Exploitation pratique et efficace du parallélisme sur processeurs multi-coeurs

Pierre Palatin

Soutenance de thèse
Université Paris XI Orsay
5 Septembre 2008

Directeur : Olivier Temam

Introduction

- Multi-cœurs pour le grand public
  - System on Chip déjà présents, mais spécialisés
  - Programmes irréguliers et variés
  - Pas de parallélisation automatique générique
- Comment exploiter efficacement ces processeurs ?
- Deux problèmes :
  - Trouver le parallélisme
  - **L'exploiter efficacement**
- Efficacité : vitesse, consommation, latence, ...

Vers Capsule

Comment passer du parallélisme à une exécution efficace ?

- Redistribuer la complexité
  - Le programmeur connaît la structure du programme et le parallélisme potentiel
  - Le système sait comment le programme se comporte en pratique
- Modifications à plusieurs niveaux :
  - Code / langage
  - Chaîne de compilation
  - Runtime system / processeur

→ Capsule

Plan

- Principes généraux
  - *La division conditionnelle*
- Version Matérielle
  - *Self Organized Multi Threaded architecture*
  - *Performance*
- Version logicielle
  - *Principes*
  - *Implémentation*
  - *Performance*
  - *Limitations*
- Le modèle Mémoire
- Contributions et évolutions possibles

Capsule – Principes

Division conditionnelle

Principe central de Capsule

- Le programmeur décrit le parallélisme *potentiel*
- Le système adapte le parallélisme à l'exécution

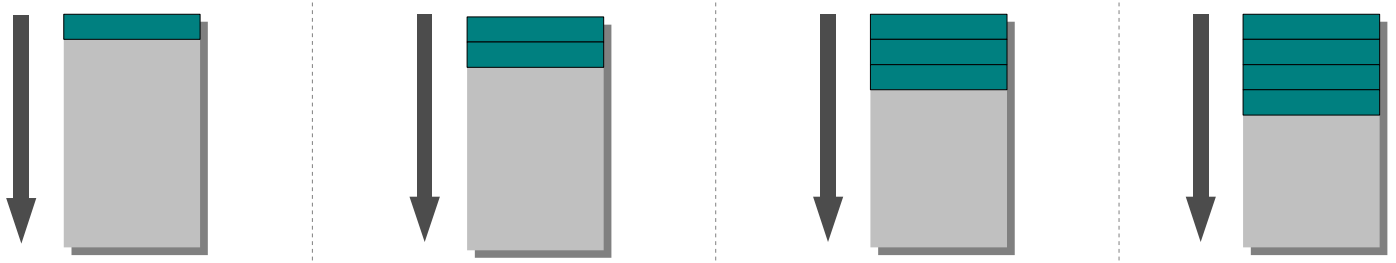
- En pratique :
  - Des créations de threads sont demandées partout où il peut y avoir du parallélisme
    - Le programmeur connaît (relativement) facilement la concurrence possible
  - Les créations sont accordées si cela est intéressant pour la performance
    - Le système connaît l'état du système en permanence
  - En cas de refus, le programme s'exécute séquentiellement
    - Jusqu'à la tentative suivante

Exemple

Code

```
Pour i de 0 à n:
  c[i] = a[i] + b[i]
```

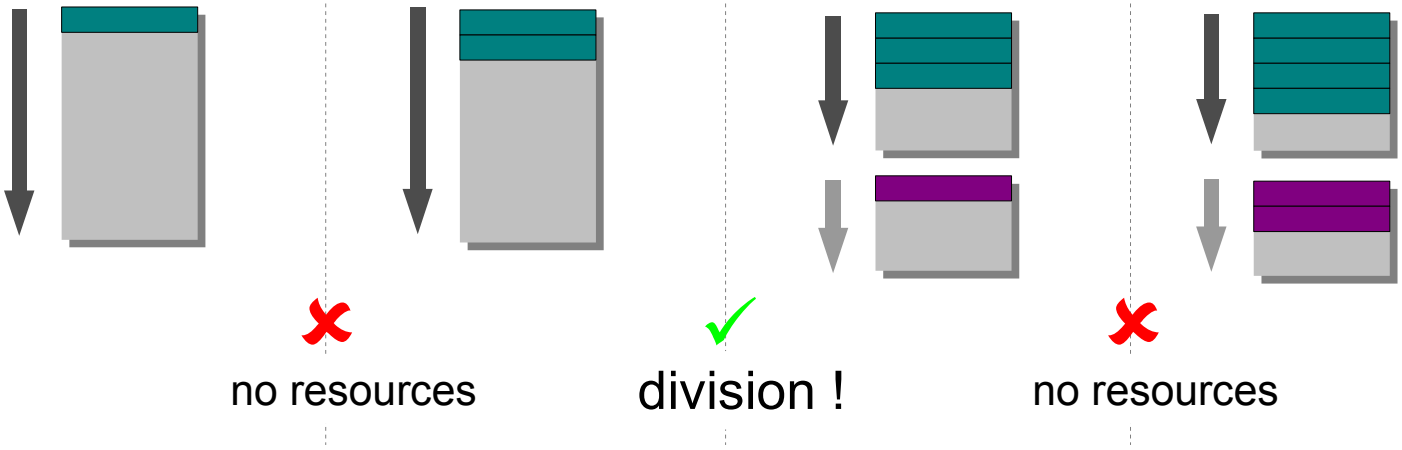
Exécution



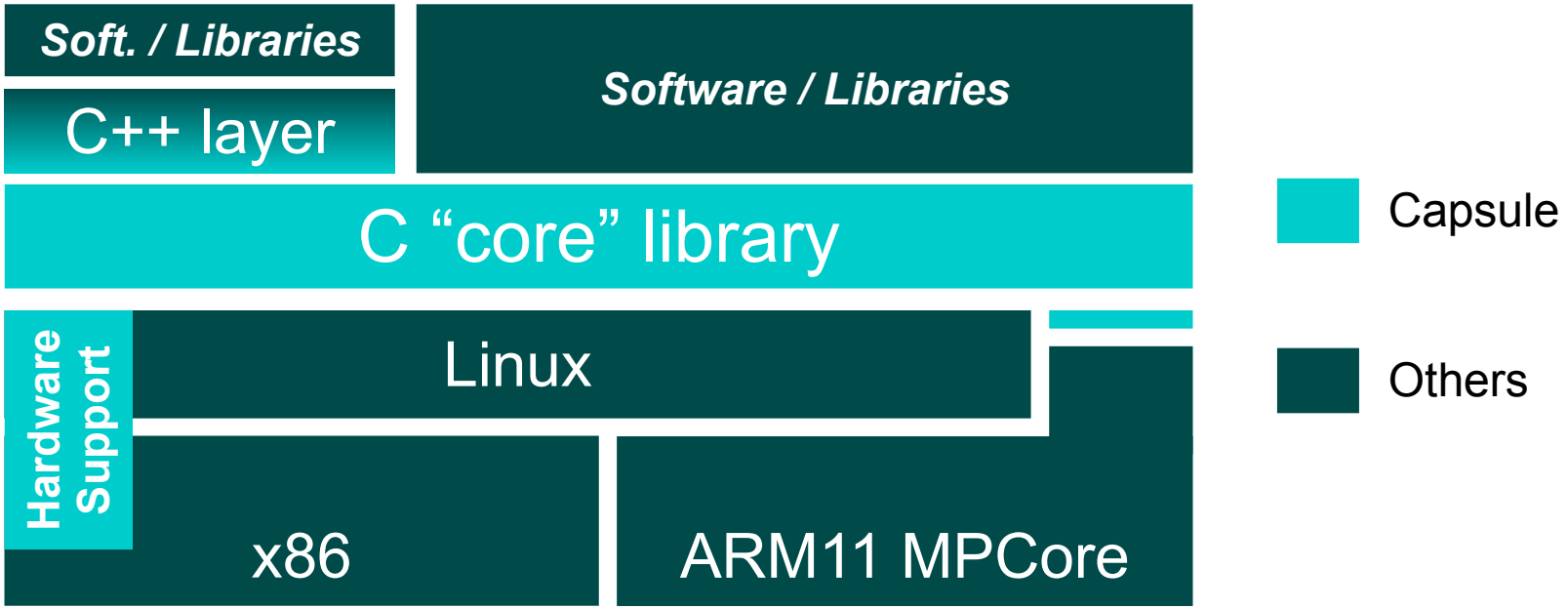
Boucle séquentielle

Boucle Capsule

```
Fonction loop(min, max):
  Pour i de min à max:
    c[i] = a[i] + b[i]
    Si probe(loop):
      divide((i+max)/2, max)
      max = (i+max)/2
```



Architecture de Capsule



- Support matériel
- Version C
  - abstraction générique de l'architecture
  - Implémentation logicielle
- Portage simple sur Arm11 sans O.S.
- Couche C++ : modèle mémoire

