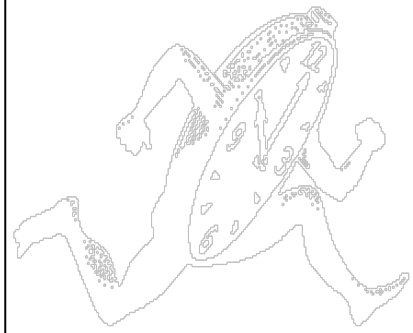
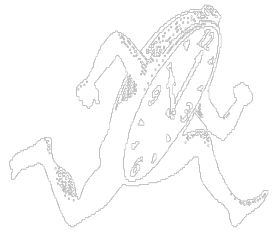
Une plate- forme d'expérimentation pour ordonnanceurs sur machines hiérarchiques

Samuel Thibault
Runtime, INRIA
LaBRI
Université Bordeaux I



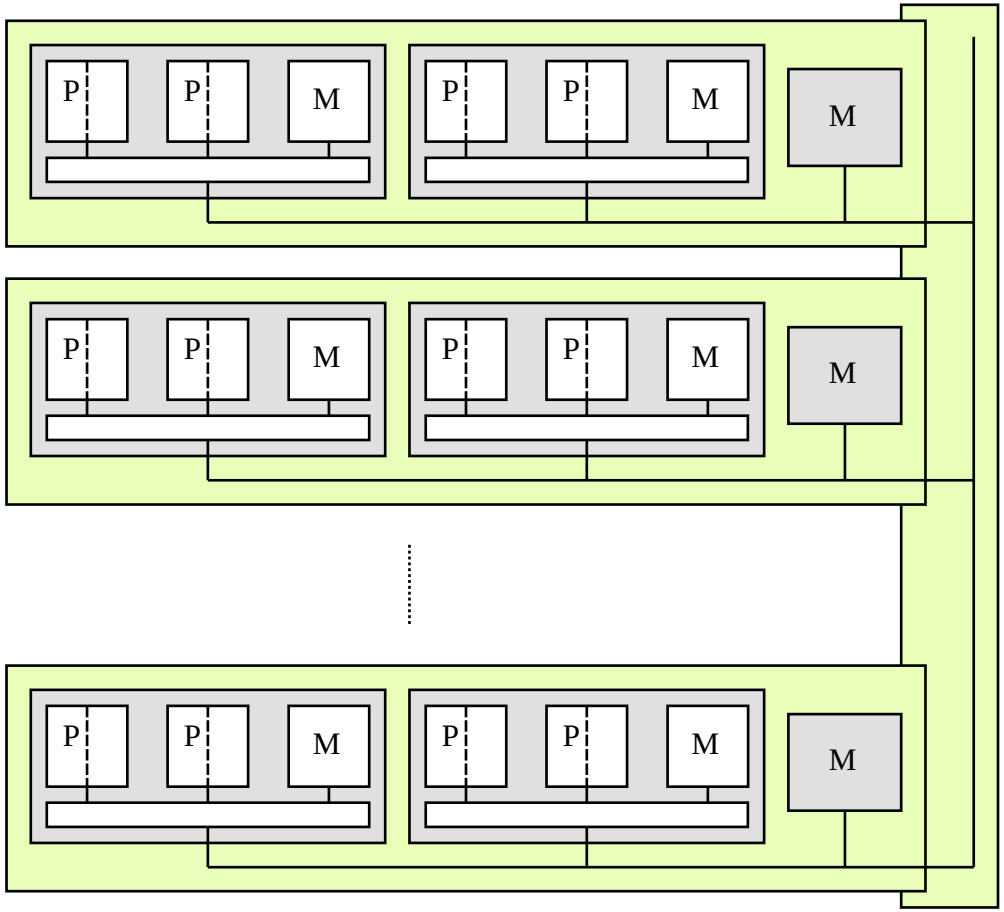
Vers une hiérarchisation accrue des ordinateurs



