Master 1 : SAR

MU4IN408 : PSAR

Un répartiteur de charge pour réseaux de machines

Encadrant : Sens Pierre

Ferche Adelin-Flaviu 3800655
Alallah Yassine 28707696

Année 2023-2024

Sommaire

- Introduction
- Sujet
- MPI
  - Introduction à MPI
  - Compilation et exécution d'un programme MPI
  - Communication
- Architecture
  - Topologie
  - Configuration SSH
- Implémentation
  - Arborescence de notre code
  - Structure interne d'un serveur
  - Gestion de la charge
    - Charge globale
    - Récupération de la charge
    - Mise à jour de la charge
    - Détection de la surcharge et de la sous-charge
      - Surcharge
      - Souscharge
    - Choix du programme à déplacer et de la machine cible
  - Commandes principales
    - start
    - gps
    - gkill
    - insert
    - remove
  - Gestion de la réception des messages par le serveur maître
- Difficultés rencontrées
- Conclusion

Introduction

L'équilibrage de charge, ou "load balancing" en anglais, est une technique essentielle pour optimiser les performances et la fiabilité des infrastructures informatiques. Il consiste à répartir la charge de travail entre plusieurs serveurs, ce qui permet de :

- Améliorer les temps de réponse et la fluidité de l'expérience utilisateur.
- Maximiser l'utilisation des ressources et réduire les coûts d'exploitation.
- Assurer la haute disponibilité et la résilience du système en cas de panne.

Dans un contexte où le volume de données et de requêtes ne cesse de croître, l'équilibrage de charge devient un élément incontournable pour garantir la performance et la scalabilité des architectures informatiques.

Ce rapport présente un projet de développement d'un répartiteur de charge à but éducatif. Ce projet a pour objet la mise en œuvre d'un ordonnancement des programmes sur un sous-ensemble de machines du réseau. Cet ordonnancement, appelé placement réparti, sera réalisé par une couche logicielle, le répartiteur de charge, qui se situera au-dessus du système d'exploitation de chaque machine.

Sujet

Les problèmes essentiels posés par l'ordonnancement sont l'évaluation de la charge de chaque machine, la gestion de la surcharge, le choix du programme à déplacer ainsi que le choix de la machine cible qui va recevoir le programme.

Les décisions pour le placement des charges se feront à l'aide d'informations sur l'état global du système. Pour cela, chaque machine participant à la répartition de charge devra échanger des informations propres à son état..

Le projet devra implémenter des stratégies dynamiques de placement. Le placement est calculé à chaque fois que le programme est lancé, en fonction de l'état global du système. Pour cela, il est nécessaire de trouver des critères d'évaluation de la charge pour chaque machine du réseau. L'équilibrage de charge nécessite le calcul de la charge globale du système. La charge globale sera définie simplement comme la moyenne des charges des machines participant au placement.

Pour la détection de la surcharge, nous commencerons par utiliser des algorithmes à base de seuils fixes puis dynamiques.

Afin d'y parvenir, nous serons amenés à implémenter les deux commandes suivantes :

- start prog arguments
  - Créer un processus exécutant " prog arguments " sur la machine la moins chargée du réseau.
- gps [-l]
  - « Global ps » affiche la liste de tous les processus qui ont été lancés par " start ". L'option –l permet d'afficher un format long (noms exécutable, machine, uid, éventuellement statistiques d'utilisation CPU, mémoire).

et nous avons implémenté gkill car celle-ci est assez complémentaire :

- gkill -sig gpid
  - « Global kill » permet d'envoyer le signal sig au processus identifié par gpid.

Enfin, nous avons choisi d'axer notre projet sur l'insertion/retrait des machines participantes.

MPI

Introduction à MPI

L'API MPI (Message Passing Interface) est une bibliothèque logicielle essentielle pour le développement d'applications parallèles sur des architectures distribuées. Elle offre un ensemble de fonctions standardisées pour la communication par message entre processus.

Caractéristiques de l'API MPI :

- **Langages supportés:** C, C++ et Fortran
- **Fonctionnalité principale:** Communication par message entre processus
- **Environnement:** Distribué, sans partage de mémoire

Chaque processus a son propre flot de contrôle et son propre espace d'adressage. De plus, les sorties standards des processus sont redirigées sur le terminal qui a lancé le programme MPI.

Compilation et exécution d'un programme MPI

Pour compiler un programme MPI, on doit s'assurer que :

- **Fichier header:** le fichier source doit inclure le fichier d'en-tête MPI (`mpi.h`) via la directive `#include <mpi.h>`.
- **Commande de compilation:** utiliser la commande `mpicc` au lieu de `gcc`. La syntaxe est la suivante : mpicc prog_file.c -o name_exec

Pour exécuter un programme MPI on doit utiliser la commande mpirun au lieu de lancer directement notre exécutable. La commande est :

mpirun [–oversubscribe] -np XX [–map-by node] [–hostfile ./hostfile.txt] ./name_exec

Options de la commande `mpirun`:

- `-oversubscribe`: Autorise la surcharge des machines, c'est-à-dire l'allocation de plus de processus que le nombre de cœurs disponibles.
- `-np XX`: Indique le nombre de processus à lancer (remplacez `XX` par le nombre réel).
- `-map-by node`: Indique que la répartition des processus sur les machines du fichier `hostfile` devra respecter une politique Round Robin.
- `-hostfile ./fichier_hostfile`: Indique le chemin d'accès au fichier contenant la liste des machines où les processus seront lancés.

Fichier `hostfile`:

Le fichier `hostfile` est un fichier texte contenant la liste des machines sur lesquelles les processus MPI seront lancés. Chaque machine est indiquée sur une ligne distincte. De plus, nous pouvons ajouter l'option SLOTS permettant d'indiquer le nombre maximum de processus à lancer par machine.

