Expériences de développement logiciel en langage de programmation X10

Marc Tajchman

CEA - DEN/DM2S/SFME/LGLS

Séminaire de l'IDRIS du 11 février 2010

X10 : présentation

Programmation “from scratch”

Appel de codes séquentiels C/C++/fortran

Programmation hybride X10 - MPI

Retour d’expérience et conclusions

X10 : présentation

Programmation “from scratch”

Appel de codes séquentiels C/C++/fortran

Programmation hybride X10 - MPI

Retour d'expérience et conclusions

X10 : présentation générale

Langage de programmation développé au sein d'IBM Research.
Réponse d'IBM à un appel du DARPA (US) pour un langage de programmation parallèle de nouvelle génération.
Autre proposition retenue par le DARPA : langage Chapel de Cray.

- Orienté-objet (syntaxe basée sur Java).

  X10 s'écarte de plus en plus de java à chaque version (pour les besoins du //).

- Structures de données distribuées (avec adressage global)

  S'appuie sur le modèle de programmation PGAS (adressage global partitionné) comme d'autres langages (Chapel, UPC, Co-Array Fortran).

- Parallélisme de tâches locales et/ou distantes

X10 : programmation séquentielle

```
1  class A {
2    var x0:double; //donnees des objets de classe A
3
4    //constructeur de A
5    public def this (n:int) {
6      x0 = n+1.5;
7    }
8    //methode run de A
9    public def run(x:double) {
10     return x*2.0 + x0;
11   }
12   //fonction statique(par ex.programme principal)
13   public static def main(s:Rail[String]!) {
14     var ob:A! = new A(3);
15     d:double = ob.run(3.5);
16     Console.OUT.println("d = " + d);
17    }
18 };
```

X10 : programmation séquentielle (2)

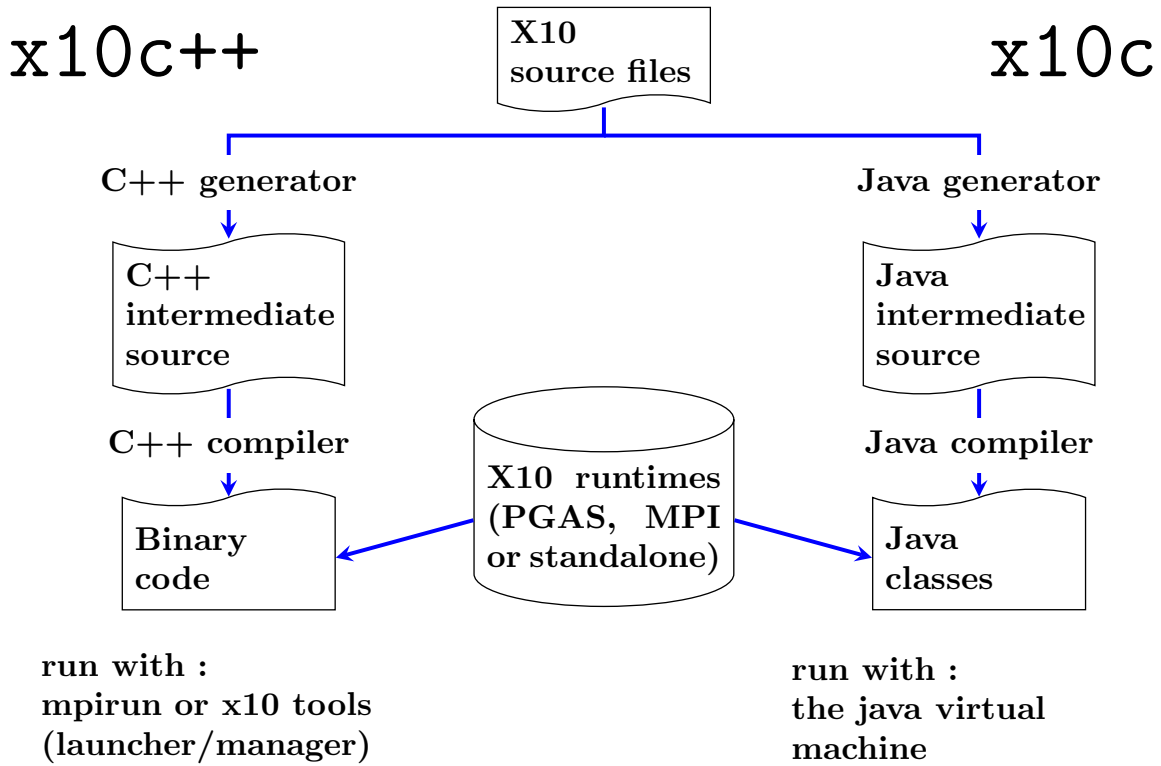
Sur l'exemple:

- Données simples (scalaires, vecteurs)
- Notions orientées-objet : classes, objets, méthodes, membres (données), héritage, ... (similaire à java)
- Ramasse-miettes (garbage collector) : pas de destructeurs (similaire à java)
- Variables (read/write) et constantes ("write once/read many")
- Fonctions statiques

Différences avec java motivées par le // (voir plus loin).

Compilation-exécution du code x10

2 voies pour compiler des fichiers source X10 :



On se limitera à la voie C++ (la voie Java ne gère pas le multi-processus).

X10 // : places et activités

- Place X10 $\sim$ processus MPI : contexte d'exécution avec espace mémoire propre
- Activité X10 $\sim$ thread dans mode MPI/multithreads: fil d'exécution séquentiel dans une place.