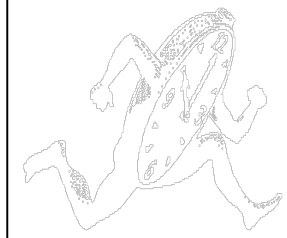
NUMA

MultiCore

HyperThreading



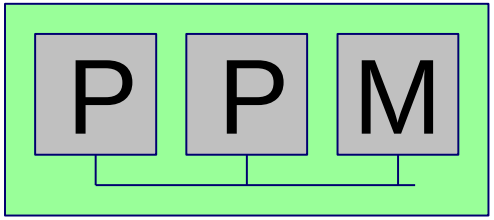
Machines SMP



Juxtaposition de processeurs

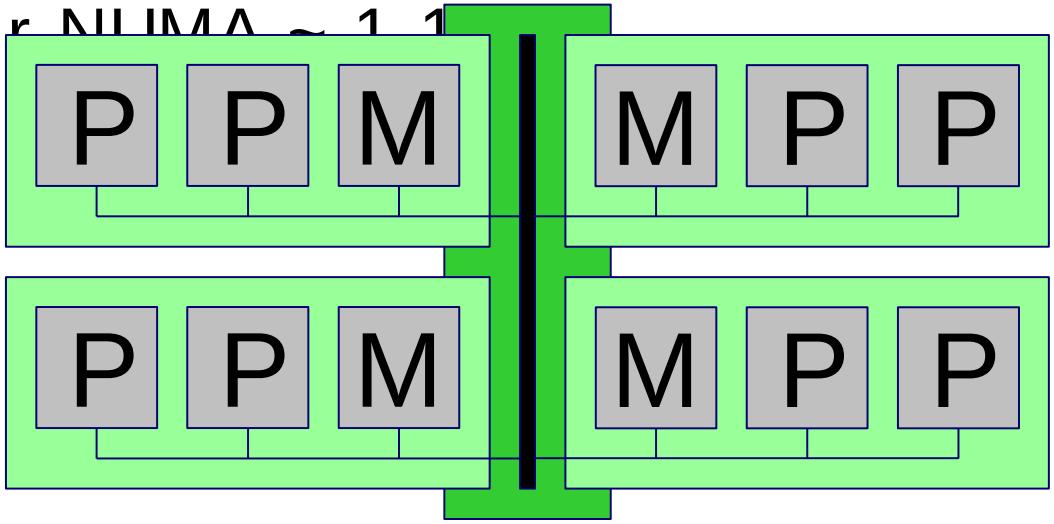
Partage bande passante mémoire

Ne passe pas à l'échelle

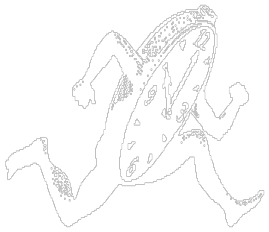


Machines NUMA

- Mémoire distribuée sur différents nœuds
- Latence des accès mémoire « distants » plus grande: facteur « NUMA » ~ 1.3, 2, 10, ...
- Les multiprocesseurs Opteron par exemple
  - Facteur NUMA ~ 1.1

