

# RAPPORT DE PROJET DE FIN D'ETUDES

*Distribution automatique de l'exécution d'une application paramétrique  
sur les noeuds d'une grille hétérogène*



# PLAN

## **I. Introduction**

## **II. Solutions existantes**

### **1. Nimrod**

- a. Installation**
- b. Installation des agents**
- c. Fonctionnement**

### **2. The AppLeS Parameter Sweep Template (APST)**

- a. Installation**
- b. Utilisation**

## **III. Une solution proposée**

### **1. Mesures des ressources**

- a. Ganglia**
- b. Network Weather Service**

**Installation**

**Utilisation de NWS**

**Utilisation de l'API NWS**

### **2. Stratégie de choix dynamique de distribution des tâches aux esclaves**

### **3. Tests**

**Test 1**

**Test 2**

**Test 3**

**Test 4**

### **4. Conclusion des tests**

### **5. Limites et évolutions possibles**

## **IV. Conclusions et perspectives**

## **V. Bibliographie**

# I. Introduction

Le contexte dans lequel nous nous sommes placés pour évaluer les logiciels existants est le suivant:

Une requête sur une base de données distante nous amène en entrée un flot discontinu de données. Ces données sont découpées en plaques numérotées de tailles égales sur lesquelles des calculs doivent être effectués avant de les envoyer en sortie. Le temps de traitement étant long, les calculs sont parallélisés sur un ensemble de machines esclaves constituant une grille.

Les plaques sont placées dans une zone de mémoire partagée avant d'être réparties par un programme distributeur aux différentes machines esclaves disponibles qui effectuent les calculs puis renvoient à la machine "maître" une plaque de taille fixe résultat. Les plaques résultats sont récupérées par le maître dans une zone de mémoire partagée. Les plaques sont réordonnées selon leur numérotation en sortie de cette zone.

## II. Solutions existantes

### 1. Nimrod

#### a. Installation

L'installation de nimrod peut se faire par un simple utilisateur la documentation du site <http://www.csse.monash.edu.au/~nimrod/nimrodg/ng.html> permet de le faire sans trop de problèmes.

Avant de faire l'installation, il faut s'assurer que python et Postgresql sont installés.

Une fois l'installation finie, il faut créer la base de données Postgresql qui servira à Nimrod. Pour cela, des scripts d'initialisation sont fournis, mais avant cela, il faut avoir un serveur postgresql. Voici les commandes que nous avons utilisé sur la machine poste1 qui servait de maître:

```
initdb -D /.../repertoireDB
```

Cette première commande servait à créer le répertoire utilisé par postgresql pour sauvegarder les informations de la base de données sur le disque dur.

```
postmaster -D /.../repertoireDB >log 2>&1 &
```

Cette seconde commande démarre le serveur qui utilise le répertoire précédemment créé.

Ensuite, les scripts de nimrod permettent de créer les tables dans la base de données: *nimrod dbcreate*.

**Attention:** il est préférable d'installer Nimrod/g à partir des sources plutôt que des binaires linux fournis sur le site, car ces derniers n'étaient pas compatibles.

#### b. Installation des agents

Pour exécuter une tâche sur une ressource, Nimrod envoie un petit programme appelé "agent" qui servira à piloter la tâche. Par défaut, lors de notre installation de nimrod, celui-ci définissait dans la base de données l'utilisation possible de plusieurs agents pour différentes architectures, mais aucun de ceux-ci n'étaient installés. Il faut donc installé soit même les agents dont on a besoin. Les fichiers binaires d'agents pour les architectures les plus courantes sont fournies sur le site de nimrod (<http://www.csse.monash.edu.au/~nimrod/nimrodg/ng.html> -> Downloads -> Download Agent Binaries) mais il est possible de compiler des agents pour ses propres ressources à partir des sources. Un agent doit être compilé pour chaque architecture

matérielle (par exemple x86) et chaque système d'exploitation (par exemple linux kernel 2.6). Pour les machines poste[1-9] nous avons recompilé un agent afin qu'il soit compatible avec le noyau 2.6.

L'installation d'agents est expliquée en détail sur le site de nimrod: <http://www.csse.monash.edu.au/~nimrod/nimrodg/ngagent.html>.

Le script à modifier pour enregistrer les agents dans la base de données est : `$NIMROD_INSTALL/libexec/nimrod_agents`.

## c. Fonctionnement

La base de données postgresql est utilisée pour enregistrer toutes les informations sur les ressources, les tâches à effectuer ou en cours, ainsi que sur l'état de Nimrod. La création d'une expérience de plusieurs tâches à affecter à plusieurs ressources se fait en 3 étapes:

- enregistrement des ressources
- définition de l'expérience et création des tâches
- affectation des ressources à une d'expérience

\*Enregistrement des ressources:

cette étape consiste à fournir à Nimrod la liste des machines disponibles ainsi que le moyen d'y accéder.

```
nimrod resource add globus poste2.int-evry.fr/jobmanager
```

La commande précédente permet de créer une ressource accessible à l'adresse `poste1.int-evry.fr` sur laquelle on peut lancer des commandes par `globus`. Nous devons ensuite indiquer à Nimrod quelle est l'architecture matérielle de la ressource. Celle-ci doit correspondre à l'architecture d'un agent disponible (voir l'installation des agents ci-dessus). La liste des architectures supportées est obtenue par la commande: *nimrod portalapi getarch*.

