Architectures multi-coeurs hétérogènes et logiciels

F. Bodin
Irisa/Caps entreprise

IRISA

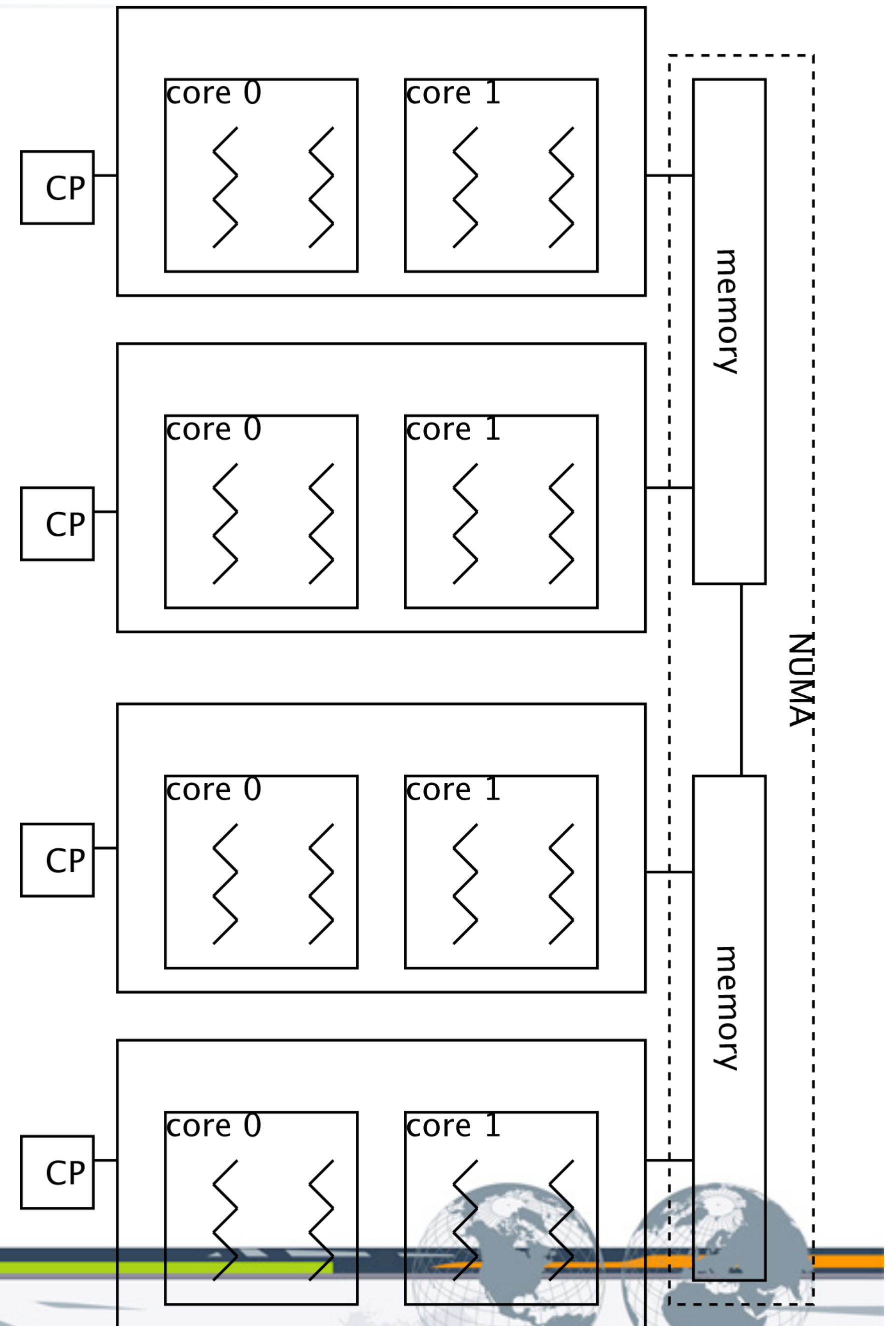
UNE UNITÉ DE RECHERCHE À LA POINTE DES SCIENCES
ET DES TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