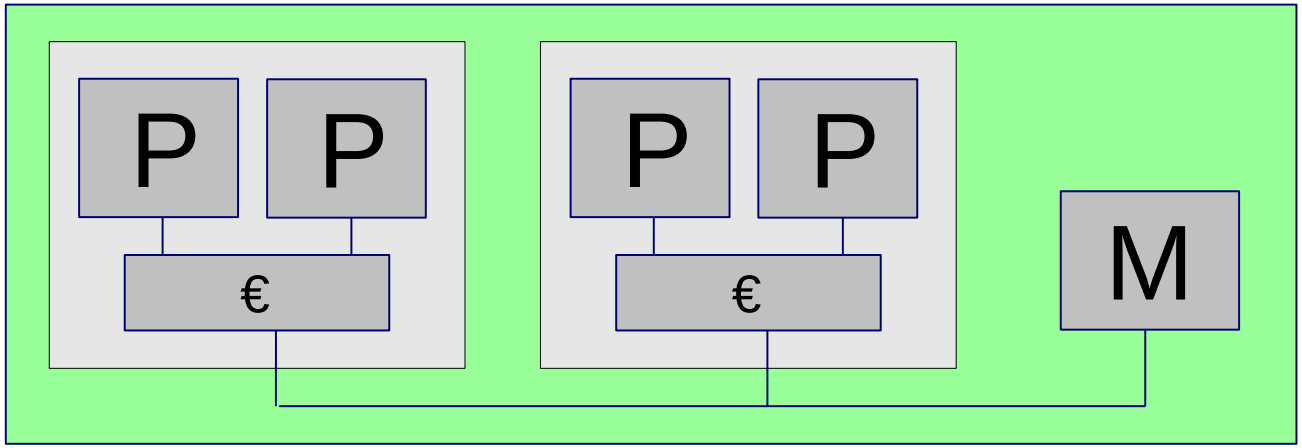


Processeurs multicore

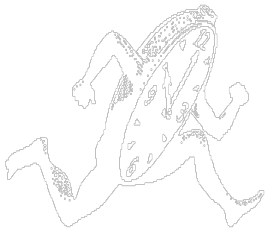


Quand on ne sait plus quoi faire de la place sur la puce...

- Deux processeurs sur une même puce
  - Partage de niveaux de cache

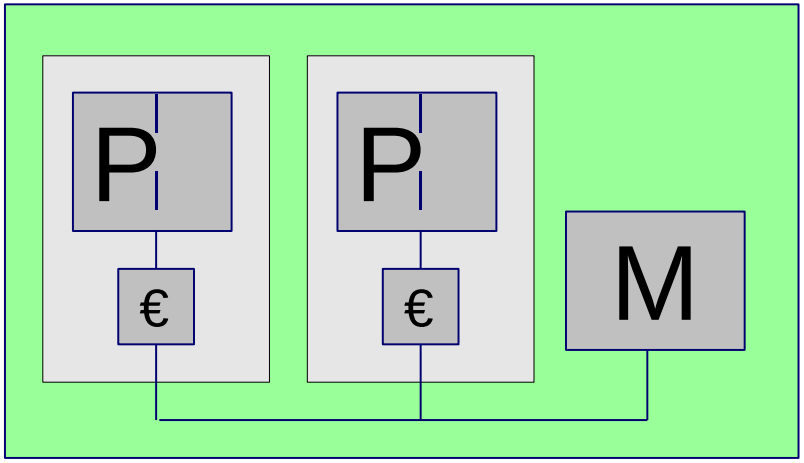


Processeurs HyperThreadés

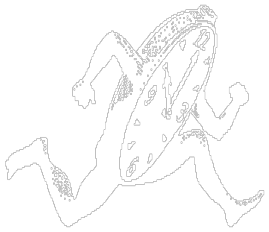


Pour occuper les unités de calcul...

- Exécuter plusieurs threads au sein d'un même processeur
  - Partage des ressources de calcul
  - Partage de tous les niveaux de cache



