TP de programmation parallèle

—

ISIA & MR2I IAHP

Serge.Algarotti@cemef.cma.fr Ronan.Keryell@enst-bretagne.fr
Carole.Torrin@cemef.cma.fr

18-19 janvier 2007

1 Déroulement et environnement logiciel du TP

Le but de ce TP est de voir aussi comment vous vous débrouillez de problèmes qu'on peut avoir l'occasion de rencontrer dans la vraie vie de l'ingénieur ou du chercheur : seuls les objectifs sont définis et il faut trouver les moyens de résoudre les problèmes rencontrés. ☺

On utilisera plusieurs classes de machines dans le cadre du TP.

Cet énoncé est trop gros pour être fait dans le temps imparti, donc on propose un parcours possible après avoir parcouru les descriptions introductives : 2.1, 2.2, 2.3, réfléchir au 2.5, 4.1, 4.2, 3.1, 5.1.1.

Cela n'empêche pas de lire le reste... ☺

Ce TP étant joué sur plusieurs sites, chaque site venant avec des logiciels différents ou des versions différentes, cela complique les choses.

1.1 Ordinateurs de l'ENST Bretagne

Les PC sous Linux/RedHat nous serviront de bac à sable pour l'optimisation de programmes séquentiels, à la programmation en OpenMP avec `gcc` et la programmation MPI avec LAM-MPI.

La liste des machines potentiellement utilisables se trouve dans
`https://intranet.enst-bretagne.fr/logistique/logistique_informatique/postes_de_travail_et_logiciels-le`
et on testera avec `ssh` celles qui sont effectivement sous Linux et non sous Windows.

Pour les mesures de performance, il faudra éviter d'utiliser une même machine à plusieurs et s'organiser avec les autres membres du TP. Évidemment, il faut aussi éviter des machines déjà effondrées. ☺

La compilation pour OpenMP sera effectuée avec un `gcc -fopenmp`.

1.2 Ordinateurs de l'ISIA

Les PC sous Linux/Mandrake serviront au TP d'optimisation séquentielle et au TP MPI en faisant une machine parallèle avec passage de messages avec le logiciel LAM-MPI.

1.3 Environnement MPI : le passage de message pour les nuls

Nous utiliserons LAM-MPI, MPICH1 ou MPICH2 avec un accès via `ssh` aux nœuds distants. A priori ce sera LAM-MPI.

Compilation d'un programme sous MPI MPI vient avec des commandes permettant plus facilement de compiler et lier directement les programmes avec les bibiliothèques MPI sans avoir à utiliser explicitement les compilateurs habituels :

- `mpicc`
- `mpif77`
- `mpiCC`
- `mpif90`

Ces commandes acceptent les options permettant de générer des exécutables :

- `-mpilog` créant des fichiers de statistique avec MPE ;
- `-mpitrace` générant des traces ;
- `-mpianim` pouvant utiliser les ordres graphiques de MPE ;
- `-show` ne compile pas mais montre ce que devrait faire la commande.

1.3.1 Configuration de ssh

Il faut aussi vérifier que `ssh` est configuré pour vous permettre de vous connecter sur les machines distantes. Si ce n'est pas le cas voici comment mettre en place un système d'authentification par clé publique avec OpenSSH.

Se créer un joli couple de clé publique-clé secrète avec

```
ssh-keygen -t dsa
```

que l'on chiffrera avec la belle phrase secrète demandée.

Mettre le contenu du `~/.ssh/id_dsa.pub` dans le fichier `~/.ssh/authorized_keys` pour autoriser les connections à partir des machines possédant `~/.ssh/id_dsa`

Pour que `sshd` soit content dans sa grande paranoïa altruiste et vous laisse vous connecter, faire un

```
chmod go-rw ~/.ssh/authorized_keys
```

Enfin, pour éviter d'avoir à taper à chaque fois sa phrase secrète depuis sa fenêtre principale, il faut lancer un agent d'authentification. Si vous lancez `ssh-add` et que vous n'avez pas d'erreur, vous pouvez taper votre phrase secrète car votre environnement a lancé automatiquement un agent d'authentification et sautez le paragraphe suivant.