Communication

La communication entre processus se fait à l'aide de primitives MPI au sein d'un communicateur. Parmi les primitives principales nous retrouvons :

- **MPI_Send** : Utilisé pour envoyer des données d'un processus à un autre.
- **MPI_Recv** : Utilisé pour recevoir des données envoyées par un autre processus.
- **MPI_Iprobe**: Permet de vérifier s'il y a des messages en attente dans un communicateur sans effectuer de réception. Cela peut être utilisé pour sonder si un message de certaines caractéristiques est disponible avant de le recevoir.
- **MPI_Probe**: Similaire à MPI_Iprobe, mais il bloque le processus jusqu'à ce qu'un message avec les caractéristiques spécifiées soit disponible. Une fois que le message est disponible, il peut être reçu.

- **MPI_Get_count**: Utilisé pour obtenir le nombre d'éléments dans un message reçu. Cela peut être utile si la taille du message est inconnue à l'avance.

Un communicateur désigne un ensemble de processus pouvant communiquer ensemble, et deux processus ne pourront communiquer que s'ils sont dans un même communicateur. Dans notre cas, le communicateur regroupe l'ensemble des processus que l'on a créés.

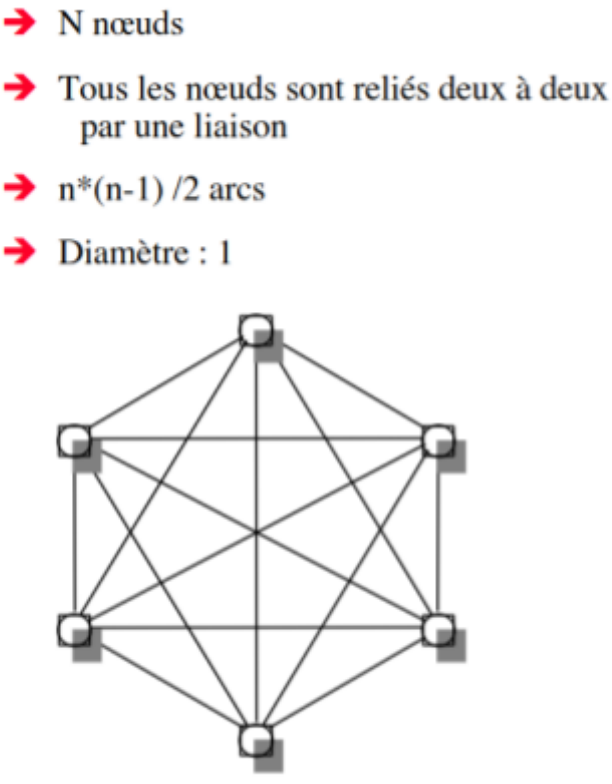
Architecture

Dans le cadre de ce projet, nous avons décidé d'utiliser les machines de la PPTI, en l'occurrence celles de la salle 509. Tous les ordinateurs de cette salle possèdent la même distribution de MPI.

Topologie

Pour notre réseau de machine, nous avons décidé d'utiliser une structure en réseau pair à pair (P2P). Nous avons pour objectif d'exploiter les performances et raccourcir les temps de réponses.

Avec le P2P, nous avons une structure en graphe complet permettant à chaque machine de communiquer les unes avec les autres. On évite ainsi un goulot d'étranglement au niveau de l'aiguilleur, de plus chaque machine peut jouer le rôle d'aiguilleur.



Source : Cours 1, Algorithmique répartie 2023-2024 (Franck Petit)

Configuration SSH

L'API MPI s'appuie sur le protocole SSH pour la communication entre processus. Ce protocole sécurisé nécessite un échange de clés de chiffrement lors de la phase de connexion. On a donc besoin de procéder à certaines étapes afin de pouvoir en profiter sereinement :

1. Génération d'une paire de clés RSA:

- Exécution de la commande suivante pour générer une paire de clés RSA :
  - **ssh-keygen -n id_rsa -t rsa** : cette commande génère deux fichiers dans le répertoire `$HOME/.ssh`: `id_rsa` (clé privée) et `id_rsa.pub` (clé publique).

2. Copie de la clé publique dans le fichier des clés autorisées:

- Copier la clé publique `id_rsa.pub` vers le fichier `authorized_keys` sur chaque machine distante où l'on souhaite exécuter notre programme MPI.
  - **cat** `id_rsa.pub > $HOME/.ssh/ authorized_keys`

3. Démarrage de l'agent SSH:

- Démarrer l'agent SSH pour gérer les clés :
  - **eval 'ssh-agent'** ou **eval "$(ssh-agent)"**

4. Ajout de la clé privée à l'agent SSH:

- Ajout de la clé privée `id_rsa` à l'agent SSH pour éviter de saisir le mot de passe à chaque connexion :
  - **ssh-add** `$HOME/.ssh/id_rsa`

5. Connexion aux machines distantes:

- Se connecter au moins une fois à chaque machine distante du réseau pour enregistrer la clé dans le fichier `known_hosts`.

6. Déploiement d'OpenMpi sur plusieurs noeuds:

- Ajout des lignes suivantes au fichier **~/.ssh/config**, pour permettre à l'agent ssh d'être transmis de noeud en noeud :

```
Host *
        ForwardAgent yes
        StrictHostKeyChecking no
```

Implémentation

Arborescence de notre code

- implem : comprend l'ensemble des fichiers implémentant notre code.
  - balancer.c : code représentant un serveur esclave
  - client.c : code représentant un serveur maître
  - inout.c : ensemble des fonctions en lien avec l'état des serveurs
  - load.c : ensemble des fonctions en lien avec la charge
  - struct.c : ensemble des variables globales manipulées
  - utils.c : fonctions de traitement de l'entrée standard
  - work.c : implémentation des fonctions principales

- simulation : comprend les tests à lancer.
- progs : comprend les programmes exécutés par les serveurs esclaves.

Structure interne d'un serveur

On différencie le serveur maître des serveurs esclaves.

