

# Programmation haute performance pour architectures hybrides

Septièmes rencontres de la communauté française de compilation

Rachid Habel

4 décembre 2013

François Irigoin

Frédérique Silber-Chaussumier



① Programmation pour architectures hybrides

② dSTEP

③ Expérimentations

④ Conclusion

# Outline

- 1 Programmation pour architectures hybrides
- 2 dSTEP
- 3 Expérimentations
- 4 Conclusion

## Contexte

Programmation parallèle d'applications scientifiques

## Objectifs

- Programmabilité
- Efficacité en temps d'exécution et en empreinte mémoire

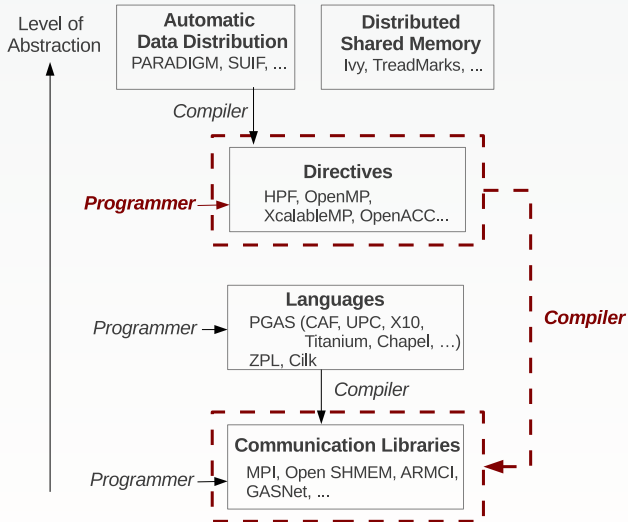
## *STEP* : Système de Transformation pour l'Exécution Parallèle

- Transformation OpenMP vers OpenMP + MPI
- Données répliquées

## *dSTEP* : distributed STEP

- Transformation source à source
- Entrée : programme annoté de directives de distribution
- Sortie : programme parallèle hybride
- Hybride
  - Différents modèles mémoire (distribuée, partagée, accélérateur)
  - Différents modèles de programmation (MPI, OpenMP, CUDA)

# Positionnement

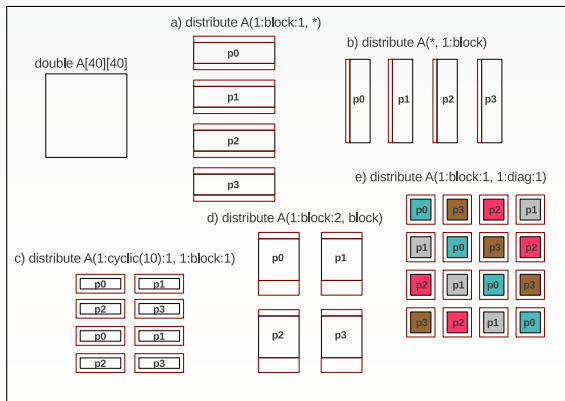


# Outline

- ① Programmation pour architectures hybrides
- ② dSTEP
- ③ Expérimentations
- ④ Conclusion

# Modèle de programmation

- Les données (tableaux)
  - Directive *distribute*
  - Type de distribution : **block**, **cyclic**, **répliquée**, **multi-partitionnée**
  - Halo





- Les calculs (nids de boucles)
  - Directive *gridify*
  - Type de distribution : *block*, *cyclic*, *répliquée*, *multi-partitionnée*
  - Type d'exécution : *parallel*, *ordered*, *owner*

```

1 #pragma step gridify(i(dist=block, sched=ordered), j,
   k)
2 for (i = 1; i < isize; i++) {
3   for (j = 1; j < grid_points[1]-1; j++) {
4     for (k = 1; k < grid_points[2]-1; k++) {
5       matvec_sub(lhs[i][j][k][AA],
6                 rhs[i-1][j][k], rhs[i][j][k]);
7       matmul_sub(lhs[i][j][k][AA],
8                 lhs[i-1][j][k][CC],
9                 lhs[i][j][k][BB]);
10  binvcrhs(lhs[i][j][k][BB], lhs[i][j][k][CC], rhs[i][j]
           ][k]);}}}
```

## Modèle d'exécution

- ① Programmes SPMD
- ② Hors `gridify` : exécution redondante
- ③ `gridify` parallèle : exécution indépendante par tous les processus
- ④ `gridify ordered` : exécution respectant l'ordre initial
- ⑤ Présence de dimension `owner` : seuls les propriétaires des données calculent
- ⑥ Des communications asynchrones sont générées pour la mise à jour des données répliquées
- ⑦ On garantit la complétion d'une communication avant toute future utilisation des éléments impliqués (synchronisation)