Lancer l'agent d'authentification avec :

```
eval ‘ssh-agent‘
```

Une fois que vous avez un agent d'authentification qui tourne, rentrez-lui la clé secrète d'authentification avec :

```
ssh-add
```

Vérifier que cela fonctionne avec un

```
ssh machine ls
```

sur chaque machine qui vous intéresse par exemple[1].

Configurez votre `ssh` pour forcer la téléportation de l'affichage graphique X11 qu'on utilisera par la suite. Pour se faire, rajouter dans son `~/.ssh/config` une ligne

```
ForwardX11 yes
```

On peut vérifier si cela marche avec un

`ssh` *`machine`* `xclock`

qui doit afficher une horloge graphique.

1.3.2 LAM-MPI

L'environnement des machines peut être mis à jour par un

```
urpmi lam-devel lam-doc lam-runtime xmpi
```

avec les droits du superutilisateur pour avoir les outils qui nous intéressent sur les machines de l'ISIA sous Mandrake.

Les processus MPI sont lancés par un démon LAM-MPI qu'il faut démarrer au préalable grâce à la commande `lamboot` et on peut préciser la liste des machines participantes dans un fichier passé en paramètre avec 1 nom de machine par ligne. Pour débuguer cette phase de démarrage (connexion `ssh` qui foire, environnement folklorique...), les options `-v`, voire `-d` sont utiles. Attention : il s'agit d'aller vite et donc les démons discutent via *sockets* non sécurisées, donc à réserver à des réseaux sécurisés...

Quelques commandes pratiques :

- `mpirun` *`commande`* permet de lancer un programme MPI en parallèle sur tous les processeurs de la machine. On peut préciser le nombre de processus avec l'option `-np` *`n`* (et des choses plus subtiles en lisant la documentation, comme `-x DISPLAY` pour passer la variable `DISPLAY`...) ;
- `lamnodes` liste les nœuds de la machine virtuelle ;
- `tping` permet de faire des `ping` sur des noeuds de la machine virtuelle, par exemple `tping C` pour tous les processeurs ;
- `lamexec` est similaire au `mpirun` mais permet de lancer des commandes autres que MPI, style `lamexec hostname` ;
- `lamclean` vire tous les processus tournant dans la machine virtuelle ;
- `lamhalt` arrête les démons qui gèrent la machine virtuelle ;
- `xmpi` permet d'afficher de manière graphique plein d'informations sur les messages échangés entre processeurs. `xmpi` peut afficher des informations en temps réel sur un programme lancé depuis `xmpi` ou en analysant des fichiers de trace *post-mortem*.

Pour information, on utilisera la bibliothèque graphique MPE qui vient avec MPICH2 qui a été compilée séparément avec :

```
cd mpich2-1.0.5/src/mpe2
./configure --with-mpiinc=-I/usr/include/lam \
  --with-mpilibs=-L/usr/lib/lam -lmpi -llam \  -lutil -ldl -lpthread --disable-f77 --disable-v
cd src/graphics/contrib/mandel/
make MPI_CC=mpicc CFLAGS=-I../../../../include/ \
  MPE_LIBDIR=../../../../lib MPE_COPTS=-L../../../../lib
```

[1] L'effet de bord positif sera qu'en plus cela mettra le fichier `.ssh/known_hosts` à jour.

1.3.3 MPICH2

Documentation et sources se trouvent sur `http://www-unix.mcs.anl.gov/mpi/mpich2`

Compilation ou installation Au cas où ce ne serait pas déjà installé, faire quelque chose du style :

```
./configure --prefix=/homes/keryell/mpich2 --enable-mpe
make
make install
```

Compilation d'un programme Vérifiez que vous avez configuré votre environnement avec les définitions suivantes

```
PATH=/homes/keryell/mpich2/bin:$PATH
MANPATH=/homes/keryell/mpich2/man:$MANPATH
```

La compilation en soi est similaire à la section 1.3.3.

Exécution d'un programme MPICH2 utilise un démon à lancer au préalable sur les machines pour exécuter ses processus. L'ensemble de ces démons représente la machine virtuelle parallèle. Pour avoir de l'aide, lancer `mpdhelp`.

