

# Travail pratique #2 — INF5170 (gr. 20)

Automne 2008

**Date de remise:** Au plus tard jeudi, 6 novembre, 13h30. Tout travail remis *après* l'heure indiquée sera considéré en retard (pénalité de 10 % par jour).

**Aucun** travail ne sera accepté après jeudi, 13 novembre, 13h30.

**Note :** Prêfêrablement, ce travail est à faire avec une (1) autre personne.

## Simulation numérique de la diffusion de chaleur dans une plaque métallique rectangulaire

Le domaine du *calcul scientifique* (*computational science*) est assurément un domaine où la programmation parallèle et l'utilisation de machines parallèles puissantes jouent un rôle primordial.<sup>1</sup>

L'un des problèmes clés de la modélisation numérique de phénomènes physiques est la résolution d'équations différentielles. On a vu en cours comment on pouvait modéliser numériquement la diffusion de la chaleur dans un cylindre de métal — une solution numérique à une équation différentielle — et ce à l'aide d'une approche itérative.

Plus précisément, le cylindre est décomposé en un ensemble de points équidistants et l'évolution dans le temps est représenté par une séquence d'itérations simulant le processus de diffusion de chaleur entre les divers points.

Un cylindre représente un espace à une dimension. Si on suppose que le cylindre a été découpé en  $n$  points et que la température aux extrémités reste fixe — on suppose qu'on a une source constante de chaleur qui maintient la température constante sur les bordures tout au de long la simulation —, alors l'état (la température) au temps  $t + 1$  du point  $s[i]$  est obtenu en calculant la moyenne des valeurs des points adjacents au temps  $t$  :

$$s_{t+1}[i] = \begin{cases} s_t[i] & i = 1 \mid i = n \\ \frac{s_t[i-1] + s_t[i+1]}{2} & 1 < i < n \end{cases}$$

Quant aux conditions au temps  $t = 0$ , elles doivent être spécifiées de façon explicite et déterminent, dans le cas des bordures, les valeurs à chacune des étapes subséquentes — problème dit avec «*initial boundary condition*».

Une approche semblable peut aussi être utilisée pour modéliser la diffusion de la chaleur dans une plaque rectangulaire à deux dimensions. Si on suppose que la plaque est décomposée en une grille de  $n \times m$  points, alors l'état (la température) au temps  $t + 1$  d'un point  $s[i, j]$  est obtenu en calculant la moyenne des valeurs des points adjacents au temps  $t$ , plus précisément, les voisins directement en haut, en bas, à gauche et à droite :

$$s_{t+1}[i, j] = \begin{cases} s_t[i, j] & i = 1 \mid j = 1 \mid i = n \mid j = m \\ \frac{s_t[i-1, j] + s_t[i+1, j] + s_t[i, j-1] + s_t[i, j+1]}{4} & 1 < i < n, 1 < j < m \end{cases}$$

Cette approche itérative est appelée *méthode de Jacobi*, méthode qui est expliquée plus en détails à la section 11.1 du manuel d'Andrews (*11.1 Grid computations*, sous-sections 11.1.[1235]).

**Note importante :** Contrairement au programme présenté dans le manuel, *vous ne pourrez pas supposer* que le nombre de points (nombre de lignes ou colonnes de la grille) est directement divisible par le nombre de *threads*. De plus, le problème sera plutôt formulé en termes de longueur et de hauteur de la plaque et la distance (appelée aussi *pas*, en anglais *step*) entre les divers points. Vous devrez donc établir la correspondance appropriée entre les positions de la plaque et les points de la grille — entre autres, voir figure à la page 6.

<sup>1</sup>Voir la page web suivante pour des informations et liens généraux sur ce domaine : [http://en.wikipedia.org/wiki/Scientific\\_computing](http://en.wikipedia.org/wiki/Scientific_computing),

## Ce que vous devez faire

La figure 1 présente les spécifications (en-têtes des procédures et fonctions avec types des arguments et des résultats, accompagnés de pré/post-conditions informelles) pour un module (**resource**) permettant d'effectuer les calculs pour la simulation numérique d'un processus de diffusion dans une plaque à deux dimensions. Divers fichiers<sup>2</sup>, y compris un squelette pour le fichier `Diffusion2D.mpd`, vous sont fournis à l'URL suivant (voir plus bas pour la description de ces divers fichiers) : <http://www.info.uqam.ca/~tremblay/INF5170/Devoirs>

