# Modèles de programmation parallèle

**Frédéric Desprez**
**INRIA**
**LIP ENS Lyon**
**Equipe Avalon**

**Florence Zara**
**Université Lyon 1**
**LIRIS**
**Equipe SAARA**



## Quelques références

- **Cours Pthreads**, John Mellor-Crummey, Rice University
- **Cours OpenMP,** F. Roch (Grenoble)
- **The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming,** F. Le Mentec, T. Gautier, V. Danjean
- **X-Kaapi C programming interface**, F. Le Mentec, T. Gautier, V. Danjean
- http://www.openmp.org
- http://www.openmp.org/mp-documents/spec30.pdf
- http://www.idris.fr
- http://ci-tutor.ncsa.illinois.edu/login.php
- **Using OpenMP , Portable Shared Memory Model**, Barbara Chapman
- **Cours composants,** C. Perez (Lyon)

# Agenda

- KaaPI
- Pthreads
- OpenMP
- Composants parallèles
- MapReduce



# Introduction

**Modèle de programmation:** comment (d)écrire un programme parallèle

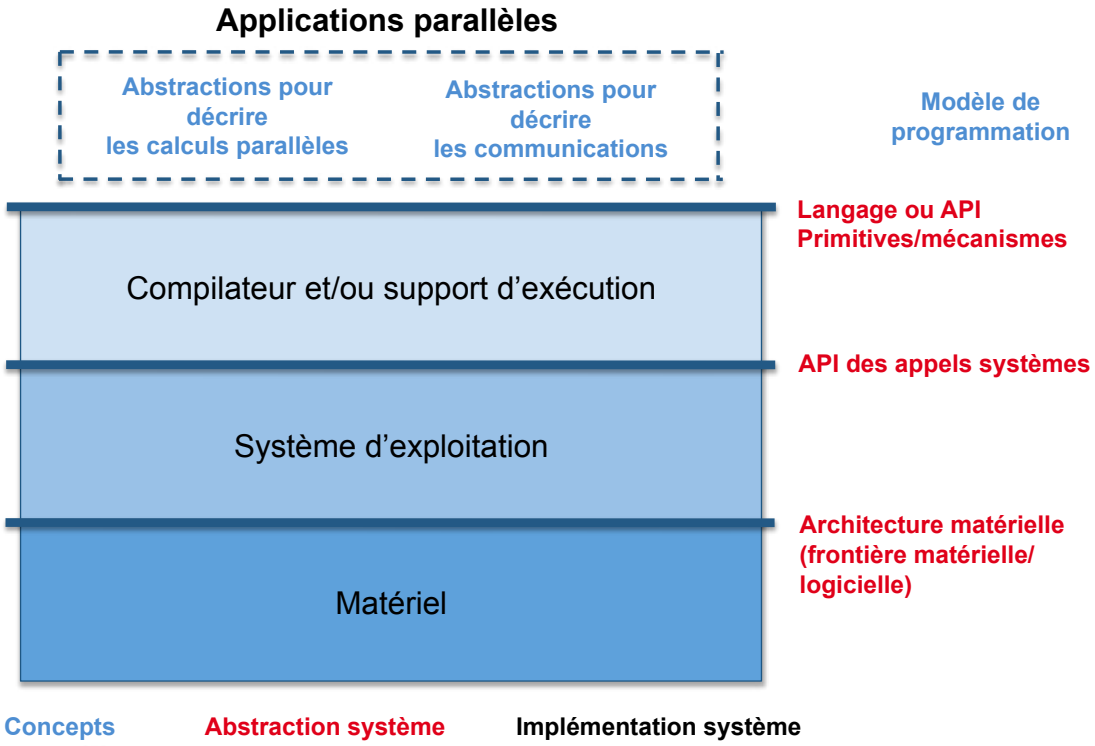
**On a vu MPI (Message Passing Interface)**

- L'utilisateur gère tout (distribution des données, distribution des calculs, synchronisation des processeurs, échanges des données)
- **Avantages**
  - Plus grand contrôle pour l'utilisateur
  - Performances (si code bien écrit !)
- **Inconvénients**
  - Assembleur du parallélisme
  - Portabilité (en termes de performances)
  - Moins de transparence

**Autre solution**

- Donner plus de travail aux environnements d'exécution !