On tape alors la commande suivante: *nimrod portalapi setarch globus poste2.int-evry.fr/jobmanager x86-linux2.6*.

Cette étapes est à répéter pour chaque ressource.

\*définition de l'expérience:

Chaque expérience pour nimrod doit avoir un sous-répertoire de `$HOME/.nimrod/experiments/` qui va contenir les informations pour créer les tâches. Le tutoriel <http://www.csse.monash.edu.au/~nimrod/nimrodg/tutorial.html> donne un exemple pour créer une expérience. Le principe est de définir les action dans un fichier plan (ex: *ngdemo.pln*) puis de générer un fichier d'exécution par la commande "*nimrod generate ngdemo.pln*". L'expérience composée des différentes tâches est alors prête à être ajoutée à la base de données par la commande: "*nimrod portalapi addrun ngdemo*".

\*affectation des ressources à une expérience:

avant de lancer l'exécution, il faut indiquer à Nimrod quelles sont les ressources, parmi toutes celles qui ont été définies à la première étape, qu'il peut utiliser pour exécuter l'expérience. Pour chaque ressource à utiliser, on exécute la commande suivante: "*nimrod portalapi addserver ngdemo poste2.int-evry.fr/jobmanager globus*"

Une fois toutes ces étapes réalisées, on peut (enfin) lancer l'expérience: "*nimrod portalapi startexp ngdemo*".

## 2.The AppLeS Parameter Sweep Template (APST)

AppLeS propose différentes méthodes d'heuristique afin de répartir les tâches parmi les esclaves. Chacune de ces méthodes repose sur le calcul d'une approximation du temps que prendra l'exécution de la tâche sur chaque machine. Le temps calculé inclut le temps de transfert des divers arguments d'entrée et de sortie par le réseau. A partir de ces évaluations de temps, trois principaux comportements de répartition des tâches sont possibles:

\*minmin:

On choisit en premier le couple tâche/machine qui finira le plus rapidement. L'idée est que si on renvoie un résultat le plus rapidement possible, cela augmentera les performances globales.

\*maxmin:

On associe les machines les plus rapides avec les tâches les plus longues à exécuter.

\*sufferage:

On compare pour la différence de temps entre l'exécution la plus rapide et la deuxième plus rapide, pour chaque tâche. Et on choisit la tâche qui serait la plus retardée (qui a donc la plus grande différence de temps), pour l'exécuter sur la machine qui lui permet d'être le plus rapidement exécutée.

Ces trois méthodes sont plus efficaces qu'une simple attribution des tâches dans l'ordre où elles arrivent, mais elles supposent toutes les trois que l'on puisse prédire le temps que prendra l'exécution de la tâche. AppLeS utilise pour cela le rapport entre le coût (donné par l'utilisateur à l'aide d'un argument 'COST' défini pour chaque tâche) et le temps mis précédemment par une autre tâche sur chaque machine. La répartition des tâches repose donc uniquement sur le rapport COST/durée d'exécution. Si l'utilisateur ne donne aucune indication de la valeur de COST, il est impossible pour AppLeS de faire une répartition convenable des tâches. Ce problème est important ici, car les hypothèses choisies dans le cadre de ce projet, ne permettent pas d'indiquer une valeur pour COST.

### **a. Installation**

L'installation d'AppLeS est très simple, il suffit de télécharger les sources à l'adresse: <http://grail.sdsc.edu/projects/apst/download.html>. Ensuite, les commandes standards *./configure*, et *make install --PATH=prefix* suffisent.

### **b. Utilisation**

L'utilisation d'AppLeS se fait en ligne de commande, par l'utilisation de fichiers xml. AppLeS fonctionne sous la forme d'un démon, (apstd) qui est démarré avec comme argument, entre autres, un fichier xml décrivant les ressources à utiliser. Les tâches sont passées au démon à l'aide d'un client (apst) avec comme argument, l'action à effectuer et les arguments propres à l'action (par exemple: *apst add taches.xml*). Le fichier xml définit les tâches une à une, avec la liste des entrées et sorties ainsi que l'exécutable.

# III. Une solution proposée

## 1. Mesures des ressources

### a. Ganglia

L'installation de Ganglia commence comme suit:

```
./configure  
make  
make install  
./gmond
```

Les problèmes commencent ici car démarrer le démon de supervision ganglia requiert les droits super utilisateur.

