Notes sur la préparation du calcul de DNS 4 milliards de mailles avec A. Toutant et F. Aulery

Benoit Mathieu

17 mai 2013

1 Tests de performance et résultats de référence

Je pars du jeu de données de F.Aulery (jdd_les_897_201_449_04072012.tgz)
Caractéristiques du maillage :

	x	y	z
Dimensions du domaine	0.375056	0.029846	0.187528
Nb nodes	897	201	449
Dimensions de la premiere maille	0.0004185892857142857	0.0000284800355704	0.0004185892857142857

Choix du maillage de test :

	x	y	z
Nb nodes	129	257	129
Dimensions de la premiere maille	0.0004185892857142857	0.0000284800355704	0.0004185892857142857

Performances avec le code actuel (secondes par solveur / secondes par pas de temps) :

Machine	Nproc	s/solveur	s / (3 solveurs)	s / dt
PC Harpertown	8	3.43	10.3	37.5
Curie standard	8	2.1	6.55	13.9
Curie standard	16	1.13	3.4	7.34

Statistiques de temps sur Curie standard, 16 coeurs :

```
Secondes / pas de temps           7.3372
Dont solveurs Ax=B                3.40165 46% (3 appels/pas de temps)
Dont operateurs convection        1.15342 15% (6 appels/pas de temps)
Dont operateurs diffusion         0.718708 9% (15 appels/pas de temps)
Dont operateurs gradient          0.156738 2% (7 appels/pas de temps)
Dont operateurs divergence        0.171927 2% (17 appels/pas de temps)
Dont operateurs source            0.056822 0% (3 appels/pas de temps)
Dont operations postraitement     0.0626825 0% (1 appel/pas de temps)
Dont calcul dt                    0.0398205 0% (4 appels/pas de temps)
Dont modele turbulence            0.209392 2% (1 appel/pas de temps)
Dont calcul divers                1.36604 18%
```

Estimation du gain de performances à atteindre :
- le quota alloué est de 6 millions d'heures CPU,
- on devrait viser $4 \cdot 10^9$ mailles sur environ 16 000 coeurs,
- wall clock time = 375 heures = 1 350 000 secondes,
- on a besoin d'un million de pas de temps, soit 1.35 secondes par pas de temps,
- il faut optimiser le solveur et les opérations hors solveur pour atteindre un gain total d'un facteur 6 à 10.

2 gprof

Calcul 32x128x64 sur 1 coeur sur mon pc :