Le module `Diffusion2D` (fichier `Diffusion2D.mpd`) exporte diverses opérations que vous devez mettre en oeuvre. La principale est évidemment l'opération `calculer` qui reçoit un argument indiquant de quelle façon le calcul doit être effectué (le `Mode` de calcul) ainsi qu'un autre argument indiquant pour combien d'itérations (combien «d'unités de temps») le calcul doit être effectué. Les quatre modes d'évaluation à réaliser sont les suivants :

- `JACOBI_SEQ` : méthode de Jacobi séquentielle, donc avec un unique processus (peu importe la valeur de `MPD_PARALLEL`).
- `JACOBI_CIW` : méthode de Jacobi parallèle, où des *threads* sont créés à *chacune des itérations (co-inside-while)*, chaque *thread* traitant une «tranche» de la grille (parallélisme à granularité grossière).
- `JACOBI_WIC` : méthode de Jacobi parallèle, où un certain nombre de processus itératifs sont créés au début du calcul pour traiter les diverses tranches (*while-inside-co*), chaque processus effectuant le nombre d'itérations appropriés sur sa tranche (donc pas de création répétitive de processus à chaque itération, avec granularité grossière).
- `GAUSS_SEIDEL_ROUGE_NOIR` : méthode de Gauss-Seidel parallèle, où on accélère la convergence en utilisant les valeurs déjà calculées à l'étape courante, mais où l'ensemble des cellules de la grille sont traitées en deux fois (deux passes) : les cellules «rouges» puis les cellules «noires».

Cette méthode est expliquée à la section 11.1.5 du manuel d'Andrews. Nous l'examinerons ultérieurement en classe.

**Note** : Pour toutes les méthodes parallèles, si le nombre de *threads* spécifié par l'utilisateur est zéro (0), alors ce sera à votre programme de déterminer le nombre optimal de *threads* à créer pour utiliser efficacement les différents processeurs (virtuels).

## Ce qui vous est fourni

Un certain nombre de fichiers vous sont fournis :

- `Diffusion2D.mpd` : Un squelette de module pour les opérations spécifiées dans la figure 1.
- `Barriere.mpd` : Un module définissant une barrière de synchronisation, que vous pouvez utiliser tel quel.
- `Animer.mpd` : Un programme qui permet d'animer — donc de visualiser — le processus de diffusion, et ce en utilisant les modules suivants, qui vous sont aussi fournis :
  - `Afficheur.mpd` : Un module permettant l'affichage à l'écran.

---

<sup>2</sup>Ce fichier vous est fourni sous forme d'archive comprimée. Pour décompresser ce fichier d'archive et obtenir les fichiers, il suffit d'exécuter la commande suivante (notez le «-» à la fin) :

```
gzcat Fichiers-dev2.tar.gz | tar xvf -
```

```

resource Diffusion2D
#####
# Les types exportes par le module.
#####
# Etat d'une position particuliere de la grille.
type Reel = real;          # On aurait aussi pu utiliser les reels a grande precision ;)

# Grille (matrice) definissant l'etat du calcul dans son ensemble.
type Grille = [*][*]Reel;

# Type pour fonction servant a initialiser la grille,
# donc indiquant la valeur initiale pour chacune des positions.
# Dans le cas des points sur les bordures, la valeur restera
# constante tout au long de la simulation, egale a la valeur specifiee
# par cette fonction.
# Note: On suppose que (0,0) est dans le coin superieur gauche et que
#       (longueur, hauteur) est dans le coin inferieur droit : voir figure page 6.
#       Cette correspondance reflète plus simplement le lien avec les indices
#       de la grille qui represente les differents points de la plaque.

optype FonctionInitialisationType( Reel longueur, Reel hauteur, Reel x, Reel y ) returns Reel;
type FonctionInitialisation = cap FonctionInitialisationType;

# Les differents modes d'evaluations qui doivent etre mis en oeuvre.
type Mode = enum ( JACOBI_SEQ, JACOBI_CIW, JACOBI_WIC, GAUSS_SEIDEL_ROUGE_NOIR );

#####
# Les procedures et fonctions exportees par le module.
#####

op grille() returns ptr Grille gr;
# POSTCONDITION
# gr = pointeur vers la grille (matrice) decrivant l'etat global du calcul.

op calculer( Mode mode, int nbIterations ) returns real maxErreur
# PRECONDITION
# nbIterations >= 1
# POSTCONDITION
# - La fonction de transformation est appliquee pour nbIterations consecutives,
#   (en utilisant le mode de calcul indique), et la grille est modifiee en consequence
# - maxErreur = la plus grande erreur (difference) parmi les differentes cellules
#   de la grille finale et celles la grille qui precede de facon immediate.
# Soit gr = la grille au temps final tf
# gr' = la grille au temps precedent tf-1
# maxErreur = MAXIMUM( 1 <= i <= nbLignes, 1 <= j <= nbColonnes ::
#   abs(gr[i,j] - gr'[i,j]) )

op utiliserNbThreads( int nb );
# POSTCONDITION
# Assure que les appels subsequents a calculer qui seront realises
# dans un mode parallele utiliseront nb threads MPD.
# Note: Si nb == 0, alors le choix du nombre de threads est laisse au module.

op imprimer();
# POSTCONDITION
# La grille a ete imprimee sur stdout.

# CONSTRUCTEUR: Diffusion2D
body Diffusion2D( real longueur, real hauteur, real pas, FonctionInitialisation etatInitial )
# ARGUMENTS
# longueur : longueur de la "plaque"
# hauteur : hauteur de la "plaque"
# pas : distance entre les points de simulation
# etatInitial : fonction d'initialisation
# PRECONDITION
# longueur >= 0.0
# hauteur >= 0.0
# (longueur / pas) est un entier
# (hauteur / pas) est un entier