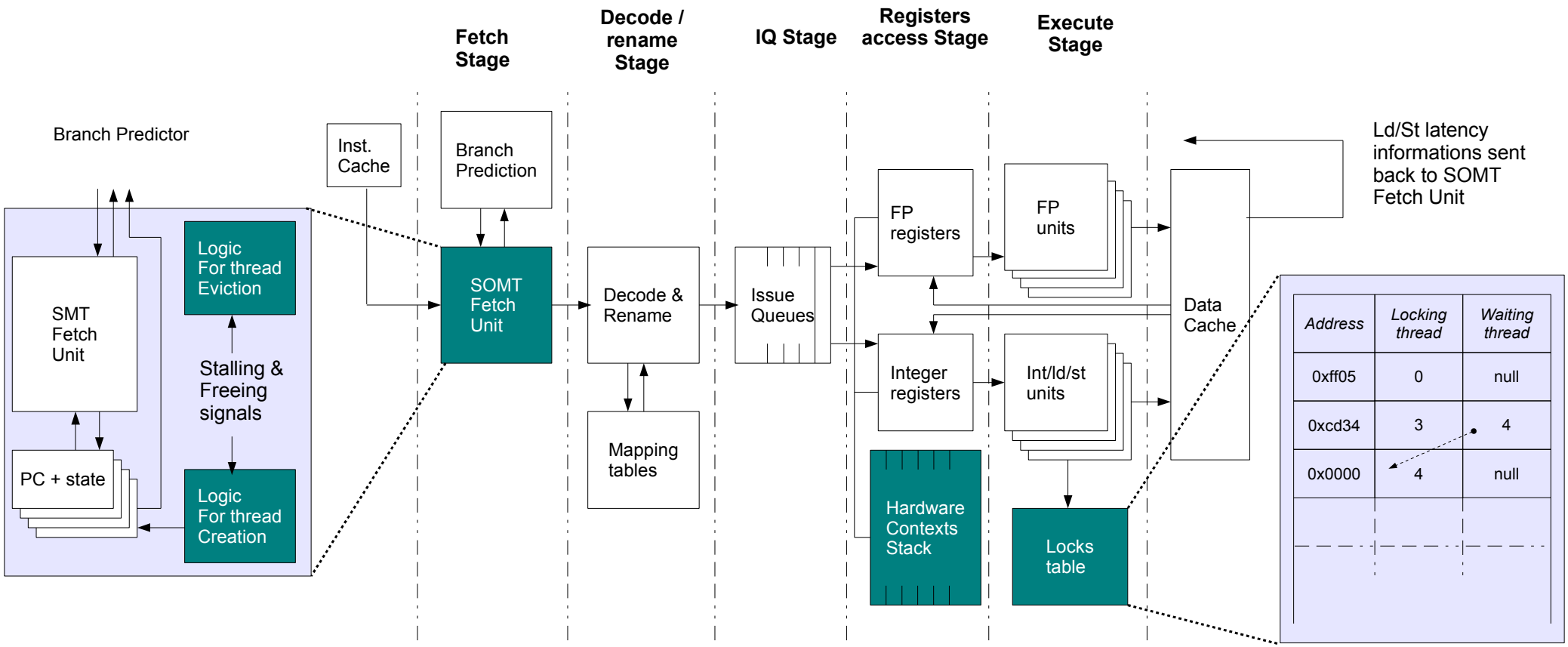
Comparaison

- OpenMP (GCC/GOMP: Mai 2007)
  - Dynamisme, adaptation à l'architecture
  - Essentiellement, gestion de boucles parallèles
- Cilk (fonctions asynchrones)
  - Découpage en tâches à la charge du programmeur
  - Pas de version séquentielle possible du code, limitation de la dynamicité
- Intel TBB (Version 1: mi-2006, version 2: mi-2007)
  - Collection d'algorithmes et de structures classiques; approche ad-hoc
  - Notion de tâche et de 'grainsize'
  - Bas niveau, ne cherche pas à cacher les propriétés sous-jacentes
- Cuda (Février 2007)
  - Suppose la présence de parallélisme matériel massif
  - Cible du code au parallélisme homogène
- Capsule: adaptation du code plutôt que scheduling

Version matérielle

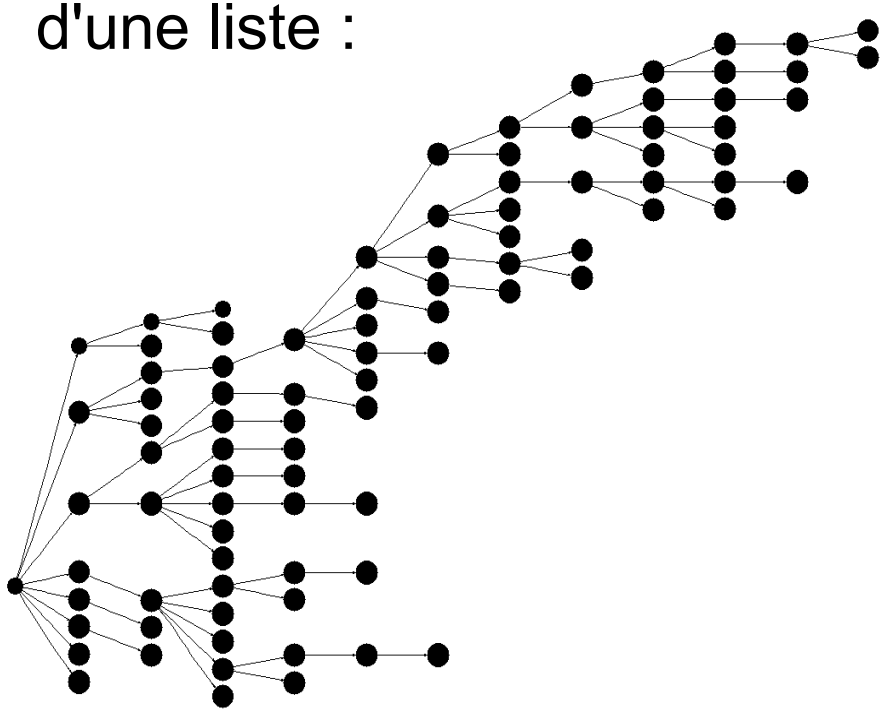
Architecture

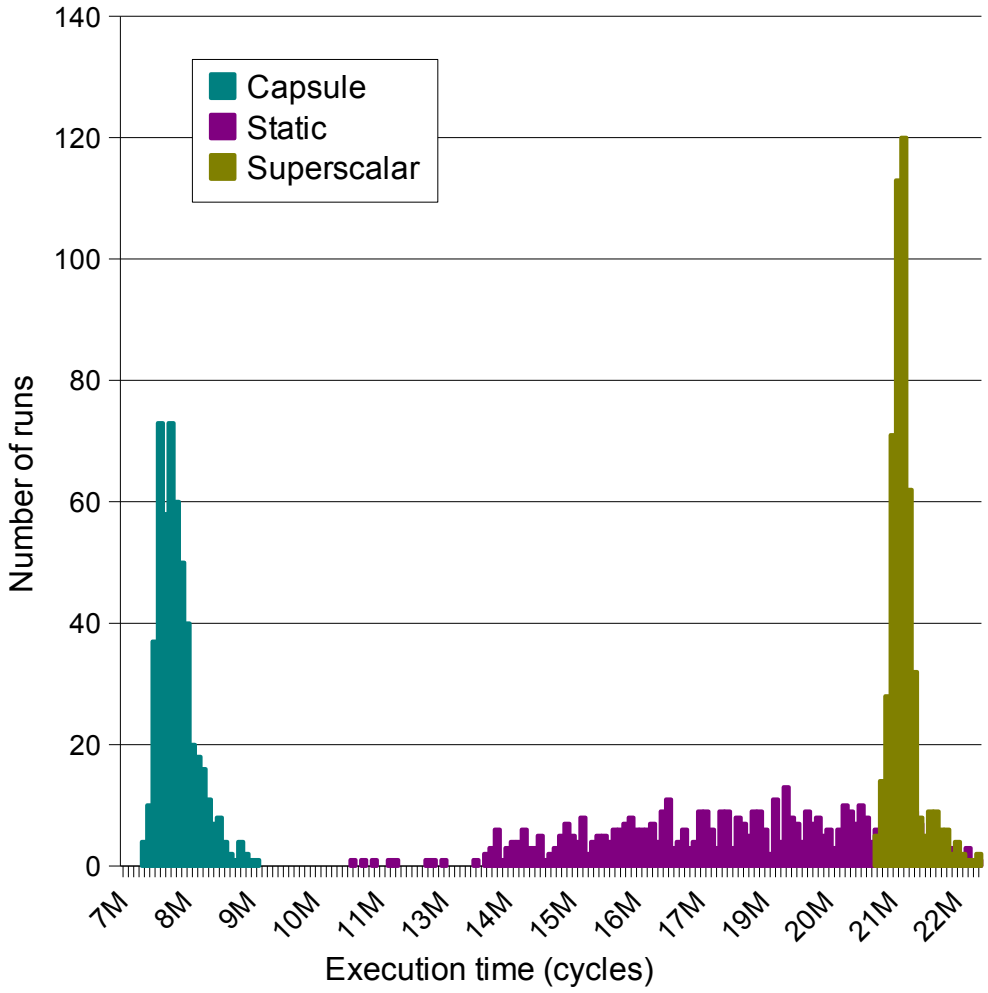
- Modification d'un SMT; 8 threads
- Création dynamique de thread : *nthr*
- « swap » des threads inefficaces