Ensuite, on récupère les données sur les machines sur lesquelles tourne un démon au format XML par la commande:

```
telnet localhost 8649
```

L'utilisation de Ganglia nécessite donc une couche de manipulation du XML pour en extraire les informations nous intéressant.

Les difficultés d'installation et d'exploitation comparées à celles de NWS nous ont fait abandonner ce logiciel.

### b. Network Weather Service

#### ***Installation***

Voici les étapes de l'installation de NWS sur les machines de la salle B313:

```
tar -xvf nws-2.10.1.tar  
cd nws-2.10.1  
configure --prefix=~/.nws  
make all  
make install  
make install
```

Le deuxième *make install* a été utile car le premier a échoué pour une raison inconnue, et le simple fait de relancer la commande a permis de continuer l'installation.

Une fois l'installation terminée, le fichier *nws-hostadmin* situé dans le répertoire *~/.nws/bin/* contenait une petite erreur, il fallait enlever le caractère "" au début de la ligne 169. Cette modification effectuée, NWS peut fonctionner.

#### ***Utilisation de NWS***

NWS fonctionne avec un serveur qui centralise les différents services NWS ainsi que des petits programmes clients (appelés **sensors**) qui tournent sur chaque machine dont on veut analyser les ressources. Le serveur est composé de deux démons, un pour enregistrer les adresses des machines clientes et les services disponibles, et un pour enregistrer les informations obtenues depuis les sensors. Ces deux démons sont appelés **nameserver** et **memory server**.

Pour faire fonctionner NWS, il faut donc déployer l'ensemble de ces démons sur les machines, ainsi que définir les activités (par exemple, observation de la bande passante entre deux machines) qui seront utilisées.

Un outil de déploiement est fourni avec la version de NWS que nous avons utilisé, mais celui-ci ne fonctionnait pas de façon sûre, nous ne l'avons donc pas utilisé. Le nom du script de l'outil est "*nws\_hostadmin*", nous donnons dans la suite les commandes qu'il faut taper pour utiliser cet outil, dans le cas où une version suivante de NWS fonctionnerais mieux.

Afin de déployer les divers démons, nous avons utilisé les commandes NWS associées à chacun. Voici un exemple de déploiement sur les machines de la salle B313:

Lancement du serveur de nom:

la commande suivante est à saisir sur la machine qui servira de serveur (ici *poste1*):

```
nws_nameserver -e $HOME/NWSdir/nameserver.err -f $HOME/NWSdir/registrations -l $HOME/NWSdir/nameserver.log &
```

```
( ou nws-hostadmin start -b ~/projet/nws/bin -s '-e $HOME/NWSdir/nameserver.err -f $HOME/NWSdir/registrations -l $HOME/NWSdir/nameserver.log' poste1.int-evry.fr:nameserver)
```

Les arguments *-e*, *-f* et *-l* servent à rediriger les sorties et les logs du démon. Nous avons choisi de les placé dans un répertoire spécifique: *mkdir ~/NWSdir* .

Lancement du serveur de mémoire:

Les commandes suivantes sont à saisir aussi sur la machine qui sert de serveur. Il est possible d'utiliser une machine différente du serveur de nom, mais pour des raisons de simplicité, nous ne le faisons pas.

```
mkdir ~/NWSdir/memory
```

```
nws_memory -d /tempo/lisiecki/NWSdir/memory -e /tempo/lisiecki/NWSdir/memory/err -l /tempo/lisiecki/NWSdir/memory/log &
```

```
(ou nws-hostadmin start -b ~/projet/nws/bin -s '-d /tempo/lisiecki/NWSdir/memory -e /tempo/lisiecki/NWSdir/memory/err -l /tempo/lisiecki/NWSdir/memory/log' poste1.int-evry.fr:memory)
```

Afin de pouvoir lancer les commandes NWS qui suivent, il est utile d'enregistrer dans le fichier *\$HOME/.nwsrc*, les adresses du serveur de nom et de mémoire. Pour cela, il faut copier les deux lignes:

```
"NAME_SERVER = poste1.int-evry.fr:8090
```

```
SENSOR_MEMORY = poste1.int-evry.fr:8050"
```

dans le fichier *~/nwsrc*. Si cela n'est pas fait, les commande suivantes devront avoir deux arguments supplémentaires ("*-N poste1.int-evry.fr -M poste1.int-evry.fr*") afin d'indiquer les adresses de ces serveurs.

Lancement des sensors:

Pour chaque machine sur laquelle on désire faire des mesures, il faut taper les commandes suivantes.

On crée un répertoire pour chaque sensor:

```
mkdir -p ~/NWSdir/sensor/poste2
```

puis on se connecte que la machine:

```
ssh poste2
```

pour lancer la commande du démon:

```
nws_sensor -e ~/NWSdir/sensor/poste2/err -l ~/NWSdir/sensor/poste2/log &
```

```
(ou nws-hostadmin start -b ~/projet/nws/bin -s '-e ~/NWSdir/sensor/poste1/err -l
```

~/NWSdir/sensor/poste1/log' poste1.int-evry.fr:sensor)

Le lancement des démons sensors permet de mesurer les informations relatives au CPU de la machine, pour ce qui est de la mémoire ou de la bande passante, il faut demander à NWS de lancer des "activités" supplémentaires.