time	seconds	seconds	calls	s/call	s/call	name
17.43	24.47	24.47	1066120	0.00	0.00	compute_residu_layer_rhovar(...
16.93	48.23	23.76	1081722	0.00	0.00	Multigrille_Adrien::compute_coefficients(TmpRhoLayers&, int, int, int, I
9.85	62.05	13.82	60	0.23	0.23	Iterateur_VDF_FaceEval_centre4_VDF_Face::ajouter_aretes_internes(Double
5.91	70.35	8.30	60	0.14	0.15	Iterateur_VDF_FaceEval_centre4_VDF_Face::ajouter_fa7_elem(DoubleTab cons
5.24	77.71	7.36	240	0.03	0.03	Iterateur_VDF_ElemEval_Dift_VDF_var_Elem::ajouter_interne(DoubleTab cons
4.49	84.01	6.30	2432	0.00	0.02	Multigrille_Adrien::jacobi_residu(IJK_Field&, IJK_Field const*, int, int
3.15	88.43	4.42	122710244	0.00	0.00	Parser::evalOp(PNode*)
3.02	92.67	4.24	60	0.07	0.07	Iterateur_VDF_FaceEval_Dift_VDF_var_Face::ajouter_aretes_internes(Double
2.67	96.42	3.75	61	0.06	0.06	Iterateur_VDF_ElemEval_Quick_VDF_Elem::ajouter_interne(DoubleTab const&,
2.40	99.79	3.37	343	0.01	0.01	Iterateur_VDF_ElemEval_Div_VDF_Elem::ajouter_interne(DoubleTab const&, D
1.85	102.39	2.60	144	0.02	0.02	Op_Grad_VDF_Face::ajouter(DoubleTab const&, DoubleTab&) const
1.75	104.85	2.46	60	0.04	0.05	Iterateur_VDF_FaceEval_Dift_VDF_var_Face::ajouter_fa7_elem(DoubleTab con
1.69	107.22	2.37	7608	0.00	0.00	DoubleVect::operator=(double)
1.62	109.49	2.27	24	0.09	0.10	Champ_Face::calcul_duidxj(DoubleTab const&, DoubleTab&, Zone_Cl_VDF cons
1.24	111.23	1.74	61	0.03	0.03	Assembleur_P_VDF::remplir(Matrice&, DoubleVect const&, Champ_Don_base co
1.19	112.90	1.67	2937600	0.00	0.00	Eval_centre4_VDF_Face::flux_arete_periodicite(DoubleTab const&, int, int
1.17	114.54	1.64	103290	0.00	0.00	IJK_Field::exchange_data(int, int, int, int, int, int, int, int, int, in
0.92	115.83	1.29	11682	0.00	0.00	ArrOfDouble::fill_default_value(Array_base::Resize_Options, int, int)
0.91	117.11	1.28	240	0.01	0.01	Iterateur_VDF_ElemEval_Dift_VDF_var_Elem::ajouter_bords(DoubleTab const&
0.88	118.35	1.24	202	0.01	0.01	Masse_VDF_Face::appliquer_impl(DoubleTab&) const
0.81	119.48	1.13	21	0.05	0.16	Turbulence_hyd_sous_maille_Wale_VDF::calculer_OP1_OP2()
0.78	120.57	1.09	116916224	0.00	0.00	Table::val(double const&) const
0.74	121.61	1.04	63	0.02	0.02	EDO_Pression_th_VDF::calculer_rho_face_np1(DoubleTab const&)
0.69	122.58	0.97	60	0.02	0.02	Iterateur_VDF_FaceEval_Dift_VDF_var_Face::ajouter_aretes_bords(DoubleTab
0.53	123.33	0.75	204	0.00	0.00	operator_divide(DoubleVect&, DoubleVect const&, Mp_vect_options)
0.47	123.99	0.66	60	0.01	0.01	EDO_Pression_th_VDF::divu_discvit(DoubleTab&, DoubleTab&)
0.46	124.63	0.64	100	0.01	0.01	rho_vitesse_impl(DoubleTab const&, DoubleTab const&, DoubleTab&)
0.44	125.25	0.62	112	0.01	0.01	ArrOfDouble::ArrOfDouble(int, double)
0.43	125.86	0.61	241	0.00	0.00	operator_add(DoubleVect&, DoubleVect const&, Mp_vect_options)
0.38	126.40	0.54	421	0.00	0.00	operator_sub(DoubleVect&, DoubleVect const&, Mp_vect_options)
0.37	126.92	0.52	948	0.00	0.00	Coarsen_Operator_K::interpolate_sub(IJK_Field const&, IJK_Field&) const
0.36	127.42	0.50	1192	0.00	0.00	Coarsen_Operator_K::coarsen(IJK_Field&, IJK_Field&, int) const

2.1 Evaluation des performances théoriques

- Bande passante par coeur de calcul : 40GB/s divisé par 8 = 5GB/s = 600M DP floats/s
- Consommation bande passante de l'opérateur Jacobi (in-place) :
  - load + store valeur
  - 4 loads coefficients (si la somme des termes extra-diagonaux est pré-calculée)
- Nombre d'opérations pour Jacobi : 2O
- Perf crête pour une passe si limitation par bande passante : 600M/6 = 100M mailles/s = 2Gflops
- Si regroupement de 4 sweeps : 8 Gflops
- Pour stocker 4 sweeps il faut 8 plans pour l'inconnue plus 16 plans de coefficients.
- Avec 2.5 Mo cache L3 par coeur cela fait des plans de 100kB, soit 110x110 valeurs au maximum.

3 Optimisation du solveur multi-grille

3.1 Stockage des coefficients de la matrice

Le solveur utilise des itérations de Jacobi relaxé et le calcul du résidu.
On utilise la convention suivante pour la numérotation des éléments et des faces :

Un scalaire $c^x_{i,j,k}$ est associé à une face de normale x séparant les éléments $(i-1,j,k)$ et (i,j,k).