Workflow :

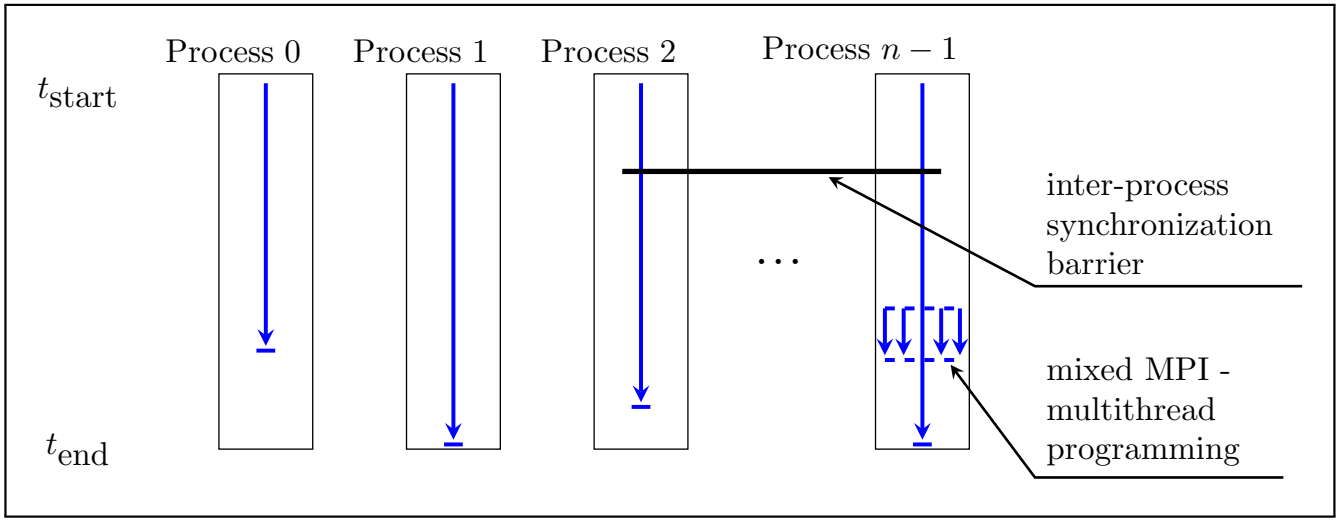
Tous les processus MPI démarrent en même temps. Chaque processus MPI peut lancer des threads locaux.	vs.	Seule la place 0 lance une activité ("root activity"). Une activité peut lancer d'autres activités dans la même place ou des places différentes.

Synchronisation :

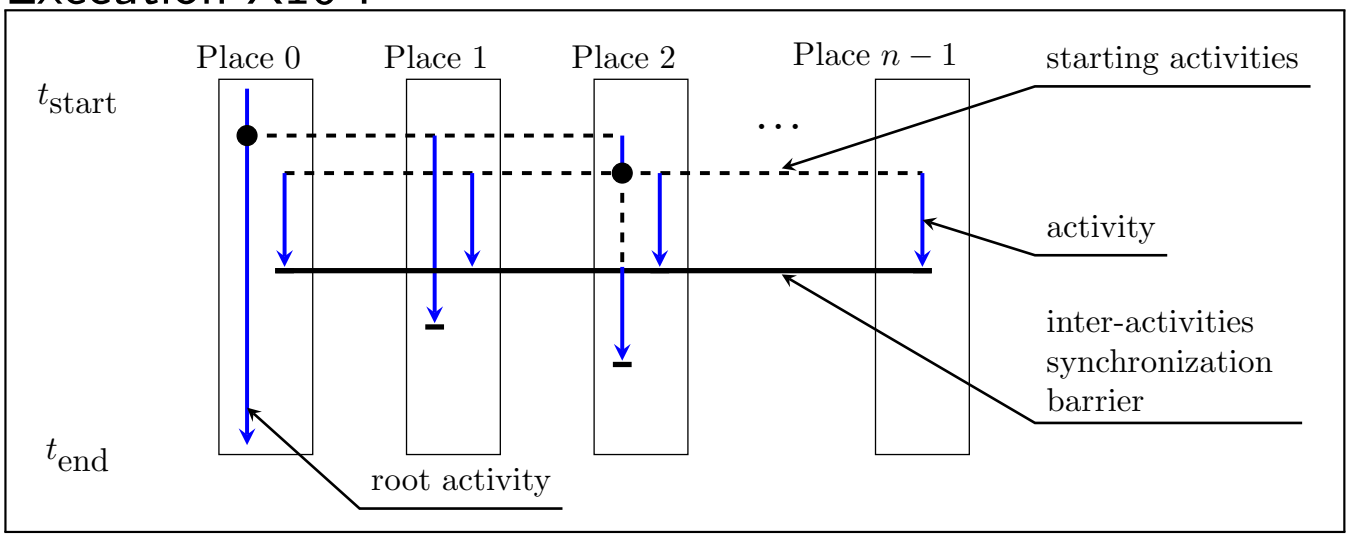
entre **processus MPI** ou entre **threads** dans un même processus (2 mécanismes indépendants)	vs.	entre **activités X10** (colocalisées dans une place et/ou dans des places différentes)

X10 // : places et activités (2)

Exécution MPI :



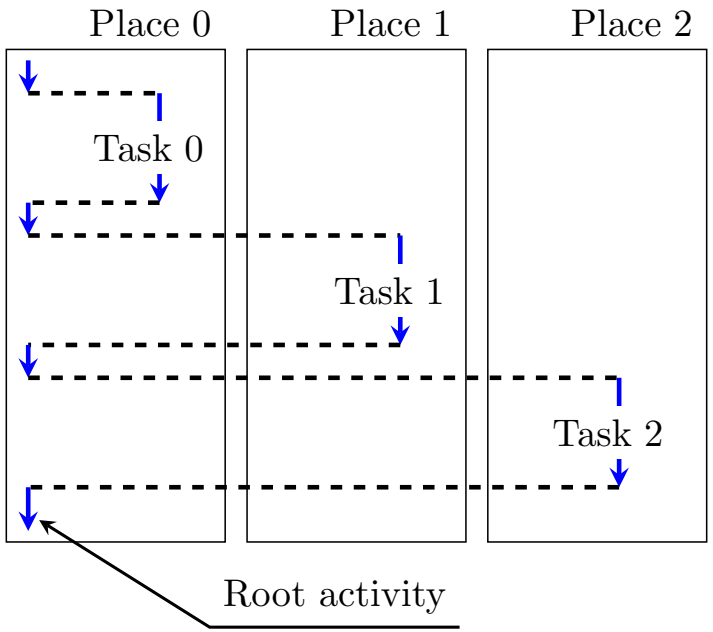
Exécution X10 :



