Les architectures parallèles et leur programmation pour le calcul scientifique

Yves Denneulin

Plan du cours

- Introduction au parallélisme
- Le parallélisme de données
  - présentation
  - illustration sur deux langages data parallèles
    - mpl
    - HPF, Open MP
- Le parallélisme d'instructions
  - présentation
  - les catégories
    - passage de messages (MPI)
    - appels de procédures distants (PM2, Athapascan-0)
- Le futur(?) du calcul parallèle : le métacomputing

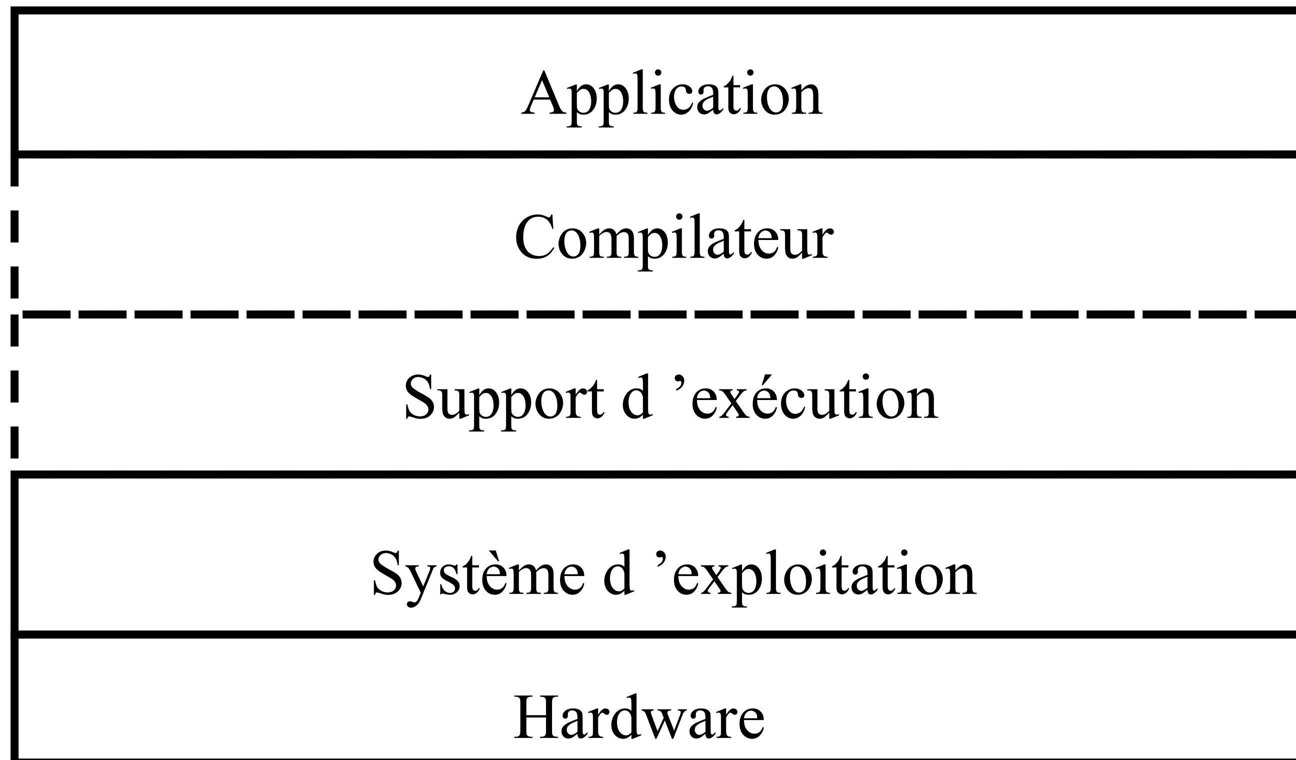
Introduction(?) au parallélisme

- Parallélisme
  - utiliser plusieurs ordinateurs ensemble pour résoudre des problèmes
    - plus gros (taille mémoire, espace disque)
    - plus rapidement (puissance CPU)
  - Mot clé : efficacité
- Différents domaines de recherche
  - théoriques
    - algorithmique
    - ordonnancement
  - pratiques
    - supports
    - modèles

➔ si on veut de l 'efficacité les deux sont évidemment **liées**

Environnement logiciel parallèle

- Composants nécessaires pour l'exécution d 'un programme parallèle



Modèles de programmation parallèle

- Définies par
  - le compilateur
  - le support d ’exécution
- Buts
  - facilité de programmation
    - proche du séquentiel
    - proche d ’un modèle de description d’algorithmes parallèles
      - PRAM
      - BSP
      - ...
  - efficacité = f(algorithme, compilateur, support, système, hardware)
  - portabilité = f(support, algorithme)
    - scalabilité
    - obsolescence du matériel
    - utilisation de plusieurs sites pour la même application (meta-computing)

Les débuts du parallélisme

- Multiprogrammation
  - notion de processus
  - système Multics (~65)
  - toujours d'actualité, au cœur des UNIX et de NT
- Programmation concurrente
  - les coroutines (Djikstra, 68)
  - multiprogrammation à grain plus fin
  - problèmes de synchronisation
- Parallélisme **simulé**
  - pas de machine contenant réellement du parallélisme

Parallélisme matériel : première époque

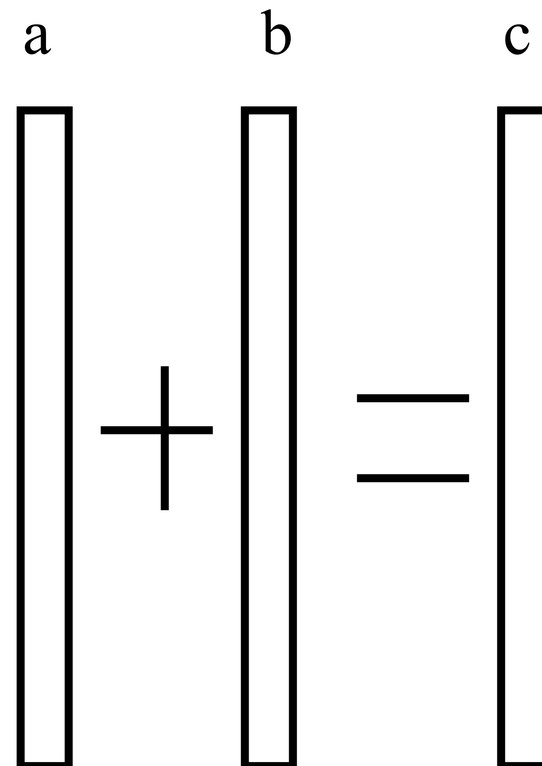
- Processeurs vectoriels (Cray, ~1976)
  - circuit spécifique
  - opérations arithmétiques élémentaires (+, …)
  - données manipulées : vecteurs
  - calcule n additions en parallèle au lieu d'une
    - contient n ALUs au lieu d 'une avec un seul microcontrôleur pour tous
- Traitement parallèle sur des données
  - même opération sur un ensemble de données
  - le parallélisme se fait sur les données (par opposition aux instructions)
  - « Parallélisme de données »
  - fonctionnement « naturellement » synchrone
- Classification traditionnelle
  - SIMD : Single Instruction (flow) Multiple Data