## Modèle de distribution des données

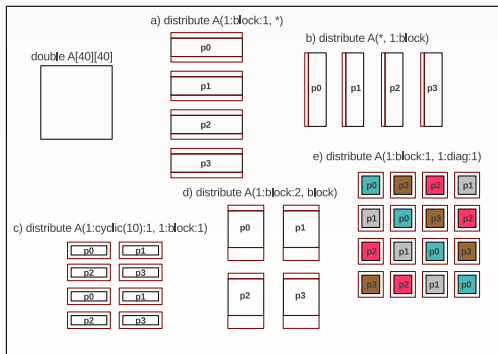
$$a = B_x P_x c_x + B_x \tilde{p}_x - H_{low_x} v_1 + l_x$$

$B_x$  : tailles de blocs,  $P_x$  : grille virtuelle,  $H_{low_x}$  : halo inférieur

$c_x$  : cycle,  $l_x$  : déplacement local,  $\tilde{p}_x$  : identifiant de processus

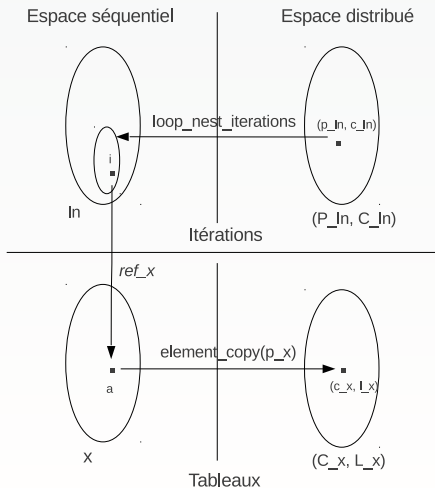
$$\tilde{p}_x = \begin{cases} (p_x + bv_1) \bmod_v P_{V_1} & \text{si une dimension } k \text{ est diagonalisée,} \\ p_x & \text{sinon} \end{cases}$$

$b$  : un entier tel que  $0 \leq b < P_{k,k}$



## Modèle de distribution des calculs

$$i = L_{In}v_1 + B_{In}P_{In}c_{In} + B_{In}\tilde{p}_{In} + I_{In}$$



# Compilation

## Nid de boucle en entrée

```
#pragma step gridify(...)  
for  $i \in \{i \mid L_{ln}v_1 \leq i < U_{ln}v_1\}$  with increment inc do  
    body(..., X[refxm(i)], ...)  
end for
```

Pour chaque nid de boucles

- Calcul des tranches d'itérations
- Pour chaque tranche d'itérations
  - Vérification de la localité des accès
  - Traduction des références dans l'espace distribué
  - Génération de communications asynchrones

Distinction de l'ordonnancement : parallèle vs ordonné

## Compilation d'un nid de boucles parallèle

```
1: compile_parallel(LN, p, b_setln, owner) ≡  
2: pln = id_in_gridln(p)  
3: for bln ∈ b_setln do  
4:    $\tilde{p}_{ln} = \text{extend\_id}_{ln}(p_{ln}, b_{ln})$   
5:   for cln ∈ Cln do  
6:     compile_iterations_slice(body,  $\tilde{p}_{ln}$ , cln, owner)  
7:     generate_sends(LN,  $\tilde{p}_{ln}$ , cln)  
8:   end for  
9: end for  
10: generate_recvs(LN, b_setln, Pln)  
11: generate_completes(LN)
```



# Compilation d'un nid de boucles ordonné

```
1: compile_ordered(LN, p, b_setln, owner)  $\equiv$ 
2:  $p_{ln} = id\_in\_grid_{ln}(p)$ 
3: Predecessors =  $\emptyset$ 
4: for bln scanb b_setln do
5:    $\tilde{p}_{ln} = extend\_id(p_{ln}, b_{ln})$ 
6:   for cln scanc Cln do
7:     for b'ln scanb b_setln do
8:       for  $p' \in \mathcal{P}$  do
9:          $p'_{ln} = id\_in\_grid_{ln}(p')$ 
10:        for c'ln scanc Cln do
11:          if happens_before( $\tilde{p}'_{ln}$ , c'ln,  $\tilde{p}_{ln}$ , cln, orderln)  $\wedge$   $\tilde{p}'_{ln} \notin$  Predecessors then
12:            generate_recvs(LN, p,  $\tilde{p}_{ln}$ , cln, owner, { $\tilde{p}'_{ln}$ })
13:            Predecessors = Predecessors  $\cup$  { $\tilde{p}'_{ln}$ }
14:          end if
15:        end for
16:      end for
17:    end for
18:    generate_completes(LN)
19:    compile_iterations_slice(body, p,  $\tilde{p}_{ln}$ , cln, owner)
20:    generate_sends(LN, p,  $\tilde{p}_{ln}$ , cln)
21:  end for
22: end for
23: generate_recvs(LN, p, owner,  $\mathcal{P}_{ln} -$  Predecessors)
24: generate_completes(LN)
```

