PAP 2010 – brouillon du cours

Programmation parallèle = chercher à effectuer plusieurs choses en // au sein d'une même application, pour en réduire le temps d'exécution.

A quoi/à qui ça sert ? (A consulter : https://computing.llnl.gov/tutorials/parallel_comp/)

* Gagner du temps
* Traiter des problèmes importants (dépasser la limitation mémoire d'une machine)
* Faire plusieurs tâches en même temps pour programmer simplement une simulation

Utilisation familiale PC : tirer partie des processeurs actuels nécessite l'utilisation des techniques du parallélisme au sein de plus en plus d'applications (surf sur internet, jeux, traitement image, compression MPEG4, etc.)

Les domaines du Calcul Intensif :

- quoi : simulations (physique, mécanique, chimie, biologie, sismologie, finance)
- pourquoi : enjeux économiques, politiques et militaires

Ex: CEA, Méteo France, EDF, IFP, Dassault, Total, FT, etc.

C'est quoi une architecture parallèle ?

- Parallélisme interne au processeur (pipeline ; unité de calcul ; core)
- Parallélisme interne à une machine (processeurs ou accélérateurs)
- Des machines connectées par un réseau (calculateurs //, grappes)
- Plusieurs centres de calcul connectés à l'échelle d'Internet (grille)
- Dizaines de milliers de pc : le peer 2 peer (seti@home, folding@home)

Plan du Cours

- Intro

- approche mémoire partagée :
  - programmation (pthreads, OpenMP)
  - architectures (processeur, cache, SMP, NUMA, accélérateurs)

- approche mémoire distribuée :
  - programmation (MPI)
  - architectures (cluster, MPP, réseaux rapides)

- compléments :
  - algorithmique distribuée (mémoire virtuellement partagée)
  - approche mixte (MPI + thread, UPC)

Besoins en calcul scientifique

Nécessité de simuler car l'expérimentation peut être

- trop difficile, trop chère, trop lente, trop dangereuse...
- impossible : exemple (simpliste) climatologie

Exemple le calcul de la météo : calculer f(x,y,z,t) -> température, pression, humidité, vent

Le problème se modélise en mécanique des fluides, on peut utiliser un modèle *discret* pour la simulation :

- discrétiser l'espace (5.10^9 points = 1 point pour 2 km3 et 20km d'atmosphère)

- discrétiser le temps (par pas de 60s)
- algo itératif issu de la mécanique des fluides (100 flop pour un point)

Coût d'une simulation : *1 pas de calcul coûte 5.10^11 flop*
temps réel : 5.10^9 x 100 / 60 = 8 Gflop/s
prévision : 7 jours en 24h -> 56 Gflop/s
climatologie : 50 ans en 30 jours -> 4,8 Tflop/s

Performances (double précision)
phenom II quad-core : 48 GFlop/s (4 x 4 FP x 3 GHz) => 39 observés
core i7 quad-core : 51,2 GFlop/s (4 x 4 FP x 3.2 GHz) => 49 observés
=> un multiprocesseur à 20 K€ propose donc quelques centaines GFlop/s

Puissance théorique des accélérateurs
- Geforce GTX 280 : 622 GFlop/s (2 * 240 * 1,296GHz) en simple précision
- Nvidia Fermi : 870GFlop/s (512 * 1,7 GHz)
- cell : 205 GFlops (double précision)

TOP 500 des plus puissantes plateformes de calcul connues

2009	Site	Manu	Computer	Country	Year	Cores	RMax	RPeak
1	Oak Ridge National Laboratory	Cray Inc.	Cray XT5-HE Opteron Six Core 2.6 GHz	United States	2009	224162	1759000	2331000
2	DOE/NNSA/LANL	IBM	PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz,	United States	2009	122400	1042000	1375780
3	University of Tennessee	Cray Inc.	Cray XT5-HE Opteron Six Core 2.6 GHz	United States	2009	98928	831700	1028850
4	Forschungszentrum Juelich (FZJ)	IBM	Blue Gene/P Solution	Germany	2009	294912	825500	1002700
5	National SuperComputer Center in Tianjin/NUDT	NUDT	Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband	China	2009	71680	563100	1206190

2008	Country	Year	Cores	RMax	RPeak
PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz , Infiniband	United States	2008	129600	1105000	1456700
Cray XT5 QC 2.3 GHz	United States	2008	150152	1059000	1381400
SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz	United States	2008	51200	487005	608829
eServer Blue Gene Solution	United States	2007	212992	478200	596378

- folding @ home 4,6 PFlops

janvier 2010

OS Type	Native TFLOPS*	x86 TFLOPS*	Active CPUs	Total CPUs
Windows	226	226	237301	3001105
Mac OS X/PowerPC	4	4	4470	133137
Mac OS X/Intel	23	23	7304	108400
Linux	69	69	40693	455888
ATI GPU	1163	1227	11398	110108
NVIDIA GPU	2026	4275	17028	166765
PLAYSTATION®3	1132	2389	40147	900532
Total	**4643**	**8213**	**358341**	**4875935**

Est-il facile d'obtenir de bonnes perfs ?

- il faut utiliser à fond le processeur
  - 20% en moyenne des perf crête pour une appli quelconque
  - exemple « idéal » produit de matrices simple précision optimisé - efficacité :
    93% sur 1 core 90% sur 4 cores 26% d'un GPU.

- en plus, il faut extraire suffisamment de parallélisme

accélération = speedup = Temps du meilleur prog. séquentiel / Temps du prog. parallèle

Loi d'Amdahl: soit s la part séquentielle de l'application l'accélération est bornée par 1 / (s + (1-s)/#processeurs)

« si 1% de l'application est séquentielle on n'arrivera pas à aller 100 fois plus vite »

speedup max

S \ #P	2	4	8	16	32	64	128	1024	1048576
0,9	1,05	1,08	1,10	1,10	1,11	1,11	1,11	1,11	1,11
0,8	1,11	1,18	1,21	1,23	1,24	1,25	1,25	1,25	1,25
0,7	1,18	1,29	1,36	1,39	1,41	1,42	1,42	1,43	1,43
0,6	1,25	1,43	1,54	1,60	1,63	1,65	1,66	1,67	1,67
0,5	1,33	1,60	1,78	1,88	1,94	1,97	1,98	2,00	2,00
0,4	1,43	1,82	2,11	2,29	2,39	2,44	2,47	2,50	2,50
0,3	1,54	2,11	2,58	2,91	3,11	3,22	3,27	3,33	3,33
0,1	1,82	3,08	4,71	6,40	7,80	8,77	9,34	9,91	10,00
0,05	1,90	3,48	5,93	9,14	12,55	15,42	17,41	19,64	20,00
0,01	1,98	3,88	7,48	13,91	24,43	39,26	56,39	91,18	99,99
0,001	2,00	3,99	7,94	15,76	31,04	60,21	113,58	506,18	999,05

Efficacité max = speedup max / p

S \ #P	2	4	8	16	32	64	128	1024	1048576
0,9	0,53	0,27	0,14	0,07	0,03	0,02	0,01	0,00	0,00
0,8	0,56	0,29	0,15	0,08	0,04	0,02	0,01	0,00	0,00
0,7	0,59	0,32	0,17	0,09	0,04	0,02	0,01	0,00	0,00
0,6	0,63	0,36	0,19	0,10	0,05	0,03	0,01	0,00	0,00
0,5	0,67	0,40	0,22	0,12	0,06	0,03	0,02	0,00	0,00
0,4	0,71	0,45	0,26	0,14	0,07	0,04	0,02	0,00	0,00
0,3	0,77	0,53	0,32	0,18	0,10	0,05	0,03	0,00	0,00
0,1	0,91	0,77	0,59	0,40	0,24	0,14	0,07	0,01	0,00
0,05	0,95	0,87	0,74	0,57	0,39	0,24	0,14	0,02	0,00
0,01	0,99	0,97	0,93	0,87	0,76	0,61	0,44	0,09	0,00
0,001	1,00	1,00	0,99	0,99	0,97	0,94	0,89	0,49	0,00

En plus il faut considérer le coût de la parallélisation : synchronisation nécessaire (protection des données partagées, synchronisation des calculs), communication entre les processeurs, calcul redondant, démarrage et terminaison des flots de calculs,.

Un pb supplémentaire est celui de l'uniformité de la répartition du travail : il faut chercher à équilibrer la charge de travail entre les processeurs.

=> Problèmes réguliers : calcul prévisible ne dépendant pas de la valeur des résultats intermédiaires (multiplication de matrice), on peut « tout » arranger à l'avance.

=> Problèmes irréguliers (ex. approximation d'une surface à partir de points, ou encore calcul fractal), impossible de distribuer à priori (à la compilation ou au départ de l'exécution) de façon équitable le travail, il faut le faire de façon dynamique en cours d'exécution.