Introduction

- **Généralisation des architectures multi-coeurs**
  → Mais construire des programmes parallèles est complexe
  → Les modèles de programmation et outils associés sont au centre de cette évolution
- **La durée de vie des applications est importante**
  → Utilisation sur de multiples générations de plateformes
- **La consommation/dissipation d'énergie porte cette évolution**
  → Tendance à long terme

Multi-coeurs (CMP)

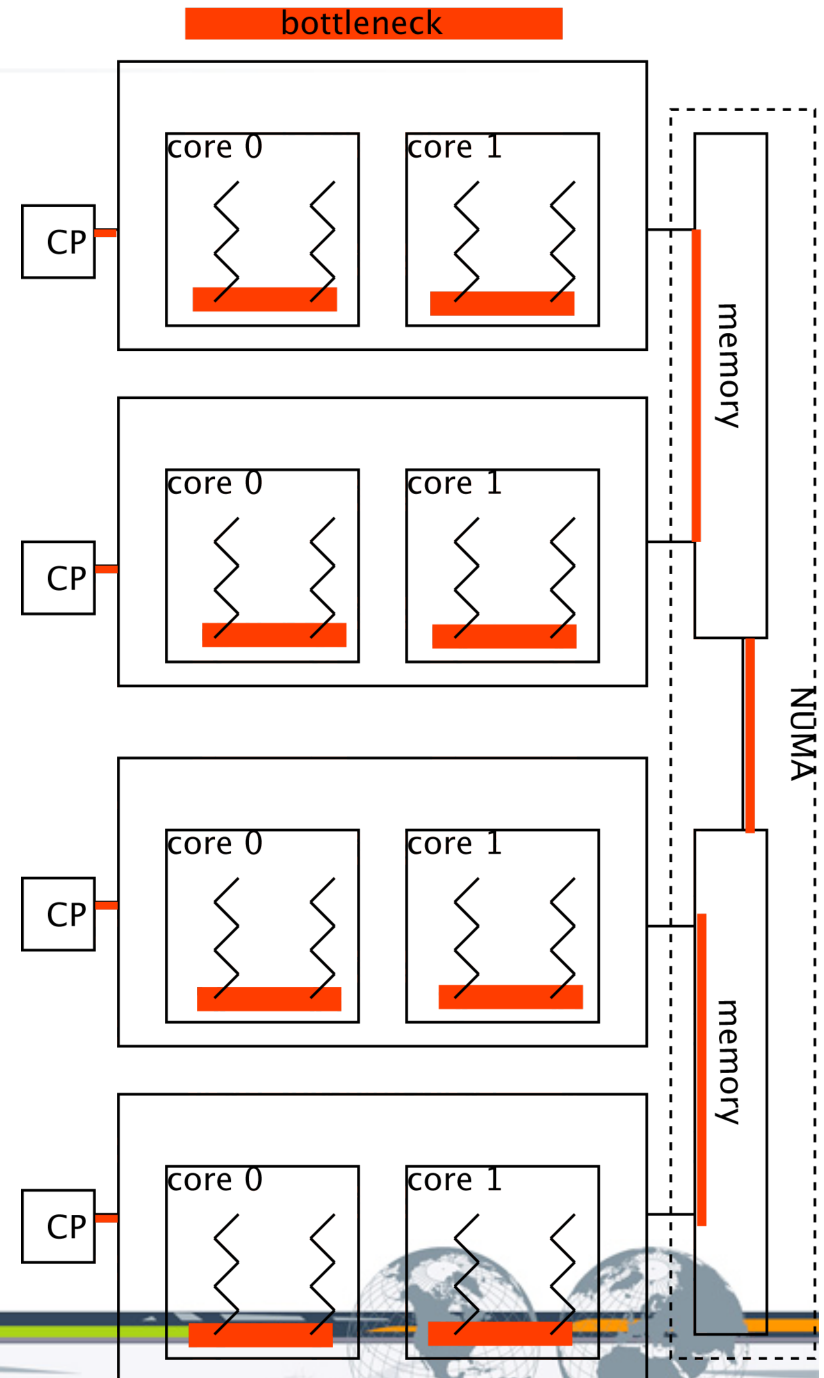
- **De nombreux niveaux de parallélisme**
  - → ILP
  - → SIMD
  - → SMT
  - → CPUs Multiple
- **Hiérarchie mémoire complexe**
  - → Mémoires caches partagées
  - → Mémoire partagées cohérentes
- **Unités de calcul homogènes ou hétérogènes**
  - → Cœurs de différents types
  - → Accélérateurs matériels
- **De nombreuses variantes architecturales**
  - → Pas de consensus