## Compilation d'une tranche d'itérations

```
1: compile_references(LN, body, p, iteration_set, owner) ≡
2: computes = true
3: skips = true
4: for each array X referenced in LN do
5:   px = id_in_gridx(p)
6:   accessedx = read_region(X, body, iteration_set) ∪ write_region(X, body, iteration_set)
7:   (cx, bx) = belongs_to(px, min(accessedx), max(accessedx))
8:   if cycle_undefined(cx) then
9:     if !owner then
10:       Abort("Bad distribution")
11:     else
12:       computes = false
13:     end if
14:   else
15:     p̃x = extend_id(px, bx)
16:     shiftx = array_element(p̃x, cx, 0)
17:     skips = false
18:   end if
19: end for
20: if !(computes ⊕ skips) then ▷ Xor
21:   Abort("Bad distribution")
22: end if
23: if computes then
24:   for i ∈ iteration_set with increment inc do
25:     body(..., X[cx][bx][refxm(i) - shiftx], ...)
26:   end for
27: end if
```

## Communications

- Éléments de tableaux écrits par une tranche d'itérations (région *WRITE*) utilisés dans le futur des calculs (régions *OUT*) : région *TO\_SEND*
- Send : pour chaque processus distant, intersecter la région *TO\_SEND* avec l'espace alloué sur ce processus pour le tableau considéré
- Communication si intersection non nulle
- Recv : générés de façon symétrique aux Send (régions *TO\_RECV*)

# Optimisations

- Calcul des voisins : réduction du nombre de processus scannés pour générer les communications
- Complétion des communications asynchrones au plus tard : recouvrement calculs/comms
- Réplication partielle des calculs : initialisation, temporaires, ...

## Complétion des communications asynchrones

```
1: complete_accessed( $X$ ,  $accessed_x$ )  $\equiv$   
2: for ( $request, region$ )  $\in$   $Pending(X)$  do  
3:   if ( $accessed_x \cap region$ )  $\neq \emptyset$  then  
4:     complete_comm( $request$ )  
5:      $Pending(X) = Pending(X) - \{(request, region)\}$   
6:   end if  
7: end for
```

## Génération de code pour GPUs

Le modèle de programmation *data parallel* des GPU impose des contraintes sur un nid de boucles pour qu'il soit éligible à une génération sur GPU :

- ① les  $k$  niveaux les plus internes du nid de boucles doivent être parallèles, avec  $k \geq 1$
- ② le nid de boucles ne doit pas comporter d'appels de fonctions

Les réductions sont implémentées comme fonctions de librairie

## Génération de code pour GPUs, suite

```
1: compile_references(LN, body, p, iteration_set, owner) ≡
2: ...
3: if computes then
4:   gpu_iterations_set = project_first_dimensions(iteration_set, k)
5:   gpu_grid = configure_gpu_grid(gpu_iterations_set, BLOCKS, BLOCK)
6:   cpu_iterations_set = project_last_dimensions(iteration_set, k)
7:   for  $i' \in \text{cpu\_iterations\_set}$  with increment inc' do
8:     kernelln(gpu_grid,  $\tilde{p}_{ln}$ , cln,  $i'$ , inc, shiftx, ...)
9:   end for
10: end if
```

```
1: kernelln(gpu_grid,  $\tilde{p}_{ln}$ , cln,  $i'$ , inc, shiftx, ...) ≡
2:  $i_{kernel} = \text{id\_in\_block\_in\_grid}()$ 
3:  $i'' = \text{loop\_iteration}(\tilde{p}_{ln}, c_{ln}, [v_0, i_{kernel}]^T)$ 
4:  $i = [i', i'']^T$ 
5: if  $L_{ln} \leq i < U_{ln}$  then
6:   body(...,  $X_{gpu}[c_x][b_x][ref_x^m(i) - shift_x]$ , ...)
7: end if
```