```

Figure 1: Interface (opérations publiques) du module Diffusion2D pour la simulation numérique d'un processus de diffusion à deux dimensions.

- `InfoMachine.mpd` : Un fichier de configuration pour générer correctement l’affichage à l’écran. Plus précisément, pour que l’affichage se fasse correctement, chaque cellule de la grille doit correspondre à plusieurs pixels de l’écran d’affichage (`FACTEUR_X` et `FACTEUR_Y`). Les valeurs exactes vont dépendre de votre machine, de votre écran, etc. Vous pouvez aussi faire varier la durée d’affichage de chaque grille (`DUREE_AFFICHAGE`). Finalement, il est important d’indiquer la machine sur laquelle vous travaillez, car le fichier de polices de caractères à utiliser (cf. fichier `Afficheur.mpd`) varie selon la machine.
- `Tester.mpd` : Un (1) programme de test qui permet de tester les divers modes de calcul en spécifiant, lors de l’appel, le mode de calcul à tester ainsi que le nombre de *threads* à utiliser. Il s’agit ici de tests publics, que vous pouvez (devriez!) augmenter — pour la correction, j’utiliserai des tests additionnels (privés).
- `MesurerTemps.mpd` : Un programme pour mesurer le temps d’exécution sur une grille simple (valeur initiale de 100.0 sur la bordure, 0.0 ailleurs).

Les différents modes dont on veut mesurer le temps d’exécution peuvent être spécifiés par une chaîne des caractères. Par exemple, pour tester uniquement les modes `JACOBI_SEQ` et `GAUSS_SEIDEL_ROUGE_NOIR` sur une plaque rectangulaire de taille  $100 \times 100$  avec un pas de 1 (donc 101 points) pendant 200 itérations, avec quatre (4) répétitions consécutives de mesure du temps d’exécution (pour calculer une moyenne, le temps d’une exécution à l’autre pouvant varier), on utiliserait l’appel suivant :

```
$ MesurerTemps.out 100 200 "03" 4
```

Deux cibles de mesure sont définies dans le fichier `makefile` : `m100` (plaque  $100 \times 100$  avec un pas de 1 pendant 200 itérations) et `m300` (plaque  $300 \times 300$  avec un pas de 1 pendant 200 itérations). Pour vos expérimentations, vous devriez toutefois ajouter des cibles supplémentaires — i.e., faire des expérimentations avec quelques autres valeurs.

- `OpsAuxiliaires.mpd`, `MPDUnit.mpd` et `MPDUnit-body.mpd` : Les mêmes fichiers que dans le devoir #1 (cadre de tests unitaires pour MPD)
- `makefile` : Un fichier qui permet d’automatiser la compilation et l’exécution des fichiers et programmes. Les principales cibles sont les suivantes :

- `make compilation` : compile l’ensemble des fichiers et programmes.
- `make tests` : exécute *l’ensemble des tests*.

Note : `tests` est la cible par défaut du `makefile`, donc «`make`» sans argument équivaut à «`make tests`», i.e., pour l’ensemble des modes de calcul, tant avec un unique processus qu’avec plusieurs. Au début, je vous suggère de changer cette cible par défaut, par exemple, de n’exécuter que pour le cas séquentiel en modifiant le `makefile` comme suit — l’espace devant «`Tester.out 0 1`» est en fait un caractère de tabulation :

```
default: testsimple0

testsimple0: Tester.out
           Tester.out 0 1
```

Ou encore :

```
default: animation0
```

- `make mesures` : effectue les deux exécutions de mesure du temps indiquées plus haut.