X10 // : places et activités (3)

Activités synchrones (exécution distribuée séquentielle) :

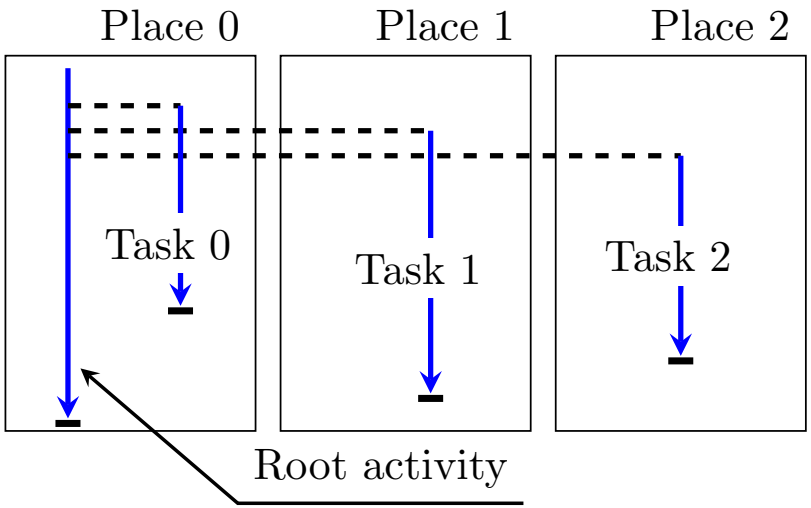
```
1  class ex2 {
2    public static def main(args:Rail[String]!) {
3      for (var i:int=0; i<3; i++)
4        at (Place.places(i))
5          Console.ERR.println("task at " + here);
6    }
7  }
```



X10 // : places et activités (4)

Activités asynchrones (exécution distribuée concurrente) :

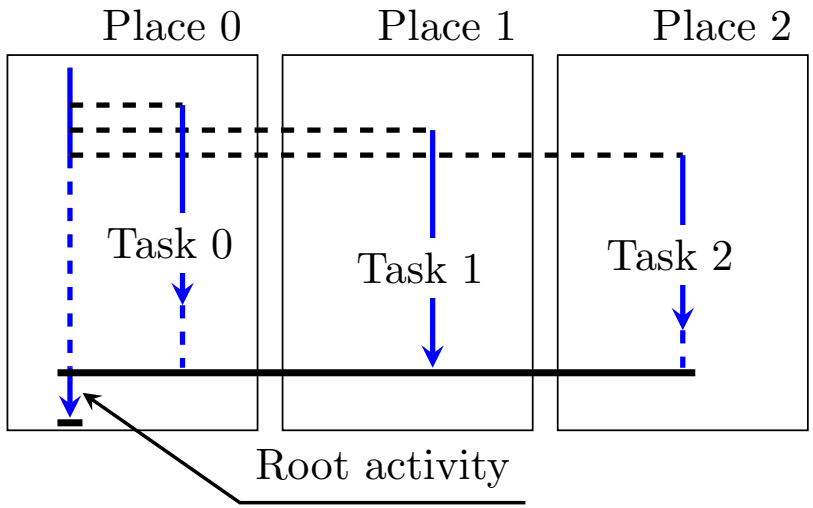
```
1  class ex2b {
2    public static def main(args:Rail[String]!) {
3      for (var i:int=0; i<3; i++)
4        async (Place.places(i))
5          Console.ERR.println("task at " + here);
6    }
7  }
```



X10 // : places et activités (5)

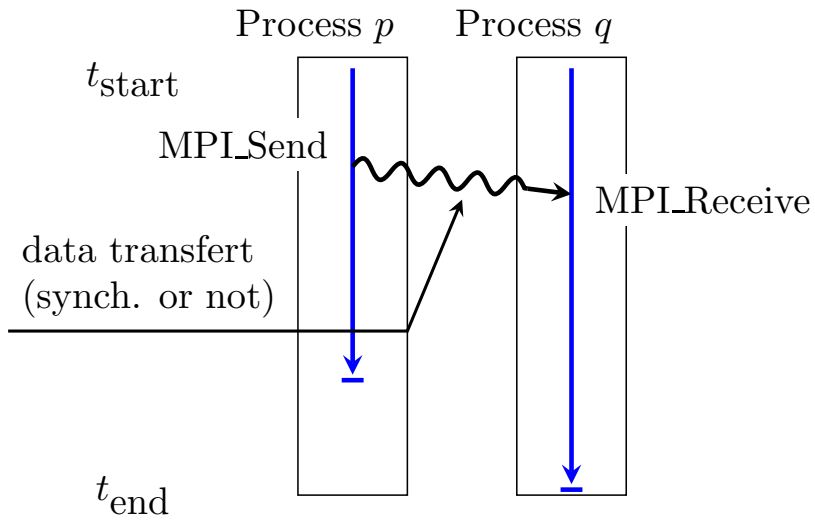
Activités asynchrones avec barrière de synchronisation :

