INF7235 : Laboratoire #8
Approche coordonnateur–travailleurs avec sac de tâches en MPI

20 mars 2017

Introduction

L'objectif de ce deuxième laboratoire MPI est de vous familiariser avec d'autres éléments du langage MPI/C — communicateurs et sous-groupes de processus, couleurs de messages, etc.

Plus spécifiquement, l'objectif est d'utiliser ces éléments dans le but de mettre en oeuvre un programme de style «coordonnateur/travailleurs» utilisant un «**sac de tâches distribué**», et ce dans le but de trouver le nombre d'occurrences d'une série de mots-clés dans un groupe de fichiers.

Pour obtenir le code source

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/Labo2-MPI.git
```

Programme `mots-cles-seq.c`

Le fichier `mots-cles-seq.c` contient une version séquentielle du programme. Cette version séquentielle peut être compilée et exécutée avec les commandes suivantes :

```
$ make compile EXEC=mots-cles-seq
$ ./mots-cles-seq  les-mots-cles.txt les-fichiers.txt
```

Programme `mots-cles-statique.c`

Le fichier `mots-cles-statique.c` contient une version parallèle avec *décomposition statique* des tâches entre les processus, c'est-à-dire que les divers fichiers à traiter sont simplement répartis, uniformémement, entre les différents processus. Cette version peut être compilée et exécutée avec la commande suivante :

