Le modèle de programmation InKS

Ksander Ejjaaouani

Compas'2018 : Parallélisme / Architecture / Système
ENSEEIHT - Toulouse, France, du 03 au 06 juillet 2018

Le modèle de programmation *InKS*

Ksander Ejjaaouani

Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay
91191 Gif-sur-Yvette, France
INRIA
ksander.ejjaaouani@inria.fr

Résumé
Améliorer la productivité et la maintenabilité des codes de simulation est un enjeu de recherche majeur compte tenu de leur complexification. Une réponse à ce défi est de séparer la sémantique des choix d'exécution. Cette scission facilite la coopération entre chercheurs en permettant à chacun de se concentrer sur un aspect. Dans ce papier, nous proposons un modèle de programmation, *InKS*, qui vise à simplifier le développement des codes en séparant sémantique et choix d'exécution. Le modèle s'accompagne d'un langage décrivant la sémantique. Les choix d'exécution sont ensuite pris en compte automatiquement, par notre compilateur, ou manuellement, par les scientifiques. L'évaluation du modèle *InKS* ne montre pas d'augmentation de la complexité d'implémentation du code ni de réduction des possibilités d'optimisation. Ainsi, le modèle permet d'obtenir les mêmes performances que des applications C standards tout en séparant la sémantique des choix d'exécution.

Mots-clés : HPC, modèle de programmation, séparation des préoccupations.

1. Introduction

De nos jours, la science s'appuie en partie sur la simulation. Des calculateurs parallèles, et parfois hétérogènes, fournissent la puissance de calcul nécessaire à ces simulations. La diversité des architectures et compilateurs force les informaticiens à écrire du code dépendant du calculateur cible (vectorisation, parallélisation, etc.) pour tirer parti de ces architectures. Ces optimisations se mêlent alors aux calculs. Les codes de simulations sont composés de deux parties entrelacées affectant la productivité et la maintenabilité. Le portage d'une application devient difficile. De plus, ce mélange de préoccupations force les scientifiques à devenir experts dans chacun des aspects du code, scientifique et informatique, pour pouvoir développer une application.
Plusieurs approches ont été proposées pour améliorer la productivité. Elles simplifient l'écriture de code en offrant notamment bibliothèques et langages spécifiques à un domaine, mais ces approches mélangent calculs et optimisations ou limitent les possibilités d'exécution.
Nous proposons un modèle de programmation visant à améliorer productivité et maintenabilité : *INdependent Kernel Scheduling*. *InKS* sépare les deux aspects d'un code HPC. D'abord, des chercheurs expriment la *sémantique* du code. Ensuite, des *choix d'exécution* sont mis en place automatiquement, pour tester le code, ou manuellement, pour obtenir une version optimisée.
Le papier s'organise comme suit. La section 2 identifie les aspects sémantique et choix d'exécution pour déterminer les prérequis du modèle. La section 3 présente les travaux connexes. La section 4 présente le modèle de programmation *InKS*. La section 5 évalue l'approche et la section 6 conclut le papier en présentant les axes d'amélioration.

```
1  #define Index3D(_nx,_ny,_i,_j,_k) ((_i)+(_nx)*((_j)+(_ny)*(_k)))
2  double* Anext=(double*)malloc(sizeof(double)*nx*ny*nz);
3  double* A0=(double*)malloc(sizeof(double)*nx*ny*nz);
4  StencilInit(nx,ny,nz,A0);
5  for (int t = 0; t < timesteps; t++){
6    for (int k = 1; k < nz - 1; k++){
7      for (int j = 1; j < ny - 1; j++){
8        for (int i = 1; i < nx - 1; i++){
9          Anext[Index3D (nx, ny, i, j, k)] = A0[Index3D (nx, ny, i, j, k + 1)] +
10         A0[Index3D (nx, ny, i, j, k - 1)] + A0[Index3D (nx, ny, i, j + 1, k)] +
11         A0[Index3D (nx, ny, i, j - 1, k)] + A0[Index3D (nx, ny, i + 1, j, k)] +
12         A0[Index3D (nx, ny, i - 1, j, k)] - A0[Index3D (nx, ny, i, j, k)];  }}}
13   swap(A0, Anext);  }
```

Listing 1 – Équation de la chaleur 3D avec double tampon

```
1  double* A[2] = {A0, Anext};
2  ...
3  A[(t+1)%2][Index3D (nx,ny,x,y,z)] = A[t%2][Index3D (nx,ny,x+1,y,z)] ...
```