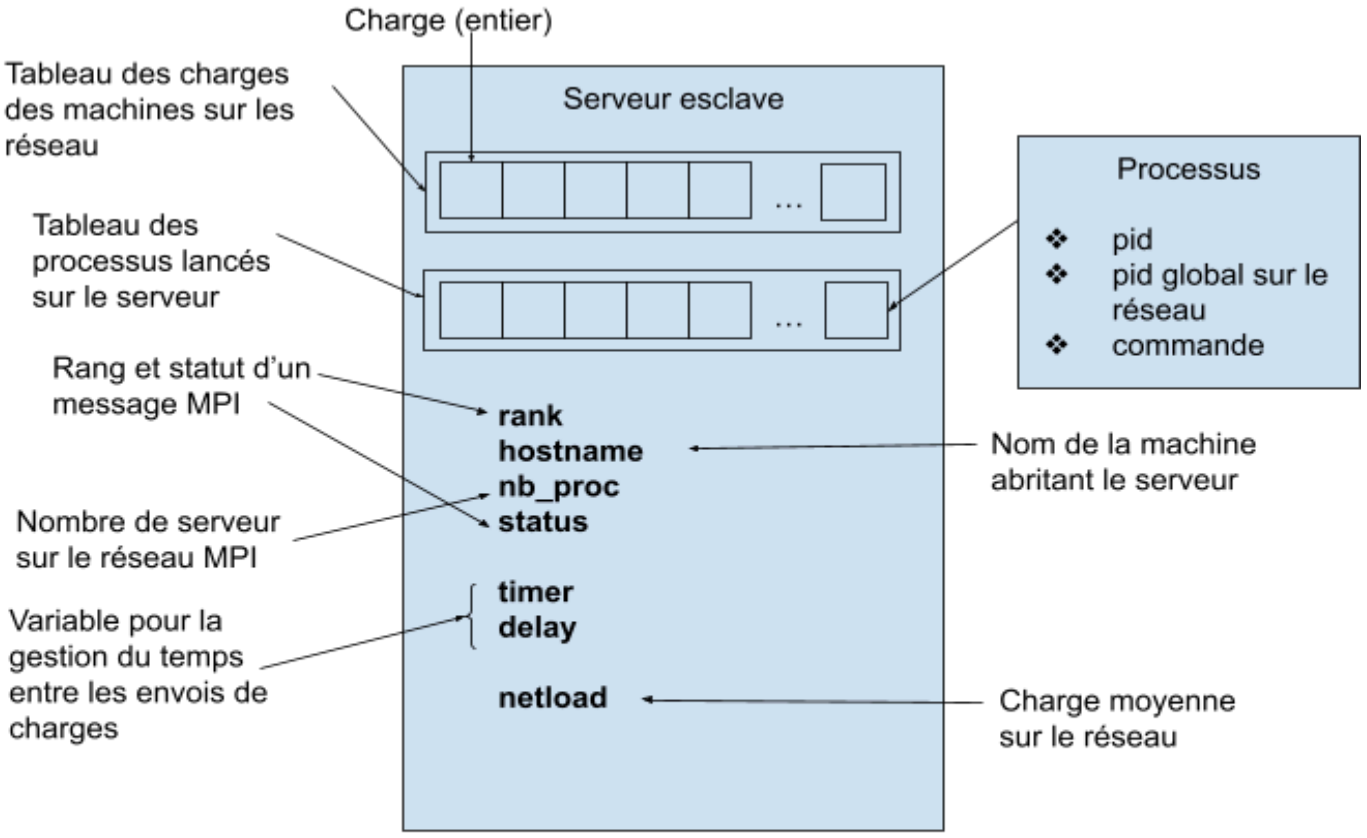


Schéma de la structure interne d'un serveur esclave

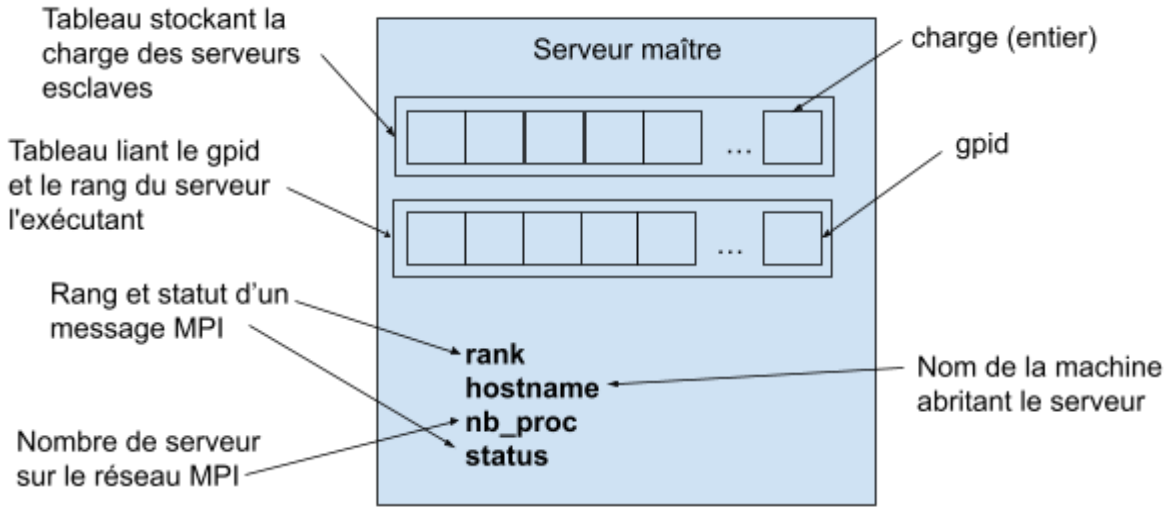


Schéma de la structure interne d'un serveur maître

Gestion de la charge

Charge globale

La charge globale est définie comme la moyenne des charges des machines participant au placement. Notons :

- C, la charge globale,
- N, le nombre de machines participantes,
- Charge_i la charge de la i-ème machine.

La charge globale correspond donc à :

$$C = \frac{\sum_1^{N-1} Charge_i}{N-1}$$

Récupération de la charge

Le load average représente le nombre moyen de processus dans la file d'attente des processus Running (state R) pour, respectivement, la dernière minute, les 5 et 15 dernières minutes.