Modèle à parallélisme de données

- Principe algorithmique
  - définir des structures de données (souvent) régulières (vecteurs, matrices)
  - effectuer une suite d 'opérations simples sur ces structures
- Exemple

```
Vecteur a[100];
Vecteur b[100];
Vecteur c[100];

for i:= 1 to 100 dopar
        c[i]:=a[i]+b[i];
done
```

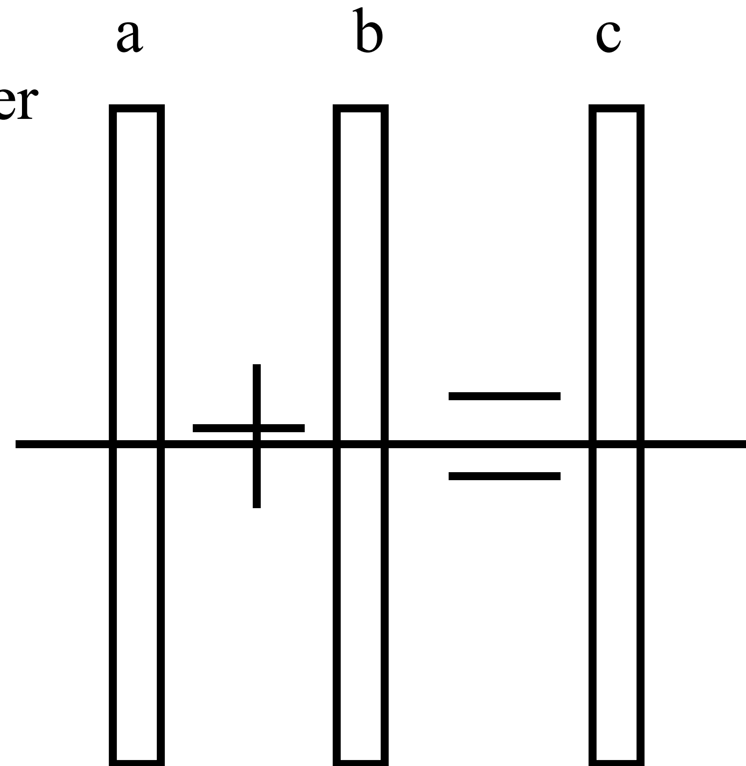


Importance de la compilation

- Le compilateur doit être intelligent
- Découpage de la boucle

Si le processeur vectoriel ne sait calculer que 50 additions en parallèle

-> 2 cycles au lieu d 'un



Caractéristiques

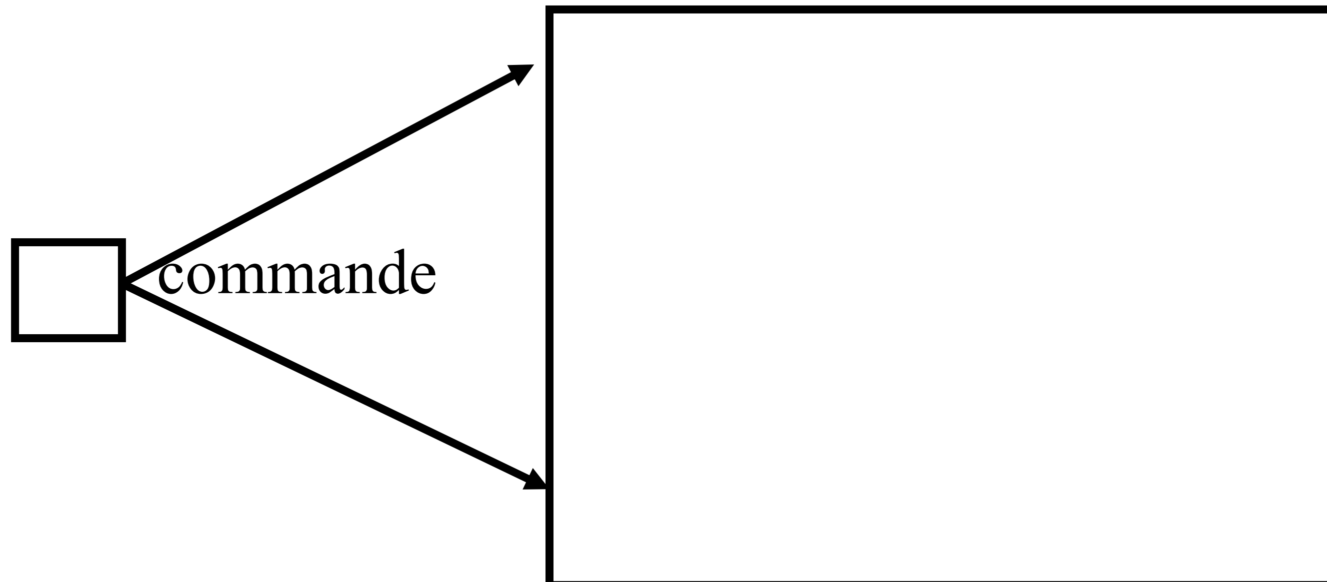
- Modèle simple et naturel
  - programmation simple
    - passage direct à partir de l 'algorithme séquentiel
  - compilateur peu compliqué à écrire
  - efficacité à peu près garantie

➘ pour les architectures vectorielles et SIMD en général
  - cas particulier : le pipe-line (processeurs super-scalaires)

- Adapté à un certain type d'applications
  - calcul sur des structures régulières
  - applications scientifiques et certaines simulations

MPL

- Langage data-parallèle
  - pour une machine précise : Maspar MP-1
  - machine SIMD de 1024 à 16384 processeurs
  - architecture : un processeur performant en scalaire qui commande 16384 processeurs scalaires qui font tous la même chose
  - chaque processeur a une mémoire locale associée
  - topologie des communications : grille 2D torique



Variables parallèles

- Deux types de variables dans le langage
  - scalaire : réside dans la mémoire du processeur maître
  - parallèle : réside dans la mémoire de tous les processeurs esclaves
- Syntaxe à la C

int i;
parallel int a,b,c; // définit les variables a,b et c sur tous les processeurs