Voici un exemple pour faire des mesures de la mémoire d'un machine:

```
start_activity      poste2      name:memoire_poste2      controlName:periodic
skillName:memoryMonitor period: 25
```

Voici un exemple permettant les mesures de la qualité du réseau entre 2 machines:

```
start_activity poste1 name:tcp_poste1_2 controlName:clique skillName:tcpMessageMonitor
member: poste1.int-evry.fr member: poste2.int-evry.fr period: 20
```

Pour obtenir les mesures faites par NWS, il est facile d'utiliser l'outil *nws\_extract*. La commande prend deux arguments principaux: la compétence que l'on veut observer et la machine à observer. Par exemple, la disponibilité du CPU de la machine poste2 peut être vue avec:

```
nws_extract -f measurement availableCpu poste3
```

Chaque activité lancée précédemment est arrêtée par la commande:

```
halt_activity nom_activite
```

Les démons peuvent être quittés de la façon suivante:

```
nws_ctrl halt posteX
```

## Utilisation de l'API NWS

Comme on a pu le voir dans le fonctionnement d'apples, l'utilisation d'un indicateur de l'état des ressources (machines, réseaux, ...) est nécessaire pour répartir les tâches. Dans le cadre de ce projet, nous avons choisi d'utiliser le programme NWS (Network Weather Service) car celui-ci permet de faire des mesures à la fois sur le CPU et sur l'état du réseau.

Afin de pouvoir utiliser les mesures fournies par NWS dans les fonctions de distribution des tâches, nous avons utilisé l'API (application program interface) NWS. La programmation à partir de cette API se fait par l'utilisation de la librairie *libnws.a* ainsi que des en-têtes *nws\_api.h* et *nws\_forecast\_api.h* fournis avec les sources de *nws*. La description de toutes les fonctions fournies par l'API est disponible sous la forme d'une page de manuel linux, fournie aussi dans les sources de NWS.

Voici un exemple de fonction utilisée pour déterminer la disponibilité du CPU d'une machine de nom 'host':

<pre>#define NWSAPI_SHORTNAMES #include "nws_api.h" #include &lt;math.h&gt; #include &lt;string.h&gt; char *servername;</pre>	<pre>NWSAPI_SHORTNAMES      permet d'utiliser les noms de fonctions nws_api abrégés.</pre>
---	--

<pre> int availableFreq(char *host){     HostSpec * NS;     Measurement mesure;     int freq_cpu;     char * series;     SeriesSpec *spec;      /* definition du serveur de nom NWS */     NS      =(NWSAPI_HostSpec*) MakeHostSpec(servername, 8090);             UseNameServer((const NWSAPI_HostSpec*) NS);      /* get the series name */     spec =(SeriesSpec *) MakeSeriesSpec ((const char*) host, "", "availableCpu");     series = (char *) SeriesName((const SeriesSpec *) spec);      if (!GetMeasurement(series, &amp;mesure))      {         fprintf(stderr,"failed to retrieve measurement (nws is down?)\n");         return 1;     }      freq_cpu=freq(host);     return (int)(mesure.measurement * freq_cpu); } </pre>	<p>'host' contient le nom du client recherché.</p> <p>'NS' : serveur de noms</p> <p>'mesure': structure de mesures</p> <p>'series': définition de la mesure</p> <p>'spec': définition des paramètres de la mesure</p> <p>'servername' est une variable globale contenant le nom du serveur de nom. On suppose ici, qu'on peut le contacter sur le port standard NWS: 8090</p> <p>Les deux lignes permettent de définir le serveur de nom à utiliser pour les fonctions qui suivront.</p> <p>Ces deux lignes permettent de définir le type de la mesure (« availableCpu ») ainsi que la machine à observer ('host', passée en argument).</p> <p>La fonction 'GetMeasurement()' demande auprès du serveur de nom, la dernière mesure définie par 'series', et écrit le résultat dans la variable 'mesure'.</p> <p>En cas de problème (NWS ne répond pas, la mesure demandée n'existe pas, ...) on affiche un message d'erreur, et on renvoi une valeur 1. La valeur renvoyée par la fonction availableFreq est le pourcentage de CPU disponible multiplié par la fréquence du CPU. Une valeur de 1 est donc une valeur très faible, ce qui permet de défavoriser une machine dont on ne peut savoir l'état.</p> <p>int freq(char * host) est censée donner la fréquence en Mhz de la machine donnée.</p> <p>On renvoie enfin le produit fréquence par pourcentage disponible.</p>
---	---

Pour obtenir la bande passante entre le maître et un esclave donné, une fonction similaire est utilisée (voir code source en annexe). La principale différence se situe au niveau de la définition de la mesure:

```
spec =(SeriesSpec *) MakeSeriesSpec(servername, (const char *) host, "bandwidthTcp");
```

On définit cette fois-ci la mesure de « bandwidthTcp » entre les machines 'servername' et 'host'. On suppose ici, que la machine maître est aussi la machine faisant office de serveur de nom.