Formule pour le résidu (r le résidu, x l'inconnue, b le second membre) :

$$r_{i,j,k} = x_{i,j,k} \cdot c^{xyz}_{i,j,k} - \Sigma_{ijk} - b_{i,j,k} \tag{1}$$

Formule pour une itération de Jacobi relaxé (itération x^{n+1} en fonction de l'itération x^n, σ est le coefficient de relaxation, l'optimum est $\sigma \simeq 0.65$ en 3D) :

$$x^{n+1}_{i,j,k} = x^n_{i,j,k} - \frac{r^n_{i,j,k} * \sigma}{c^{xyz}_{i,j,k}} \tag{2}$$

Avec

$$c^{xyz}_{i,j,k} = \begin{pmatrix} c^x_{i,j,k} + c^x_{i+1,j,k} \\ +c^y_{i,j,k} + c^y_{i,j+1,k} \\ +c^z_{i,j,k} + c^z_{i,j,k+1} \end{pmatrix} \tag{3}$$

$$\Sigma_{ijk} = \begin{pmatrix} x_{i-1,j,k} \cdot c^x_{i,j,k} + x_{i+1,j,k} \cdot c^x_{i+1,j,k} \\ +x_{i,j-1,k} \cdot c^y_{i,j,k} + x_{i1,j+1,k} \cdot c^y_{i,j+1,k} \\ +x_{i,j,k-1} \cdot c^z_{i,j,k} + x_{i1,j,k+1} \cdot c^z_{i,j,k+1} \end{pmatrix} \quad (4)$$

Les coefficients aux faces sont calculés comme suit en fonction des masses volumiques $\rho_{i,j,k}$ aux éléments et des tailles des mailles (exemple pour la direction x) :

$$c^x_{i,j,k} = \frac{\Delta y \Delta z}{\Delta x} \frac{2}{\rho_{i-1,j,k} + \rho_{i,j,k}} \quad (5)$$

$$c^y_{i,j,k} = \frac{\Delta x \Delta z}{\Delta y} \frac{2}{\rho_{i,j-1,k} + \rho_{i,j,k}} \quad (6)$$

$$c^z_{i,j,k} = \frac{\Delta x \Delta y}{\Delta z} \frac{2}{\rho_{i,j,k-1} + \rho_{i,j,k}} \quad (7)$$

Si la face séparant deux éléments est une paroi (cas des plans $k = 0$ et $k = k_{max}$), les coefficients correspondants sont nuls :

$$k = 0 \text{ ou } k = k_{max} \Rightarrow c^z_{i,j,k} = 0 \quad (8)$$

Les coefficients pour la grille grossière sont calculés en utilisant un champ ρ aux éléments du maillage grossier calculé par moyenne arithmétique des masses volumiques dans les mailles fines.

3.2 Optimisation : choix de pré-calculer les coefficients des matrices

La différence entre un stockage des coefficients aux faces et leur calcul à la volée est un facteur 1.75 sur la bande passante mémoire consommée (dans le cas du stockage, lecture/écriture de l'inconnue, lecture des 4 coefficients par élément, lecture du second membre, soit 7 accès mémoire et 4 accès mémoire dans le cas du calcul à la volée). Le calcul à la volée nécessite un espace de stockage complexe à gérer pour réutiliser les coefficients pour les N-passes simultanées et le calcul à la volée des conditions aux limites.

Si la bande passante mémoire est effectivement de l'ordre de 40Go/s par processeur et avec 6 passes simultanées, on arrive à $17 \cdot 6 = 102$ flops pour $7 \cdot 8 = 54$ octets transférés. À saturation de la bande passante mémoire, cela donne 10Gflops par coeur de calcul en double précision. Si on décidait de ne pas précalculer les coefficients, il faudrait le faire à la volée et atteindre 15 à 20 Gflops pour compenser le calcul des coefficients à la volée, ce qui paraît difficile. Une optimisation poussée du code permet de saturer la bande passante mémoire avec moins de coeurs de calcul et de réserver un coeur à des opérations réseau par exemple.