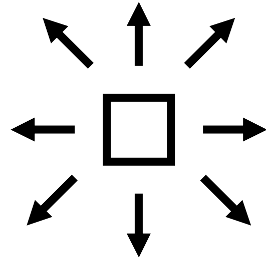
i=1; // effectué sur le maître seulement

c=a+b; // effectué sur tous les processeurs en même temps

a[0]=12; // effectué sur le processeur 0 seulement

Communication entre les processeurs

- Accès à des cases mémoire distantes
  - par localisation géographique dans les 8 directions possibles
    - N,S,E,W,NE,NE,SE,SW



NW[4].a=12; // accède à la valeur de a d ’un autre processeur

Fonctionnement synchrone

- Chaque instruction est décodée par le maître et exécutée par tous les processeurs
  - modèle fondamentalement synchrone
  - pas de conflit d 'accès à des variables possible
- Comment est fait a[0]=12 ?
  - Chaque processeur a un bit d 'activation
  - dit au processeur s 'il doit exécuter l 'instruction ou pas
  - activation calculée par le compilateur ⇒ complexité
- Programmation de bas niveau
  - C-like != fortran par exemple
  - découpage des structures par le programmeur
  - mais compilateur sophistiqué quand même

➘ Parallélisme de données sur architecture SIMD

Évolution architecturale

- Début 90 : SIMD en perte de vitesse
  - composants spécifiques donc chers
  - idem pour les réseaux
  - dépassé en performances pures par des composants standards
  - petit marché donc prix élevé
  - rapidement obsolètes
- Réseaux locaux standards deviennent performants
  - Fast-Ethernet, Myrinet, etc.
  - vous avez vu ça hier
- Les machines parallèles deviennent MIMD
  - processeur puissant standard+réseau d ’interconnexion rapide

Problèmes du data-parallélisme

- Intelligence du compilateur
  - parallélisation automatique des boucles
  - calcul des dépendances entre les données
    - compliqué, sujet de recherche encore actuelle
  - exploitation correcte des machines **même** non SIMD
    - **répartition** des données sur les processeurs
    - problèmes d'**alignement**, de défaut de cache
- Gestion des structures de données irrégulières
  - type liste, arbres, …
  - toutes les dépendances complexes entre objets
  - ↘ le compilateur ne s 'en sort plus!
- Nécessité de langages plus évolués
  - HPF ou Fortran 90

HPF

- Extension avec modification de la norme Fortran-90
- Défini en 93
- But : avoir rapidement des compilateurs pouvant l 'implanter
- Purement orientés données, ne contient plus de primitives de gestion de messages ou de synchro
- Vise à l 'efficacité sur architectures distribuées aussi
- Parallélisation non complètement automatique
  - directives fournies par le programmeur pour le placement des données
  - mais conserve un modèle data-parallèle (boucles FORALL)

Le modèle HPF

- Modèle à 4 niveaux
  - Objets HPF « normaux » (tableau)
  - mappés sur des *templates* qui représentent des groupes d 'alignement
    - définit les contraintes d 'alignement sur et entre les objets
    - permet de spécifier aussi la façon de les distribuer sur les processeurs
  - ces templates sont mappés sur des **processeurs virtuels** (machine abstraite possédant une topologie), spécification d'une distribution
  - les processeurs virtuels sont mappés sur les processeurs physiques
- Intuitivement
  - une opération entre deux objets est plus efficace s 'ils sont sur le même processeur
  - ces opérations peuvent être exécutées en parallèle si elles peuvent être réalisées sur des processeurs différents

Fonctionnement de HPF

- Spécification de dépendances entre les objets
  - par le mapping sur les templates
  - possibilité de lier deux objets pour les aligner de la même manière
  - les opérations entre objets alignés à l 'identique sont plus efficaces
  - possibilité de réaligner dynamiquement les objets
- Distribution des objets
  - définit le mapping des objets sur un **ensemble** de processeurs **abstraits**
  - but : rapprocher sur la grille des processeurs abstraits les objets qui interagissent
  - deux possibles : BLOCK (suite d 'éléments contigües) et CYCLIC (pareil mais avec bouclage)
- Instruction FORALL
  - ~= au for parallèle mais en plus puissant (description d 'intervalles,…)

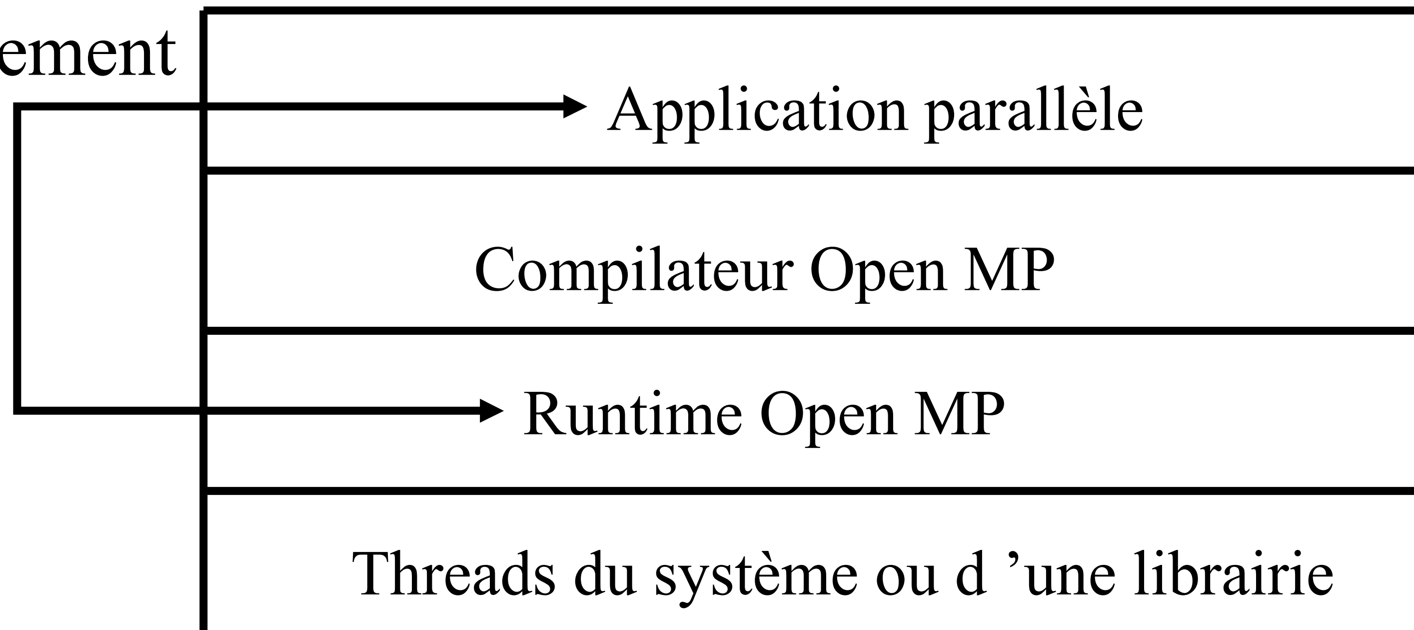
Fonctionnement de HPF(2)