Les valeurs renvoyées par la mesure de la bande passante sont des valeurs instantanée qui

peuvent donc varier assez rapidement. Pour éviter les erreurs provoquées par une mesure temporairement éloignée de la réalité, nous faisons une moyenne des cinq dernières mesures effectuées. On utilise pour cela la fonction *GetMeasurements(char \*series,double date\_debut,Measurement \* mesure,int n,int \*nombre\_renvoyé)* qui donne au maximum les 'n' dernières mesures:

```
GetMeasurements(series,0, mesure,5,&n)
```

Bien que l'on utilise cinq mesures, la dernière est censée être la plus représentative de l'état actuel du réseau. On fait donc une moyenne pondérée des mesures en attribuant un coefficient décroissant de 5 à 1 en fonction de l'ancienneté.

## **2. Stratégie de choix dynamique de distribution des tâches aux esclaves**

L'algorithme statique initialement utilisé par AIPE, classe les machines dans l'ordre décroissant de leur fréquence CPU pour déterminer l'esclave à privilégier. Le problème est que cette méthode ne tient pas compte de l'évolution de l'état de la charge des machines et du réseau.

On parcourt l'ensemble des machines inoccupées. Pour chacune on relève sa fréquence cpu disponible. Si cette dernière dépasse le seuil FREQLIMIT (défini à 200 MHz pour les tests), on s'intéresse à sa bande passante avec le maître disponible. On retient les valeurs du premier esclave disponible. Pour le suivant, on regarde si sa fréquence disponible est supérieure. Si c'est le cas et que la bande passante disponible n'est pas trop inférieure, ou si la fréquence est légèrement inférieure mais que la bande passante est largement meilleure, alors on garde les valeurs de ce nouvel esclave et on le sélectionne.

```
if( (FreqTemp > Freq && bwTemp > 0.8 * bw)
    || (FreqTemp > 0.9 * Freq && bwTemp > 3 * bw)){
    Freq=FreqTemp;
    bw=bwTemp;
    Slave=iSlave;
}
```

Une fois tous les esclaves parcourus ainsi, on envoie la tâche au dernier sélectionné.

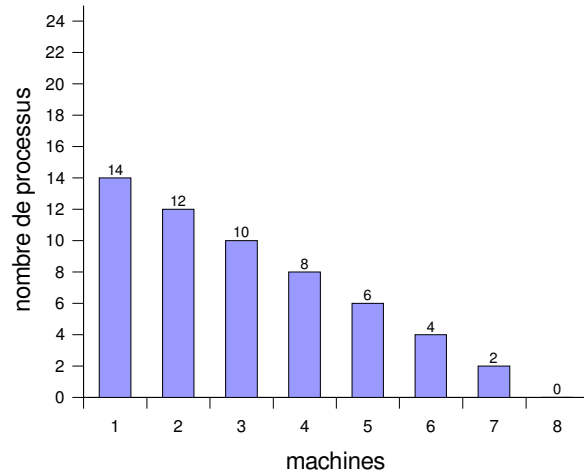
## **3. Tests**

Dans tous nos tests, le flot de données en entrée est continu. Le logiciel utilisé est AIPE (Automatic Integration for Parallel Execution), en cours d'évolution dans le département parallélisme de l'Institut National des Télécommunications, dont nous avons modifié la manière de choisir l'esclave à qui envoyer la tâche courante. Ce logiciel utilise la bibliothèque de communication MPI. Les tests tentent de mettre en évidence l'amélioration de l'algorithme statique par un algorithme dynamique.

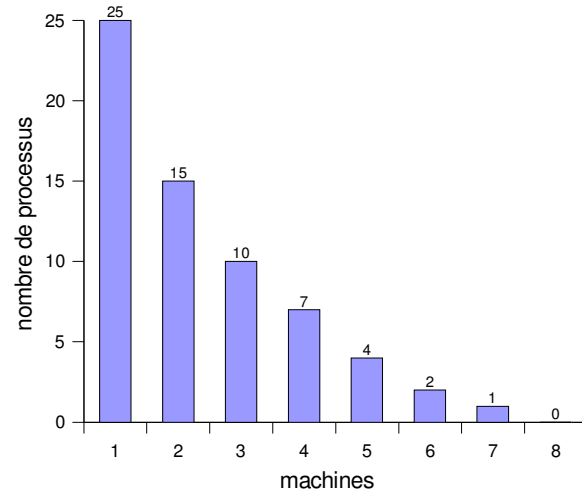
Les données sont des matrices et la tâche demandée est l'élévation à une puissance donnée d'une matrice. Les tâches sont donc assez homogènes (les matrices sont générées aléatoirement dans un premier temps puis toutes identiques, remplies par des 3).