Approche mémoire partagée

Programmation orientée threads https://computing.llnl.gov/tutorials/pthreads/

Objectif du chapitre :
* voir comment on peut distribuer une boucle for,
* faire un rappel sur les threads, s'amuser avec les barrières
* coût de la parallélisation

La programmation multi-threadée est le paradigme de programmation des machines à mémoire commune. Il existe d'autre paradigme (processus ou encore exploitation d'un graphe de tâches, d'arbre de calculs), mais à la fin on retrouve souvent les threads au niveau du support d'exécution. De plus les processeurs actuels sont définis pour être utilisé par des threads.

Rappels : thread = unité d'exécution au sein d'un processus détient
* ses propres registres dont le pointeur de pile et le PC,
* le masque des signaux, ses signaux pendants
* son mode d'ordonnancement (priorité)
* données spécifiques (errno)

Nouvelle difficulté : gestion de la synchronisation pour l'accès aux données partagées
-> Utilisation de mutex, verrou, condition (= liste d'attente pour un mutex donné)
Avantages par rapport aux processus : flexibilité, faible coût relatif des opérations de base, faible consommation mémoire.

Calcul en // de Y[i] = f(T,i)
approche statique : on distribue les indices au moment de la création des threads
approche dynamique : on distribue les indices au fur et à mesure

```
#define NB_ELEM (TAILLE_TRANCHE * NB_THREADS)
pthread_t threads[NB_THREADS];
double input[NB_ELEM], output[NB_ELEM];

void appliquer_f(void *i)
{
 int debut = (int) i * TAILLE_TRANCHE;
 int fin = ((int) i+1) * TAILLE_TRANCHE;

 for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

//  pthread_exit(NULL);
}

int main()
{
…
  for (int i = 0; i  < NB_THREADS; i++)
   pthread_create(&threads[I], NULL, appliquer_f, (void *)i);

 for (int i = 0; i  < NB_THREADS; i++)
   pthread_join(threads[I], NULL);

 ...
}
```

La parallélisation est efficace lorsque le calcul est équilibré... mais imaginons qu'un thread ait la moitié du travail alors le *speedup* serait limité à 2.

⇨ approche dynamique pour diminuer le risque de déséquilibre

```
pthread_mutex_t mutex;
int indice = 0;

int
obtenir_indice()
{
  int k;
  pthread_mutex_lock(&mutex);
  k = indice++;
  pthread_mutex_unlock(&mutex);
  return (indice > NB_ELEM) ? -1 : indice;
}

void appliquer_f(void *i)
{

 for( int n= obtenir_indice(); n > 0; n=obtenir_indice())
  output[n] = f(input,n);
}

int
main()
{
  pthread_mutex_init(&mutex, NULL);
  ...

  for (int i = 0; i  < NB_THREADS; i++)
   pthread_create(&threads[I], NULL, appliquer_f, (void *)i);
 ...
  pthread_mutex_destroy(&mutex);

}
```

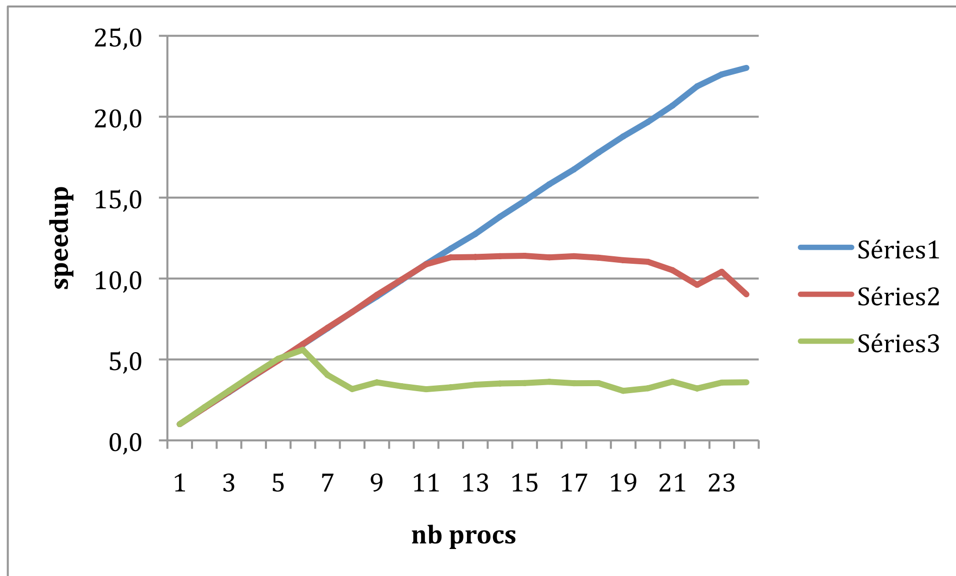
Application (Examen 2008)

Il s'agit de paralléliser le plus efficacement possible la boucle suivante (en modifiant au besoin le code):

```
for(i=0 ; i < 1000 ; i++)
   s += f(i) ;
```

1. *En supposant que le temps de calcul de f(i) ne dépends pas de la valeur de i ;*
2. *En supposant que le temps de calcul de f(i+1) est toujours (très) supérieur à celui de f(i).*

Commentaires : l'équilibrage de charge est un problème difficile qui apparaît en particulier dans les problèmes *irréguliers* où la complexité du traitement est plus liée à la valeur des données qu'à leur structuration (exemple type : décomposition en facteurs premiers d'un ensemble de nombre). L'approche dynamique permet de limiter les risques, cependant on peut toujours tirer le mauvais numéro en dernier... un défaut majeur de l'approche dynamique est de reposer sur des mécanismes de synchronisation, typiquement des mutex. Lorsque les threads accèdent fréquemment au mutex il se peut qu'un embouteillage se crée, on parle alors de *contention*, les threads se bloquent et la part de programme séquentiel augmente indirectement. Il faut faire la chasse au mutex pour favoriser le parallélisme !



Speed-up obtenus avec un programme trivial comprenant une section critique représentant 5% 10% et 20% du temps de calcul sur une machine à 24 processeurs.

Une solution intermédiaire est de régler la taille des tranches d'indices à distribuer => on s'attaque ainsi à la *granularité* du calcul. Une autre façon de procéder est de rééquilibrer dynamiquement le travail en utilisant des techniques de vol : par exemple un thread inactif peut aller *voler* des indices à un autre thread.
Une technique radicalement différente est d'augmenter le nombre de threads (en créer n fois plus que de coeurs) et laisser le système d'exploitation répartir la charge de travail et d'utiliser un ordonnanceur *préemptif*. On perd cependant du contrôle sur l'application et on augmente la charge du système (et les défauts de cache). En général les utilisateurs, même avancés, n'aiment pas cela, c'est pour eux un signe qu'ils ne maîtrisent pas vraiment le problème.

Nouveau Problème on veut calculer f^k... on ne va pas gâcher du temps à créer nthreads à chaque tours. Il s'agit de synchroniser tous les threads au même endroit dans le code... on appelle cela une « barrière »

```
barrier_t b;

void appliquer_f(void *i)
{
 int debut = (int) i * NB_THREADS;
 int fin = ((int) i+1) * NB_THREADS;

double *entree = input;
double *sortie = output;
double *tmp;

for( etape=0; etape < k; etape++)
{
 for( int n= debut; n < fin; n++)
  sortie[n] = f(entree,n);
  echanger(entree,sortie);
  barrier_wait(&b); // attendre
}

  pthread_exit(NULL);
}
```

Implémentation d'une barrière

Les étudiants ayant suivi l'UE Système savent comment réaliser une barrière à l'aide de condition, sémaphore et moniteur de Hoare… pour les autres voici un exemple :

```
typedef struct
{
  pthread_cond_t condition;
  pthread_mutex_t mutex;
  int attendus;
  int arrives;
} barrier ;

int
barrier_wait(barrier *b)
{
int val = 0;
pthread_mutex_lock(&b->mutex);

b->arrives++;

if ( b->arrives != b->attendus)
  pthread_cond_wait(&b->condition, &b->mutex) ;
else
 {
   val=1;
   b->arrives = 0;
   pthread_cond_broadcast(b->condition);
 }

pthread_mutex_unlock(&b->mutex);
return val;
}
```

Pour obtenir des performances il est fondamental de minimiser les parties séquentielles (Amdahl) donc de minimiser la synchronisation entre les différents threads. Il faut faire la chasse aux mutex et aux barrières ou tout au moins limiter le plus possible leur effet de « séquencement ».