Dans notre projet, on n'utilisera seulement la première valeur dans le fichier /proc/loadavg.

```
Pseudo-Code de load: retourne la charge globale en pourcentage
```

```
load():
  Ouverture du fichier /proc/loadavg en mode lecture
  Si fichier est nul:
    Retourner erreur
  Lire la première valeur du fichier loadavg et la stocker dans buffer
  Analyser la valeur du buffer en tant que float et la stocker dans load
  Fermer le fichier
  Calculer la charge en pourcentage (load/12) * 100 # 12 = nombre de coeurs
  Retourner le pourcentage s'il est supérieur à 0 sinon 1
```

Mise à jour de la charge

```
Pseudo-Code de publish_load: publie la charge globale actuelle aux autres
machines
```

```
publish_load():
  Récupération du temps actuel
  Si la différence entre le temps actuel et celle de la dernière publication
est supérieur ou égale au délai spécifiée:
    Mettre à jour le temps de la dernière publication dans timer
    Si la machine est endormie:  # idle = 1
      Retourner
    Sinon:
      Récupération de la charge de la machine avec load()
      Stocker la charge dans le tableau workers à l'indice rank
      Pour chaque machine i de 0 à nb_proc exclus:
        Si i est différent de rank:
          Envoyer la charge à i avec le tag LOAD
```

Détection de la surcharge et de la sous-charge

Surcharge

Le seuil de surcharge est un point de référence arbitraire fixé à **45%** au-delà de la charge globale du système. Ce seuil est dynamique, car la charge globale peut varier au cours du temps. **La surcharge d'une machine peut avoir des effets néfastes sur ses performances, sa puissance de calcul, sa consommation de mémoire et son temps de réponse.**

Souscharge

Le seuil de surcharge est un point de référence arbitraire fixé à **55%** en dessous de la charge globale du système. Ce seuil est dynamique, car la charge globale peut varier au cours du temps. **Le fonctionnement en sous-charge d'une machine peut entraîner une utilisation inefficace des ressources, un gaspillage d'énergie et une augmentation des coûts.**

```
Pseudo-Code de Balancer: équilibrer la charge entre les machines
```

```
balancer():
  Mise à jour des informations du réseau avec la fonction update()
  Récupération du temps actuel
  Si la différence entre le temps actuel et celle de la dernière publication
est supérieur ou égale au délai spécifiée:
```

```
    Publication de la charge de la machine locale aux machines du réseau avec
publish_load()
    Récupération de l'état de la machine locale avec state_load()
    Selon son état:
      S'il est égal à 0:
        On ne fait rien, la machine est en équilibre
      S'il est égal à 1:
        Alors la machine est en surcharge et on transfert des tâches avec
work_transfer() à d'autres machines
      S'il est égal à -1:
        Alors la machine est en souscharge et on réclame des tâches avec
claim() à d'autres machines
```

Choix du programme à déplacer et de la machine cible

Lors de la surcharge d'une machine, nous avons choisi de déplacer la tâche ayant le temps CPU le plus élevé. Ce temps est représenté par la somme des variables stime et utime, respectivement les temps CPU en mode S et U.

Pseudo-Code de buzy_proc: détermine le processus le plus chargé de la machine locale

```
buzy_proc():
  Pour chaque processus i allant de 0 à NB_MAX_PROCS exclus:
    Si le pid du processus i est différent de 0, c'est-à-dire s'il est actif:
      Récupérer la charge CPU du processus i avec cpu_proc(pid)
      Si la charge CPU est supérieur ou égale à la charge maximale trouvée
jusqu'à présent:
        Mettre à jour la charge maximale
        Mettre à jour la position du processus le plus chargé
  Retourner la position, dans le tableau des processus, du processus le plus
chargé
```

Maintenant que nous avons récupéré le processus à déplacer, nous pouvons itérer sur le tableau des charges du réseau afin de récupérer la machine la moins chargée.

Pseudo-Code de work_transfer: transfère des tâches vers une autre machine

```
work_transfer(int to, int opt):
  Si opt est égale à 0: # transférer seulement une tâche
    Récupérer la position de la tâche la plus chargée avec busy_proc()
     Si la position est différente de -1:
```

```
      Si to est différent de 0:
        Stocker to dans une variable torank
      Sinon:
        Mettre à jour les informations du réseau avec update()
        Récupérer la position de la machine la moins chargée avec lazyone()
et la stocker dans torank
        Si torank est égale à 0: # aucune machine disponible
          Retourner
      Envoyer la structure de la tâche à la machine spécifiée par torank
      Pour j allant de 0 à procs[pos].size exclus:
        Envoyer procs[pos].commandLine[j] à torank avec le tag CMD
      Terminer la tâche sur la machine locale avec end_task()
      Envoyer des informations sur le transfert à la machine maître
    Retourner

  Sinon: # transférer toutes les tâches
    Envoyer des informations sur le transfert à la machine maître
    Pour chaque processus i allant de 0 à NB_MAX_PROCS exclus:
      Si le processus i est actif: # son pid différent de 0
        Si to est différent de 0:
          Stocker to dans une variable torank
        Sinon:
          Mettre à jour les informations du réseau avec update()
          Récupérer la position de la machine la moins chargée avec lazyone()
et la stocker dans torank
          Si torank est égale à 0:  # aucune machine disponible
            Retourner
        Envoyer la structure de la tâche à la machine spécifiée par torank
        Pour j allant de 0 à procs[pos].size exclus:
          Envoyer procs[pos].commandLine[j] à torank avec le tag CMD
        Terminer la tâche sur la machine locale avec end_task()
  Retourner
```

Commandes principales

start

La commande start prend deux arguments, progs représentant le chemin vers l'exécutable et argument représentant les différents argument nécessaires à l'exécution de l'exécutable. Une fois cette commande tapée, la machine maître récupère l'identifiant de la machine la

moins chargée du réseau et lui enverra progs et les arguments pour permettre à la machine désignée de créer un processus exécutant cette nouvelle tâche.