Listing 2 – Représentation mémoire pour l'algorithme *Cache Oblivious*

2. Analyse

Dans cette section, nous identifions les deux parties entrelacées qui existent dans un code HPC en utilisant trois implémentations de l'équation de la chaleur 3D, résolue par la méthode des différences finies à 7 points, écrites par S. Kamil [12]. Le listing 1 présente une des implémentations. Elle est basée sur un double tampon. La seconde implémentation parcourt le tableau par bloc sur deux dimensions (*blocking*) pour optimiser l'utilisation du cache. La troisième utilise la récursivité pour créer des blocs imbriqués sur les quatre dimensions (*cache oblivious*).

Dans les trois exemples, des tableaux linéarisés stockent les valeurs de températures (ligne 2 et 3 du listing 1) où un indice est accessible au travers de la macro `Index3D`. Le premier tableau stocke les valeurs de température au temps $t - \Delta t$ et le second celles au pas de temps t. La stratégie *cache oblivious* nécessite un tableau de tableaux (listing 2) ce qui modifie l'accès aux données. La sémantique spécifie l'ensemble de valeurs calculées mais la représentation en mémoire physique est un choix.

Ensuite, les trois implémentations utilisent des boucles. On identifie deux parties dans une boucle. La partie interne contient les calculs. La partie de contrôle donne une valeur aux indexes, ordonnant les calculs de la partie interne. La partie interne des boucles des trois exemples correspond au calcul du stencil à 7 points (ligne 9 à 12) bien que la partie de contrôle soit différente. Les boucles du premier exemple sont écrites *"naïvement"* tandis que dans le second (listing 3) elles créent le *blocking*. Les boucles du dernier exemple sont contrôlées par des paramètres de la fonction récursive. Ces ordonnancements diffèrent mais respectent les contraintes sur l'ordre des calculs du stencil. Cet ordre vient de la méthode de résolution et donc de la sémantique.

Enfin, les exemples terminent lorsque les valeurs au temps final sont obtenues. Parmi toutes les valeurs calculées, on peut considérer ce sous-ensemble comme le résultat du programme.

Finalement, ces trois exemples réalisent les mêmes calculs et partagent le même but, mais l'ordonnancement de ces calculs et la représentation mémoire diffèrent et se mêlent aux calculs.

Dans cette section, nous avons identifié quatre préoccupations formant la sémantique : les valeurs calculées durant l'exécution, les calculs, les contraintes sur l'ordre des calculs et le résultat attendu. Nous avons aussi identifié deux types de choix d'exécution : la représentation mémoire et l'ordonnancement des calculs. Le mélange de la sémantique et des choix d'exécution vient du modèle de programmation C utilisé ici. On souhaite proposer un outil qui sépare ces aspects pour simplifier le

```
for (int t = 0; t < timesteps; t++){
  for (int jj = 1; jj < ny-1; jj+=TJ){
    for (int ii = 1; ii < nx - 1; ii+=TI){
      for (int k = 1; k < nz - 1; k++){
        for (int j = jj; j < MIN(jj+TJ,ny - 1); j++){
          for (int i = ii; i < MIN(ii+TI,nx - 1); i++){
```

Listing 3 – Boucle créant le *blocking* dans le cas de l'équation de la chaleur 3D

développement d'applications HPC sans complexifier l'écriture du code. Cet outil ne doit pas restreindre les possibilités d'exécution ou l'efficacité de l'application. Il doit pouvoir s'associer facilement aux approches classiques, tels que le C ou le Fortran. Enfin, il doit être possible de tester la sémantique sans devoir s'occuper des choix d'exécution.