Imaginons qu'on traite un programme comme le "jeu de la vie"... ou si vous préférez l'étude de la dissipation thermique sur une surface (équation de Laplace)… bref un truc du genre :

```
int T[2][DIM][DIM];

for(etape = 0; etape < ETAPE; etape++)
{
        in = 1-in;
        out = 1 - out;
        nb_cellules = 0;
     for(i=1; i < DIM-1; i++)
                for(j=1; j < DIM-1; i++)
                {
                  T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
                                 if (T[out][i][j] > 0)
                    nb_cellules++;
                }
       printf("%d => %d", etape, nb_cellules);
}
```

On parallélise en utilisant nb_threads suivant la première dimension.

```
void calculer(void *id)
{
   int mon_ordre = (int) id;
   int etape, in = 0, out = 1 ;

   int debut = id * …
   int fin = (id +1) * …
  for (etape=0 ...)
  {
  int mes_cellules = 0;
   for(i = debut ; i <  fin ; i++)
       …
       mes_cellules++ ;
       …

   pthread_mutex_lock(&mutex_cell);
   nb_cellules += mes_cellules;
   pthread_mutex_unlock(&mutex_cell)
   pthread_barrier_wait(&bar);

   if (mon_ordre == 0)
    {
     printf(...);
     nb_cellules = 0;
    }
   pthread_barrier_wait(&bar);
  }
```

Le défaut de ce code est qu'on synchronise beaucoup les threads… une optimisation est de commencer par le calcul du *pourtour* de la région considérée puis de signaler la fin de ce calcul aux voisins puis de passer au calcul *interne* et enfin d'attendre que les autres threads aient signalé la fin des calcul externes pour recommencer.

```
int pthread_barrier_wait_begin(barrier_t *bar);
int pthread_barrier_wait_end(barrier_t *bar);
```

```
calculer_bordure(mon_ordre, in, out);
pthread_barrier_wait_begin(&bar);
calculer_centre(mon_ordre, in, out);
pthread_mutex_lock(&mutex_cell);
nb_cellules += mes_cellules;
pthread_mutex_unlock(&mutex_cell)

if( pthread_barrier_wait_end(&bar) == 0) // dernier thread a avoir franchi la barrière
{
  pthread_mutex_lock(&mutex_cell);
              printf(nb_cellules);
              pthread_mutex_unlock(&mutex_cell) ;
}
```

Pb le nb_de cellules vivantes n'est plus toujours le bon => utiliser nb_cellules[out]

Implémentation de la barrière en deux temps
Le piège se situe dans la différence de vitesse entre les threads. Une restriction naturelle est qu'un thread ne peut franchir la première barrière que si tous les autres threads ont effectivement franchi la seconde.

```
typedef struct
{
  pthread_cond_t conditionB;
  pthread_cond_t conditionE;
  pthread_mutex_t mutex;
  int attendus;
  int leftB;
  int leftE;
} barrier ;

void pthread_barrier_init(barrier_t *b, int attendus)
{
...
 b->leftB = attendus;
 b->leftE = 0;
}

int pthread_barrier_wait_begin(barrier_t *b)
{
 int ret = 0;
 pthread_mutex_lock(&b->mutex);

if (b->leftE)
  pthread_cond_wait(&b->conditionB, &b->mutex) ;

 ret = --b->leftB;

 if (ret == 0)
 {
   b->leftE = b->attendu;
  pthread_cond_broadcast(b->conditionE);
 }
pthread_mutex_unlock(&b->mutex);
return ret;
}

int pthread_barrier_wait_end(barrier_t *b)
{
int ret;
pthread_mutex_lock(&b->mutex);

if (b->leftB)
  pthread_cond_wait(&b->conditionE, &b->mutex) ;

ret = --b->leftE;

if(b->letfE == 0)
 {
  b->leftB = = b->attendu;
  pthread_cond_broadcast(b->conditionB);
 }
pthread_mutex_unlock(&b->mutex);
return ret;
}
```

Remarques : On peut désynchroniser encore plus en mettant une barrière par frontière.
On peut surtout réduire le nombre de barrière en faisant se chevaucher les zones traitées par chaque threads en se basant sur le fait qu’on peut calculer l’état d’une cellule à l’étape i + k si

on connaît l’étape i son état et l’état de ses voisins à distance k. Ainsi deux threads 0 ont en commun 2.k lignes alors il suffit de les synchroniser que toutes les k + 1 étapes.
Pour s’en convaincre on peut marquer dans un tableau l’identité des threads qui possèdent la véritable valeur de la cellule en fonction de l’étape ; par exemple au départ le thread 0 possède la valeur des cellules de 0 à 15 et le thread 1 les cellules 6 à 19, 10 cellules sont donc partagées (k = 5) :

Etape\cellule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	1	1	1	1

Un thread peut calculer l’état à l’étape i + 1 de toute cellule dont il connaît l’état du voisinage à distance 1 à l’étape i on a donc :

Etape\cellule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	1	1	1	1
1	0	0	0	0	0	0	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	1	1	1	1	1
2	0	0	0	0	0	0	0	**01**	**01**	**01**	**01**	**01**	**01**	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	**01**	**01**	**01**	**01**	1	1	1	1	1	1	1
4	0	0	0	0	0	0	0	0	0	**01**	**01**	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	0	0	0	0	**?**	**?**	1	1	1	1	1	1	1	1

A l’étape 6 on ne peut calculer l’état des cellules 10 et 11 sans échange d’information. On doit synchroniser donc les threads qu’à ce moment là.

Etape\cellule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
4	0	0	0	0	0	0	0	0	0	**01**	**01**	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
Barrier()	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
Copy()	0	0	0	0	0	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	1	1	1	1
6	0	0	0	0	0	0	**01**	**01**	**01**	**01**	**01**	**01**	**01**	**01**	1	1	1	1	1
7	0	0	0	0	0	0	0	**01**	**01**	**01**	**01**	**01**	**01**	1	1	1	1	1	1

On échange du « temps de synchronisation » contre du temps de calcul. Si le volume de communication est identique, le nombre de communications est lui aussi divisé par k. Il s’agit de déterminer le bon compromis.

OpenMP : open Multi Processing

https://computing.llnl.gov/tutorials/openMP/

Ensemble d'annotations pour paralléliser les applications écrites en C ou en Fortran. On utilise donc des directives pour marquer dans le code les portions à exécuter en parallèle ou en section critique.

```
int
main()
{
  printf("bonjour\n");
  printf("au revoir

  return EXIT_SUCCESS;
}

#include <omp.h>

int
main()
{
#pragma omp parallel
  printf("bonjour\n");

  printf("au revoir\n");
  return EXIT_SUCCESS;
}
```

```
> gcc -fopenmp -lgomp bonjour.c
> ./a.out
bonjour
bonjour
bonjour
bonjour
au revoir
```

=> par défaut la portion parallèle sera exécutée par autant de threads qu'il y a de cœurs...

On peut demander la modification de ce nombre :

```
> export OMP_NUM_THREADS=4 ; ./fichier-executable
> OMP_NUM_THREADS=4 ./fichier-executable
  #pragma omp parallel num_threads(4)
  omp_set_num_threads (4)
```

Cependant le nombre de threads réellement utilisés est dépendant de l'implémentation OpenMP.

OpenMP c'est principalement le parallélisme de type fork-join :

- Au départ, un unique thread exécute séquentiellement de code de *main*.
- Lors de la rencontre d'un bloc parallèle, tout thread crée une *équipe* de threads et la charge d'exécuter une fonction correspondant au bloc parallèle puis rejoint en tant que maitre l'équipe.
- À la fin du bloc les threads d'une même équipe s'attendent au moyen d'une barrière (implicite) ; les threads esclaves sont démobilisés, le thread maître retrouve son équipe précédente.

Objectif du jour : paralléliser le jeu de la vie

```
int T[2][DIM][DIM];
for(etape = 0; etape < ETAPE; etape++)
{
  in = 1-in;
  out = 1 - out;
  nb_cellules = 0;

    for(i=1; i < DIM-1; i++)
        for(j=1; j < DIM-1; i++)
        {
            T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
            if (T[out][i][j] > 0)
                nb_cellules++;
        }
    printf("%d => %d", etape, nb_cellules);
}
```

Un premier point fort d'OpenMP est de fournir un moyen simple de distribuer à une équipe de threads les indices d'une boucle

```
 #pragma omp parallel
{
  #pragma omp for
  for(i; i < DIM; i++)
   {....}
 // ici il y a une barrière implicite
}
```

```
#pragma omp parallel for [schedule(politique de distribution [, taille d'une tranche])]
```

Politiques de distribution d'indices :
static => "block cyclic" par tranche de la taille donnée (par défaut on coupe en parts égales)
dynamic => get_index() par tranche de la taille donnée (par défaut 1)
guided => get_index (nb_indice restant)/ nb threads (la taille est un min)
runtime on utilise la variable d'environnement OMP_SCHEDULE
auto c'est le runtime qui décide

static est adapté au calcul régulier (facilite les optimisations, diminue la contention) on utilise dynamic dans les autres cas.

```
#pragma omp for  schedule(static)
```

Pour l'accès concurrent à nb_cellule++ on peut utiliser soit une section critique soit une opération atomique (si disponible le jeu d'instruction du processeur)

```
#pragma omp critical NOM_DE_LA_SECTION
nb_cellule++;
```

=> traduit par un mutex... attention s'il n'y a pas de nom on utilise <u>*le*</u> mutex par défaut

```
#pragma omp atomic
nb_cellules++
```

=> permet d'utiliser les opérations arithmétiques simples de façon atomique, le processeur empêche « physiquement » l'accès à la donnée (en verrouillant le bus par exemple).