```
$ make run EXEC=mots-cles-statique NP=5
```

Question : Pourquoi une telle solution risque-t-elle de ne pas être efficace?

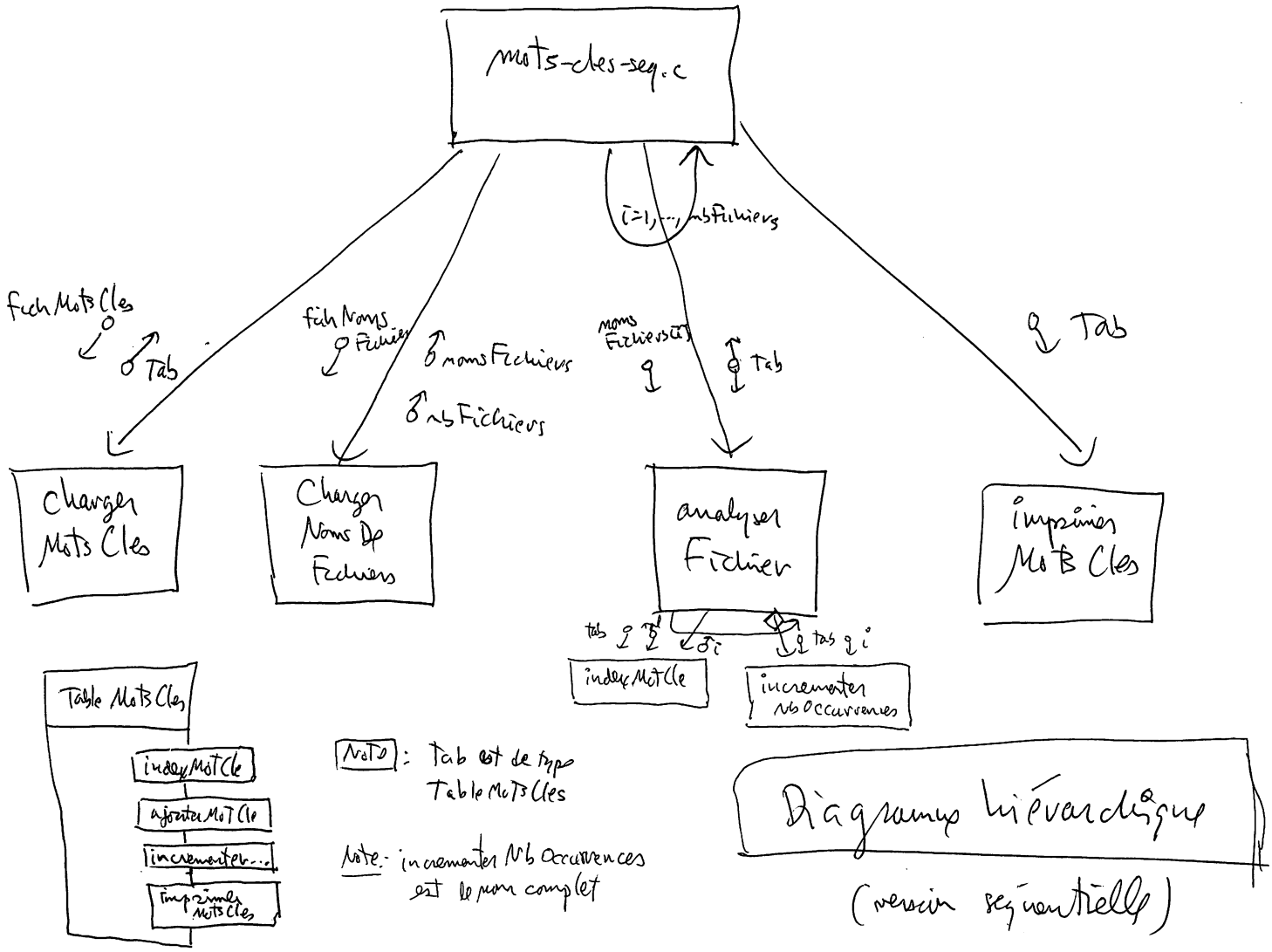
Programme `mots-cles.c`

Le programme `mots-cles.c` contient un squelette de programme permettant de trouver le nombre d'occurrences d'une série de mots clés dans un ensemble de fichiers. La stratégie de programmation utilisée est celle avec coordonnateur/travailleurs et allocation dynamique des tâches (sac de tâches distribué).

Le programme principal vous est fourni, de même qu'un certain nombre de routines auxiliaires, entre autres : mise en oeuvre d'une forme simplifiée de dictionnaires pour conserver le nombre d'occurrences des divers mots clés ; lecture des noms de fichiers à analyser ; analyse (séquentielle) du contenu d'un fichier. Par contre, les procédures qui réalisent le `coordonnateur` et le `travailleur` ne sont définies que de façon partielle — déclaration de l'en-tête + certains éléments du corps.

La Figure 1 (p. 3) présente un *diagramme hiérarchique* (approche procédurale de conception structurée) décrivant la structure générale du programme.

Votre tâche consiste à développer la mise en oeuvre des procédures pour le `coordonnateur` et les `travailleur`s — voir le fichier `mots-cles.c` pour le pseudocode.



Légende :

- Rectangle = procédure
- Lien (flèche) entre rectangles = appel de procédure (ou fonction)
- Flèche en arc de cercle = série d'appels dans une boucle
- Identifiant avec petite flèche = argument transmis ou résultat retourné lors de l'appel de procédure
- Rectangle avec petite boites = Type abstrait (≈ classe d'objects)

Figure 1: Diagramme hiérachique du programme, **version séquentielle**.

Quelques remarques :

- Pour transmettre des chaînes de caractères dans un message, il est préférable d'utiliser le type `MPI_BYTE`.
- Lorsqu'on désire simplement transmettre un «signal» — donc lorsque le nombre d'items à transmettre est nul — alors l'adresse du tampon source peut être `NULL` et on transmet/reçoit ... 0 octets :

```
MPI_Send( NULL, 0, MPI_BYTE, ... );
MPI_Recv( NULL, 0, MPI_BYTE, ... );
```

- Lorqu'on reçoit un message, on peut déterminer l'envoyeur du message, ainsi que la couleur du message reçu, en examinant l'objet `MPI_Status` retourné par l'appel à `MPI_Recv` :