- Expression d ’indépendance d ’instructions
  - par exemple deux instructions dans une boucle
- Spécification de fonctions « locales »
  - ne nécessitent pas de communications pour être exécutées -> efficacité
- Résumons
  - langage data-// pour architectures SIMD et MIMD
  - alignement/distribution des données
  - directives d ’aide au compilateur
  - mélange des apports de différents langages
  - largement utilisé car disponibilité de compilateurs (Adaptor)

Le modèle Open MP

Présentation de Open MP

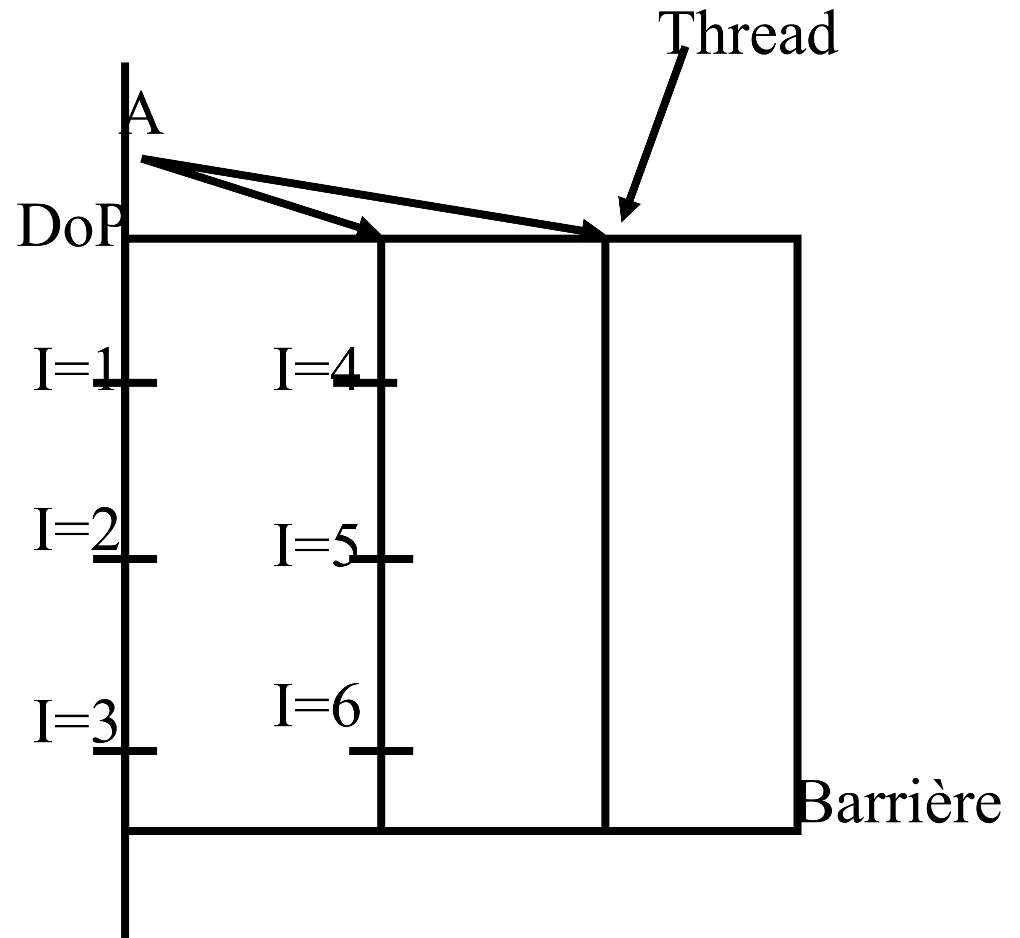
- **API** portable, orientée mémoire partagée
  - existe pour C, C++, Fortran 77 et 90
  - existe sur plusieurs architectures (UNIX et NT)
- Conçu pour du parallélisme à grain fin
  - essentiellement pour les boucles
  - utilisable aussi pour des algorithmes à grain moyen
- Positionnement



Exemple de boucle

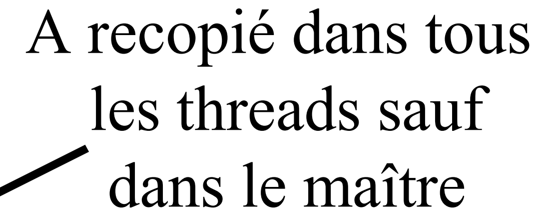
- Annotation sur les threads, les objets partagés et privés
  - A est partagé, I est privé (local à chaque thread)

```
     program
c$omp parallel do
c$omp& shared (A)
private(I)
     do I=1,100
     ...
     enddo
c$omp parallel
     end
```

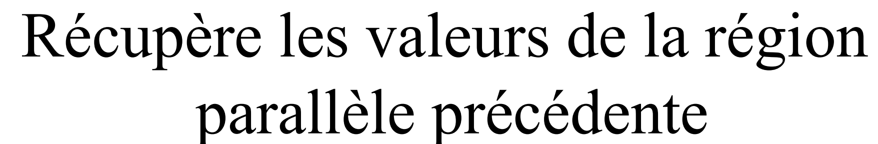


Duplication et recopie de variables

- Recopie temporaire de variables partagées (en C)




```
#pragma omp threadprivate(A)
struct Astruct A;
#pragma omp parallel copyin(A)
{ ... }
/* région parallèle suivante */
#pragma omp parallel
{ ...}
```

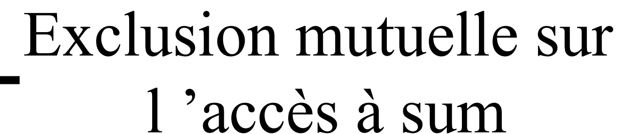


Section critique et partage de variables

- Barrière et exclusion mutuelle

```
#pragma omp parallel private(i,j)
     shared(a,b,n,m,sum)

     for(j=1;j<=n;j++)
#pragma omp for
          for(i=1;i<=n;i++) {
#pragma omp critical
               sum=sum+a[i,j];
          }
#pragma omp barrier
     b(j)=sum;
     }
```



Comparaison avec les autres modèles

	MPI	Threads	OpenMP
Portable	√	~=√	√
Scalable	√	√	√
Performance	√		√
Data-parallel	√		√
Haut niveau			√
Proche du séquentiel			√
Prouvable			√

Particularités

- Ordonnancement peut être effectué par le runtime
  - directive `schedule` spécifiable pour chaque boucle
- Possibilité de relaxer les contraintes de symchronisation
  - laissé des threads continuer leur exécution même si tous n 'ont pas fini la boucle

Pourquoi utiliser OpenMP ?