Mais on a déjà vu mieux : définir une variable locale et puis faire la synthèse à la fin. Cela se fait de façon automatique au travers d'une *réduction*.

```
#pragma omp parallel for reduction(+:nb_cellules)
```

On peut faire plusieurs opérations de réduction à la fois.

Il est possible d'exprimer qu'un bloc doit être réalisé une seule fois par le master seul

```
 #pragma omp single
```

il y a une barrière implicite à la suite du single contrairement à la directive master :

```
#pragma omp master
```

une barrière explicite

```
#pragma omp barrier
```

On peut faire sauter les barrières implicites à la fin des blocs en mettant une clause nowait

Variables partagées / privées

Pour tout bloc parallèle, il est possible de préciser le caractère partagé ou privé de chaque variable déclarée à l'extérieur du bloc ; un choix par défaut est possible. Notons que les tableaux sont toujours partagés. Il est aussi possible de déclarer des variables globales privées (*spécifiques*) aux threads.

int x;
#pragma omp threadprivate(x)

```
#pragma omp  parallel num_threads(n)  default(shared)
{

int etape, j;
for(etape=0; etape < N; etape++)
{
   #pragma omp for  schedule(static) reduction(+:nb_cellules)
   for(i=1; i < DIM-1; i++)
                for(j=1; j < DIM-1; J++)
                {
                  T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
                            if (T[out][i][j] > 0)
                                  nb_cellules++;
                }
   //barrier implicite
   #pragma omp master
   {
      printf("%d => %d", etape, nb_cellules);
    {  nb_cellules = 0; in = 1-in; out = 1 - out;
}
}
```

Parallélisme imbriqué

Dans l'inconscient collectif des programmeurs OpenMP le modèle de programmation c'est *un thread par processeur*. De ce point de vue, le parallélisme imbriqué sert à déterminer des zones de travail auxquelles on va associer une équipe de threads.

```
omp_set_nested(1);

#omp parallel num_threads(externe)
{

// distributions des zones de travail

#omp parallel for num_threads(interne)
  {
  // parallélisation du travail au sein d'une zone
   }

}
```

Avec cette technique, on exploite le parallélisme imbriqué sur 2 ou 3 niveaux de telle façon à ce que l'expert OpenMP puisse réaliser algorithmiquement l'équilibrage de charge (distribution statique des régions découpées de façon adhoc)… cette tactique n'est possible que sur les pb à peu près réguliers. On peut vouloir utiliser plus de threads pour faire de l'équilibrage de charge mais c'est contre les habitudes de la communauté parce qu'au début (et même encore maintenant) les environnements n'étaient pas au point (le placement des équipes de threads n'est pas optimisé).

Outer × Inner	LIBGOMP3	INTEL	Cache	Cache + load info	Tuned Cache + load info
4×4	9.4	13.8	14.1	14.1	14.1
16×1	**14.1**	**13.9**	14.1	14.1	14.1
16×2	11.8	9.2	14.1	14.2	14.3
16×4	11.6	6.1	14.1	14.9	14.9
16×8	11.5	4.0	14.4	**15.0**	**15.2**
32×1	12.6	10.3	13.5	13.8	13.8
32×2	11.6	5.9	14.2	14.2	14.3
32×4	11.2	3.4	14.3	14.8	14.8
32×8	10.9	2.8	**14.5**	14.7	14.7

BT-MZ : calcul en dynamique des fluides – 256 zones, quantité de travail déséquilibrée (rapport 25 entre les extrêmes).

Parallélisme externe = nombre de paquets de zones
Parallélisme interne = nombre de threads travaillant zone apres zone dans un paquet

Beaucoup de paquets => déséquilibre entre les paquets => il faut beaucoup de thread par paquet pour récupérer la zone…
Peu de paquets et beaucoup de threads => dispersion des threads sur toute la machine

PB : comment déterminer le bon nombre de threads ?

Section //

```
#pragma omp sections
{
  #pragma omp section
     {
      ...
```

```
      }
      #pragma omp section
      {
      ...
      }
}
```

Les sections sont distribuées de façon dynamique au threads, à l'image d'une distribution d'indice.

Parallélisme à grain fin

Idée : avoir (beaucoup) plus de flots parallèles que de coeurs

→ Utilisation de la récursivité

```
void fun ( int p)
{
  #pragma omp parallel for if (p < PROFMAX)
  for(i …)
            …
            fun(p+1) ;

}
```

→ Utilisation de tâches (OpenMP-3.0) pour éviter le surcoût de gestion de trop nombreux threads

```
#pragma omp parallel // crée une file de tâches pour l'équipe
{
  …
  #pragma omp task // initier une tâche, créer une file pour ses sous taches
  {
            …
  }
  …
  #omp taskwait // attendre la fin de toutes les tâches créées dans le bloc
  …
}
```

Exemple fibonacci

```
int fib ( int n )
{
  int x,y;
  if ( n < 2 ) return n;

  #pragma omp task shared(x) firstprivate(n)
  x = fib(n-1);

  #pragma omp task shared(y) firstprivate(n)
  y = fib(n-2);

#pragma omp taskwait
return x+y;
}
```

Exemple liste

```
#pragma omp parallel
#pragma omp single private(e)
{
for ( e = l->first; e ; e = e->next )
  #pragma omp task
  process(e);
}
```

Une tâche initiée peut être exécutée par tout thread de l'équipe du thread initiateur.
Un thread peut prendre / reprendre en charge une tâche :

A la création d'une tâche
A la terminaison d'une tâche
A la rencontre d'une barrière (taskwait)

Un thread ayant pris en charge l'exécution d'une tâche doit la terminer à moins que la clause *untied* ait été spécifiée.

Exemples tirés de http://wikis.sun.com/display/openmp

QSORT

```
void quick_sort (int p, int r, float *data)
{
   if (p < r) {
         int q = partition (p, r, data);
         #pragma omp parallel sections firstprivate(data, p, q, r)
         {
            #pragma omp section
            quick_sort (p, q-1, data, low_limit);
            #pragma omp section
            quick_sort (q+1, r, data, low_limit);
         }
      }
   }
}

void par_quick_sort (int n, float *data)
{
   quick_sort (0, n, data);
}

// Using an OpenMP 3.0 Task
void quick_sort (int p, int r, float *data)
{
   if (p < r) {
         int q = partition (p, r, data);
         #pragma omp task
         quick_sort (p, q-1, data, low_limit);
         #pragma omp task
         quick_sort (q+1, r, data, low_limit);
      }
   }
}

void par_quick_sort (int n, float *data)
{
   #pragma omp parallel
   {
      #pragma omp single nowait
      quick_sort (0, n, data);
   }
}
```

NUM_THREADS	task	Sect.
2	2.6s	1.8s
4	1.7s	2.1s
8	1.2s	2.6s

Count_good : bricolage pour faire une réduction avec les tâches

```
int count_good (item_t *item)
{
  int n = 0;
  while (item) {
     if (is_good(item))
        n ++;
     item = item->next;
  }
  return n;
}
```

Count_good : Atomique vs somme partielle

```
int count_good (item_t *item)
{
  int n = 0;
  #pragma omp parallel
  {
    #pragma omp single nowait
    {
       while (item) {
           #pragma omp task firstprivate(item)
           {
              if (is_good(item)) {
                #pragma omp atomic
                n ++;
              }
           }
           item = item->next;
       }
    }
  }
  return n;
}
```

```
int count_good (item_t *item)
{
  int n = 0;
  int pn[P]; /* P is the number of threads used. */
  #pragma omp parallel
  {
    pn[omp_get_thread_num()] = 0;
    #pragma omp single nowait
    {
       while (item) {
           #pragma omp task firstprivate(item)
           {
              if (is_good(item)) {
                pn[omp_get_thread_num()] ++;
              }
           }
           item = item->next;
       }
    }
    #pragma omp atomic
    n += pn[omp_get_thread_num()];
  }
  return n;
}
```

Foo : du bricolage pour fabriquer un pseudo groupe de tâches

```
int foo ()
{
   int a, b, c, x, y;

   a = A();
   b = B();
   c = C();
   x = f1(b, c);
   y = f2(a, x);

   return y;
}

avec pour dépendances

    A ------------------+
    B ------+            |--> f2
            |--> f1 ---+
    C ------+
```

```
void foo ()
{
   int a, b, c, x, y;

   #pragma omp task shared(a)
   a = A();

   #pragma omp task if (0) shared (b, c, x)
   {
      #pragma omp task shared(b)
      b = B();

      #pragma omp task shared(c)
      c = C();

      #pragma omp taskwait
   }
   x = f1 (b, c);

   #pragma omp taskwait

   y = f2 (a, x);
}
```

Il existe plusieurs façons de répartir les tâches entre les threads d'une équipe
une file globale (contention ?) *vs* des files locales (équilibrage de charge ?)