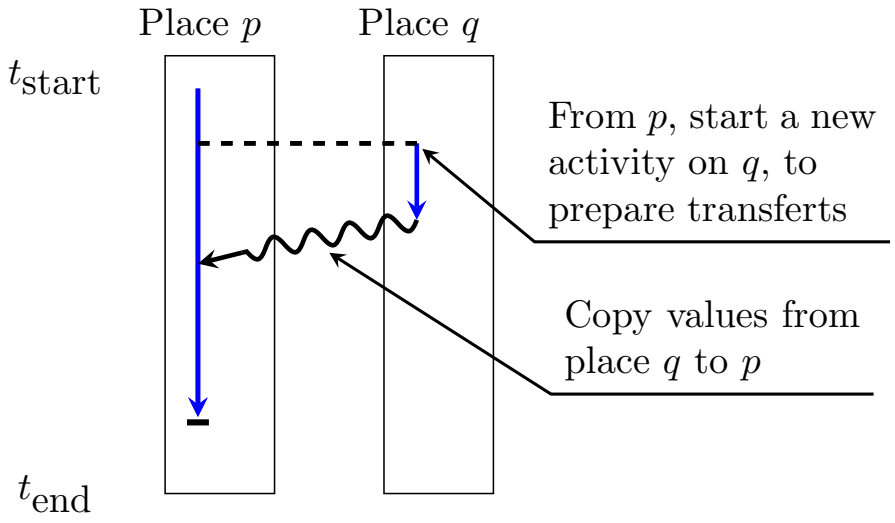
```
1  class ex2c {
2    public static def main(args:Rail[String]!) {
3      finish {
4        for (var i:int=0; i<3; i++)
5          async (Place.places(i))
6            Console.ERR.println("task at " + here);
7        }
8    }
9  }
```



X10 // : principe du transfert de données entre places

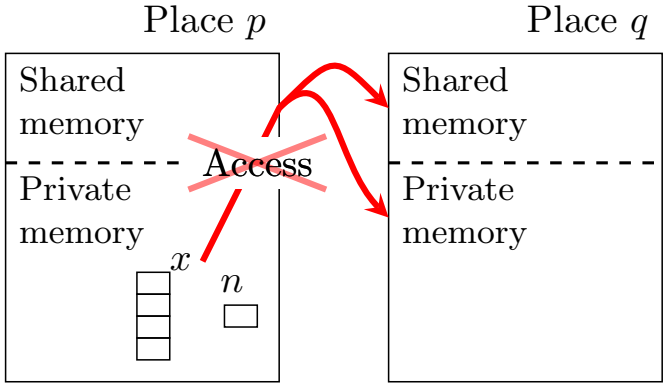
MPI : les 2 processus doivent prévoir d'exécuter une instruction d'envoi ou de réception des données





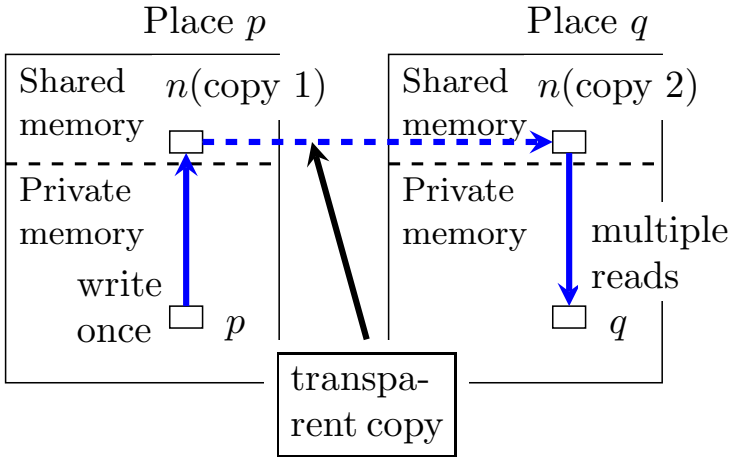
X10 : la place p crée une activité sur la place q pour préparer et tranférer les données.

X10 // : mémoires privée et partagée inter-places



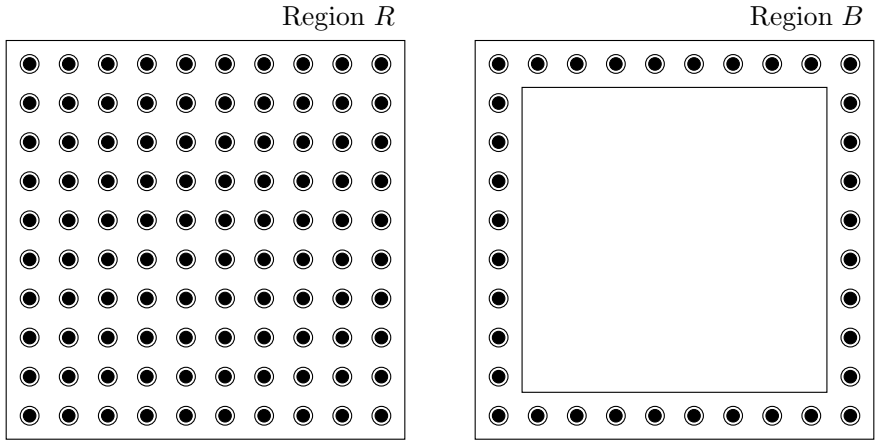

```
1  class ex3a {
2      public static def main(args:Rail[String]!) {
3          var x:Rail[double]! =
4            Rail.make[double](12,(i:int)=>2.0*i);
5          var n:int;
6          n = 17;
7          x(2) = 2.4*x(3);
8          at (Place.place(1)) {
9          // n = n*2;  // error:bad remote access
10         // x(4) = 1.4*x(2);
11         }
12     }
13 }
```