Pseudo-Code de start: envoi d'une nouvelle tâche à exécuter à la machine la moins chargée du réseau.

```
start(char *argv[], int first, int gpid):
  Mise à jour des informations du réseau avec update()
  Récupérer la position de la machine la moins chargée avec lazyone() et la
stocker torank
  Si torank est égale à 0: # aucune machine disponible
    Retourner 0
  # Test existence fichier
  Ouverture du fichier exécutable
  S'il n'existe pas:
    Retourner 0
  Sinon:
    Fermeture
  #Calculer la longueur de la commande
  Initialisation de size à 0
  Pour i allant de first jusqu'à ce que argv[i] est égal à nul:
    Incrémenter size de 1
  Envoyer size à torank avec le tag START
  Envoyer gpid à torank avec le tag CMD
  #Envoi les éléments de la commande start à la machine désignée
  Pour i allant de first jusqu'à ce que argv[i] est égal à nul:
    Envoi de argv[i] à torank avec le tag CMD
  Retourner 1
```

```
rcv_start():
  Stocker l'expéditeur des messages dans sender
  # Recherche d'une position libre dans le tableau de processus
  Pour i allant de 0 à NB_MAX_PROCS exclus:
    Si procs[i].pid est égale à 0:
      Stocker i dans une variable free_pos
      Sortir de la boucle
  Réception de size venant de sender et allocation de la mémoire nécessaire
pour la commande
  Réception du gpid venant de sender et stockage dans procs[free_pos].gpid
  # Réception de chaque argument de la commande
  Pour i allant de 0 à size exclus:
    Réception de procs[free_pos].commandLine[i] venant de sender
  Si la machine est endormi: # idle = 1
    Démarrer la tâche avec start()
    Libérer la mémoire allouée pour le champ commandLine du processus
free_pos
    Réinitialiser les champs pid, global_pid et size du processus free_pos à
```

```
0
    Retourner
  Sinon:
    Si start_task(free_pos) est égale à 1:
      Envoyer un acquittement d'exécution de la commande réussie à la machine
maître
    Sinon:
      Réinitialiser les champs pid, global_pid et size du processus free_pos
à 0
      Libérer la mémoire alloué pour le champ commandLine du processus
free_pos

  Retourner
```

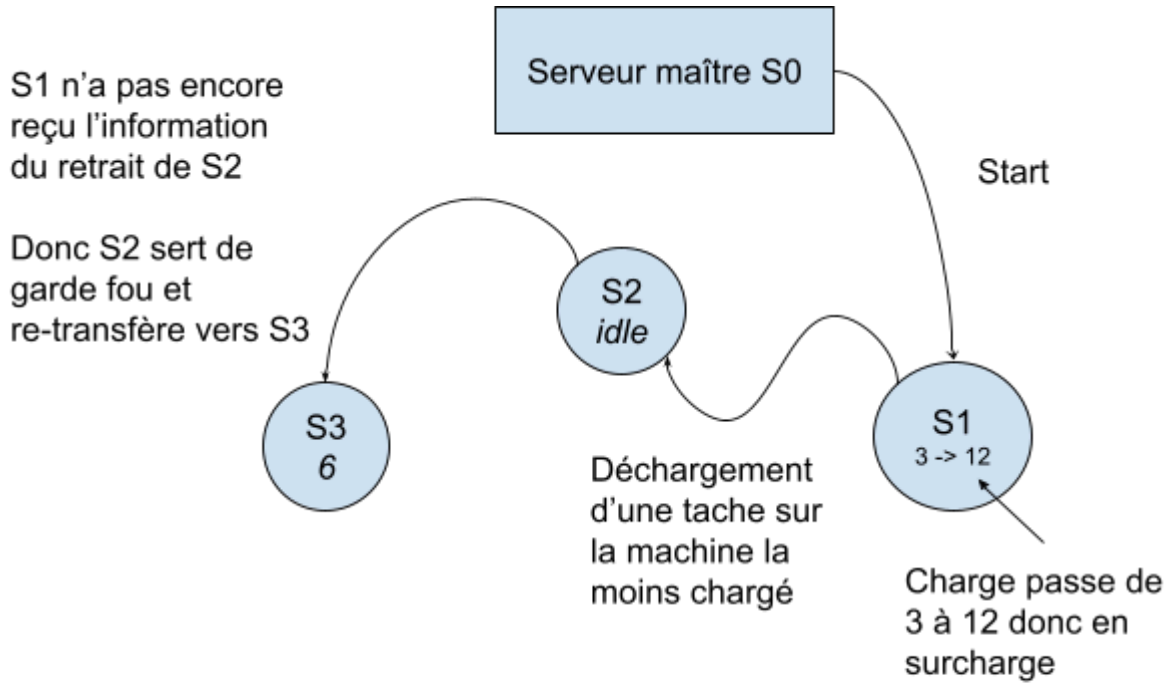


Schéma du procédé de l'instruction start

gps

La commande gps permet d'afficher la liste des processus qui ont été lancés par une commande 'start' et leurs informations. Elle prend également une option -l qui permet d'afficher la liste dans un format long ou non. Si l'option -l n'est pas présente, alors pour chaque processus on affichera: le rang de la machine, le hostname de la machine, son pid, son gpid, le nom de l'exécutable et sa charge. Si l'option est présente, on ajoute à l'affichage son temps d'utilisation CPU, exprimée en ticks horloge, et la mémoire réservée, exprimée en kilobyte (kb).
Pour cela, on a mis au point deux fonctions qui permettent de récupérer ces valeurs dans les fichiers correspondants:

- Pour connaître le temps d'utilisation CPU d'un processus, il faut faire la somme de son temps passé en utilisateur (utime) et son temps passé en système (stime). Chaque processus en cours d'exécution possède un fichier /proc/$pid/stat qui contient les statistiques d'utilisation CPU et où la 14ème et la 15ème valeurs correspondent respectivement à utime et stime, exprimées en ticks horloge.
- Pour connaître la mémoire réservée d'un processus, il faut regarder le resident set size (RSS) qui représente la quantité de données occupées par un processus et contenues dans la mémoire vive. Cette valeur peut être récupérée dans différents fichiers du système mais nous allons l'extraire dans le fichier /proc/$pid/status, dont le format est simplifié pour la compréhension humaine. Elle se situe sur la ligne du champ VmRSS.

