configurer **LUSTRE** pour l'utiliser sur les trois Cluster qu'ont a crée avant.

IV.1.1 Architecture LUSTRE

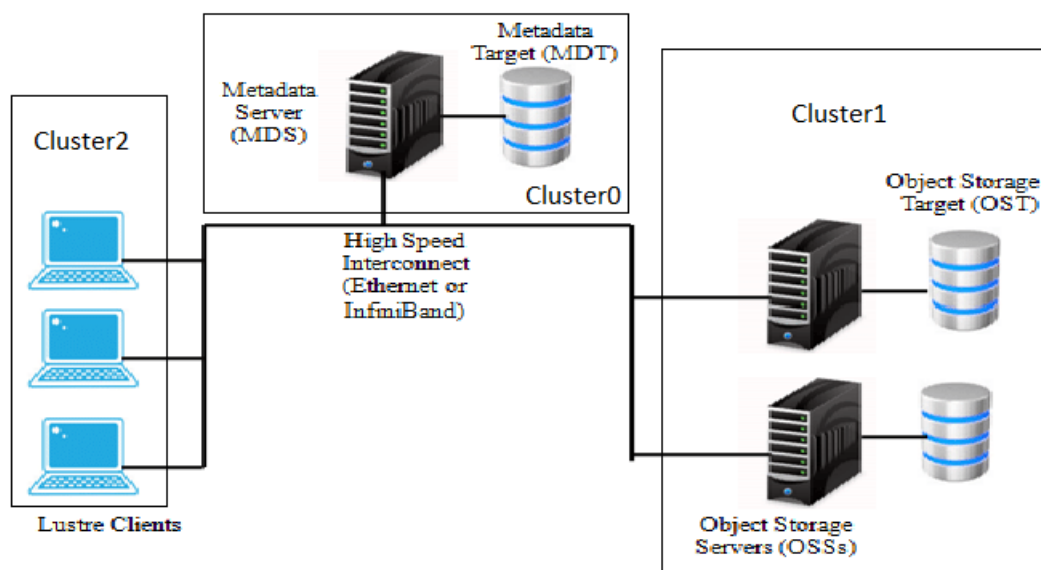


FIGURE IV.1 – Architecture du LUSTER

La figure IV.1 illustre l'architecture du **LUSTRE** qui comporte trois parties :

- **OSS** : Il stocke les données et gère les entrée sortie des fichiers du systèmes, le Cluster1 sera le **Lustre OSS** .
- **MDT/MGS** : **MDT** gère les méta données des fichier (les nomes, hiérarchie du dossier et les autres informations sur les fichier) et **MDS** constitue une interface logique pour ce dernier, **MGS** enregistre la configurations des fichiers du système, Le Cluster0 sera **MDT** et **MGS**.
- **CLIENT** : L'application qui va permettre au client d'accéder au système de fichier des données stockes dans **OSS**, le Cluster2 sera le **Lustre client**.

IV.1.2 Installation des paquets nécessaires

il faudra installer dans chaque Cluster des paquets, en utilisant la commande RPM.

Cluster0 (MDT + MGS)

```
1 rpm -ivh lustre-1.8.1.1-2.6.18_128.7.1.el5_lustre.1.8.1.1.i686.rpm
2 rpm -ivh kernel-lustre-2.6.18-128.7.1.el5_lustre.1.8.1.1.20091003130007.i686.rpm
3 rpm -ivh lustre-modules-1.8.1.1-2.6.18_128.7.1.el5_lustre.1.8.1.1.i686.rpm
4 rpm -ivh lustre-ldiskfs-3.0.9-2.6.18_128.7.1.el5_lustre.1.8.1.1.i686.rpm
5 rpm -ivh --force e2fsprogs-1.41.6.sun1-0redhat.rhel5.i386.rpm
```

Cluster1 (OSS)

Dans Cluster1 on va installer les même paquets

```
1 rpm -ivh lustre-1.8.1.1-2.6.18_128.7.1.el5_lustre.1.8.1.1.i686.rpm
2 rpm -ivh kernel-lustre-2.6.18-128.7.1.el5_lustre.1.8.1.1.20091003130007.i686.rpm
3 rpm -ivh lustre-modules-1.8.1.1-2.6.18_128.7.1.el5_lustre.1.8.1.1.i686.rpm
4 rpm -ivh lustre-ldiskfs-3.0.9-2.6.18_128.7.1.el5_lustre.1.8.1.1.i686.rpm
5 rpm -ivh --force e2fsprogs-1.41.6.sun1-0redhat.rhel5.i386.rpm
```

Cluster2 (Client)

```
1 rpm -ivh lustre-1.8.1.1-2.6.18_128.7.1.el5_lustre.1.8.1.1.i686.rpm
2 rpm -ivh kernel-lustre-2.6.18-128.7.1.el5_lustre.1.8.1.1.20091003130007.i686.rpm
3 rpm -ivh lustre-modules-1.8.1.1-2.6.18_128.7.1.el5_lustre.1.8.1.1.i686.rpm
```

IV.1.3 Création des espaces nécessaires nécessaires

On va simuler une partition à l'aide d'un fichier en utilisant la commande LOSETUP.