X10 // : mémoires privée et partagée inter-places (2)




```
1  class ex3b {
2      public static def main(args:Rail[String]!) {
3          var p:int = 17;
4          p = p*2;
5
6          val n = p;
7          at (Place.place(1)) {
8            var q:int = n;
9            q = q + 2*n;
10         }
11     }
12 }
```

X10 // : notion de région



Région : ensemble de points dans $\mathbb{Z}^n$ (points à coordonnées entières)

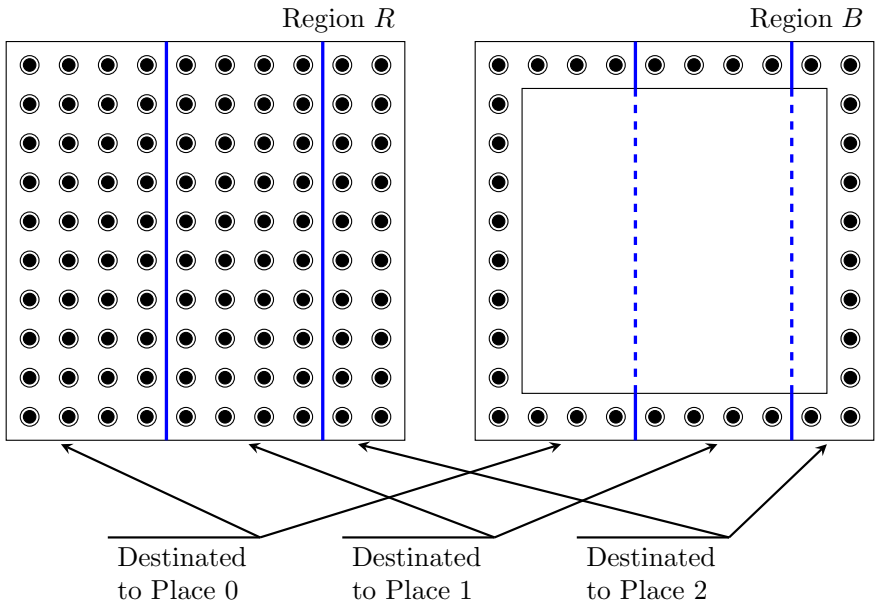
```
1  class ex6a {
2    public static
3    def main(args:Rail[String]!) {
4        var N:int = 9;
5        var R:Region(2) = [0..N, 0..N];
6        var R_in: Region(2) = [1 .. N-1, 1 .. N-1];
7        var B:Region(2) = R - R_in;
8        Console.ERR.println ("R = " + R);
9        Console.ERR.println ("B = " + B);
10   }
11 }
```

Résultat:

```
R=[0..9,0..9]
B=([0..0,0..9]||[1..9,0..0]||[1..9,9..9]||[9..9,1..8])
```

X10 // : notion de distribution

Distribution : indice de place attribué à chaque point d'une région suivant le critère de répartition choisi.



Un certain nombre de critères de répartition standards sont définis dans le langage (suivant I, J, ..., par bloc suivant I, J, ..., cyclique, bloc cyclique, ...).

Cas illustré ici : répartition par bloc suivant le 1^{er} indice.

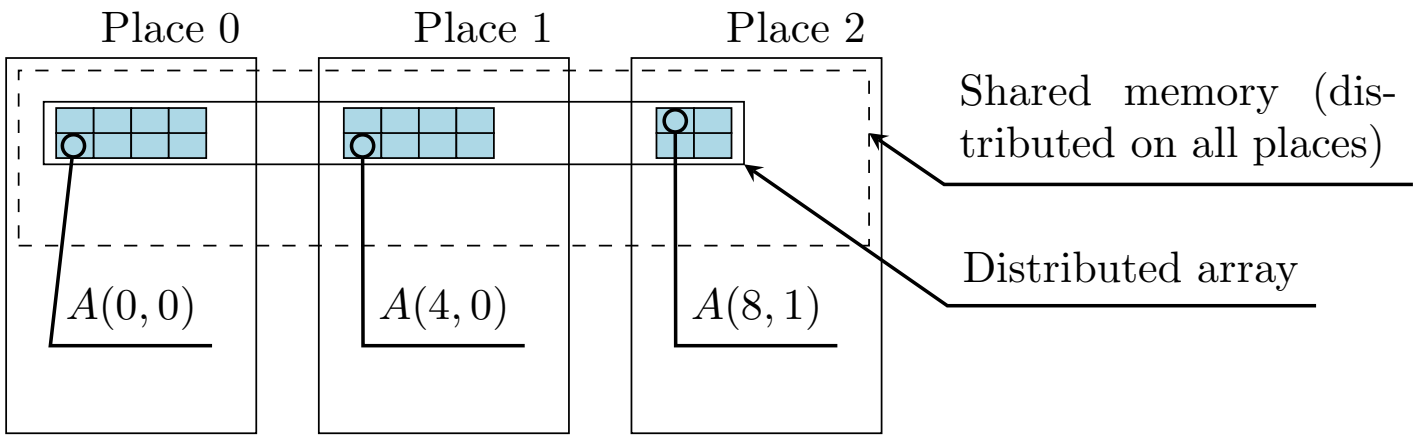
X10 // : notion de distribution (2)

```
 1  class ex6b {
 2    public static def main(args:Rail[String]!) {
 3        var n:int = 9;
 4        var R:Region(2) = [0..n, 0..n];
 5        var R_in: Region(2) = [1..n-1, 1..n-1];
 6        var B:Region(2) = R - R_in;
 7
 8        D : Dist = Dist.makeBlock(R, 0);
 9        Console.ERR.println("D = " + D);
10        E : Dist = Dist.makeBlock(B, 0);
11        Console.ERR.println("E = " + E);
12    }
13  }
```