Pour équilibrer les files locales, une technique efficace est d'utiliser un vol de travail à la *Cilk* (Leiserson & al) via l'utilisation de DEQUE (double end queue) tout thread utilise sa DEQUE comme une pile pour travailler et lorsque sa pile est vide il vole du travail à un autre voisin en utilisant la DEQUE comme une FIFO.

=> exécution en profondeur d'abord et vol en largeur d'abord.
Favorise la localité du calcul, minimise à priori le nombre de vols.

Intel TBB thread building blocks : cocktail C++ à base de Cilk et d'openMP conçu pour exploiter les processeurs multicore :
De Cilk : vol de travail + notion de profondeur paramétrable
D'OpenMP : distribution de boucle via découpage en 2 de l'intervalle

Exemple : tâche tbb sur TSP

```
task * execute() {

 if(this->is_stolen_task()) {
   was_stolen = 1;
   int ptr[anz];
   memcpy(ptr, new_path, (hops-1) * sizeof(int));
   new_path = ptr;
 }

 if (length >= minimum)
   return;

 new_path[hops-1]=current;

 if(hops == n) {
   if (length < minimum) {
     MutexType::scoped_lock lock(Mutex);
       if(length < minimum)
         minimum = length;
   }
 }
 else {
   int count = n-hops;
   set_ref_count(1+count);
   int j = 0;

   for (int i=1; i < n; i++) {
     if (!present(i, hops, new_path)) {
     j++;

     int dist = distanceArray[current+i*30];
     int tmp = length + dist;

     TSPTask& a  = *new(task::allocate_child() ) TSPTask(tmp,hops+1,new_path,i);
     j < count ? spawn(a) : spawn_and_wait_for_all(a);
   }
 }
```

Niveau	Tasks	davon g	in %
1	15	14	93
2	210	50	24
3	2730	41	1,5
4	32760	63	0,2
5	356213	117	~0
6	3109590	146	~0
7	18182241	91	~0
8	64528584	62	~0
9	133033964	24	~0
10	153320304	7	~0
11	96375125	6	~0
12	32810436	0	-
13	5968656	0	-
14	552590	0	-
15	17144	0	-

Version	15 Städte	16 St
sequentiell	5,75 s	30,68
task + stolen	1,66 s	9,65 s
Speedup	3,46	3,18
Effizienz	0,43	0,40

Version	15 Städte	16 Städte
Sequentiell	5,75 s	30,68 s
task + cut	0,58 s	3,26 s
Speedup	9,91	9,41

CONCLUSION
Souvent OpenMP est vendu ou compris comme un outil de parallélisation de programmes séquentiels… du coup on entend des *profanes* dirent « *OpenMP c'est facile, mais c'est pas performant, ça marche avec 2 ou 4 processeurs mais au-delà faut prendre MPI…* ». Certes OpenMP permet de réaliser des programmes parallèles assez facilement et de paralléliser progressivement une application. Cependant paralléliser brutalement un programme séquentiel ne permet généralement pas d'obtenir de bons speedups… Pour correctement paralléliser un programme il s'agit de :

- ⇨ connaître des techniques de bases d'équilibrage de charge et de minimisation des synchronisations
- ⇨ connaître un minimum d'architecture parallèle pour ne pas faire de contre-emploi (provoquer bêtement des congestions du « bus » mémoire, utiliser les caches)
- ⇨ revoir les algorithme en conséquence afin de maximiser le parallélisme (introduction / suppression de calcul redondant)

Malheureusement, les performances viennent rarement gratuitement et OpenMP doit être avant tout compris comme un outil qui permet programme d'obtenir un programme séquentiel à partir d'un code parallèle.

Architecture mémoire partagée

- *Computer architecture: a quantitative approach* par John L. Hennessy,David A. Patterson, Andrea C. Arpaci-Dusseau
- Cours de François Pelegrini
  http://uuu.enseirb.fr/~pelegrin/enseignement/enseirb/archsys/cours/c.pdf

Plan :

- parallélisme interne au processeur
- Technologie des caches
- Cohérence mémoire dans les multiprocesseurs symétriques
- Machine NUMA

Parallélisme au sein du processeur

Deux types de parallélisme au sein d'un microprocesseur sont à distinguer le parallélisme d'instruction (ILP) et celui de flot (TLP) :

- Instruction Level Parallelism (ILP) vise pour un thread donné à exécuter les instructions de celui-ci en minimum de temps (on privilégie la latence).
- Thread Level Parallelism (TLP) vise à occuper le plus possible toutes les unités fonctionnelles du processeur (on privilégie le débit).

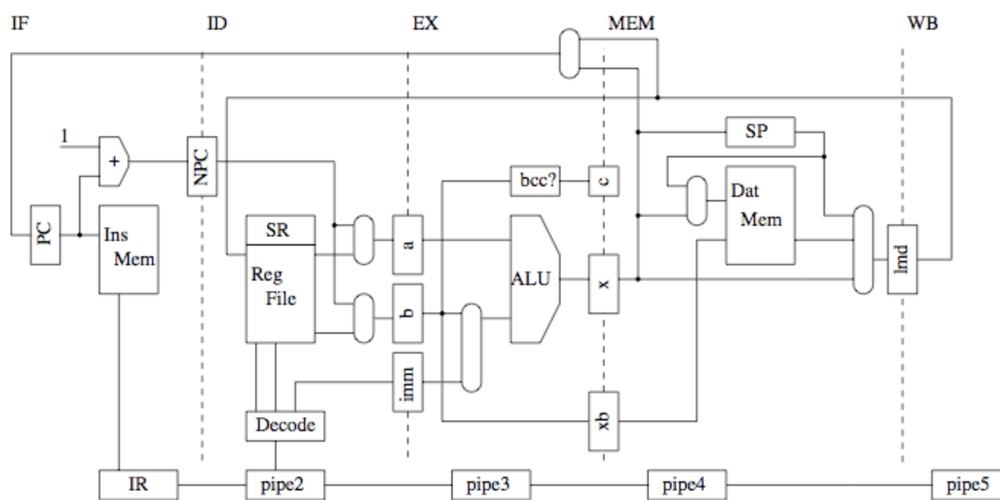
Approche ILP

La technologie du pipeline – le travail à la chaine

Voir http://www.hardware.fr/articles/623-1/intel-core-2-duo-dossier.html

L'exécution d'une instruction peut être découpée en plusieurs phases, par exemple :

Fetch : lire l'instruction	Fetch
Decode : l'analyser	Decode
Read : chercher les arguments	Execute
Execute : Réaliser l'opération	Memory
Write : écrire le résultat	WriteBack

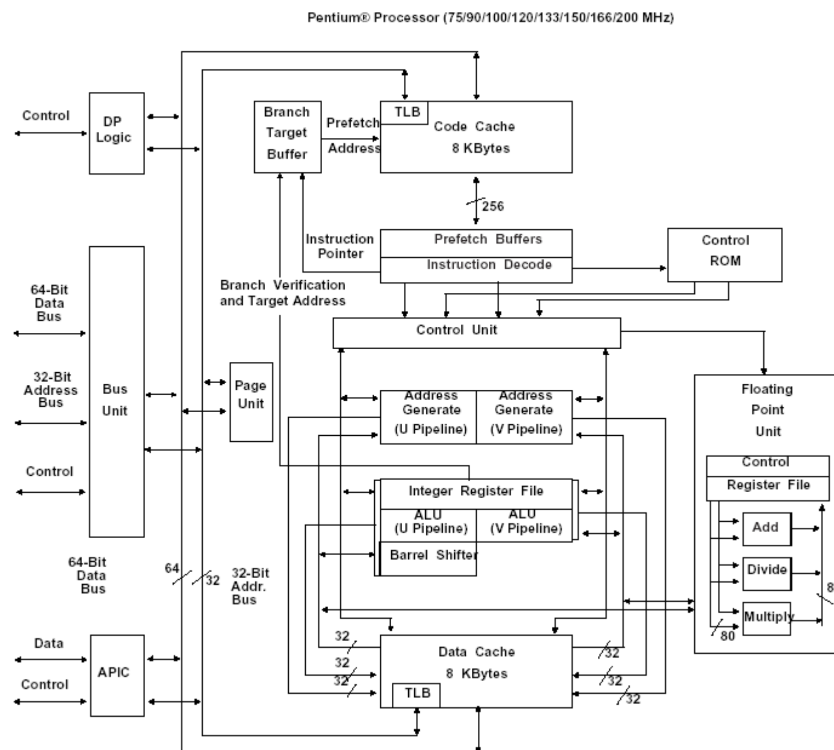


Pipeline du processeur MIPS de type RISC

Chaque phase peut être réalisée par un circuit spécialisé, par une unité fonctionnelle dédiée. L'idée du pipeline est de faire en sorte que les unités fonctionnelles puissent travailler en parallèle sur des instructions différentes (issues d'un même flux). Idéalement l'instruction progresse d'étage en étage à chaque top horloge. Ainsi un pipeline à 5 étages (on dit aussi de profondeur ou de longueur 5) peut exécuter jusqu'à 5 instructions en parallèle ce qui permet d'espérer une accélération approchant 5. Il est donc intéressant d'avoir des pipelines assez profonds :