On va commencer d'abord par créer 2 fichiers de taille 100MB dans les deux Cluster 0 et 1.


```
#entrée
dd if=/dev/zero of=/lustre/mds bs=1M count=100
```

```
#sortie
100+0 records in
100+0 records out
104857600 bytes (105 MB) copied, 1.56641 s, 66.9 MB/s
```

Ensuite on va simuler les devices en utilisant la commande LOSETUP.

```
#entrée
losetup -f /lustre/mds
losetup -a
```

```
#sortie
/dev/loop0: [64768]:25661720 (/lustre/mds)
```

IV.1.4 Installation de la configuration Lustre

Installation du serveur de méta-données et activation (Cluster0)

On va formater la device de méta-données sur le noeud serveur **MDS**.

```
#entrée
mkfs.lustre --fsname=lustre --mgs --mdt /dev/loop0
```

```
#sortie
Permanent disk data:
Target:      lustre-MDTffff
Index:       unassigned
Lustre FS:   lustre
Mount type:  ldiskfs
Flags:       0x75
              (MDT MGS needs_index first_time update )
Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
Parameters: mdt.group_upcall=/usr/sbin/l_getgroups
checking for existing Lustre data: not found
device size = 100MB
3 10 0
formatting backing filesystem ldiskfs on /dev/loop0
target name  lustre-MDTffff
4k blocks    0
options      -i 4096 -I 512 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L lustre-MDTffff -i 4096 -I
512 -q -O dir_index,uninit_groups -F /dev/loop0
```

Ensuite on démarre **MDT**.

```
mount -t lustre /dev/loop0 /lustre
```

Installation d'un serveur de donnée et activations(Cluster1)

Pour formater la device de données il faudra spécifier l'adresse IP du serveur MDS, or on a fait le renommage des Cluster dans le fichier hosts, on va le remplacer par Cluster0.

```
#entrée
mkfs.lustre --fsname=lustre --mgsnode=Cluster0
--ost /dev/loop0
```

```
#sortie
Permanent disk data:
Target:      lustre-OSTffff
Index:       unassigned
Lustre FS:   lustre
Mount type:  ldiskfs
Flags:       0x72
              (OST needs_index first_time update )
Persistent mount opts: errors=remount-ro,extents,malloc
Parameters: mgsnode=192.168.10.2@tcp
checking for existing Lustre data: not found
device size = 100MB
3 10 0
formatting backing filesystem ldiskfs on /dev/loop0
      target name  lustre-OSTffff
      4k blocks    0
      options      -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L lustre-OSTffff -I 256 -q
-O dir_index,uninit_groups -F /dev/loop0
Writing CONFIGS/mountdata
```

Démarrage de OST

```
mount -t lustre /dev/loop0 /lustre
```

Montage sur le noeud client (Cluster2)

Il nous reste que la configuration du noeud client pour qu'il puisse accéder aux données.

```
mount -t lustre Cluster0:/dev/loop0 /lustre/
```

IV.1.5 Exploration des fonctionnalités annexes

Création d'un deuxième OST (Cluster1)

On va utiliser les mêmes commandes pour créer un deuxième OST dans le noeud OSS.

```
#entrée
dd if=/dev/zero of=/lustre2/ost bs=1M count=100
```

```
#sortie
100+0 records in
100+0 records out
104857600 bytes (105 MB) copied, 1.56641 s, 66.9 MB/s
```

```
#entrée
losetup -f /lustre2/ost
losetup -a
```

```
#sortie
/dev/loop0: [64768]:25661720 (/lustre/ost)
/dev/loop1: [64768]:10319879 (/lustre2/ost)
```

```
#entrée
mkfs.lustre --fsname=lustre --mgsnode=Cluster0
--ost /dev/loop1
```

```
#sortie
Permanent disk data:
Target:      lustre-OSTffff
Index:       unassigned
Lustre FS:   lustre
Mount type:  ldiskfs
Flags:       0x72
              (OST needs_index first_time update )
Persistent mount opts: errors=remount-ro,extents,malloc
Parameters:  mgsnode=192.168.10.2@tcp
checking for existing Lustre data: not found
device size = 100MB
3 10 0
formatting backing filesystem ldiskfs on /dev/loop1
      target name  lustre-OSTffff
      4k blocks    0
      options      -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L lustre-OSTffff -I 256 -q
-O dir_index,uninit_groups -F /dev/loop1
Writing CONFIGS/mountdata
```

```
mount -t lustre /dev/loop1 /lustre2
```

Gestion des politiques des stripping

Pour consulter ou configurer la manière de répartition des fichiers sur les noeuds de données **OST** on utilise respectivement les deux commandes : **lfs getstripe** et **lfs setstripe**.