Les tests ont été effectués sur un ensemble de 9 machines homogènes de fréquence CPU 2800 Mhz reliées par un réseau Ethernet à 100 Mbps. Le maître choisi distribue les tâches uniquement aux huit autres machines.

Pour simuler une hétérogénéité des ressources, nous avons adopté comme stratégie de lancer sur les esclaves un nombre variable de processus utilisant du temps CPU. Pour se placer dans des conditions favorables à l'algorithme dynamique, nous avons chargé les machines selon leur ordre dans la liste utilisée par l'algorithme statique. Ainsi, sur la première machine de cette liste tournait le plus grand nombre de ces processus et sur la dernière, aucun. Dans un premier temps (hétérogénéité n°1), le nombre de processus décroissait linéairement de 14 à 0 avec un pas de 2, pour les 8 esclaves. Et dans un second temps (hétérogénéité n°2), il décroissait de manière plus logarithmique: 25, 15, 10, 7, 4, 2, 1, 0.



hétérogénéité n°1



hétérogénéité n°2

## Test 1

**Nombre de tâches:** 100

**Taille matrice:** 10x10

**Exponentiation:** 2

**Hétérogénéité:** non

**Temps total:**

**Algorithme statique:** 15.7s

**Algorithme dynamique:** 23.6s

Pour ce test (calcul petit grain) la répartition des tâches dégrade les performances. Cependant, l'algorithme dynamique donne des performances médiocres par rapport à l'algorithme statique.

**Interprétation:**

Les machines étant homogènes, notre algorithme perd du temps à chercher le meilleur esclave pour chaque tâche alors que l'algorithme statique l'envoie directement à l'esclave suivant dans sa liste.

## Test 2

**Nombre de tâches:** 100

**Taille matrice:** 10x10

**Exponentiation:** 2

**Hétérogénéité:** n°1

**Temps total:**

**Algorithme statique:** 34.6s

**Algorithme dynamique:** 31.9s

Pour ce test (calcul petit grain, encore) les performances des deux algorithmes sont comparables avec un léger avantage pour l'algorithme dynamique. On remarque que l'algorithme statique amène une répartition plus homogène des tâches alors que l'algorithme dynamique envoie beaucoup plus de tâches aux machines performantes qu'aux autres.

**Interprétation:**

Le temps perdu à chercher le meilleur esclave est regagné parce que l'algorithme statique envoie d'abord des tâches aux machines les plus chargées et que le temps de calcul de chaque tâche est négligeable devant le temps d'envoi d'une matrice, du maître à l'esclave.

## Test 3

**Nombre de tâches:** 100

**Taille matrice:** 100x100

**Exponentiation:** 100

**Hétérogénéité:** n°1

**Temps total:**

**Algorithme statique:** 3m06s

**Algorithme dynamique:** 3m02s

Pour ce test (calcul gros grain), les performances des deux algorithmes sont comparables avec toujours le même avantage de quelques secondes pour le dynamique alors que le traitement total est environ six fois plus long. La répartition des tâches selon les esclaves est quasiment la même.

**Interprétation:**

Le calcul sur une tâche est ici plus long que les envois de matrices. Au début, chaque machine reçoit une tâche puis le maître attend que la plus rapide se libère pour lui envoyer la tâche suivante. Ainsi il n'y a pas de différences entre les deux algorithmes jusqu'à la fin du traitement.