Le démon se lance avec la commande `mpd` mais il y a une commande plus simple pour lancer ce démon sur tous les nœuds qui nous intéressent d'un coup : `mpdboot -n` *`nombre`* qui lance *`nombre`* démons choisis, en plus de la machine locale, parmi les machines précisées dans le fichier local `mpd.hosts` qui contient un nom de machine par ligne.

`mpdtrace` affiche le nom des nœuds de votre machine parallèle.

`mpdallexit` arrête tous les démons de votre machine parallèle.

Les démons sont organisés en anneau dont on peut estimer les performances de communication par

```
mpdringtest 10
```

par exemple pour mesurer 10 tours d'anneau.

Une fois ces petits tests de fonctionnement effectués on peut s'essayer à lancer des commandes simples en parallèles avec `mpiexec` tel que

```
mpiexec -n 4 hostname
```

qui lance 4 instances de la commande `hostname` qui affiche le nom de la machine où elle tourne.

Mise au point `mpiexec -gdb` lance un programme avec autant de `gdb` que de processus.

1.3.4 MPICH1

La documentation est accessible en `http://www-unix.mcs.anl.gov/mpi/mpich`.

Compilation ou installation La compilation a été effectuée à partir de la configuration téléchargée de `ftp://ftp.mcs.anl.gov/pub/mpi/mpich.tar.gz` :

```
./configure --prefix=/usr/local/mpich-1.2.4 -rsh=ssh --enable-mpedbg
```

Il faut donc rajouter dans son `PATH` :
`/usr/local/mpich-1.2.4/bin`

Sous Linux/Debian on pourrait par exemple installer les logiciels nécessaires simplement avec :

```
apt-get install mpich mpi-doc
```

Au cas où MPICH n'avait pas été configuré pour utiliser `ssh` on peut toujours forcer son utilisation plus tard par exemple en mettant la valeur de la variable d'environnement `RSHCOMMAND` à `ssh`.

Exécution d'un programme Le lancement d'un *programme* sur n processeurs se fait avec :

`mpirun -np` *n programme*

qui va lancer via `ssh` autant de processus que nécessaires sur les machines décrites dans le fichier `/usr/local/mpich-1.2.4/share/machines.LINUX` défini lors de l'installation de MPICH.

Pour mieux préciser l'exécution d'un programme on peut appeler :
`mpirun -p4pg` *pgfile programme*

où *pgfile* décrira ligne par ligne sur quelle machine on veut lancer les processeurs de manière cyclique :

nom-machine p chemin-programme nom-de-login

avec p qui est le nombre de processeurs contenus dans chaque nœud (1 pour un PC simple...). Ce fichier aura pu être créé par un `mpirun -leave_pg` préalable.

Mise au point d'un programme Bien qu'un élève de l'ISIA ou de MR2I programme sans erreur, au cas où ce sujet se perdait ailleurs ☺, voici quelques méthodes :

- rajouter l'option `-dbg=`*debugger* à `mpirun` pour préciser qu'on lance le programme principal dans un debugger ;
- rajouter l'option `-mpedbg` à `mpirun` pour qu'un `xterm` avec un debugger soit lancé autour d'un programme parti en erreur ;
- attacher de manière classique un debugger en cours de route.

1.4 Ordinateurs du CEMEF

Le CEMEF prête bien gentiment des comptes sur leur machines parallèles à mémoires partagée pendant la durée des TPs afin de se faire les dents sur de vrais ordinateurs parallèles. Un résumé du contexte se trouve dans `http://www-cemef.cma.fr/fr/presentation/materiel.html`.

Les comptes sont nommés `tp`*n* avec n variant de 1 à 13 (prendre par exemple son classement dans l'ordre alphabétique pour éviter les conflits).

Les machines ne sont accessibles de l'extérieur que via la machine intermédiaire `ssh.cemef.ensmp.fr` avec par exemple un `ssh -X` (OpenSSH).

Il y a un environnement un peu spécial qui nécessite d'avoir un fichier `.gnurlogin` pour ne pas écraser la variable `DISPLAY` initialisée par `ssh`...