Résultat (sur 3 places):

```
D = Dist(0->[0..3,0..9],1->[4..7,0..9],2->[8..9,0..9])
E = Dist(0->([0..0,0..9] || [1..3,0..0] || [1..3,9..9]),
         1->([4..7,0..0] || [4..7,9..9]),
         2->([8..9,0..0] || [8..9,9..9] || [9..9,1..8]))
```

X10 // : tableaux distribués et adressage global




```
1  R : Region(2) = [0..9, 0..1];
2  D : Dist = Dist.makeBlock(R, 0);
3  A : Array[double](2) = Array.make[double](D);
4
5  A(0,0) = 1.0;
6  at (Place.places(1)) {
7       A(4,0) = 2.0;
8  }
9  at (Place.places(2)) {
10      A(8,1) = (at(Place.places(1)) A(4,0))+4.5;
11 }
```

Exercice classique : Equation de la chaleur sur $[0,1] \times [0,1]$

- $\frac{\partial u}{\partial t} = -\Delta u$ sur $[0,1] \times [0,1]$
- $u(t \geq 0) = 1$ sur un des côtés, $u(t) = 0$ sur le reste du bord du carré
- $u(t = 0) = 0$ initialement à l'intérieur du carré.
- discrétisation : Euler explicite en temps, maillage régulier et différences finies centrées en espace

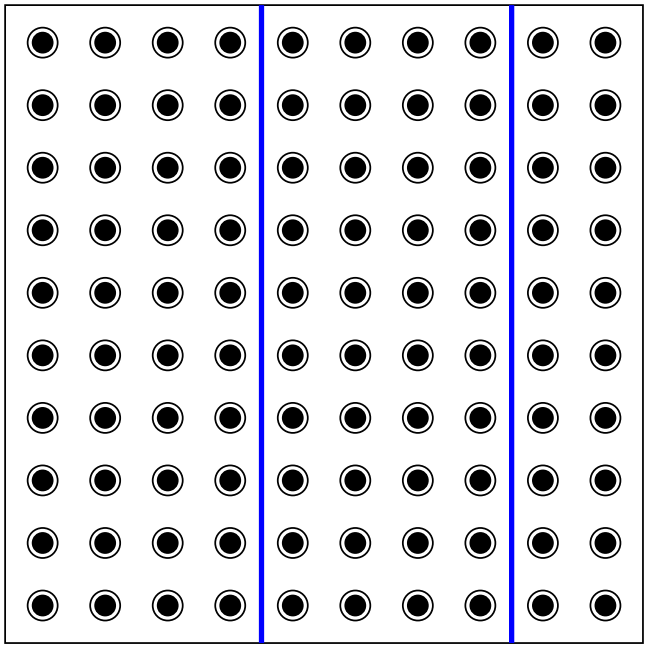
Passage de t_n à $t_{n+1} = t_n + \Delta t$:

$$\begin{aligned} u_{i,j}^{n+1} &= (1 - 4\lambda) u_{i,j}^n \\ &+ \lambda(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n) \end{aligned}$$

où $\lambda = \frac{\Delta t}{\Delta x^2}$ et (i,j) parcourt les indices des points à l'intérieur du carré.

Exercice classique : Equation de la chaleur sur $[0,1] \times [0,1]$

On suppose que les tableaux u et v, de taille $(n+2)^2$ sont répartis sur p places, suivant le schema de principe :



Il y a n^2 points intérieurs au domaine, dont $2(p-1)n$ ont un voisin direct sur une autre place. La taille du premier sous-domaine est $m * n$.

1er : approche “mémoire partagée pure”

La première version de code pour ce cas, utilise la possibilité d’accès uniforme à l’ensemble des coefficients des tableaux.

La boucle principale de calcul exécutée depuis la place 0, s’écrit :

```
 1  var i:int;
 2  var j:int;
 3
 4  finish
 5    for (i=1; i<n+1; i++) {
 6      for (j=1; j<n+1; j++) {
 7        val ii = i;
 8        val jj = j;
 9        async (u.dist(i,j))
10          v(ii,jj) = (1-4*lambda)*u(ii,jj)+lambda*
11           ( (at (u.dist(ii+1,jj)) u(ii+1,jj)) +
12             (at (u.dist(ii-1,jj)) u(ii-1,jj)) +
13             (at (u.dist(ii,jj-1)) u(ii,jj-1)) +
14             (at (u.dist(ii,jj+1)) u(ii,jj+1)));
15      }
16    }
```

1^{er} essai : commentaires

- La boucle est une adaptation directe d'une version séquentielle du code.
- L'écriture ne dépend pas de la distribution de u et v.
- Pour i et j différents, les évaluations sont indépendantes, d'où le lancement d'activités asynchrones (avec une barrière de synchronisation en fin de boucle).
- Quand on évalue $v_{i,j}$, rien ne dit que $u_{i+1,j}$, $u_{i-1,j}$, $u_{i,j+1}$, $u_{i,j-1}$ se trouvent sur la même place.
- Les transferts entre places sont :
  - $2n(p-1)$ transferts d'un coefficient du tableau u
  - $n^2 - mp$ transferts d'un couple d'indices de boucle (i, j)
  - $p - 1$ transferts de la valeur `lambda`
- On crée $5n^2$ activités.