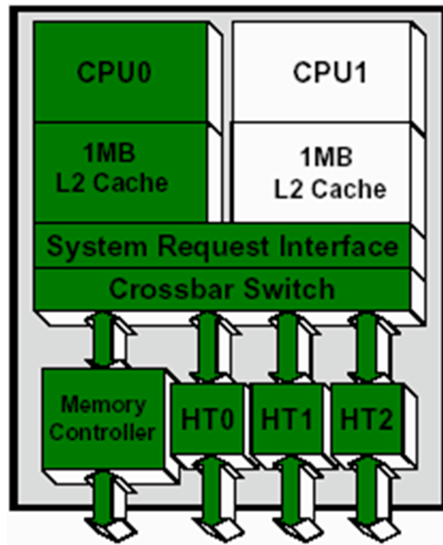
Multi-coeurs

- **Nombreux goulots d'étranglement**
  → Limitation de la bande passante mémoire
  → Latence de l'accès mémoire

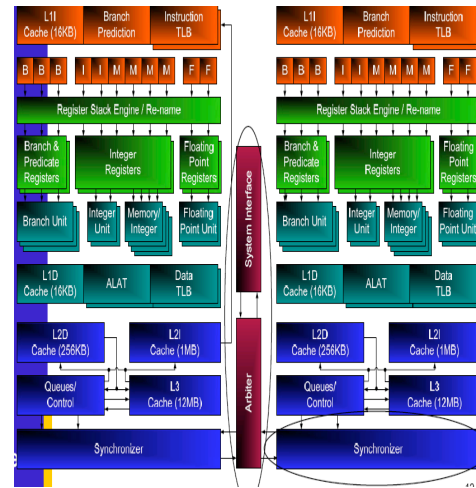
- **Instabilités des performances**
  → Nombreuses interactions entre composants
  → Faux partages



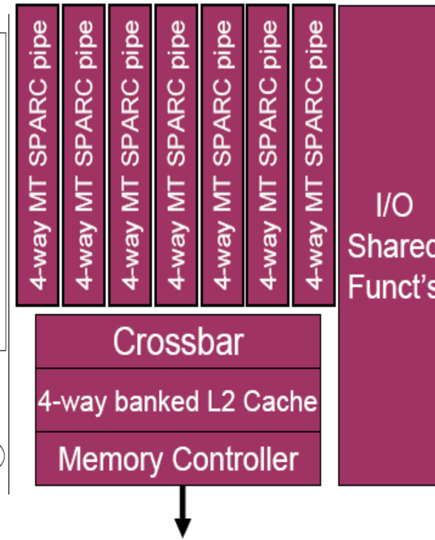
Exemples de multi-coeurs



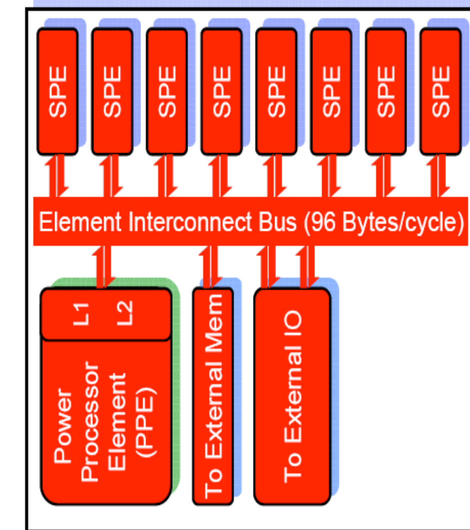
AMD dual-core



Intel Montecito



Sun T-1 (Niagara)