Stabilité : exemple de Quicksort

- Tentative de division pour chaque sous liste
- Comparaison de 3 versions :
  - Séquentielle / super-scalaire
  - Statique
  - Capsule
- Exemple de divisions sur le tri d'une liste :





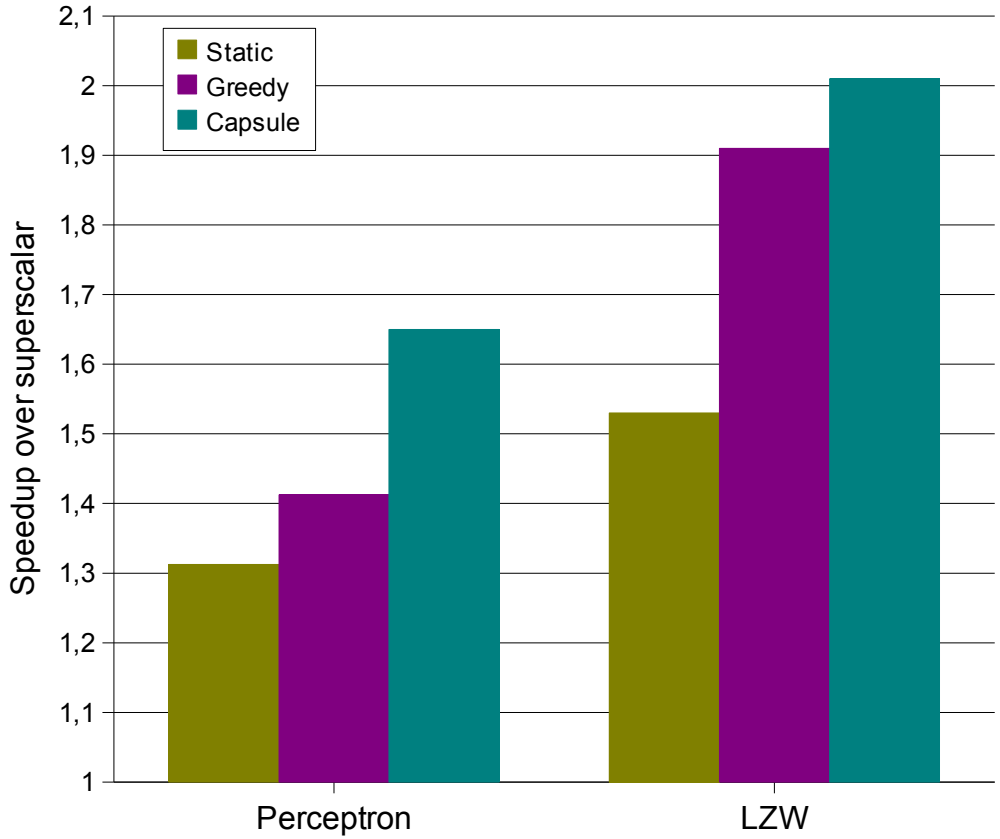
QuickSort, sur 500 listes de même taille

Limitation de la division

- Principe de base :

  thread disponible → division ok

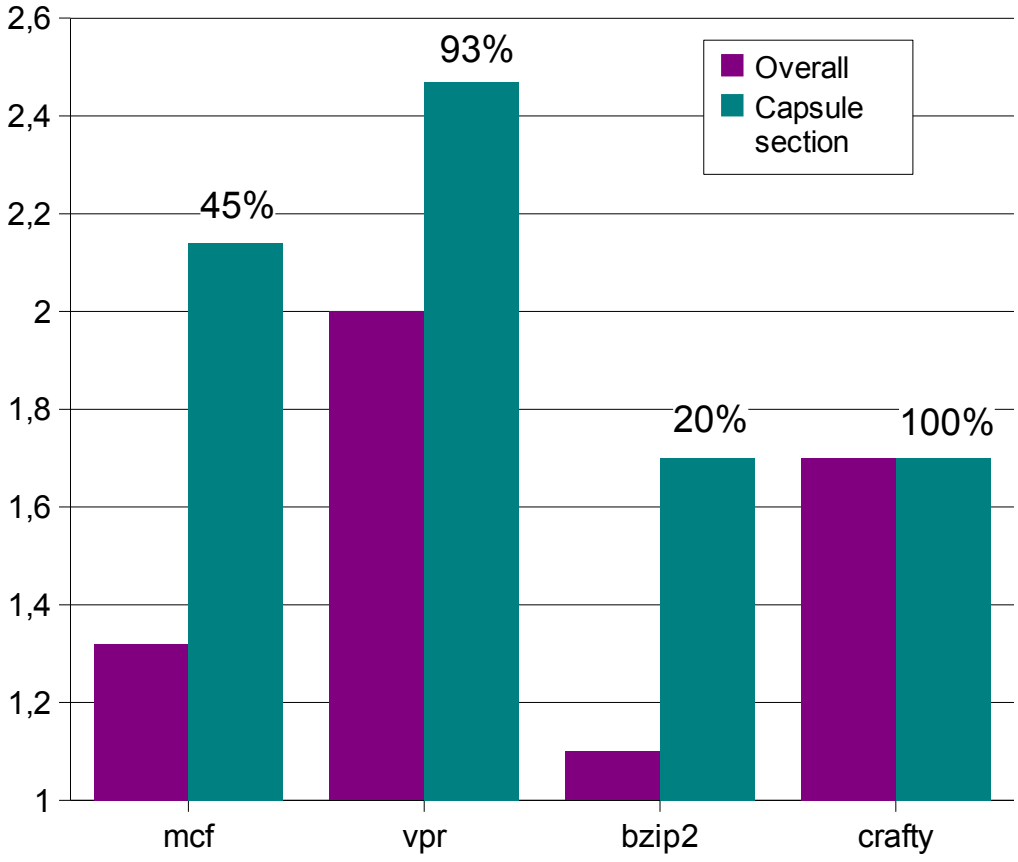
- Division coûteuse en fin d'algorithme
  - Faible quantité de travail
  - Coût division constant
- Comparaison :
  - Greedy : pas de limitation
  - Capsule : divisions bloquées en cas de morts rapides



Exemple sur les SPEC-CINT 2000

- 4 SPEC CINT 2000 modifiés
- Nécessite d'analyser tout le code pour l'adaptation
- Mais peu de modifications au final
- Crafty : busy-waiting sur version déjà parallélisée
- Statistiques de divisions :

Benchmark	# divisions requested	# divisions granted	% divisions granted	#insts per division granted
mcf	99598	40532	40%	3.7K
vpr	67560	2702	4%	4.5M
bzip2	38656	2319	6%	30M



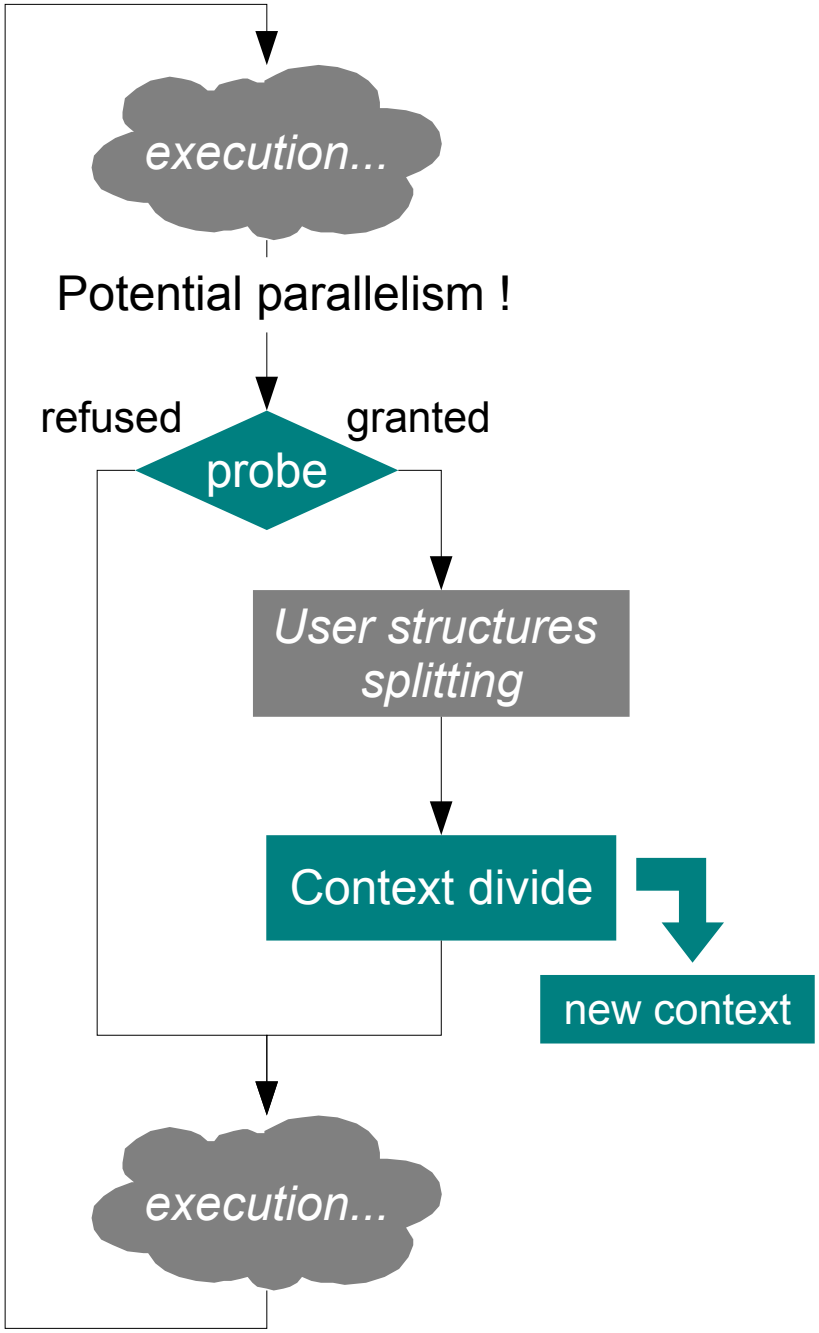
Speedup par rapport à la version originale séquentielle