- `make remise` : effectue la remise électronique (avec l'outil `oto`).  
**Note** : *Avant* d'effectuer cette commande, vous devez modifier la valeur de la variable `CODES_PERMANENTS` dans le fichier `makefile`, de façon à indiquer vos codes permanents si vous travaillez en équipe (de deux personnes maximum) ou votre code permanent si vous travaillez seul.
- `make clean` : fait le ménage, c'est-à-dire, supprime les fichiers qui peuvent être générés de façon automatique.

## Ce que vous devez remettre

Vous devez remettre les éléments suivants, pour lesquels la pondération sera celle indiquée :

1. [10 pts] Code source (*listing* papier) pour le fichier `Diffusion2D.mpd` que vous aurez développé, fichier qui devra comprendre toutes les procédures auxiliaires que vous aurez définies et utilisées.

**Note** : comme dans le devoir 1, il est interdit de modifier l'interface du module. En d'autres mots, *vous ne pouvez pas ajouter d'opérations publiques* (op déclarées dans l'en-tête du module). Par contre, vous pouvez (devez) évidemment définir des *procédures* auxiliaires dans le corps du module (après le mot-clé `body`).

La *qualité* (style) de votre code (présentation, clarté et simplicité, structure, choix des identificateurs, etc.) sera évaluée. Par contre, même si la présence de commentaires est suggérée, je n'évaluerai pas la présence, ou l'absence, de tels commentaires.

Vous devrez aussi me transmettre une version sous forme électronique de votre code source. Pour ce faire, vous devrez utiliser la commande «`make remise`», telle qu'indiquée plus haut.

2. [20 pts] Bon fonctionnement de votre module, que vous devez vérifier en exécutant le programme de test qui vous est fourni (`make tests`) — pas nécessaire de me remettre une trace papier de l'exécution — et respect du mode de fonctionnement demandé. Notez que je corrigerai évidemment votre module à l'aide de *tests additionnels* (*privés*).
  - [5 pts] Bon fonctionnement de la procédure `calculer` en mode `JACOBI_SEQ`.
  - [5 pts] Bon fonctionnement de la procédure `calculer` en mode `JACOBI_CIW`.
  - [5 pts] Bon fonctionnement de la procédure `calculer` en mode `JACOBI_WIC`.
  - [5 pts] Bon fonctionnement de la procédure `calculer` en mode `GAUSS_SEIDEL_ROUGE_NOIR`.
3. [10 pts] Bonne parallélisation des diverses mises en oeuvre parallèles : une (ou plusieurs) version(s) parallèle(s), lorsqu'exécutée(s) sur *plusieurs* processeurs, devrai(en)t être plus rapide(s) que la version séquentielle.

De plus, vous devez produire un histogramme illustrant l'*accélération absolue* obtenue pour votre version parallèle la plus rapide, et ce pour divers nombres de «processeurs» (valeur de `MPD_PARALLEL`) et diverses tailles de grille. (Vous pouvez dessiner cet histogramme à la main, en autant qu'il soit clair et qu'il respecte l'échelle appropriée.)

Vous devez aussi répondre brièvement aux questions suivantes : Quelle semble être la meilleure accélération obtenue? Est-ce que l'accélération semble dépendre de la taille de la matrice? Du nombre d'itérations? Si oui, pour quelle taille de matrice et quel nombre d'itérations l'accélération semble-t-elle la plus élevée?

4. (**Bonus**) [10 %] La personne ou l'équipe qui remettra le module dont la procédure `calculer` avec parallélisme fonctionnant sur plusieurs processeurs sera *nettement* la plus rapide recevra un bonus de dix pour cent (10 %).

**Informations additionnelles :**

- Les fonctions prédéfinies `lb` et `ub` peuvent, en plus d'un tableau, prendre un deuxième argument entier qui indique la dimension sur laquelle on désire obtenir la borne, par exemple, soit le tableau `A` déclaré comme suit : `int A[10][2:18]` ;. Alors :

```
lb(A, 1) = 1, ub(A, 1) = 10
lb(A, 2) = 2, ub(A, 2) = 18
```

- Voici la façon d'interpréter les positions (`real x, y`) dans la plaque rectangulaire pour l'initialisation de la grille :