- API **portable** et **standard** pour toutes les machines à mémoire partagée
- extension de langages existants
- facilité de portage d 'un code séquentiel
- suffisament général pour des applications data-parallèles
- compilateurs largement disponibles

Conclusion (provisoire) sur le data-//

- Premier type de parallélisme largement utilisé
  - calculateurs vectoriels
  - architecture SIMD
- Adapté au
  - traitement sur des données régulières
  - architectures fortement couplées
- Programmation aisée
  - mais écrire les compilateurs est très compliqué
    - évolution architecturale (importance des caches, des pipe-lines, …)
    - constructeurs jugés plus sur la qualité du code produit que sur les caractéristiques de la machine (ex : Cray)
- Largement utilisé pour beaucoup de codes scientifiques
  - météo, aéronautique, ...

Parallélisme matériel : seconde époque

- Fin des années 80
  - déclin des architectures tout-propriétaires
  - exemple type : Cray avec le T3D (processeurs ALPHA, réseau haut débit propriétaire)
  - déclin des architectures SIMD : nouveaux types d 'applications parallèles envisageables
  - émergence de nouveaux modèles de programmation

→ Orientés **instructions**

- Tendance actuelle
  - processeurs standards + réseau standard (structure de grappes de machines) : pas trop cher
  - architectures spécifiques (SGI Origin 2000, T3?) : chers, offrent des fonctionnalités spécifiques tant matérielles que logicielles

Parallélisme d'instructions

- Parallélisme « traditionnel »
  - application parallèle = ensemble de flots d'exécution concurrents
  - programmation guidée par les instructions
  - nécessité de partager de données, éventuellement d 'opérations de synchronisation, …
  - structuration en tâches /= en blocs de données sur lesquels effectuer des opérations
- Application parallèle = ensemble de tâches partageant des données (y accédant en parallèle)
- Deux problèmes principaux
  - comment structurer une application en tâches ?
  - comment communiquer pour
    - partager les données,
    - se synchroniser ?

Paradigmes de programmation

- Interface des systèmes d 'exploitation peu adaptée
  - gestion des processus : `fork`, `exec`, …
  - Communications : IPC, sockets, XDR, …
- Différentes abstractions fournies par les supports d 'exécution
  - Activités
    - Acteur, processus, objet actif, *threads*, …
  - Communication
    - Envoi de messages (synchrone ou asynchrone)
    - Mémoire partagée (objet, page, synchronisation, cohérence, …)
    - Appel de procédure à distance

➘ Définissent des **modèles** de programmation

Schémas de programmation parallèle

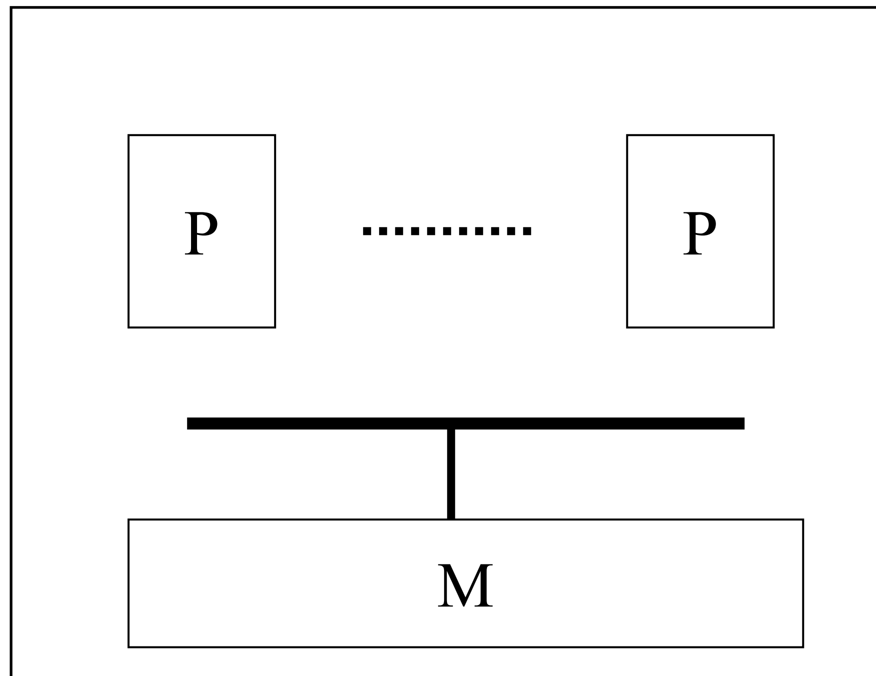
- Maître/Esclave
  - un coordinateur lance toutes les tâches esclaves
  - coordination centralisée du travail et des I/O
- SPMD
  - le même programme s ’exécute sur différentes parties du même problème
- Asynchrone (MIMD?)
  - ensemble de programmes différents
  - chaque programme exécute une fonction différente de l'application
- Schémas hybrides
  - data-flow, task farming, …

➘ Définissent des schémas de communication et de synchronisation != modèles de programmation

Architectures de type SMP à mémoire partagée

Caractéristiques des SMP

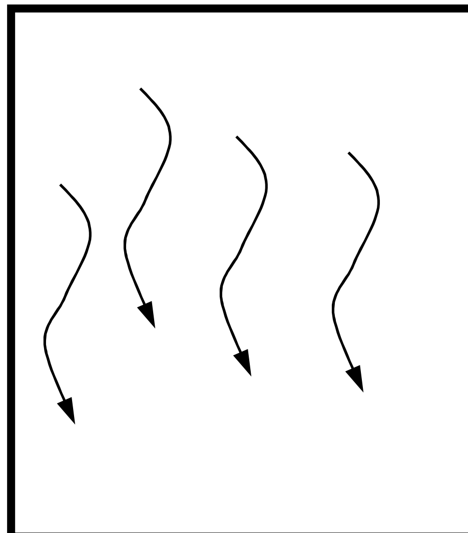
- Processeurs partagent physiquement la mémoire
  - accès au même bus
- Parallélisme matériel au sein du système
  - ex: ensibull : 8(?) processeurs



Comment exploiter ce parallélisme ?