## Test 4

**Nombre de tâches:** 50

**Taille matrice:** 201x201

**Exponentiation:** 300

**Hétérogénéité:** n°2

**Temps total:**

**Algorithme statique:** 21m36s

**Algorithme dynamique:** 16m13s

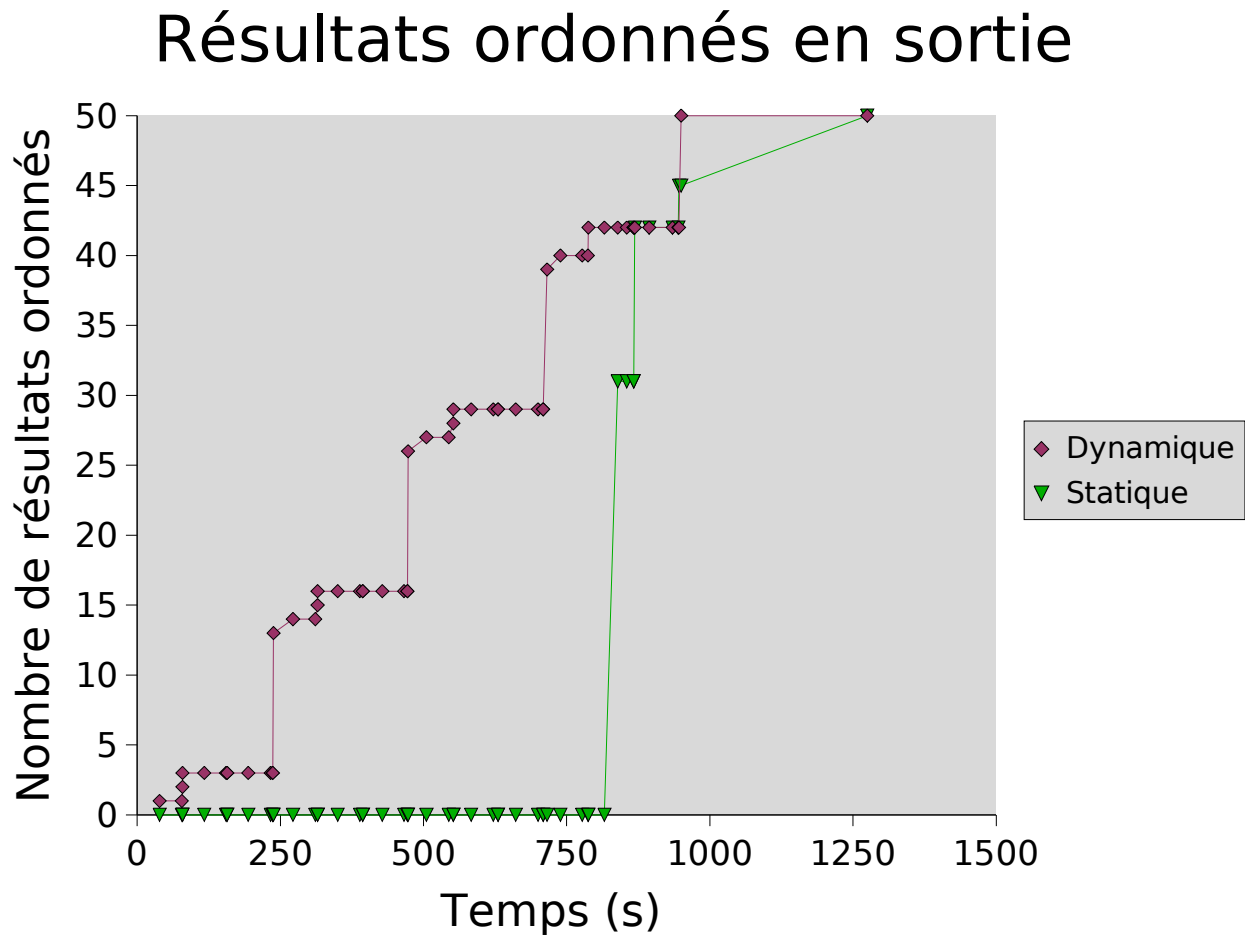
Pour ce test, aussi de calcul gros grain, on a récupéré l'heure d'arrivée de chaque résultat. On a aussi utilisé une valeur limite de fréquence en dessous de laquelle on ne distribue pas de tâche à un esclave. On remarque une distribution des tâches très hétérogène pour l'algorithme dynamique car seuls les esclaves les plus performants travaillent. On voit que la distribution des tâches a permis de gagner 5 minutes sur les 21 de l'algorithme statique.

**Interprétation:**

Avec l'algorithme statique, on remarque qu'une tâche est distribuée à l'esclave le moins

performant (le premier dans la liste) et deux tâches au deuxième de la liste. Et c'est la deuxième tâche de ce dernier qu'on attend pendant 5 minutes. Or, si cette tâche avait été distribuée au dernier esclave (le plus performant), elle aurait été effectuée en un temps de l'ordre de 40 secondes.

Les résultats, une fois retournés par les esclaves doivent être réordonnés avant de poursuivre leur traitement. On se rend compte alors que l'attente d'un résultat bloque tous ceux qui le suivent. Pour cela, nous avons tracé le graphe représentant le nombre de résultats ayant pu être délivrés en fonction du temps:



Dans le cas de l'algorithme statique, la première tâche étant distribuée à l'esclave le moins performant, les trente tâches suivantes sont bloquées, et pendant 839 secondes, aucun résultat n'est délivré. On s'est placé dans un modèle de pipeline, il est donc beaucoup plus intéressant de recevoir chaque résultat dans l'ordre le plus rapidement possible. L'algorithme dynamique, comme le schéma ci-dessus le met en évidence, est sur ce point beaucoup plus efficace.

## Conclusion des tests

Pour que l'algorithme dynamique soit utile, il faut que les calculs soient de type gros grain (le temps de communication est négligeable devant le temps de calcul), et que les ressources soient hétérogènes.