3. Travaux connexes

Plusieurs types d'approches ont été proposées pour simplifier l'écriture des codes de simulation toujours plus complexes. Un premier type d'approche propose de simplifier la mise en place de certains choix d'exécution. OpenMP [6], par exemple, facilite l'écriture de code parallèle en mémoire partagée. Le modèle PGAS, au travers du langage UPC [11] ou X10 [7], fait partie de ce type d'approche : la mémoire distribuée est invisible. Ces outils sont efficaces et permettent de s'affranchir de certains aspects, mais sémantique et choix d'exécution restent fortement liés. Par exemple, OpenMP est ajouté à un code séquentiel mélangeant déjà ces aspects.
D'autres approches séparent, en partie seulement, la sémantique et les choix d'exécution. Kokkos [5] offre des représentations mémoires et des itérateurs, facilitant l'utilisation de calculateurs hétérogènes. La partie représentation des données est séparée mais l'ordre des calculs est fixé. Par exemple, une opération matricielle telle que $R = A * B + C$ peut aussi s'écrire en deux étapes : $R1 = A * B$ puis $R2 = R1 + C$. Bien que sémantiquement équivalent, la première notation mène à l'écriture d'un nid de boucles tandis que la seconde en demande deux. StarPU [1] ou Legion [3] rendent la distribution des tâches transparentes mais obligent ces tâches à être suffisamment longues pour limiter le surcoût lié à l'environnement d'exécution.
Un dernier type d'approche, représenté par PATUS [8], PIPES [13], Listz [10] ou Nabla [4], propose un langage décrivant la sémantique et, éventuellement, des stratégies d'optimisations. Un compilateur ou un environnement d'exécution assurent ensuite de bonnes performances. Ces approches sont efficaces et améliorent largement la productivité. Par exemple, PIPES génère un code efficace de plusieurs milliers de lignes à partir d'une courte spécification. Cependant, un compilateur ne peut offrir les meilleures performances dans toutes les situations et est difficilement modifiable par son utilisateur.

4. Le modèle de programmation *InKS*

Pour séparer sémantique et choix d'exécution, nous proposons le modèle de programmation *InKS*. Le modèle s'appuie sur le langage *InKS* permettant aux scientifiques du domaine d'exprimer la sémantique d'une application sans tenir compte des choix d'exécution et pour une grande catégorie d'applications. Plus spécifiquement, le modèle considère l'ensemble des programmes pouvant être modélisés par un graphe paramétré par des constantes connus à l'exécution, appelées *données structurantes*. Dans ce graphe, chaque nœud est un calcul et chaque arrête une dépendance entre ces calculs, ce qui s'approche du concept de *Graphe de tâches paramétré* [9]. Dans un second temps les choix d'exécution peuvent être considérés via deux approches. La première, appelée *générique*, est basée sur un compilateur pour préserver les scientifiques du domaine de choix d'exécution com-

```
1  kernel Boundary(x, y, z, t) : ( H {in: (x, y, z, t-1) | out : (x, y, z, t)} )
2  #CODE (C++)
3    H(x, y, z, t) = H(x, y, z, t-1);
4  #END
5  kernel Inner(x, y, z, t) : (
6    H {in: (x, y, z, t-1); (x-1:x+2, y, z, t-1); (x, y-1:y+2, z, t-1);
7       (x, y, z-1:z+2, t-1); | out : (x, y, z, t)},
8    fac  {in} )
9  #CODE (C++)
10   H(x, y, z, t) = H(x+1, y, z, t-1) + H(x-1, y, z, t-1) +  ...
11 #END
12 public kernel inks_heat(nx, ny, nz, n_iter) : (
13   double Heat(4) {out: (0:nx, 0:ny, 0:nz, n_iter-1) }, double fac )
14 #CODE (INKS)
15   Boundary it=[1:INF[ {
16     (0, 0:ny, 0:nz, it); (nx-1, 0:ny, 0:nz, it); (0:nx, 0:ny, 0, it);
17     (0:nx, 0:ny, nz-1, it) ; (0:nx, 0, 0:nz, it); (0:nx, ny-1, 0:nz, it)} : (Heat),
18   Inner (1:nx-1, 1:ny-1, 1:nz-1, 1:INF) : (Heat, fac),
19 #END
```

Listing 4 – Implémentation de l'équation de la chaleur 3D avec le modèle *InKS*

plexes et permet de tester leurs modèles. La seconde, que l'on nomme *spécialisée*, permet, à partir de la sémantique, d'implémenter manuellement des choix d'exécution performants. Nous voulons permettre aux chercheurs d'utiliser le modèle sur tout ou une partie d'une application, souvent composée de milliers de lignes. Le résultat peut alors s'intégrer à une application plus grande. Pour l'instant, la sémantique et les choix d'exécutions sont écrits en C++ pour faciliter l'implémentation du compilateur. Le modèle vise à améliorer productivité et maintenabilité des codes sans perte de fonctionnalités ou de performance.

4.1. La partie sémantique

Dans le modèle *InKS*, un programme est un ensemble de données et de noyaux de calculs effectuant un calcul en utilisant et générant ces données. Cette section se base sur une mise en œuvre *InKS* de l'équation de la chaleur 3D (listing 4) pour présenter les concepts et éléments du langage.