- Peu de divisions acceptées
  - Les ressources sont bien utilisées

Version logicielle

Principe général

- Origine
  - Objectif initial : faciliter le débuggage d'application Capsule
  - Plusieurs versions successives
  - Mais performance possible !
- Basé sur pthread
- « contexte »
  - Exécuté sur un thread
  - Durée de vie limitée
- 2 fonctions principales
  - ***« probe »*** : Demande au système la création d'un contexte
  - ***« divide »*** : Lance l'exécution du contexte

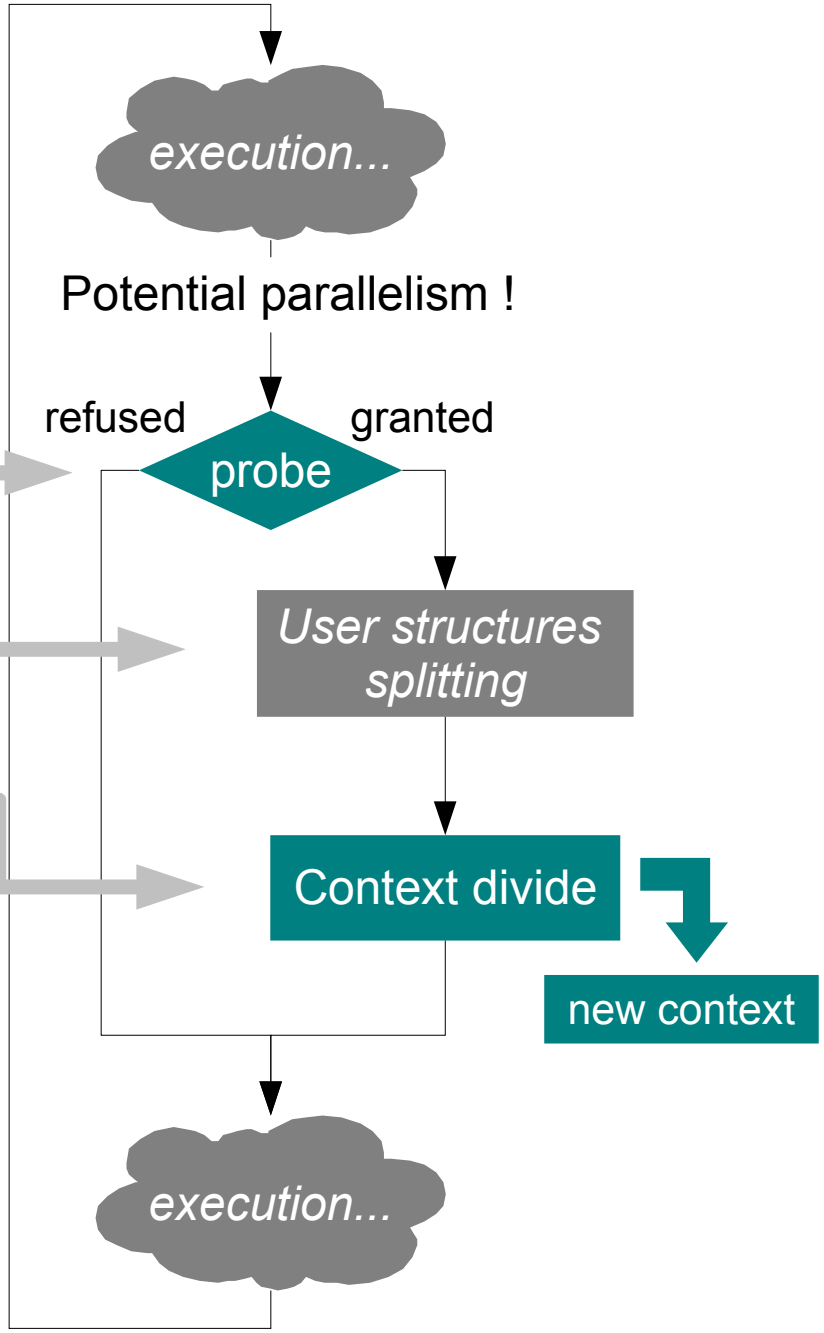


Addition de vecteurs simple

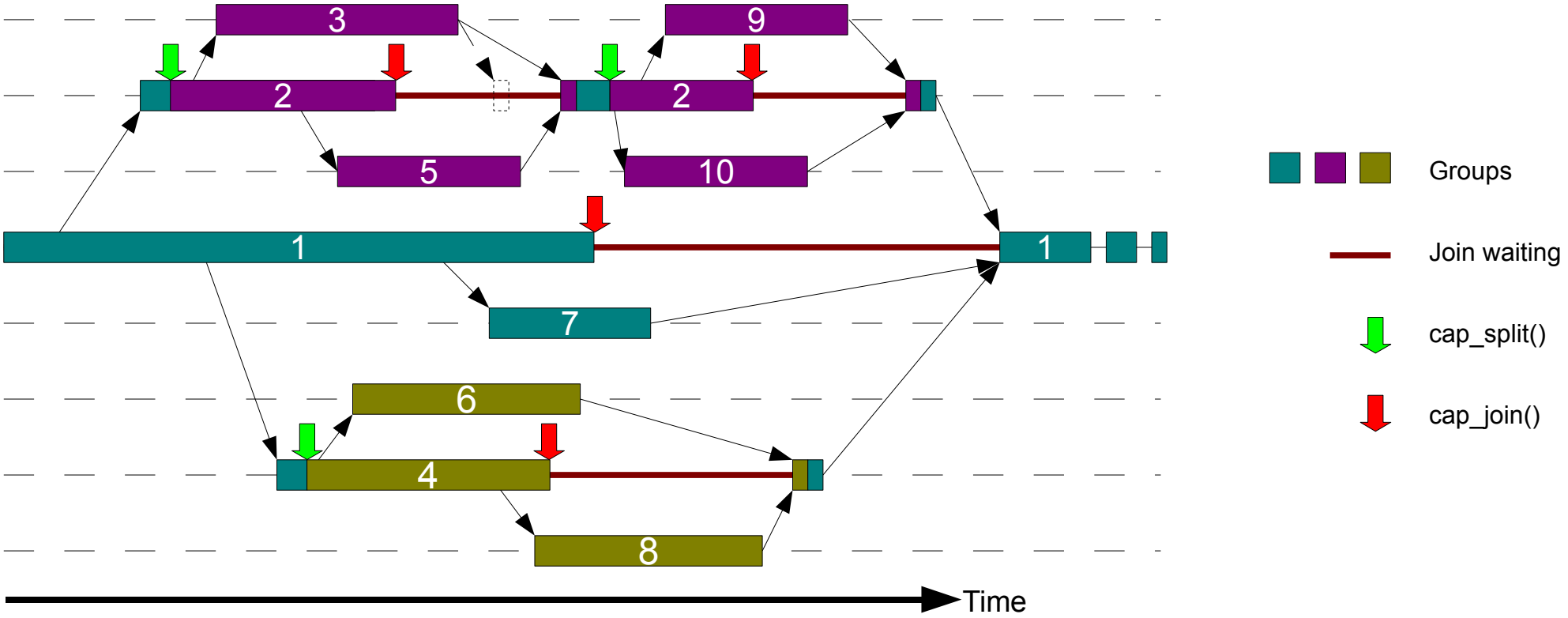
```
void loop(void *arg) {
    context_t *ctx;
    looparg_t* lt = (looparg_t*) arg;
    int i, newmin, newmax;

    for (i = lt->min; i<lt->max; lt ++) {
        lt->c[i] = lt->a[i] + lt->b[i];

        ctx = capsys_probe(loop);
        if (ctx) {
            newmax = lt->max;
            newmin = (i+lt->max) / 2;
            max = newmin - 1;
            capsys_divide(alloc_looparg(
                        lt,newmin,newmax));
        }
    }
}
```