2ème essai : utilisation explicite du découpage 1D

Le type de découpage utilisé implique que $u_{i,j}$, $u_{i,j+1}$ et $u_{i,j-1}$ sont forcement dans la même place (accès direct).

```
 1  var i:int;
 2  var j:int;
 3
 4  finish
 5  for (i=1; i<n+1; i++) {
 6    for (j=1; j<n+1; j++) {
 7       val ii = i;
 8       val jj = j;
 9       async (u.dist(i,j))
10          v(ii, jj) = (1-4*lambda)*u(ii,jj) + lambda*
11             ((at (u.dist(ii+1,jj)) u(ii+1,jj)) +
12              (at (u.dist(ii-1,jj)) u(ii-1,jj)) +
13               u(ii,jj-1) +
14               u(ii,jj+1));
15       }
16  }
```

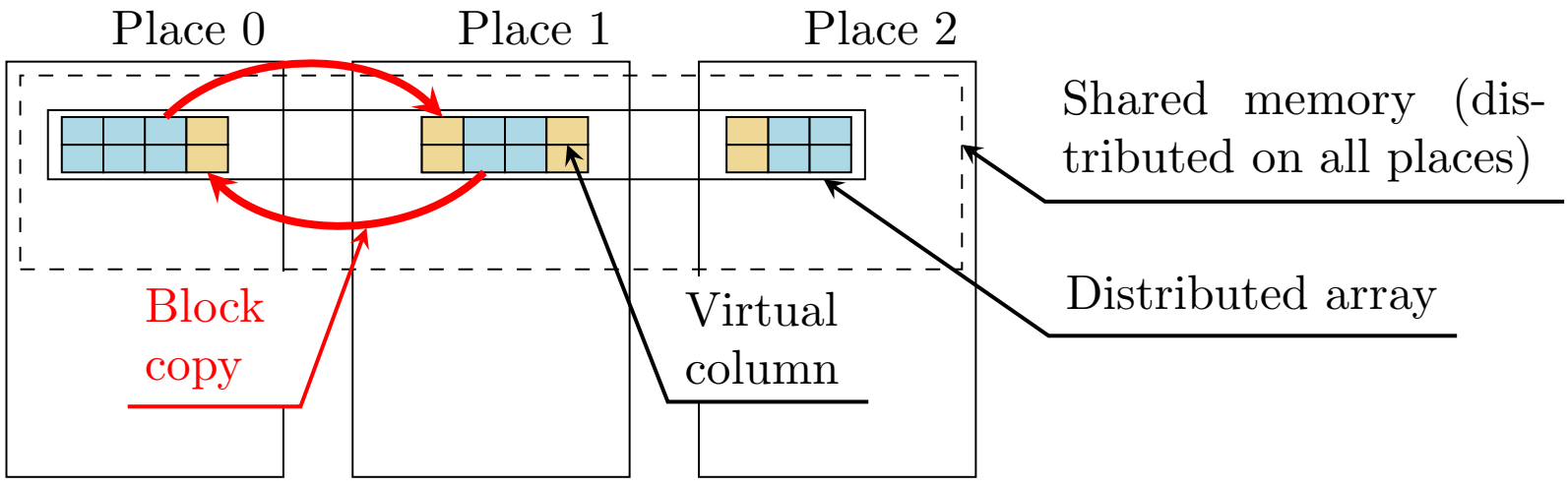
3ème essai : utilisation des opérations sur les régions

Au lieu de parcourir globalement tous les coefficients de u, on démarre une activité sur chaque place et on parcourt la sous-région (restriction à la place courante de la région globale). Le but est de diminuer le transfert d'indices.

```
 1  val m = n;
 2
 3  for (p:Place in Place.places)
 4    at (p) {
 5      var R  : Region(2) = (u.dist | here).region;
 6      R = R.intersection([1..m, 1..m]);
 7      for ((i,j):Point(2) in R)
 8        v(i, j) = (1-4*lambda)*u(i,j) + lambda*
 9              ( (at (u.dist(i+1,j)) u(i+1,j)) +
10                (at (u.dist(i-1,j)) u(i-1,j)) +
11                 u(i,j-1) +
12                 u(i,j+1));
13      }
```

4ème essai : zones virtuelles et transferts par bloc

On ajoute à chaque sous-région une colonne à la frontière avec les régions voisines.



La recopie des valeurs à l'interface se fait par bloc. On remplace donc n transferts d'un scalaire par un transfert de n scalaires.

4ème essai (2) : calcul local

Ces “zones virtuelles” permettent d’utiliser un algorithme de calcul uniquement local à l’intérieur de chaque sous-région.

```
 1  finish
 2    for (var i:int=0; i<Place.MAX_PLACES; i++)  {
 3      val ii = i;
 4      async (Place.place(i)) {
 5        val j0:int = ii*(l+2)+1;
 6        val j1:int = j0+l;
 7        foreach ((j,k) : Point in [j0..j1-1, 1..n]) {
 8          v(j,k) = (1-4*lambda)*u(j,k) + lambda*
 9              ( u(j+1, k) +
10                u(j-1, k) +
11                u(j, k-1) +
12                u(j, k+1) );
13            }
14          }
15      }
```

4ème essai (3) : échange par bloc

```
1  for (var i:int=1; i<Place.MAX_PLACES; i++) {
2      val i0 = i*(1+2);
3
4      val v = at (Place.places(i-1))
5          ValRail.make[double](n,(j:Int)=>u(i0-2,j+1));
6
7      at (Place.places(i))
8          for (var j:int=0; j<n; j++)
9              u(i0,j+1) = v(j);
10
11     val w = at (Place.places(i))
12         ValRail.make[double](n,(j:Int)=>u(i0+1,j+1));
13
14     at (Place.places(i-1))
15         for (var j:int=0; j<n; j++)
16             u(i0-1,j+1) = w(j);
17 }
```

Comparaison de performance des diff. versions

Calcul sur 2 places (sur un ordinateur bi-cœur).

versions	temps calcul (s)
1. “mémoire partagée pure”	46.979
2. suppression d’activités	45.427
3. opérations sur les régions	45.983
4. transferts par bloc	0.253

Le cas est trop simple pour en déduire des conclusions définitives.

Le principal enseignement est qu’il faut absolument grouper les transferts par bloc.

Si l’utilisation d’opérations sur les régions n’apporte pas d’amélioration, cela est sans doute dû aux variables temporaires (de type Point) générées.