1.4.1 Machines plutôt OpenMP

La variable d'environnement `OMP_NUM_THREADS` permet de préciser le nombre de processus légers (*threads*) utilisés pour exécuter les portions parallèles d'un programme OpenMP.

Multiprocesseurs à mémoire partagée Sun Des ordinateurs Sun avec Solaris 8, les compilateurs WorkShop 6 Forte (donc OpenMP, le multi-*thread* pour les nuls !), MPI de Sun et MPICH :

- 2 Sun E450 quadri-processeurs avec 2 Go de mémoire, nommés `capelet.cma.fr` et `diable.cma.fr` ;
- 1 Sun E4000 à 14 processeurs nommé `gelas.cma.fr`.

Elles ne sont pas équipées de `ssh`.

La documentation peut être trouvée à
`http://www.sun.com/forte/fortran/documentation/index.html`

Il semble que sur Sun on ne puisse faire de l'OpenMP qu'en Fortran 95 et pas en C ou C++... N'est-ce pas là l'occasion rêvée d'acquérir *enfin* un verni sur Fortran ? ☺

La compilation s'effectue typiquement avec l'incantation

```
f90 -openmp
```

Multiprocesseurs à mémoire partagée IBM Des ordinateurs IBM quadri-processeurs avec 4 Go de mémoire, nommés `aiglun.cma.fr` et `maglia.cma.fr` et équipés de compilateurs C & C++ (version 5) et Fortran (version 7.1).

La documentation sur les compilateurs peut être trouvée à
`http://www-4.ibm.com/software/ad/vacpp/`
`http://www-4.ibm.com/software/ad/fortran/`

Pour savoir comment utiliser le mode OpenMP, regarder les documentations (`man`,...). On peut mentionner par exemple :

- `f90 -qfixed` ;
- `xlf90_r -qsmp=omp -qfixed` ;
- `xlc_r -qsmp=omp` ;

On peut trouver un résumé des commandes dans `http://www.navo.hpc.mil/usersupport/MARC/ProgEnv.html`

1.4.2 Multiprocesseurs à mémoire distribué PC sur Myrinet

Il s'agit de machines à mémoire distribuée sous GNU/Linux.

Les commandes MPI classiques `mpicc` ou `mpif77` ou `mpi90` permette de compiler un programme.

Un certain nombre d'outils sont disponibles pour connaître l'état de chaque cluster :

- `/usr/local/tools/online` donne la liste des nœuds disponibles ;
- `/usr/local/tools/topnode -t 31` affiche l'utilisation CPU des 32 nœuds ;
- `/usr/local/tools/topnode -s 4-8` affiche l'utilisation CPU des nœuds 4 à 8 ;
- `/usr/local/tools/topnode 4 6 10` affiche l'utilisation CPU des nœuds 4 6 10.

Cluster p3 : 32 nœuds bi-processeurs Pentium III Il s'agit d'un réseau de 32 nœuds bi-processeurs Pentium III à 1 GHz avec 512 Mo de RAM, interconnectés par un réseau Myrinet à 2,5 Gb/s, le tout tournant sous Red Hat Entreprise WS 2.1 32 bits.

On se connecte par `ssh` sur le point d'accès `mast-p3-cluster`.

Cluster p4 : 32 nœuds bi-processeurs Pentium 4 Xeon Il s'agit d'un réseau de 32 nœuds bi-processeurs Pentium 4 Xeon à 2,8 GHz nommés p4-1 à p4-32 avec 2Go de RAM sauf p4-1 à p4-4 qui en ont 4 Go, interconnectés par un réseau Myrinet à 2,5 Gb/s, le tout tournant sous Red Hat Entreprise WS 2.1 32 bits.

On se connecte par `ssh` sur le point d'accès `dev-p4`.

Cluster op : 8 nœuds bi-processeurs Opteron Il s'agit d'un réseau de 8 nœuds bi-processeurs Opteron 244 à 1,8 GHz nommés op-1 à op-32 avec 2Go de RAM, interconnectés par un réseau Ethernet, le tout tournant sous Red Hat Entreprise WS 3.2 64 bits.

On se connecte par `ssh` sur le point d'accès `mast-op`.