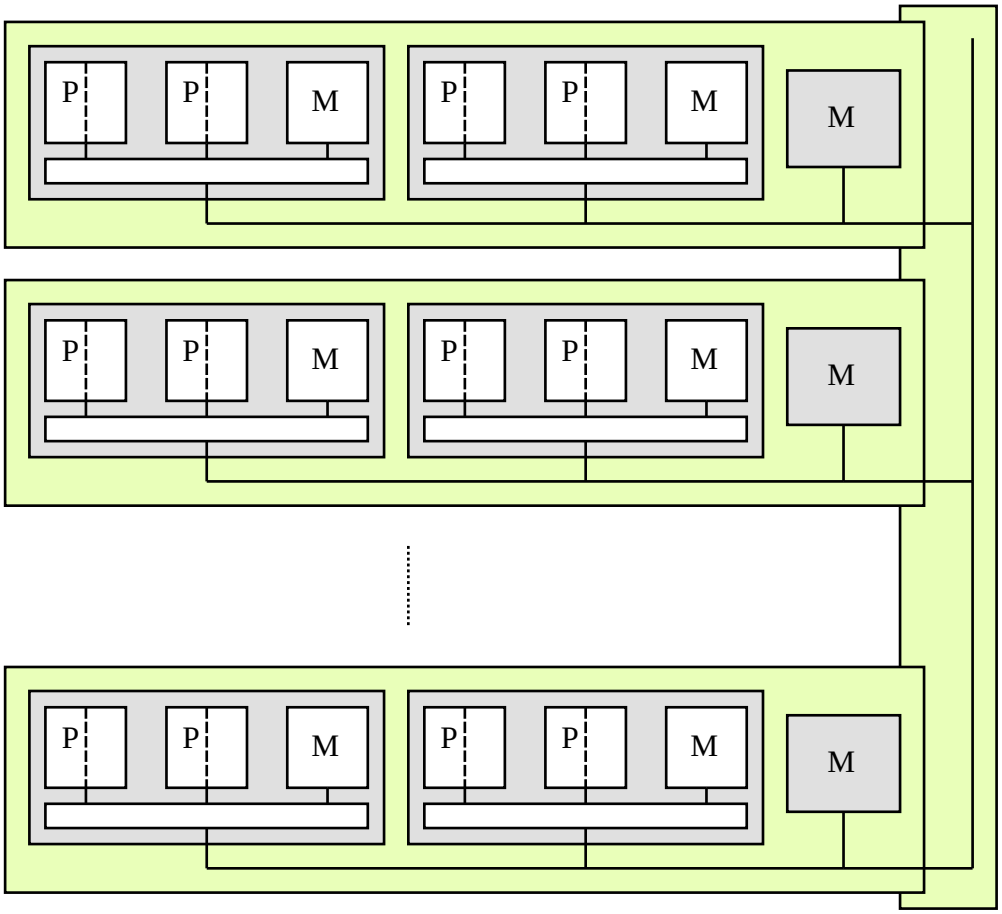
Vers une hiérarchisation accrue des ordinateurs



NUMA

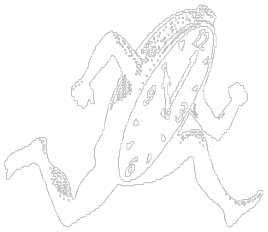
MultiCore

HyperThreading



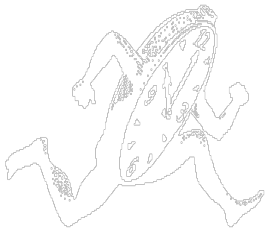
Comment les exploiter efficacement ?

Le challenge : ordonnancer efficacement les applications



- Performances
  - Prendre en compte les affinités entre threads et mémoire
- Flexibilité
  - Pouvoir contrôler facilement l'exécution
- Portabilité
  - Ne pas avoir à adapter les applications à chaque machine

Stratégies actuelles d'exploitation des machines hiérarchisées



Approches:

- Prédéterminée
- Opportuniste
- Négociée

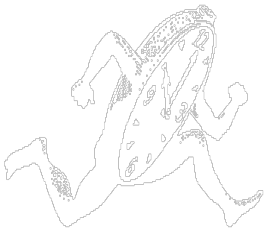
Ordonnancement prédéterminé

- Deux phases
  - Prétraitement: calcul
    - du placement mémoire
    - de l'ordonnancement
  - Exécution: threads et données fixés selon ce prétraitement
- Exemple: PaStiX

✔ Performances excellentes pour des problèmes réguliers

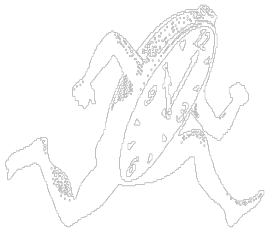
✗ Inadapté aux problèmes fortement irréguliers...

Ordonnancement opportuniste



- Divers algorithmes gloutons
  - Une, plusieurs, une hiérarchie de listes de tâches
- Utilisés par les systèmes d'exploitation
  - Linux, BSD, Solaris, Windows, ...
- ✔ Portabilité : passe bien à l'échelle
- ✗ Performances : n'a pas connaissance directe des affinités

Ordonnancement négocié



Extensions de langages

OpenMP, HPF, UPC, ...