Exploitation des SMP

- Applications « traditionnelles »
  - un processus UNIX par CPU
  - partage du temps au niveau du processus lourd
- Applications parallèles
  - parallélisme **intra**-processus
  - nécessité de pouvoir multiprogrammer un processus lourd
  - activités encapsulées dans le même process UNIX
    - ⇨ processus légers (threads)



Les processus légers (threads)

- Principe
  - processus lourd encapsule les données
  - processus léger contenu dans un processus lourd
  - contexte d 'exécution seulement : peu coûteux
- Historique
  - utilisé pour la programmation système
    - démons
    - recouvrement latence communication/accès disque
  - plus récemment utilisé pour le calcul
    - permet l 'exécution efficace de petites tâches
    - exploitation des SMP

Les threads POSIX : Pthreads

- Interface standard de programmation pour les threads
  - portabilité des programmes utilisant les threads
- Fontionnalités
  - création/destruction de threads
  - synchronisation (conditions, mutex, sémaphores)
  - ordonnancement, priorités
  - signaux
- Attributs d ’un thread POSIX
  - caractérisation de son état
    - pile, ordonnancement, données privées

Création d ’un thread POSIX

- Crée un thread avec les attributs attr
- exécute la fonction start_routine avec arg comme argument
- tid : identifieur du thread créé (équivalent au pid UNIX)

```
int pthread_create (
          pthread_t *tid,
          pthread_attr *attr,
          void* (*start_routine)(void *),
          void *arg);
```

+ Fork/join
+ Primitives de synchronisation

Utilisation des threads pour le calcul

- Exemple : produit de matrices
  - un thread = calcul d ’une case
  - un thread = calcul d ’une ligne/colonne
- Granularité quelconque
  - impossible avec des processus lourds : coût excessif
  - efficacité => coût gestion activités << coût activités
- « Assembleur » de la programmation parallèle
  - peut amener une grande efficacité
  - pénible à utiliser
- Quid des SMPs ?
- Quid des threads en distribué ?

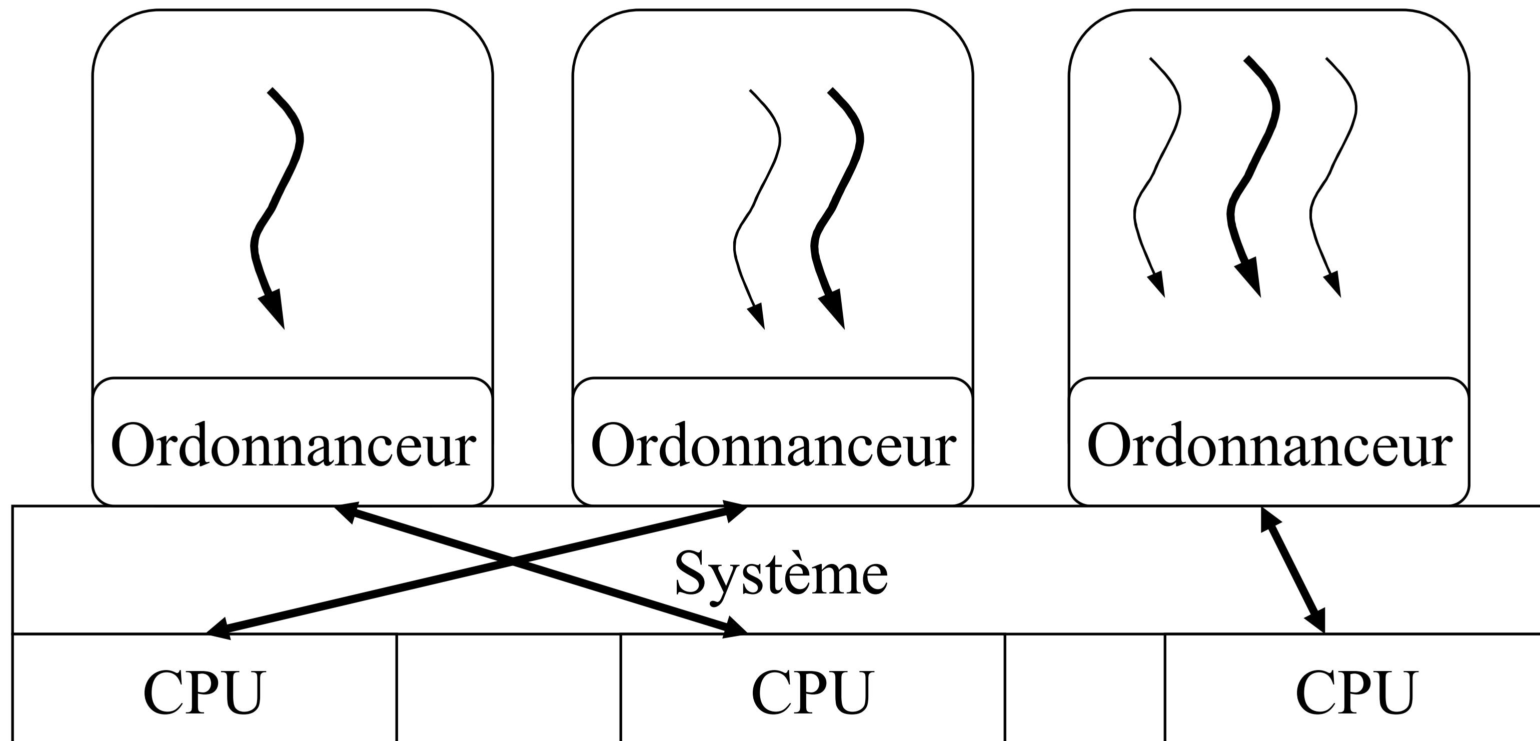
Utilisation des threads pour exploiter les architectures SMP

Plan

- Les différents types de threads
  - Threads utilisateurs
  - Threads système
- Correspondance entre les classes de threads dans Solaris
- Ordonnancement
  - Politique standard
  - Politique spécifique (priorité, temps réel, …)
- Conclusion