Les conditions de nos tests font qu'on obtient une grande différence en temps total pour le test n°4. Cependant, si les machines avaient été chargées différemment, celle-ci aurait certainement été moins significative. Par contre, dans le cas général de ressources hétérogènes, intuitivement, la différence entre les deux algorithmes pour ce qui est de la délivrance des résultats ordonnés serait toujours présente.

## 5. Limites et évolutions possibles

On peut imaginer des cas tordus où les fréquences disponibles des esclaves seraient faibles et croissantes (dans l'ordre de parcours de la fonction whichslave() de l'algorithme dynamique), avec des bandes passantes légèrement décroissantes. Imaginons par exemple 8 esclaves de fréquence CPU disponible 1000, 1001, 1002, 1003, 1004, 1005, 1006 et 1007 et de bande passante disponible respectivement 99%, 80%, 65%, 53%, 43%, 35%, 29% et 24%. En parcourant ces esclaves, l'algorithme dynamique voit à chaque itération une fréquence supérieure à la précédente et une bande passante disponible supérieure à 80% de la précédente. Il choisira donc le dernier esclave. Cependant le meilleur choix est sûrement le premier (1000 MHz avec une bande passante de 99% plutôt que 1008 MHz avec une bande passante de 24%).

Afin de choisir un esclave, nous regardons deux critères : la CPU disponible, et la bande passante entre le maître et l'esclave. Suivant, les caractéristiques du réseau, des machines, et du traitement à effectuer, il est intéressant de laisser à l'utilisateur le choix de favoriser la CPU ou la Bande Passante. Nous utilisons pour cela une variable nommée BWIMP, qui note de 1 à 10 l'importance accordée à la bande passante (1: faible, 10: forte). Dans le cadre du projet, nous avons toujours favorisé les performances du CPU de chaque ressource. La valeur de BWIMP était donc de 1. Mais il est possible de la régler facilement dans le fichier de paramètre de AIPE (AIPEparameters).

Le test à vérifier pour déterminer l'esclave à choisir est de la forme:

$$F > a(\text{BWIMP}) * F_{\text{max}} \text{ ET } \text{BP} > b(\text{BWIMP}) * \text{BP}_{\text{max}}$$

$$\text{OU } F > c(\text{BWIMP}) * F_{\text{max}} \text{ ET } \text{BP} > d(\text{BWIMP}) * \text{BP}_{\text{max}}$$

Les quatre fonctions a, b, c et d sont linéaires et de valeurs aux bornes :

$$a(1)=1 \text{ et } a(10)=0.8$$

$$b(1)=0.8 \text{ et } b(10)=1$$

$$c(1)=0.9 \text{ et } c(10)=3$$

$$d(1)=3 \text{ et } d(10)=0.9$$

Tous les tests ayant été effectués avec la valeur BWIMP=1, il serait intéressant de tester d'autres valeurs, notamment dans des cas où la ressource critique est la qualité du réseau.

## IV. Conclusions et perspectives

Une stratégie imaginée était, lorsque le maître a une tâche à envoyer et que tous les esclaves sont occupés, qu'il l'envoie néanmoins à l'un d'eux choisi de la même manière que s'il n'avait pas de tâche en cours. Nous avons commencé à implémenter cette idée pour la tester mais cela nécessitait une trop importante restructuration du logiciel, notamment au niveau de la gestion des mémoires partagées.

Dans un environnement complexe où de nombreux utilisateurs font travailler de nombreuses machines, une application se doit d'essayer de ne pas trop déranger les autres. Or on sait qu'une communication a un coût fixe et un coût quasiment proportionnel à la taille des données. Avec l'hypothèse que la famine en données à l'entrée est rare, on peut alors imaginer que le nombre de plaques envoyées à chaque esclave pourrait dépendre des informations relevées par NWS. Ainsi on enverra d'autant plus de plaques de données à une machine si elle a d'autant plus de fréquence CPU et de bande passante disponibles. De plus, une telle stratégie réduirait le nombre d'appels à la bibliothèque de communication.

Dans ce cas, du côté esclave, il pourrait y avoir une mémoire tampon stockant les tâches reçues en attente de traitement, l'esclave ne traitant qu'une tâche à la fois.

Dans le cas de nombreuses machines, la mesure de la bande passante avec chaque esclave peut gêner le maître, étant donné que nous sommes dans une architecture en étoile. Une solution qui pourrait être envisagée, serait de demander à NWS d'observer la bande passante qu'à des moments précis. Mais cette solution ne règle pas le problème de l'amorçage, où l'on doit tester tous les esclaves avant de démarrer.

# V. Bibliographie

Nimrod/G:

*<http://www.csse.monash.edu.au/~nimrod/nimrodg/>*

AppLeS:

*<http://grail.sdsc.edu/projects/apst/>*

Ganglia:

*<http://ganglia.sourceforge.net/>*

NWS:

*<http://nws.cs.ucsb.edu/>*

Tutoriel sur NWS:

*<http://www.nsf-middleware.org/documentation/NMI-R3/0/NWS/>*