Solveurs parallèles non linéaires creux des problèmes de l'obstacle sur des grappes GPU

Lilia Ziane Khodja

FEMTO-ST, IUT de Belfort-Monbéliard

Réseaux Grand Est (RGE)

14 février 2013

Problème de l'obstacle

Problème de l'obstacle

Modèle mathématique dans un domaine tridimensionnel Ω:

$$\begin{cases} \frac{\partial u}{\partial t} + b^t \nabla u - \eta \Delta u + cu - f \geq 0, \; u \geq \phi, \; sur \; tout \; [0, T] \times \Omega, \; \eta > 0, \\ (\frac{\partial u}{\partial t} + b^t \nabla u - \eta \Delta u + cu - f)(u - \phi) = 0, \; sur \; tout \; [0, T] \times \Omega, \\ u(0, x, y, z) = u_0(x, y, z), \\ C.L. \; sur \; u(t, x, y, z) \; défini \; sur \; \partial\Omega, \end{cases}$$

- **Objectif:** Trouver une position d'équilibre d'une membrane élastique contrainte à se situer au-dessus d'un obstacle solide et qui tend à minimiser sa surface et/ou son énergie.

- **Exemples:**
  - la mécanique des fluides dans les milieux poreux,
  - la bio-mathématique (cicatrisation des plaies ou croissance tumorale),
  - les mathématiques des finances (prix des options Américaines).

Problème de l'obstacle

Après discrétisation, résoudre à chaque pas de temps k un système non linéaire:

$$\begin{cases} \text{Trouver } U \in \mathbb{R}^M \text{ tels que} \\ (A+\delta I)U - G \geq 0, U \geq \bar{\Phi}, \\ ((A+\delta I)U - G)^T(U - \bar{\Phi}) = 0. \end{cases}$$

Algorithme:

1: Initialiser les paramètres du problème de l'obstacle
2: Allouer et copier les données dans la mémoire GPU
3: **for** pas=1 **to** NbPasTemps **do**
4: G $= \frac{1}{k}$U + F
5: Resoudre(A, U, G, ε, MaxRelax)
6: **end for**
7: Copier la solution U de la mémoire GPU vers la mémoire CPU

Résolution sur un GPU

- Méthode itérative de résolution: Richardson projetée:

$$U^{p+1} = F_\gamma(U^p) = P_K(U^p - \gamma(\mathcal{A}U^p - G)), \quad \gamma > 0, \quad p \in \mathbb{N}.$$

P_K: fonction de projection sur l'ensemble convexe K

- Les principaux kernels de la méthode itérative:

```
int bx=64, by=4;
int gx=(NX+bx-1)/bx, gy=(NY+by-1)/by;
int n = NX * NY * NZ;//taille du probleme
dim3 Bloc(bx, by);   //dimensions d'un bloc de threads
dim3 Grille(gx, gy); //dimensions de la grille de threads
double *Y;
...
Multiplication_MV<<<Grille,Bloc>>>(..., U, Y);
Mise_A_Jour_Vecteur<<<Grille,Bloc>>>(..., G, Y, U);
```

Résolution du problème de l'obstacle sur un GPU

- Basée sur les itérations de la méthode Jacobi:

$$\begin{aligned} u^{p+1}(x,y,z) \quad = \quad & \frac{1}{centre} \cdot (g(x,y,z) - (centre \cdot u^p(x,y,z) + \\ & ouest \cdot u^p(x-h,y,z) + est \cdot u^p(x+h,y,z) + \\ & sud \cdot u^p(x,y-h,z) + nord \cdot u^p(x,y+h,z) + \\ & arriere \cdot u^p(x,y,z-h) + avant \cdot u^p(x,y,z+h))). \end{aligned}$$

  - Coefficients de la matrice de discrétisation tridimensionnelle: *center, ouest, est, sud, nord, arriere et avant*
  - p rang de l'itération

Résolution du problème de l'obstacle sur un GPU

Un problème de l'obstacle de taille: $NX \times NY \times NZ$

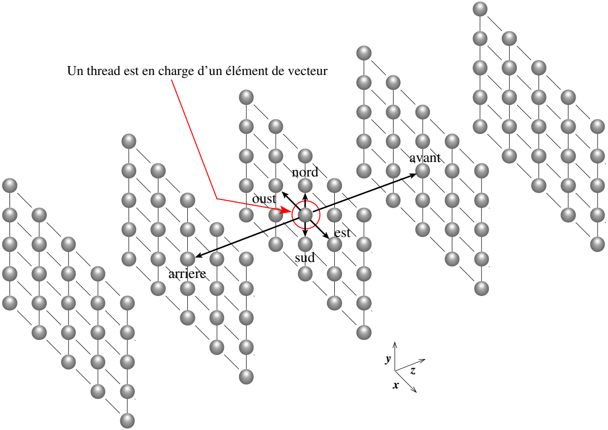

Figure: Calcul d'un élément de vecteur basé sur les valeurs: de deux éléments de chaque dimension et de l'élément à l'intersection