En *InKS*, les données sont stockées dans des tableaux logiques multidimensionnels infinis reposant sur le principe du *dynamic single assignment* [14], appelés *données de valeurs*. Cette séparation entre espace logique et physique permet de ne pas se préoccuper de l'implémentation sous-jacente lors de l'utilisation de ces tableaux. Un tableau, nommé `Heat`, à quatre dimensions (positions et temps) est nécessaire pour stocker toutes les valeurs créées durant la simulation (ligne 13 du listing 4).

Un calcul est décrit par un `noyau` *InKS* en C++. Un noyau utilise deux types de paramètres : des *données de valeurs* et des *données structurantes* sous la forme d'un n-uplet d'entiers utilisé en tant que coordonnée pour accéder aux valeurs des tableaux, via son opérateur parenthèse. Ici, le noyau `Inner` (ligne 5) utilise le tableau `Heat` et le quadruplet `(x, y, z, t)` pour le calcul du stencil.

Des dépendances s'ajoutent aux noyaux en utilisant *les données structurantes* pour spécifier les données impliquées dans les entrées et sorties d'un noyau. Ici, le noyau `Inner` génère une valeur à partir de sept autres. Les lignes 6 et 7 expriment cette contrainte, où 7 points sont définis comme entrée `in` et un en sortie `out`. L'opérateur *':'* définit un ensemble d'entiers consécutifs. Ainsi, `(A:B)` $\leftrightarrow [\![A\,;\,B]\![$. Par commodité, on supporte la définition de variables stockant cet ensemble (ligne 15).

Enfin, un programme *InKS* comporte au moins un `noyau public` contenant du code *InKS*. Il s'agit d'un noyau définissant des tableaux logiques et appelant des noyaux précédemment définis pour générer ses sorties. Ce noyau public utilise également des *données structurantes*. En utilisant ces données, il spécifie le *domaine de validité* de chaque noyau appelé. Il s'agit de l'ensemble des valeurs que peuvent prendre les *données structurantes* du noyau. Le domaine de validité d'`Inner` (ligne 18) équi-

```
1  #include "inks_kernels.hpp" //contient les noyaux inks
2  #include "InKSArray.h"      //facilite l'ecriture du code
3  template <typename T1>
4  void inks_heat(T1& Heat){ //parametre, surcharge l'operateur ()
5  for(int t=1; t<timesteps; t++){
6    for(int z=1; z<nz-1; z++){
7      for(int y=1; y<ny-1; y++){
8        for(int x=1; x<nx-1; x++){
9          Inner(Heat, x, y, z, t); }}}
10   Heat.swap();  } }
```

Listing 5 – Implémentation *specialisée* des choix d'exécution à partir de la sémantique *InKS* de l'équation de la chaleur 3D

vaut à : $\forall x,y,z,t \in \mathbb{N}^4,\ Inner(x,y,z,t) \implies (0<x<nx-1)\wedge(0<y<ny-1)\wedge(0<z<nz-1)\wedge(0<t)$
Le noyau public *InKS* est une boîte noire qui utilise des données (entrées) pour en créer (sorties). Dans l'exemple de l'équation de la chaleur, on veut toutes les valeurs au temps final. Pour cela, on définit ce sous-ensemble du tableau `Heat` comme sortie du noyau (ligne 13).