Synchronisation: Groupes Capsule

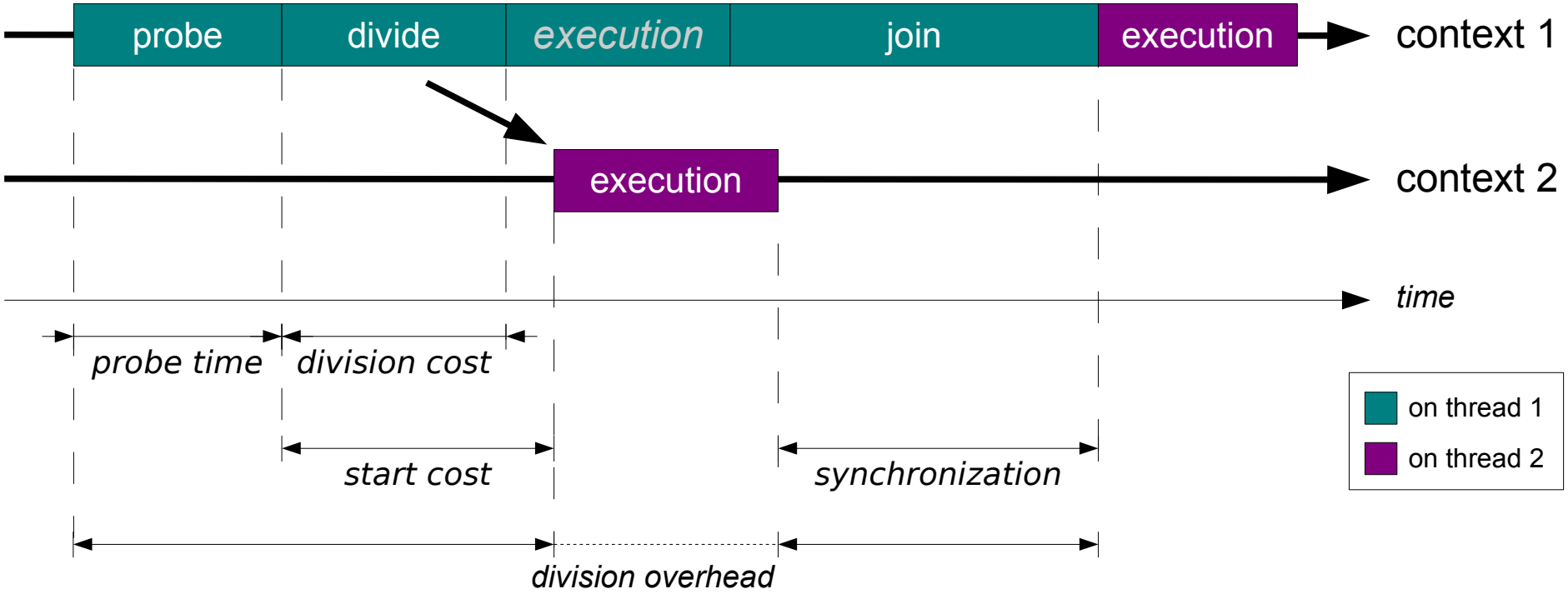


- Synchronisation nécessaire
- Réutilisables, encapsulables
- 2 opérations simples:
  - « cap_split() » : création
  - « cap_join() » : destruction
- Pas de groupes arbitraires

- Règles des groupes
  - Un contexte appartient à un seul groupe
  - Groupes hiérarchiques
  - Nouveaux contextes crées dans le groupe du contexte créateur

Version logicielle : Performance

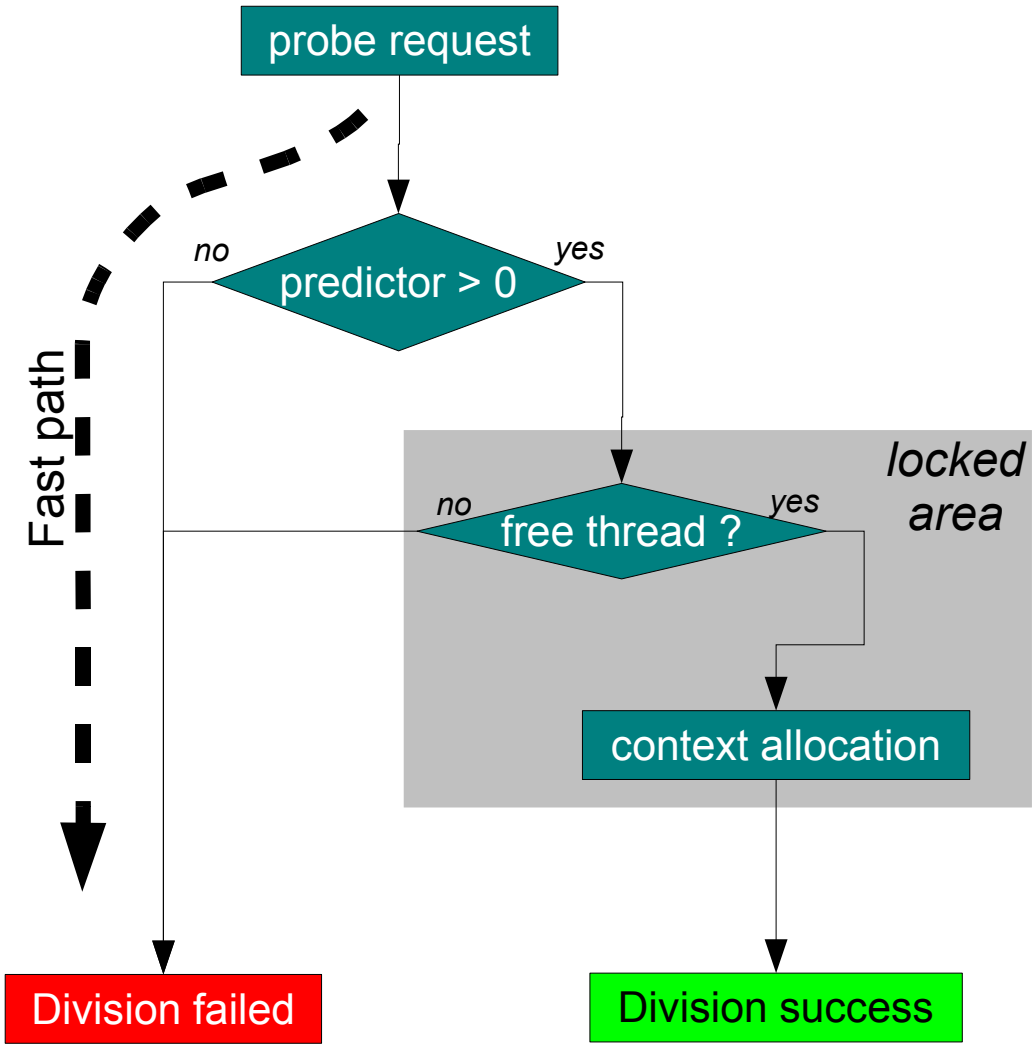
Scénario d'une division réussie



- Ordre de grandeurs, en cycles :
  - Probe time: 1150 (valeur pour cas division réussie)
  - Division cost: 10
  - Start cost: 480
  - Synchronization: 3450
  - Division overhead: 5080
- Valeurs très variables selon les architectures

Prédicteur logiciel

- Lock nécessaire pour réservation thread
- Lock coûteux; incompatible avec des probes très fréquents
- Taux de divisions réussies faible
  - Ex. QuickSort < 3%
- Prédicteur logiciel
  - Estime s'il est intéressant de prendre le lock
  - Correct la plupart du temps
  - Mis à jour lors de la création réussie et de la mort de contextes
  - Introduction d'une race condition, non critique



Résultats du prédicteur logiciel

- Majorité des cas : vérification de variable globale
  - Code assembleur généré :

```
movl    capsule_hint(%rip), %eax
testl   %eax, %eax
jle .L7
...   ;division code
.L7:
...   ;normal continuation of code block
```

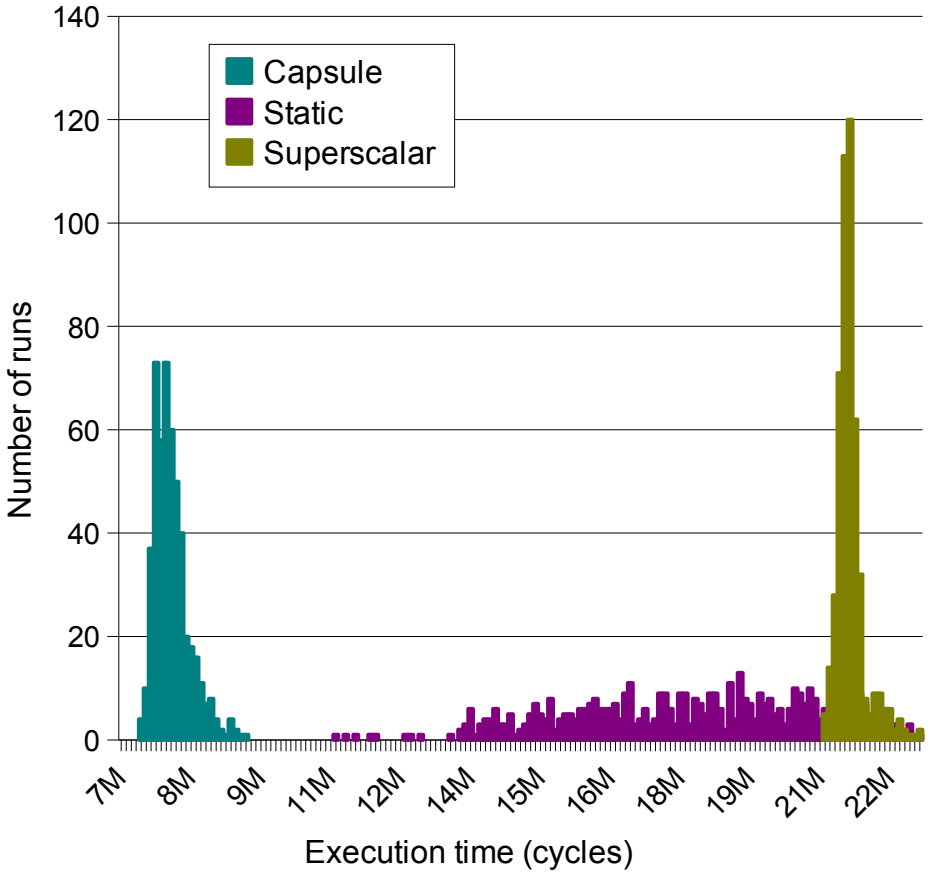
- Speedup en séquentiel :