2 Optimisation de programme séquentiel

Le code du produit scalaire montre un exemple de procédure de mesure de temps qu'on peut utiliser.

2.1 Addition de matrice

On essayera de programmer une addition de matrice qu'on s'acharnera à optimiser. On partira du programme donné en exemple dans la section 2.2 pour aller plus vite.

On instrumentera le code pour qu'il affiche le nombre de MFLOPS et le débit mémoire en Mo/s.

La marche à suivre :

- écrire un beau programme bien configurable[2] ;
- initialiser avec des valeurs quelconques 2 matrices en double précision de taille 1000×1000 ;
- faire un certain nombre de fois des additions de matrices afin d'amortir le temps de démarrage du programme ;
- calculer la complexité de l'algorithme et comparer à la vitesse théorique du processeur[3] trouvée sur Internet ;
- essayer d'optimiser le programme :
  - utiliser plein d'options du compilateur (`-On`, SSE, prefetch...
  - regarder l'assembleur généré par le compilateur ;
  - inverser les boucles. Conclusions ?
  - tester avec différents types de données (`float`, `double`, `int`, `short int`,...). Conclusions ?
  - estimer l'importance du cache et de la pagination ;
- faire varier la taille des matrices (puissance de 2 ou pas ?)et regarder comment évolue le temps d'exécution.

Quel est le facteur limitant dans cette histoire ?

[2] Préférer les `enum` nommés aux `#define`. Pourquoi, au fait ?

[3] Pour avoir le modèle de processeur on essayera sous Linux de regarder /proc/cpuinfo, sous Solaris on exécutera `prtconf`, ou de manière générale on farfouillera dans les messages de fonctionnement dans `/var/log/syslog`,... ou renvoyés par l'exécution de `dmesg`.

2.2 Produit scalaire

On pourra s'inspirer de `~keryell/sources/produit_scalaire/produit_scalaire.c` qui contient aussi des procédures de mesure du temps.

2.3 Multiplication de matrices

Les plus fous pourront essayer de s'attaquer à la multiplication de matrice. Long...

```
for(i = 0; i < N; i++)
   for(j = 0; j < N; j++) {
      c[i,j] = 0;
      for(k = 0; k < N; k++)
         c[i,j] += a[i,k]*b[k,j]
   }
```

On cherchera à exploiter le maximum d'opérations en parallèle d'une part et à augmenter la localité dans les caches.

2.4 Calcul de fractales

2.5 Vers une analyse automatique de la machine ?

Concevoir un petit programme de test qui serait capable de déterminer (plus ou moins) automatiquement les caractéristiques de la hiérarchie mémoire simplement en utilisant une routine de mesure de temps :

- taille de la mémoire vive ;
- taille du cache de second niveau et son organisation (taille des lignes du cache, associativité) ;
- idem pour le premier niveau (et éventuellement le troisième niveau le cas échéant) ;
- nombre de registres entiers du jeu d'instructions ;
- nombre de registres entiers physiques totaux utilisés pour le renommage ;
- la même chose pour les registres flottants.

C'est un problème difficile mais intellectuellement valorisant.

2.6 Dénormalisation

Comparer sur une boucle optimisée la différence de temps d'exécution entre des opérations flottantes normales et dénormalisées.

3 Quelques exemples

On décrit ici 2 exemples qui serviront de test par la suite.

3.1 Propagation d'onde

On se propose de simuler la propagation d'onde sur une surface liquide de profondeur variable pouvant contenir des obstacles. Utiliser la souris pour pouvoir mettre des gouttes d'eau où l'on veut.

On discrétisera les équations de base suivantes avec un schéma numérique d'ordre 1 :

$$\frac{\partial \vec{v}}{\partial t} = g(p+w)\vec{\nabla} w \quad (1)$$

$$\frac{\partial w}{\partial t} = \mathrm{div}\vec{v} \quad (2)$$

avec w la différence de profondeur par rapport à la profondeur moyenne p et $\vec{v}$ le vecteur vitesse.