386, 486, Pentium : 5 étages
PII, PIII : 12 étages
P IV : 20, 31 (architecture netburst)

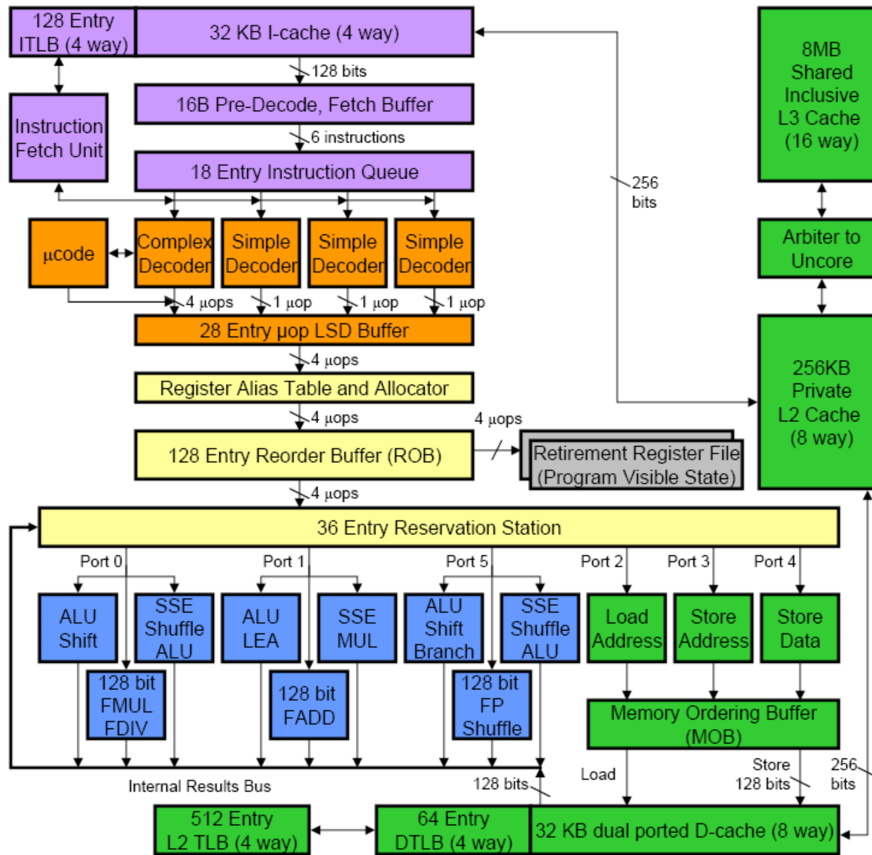
C'est d'autant plus intéressant que la période de l'horloge doit être supérieure au temps de traitement de l'étage le plus lent. Plus on coupe l'exécution des instructions en fines tranches plus on peut augmenter la fréquence de l'horloge... Cependant certaines unités de traitement sont difficiles à découper. Pour contourner ce problème on peut mettre plusieurs unités fonctionnelles complexes en parallèle. Ainsi sur nos processeurs on a plusieurs unités de calculs (flottants ou entier), on appelle cela une *architecture super-scalaire***.**

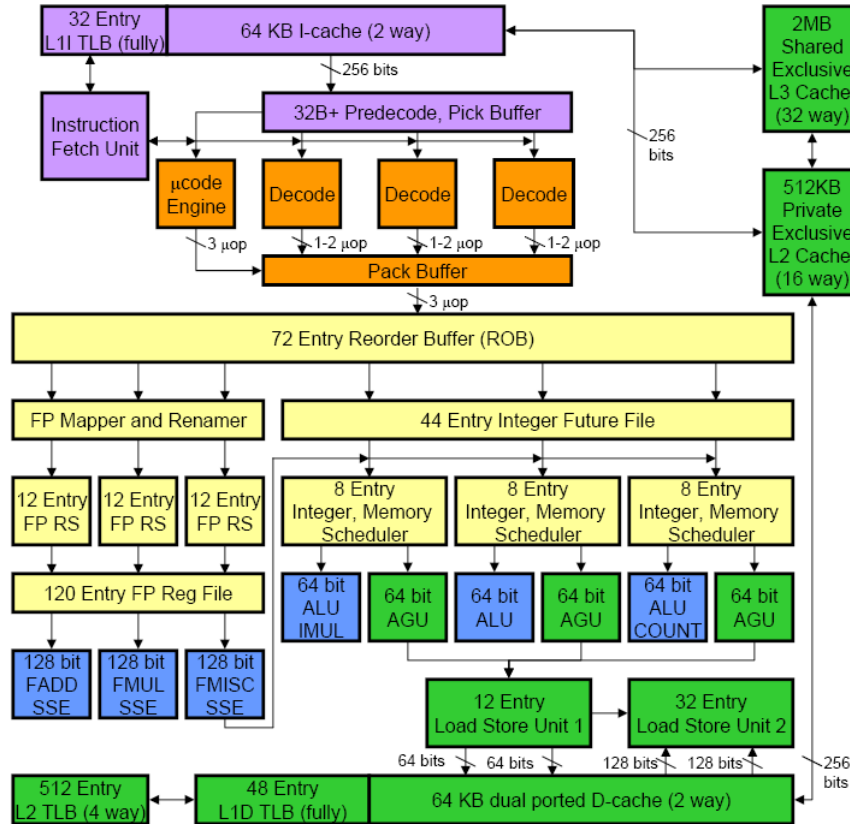


Le pentium dispose de 2 ALU : U et V

On dispose aussi de plusieurs unités de décodage... de chargement... ainsi on multiplie les unités de traitement en parallèle => architecture multi-pipeline. C'est cette recette magique qui a permis à Intel d'atteindre les 4 GHz et les 6 GHz pour IBM. Intel avait prévu un PIV à 45 étages... avant de redescendre à une douzaine avec l'architecture core 2. Pourquoi n'ont-ils pas augmenter la profondeur du pipeline ?

- dissipation thermique,
- latence induite par les circuits complémentaires,
- difficultés pour approcher le rendement idéal.





au CREMI Nehalem = infini & Barcelona = fridulva

Optimisation du rendement du pipeline
Augmenter la profondeur des pipelines accroit inévitablement les problèmes de dépendances de calcul :

```
ADD R1, R1, 1
MUL R2, R2, R1
```

Il faut attendre que la première instruction ait terminée son écriture avant que la seconde puisse faire sa lecture... de telles *dépendances de données* introduisent des "*bulles*" dans le pipeline. Des bulles peuvent apparaître aussi lorsqu'on ne connaît pas assez tôt l'adresse des instructions à exécuter on parle alors de *dépendance de contrôle* (saut conditionnel, pointeur de fonction).

⇨ plus le pipeline est long, plus il y a de bulles et plus les bulles sont plus longues.

Pour optimiser le rendement du pipeline les électroniciens ont inventé diverses techniques pour limiter le nombre et l'impact des bulles. Ces techniques peuvent être mise en œuvre matériellement mais aussi de façon logicielle (options de compilation).

Techniques d'optimisation basées sur le ré-ordonnancement des instructions
L'ordre des instructions peut être changé par le compilateur, mais aussi par le processeur, dynamiquement (OoO : out of order). Dans les architectures multi-pipelines les micro-instructions peuvent se doubler les unes, les autres…
Ainsi supposons que x = y = 0 et qu'on exécute le programme « x = 2; y = 4; » A priori les états observables, notés (x,y), sont (0,0) (2,0) (2,4) mais avec la technologie OoO, il se peut que (0,4) soit aussi observé. Heureusement on dispose d'instructions (niveau assembleur) pour interdire certains ré-ordonnancements : on les appelle les *barrières mémoire*.

```
#define memory_barrier() __asm__ __volatile__("mfence":::"memory")
#define memory_read() __asm__ __volatile__("lfence":::"memory")
#define memory_write() __asm__ __volatile__("sfence":::"memory")
```

Techniques d'optimisation basées sur le *renomage* de registres
Afin d'éliminer des pseudo dépendances de données on peut utiliser des variables à assignation unique. Cela peut se faire au niveau de la compilation (technique SSA : static single assignation) mais aussi au niveau du processeur. En effet les processeurs modernes disposent de bien plus de registres physiques que le jeu d'instruction en présente. Sachant que les opérations portant sur des registres différents commutent, on a intérêt à utiliser le plus grand nombre possible de registre à la fois. Le processeur dispose d'une unité fonctionnelle de renomage de registre qui alloue aux instructions ses registres (cette unité fonctionnelle occupe du reste pas mal de silicium sur nos processeurs). Par exemple, à une affection du type `x++` est associée une allocation de registre et une libération ceci permet d'augmenter le degré de parallélisme des instructions à exécuter :

f = f * x; x = x + 1 ;	allouer f1 f1 = f0 * x0 libérer f0 allouer x1 x1 = x0 + 1 libérer x0	Allouer x1 x1 = x0 + 1 libérer x0	allouer f1 f1 = f0 * x0 liberer f0

Problème de l'aliasing : lorsque des pointeurs sont calculés pour accéder à des zones mémoire on est obligé d'attendre le résultat du calcul des adresses pour déterminer la nature de la dépendance entre deux instructions : plusieurs pointeurs peuvent finalement désigner la même adresse (pb chevauchement).

```
void somme (int *a, int *b, int*c)
{
  for (int i=0; i < 10; i++)
    a[i] = b[i] + c[i]
 }
```

Pour optimiser ce programme il faut être capable de déterminer si une case mémoire peut être accédée par deux pointeurs différents… en fortran traditionnel pas de pb car pas de pointeur ! en C99 on le dit comme cela :

```
void somme (restricted int *a, restricted int *b, restricted int *c)
```

Notons que pour le moment gcc n'utilise pas cette information cependant les processeurs modernes sont équipés de circuit pour détecter les conflits (*disambiguition*).