4.2. Exécution du code *InKS*

Une option de notre compilateur permet de transformer les noyaux `C++` en fonctions C++ valides. Ainsi, pour exécuter ce code, il faut écrire les choix d'exécution : la représentation mémoire et l'ordonnancement des noyaux en respectant les relations de dépendance pour obtenir les valeurs attendues. Ces choix peuvent être écrit automatiquement par notre compilateur, grâce à une analyse polyédrique basée sur *isl* [15]. L'approche automatique, ou *générique*, permet aux scientifiques du domaine de tester la sémantique. D'abord, des nids de boucles appelant les noyaux dans un ordre valide, dans le but de produire les valeurs attendues, seront inférés de la sémantique *InKS*. Ensuite, une représentation mémoire respectant cet ordonnancement sera calculée avec pour objectif de minimiser la quantité de mémoire utilisée. La description du compilateur et de son implémentation est disponible en [2]. Une seconde méthode se base sur l'expertise des spécialistes de l'optimisation : l'approche *spécialisée*, ou manuelle. Ils peuvent inclure le fichier contenant les fonctions C++ dans un fichier C++ standard puis mettre en place les choix d'exécution manuellement. Ainsi, les noyaux sont des boîtes noires dont la compréhension n'est pas nécessaire. Pour ne pas réduire les possibilités d'ordonnancement, les noyaux dans la sémantique doivent s'appliquer à la granularité la plus fine. Concernant l'accès aux indices, le compilateur utilise une classe qui contient un tableau linéarisé et surcharge l'opérateur parenthèse utilisant le n-uplet d'entiers pour accéder aux indices. L'approche *spécialisée* doit utiliser la même stratégie, mais avec une représentation possiblement différente. Par exemple, un tableau alloué en colonne avec l'accesseur associé.
Le listing 5 montre une implémentation *spécialisée* à partir de la sémantique du listing 4. Il montre aussi une des possibilités du modèle *InKS* : en utilisant la sémantique, il peut fournir des outils (classes, fonctions, etc.) qui facilitent l'écriture des choix d'exécution. Ici, il s'agit d'une classe qui contient un tableau linéarisé et surcharge l'opérateur parenthèse (ligne 2 et 4).

5. Évaluation

Pour valider notre approche, nous avons décrit la sémantique de l'équation de la chaleur 3D en *InKS*. Cette sémantique est ensuite conservée pour trois implémentations : une automatique par notre compilateur et deux *specialisées* (dont le listing 5) équivalentes aux implémentations *double buffer* et *blocking* présentées en 2. Nous avons évalué la complexité des versions *spécialisées* et des versions de référence en utilisant GNU complexity score. Ce score se base notamment sur la complexité cyclomatique. Pour les évaluations de performances, nous avons compilé tous les codes avec `Intel 15`

	Durée par itération		Score GNU Complexity	
Version	Référence	*InKS*	Référence	*InKS*
Générique	N/A	4.09 s (±1.27%)	N/A	0
Double buffer	2.93 s (±0.95%)	2.98 s (±1.94%)	5	3
Cache oblivious	3.47 s (±0.08%)	2.21 s (±2.71%)	22	13

TABLE 1 – Durée par itération et complexité de cinq versions de l'équation de la chaleur 3D sur un cas ($1024 \times 1024 \times 1024$). La durée est une médiane de 12 exécutions de 15 pas de temps avec la différence maximale observée entre parenthèses.

et les options `-O3 -ipo -fpic -fno-exceptions`, puis les avons exécutés sur un cœur d'un Intel Xeon E5-2670 avec 32 GB de RAM. Les résultats sont présentés en table 1.
Le modèle *InKS* sépare suffisamment sémantique et choix d'exécution pour nous permettre d'implémenter plusieurs choix d'exécution différent à partir d'un unique fichier de sémantique. Par ailleurs, les versions *spécialisées* sont aussi performantes que les versions de référence. Notons que d'après de plus récents tests, l'efficacité de la version *cache oblivious* de référence serait due à notre ancienne version du compilateur. Il est difficile de proposer une mesure cohérente pour évaluer la simplicité d'écriture de la sémantique. Cependant, le code du Listing 4 est très proche de l'implémentation la plus naïve possible en C ou Fortran.L'utilisation du C++ via l'approche *spécialisée* permet aux spécialistes de l'optimisation d'utiliser leurs compétences comme à leur habitude. En outre, le score de complexité n'est pas plus haut dans les versions *spécialisées* que dans les versions de référence. Tout cela montre, dans une certaine mesure, qu'*InKS* ne complexifie pas l'écriture du code tout en ne limitant pas les possibilités d'optimisation. Par ailleurs, notre compilateur permet de tester la validité de la sémantique sans avoir à s'occuper de choix d'exécution.
Le résultat de l'approche *générique* et *spécialisée* est une fonction C++ standard. Ainsi, cette fonction peut être appelée par un programme plus grand implémenté dans un langage supportant la convention d'appel C, tels que le C, le C++, le Fortran ou le Python.
Cette évaluation basée uniquement sur l'équation de la chaleur ne permet pas de discuter des applications d'*InKS* sur de nombreux domaines. Néanmoins, le langage *InKS* ne fait pas de supposition sur le domaine simulé et offre une abstraction plus proche de notions informatique que relative à un domaine. Si un programme peut être modélisé par un graphe paramétré à l'exécution, il peut être exprimé en *InKS*. Ainsi, bien que la version actuelle empêche l'expression de maillages adaptatifs ou l'arrêt du programme après convergence, par exemple, il est possible d'inclure le résultat d'*InKS* dans un programme plus grand gérant ces fonctions. De cette façon, on peut avoir un appel du résultat d'*InKS* à chaque test de convergence par exemple.