# Introduction






# KAAPI

# KAAPI

**Projet INRIA-MOAIS**

http://kaapi.gforge.inria.fr

Librairie C / C++ (simplement ajout d'instructions dans le code)

**Portabilité**

- Pas besoin de changer le code selon l'architecture
- Exécution possible en CPU, multi-CPU, grappe, GPU
- Nombre de nœuds variable

**Langage de programmation parallèle de haut niveau**

- Abstraction de l'architecture (mémoire virtuelle partagée)
- Ordonnancement dynamique géré par l'environnement



# Concept

Problème initial est décomposé en **tâches de calcul**
Tâches s'exécutent en parallèle

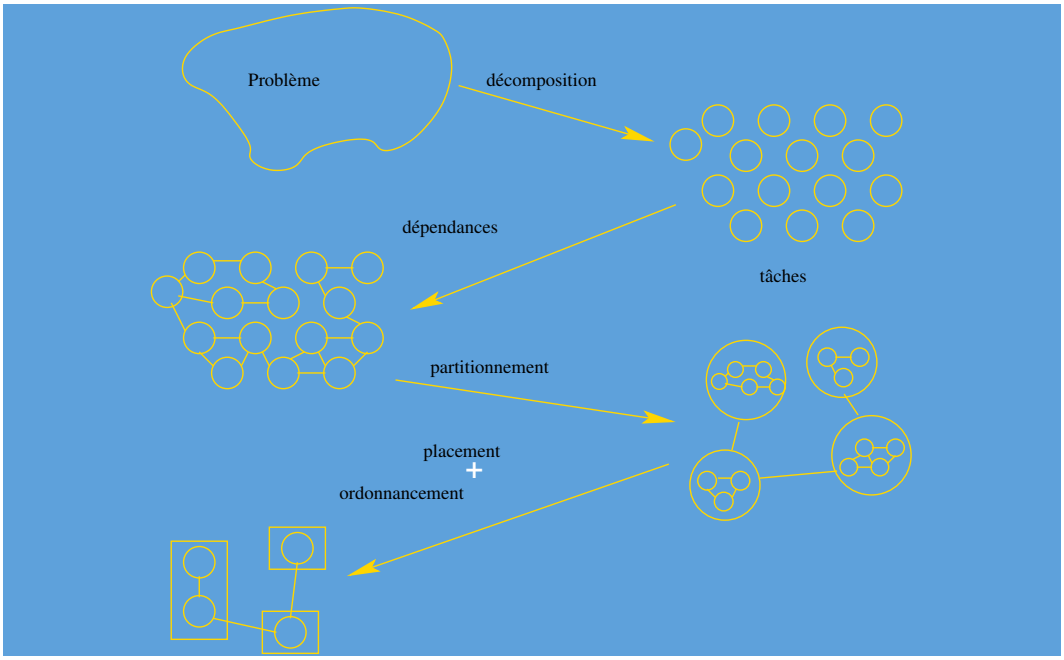
**Tâches travaillent sur des données (virtuellement) partagées**

- Différents accès possibles pour ces données
- Taille des données partagées définit la granularité

**Exécution des tâches de calculs en parallèle**

- Ordre d'exécution des tâches selon le **graphe de flots de données**
- Partitionnement du DFG (*Data Flow Graph*)
- Placement et ordonnancement sur les nœuds

# Illustration






# API C++ de Kaapi - Kaapi++

Concrètement dans le code

`#include «kaapi++»`

Utilisation du namespace **`ka::`**

```
// Initialisation de la librairie
int main(int argc, char** argv) {
 /* Join the initial group of computation */
  ka::Community com =
ka::System::join_communit(argc,argv);

 /* Start computation by spawning the main task */
  ka::SpawnMain<doit>()(argc, argv);

  /* Leave the community */
  com.leave();
  /* */
  ka::System::terminate();
}
```

# Définition des tâches – Hello World

Prototype de la tâche

- Définit le nombre de paramètres / type / et les modes d'accès pour chacun des paramètres

```
/* Kaapi Hello task: print an integer n */
struct TaskHello: public ka::Task<1>::Signature<int>{};
```

Déclaration de la tâche

- Spécification de l'architecture (CPU / GPU)

```
/* CPU implementation */
template<>
struct TaskBodyCPU<TaskHello> {
 void operator() (int n) {
   std::cout << "Hello World !, n=" << n << std::endl;
 }
};
```



# Exécution des tâches – Hello World

Appel de la fonction = création de la tâche (emploi du mot-clé **ka::spawn)**

```
/* The "doit" main task */
template<class T>
struct doit {
       void operator()(int argc, char** argv)
       {
          ka::Spawn<TaskHello>()(atoi(argv[1]));
       }
};
```

```
/* Ecriture de tous les arguments */
template<class T>
struct doit {
       void operator()(int argc, char** argv )
       {
              for (int i=1; i<argc; ++i)
                  ka::Spawn<TaskHello>()(atoi(argv[i]));
       }
};
```

**Ceci est un exemple, cliquez sur le lien de téléchargement pour obtenir le cours complet.**