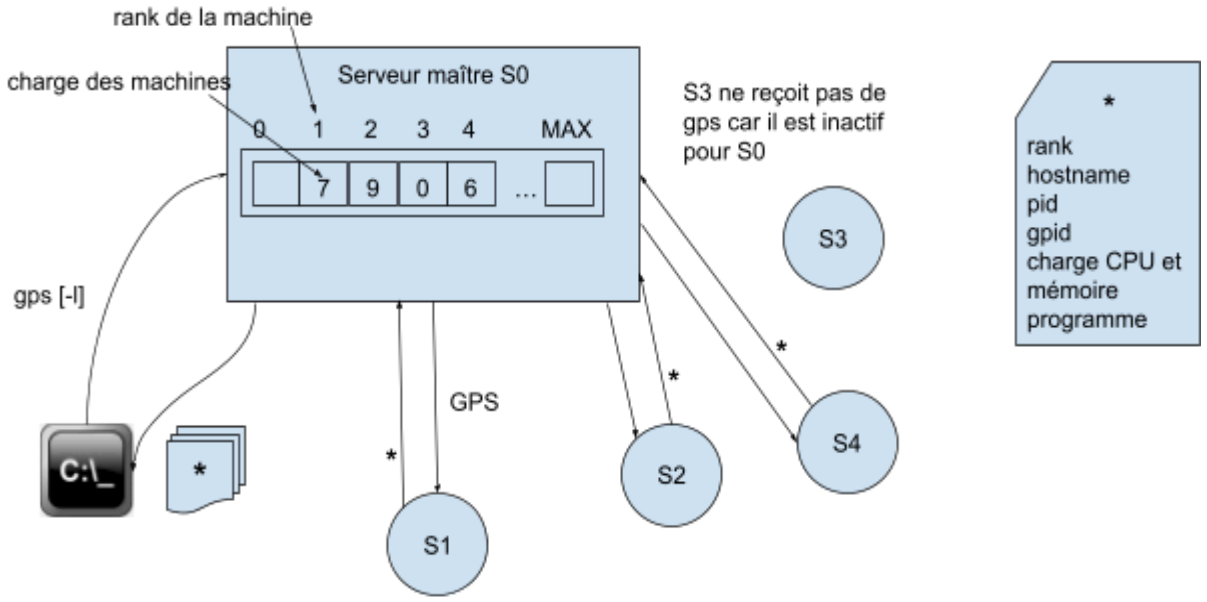


Schéma de la procédure gps

Pseudo-Code de gps: demande au processus l'envoi sur la sortie standard des informations concernant les tâches en cours d'exécution (en lien avec start) sur leur machine.

```
gps(int opt):
  Pour i allant de 1 à nb_proc exclus:
    Si workers[i] est différent de 0:  # charge de la machine i non nul
      Envoyer opt a i avec le tag gps

  Allocation mémoire nécessaire pour le tableau gps_res contenant toutes les
informations pour l'affichage de gps
  Initialisation de nb_gps à 1
  Initialisation de curr à 0
  Initialisation de cpt à 0

  Tant que nb_gps est inférieur ou égal à nb_proc_not_idle:
    Vérification de la présence de messages portant le tag GPS_INFO
    Si c'est le cas:
      Récupération de la longueur du message dans len
      Allouer mémoire nécessaire pour gps_res[curr]
      Réception du message dans gps_res[curr] avec le tag GPS_INFO
      Incrémenter curr de 1
  Vérification de la présence de messages avec tag GPS
  Si c'est le cas:
    Réception du message dans tmp avec le tag GPS
    Incrémenter cpt de tmp
    Incrémenter nb_gps de 1

  Tant que curr est inférieur à cpt:
    Vérification de messages avec tag GPS_INFO
    Récupération de la longueur du message
    Allouer la mémoire nécessaire dans gps_res[curr]
    Réception du message dans gps_res[curr]
    Incrémenter curr de 1

  Affichage de l'en-tête correspondante selon si l'option est activée ou non

  Pour i allant de 0 à gps_res[i] différent de NULL:
    Afficher les informations contenues dans gps_res[i]

  Désallouer gps_res
```

```
rcv_gps(int opt):
  Initialiser cpt à 0

  Si idle est égale à 1:
    Envoyer cpt à la machine maître
    Retourner

  Pour i allant de 0 à NB_MAX_PROCS exclus:
    Si pid du processus i est différent de 0 et si le processus est en cours
d'exécution:
      Incrémenter cpt de 1
      Si opt est égale à 0:  # sans option
        Envoyer les informations nécessaires pour l'affichage à la machine
maître
      Sinon: # avec option
        Récupérer le temps CPU utilisé par le processus i avec cpu_proc(pid)
et le stocker dans une variable cpu
        Récupérer la mémoire utilisée par le processus i avec mem_proc(pid)
et la stocker dans une variable mem
        Envoyer les informations nécessaire pour l'affichage, avec en plus
cpu et mem, à la machine maître

    Sinon:
      Si pid du processus i est différent de 0:
        Terminer la tâche avec end_task(i)


  Si cpt est égale à 0:
    Envoyer qu'il n'y a pas de processus en cours d'exécution à la machine
maître
    Incrémenter cpt de 1
    Envoyer cpt à la machine maître
    Retourner

  Envoyer cpt à la machine maître
```