✔ Performances : s'adapte à la machine

✗ Flexibilité : peu générique

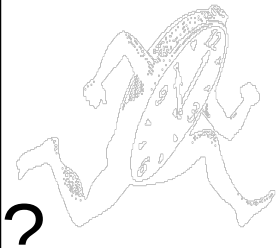
Extensions du système d'exploitation

NSG, libIgroup, libnuma, compteurs de perf.

✔ Liberté du programmeur

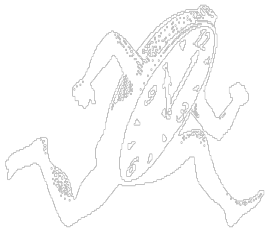
✗ Flexibilité : nécessite la réécriture d'un ordonnanceur

Difficultés



- Quelles stratégies d'ordonnancement ?
  - Préemption ?
  - Gang scheduling ?
  - Round-Robin ?
  - ...
- Selon quels critères prendre des décisions ?
  - Affinités entre threads ?
  - Occupation mémoire ?
  - Consommation bande passante mémoire ?
  - ...
- Pas de solution miracle

Et pourtant



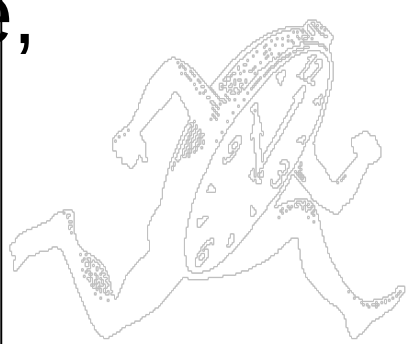
Le programmeur de l'application en a souvent une assez bonne idée

Comment lui « donner le volant » tout en lui épargnant les difficultés d'implémentations ?

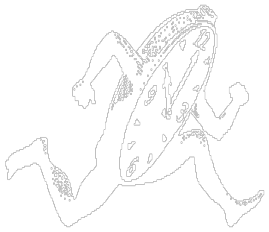
- Manipulation de flots d'exécution
- Synchronisation
- Efficacité
- Portabilité

Proposition: laisser l'application guider l'ordonnanceur

Une approche à la fois précalculée, opportuniste et négociée

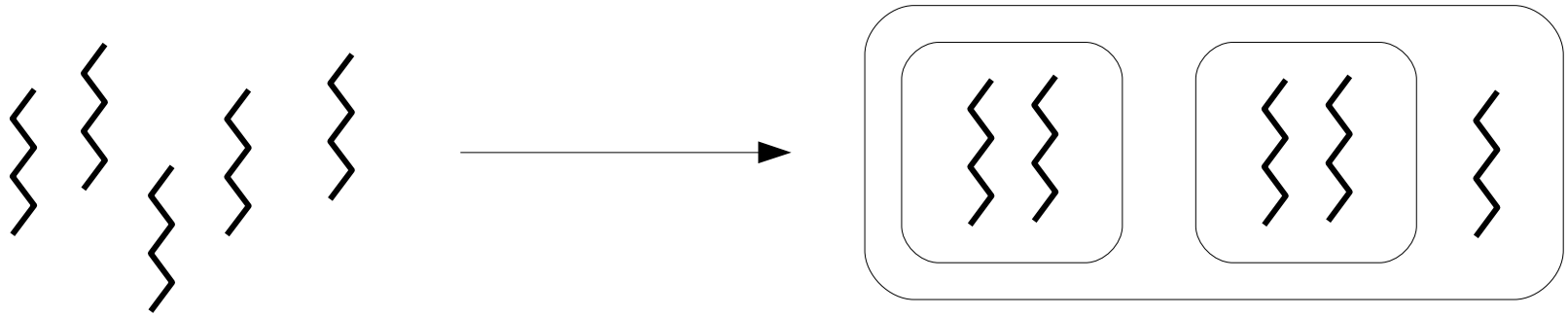


Structure de l'application

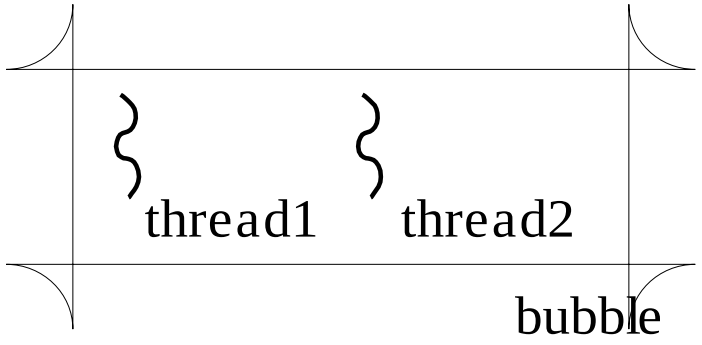


Affinités entre threads exprimées sous forme de bulles

- Partage de données
- Opérations collectives
- ...