Consultation de la configuration (Cluster2).

```
#entrée
lfs getstripe /lustre/
```

```
#sortie
OBDS:
0: lustre-OST0000_UUID ACTIVE
1: lustre-OST0001_UUID ACTIVE
/lustre/
(Default) stripe_count: 1 stripe_size: 1048576 strip_offset: 0
```

La taille de la bande (stripe) est par défaut 1MB, on va l'augmenter de 3MB.

```
#entrée
lfs setstripe -s 4MB /lustre/
```

```
#sortie
OBDS:
0: lustre-OST0000_UUID ACTIVE
1: lustre-OST0001_UUID ACTIVE
/lustre/
(Default) stripe_count: 1 stripe_size: 4194304 strip_offset: -1
```

Gestion des quotas

Les quotas permettent d'administrer et de limiter la quantité d'espace disque qu'un utilisateur ou un groupe d'utilisateurs peuvent utiliser sur une partition.

Avant de d'activer les quotas sur notre système, nous devons d'abord démonter la partition de notre système de fichier, puis la reformater à l'aide de **mkfs** pour pouvoir enfin la monter avec les nouvelles options d'activation des quotas. Ceci est valable pour toutes les partitions de notre système de fichier.

Pour le premier **OST** (Cluster1) :

```
#entrée
umount /lustre/
mkfs.lustre --reformat --fsname=lustre --mgsnode=Cluster0
--param ost.quota_type=ug --ost /dev/loop0
```

```
#sortie
Permanent disk data:
Target:      lustre-OSTffff
Index:       unassigned
Lustre FS:   lustre
Mount type:  ldiskfs
Flags:       0x72
              (OST needs_index first_time update )
Persistent mount opts: errors=remount-ro,extents,malloc
Parameters: mgsnode=192.168.10.1@tcp ost.quota_type=ug
device size = 100MB
3 10 0
formatting backing filesystem ldiskfs on /dev/loop0
          target name lustre-OSTffff
          4k blocks    0
          options      -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L lustre-OSTffff -I 256 -q
-O dir_index,uninit_groups -F /dev/loop0
```

Pour le deuxième **OST** (Cluster1) :

```
#entrée
umount /lustre/
mkfs.lustre --reformat --fsname=lustre --mgsnode=Cluster0
--param ost.quota_type=ug --ost /dev/loop1
```

```
#sortie
Permanent disk data:
Target:      lustre-OSTffff
Index:       unassigned
Lustre FS:   lustre
Mount type:  ldiskfs
Flags:       0x72
              (OST needs_index first_time update )
Persistent mount opts: errors=remount-ro,extents,malloc
Parameters: mgsnode=192.168.10.1@tcp ost.quota_type=ug
device size = 100MB
3 10 0
formatting backing filesystem ldiskfs on /dev/loop1
          target name lustre-OSTffff
          4k blocks    0
          options      -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L lustre-OSTffff -I 256 -q
          -O dir_index,uninit_groups -F /dev/loop1
```

Ainsi que pour le nœud client il faudra remonter la partition du système de fichier (Cluster2).

```
umount /lustre/
mount -t lustre Cluster0:/lustre /lustre/
```

Mise à plat pour démarrage.

```
lfs quotacheck ug /lustre
```

Démarrage.

```
lfs quotaon ug /lustre/
```

Positionnement, sur un client monté.

```
lfs setquota u root 40720 40920 1000 100 /lustre
```

Consultation.

```
lfs quota u root /lustre
```

Arrêt.

```
lfs quotaoff /lustre/
```

Conclusion

Dans ce projet nous avons pu nous initier au monde du calcul haute performance, nous avons pu appliqué les notions étudiées dans le cours de grilles et grappes dans la programmation parallèle et implémenté nos propres solutions.

Comme nous avons vu aussi dans une autre partie réservée pour l'administration système et clusters de multitudes d'outils qui permettent de déployer un cluster et de le gérer de différentes manières, nous avons eu à creuser amplement sur chacun d'entre eux pour savoir le mode de leur fonctionnement.

Bibliographie

- [1] Olivier LAHAYE. <https://github.com/finley/SystemImager/wiki>.
- [2] Dwaraka NATH. *Running an MPI Cluster within a LAN*. <https://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/>.
- [3] « NFS : Network File System - le partage réseau sous Linux ». In : (2009).
URL : <http://manpages.ubuntu.com/manpages/cosmic/en/man5/exports.5.html>.