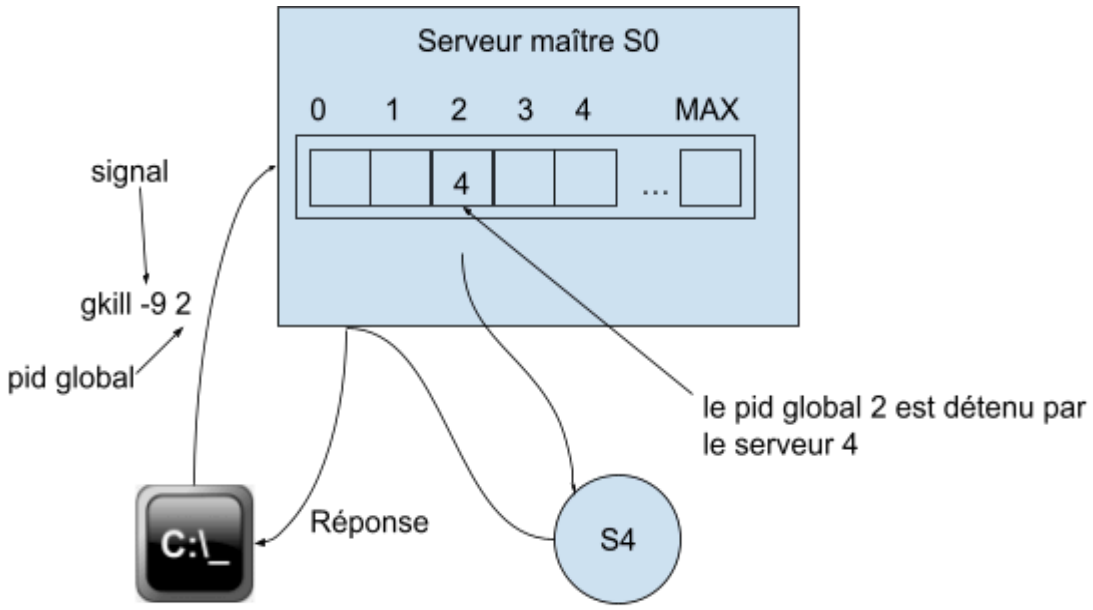
gkill

La commande gkill prend deux arguments, -sig représentant le numéro du signal qu'on souhaite envoyer et le pid du processus à qui on souhaite l'adresser. La machine maître envoie à chaque machine esclave les arguments pid et sig. Ceux-ci vont alors parcourir leur tableau de processus pour vérifier s'il possède bien le processus désigné par la commande. Celui qui le possède mettra fin à son exécution en lui envoyant le signal spécifié et en informera la machine maître.

Pseudo-Code de gkill: envoi du signal sig à la machine détenant la tâche avec le pid global gpid, afin que celui-ci l'envoie directement au processus sur sa machine.

```
killer(int sig, int torank, int pid):
  Initialise res à 0
  Pour chaque machines esclaves i:
    Envoyer pid à i avec tag KILL
    Envoyer sig à i avec tag SIG
  Réception de res venant de la machine concernée par le gkill
  Retourner res
```

```
rcv_kill(int sig, int pid):
  Si la machine est endormie:
    Envoyer 0 à la machine maître avec le tag KILL pour signaler que la terminaison a échoué
    Retourner
  Initialiser pos à -1
  Pour chaque processus sur la machine locale:
    Si son gpid est égale à pid:
      Stocker son indice dans la tableau procs dans pos
      Sortir de la boucle
  Si pos est égale à -1:  # processus non trouvé
    Envoyer 0 à la machine maître avec le tag KILL pour signaler que la terminaison a échoué
    Retourner
  Si le processus est toujours en cours d'exécution:
    Terminer le processus en envoyant le signal sig avec kill(procs[pos], sig)
    Réinitialiser les champs pid, global_pid et size du processus pos à 0
    Libérer la mémoire allouée pour le champ commandLine du processus pos
  Envoyer 1 à la machine maître avec le tag KILL pour signaler que la terminaison a réussi
```



Si la réponse est négative, on retraite du buffer les messages de transfert afin d'avoir le nouveau serveur possédant le processus avec le gpid 2

Schéma de la procédure gkill

insert

La commande insert prend en argument l'identifiant de la machine qu'on souhaite insérer dans le réseau. Pendant l'insertion, on calcule sa charge et, dans la plupart des cas, la nouvelle machine sera en sous-charge et, donc, réclame des tâches à d'autres machines.

Pseudo-Code pour l'insertion: insérer une nouvelle machine dans le réseau

`insert(int to):`
`  Si la charge à la position to dans le tableau workers est égale à 0:`
`  Envoyer workers à to avec le tag INSERT`

`receive_insert():`
`  Initialisation de idle à 0  # marquer la machine comme non endormie`
`  Réception du tableau workers avec le tag INSERT`
`  Récupérer dans loader la charge de la nouvelle machine avec load()`

```
  Stocker dans workers, à la position rank, loader
  Pour chaque machine i allant de 0 à nb_proc exclus:
    Si i est différent de rank:
      Envoyer loader à i avec le tag LOAD
```

remove

La commande remove prend en argument l'identifiant de la machine qu'on souhaite retirée du réseau. Lors du retrait de la machine, l'ensemble des tâches locales devront être transférés et relancés sur d'autres machines.

```
Pseudo-Code pour le retrait: retirer la machine spécifiée du réseau
```

```
remove_worker(int rank_to_remove):
  Initialisation de ul à 1 # signaler la suppression
  Envoyer ul à rank_to_remove avec le tag REMOVE
```

```
receive_remove():
  Appel à la fonction gobacktosleep() qui met la charge de la machine dans le
tableau workers à la position rank à 0 et transfère toutes ses tâches avec
work_transfer() à d'autres machines avant d'être retirer
```