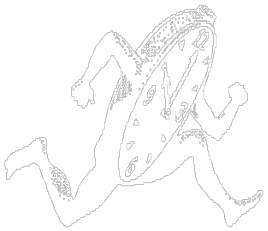


API utilisateur

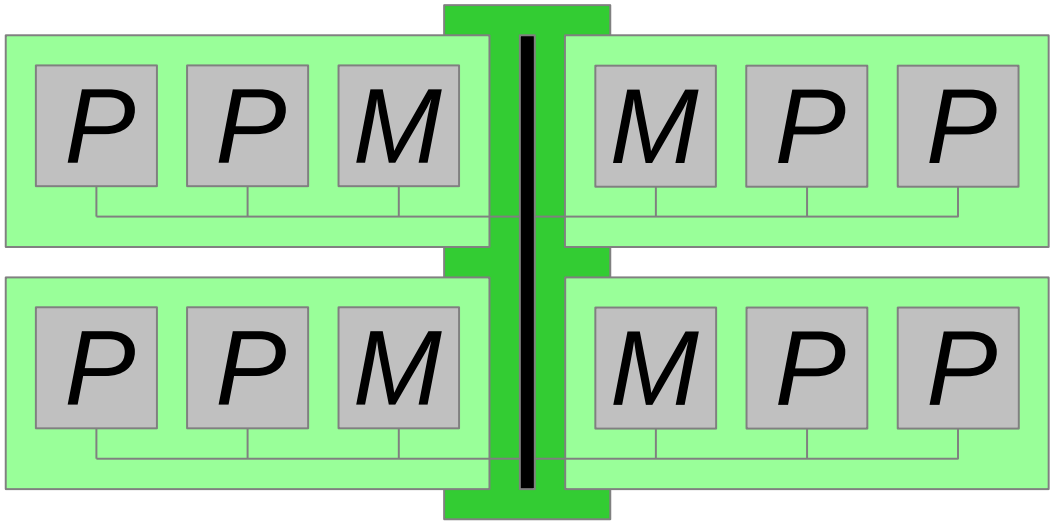


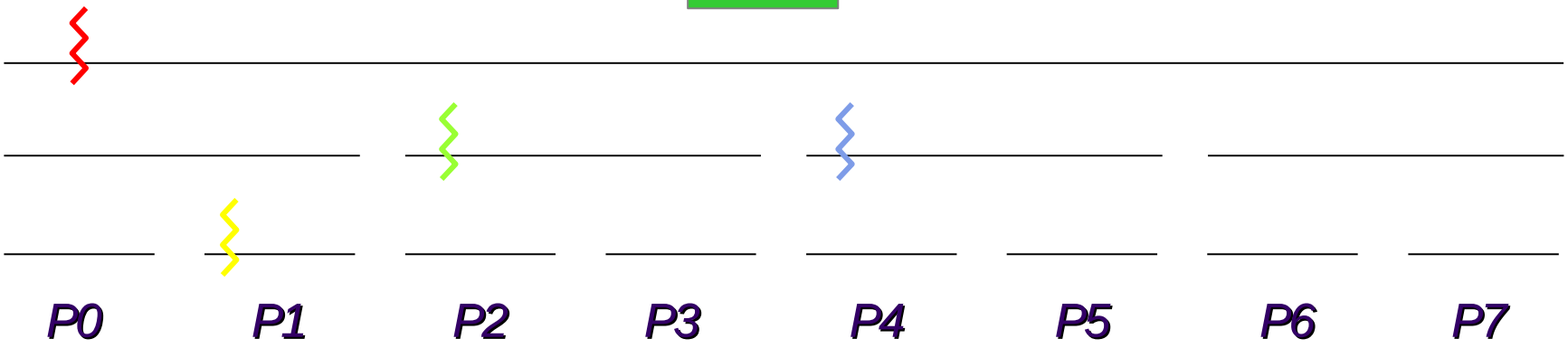

```
marcel_t thread1, thread2;
marcel_bubble_t bubble;

marcel_bubble_init(&bubble);

marcel_create_dontsched(&thread1,NULL,f1,param1);
marcel_create_dontsched(&thread2,NULL,f2,param2);

marcel_bubble_inserttask(&bubble,&thread1);
marcel_wake_up_bubble(&bubble);
marcel_bubble_inserttask(&bubble,&thread2);
```