```
MPI_Status statut;
MPI_Recv( ..., &statut );
// Envoyeur = statut.MPI_SOURCE
// Couleur du message reçu = statut.MPI_TAG
// Taille du message reçu: MPI_Get_count( statut, <type>, &nbRecus );
```

- Lorsqu'on ne veut pas qu'un processus fasse partie d'un nouveau communicateur qu'on va créer, il suffit d'indiquer l'étiquette `MPI_UNDEFINED`. Dans cet exemple, le processus 0 ne fera donc pas partie du communicateur `comm` :

```
if ( myID == 0 ) {
  MPI_Comm_split( MPI_COMM_WORLD, MPI_UNDEFINED, myId, &comm );
  ...
```

- Lorsque c'est possible, essayez de faire en sorte que certaines communications se fassent de façon **concurrente** avec les traitements.
- Il faut s'assurer que tous les travailleurs terminent correctement, lorsqu'il n'y a plus aucune tâche, avant que le coordonnnateur termine.

Fichier `makefile`

Le fichier `makefile` qui vous est fourni permet d'automatiser certaines tâches :

- `make compile` : Compilation du programme.
- `make NP=`*`np`* `debug1` : Exécution du programme avec plusieurs processus sur un unique processeur.
- `make NP=`*`np`* `debug2` : Exécution du programme avec plusieurs processus sur deux processeurs physiques.
- `make NP=`*`np`* `run` : Exécution du programme avec plusieurs processus sur plusieurs processeurs physiques.

  Dans tous les cas, si «`NP=`*`np`*» est omis, alors `NP=4`.

Protocoles des processus `coordonnateur` et `travailleur`

Quatre (4) sortes de messages peuvent être transmis/reçus par le coordonnateur et les travailleurs — donc quatre (4) «couleurs» possibles pour les messages :

```
#define MSG_NOM_FICHIER   1
#define MSG_TERMINER      2
#define MSG_REQUETE_TACHE 3
#define MSG_TABLE         4
```

Voir le document suivant qui illustre graphiquement le comportement de trois processus avec cinq noms de fichiers :

`http://www.labunix.uqam.ca/~tremblay/INF7235/Labos/messages.pdf`.

Ce comportement est décrit de façon plus formelle à la page suivante, et ce à l'aide de protocoles exprimés sous forme d'expressions régulières.

Un «protocole» spécifie les séquences possibles de messages pouvant être transmis/reçus par un processus. Un tel protocole peut être décrit à l'aide d'une expression régulière.

- Notons par «`!M`» l'envoi d'un message `M`.
- Notons par «`?M`» la réception d'un message `M`.
- Soit `P` le nombre de processus.
- Soit `N` le nombre de fichiers à traiter.
- Soit `n` un entier arbitraire inférieur ou égale à `N`.

Le travailleur et les coordonnateurs obéiront alors aux protocoles suivants :

```
Coordonnateur:
  # Recevoir exactement N requetes pour un nom de fichier et,
  # a chaque fois, en transmettre un.
  (?MSG_REQUETE_TACHE; !MSG_NOM_FICHIER){N};
  # Recevoir P messages pour un nom de fichier,
  # en repondant a chaque fois avec l'indication de terminer.
  (?MSG_REQUETE_TACHE; !MSG_TERMINER){P};
  # Recevoir la table resultante.
  ?MSG_TABLE

Travailleur 0:
  # Recevoir un certain nombre de noms de fichiers puis l'indication de terminer.
  (!MSG_REQUETE_TACHE; ?MSG_NOM_FICHIER){n};
   !MSG_REQUETE_TACHE; ?MSG_TERMINER;
  # Transmettre la table au coordonnateur.
   !MSG_TABLE

Travailleur non 0:
  # Recevoir un certain nombre de noms de fichiers puis l'indication de terminer.
  (!MSG_REQUETE_TACHE; ?MSG_NOM_FICHIER){n};
   !MSG_REQUETE_TACHE; ?MSG_TERMINER
```