IBM Cell

- **AMD dual-core**
  → 2 coeurs, mémoire partagée, pas de cache commun
- **Intel Montecito**
  → Mémoire partagée, pas de cache partagé, SMT (2)
- **Sun T-1**
  → Cache partagé, 8 coeurs, MT (4)
- **IBM Cell**
  → Mémoire distribuée, 32 bits FP, 8 coeurs

Accélérateurs matériels

- **Plusieurs types**
  - →FPGA
    - Cray XD1, Mitrionics, Celoxica, …
  - →Opérateurs spécialisés
    - Tsubane, Clearspeed, Cell, …
  - → Processeurs graphiques
    - Nvidia, AMD/ATI

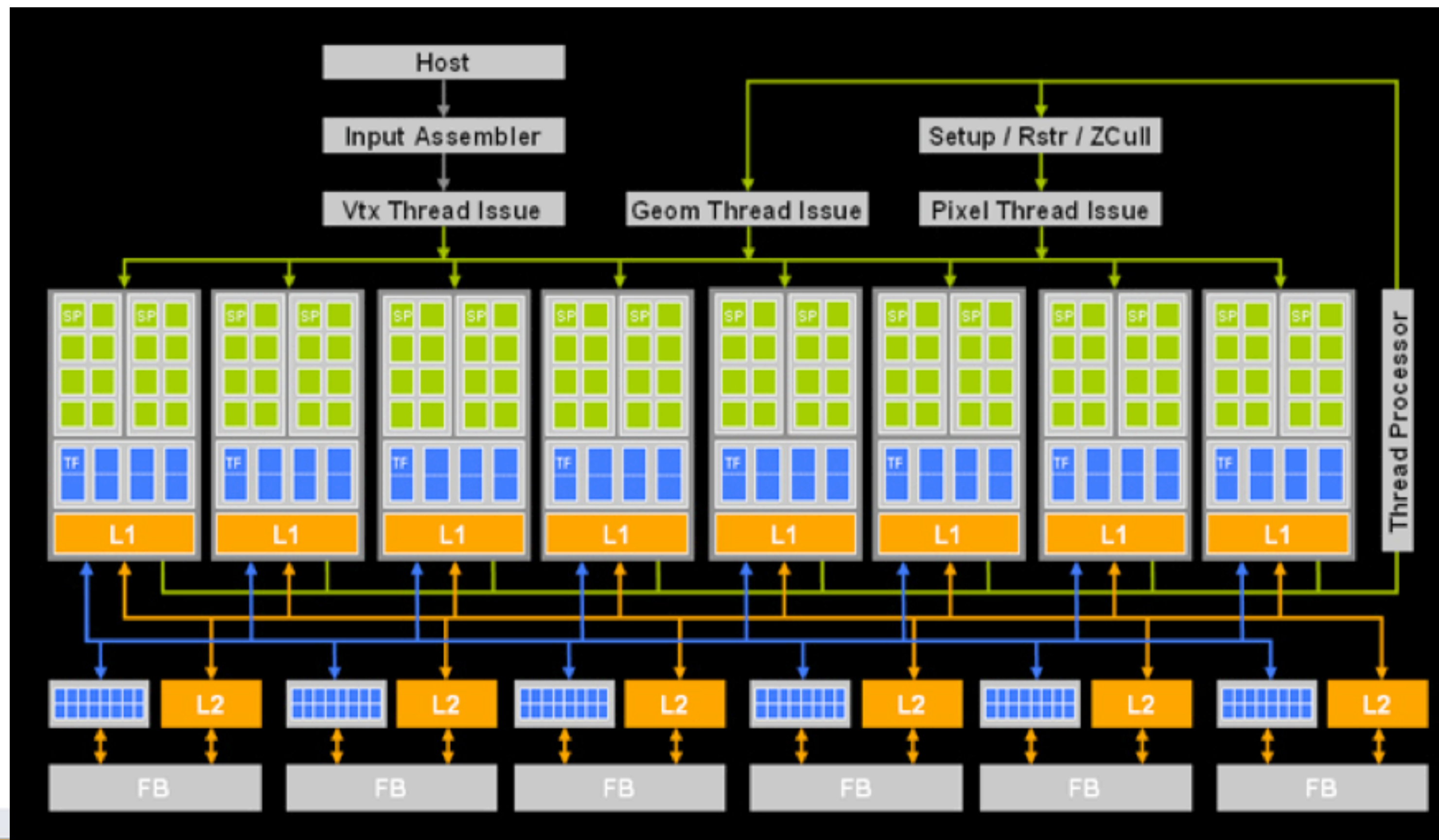
- **Mais tous contraints par les communications avec l'hôte (PCIe, …)**