Techniques d'optimisation pour les branchements

Il s'agit de limiter les dépendances de contrôle car les sauts (conditionnels) créent des bulles. On peut tout d'abord limiter le nombre de boucles au niveau de la programmation (fusion de boucles for successives) voire en les déroulant à la compilation :

<table>
<tr>
<td><pre>int f(int u, int v)
{
 for(int i=0; i <3; i++)
 {
   tmp = u;
   u = u + v;
   v = tmp;
 }
 return u;
}</pre></td>
<td>gcc -O3 -funroll-loops

=> recopier 3 fois le code et supprimer i - utilisation de variables uniques</td>
<td><pre>U1 = u0 + v
u2 = u1 + u0
u3 = u2 + u1</pre></td>
</tr>
</table>

A l'exécution le processeur peut aussi parier sur le fait qu'un saut va être pris ou non pour anticiper l'exécution de quelques instructions, on parle d'*évaluation spéculative.* Le processeur parie qu'une branche va être prise plutôt qu'une autre et précharge des instructions des données, réalise des calculs tout en retardant les écritures (mémoire / registre). Les écritures ne seront réalisées que si le pari est gagné, introduction d'une phase de validation :

fetch decode read execute write back *commit*

On peut même exécuter en // plusieurs branches pour n'en valider qu'une seule. Cependant évaluer plusieurs branches complexifie les circuits pour une efficacité relative (au final on a des « bulles » partout sauf dans une branche). Les fondeurs ont privilégié une autre approche : augmenter ses chances de gagner les paris. Il s'agit de déterminer avec une « bonne » probabilité l'adresse de la prochaine instruction à réaliser, on parle de *prédiction de branchements.*

Exemple :

```
for (i= 0; i < 100; i++) // 101 tests
  for(j=0; j< 10; j++)  // 1100 tests
   ...
```

Stratégies de prédiction simples :

- stratégie toujours… on se trompe à chaque fin de boucle : 1 + 100 erreurs
- stratégie jamais… complément de toujours 99 + 1000
- stratégie 1 bit… (comme la dernière fois) 1 + 200
- stratégie 2 bits… (comme d'habitude) équivalente ici à toujours

Dans les processeurs on trouve une table de hachage indexée par l'adresse du saut et un historique des *x* dernières décisions donne accès à une fonction de prédiction (1 ou 2 bits). Sur notre exemple le microprocesseur apprends la séquence interne au bout d'un tour de boucle s'il dispose d'un historique sur 11 bits ou d'un compteur sur 4 bits.

Les derniers processeurs prennent aussi en compte l'historique globale comme le core 2 :
addresse du saut + historique global + local (ou compteur) → prédicteur 2 bits.

```
If (c==1) …. ;
 …
If (c == 1) … ; // repéré par une stratégie globale
```

NB. il faut noter l'important coût matériel de ce type de prédiction, core 2 se limite à pister une dizaine de branchement – il faut donc faire la chasse aux branchements conditionnels inutiles, regrouper les boucles.

De plus, pour prédire l'adresse d'un saut un historique est utilisé pour les pointeurs de fonctions et les switchs, une pile interne au microprocesseur est utilisée pour l'adresse de retour (calls / ret).

MultiThreading au sein du processeur

Malgré toutes ces techniques il reste encore des bulles dans le pipeline, des unités fonctionnelles au repos, des registres non utilisés... en particulier lors des défauts de cache où le pipeline peut même se vider.

Registres -> mémoire cache -> mémoire principale

Temps d'accès

	ns	cycle
L1	1	2-4
L2	10	7 (opteron) - 22 (p4) - 14 (core 2)
Mémoire	60	240

Par exemple les latences mesurées des caches du K10 sont de 3 cycles pour le L1, 15 cycles pour le L2, 30 à 45 cycles pour le L3.

Un défaut de cache coûte donc de l'ordre 200 cycles, il faut donc les éviter en pré-chargeant le cache (instructions de prefetching) soit de façon logicielle soit au niveau du processeur. Une technique orthogonale est de faire tourner plusieurs threads sur le même pipeline, on parle de *MultiThreading*, il y en a différentes sortes :

- - tourniquet comme machine processeur HEP (1980), machine (Tera, 1998) : on entrelace de facon cyclique jusqu'à 64 (128) threads… on a plus besoin de cache.
- Super threading : à chaque cycle on peut changer de thread (sun Niagara, graup au cremi)

- SimultaneousMT : les instructions sont entremêlées (Pentium IV, Nehalem (infini au cremi), Power 5 - 6)

SMT ainsi on a des flux indépendants et le réordonnancement des instructions peut jouer à pleinement son rôle. Pour mettre en œuvre cette technique il faut ajouter du matériel:
- dupliquer PC et la TLB (pour les processus – ou indiquer le numéro ds la table)
+ augmenter le nombre de registres
+ augmenter les files d'attentes des unités de traitement
+ augmenter les caches

D'après la publicité d'Intel 5% de matériel en plus permet d'obtenir jusqu'à 20% de perf en plus… mais ce n'est pas garanti parfois on observe des pertes de performances - incompréhension du public 1 ou 2 processeurs ?

Difficultés de l'approche
- priorité des threads, équité => file d'attente disjointe pour traiter équitablement les threads ?
- cohabitation problématique (*symbiotic threads* - appariement)
Problème de concurrence sur le cache

Commentaire général
De nombreuses autres techniques d'optimisation sont utilisées dans les processeurs par exemple sur le core 2 un cache de micro-operation (trace-cache) permet d'éviter de décoder sans cesse les *petites* boucles et permet de fusionner certaines micro-instructions.

- couple (compilateur , microprocesseur) indissociable ;
- superscalaire + renomage + réordonnancement dynamique + prédiction de branchement permettent d'améliorer le rendement d'un pipeline ;
- coût en silicium et en énergie très important du réordonnancement (tampon, logique de sélection, cache non bloquant)

Écoles actuelles
➔ proposer du réordonnancement dynamique (pas satisfaisant intellectuellement... un peu comme si on demande a un programmeur C de travailler en lisp)

➔ demander au compilateur d'ordonnance et l'assemblage des instructions (pas de réordonnancement) et proposer un jeu d'instructions permettant de le faire (instruction prédicative) architecture EPIC : processeur Itanium Nécessite un compilateur très évolué... et finalement il faut faire des benchmarks pour déterminer la bonne tactique à employer... échec commercial !

➔ Utiliser des processeurs très simples en parallèle (approche *manycore*)

Les caches
Objectif du cours = comprendre ce schéma (Hennessy & Patterson)

- Plus une mémoire est petite plus son temps de réponse est faible, *Small is beautyfull…*
- Tirer parti de la localité spatio-temporelle des programmes « *90% de l'exécution se déroule dans 10% du code* »

Un cache c'est une mémoire très rapide qui contient en « copie » certaines zones de mémoire, les données utilisées par le pipeline sont copiées dans le cache, éventuellement modifiée à travers celui-ci et recopiées en mémoire lorsque nécessaire. Comme le cache est bien plus petit que la mémoire, il arrive fatalement qu'une donnée en chasse une autre : on parle alors de conflits de cache. Il faut absolument éviter les parties de ping-pong entre le cache et la mémoire. Lorsqu'une donnée recherchée est dans le cache on parle d'échec (cache-hit) autrement d'échec ou de défaut de cache (cache-miss).