	Probe échoué	QuickSort
Sans prédicteur	121 cycles	194 M cycles
Avec prédicteur	~ 1-2 cycles	141 M cycles
Speedup	121	1,37

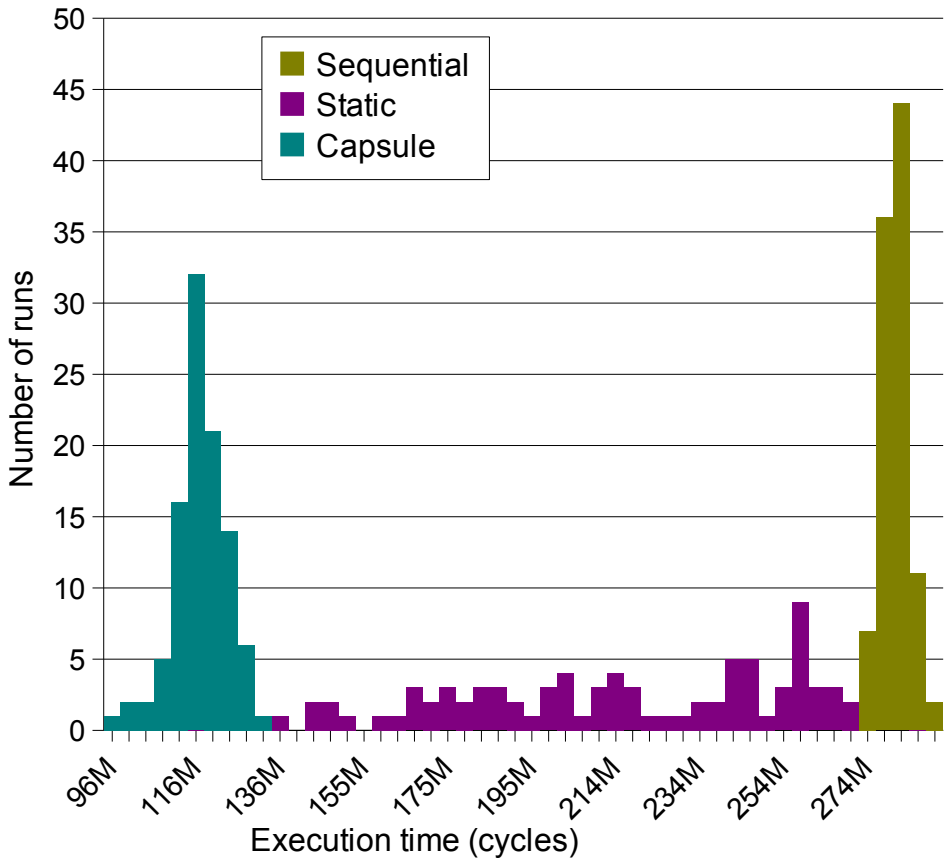
- Peut être utilisé sans souci pour le cas séquentiel

Stabilité du QuickSort

- Comparaisons versions superscalaire, statique et Capsule
- Profil similaire à la version matérielle



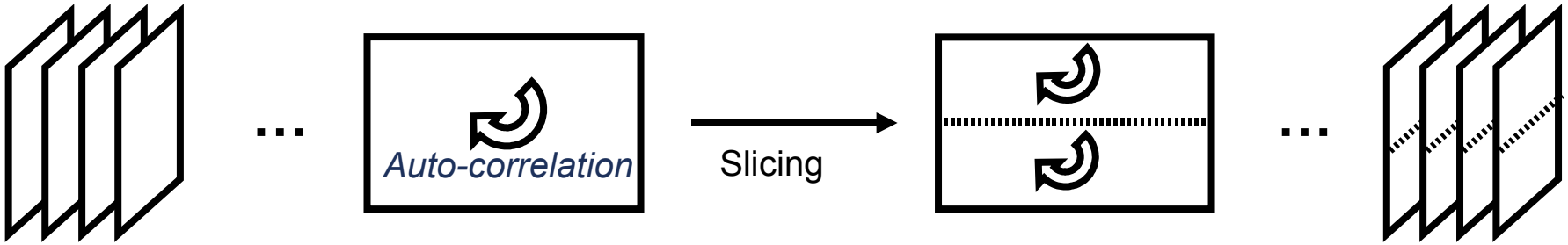
Version matérielle



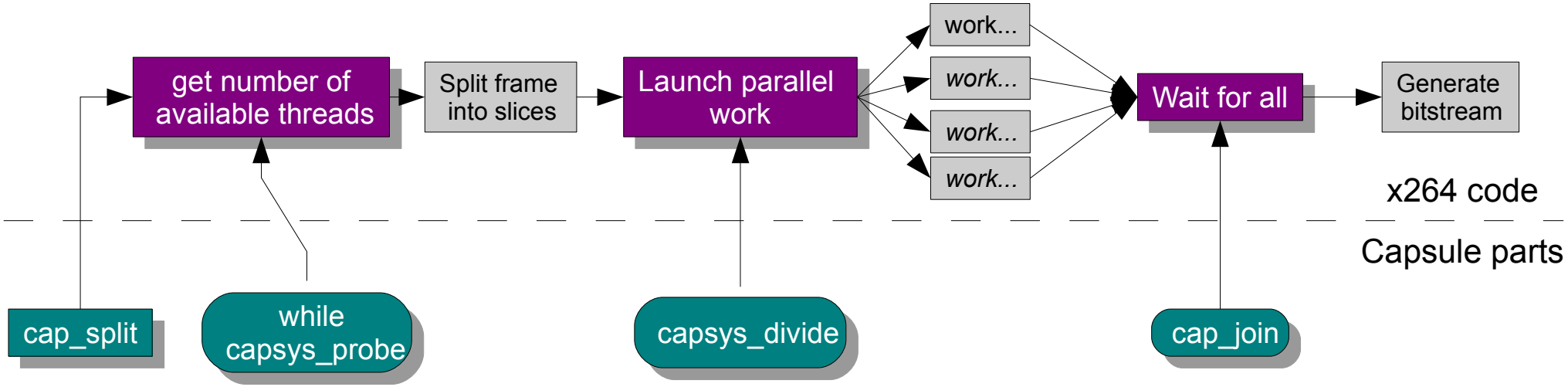
Version logicielle

Gestion de threads : x264

- Encodeur vidéo H.264
- Découpage de l'image en blocs indépendants pour la parallélisation