Résolution du problème de l'obstacle sur un GPU

Un problème de l'obstacle de taille: $NX \times NY \times NZ$

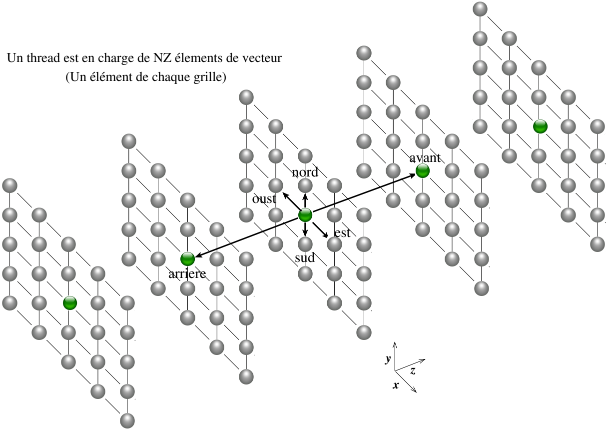

Figure: Chaque thread calcule les NZ éléments de vecteur, le long de l'axe Z, dans une boucle *for*

Kernel de la multiplication matrice-vecteur

```
/* Kernel de la multiplication matrice-vecteur */
__global__ void Multiplication_MV(..., double* U, double* Y)
{
  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;
  int tid = tx + ty * NX;
  //Charger les coefficients de la matrice dans des registres
  ...
  for(int tz=0; tz<NZ; tz++){
    if((tx<NX) && (ty<NY) && (tid<n)){
      double sum = centre * fetch_double(U, tid);
      if(tx != 0)    sum += ouest   * fetch_double(U, tid-1);
      if(tx != NX-1) sum += est     * fetch_double(U, tid+1);
      if(ty != 0)    sum += sud     * fetch_double(U, tid-NX);
      if(ty != NY-1) sum += nord    * fetch_double(U, tid+NX);
      if(tz != 0)    sum += arriere * fetch_double(U, tid-NX*NY);
      if(tz != NZ-1) sum += avant   * fetch_double(U, tid+NX*NY);
      Y[tid] = sum;
    }
    tid += NX * NY;
```

Kernel de la mise à jour du vecteur itéré U

```
/* Kernel de la mise à jour */
__global__ void Mise_A_Jour_Vecteur(..., double* G, double* Y, double* U)
{
  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;
  int tid = tx + ty * NX;
  //Charger le coefficient centre de la matrice dans un registre
  ...
  for(int tz=0; tz<NZ; tz++){
    if((tx<nx) && (ty<ny) && (tid<n)){
      double var = (G[tid] - Y[tid]) / centre + fetch_double(U, tid);
      if(var < 0) var = 0; //projection
      U[tid] = var;
    }
    tid += NX * NY;
   }
}
```

Mise en œuvre sur GPUs

- Programmation CUDA des deux kernels `Multiplication_MV()` et `Mise_A_Jour_Vecteur()`,
- Routines CUBLAS opérant sur des vecteurs double-précision: `cublasDaxpy()`, `cublasDnrm2()`, `cublasDcpy()`,
- Stockage du vecteur itéré U dans la mémoire cache texture $\Rightarrow$ éviter les accès non coalescents à la mémoire globale,
- Stockage des coefficients de la matrices dans des registres de chaque thread.

Résolution parallèle sur une grappe GPU

Partitionnement de données

Partitionnement du problème de l'obstacle tridimensionnel entre les différents nœuds de la grappe GPU:

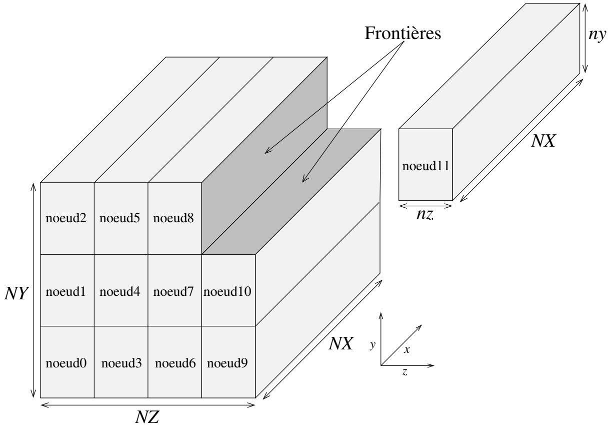

Figure: Partitionnement entre $\alpha = 3 \times 4$ nœuds de calcul

Itérations parallèles asynchrones

- Décomposition du problème en α blocs:

$$U_i^{p+1} = F_{i,\gamma}(U^p) = P_{K_i}(U_i^p - \gamma(\mathcal{A}_i.U^p - G_i)), \quad \gamma > 0, \quad p = 0, 1, ...$$

où: $i \in \{1, 2, ..., \alpha\}$ et $\mathcal{A}_i.U = \sum_{j=1}^{\alpha} \mathcal{A}_{i,j}.U_j$