Il s'agit bien d'une propagation d'onde puisque le problème peut se mettre sous la forme :

$$\frac{\partial^2 w}{\partial t^2} = g(p+w)\Delta w$$

Cet exercice gagnera beaucoup à avoir une interface graphique. Pour ce faire on pourra utiliser la bibliothèque MPE qui vient avec MPI. Il semble qu'en utilisant une version spéciale de la bibliothèque MPE n'utilisant pas MPI (`libmpe_nonmpi.a`) on puisse l'utiliser par exemple aussi avec OpenMP.

On peut partir du code de l'ancien TP qui avait lieu en HyperC et qu'on trouve dans `~keryell/examples/drop.hc` ou encore plus simplement d'une version déjà équipée en MPI et MPE mais séquentielle dans `~keryell/MPE/onde_MPE.c`.

L'idée va être de découper l'espace virtuel physique en morceaux distribués sur les processeurs et chaque processeur travaillera sur ces données. On réfléchira au découpage pour diminuer les communications et simplifier la programmation. On cherchera à économiser la mémoire (ne pas allouer en particulier *tout* l'espace virtuel sur chaque processus...). On peut imaginer qu'un seul processeur (le maître) récupère les données des autres processus pour les afficher et gérer les événements souris.

On peut généraliser cette propagation d'onde à d'autres phénomènes physiques. Essayez par exemple d'envoyer l'onde calculée sur le haut-parleur d'une station de travail. Écouter la résonnance d'un tore de dimension 4 ou autre son venus de mondes de dimensions supérieures...

3.2 Jeu de la vie

Le milieu de vie consiste en un espace cartésien de sites (une grille...) qui sont éventuellement remplis d'une cellule.

À chaque itération d'un cycle de vie :
- un site vide entouré de 3 cellules se remplit d'une cellule ;
- une cellule entourée de plus de 3 cellules meurt d'étouffement ;
- une cellule entourée de moins de 2 cellules meurt d'isolement ;
- dans tous les autres cas l'état d'une cellule reste inchangée.

La mise à jour doit être synchrone, c'est à dire que toutes les cellules évoluent ensembles.

3.3 Autres exercices possibles

- transposer une matrice lue dans un fichier ;
- multiplier des matrices lues dans un fichier ;
- faire un programme de calcul de fractales avec interface souris ;
- programmer un jeu de la vie (graphique) ;

– trouver et représenter graphiquement le potentiel d'une surface carrée contenant des électrodes de potentiel et formes diverses. Pour simplifier, rajouter une électrode tout autour du domaine à un potentiel fini. Équation de Poisson :

$$\Delta V = 0$$

en supposant qu'il n'y a pas de charges (que des potentiels) ;
– programmer une méthode de gaz sur réseau (très long...).

4 OpenMP sur machine à mémoire partagée

4.1 Bac à sable

On se propose de s'initier au comportement d'OpenMP en partant du programme `essai.f90` qui se trouve peut-être dans `~keryell/OpenMP`.

On mesurera le temps d'exécution avec `time` *`programme`* (dont on interprétera le résultat dans le cas de plusieurs processeurs) et on surveillera l'exécution avec `top` (mais que représente 100 % ?). On peut tracer les appels systèmes effectués par un processus grâce à la commande `truss -f` (Solaris) ou `strace -f`. En particulier regarder la différence d'exécution si on compile avec ou sans l'option pour OpenMP, et ensuite avec différents nombres de processus légers).

Rajouter un message d'impression dans la boucle. Essayer plusieurs nombres de *threads*. Que constatez-vous ?

4.2 Multiplication de matrice

Programmer un programme de multiplication de matrice qu'on initialisera statiquement à 0. Paralléliser en OpenMP l'initialisation à 0.

Comment évolue le temps d'exécution en fonction du nombre de thread ?

Juste pour voir, comparer les vitesses d'exécution en faisant le calcul sur la transposée de la dernière matrice.

Essayer d'optimiser la multiplication pour faire du blocage afin de rester dans les caches des processeurs.

Inverser les boucles (i, j, k) jusqu'à trouver la combinaison gagnante.

Essayer de mieux exploiter les données qui sont chargées de toute manière dans le cache.

Tester le mode parallélisation automatique du compilateur sur les exemples précédents.