Appel de fonction C++

X10 prévoit la possibilité d'appeler du code C++ quand on utilise le générateur interne C++ (et d'appeler du code java quand on utilise le générateur java).
Cette fonctionnalité est très peu documentée. Une des sources d'information est le source des runtimes X10 qui utilisent ce mécanisme.

- la fonction C++ sera vue depuis le code X10 comme une fonction membre d'une classe ou statique (hors classe), avec l'annotation "Native",
- on fournit une chaîne de caractères qui sera insérée dans le code C++ généré et qui donne la déclaration de la fonction et la façon de l'appeler,
- les arguments seront de type simple (scalaire ou vecteur de types simples) et peuvent être de type entrée ou sortie,
- les variables passées sont locales (non distribuées),
- il n'est pas possible actuellement de récupérer une valeur de retour.

Exemple

```
1  import x10.compiler.Native;
2
3  class ex8 {
4
5    @Native("c++","void HelloWorld(x10_int,x10_double);"
6                  + "HelloWorld(#1,#2)")
7    native def test(i:Int, d:Double):void;
8
9    public static def main(s: Rail[String]!) {
10     val D = Dist.makeUnique();
11     val H = Array.make[ex8](D);
12
13     for(var j:int =0; j<Place.MAX_PLACES; j++)
14       async(Place.places(j)) {
15         Console.OUT.println("Hello from X10 "+here)
16         H(here.id()).test(here.id(),3.5*here.id());
17       }
18   }
19 }
```

Exemple (suite)

```
1  #include <iostream>
2  #include <x10/lang/Integer.h>
3  #include <x10/lang/Double.h>
4
5  using namespace x10::lang;
6
7  void HelloWorld(x10_int n, x10_double d)
8  {
9    double dd = d * 2.0;
10   std::cout << "Hello from C n = " << n
11             << " 2*d = " << dd << std::endl;
12 }
```

Appel de code C ou fortran

Appeler des routines C ou fortran (77) depuis C++ ne pose pas de problème (il faut seulement faire attention au mode de passage des arguments et à utiliser “extern ”C”” ...).

Il sera donc facile d’appeler ces routines depuis X10.

Types de données utilisateur

La principale difficulté consistera à lire ou écrire des structures C/C++ ou classes C++ depuis X10.

Le moyen utilisé dans cette étude (voir la section sur la programmation hybride X10-MPI) a consisté à :

- cacher les structures C/classes C++ dans un pointeur "boîte noire" (type void *) et transmettre ce pointeur à X10 sous la forme d'un entier suffisamment long pour ne pas perdre d'information,
- écrire un jeu de fonctions d'accès (type "get/set") pour extraire les données internes des ces structures; ces fonctions d'accès devront "retransformer" les pointeurs "boîte noire" en pointeurs sur des classes/structures,
- en ce qui concerne les classes C++, écrire une classe X10 "miroir" (ayant les même méthodes que la classe interne).

Il est possible que des versions futures de X10 spécifient plus complètement la marche à suivre.

Compatibilité MPI - X10

Au moment où cette étude à commencé la compatibilité X10 - MPI n'était pas évoquée.

On s'est mis "du côté de la sécurité" pour :

- choix du runtime X10 (runtime MPI)
- choix de l'implémentation de MPI, en particulier, niveau de compatibilité avec les threads (OpenMPI, niveau MPI_THREAD_MULTIPLE)
- pas d'utilisation de threads côté code MPI

Dans les versions les plus récentes de X10, il est mentionné la l'amélioration de la compatibilité MPI - X10, sans plus de précision. Il est probable que certaines restrictions ne sont, ou ne seront, plus valables.

Exemple de contrôle depuis X10 d'un code // MPI

On a choisi d'utiliser depuis X10, `pARMS` (`http://www-users.cs.umn.edu/~saad/software/pARMS`), une librairie de solveurs linéaires parallélisés (combinaison de techniques de décomposition de domaines et méthodes itératives type Krylov avec préconditionnement).

Cette librairie est écrite en C et fortran77, fortement parallélisée (appels MPI dans les profondeurs du code).

Parmi les choix possibles :

- tout réécrire en X10 (irréaliste)
- développer une "couche de compatibilité" (i.e. coder une transfert X10 (travail conséquent et qui risque de dépendre de fonctionnalité X10 non stables)
- utiliser le runtime MPI (runtime X10 basé sur MPI) et définir une API "haut niveau" de pARMS (superficiel, il vaudrait mieux utiliser le runtime standard de X10 : PAGS)

Exemple de contrôle depuis X10 d'un code // MPI

On a défini les couches suivantes :

programme principal X10

classe X10 utilisant une API similaire et qui met les pointeurs internes dans des entiers long

Couche C++

Fonctions "haut niveau C" (construit et interprète les pointeurs boîtes noires)

Librairie pARMS (C/f77 + MPI)

Etat actuel de X10 vu d'un développeur de code

Modèle de programmation très souple

- // à mémoire distribuée, exécution multi-processus
- // à mémoire partagée, exécution multi-threads
- combinaison des deux
- prise en compte plus facile de la répartition de charge (si on peut découper le travail en de nombreuses tâches)

Il peut être difficile de s'adapter si on est habitué au modèle "passage de messages".

Etat actuel de X10 vu d'un développeur de code

Langage expérimental, outils pas assez mûr pour du code de production. Par contre, fonctionnalités et mode de conception très intéressant.

- Syntaxe différente suivant les versions (en cours de stabilisation), très (trop ?) riche (en cours de simplification),
- Outils (compilateur, librairie de communication, etc) incomplets,
- Ramasse-miette distribué apparemment non disponible actuellement : risque de poser des problèmes sur des calculs massifs et longs
- Conclusions sur les performances à prendre avec précaution

Les performances évoluent d'une version à l'autre.