Structure de la puissance de calcul

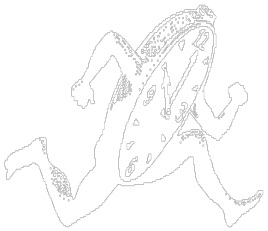


Une hiérarchie de listes de tâches:





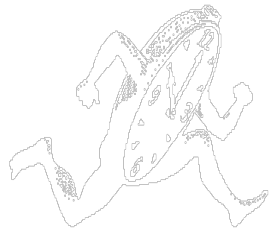
Ordonnancement à bulles



- Exemple d'ordonnanceur
  - Les processeurs « tirent » les bulles et threads vers eux.
  - Les affinités sont prises en compte

bulles_evolution.swf

API ordonnanceur



```
ma_runqueue_t ma_main_rq, ma_node_rq[], ma_cpu_rq[];

ma_runqueue_lock(ma_runqueue_t *rq);
ma_runqueue_unlock(ma_runqueue_t *rq);

runqueue_for_each_entry(ma_runqueue_t *rq, marcel_entity_t **e)
int ma_runqueue_empty(ma_runqueue_t *rq);
marcel_entity_t *ma_runqueue_entry(ma_runqueue_t *rq);

enum marcel_entity ma_entity_type(marcel_entity_t *e);

ma_deactivate_entity(marcel_entity_t *e, ma_runqueue_t *rq);
ma_activate_entity(marcel_entity_t *e, ma_runqueue_t *rq);
```

Un exemple simple

Gang Scheduling à l'aide de bulles

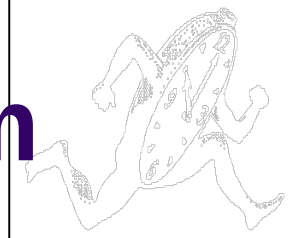
- Chaque bulle contient un gang
- Elles sont placées tour à tour pour exécution

bulles-gang.swf

Implémentation du Gang scheduling

```
while(1) {
   marcel_delay(1);
   ma_runqueue_lock(&ma_main_rq);
   ma_runqueue_lock(&ma_nosched_rq);
   runqueue_for_each_entry(&ma_main_rq, &e) {
      ma_deactivate_entity(e, &ma_main_rq);
      ma_activate_entity(e, &ma_nosched_rq);
   }
    if (!ma_runqueue_empty(&ma_nosched_rq)) {
       e = ma_runqueue_entry(&ma_nosched_rq);
       ma_deactivate_entity(e, &ma_nosched_rq);
       ma_activate_entity(e, &ma_main_rq);
    }
    ma_runqueue_unlock(&ma_main_rq);
    ma_runqueue_unlock(&ma_nosched_rq);
}
```

Outils d'analyse post- mortem



Enregistrement efficace d'une trace des évènements élémentaires

Analyse post- mortem

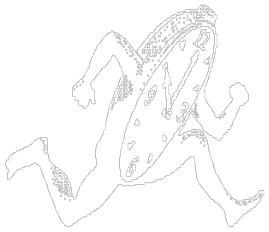
- Observer à volonté ce qui s'est passé
- Intégrer dans les présentations ☺

Implémentation : au sein de Marcel

Bibliothèque de threads utilisateurs de l'environnement d'exécution PM2 développé au sein du projet Runtime

- Portable
- Modulaire
- Performante
- Extensible

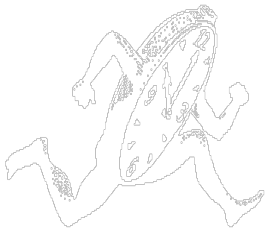
Difficultés d'implémentation



Sont épargnées au programmeur d'applications (c'est le but !)