Performances du code après avoir remplacé le calcul des coefficients à la volée par un pré-calcul (version 1, cas test dns_curie_16_cores_ghost1) :

Version	Machine	Nproc	s/solveur	s / (3 solveurs)	s / dt
v0	Curie standard	16	1.13	3.4	7.34
v1	Curie standard	16	0.70	2.11	6.1

Dans la version 1, le code a passé 1.11 secondes au total dans Jacobi sur le maillage le plus fin pour 308 appels et $1.34 \cdot 10^9$ opérations par coeur, soit 1GFlop environ par coeur.

```
   ijk_exchange (01)       0.63    4.6% (0.57,0.64)    2.71e+03           0
jacobi_fine_mesh (01)       1.11    8.2% (1.11,1.12)         308    1.34e+09
     jacobi_all (01)       3.40   25.1% (3.38,3.42)         386    1.08e+10
  coarse_solver (01)       0.34    2.5% (0.34,0.35)          38           0
```

Les déraffinements en z seulement coûtent cher : 4 niveaux, soit le coût du maillage le plus fin multiplié par $1 + 1/2 + 1/4 + 1/8 + 1/16 = 1.94$, ce qui explique le temps passé dans Jacobi au total : 1.94 fois le temps du maillage fin, plus les communications et les coûts de démarrage.

Il faut chercher à réduire le temps des échanges réseau et jacobi_all.

3.3 Tests de bande passante sur Curie

Résultats d'exécution du code test_bw.cpp :

Machine	Ncoeurs	Débit RAM (Go/s)	Débit caches (L3 / L2 / L1)
Curie standard	1	9	23 / 40 / 83
Curie standard	16	4.5	23 / 40 / 83
Curie large	1	5.1	17 / 27 / 54(Add) à 70(Move)
Curie large	32	1.9	17 / 27 / 54(Add) à 70(Move)

3.4 Optimisation du kernel Residu

Première version SSE : 14-15GFlops sur Curie standard (strides en dur 66x64). Version AVX est 20% moins rapide avec GCC (probablement les loads non alignés).

Test du compilateur Intel 12.1.7.256 :

– avec AVX : 11GFlops dans le cache L2, 15GFlops dans le cache L1
– avec SSE (-xAVX) : 11.5 et 13.4GFlops

Algorithme jacobi-residu multipasse (réalise simultanément N passes de Jacobi et éventuellement calcul du résidu) :

– code compilé avec GCC 4.7 -mavx -DWITH_SSE sur Curie et GCC 4.3 sur PC Harpertown (code utilisant une vectorisation SSE), lancé en parallèle sur tous les coeurs d'un noeud, les performances affichées sont par coeur de calcul.
– bloc de données de taille 64x64x128, plus les cellules fantômes nécessaires au nombre de passes effectuées, type "float" (les performances pour le type "double" sont moitié plus faibles),
– on utilise les algorithmes avec tailles de blocs codés en dur (et non l'algorithme générique où la taille des blocs est paramétrable, la différence de performance est de 5 à 10 pourcent sur le Kernel seul mais n'est pas visible sur l'algorithme complet).
– débit RAM estimé si le cache fonctionne de façon optimale (on charge une seule fois les coefficients, le second membre, le residu et l'inconnue, on écrit une fois le résidu et l'inconnue),
– GFlops estimés en comptant 14 opérations par cellule pour le résidu et 17 pour Jacobi (attention, pour les passes simultanées, ce chiffre inclut des opérations supplémentaires pour les cellules fantômes)
– Npasses est le nombre de passes simultanées (plus de passes diminuent la bande passante mémoire moyenne nécessaire mais entraînent des calculs supplémentaires sur les cellules fantôme).
– MCells/s est le nombre de cellules réelles calculées par seconde (sans les cellules fantômes), multiplié par le nombre de passes effectuées pour chaque cellule (ce chiffre donne la performance utile de l'algorithme),
– GFlops Kernel est la performance pour une boucle sur un unique plan ij de données mis en cache (calcul d'une iteration jacobi sur les lignes "sans résidu" et calcul du résidu pour les lignes "avec résidu"), seuls les caches L1 et L2 sont mis en oeuvre.