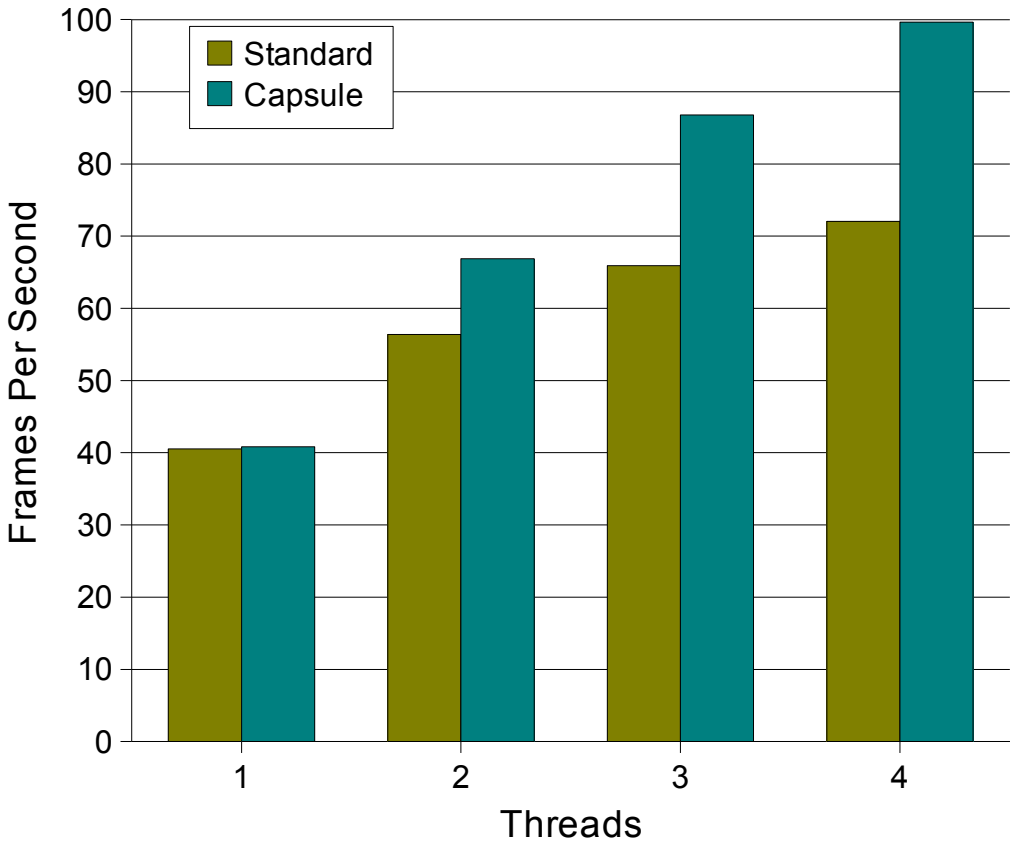


- Réutilisation parallélisme naïf existant
- Capsule en tant que « pool » de threads

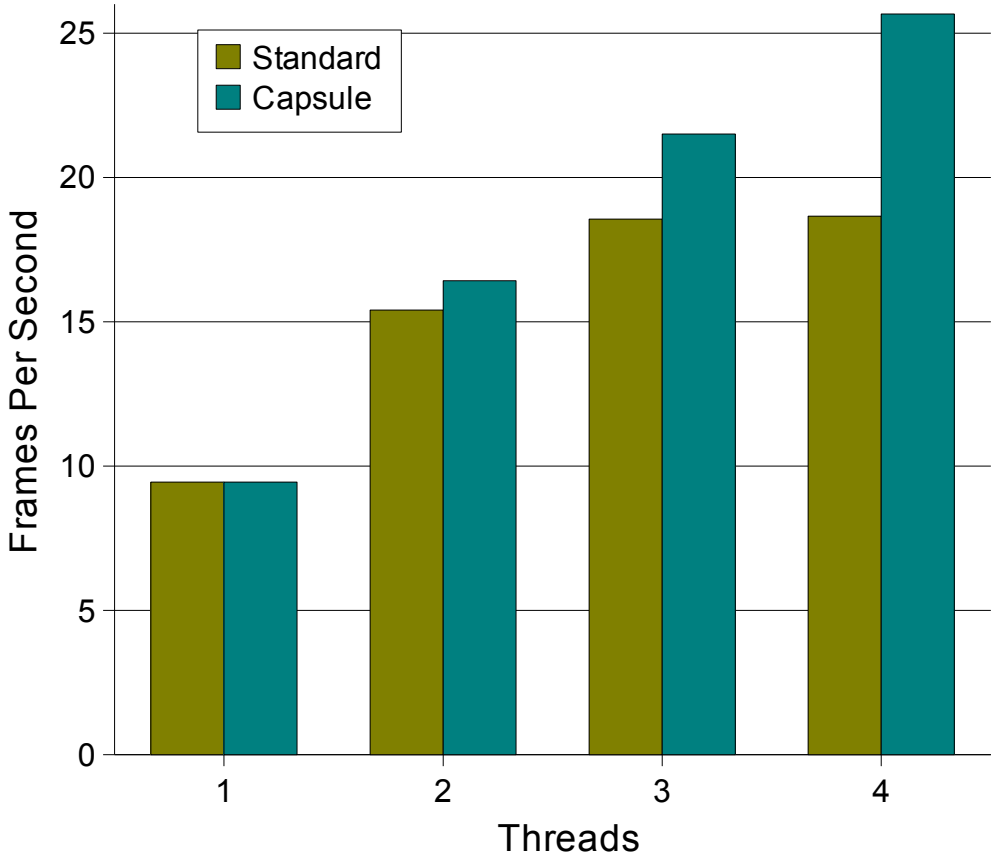


Gains sur x264

- Augmentation taille : ~2% pour 4 slices, ~10% pour 128



Vidéo 480x270

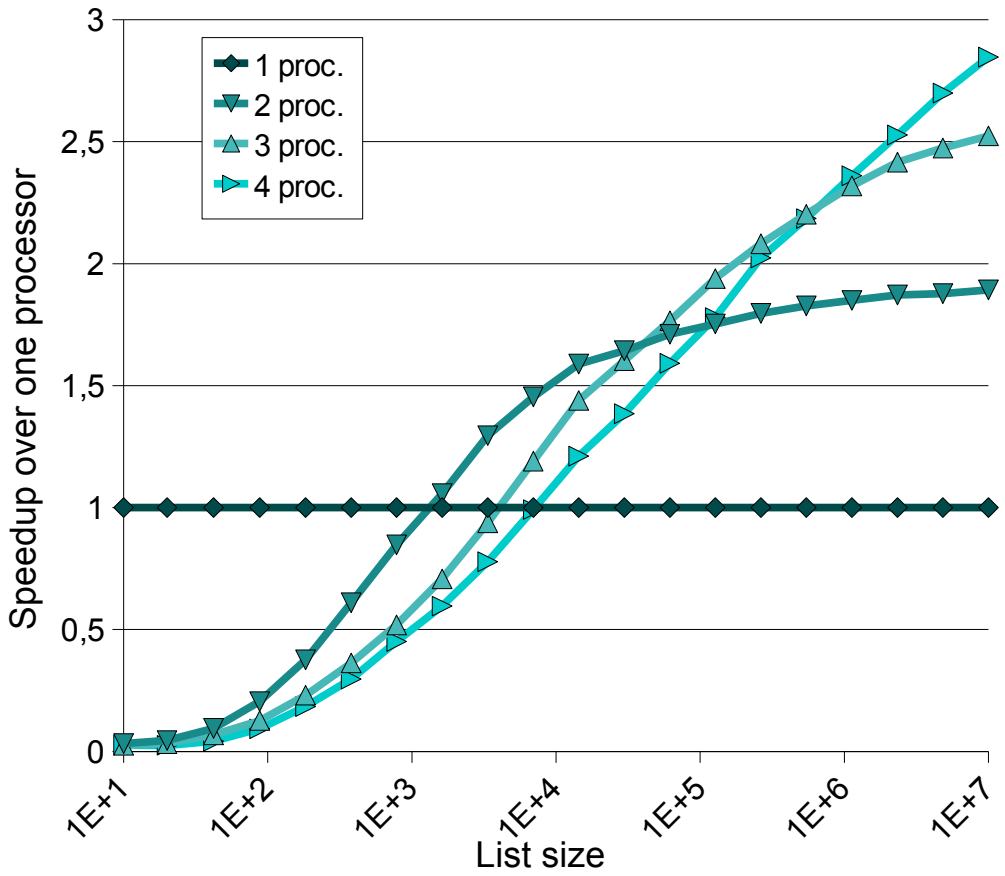


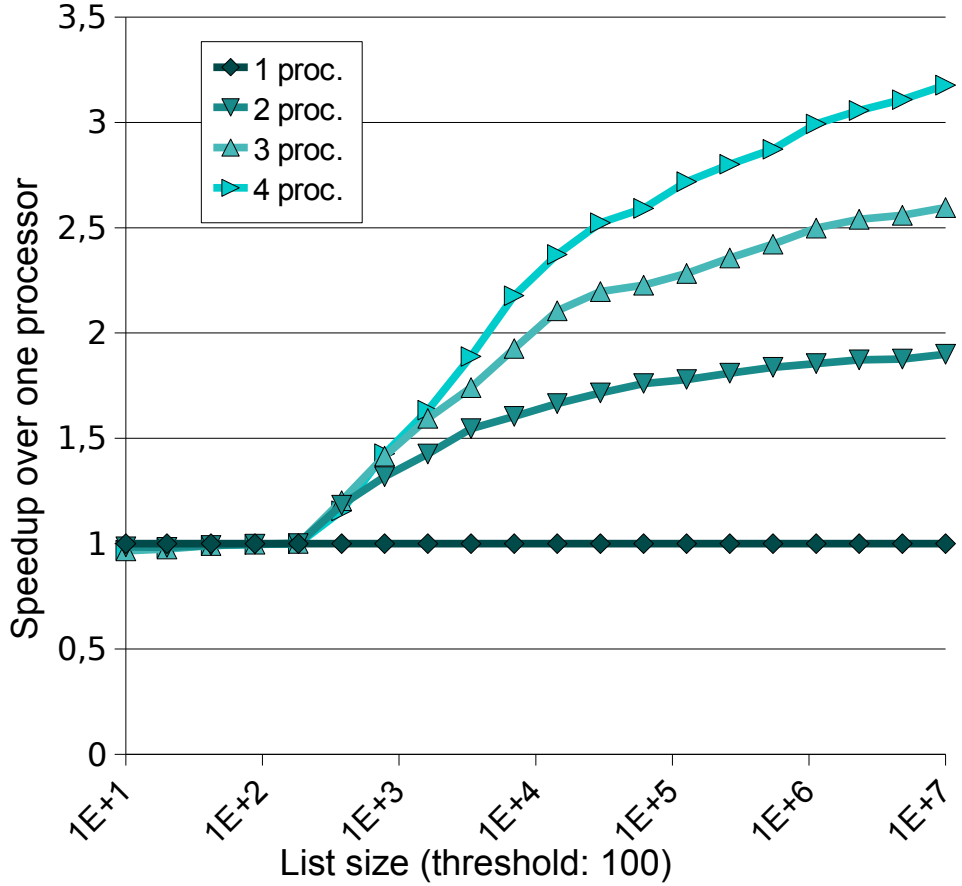
Vidéo 1024x576

- Nouvelle version de x264 avec parallélisme plus évolué

Taille des jeux de données

- Division coûteuse, accentué par le nombre de threads
- Le travail à effectuer doit être plus long que la division
- Quicksort : introduction d'un seuil de parallélisation ad-hoc
- Prédicteur supplémentaire ?
  - Matériel, détections morts récentes
  - Version logicielle ?