Utilisation des processeurs graphiques comme accélérateurs pour le calcul numérique

- **Approche très prometteuse**
  → Pas encore d'implémentation complète de la norme des nombre flottants
    - 32 bit FP partiellement IEEE 754 (denorm, …)
  → Pas de ECC
- **Fondée sur l'exploitation du parallélisme**
  → Nombreuses restrictions de programmation
  → Jeu d'instructions SIMD
- **Communauté active**
  → http://www.gpgpu.org/

Exemple : NVIDIA GeForce 8800

- **128 unités de calcul, freq. 1.2 GHhz, recouvrement possible entre accès mémoire et calcul, 518 Gflops crête**



Stream Computing

- **Un noyau calcul est effectué sur un ensemble de données (collection / *stream*)**
  - → Pas de dépendances entre les calculs d'une même collection
  - → Parallélisme de données
  - → Adapté à certaines applications
  - → Exploite des mémoires à fort débit

Exemples d’environnements de programmation

- **Cuda**
  → Un ensemble d’outils
  - bibliothèque,
  - extension du langage C
  - expose une hiérarchie mémoire complexe

  → NVidia
- **Peakstream / RapidMind**
  → Une bibliothèque C++
  - Exécutif + JIT
  - Fondé sur un langage exprimant le parallélisme de données
- **Heterogeneous extension to OpenMP (HOMP)**
  → Intégration de l’usage d’un accélérateur dans une application

  → Préservation de la portabilité du code

Heterogenous extension to OpenMP

- **Ensemble de pragma pour spécifier l’usage d’un accélérateur**
  → Complémentaire à OpenMP
  → Utilisation *portable* des accélérateurs
  → Le code reste un code C standard
- **Gestion des ressources transparente**
  → La configuration d’un nœud est prise en compte par l’exécutif
- **http://www.caps-entreprise.com**

Exemple HOMP

```
#pragma homp csmain codelet, param n1 = 16384, &
#pragma homp param n2 = 16384, param n3 = 16384
static void codelet(float alpha, float beta, float X[N], int n1,
                    float Yin[N], int n2, float Yout[N], int n3){
  int i;
  for( i = 0 ; i < n1 ; i++ ) {
    Yout[i] = alpha*X[i] + beta*Yin[i];
  }
}

int main(int argc,char **argv) {
  int i;
  float alpha;
  float beta;
  float X[N];
  float Y[N];
  int n = N;
  ...
#pragma homp csmain callsite
    codelet(alpha,beta,X,n,Y,n,Y,n);
  ...
  return 0;
}
```

Qu'est ce qui a changé avec les multi-coeurs ?