4.3 Propagation d'onde

Paralléliser avec des directives OpenMP et essayer d'utiliser la bibliothèque MPE sans MPI pour l'affichage.

4.4 Jeu de la vie

Faire la même chose que pour la propagation d'onde.

5 MPI sur machine à mémoire distribuée

Bien que MPI soit tout à fait utilisable sur ordinateur à mémoire partagée, on s'intéresse plutôt ici au cas des ordinateurs à mémoire distribuée.

On pourra commencer par s'amuser en LAM-MPI avec :

```
cd ~keryell/mpich2-1.0.5/src/mpe2/src/graphics/contrib/mandel
mpirun -np 4 pmandel
```

ou si MPICH2

```
cd ~keryell/mpich2-1.0.3/src/mpe2/src/graphics/contrib/mandel
mpiexec -n 4 pmandel -i cool.points -loop
mpiexec -n 4 pmandel
```

selon l'existence.

5.1 Propagation d'onde

5.1.1 Méthode SPMD

On essayera de programmer la propagation d'onde en utilisant le modèle de programmation parallèle par passage de messages (programmation en C avec appels à la bibliothèque MPI) :

- initialiser MPI ;
- créer un intra-communicateur contenant tous les processeurs ;
- initialiser l'espace ;
- répartir les données sur les processeurs : découper l'espace en autant de tranches que l'on a de processeurs (toutes les tranches sont de taille identique sauf peut-être la dernière) ;
- envoyer l'état de chaque bord de bloc aux processus voisins ;
- calculer le nouvel état ;
- effectuer un affichage à partir du processeur maitre en utilisant MPE ou à partir de chaque processeur.

La modification à faire au programme doit être minimale. Chaque processus travaillera dans ses tableaux en mémoire de manière inchangée et c'est seulement pour l'affichage qu'on fera le changement de repère.

On aura intérêt à utiliser la technique d'*overlap* pour gérer les bords : chaque bloc est agrandi pour recevoir une copie des bords voisins afin de simplifier l'écriture du programme (mis à part la phase de communication, les bords ne sont pas différenciés dans la phase de calcul).

De même, on tirera profit des caractéristiques du rangement des éléments de tableaux en C pour choisir la distribution la plus judicieuse.

5.1.2 Version MPMD

On pourrait aussi considérer que les entrées-sorties sont prises en charge par une autre application qui communique avec l'application chargée du calcul.

Cette application est aussi executée sur plusieurs processeurs (affichage à l'écran, sauvegarde sur disque,...)

Elle est chargée de l'interface utilisateur et demande à celui-ci la taille du problème, ainsi que l'origine de la goutte (obtenu par un clic de la souris dans la zone désirée).

L'intérêt est ici de faire communiquer entre elles les deux applications à travers différents communicateurs.

5.2 Jeu de la vie

5.2.1 Méthode

Paralléliser d'une manière similaire à celle utilisée pour la propagation d'onde.
Pour écrire les résultats :

- effectuer un affichage à partir du processeur maitre en utilisant MPE ou à partir de chaque processeur ;
- écrire les données dans un fichier à l'aide de la procédure fournie (cf. ci-après).

5.2.2 Vérifions que la vie se déroule bien comme prévu

Initialiser le jeu de la vie avec les paramètres suivants :

- Dimensions de la grille : 100×100
- Durée de vie : 100
- Initialiser le jeux avec

0	1	1
1	1	0
0	1	0

au centre de l'espace vital (50,50) — sachant que l'origine est notée (0,0).

En utilisant la procedure ci-dessous, créer un fichier contenant l'état de la grille au début de la vie et un autre contenant son état à la fin de la vie. Comparer les résultats obtenus en exécutant le programme sur 1,2 puis 8 processeurs.

```
/* Affichage de toutes les valeurs de la grille en mode texte */
void Resultat(char * grille, int Largeur, int Hauteur, char * nom) {
    FILE * f;
    int i, j;
    f = fopen(nom, w);
    for(j = 0; j < Hauteur; j++) {
        for(i = 0; i < Largeur; i++)
            fprintf(f, %d , grille[j*Largeur+i]);
        fprintf(f, \n);
    }
    fclose(f);
}
```