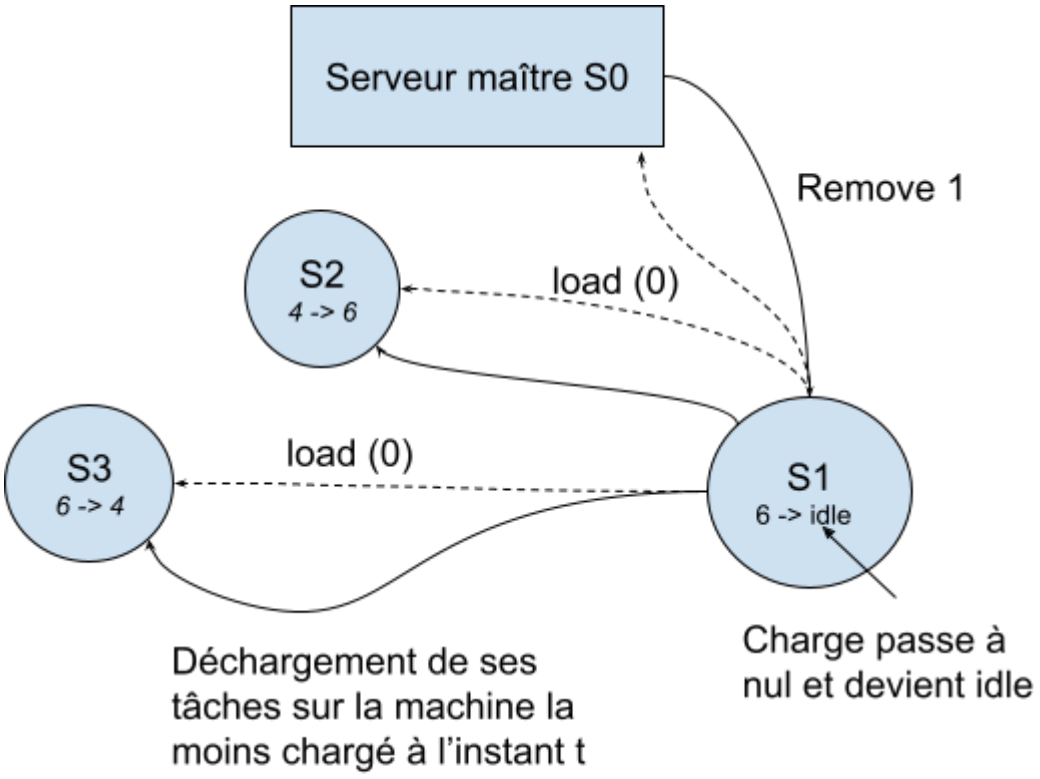


Schéma du procédé de l'instruction remove

Gestion de la réception des messages par le serveur maître

Afin de dissocier la lecture sur l'entrée standard et la réception des réponses éventuelles à nos instructions de l'écoute des messages d'information provenant du réseau de serveur. Nous avons choisi de déléguer ce dernier à un thread créé par le serveur maître. Enfin, les accès aux variables globales tel que **status** utilisé par les primitives MPI sont protégés par un mutex.

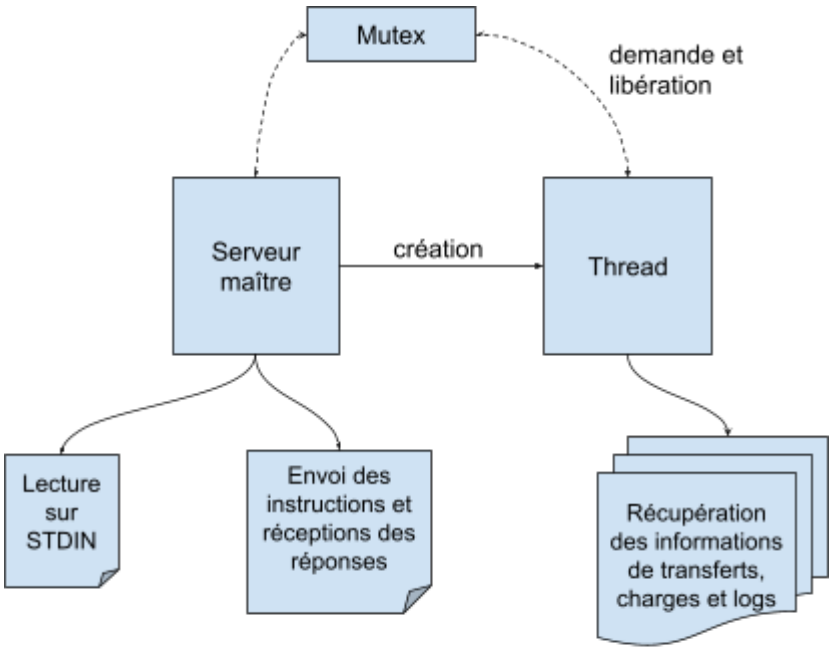


Schéma du CLI, point d'entrée du programme.

Difficultés rencontrées

Les difficultés auxquelles nous avons dû faire face sont inhérentes à tout programme s'exécutant en Pair à Pair. Les deux principales sont le maintien de la connexion SSH avec les machines de la PPTI et le débogage des problèmes d'exécution de notre programme.

Pour le premier point, il n'est pas de notre ressort, nous devons simplement relancer le programme lorsqu'une personne se connecte à une machine du réseau en enlevant de notre hostfile ces mêmes machines. Pour le second, les traces MPI n'étant pas les plus exhaustives, nous avons donc adapté notre architecture afin d'avoir des fonctions plus facilement déboggables.

Conclusion

Ce projet s'inscrit parfaitement dans la continuité des cours suivis cette année. Il s'agit d'une application concrète des connaissances acquises, notamment dans les domaines suivants :

- **Systèmes répartis Client-Serveur:** Cette unité d'enseignement a fourni les bases nécessaires pour la conception et l'implémentation d'une architecture distribuée via notamment la gestion de thread.

- ❖ **Algorithmique répartie:** La découverte de l'API MPI a permis de développer un code optimisé pour le traitement parallèle des données et de réduire le temps de prise en main de celle-ci.
- ❖ **Sécurité et administration des systèmes:** Cette unité d'enseignement a fourni les bases nécessaires à la configuration ssh, comme la génération de clés asymétriques et le chargement de celles-ci en mémoire, pour une exécution du programme plus optimisée sous MPI.

Le projet nous aura également donné une brève vue d'ensemble de ce qui se déroule dans l'industrie, de saisir les défis liés aux performances et les différentes solutions proposées. Il a également éveillé notre intérêt pour les conceptions de systèmes en général : nous nous sommes demandé ce qu'était un CDN, DNS, API Gateway, etc.

Enfin, nous avons mis le doigt sur des points d'amélioration :

- **Documentation:** La documentation du projet pourrait être plus détaillée.
- **Tests:** Un plan de tests plus complet aurait permis de garantir la qualité du code et de faciliter le débogage.
- **Gestion du temps:** La gestion du temps a été un défi, et le projet a été réalisé en respectant le délai imparti, mais sans marge de sécurité.
- **Multithreading :** confier la réception des commandes par les serveurs esclaves à des threads.

Bibliographie

Franck Petit. "MPI:I Une Bibliothèque de communication par messages". Janvier 2024. Disponible sur : https://moodle-sciences-23.sorbonne-universite.fr/pluginfile.php/28541/mod_resource/content/2/02-MPI.pdf

Open MPI. "Open MPI Documentation". Disponible sur : https://www.open-mpi.org/doc/v3.1