- **Les futurs gains de performance sont liés à la multiplication des coeurs**
  → Pas de la fréquence des processeurs
- **La hiérarchie du parallélisme est inévitable**
  → Comment la traduire au niveau applicatif ?
  → La distance entre le code et l'architecture s'accroît
- **Homogènes / hétérogènes**
  → L'efficacité énergétique encourage l'hétérogène
- ***Puissance CPU bon marché***
  → Une partie peut être consacrée au support de l'exécutif (i.e. helper threads)

Pourquoi le développement des applications est si difficile ?

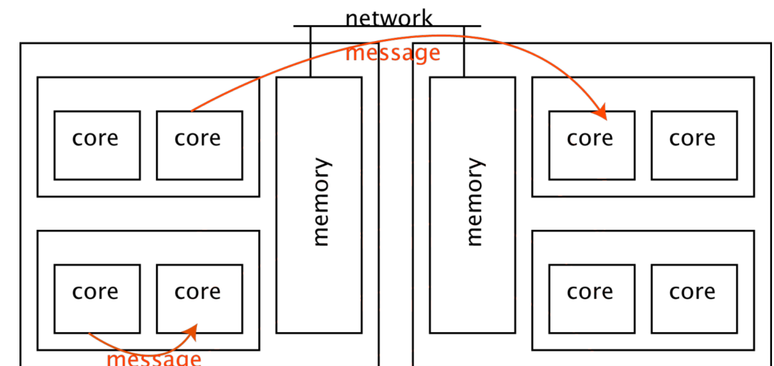
- **L'objectif est la réduction des temps d'exécution**
- **Correction**
  → Processus de vérification des programmes // complexe
- **Prédiction de performance impossible en général**
  → Pas de modèle des plateformes
- **De nombreux paramètres interdépendants**
  → Ressources partagées entre processus
  → Localité des données
  → Loi d'Amdahl
- **A quel niveau mettre l'accent**
  → Parallélisme d'instruction
  → Parallélisme de thread/entre coeurs
  → Utilisation d'accélérateurs matériels
- **Exécutif**
  → Allocation des threads et des données
- **Configurations multiples**
  → Comment produire des codes robustes

Approches disponibles

- **Au niveau cluster**
  - → Bibliothèques
    - Echange de messages, tableaux globaux, …
  - → Langages avec espace global d'adresses partitionné/uniforme (+ modèle SPMD)
    - Co-Array Fortran, UPC, Global Arrays, …
  - → OpenMP + *Distributed Shared Memory* (logicielle)
- **Au niveau mémoire partagée**
  - → OpenMP, Pthreads, …
  - → Parallélisation automatique
- **Clusters de nœuds parallèles à mémoire partagée**
  - → Modèle hybride
- **Approche pragmatique actuelle**
  - →MPI + OpenMP (et/ou parallélisation automatique)

Message Passing Interface (MPI)

- **Echange de messages**
  → Impose des synchronisations entre processus
- **Un standard de facto**
  → Codes portables, large disponibilité
  → Bon contrôle des aspects performances
    - Affinité entre les *threads* et les données
  → Séparation des problèmes
    - Optimisations séquentielles des programmes
    - Organisation du parallélisme



Message Passing Interface (MPI)

- **Les faiblesses de MPI ?**
  → Programmation et déboggage complexes
    - *Plus petit dénominateur commun*

  → Adapté aux applications régulières
    - Code *bulk-synchronous* avec une charge équilibrée

  → Efficacité parfois limitée sur une architecture à mémoire partagée
    - Surcoût des copies

OpenMP

- **Parallélisme de *threads* (tâches)**
  → Pour plateforme parallèle à mémoire partagée, C/C++ et Fortran
  → Utilisation de directives/pragmas

```
C$OMP PARALLEL DO PRIVATE(B)
C$OMP& SHARED(RES)
  DO I=1,NITERS
      B = do some work (I)
C$OMP CRITICAL
      Res += Only one at a time
C$OMP END CRITICAL
  ENDDO
```

- **Plus simple que MPI**
  → Mais rarement efficace sur les architectures à mémoire distribuée