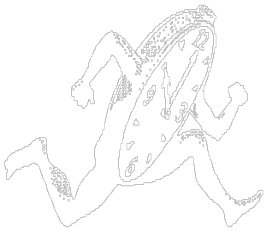
- La gestion du placement d'une entité
  - Son conteneur logique
  - Son placement voulu
  - Son placement actuel
- La préemption
- Le verrouillage
- Le réveil

Un exemple d'application: simulation de conduction



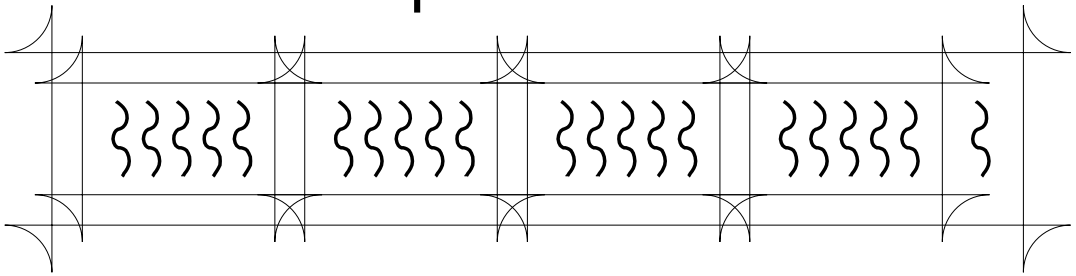
- Simulation de conduction de chaleur dans un matériau (CEA/ DAM)
- Maillage découpé en bandes selon le nombre de processeurs, communications hiérarchisées entre bandes
- Machine cible: NovaScale 5160 (Bull)
  - 4 nœuds NUMA comprenant chacun
    - 4 processeurs Itanium II
    - 16 Go de mémoire

Comment ordonnancer la simulation de conduction ?

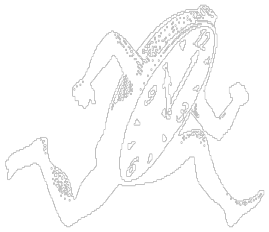


Trois approches

- Marcel simple: ordonnancement opportuniste
- Marcel fixé: fixation manuelle des threads sur les processeurs
- Marcel bulles: stratégie de bulles récursives, réparties automatiquement sur la machine

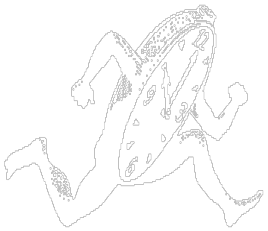


Simulation de conduction: résultats sur 16 processeurs



	Temps(s)	Speedup
Séquentiel	250,2	
Marcel Simple	23,65	10,58
Marcel Fixé	15,82	15,82
Marcel Bulles	15,84	15,80

Conclusion



Machines de plus en plus hiérarchisées

➔ Exploitation difficile

Laisser le programmeur d'application guider l'ordonnancement

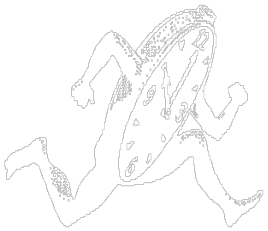
Exprimer la structure de l'application

Ordonnancer des bulles à haut niveau

➔ Portable, flexible et performant

➔ Se concentrer sur l'algorithmie plutôt que sur les détails techniques

Perspective



Informations attachées aux bulles et threads

- Allocations mémoire
- Charge de calcul estimée
- Bande passante mémoire nécessaire

➔ Décisions de placement plus raffinées

Étudier différents ordonnanceurs pour différentes applications

Intégrer dans le noyau

Ordonnancement de bulle basé sur le vol

Encore en réflexion...

vol-enaction.swf