6. Conclusion et perspectives

Les applications HPC sont de plus en plus complexes et le mélange de la sémantique et des optimisations aggrave le problème. Nous avons présenté le modèle de programmation *InKS* qui offre une séparation de ces aspects. À l'heure actuelle, le langage *InKS* permet aux scientifiques d'exprimer la sémantique de leur application. Ensuite, les choix d'exécution sont écrit manuellement ou par un compilateur. L'objectif des prochaines recherches est de proposer aux spécialistes de l'optimisation de n'écrire que les choix d'exécution via des outils utilisant la sémantique comme support. L'objectif étant de renforcer la séparation entre les différents aspects, initié par ce travail préliminaire, pour faciliter le développement des codes de simulation.

Bibliographie

1. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – Starpu : A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, vol. 23, n 2, février 2011, pp. 187–198.
2. Aumage (O.), Bigot (J.), Ejjaaouani (K.) et Mehrenberger (M.). – *InKS, a programming model to decouple performance from semantics in simulation codes*. – Rapport technique, Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay, Inria, Inria-LaBRI, IRMA, Université de Strasbourg, France, 2017. to appear.
3. Bauer (M.), Treichler (S.), Slaughter (E.) et Aiken (A.). – Legion : Expressing locality and independence with logical regions. – In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, SC '12, pp. 66 :1–66 :11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
4. Camier (J.-S.). – Improving performance portability and exascale software productivity with the &nabla ; numerical programming language. – In *Proceedings of the 3rd International Conference on Exascale Applications and Software, EASC '15*, EASC '15, pp. 126–131, Edinburgh, Scotland, UK, 2015. University of Edinburgh.
5. Carter Edwards (H.), Trott (C. R.) et Sunderland (D.). – Kokkos. *J. Parallel Distrib. Comput.*, vol. 74, n12, décembre 2014, pp. 3202–3216.
6. Chandra (R.), Dagum (L.), Kohr (D.), Maydan (D.), McDonald (J.) et Menon (R.). – *Parallel Programming in OpenMP*. – San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2001.
7. Charles (P.), Grothoff (C.), Saraswat (V.), Donawa (C.), Kielstra (A.), Ebcioglu (K.), von Praun (C.) et Sarkar (V.). – X10 : An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, vol. 40, n10, octobre 2005, pp. 519–538.
8. Christen (M.), Schenk (O.) et Burkhart (H.). – PATUS : A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. – In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 676–687. IEEE, mai 2011.
9. Danalis (A.), Bosilca (G.), Bouteiller (A.), Herault (T.) et Dongarra (J.). – Ptg : An abstraction for unhindered parallelism. – In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14*, WOLFHPC '14, pp. 21–30, Piscataway, NJ, USA, 2014. IEEE Press.
10. DeVito (Z.), Joubert (N.), Palacios (F.), Oakley (S.), Medina (M.), Barrientos (M.), Elsen (E.), Ham (F.), Aiken (A.), Duraisamy (K.), Darve (E.), Alonso (J.) et Hanrahan (P.). – Liszt : A Domain Specific language for Building Portable Mesh-based PDE Solvers. – *SC '11*, SC '11, pp. 9 :1–9 :12, New York, NY, USA, 2011. ACM.
11. El-Ghazawi (T.), Carlson (W.), Sterling (T.) et Yelick (K.). – *UPC : Distributed Shared Memory Programming (Wiley Series on Parallel and Distributed Computing)*. – Wiley-Interscience, 2005.
12. Kamil (S.). – Stencilprobe : A microbenchmark for stencil applications. – `http://people.csail.mit.edu/skamil/projects/stencilprobe/`. Accessed : 21-02-2017.
13. Kong (M.), Pouchet (L.-N.), Sadayappan (P.) et Sarkar (V.). – Pipes : A language and compiler for task-based programming on distributed-memory clusters. – In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16*, SC '16, pp. 39 :1–39 :12, Piscataway, NJ, USA, 2016. IEEE Press.
14. Vanbroekhoven (P.). – Dynamic single assignment, 2002.
15. Verdoolaege (S.). – *isl : An Integer Set Library for the Polyhedral Model*, pp. 299–302. – Berlin, Heidelberg, Springer Berlin Heidelberg, 2010.