OpenMP

- **Quels problèmes avec OpenMP?**

→OpenMP ne passe pas à l'échelle

- Parallélisme modéré
- Mémoire partagée et non hiérarchique

→ Contrôle de la localité insuffisant

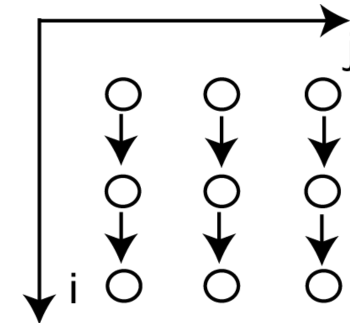
- Comment diriger l'ordonnancement des *threads* ?

→ Portabilité des performances ?

Parallélisation automatique

- **Technique de compilation pour détecter des itérations de boucle indépendantes**
  - → Fondées sur la notion de dépendance de données
  - → Suppose une mémoire partagée
- **Disponible dans de nombreux compilateurs**
  - → Mais la programmation doit être *propre*

```
DO i = 2,n-1
  DO j = 2, n-1
       A(j,i) = 0.5*(A(j,i+1)
               + A(j,i-1))
  ENDDO
ENDDO
```




```
(j=2,i=2) A(j+0,i) = 0.5*(A(j+0,i+1) + A(j+0,i-1))
(j=3,i=2) A(j+1,i) = 0.5*(A(j+1,i+1) + A(j+1,i-1))
(j=4,i=2) A(j+2,i) = 0.5*(A(j+2,i+1) + A(j+2,i-1))
(j=2,i=3) A(j+0,i+1) = 0.5*(A(j+0,i+2) + A(j+0,i))
(j=3,i=3) A(j+1,i+1) = 0.5*(A(j+1,i+2) + A(j+1,i))
(j=4,i=3) A(j+2,i+1) = 0.5*(A(j+2,i+2) + A(j+2,i))
(j=2,i=4) A(j+0,i+2) = 0.5*(A(j+0,i+3) + A(j+0,i+1))
(j=3,i=4) A(j+1,i+2) = 0.5*(A(j+1,i+3) + A(j+1,i+1))
(j=4,i=4) A(j+2,i+2) = 0.5*(A(j+2,i+3) + A(j+2,i+1))
```

Parallélisation automatique

- **Avantages**
  - → Peu d'efforts de programmation
- **Désavantages**
  - → Efficacité très variable
    - Limite des analyses statiques
  - → Restreint aux mémoires partagées, parallélisme limité

```
void func(int n,int *nx,
          int *xint, int *yint,
          int *ny, int *xh,
          int *yh, int src,
          int lx2, int y,
          int x, int *s){
  for (int k = 0; k < n; k++){
    xint[k] = nx[k]>>1;
    xh[k] = nx[k] & 1;
    yint[k] = ny[k]>>1;
    yh[k] = ny[k] & 1;
    s[k] = src+lx2*(y+yint[k])
           +x+xint[k];
  }
}
```

Directions de recherche

- **Compilation dynamique ou itérative**
  - → Intégration des paramètres d'exécution
  - → Adaptation à l'exécution
- **Bibliothèques/composants adaptatives / auto-tuning**
  - → ATLAS, …
- **Solutions spécialisées**
  - → Mémoires transactionnelles
  - → Matlab, …
- **Nouveaux langages de programmation**
  - → Fortress, X10, …

Conclusion

- **Le langage de programmation n'est qu'un aspect**
  → Approches orientées domaine permettent de mieux prendre en compte l'ensemble des problèmes
  → Besoin de bibliothèques/composants parallèles de bonne qualité
  → Gestion des ressources
- **MPI reste une approche « sure »**
  → Pas de remise en cause dans un futur proche
  → MPI + OpenMp?
- **Besoins de nouvelles approches globales**
  → Qui considèrent les aspects hétérogènes (HOMP, …)
  → Et la croissance exponentielle du nombre de coeurs