## Communications Host/Device

```
1: device_to_host( $X_{gpu}[c_x][b_x]$ ,  $X[c_x][b_x]$ , to_send, shift_x)
2: async_send( $X[c_x][b_x]$ ,  $p'$ , to_send, shift_x)
3: ...
4: for (request, region)  $\in$  Pending( $X$ ) do
5:   if ( $accessed_x \cap region \neq \emptyset$ ) then
6:     complete_comm(request)
7:     Pending( $X$ ) = Pending( $X$ ) - {(request, region)}
8:     if is_rcv(request) then
9:       ( $c_x$ ,  $b_x$ , shift_x) = get_local_information(request)
10:      host_to_device( $X[c_x][b_x]$ ,  $X_{gpu}[c_x][b_x]$ , region, shift_x)
11:    end if
12:  end if
13: end for
```



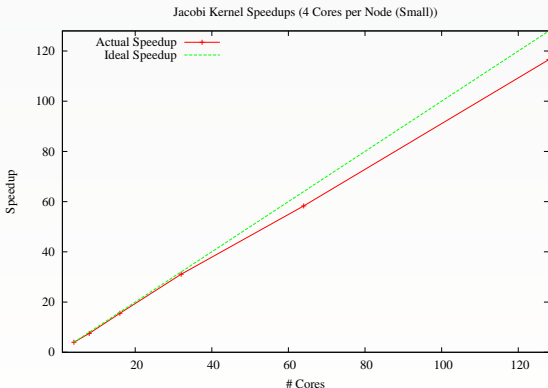
# Outline

- ① Programmation pour architectures hybrides
- ② dSTEP
- ③ Expérimentations
- ④ Conclusion

# Jacobi, Multi-CPU, Small

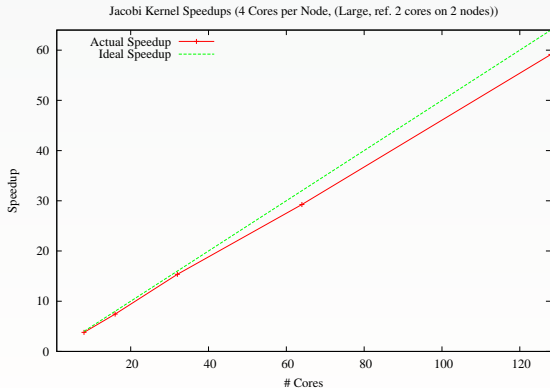
Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see <https://www.grid5000.fr>).

- Matrices  $65536 \times 1024$ , double précision
- 32 noeuds  $\times$  4 coeurs (Intel Xeon E5520 2.27 GHz, InfiniBand, GCC 4.4.5)



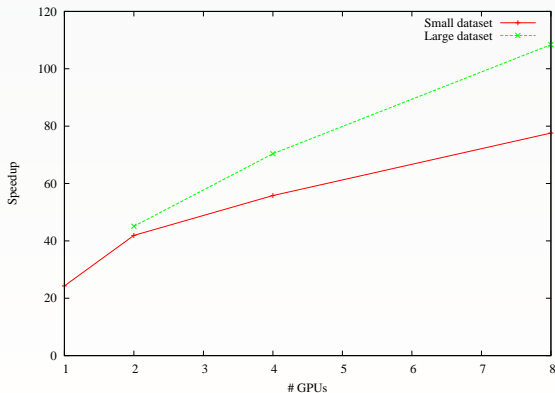
# Jacobi, Multi-CPU, Large

- Matrices  $131072 \times 16000$ , double précision
- 32 noeuds  $\times$  4 coeurs (Intel Xeon E5520 2.27 GHz, InfiniBand, GCC 4.4.5)



# Jacobi, Multi-GPU

- Grappes de PCs équipés de GPUs, Ethernet
- Intel Xeon CPU (2.80 GHz), GCC 4.6.2)
- Nvidia Quadro 2000, NVCC 4.2
- Small :  $8192 \times 4096$ , double précision
- Large :  $8192 \times 8192$ , double précision
- Speedup par rapport à un coeur CPU



# Outline

- ① Programmation pour architectures hybrides
- ② dSTEP
- ③ Expérimentations
- ④ Conclusion

# Conclusion

## *dSTEP*

- Compilateur source à source
- Applications scientifiques denses
- Contrôle de la distribution des données et des calculs
- Traitement de plusieurs dimensions, calculs en front d'onde, ...
- Communications automatiques efficaces
- Extension aux GPUs
- Prototype *dSTEP*

Merci de votre attention !

Questions ?