Machine	avec résidu	Npasses	RAM Go/s	GFlops	MCells/s	GFlops Kernel
Curie std	non	2	6.19	5.46	295	9.48
E5-2680		4	4.74	8.04	392	9.57
2.7GHz		6	3.43	8.4	372	8.98
		8	2.64	8.32	334	5.66
Curie std	oui	2	4.87	3.92	232	11.48
		4	4.48	7.29	371	11.74
		6	3.44	8.2	372	11.35
		8	2.65	8.2	335	10.39
Hpt	oui	2	1.0	0.81	48	6.74
E5440		4	0.96	1.56	79	6.83
2.83GHz		6	0.90	2.13	97	6.90
		8	0.57	1.76	72	6.80

– Les chiffres sur Curie atteignent une fraction significative de la bande passante mémoire crête (6.19Go/s et par coeur représentent 49Go/s par socket pour un maximum théorique de 50).
– Il semblerait que la bande passante des caches L2 et L3 soit le facteur limitant de cet algorithme sur Sandy Bridge pour 6 et 8 passes (pour 8 passes sur un bloc 64x64x128 réelles, soit 80x80x144 au total, 335 millions de cellules par secondes imposent un débit de 22Go/s sur le cache L3).
– Sur architecture Harpertown, quel que soit le nombre de passes, la performance est limitée par la bande passante mémoire (environ 10Go/s pour les 8 coeurs, soit 1.2Go/s par coeur).
– L'architecture Sandy Bridge est 4 fois plus rapide, par coeur, que l'architecture Harpertown.

3.5 Calculs de performances

3.5.1 Débit dans le cache L3

Entre deux passes, seule une couche de maille est réutilisée en cache L2 (le résultat de l'itération Jacobi précédente). Il faut charger en cache L2 2 couches de l'inconnue (une déjà présente), 4 couches de coefficients, 1 couche second membre et une couche résidu (sauf streaming). Il faut écrire deux résultats (avec résidu). Cela fait un trafic entre le cache L2 et L3 de 10 couches de mailles par passe et par couche de maille de l'inconnue.

4 Simplification de l'opérateur de convection "centré 4"

5 Tests dns

5.1 Influence du seuil du solveur en pression

5.2 Premier essai sur calcul grossier

5.3 Tests avec channelflow

Channel flow est un code de dns opensource.

```
http://www.channelflow.org\end{channelflow}

Installation:
\begin{verbatim}
./configure --prefix=/work/mathieu/channelflow/
            --with-fftw3=/work/mathieu/usr/local/
```

Problème : le code n'est pas parallélisé. Ajouter les directives threads pour fftw ne fonctionne pas : code beaucoup plus lent qu'avant (probablement appels à fftw pour des blocs trop petits).

Performances observées : plus de 2 minutes par pas de temps avec un maillage 128x128x96.

6 Documentation of the IJK operator structure

7 Expression de la dérivée de la force

On cherche une expression de l'évolution du terme de forçage sous cette forme :

$$\frac{\partial f}{\partial t} = a \cdot (v_0 - v) - b \cdot f \tag{9}$$

En première approximation, la vitesse fluide est l'intégrale de la force, moins un frottement F :

$$\frac{\partial v}{\partial t} = \frac{f}{\rho} - F \tag{10}$$

Donc

$$\frac{\partial^2 v}{\partial t} = \frac{1}{\rho}\frac{\partial f}{\partial t} = \frac{a}{\rho} \cdot (v_0 - v) - \frac{b}{\rho}\frac{\partial v}{\partial t} \tag{11}$$

ou encore

$$v'' + \frac{b}{\rho}v' + \frac{a}{\rho}v = \frac{a\,v_0}{\rho} \tag{12}$$

On veut un oscillateur avec l'amortissement critique et une constante de temps de δt. Pour cela il faut que le polynôme caractéristique de l'équation différentielle soit de déterminant nul. Le polynôme est :

$$x^2 + \frac{b}{\rho}x + \frac{a}{\rho} = 0 \tag{13}$$

Déterminant nul :

$$\frac{b^2}{\rho^2} - 4\frac{a}{\rho} = 0 \tag{14}$$

La pulsation propre de l'oscillateur est $\omega_0 = \sqrt{a/\rho}$. On va donc prendre :

$$a = \omega_0^2 \cdot \rho, b = 2\omega\rho \tag{15}$$