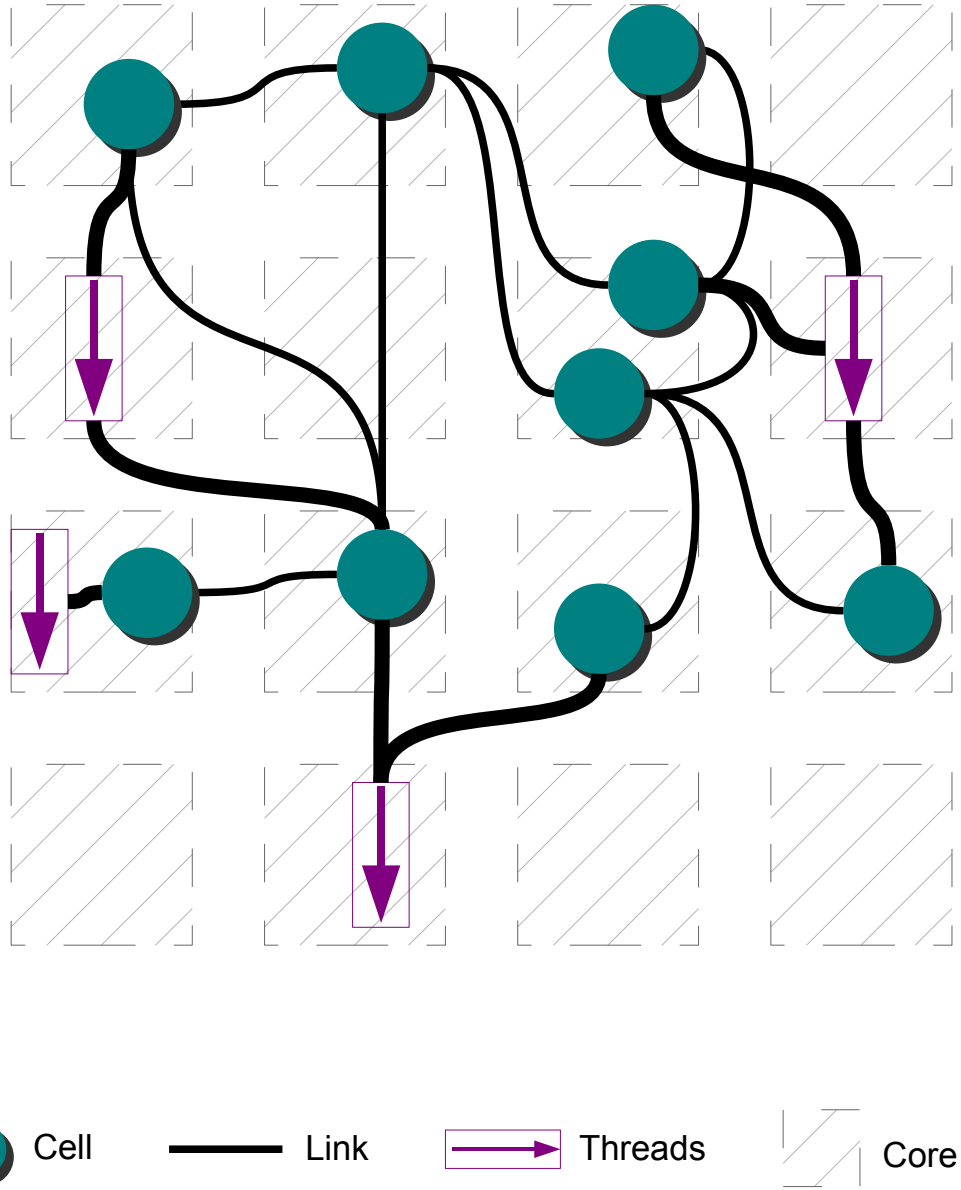


Variation du speedup en fonction de la taille des listes et des threads

Modèle mémoire

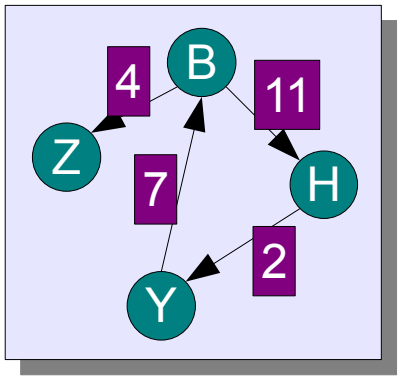
Mémoire par cellules

- Modèle abstrait de machine
  - Machines à mémoire non régulière
  - Structures de données irrégulières
  - Scalabilité
- Cellule
  - Encapsule un élément de structure de données; ~ *struct*
  - Seule entité porteuse d'information
- Lien
  - Relation entre cellules; ~ pointer
  - Pas de données associées
- Suit le modèle mémoire classique de plusieurs langages, peu de contraintes
- Structures mémoire connues par le système



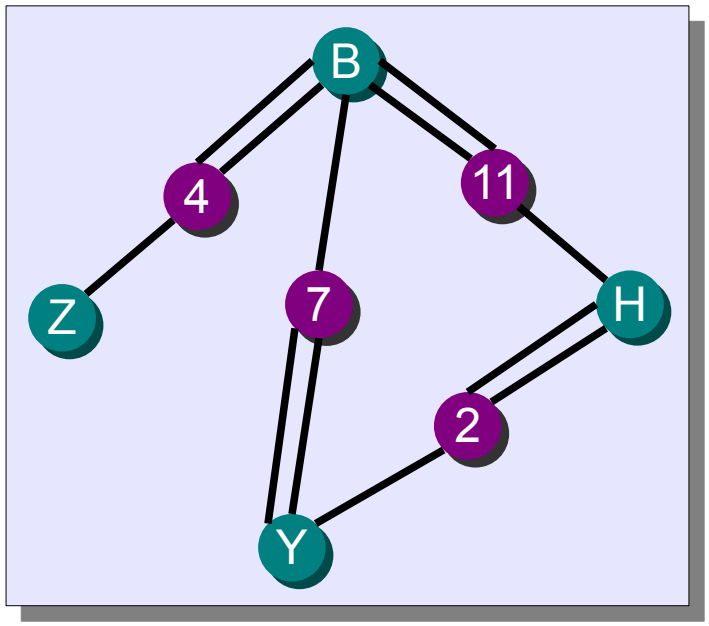
Représentation d'un graphe

- Cellules = données / état
- Les nœuds et arcs d'un graphe sont des cellules
- Équivalent à la représentation classique en C



Graph

Capsule memory




```
struct Edge : capsule::Cell {
   Link<Node> head;
   Link<Node> tail;
   int length;
   Edge(Link<Node> _tail,
      Link<Node> _head, int _length);
};

struct Node : capsule::Cell {
   LinkList<Edge> edges;
   int id;
   int distance;
   Link<Edge> path;

   Node(int _id) : id(_id), distance(-1),
      path(this, NULL);

   // Atomically update distance if needed
   bool update(int newdist, Link<Edge> from);
};
```

Conclusion

Contributions

- Aide à la parallélisation : la division conditionnelle
  - Pas de modifications majeures, tire parti de l'existant
  - Principe simple
  - Modèle et abstraction et de l'architecture
- Version matérielle (processeur multithreadé)
  - Démonstration de l'intérêt de l'approche
  - Prise en compte de métriques de comportement bas-niveau
- Version logicielle (mémoire partagée)
  - Faisabilité d'une adaptation dynamique, sans perte de performance ni modifications matérielles
  - Possibilité d'utiliser le modèle Capsule dès maintenant, transition simple vers un support matériel
- Modèle mémoire (mémoire irrégulière, distribuée)
  - Extraction d'informations sur l'usage de la mémoire
  - Prometteur

Évolutions possibles

- Intégration au niveau du noyau
- Modèle mémoire
  - Adressage, notamment sur des architecture mémoires complexes
  - Gestion efficace des tableaux
- Prise en charge d'architectures telles que Cuda ou Cell
- Intégration dans d'autres environnements, dont des langages de script
- Intégration à la chaîne de compilation

Merci !