Les processus légers utilisateurs

- Liés à l’espace mémoire d’un processus
- Indépendant du système sous-jacent



Avantages

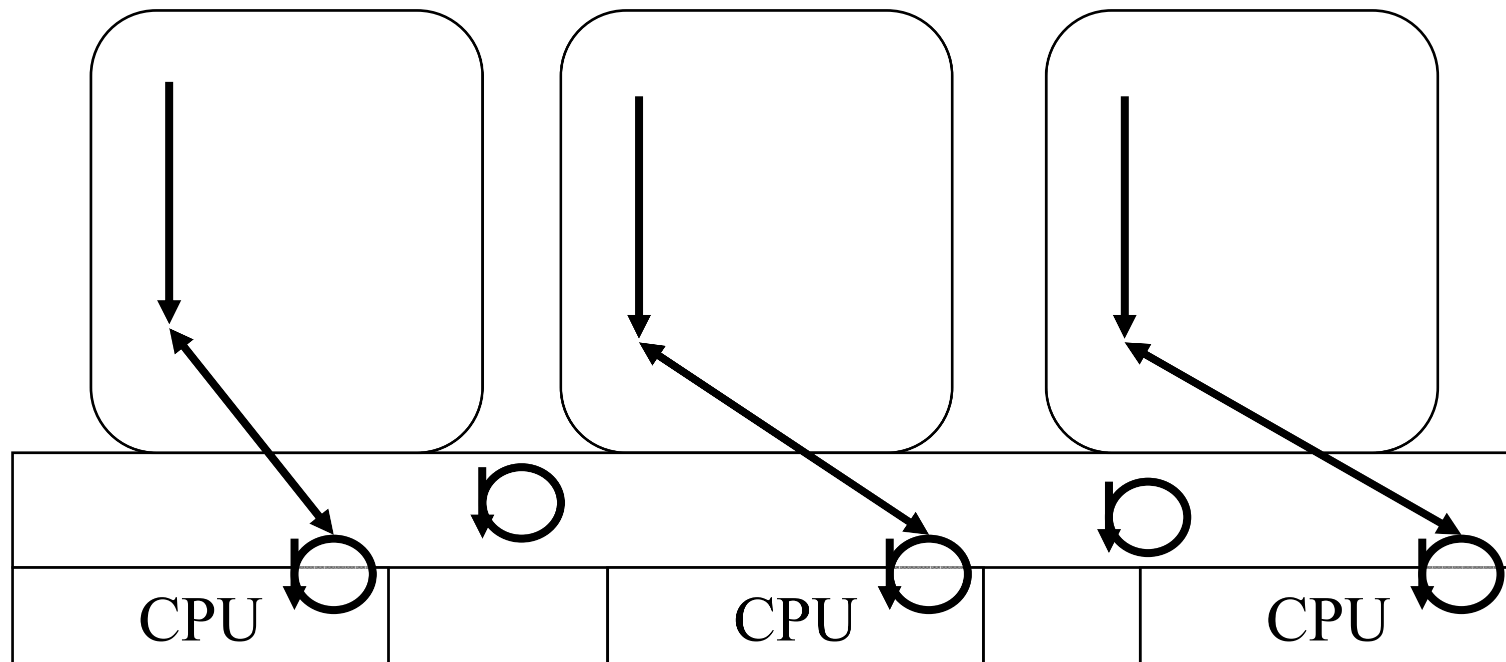
- Faible coût des opérations
  - Création, destruction
  - Changement de contexte « léger »
  - Possibilité d’en avoir un grand nombre
- Souple
  - Répartition du temps gérable par le programmeur, pas de risque de blocage du système

Inconvénients

- Pas de parallélisme intra-processus
  - Sous exploitation des machines SMP
- Partage du temps à deux niveaux
    - Entre les processus lourds
    - Entre les processus légers d'un même processus
  - Pas de partage global au prorata du nombre de processus légers
- Pas de protection mémoire entre les activités

Les processus légers système

- Gérés par le système d 'exploitation
- Existent dans le noyau hors des applications



Avantages

- Exploitation des SMP
  - Interactions entre les activités applicatives et les activités noyau ?
- Ordonnancement global de toutes les activités
  - Prise en compte par le système
  - Répartition équitable (?) entre les utilisateurs au prorata de leurs activités
- Mécanisme de protection entre les activités

Inconvénients

- Gestion coûteuse
  – Passage par le noyau = surcoût important

  ➔ « Medium-weight » threads
- Plusieurs activités accèdent au noyau simultanément

  ➔ **Réentrance** obligatoire du noyau
- Ordonnancement non contrôlable par le programmeur

Historique de SOLARIS

- Années 80
  - SunOS 4 (Solaris 1) : Noyau monolithique non réentrant de la famille BSD
  - Bibliothèque de threads utilisateurs LWP au standard POSIX
- Réécriture complète début des années 90
  - Solaris 2, threads système, standard SysV, réentrant => Support des SMP

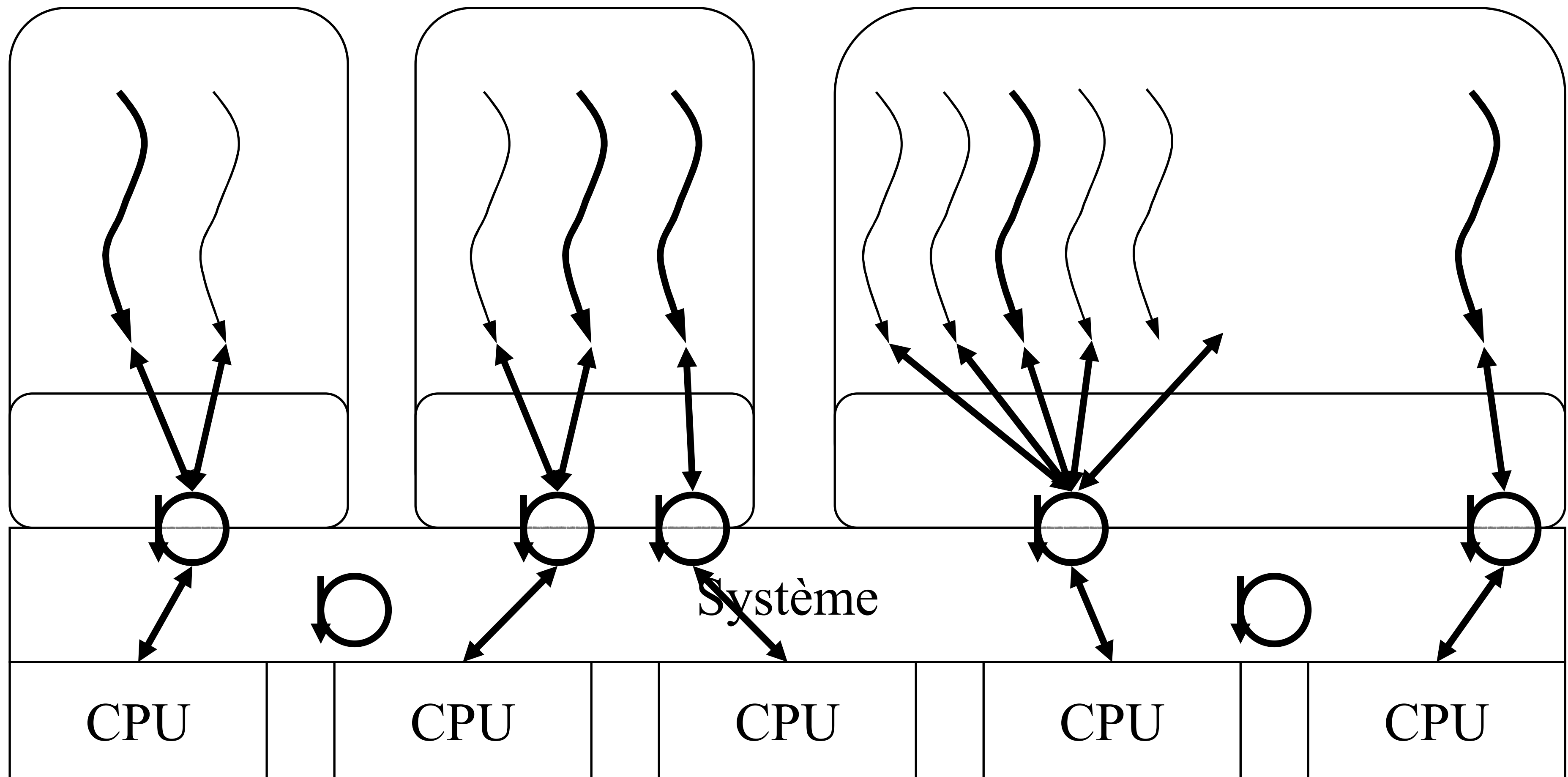
Les threads de Solaris

- Deux types de threads
  - Système : propres au noyau, schedulés par lui sur les processeurs => entités d'exécution
  - Utilisateurs : standard POSIX, propre à l'espace mémoire d'un processus => contexte d'exécution (pile)
- Entités différentes de nature distincte
  - Comment les faire correspondre ?