- Formalisme des itérations asynchrones:

$$U_i^{p+1} = \begin{cases} F_{i,\gamma}(U_1^{\rho_1(p)}, \ldots, U_\alpha^{\rho_\alpha(p)}) & si \quad i \in s(p), \\ U_i^p \quad sinon, & \end{cases}$$

où: $s(p) \subset \{1, ..., \alpha\}$ et $0 \leq \rho_j(p) \leq p$ et $j \in \{1, ..., \alpha\}$

Échanges de données entre nœuds de calcul

A chaque relaxation p:

- Déterminer les éléments de vecteur associés aux frontières,
- Copier ces éléments de la mémoire GPU vers la mémoires CPU,
- Echanger les éléments de vecteur partagés entre les nœuds voisins,
- Copier les éléments reçus des voisins de la mémoire CPU vers la mémoire CPU
- Calculer les nouvelles valeurs du vecteur itéré U.

Routines de communications

- Version synchrone:
  - Entre un CPU et son GPU: `cublasGetVector()` et `cublasSetVector()`
  - Entre CPUs: `MPI_Isend()`, `MPI_Irecv()` et `MPI_Waitall()`
- Version asynchrone:
  - Entre un CPU et son GPU: `cublasGetVectorAsync()` et `cublasSetVectorAsync()`
  - Entre CPUs: `MPI_Isend()`, `MPI_Irecv()` et `MPI_Test()`

Tests expérimentaux

Grappe de GPUs

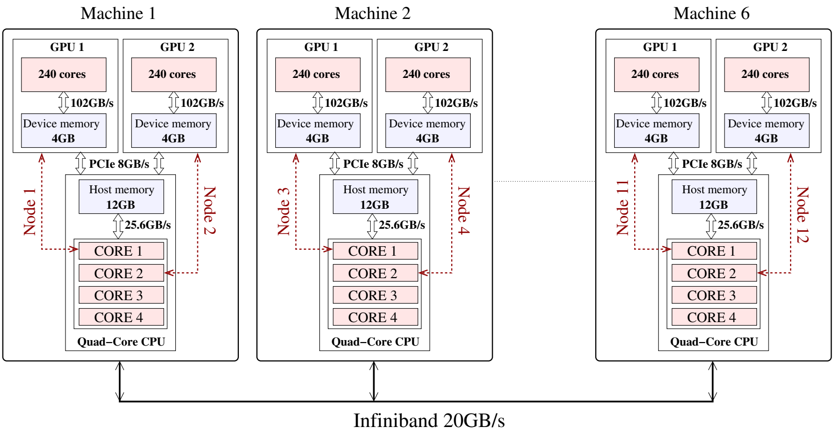

Figure: Grappes GPU composée de 12 nœuds de calcul (six machines, chacune ayant 2 GPU Tesla C1060)

Résultats expérimentaux

Table: Temps d'exécution en secondes sur une grappe de 24 CPUs

Taille du pb.	Synchrone		Asynchrone		gain %
	T_{cpu}	# relax.	T_{cpu}	# relax.	
256^3	575,22	198.288	539,25	198.613	6,25
512^3	19.250,25	750.912	18.237,14	769.611	5,26
768^3	206.159,44	1.577.004	183.582,60	1.635.264	10,95

Table: Temps d'exécution en secondes sur une grappe de 10 GPUs

Taille du pb.	Synchrone		Asynchrone		gain %
	T_{gpu}	# relax.	T_{gpu}	# relax.	
256^3	29,67	100.692	18,00	94.215	39,33
512^3	521,83	381.300	425,15	347.279	18,85
768^3	4.112,68	831.144	3.313,87	750.232	19,42

Amélioration de la convergence

Utilisation de la numérotation rouge-noir:

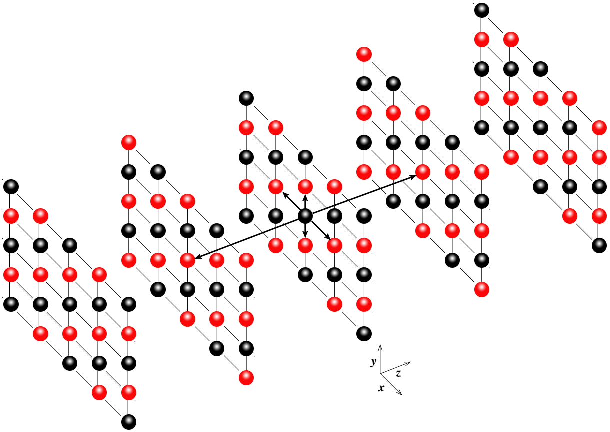

Figure: Numérotation rouge-noir pour le calcul des éléments du vecteur itéré

Amélioration de la convergence

Pour éviter les accès non coalescents à la mémoire globale:

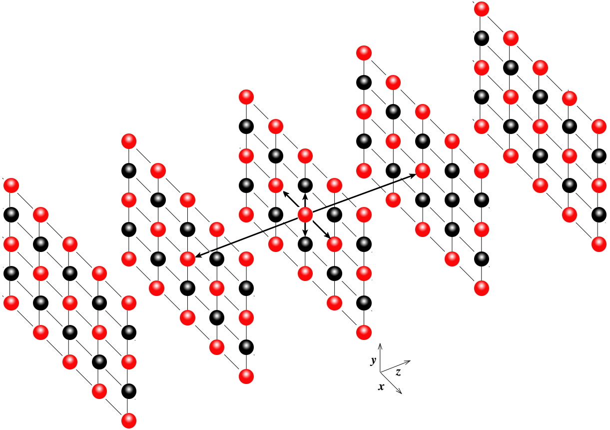

Figure: Numérotation rouge-noir le long de l'axe y

Amélioration de la convergence

- Calculer les éléments de vecteur rouges en fonction des éléments noirs puis,
- calculer les éléments de vecteur noirs en fonction des éléments rouges.
- Le long de l'axe *z*, chaque thread calcule la nouvelle valeur d'un élément sur la grille *i* en fonction de celle de l'élément sur la grille *i-1*.
- Les principaux kernels:

```
int rouge=0, noir=1;
...
Multiplication_MV<<<Grille,Bloc>>>(..., U, Y);
Mise_A_Jour_Vecteur<<<Grille,Bloc>>>(..., rouge, G, Y, U);
Mise_A_Jour_Vecteur<<<Grille,Bloc>>>(..., noir, G, Y, U);
```

Résultats expérimentaux

Table: Temps d'exécution en secondes sur une grappe de 12 GPUs

Taille du pb.	Synchrone		Asynchrone		gain %
	T_{gpu}	# relax.	T_{gpu}	# relax.	
256^3	18,37	71.988	12,58	67.638	31,52
512^3	349,23	271.188	289,41	246.036	17,13
768^3	2.773,65	590.652	2.222,22	532.806	19,88

Table: Gains % obtenus avec l'utilisation de la numérotation rouge-noir

Taille du pb.	Synchrone	Asynchrone
256^3	38,08%	30,11%
512^3	33,07%	31,93%
768^3	32,56%	32,94%

Passage à l'échelle

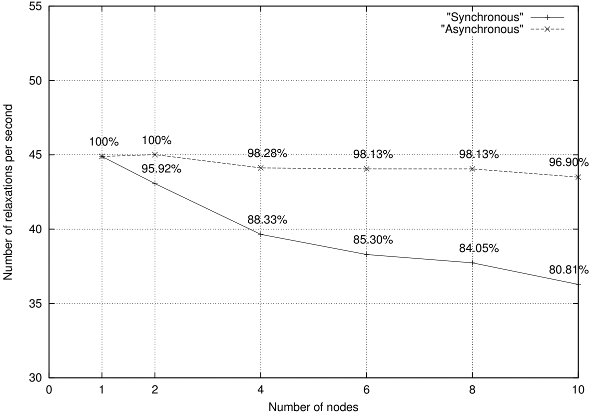

Figure: Passage à l'échelle faible, un sous-problème de 256^3 par nœud

Conclusion

- Maximiser au mieux l'utilisation des cœurs et réduire les accès non coalescents à la mémoire globale,
- Résolution des problèmes de l'obstacle de grandes tailles plus efficace sur une grappe GPU,
- Utilisation des GPUs permet de réduire le rapport entre calcul/communication $\Rightarrow$ pas favorable pour le calcul parallèle,
- Utilisation des algorithmes parallèles à itérations asynchrones,
- Tester les performances des algorithmes asynchrones sur des grappes GPU de grandes tailles ou sur des grappes géographiquement distante.

Merci pour votre attention!