Interaction

- Notion de processeur virtuel (LWP)
  - Interface entre les user et les kernel threads
- Ils sont associés aux processus lourds
- Le nombre de LWPs est indépendant du nombre de threads utilisateurs du processus
- Au moins 1 LWP par processus
- Association possible entre threads utilisateurs et LWP

Exemple



Utilisation des threads

- Un paramètre à l'appel de **`pthread_create`** choisit entre la création d 'un LWP ou d 'un thread user
- set_concurrency fixe le nombre de LWPs du processus
- Quand tous les LWPs d 'un processus sont bloqués, le système en crée un d'autorité
- Le noyau schedule les LWPs et ses threads propres sur les processeurs

Caractéristiques

- Conservation d’une interface applicative portable (POSIX)
- Efficacité des threads
  - Création/destruction
  - Synchronisation
- Mapping sur les threads noyaux
  - Indépendants des threads users

Performances

- Temps de création d'un thread

(Ultra SPARC 1, Solaris 2.5)

	Micro secondes	Ratio
Thread non lie	50	1
Thread lie	350	6.7
Fork()	1700	32.7

Performances (2)

- Temps de synchronisation de threads

(Ultra SPARC 1, Solaris 2.5)

	Micro secondes	Ratio
Threads non lies	60	1
Threads lies	360	6
Entre processus	200	3.3

Ordonnancement

- Ordonnancement à deux niveaux
  - Threads sur les LWPs
  - LWPs sur les processeurs
- Politique « standard » : partage du temps
  - Répartition homogène entre les LWPs (modulo les priorités)
  - Répartition entre les utilisateurs

Classes d ’ordonnanceur

- Possibilité de choisir un type d’ordonnancement par LWP
  - Time-Sharing : priorité = pourcentage de temps processeur alloue
  - Real-Time : priorité = ordre d’exécution
- LWP schedulé par le noyau
- Nouvelle version
  - Répartition entre les utilisateurs priment

Architecture distribuée

- Architectures SMP
  - limitées en taille
    - maximum : une centaine
  - spécialisées
    - liées à des constructeurs : coûteuse
- Développement des réseaux
  - locaux : Fast-Ethernet, Myrinet, …
  - grande échelle : internet
  - prix baissent, débits augmentent
- Utilisation d 'architectures distribuées
  - petite échelle (une salle, un bâtiment)
  - grande échelle (pays, monde) : métacomputing

⇨ Nouveau modèle de programmation

Modèles basés sur l’envoi de messages

Conclusion sur le passage de messages

- Modèle simple
  - efficace sur tout réseau car proche du matériel
  - abstractions manipulées de bas niveau
  - « assembleur » de la programmation parallèle
- Adapté pour des schémas de programmation simples
  - maître/esclave de base
  - SPMD avec protocole de partage rudimentaire (centralisée)
- Inadapté pour des schémas compliqués
  - algorithmique distribuée complexe
  - debugage compliqué (interblocage, non déterminisme)
    - résultat…
    - et performances!

Threads et distribution

- Comment utiliser les threads dans un contexte distribué ?
  - Activités de faible granularité
  - SMP en distribué
  - recouvrement calcul/communication
- Deux approches
  - envoi de messages
    - les threads communiquent directement entre eux
      - problème de la désignation
      - algorithmique distribuée compliquée
  - appel de procédure à distance
    - idem au RPC (Remote Procedure Call)
    - demande d 'exécution d 'un **service** distant
    - création d 'un processus pour exécuter l 'appel
    - à la sauce JAVA : RMI

Modèles basés sur le RPC

Pourquoi le RPC ?

- Constat
  - possibilité de décomposer une application en tâches indépendantes (≠processus)
  - passage de données au début de l'exécution, collecte des résultats à la fin
  - ⇨ comportement typique d 'un RPC
- Mais…
  - les tâches sont souvent de faible durée
  - si ce sont des processus Unix
    - ⇨ coût gestion (création, destruction, ordonnancement) > coût du travail
- Solution
  - les regrouper ⇨ on retombe sur un système à passage de messages
  - utiliser des processus peu coûteux
- Les Processus légers (threads)

Comparaison avec les autres modèles

	MPI	Threads	OpenMP
Portable	√	~=√	√
Scalable	√	√	√
Performance	√		√
Data-parallel	√		√
Haut niveau			√
Proche du séquentiel			√
Prouvable			√

Conclusion

- But de la programmation parallèle
  – efficacité
- Différents paramètres entrent en jeu
  – algorithme parallèle
  – modèle de programmation
  – performances du support d 'exécution
  – équilibrage de charge, ordonnancement
- Modèles à mémoire partagée
  – facile à utiliser
  – performances pas toujours au rendez-vous (sauf matériel spécifique)
- Modèles à passage de message
  – performant sur architectures traditionnelle
  – compliqué à utiliser (peu « naturel » pour un programmeur)

Solution ?

- Modèle émulant une mémoire partagée
  - décomposition en tâches (≠processus)
  - chaque tâche accède à des variables partagées
    - liste des variables
    - type d 'accès (lecture, écriture)
  - les variables définissent des **dépendances** entre les tâches
    - construction d 'un graphe de dépendances
    - utilisation de ce graphe pour décider qui doit posséder la variable
- Caractéristiques
  - programmation facilitée (proche du séquentiel)
  - minimisation des communications ⇒ efficacité en réseaux et en SMP

➔ Athapascan-1

Publié sur Developpez.com avec l'aimable autorisation de Yves Denneulin.

Retrouvez tous les cours Systèmes de Developpez : http://systeme.developpez.com

- Architecture des ordinateurs
- Systèmes d'exploitation
- Systèmes temps réels
- Systèmes embarqués
- Systèmes répartis et middleware
- Parallélisme
- Annuaires
- Sûreté de fonctionnement