Organisation matérielle des caches
Il y a plusieurs niveaux de caches L1, L2, L3 qui sont plus en plus grand et donc de plus en plus lent. Données et instructions ne sont pas rangés systématiquement dans les mêmes caches. Souvent donnée et instructions sont séparés pour le L1 et pas pour le L2. Le pipeline réclame ses données au cache L1 qui d'une certaine façon réclame lui même ses données au cache L2 qui … lui même les réclame à la mémoire qui elle même (enfin ce n'est pas du tout elle) s'adresse au disque de swap… Voici grossièrement l'impact sur les performances de la localisation des données :

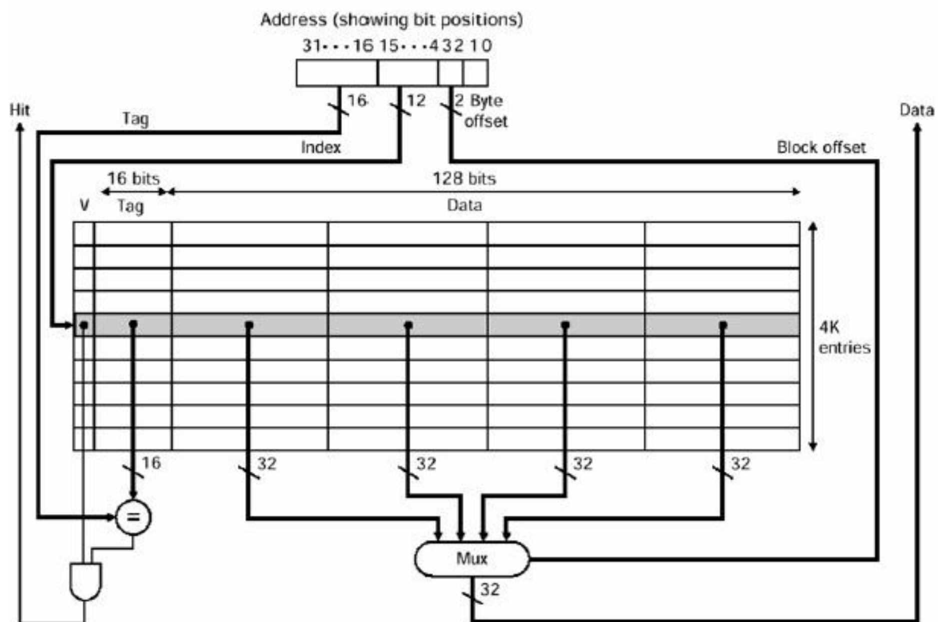
- L1 : ~1 ns soit 2 à 4 cycles => pas d'impact sur le pipeline
- L2 : ~4 ns soit 10 cycles => ralentissement du pipeline qu'OoO permet d'absorber en partie,
- Mémoire ~60 ns soit 200 cycles => interruption du pipeline (multithreading, pré-fechting introduit par le compilateur)
- Swap ~10ms => il vaut mieux changer de processus.

Au niveau du cache l'unité d'information n'est pas l'octet ni même le mot mais la *ligne de cache* qui contient un *certain nombre* d'octets (souvent 64 octets) d'adresses consécutives. Cette technique d'optimiser le coût matériel nécessaire à la logique de sélection et aussi le temps d'accès séquentiel aux données rangées séquentiellement dans des tableaux. Comme les bus de données ne font pas encore 64 octets de large des optimisations sont nécessaires pour obtenir le plus rapidement possible le premier octet demandé en lecture et différer les écritures en mémoire (tampon de sortie).

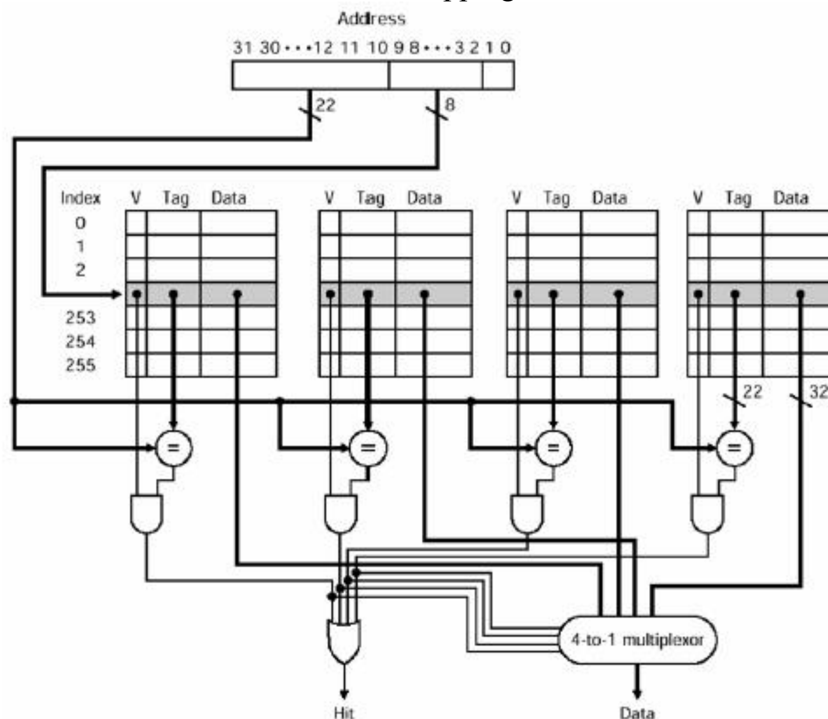
Associativité des caches
Pour obtenir une donnée le pipeline présente au cache l'adresse du mot mémoire recherché. Pour déterminer si la donnée est dans un cache, il est nécessaire de stocker non seulement les données mais aussi l'adresse correspondant (sauf les 6 bits de poids faible correspondant aux 64 octets de la ligne de cache). Le couple ligne de cache / adresse est appelé *entrée* du cache. Étant donnée une adresse il s'agit donc de déterminer si elle correspond ou non à une entrée, cela ce fait à l'aide de circuits de comparaison gravés dans le silicium.

Lorsque que toute adresse peut correspondre à toute entrée on parle de *cache associatif* : l'entrée contient en fait toute l'adresse et on dispose d'un comparateur par entrée. Par exemple le cache L1 de la TLB de l'opteron est un cache associatif, il ne contient que 40 entrées. En

cas de conflit un algorithme proche de LRU (Least Recent Used) est utilisé pour désigner une *victime* et la remplacer une autre ligne de cache.
À l'opposé lorsque que chaque adresse ne peut correspondre qu'à une unique entrée on parle de *cache à correspondance directe* et il suffit d'un seul comparateur pour l'ensemble des entrées. Pour établir la correspondance adresse / entrée on utilise les bits de poids faible de l'adresse cible sauf les 6 derniers car il n'y a aucun intérêt à choisir les bits de poids forts (très forte probabilité de conflits).
Les caches modernes utilisent en fait un modèle mixte : à une adresse donnée peut correspondre *n* entrées (*n-way cache*) et donc n comparateurs sont nécessaires.



Direct mapping



Set-associative mapping
4 way cache

Dans les schémas sont souvent représenté la largeur des champs (étiquette, index, offset) par exemple pour une adresse physique sur 40 bits et un cache de 64 ko on a :
Cache à correspondance direct (étiquette, index, offset) = (24,10,6) car on a 1k d'entrées

6 bits de poids faibles pour les 64 octets dans le cache ;
10 bits de poids intermédiaires pour désigner une entrée parmi 1k ;
24 bits sont à mémoriser dans l'étiquette.

Cache semi associatif 2-way de 64ko 512 x 2 entrées de 64 bits (25, 9, 6)
Cache semi associatif 8-way de 64ko 128 x 8 entrées de 64 bits (27, 7, 6)

Les conflits
On dispose d'un cache de 10 entrées les lignes de cache font 1 octets
Cache direct : 2 adresses sont en conflit si elles terminent par le même chiffre
Cache 2-way : 2 adresses sont potentiellement en conflit si elle sont égale modulo 5

Soit la séquence : 1 3 5 13 12 24 34 3 4 13
Cache direct : 1 3 5 **13** 12 24 **34 3 4 13** 29 18
Cache 2-way + LRU : 1 3 5 13 12 24 34 3 **4** 13 **29 18**

On observe qu'un cache 2-way de taille T est aussi performant qu'un cache direct de taille 2 T.

Maîtrise de l'utilisation du bus
2 techniques d'écritures lors d'un cache-miss
write-allocate => la ligne est chargée dans le cache avant d'être modifiée *: il faut lire avant d'écrire !!!*
write-no allocate => on modifie directement la donnée en mémoire

2 techniques d'écritures si ligne présente
write-through => écriture simultanée on écrit toujours à la fois dans le cache et la mémoire (buffer)
write-back => on marque la ligne de cache comme modifiée et on la sauve en mémoire que sur obligation (au moment du remplacement)

→ couple write-back+ write-allocate minimise l'utilisation du bus

Impact des caches sur la programmation

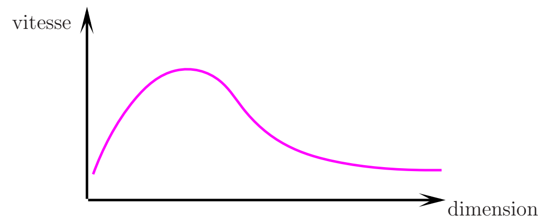
il faut faire attention à l'ordre de l'imbrication des boucles...

```
for(i = 0; i < k; i++)
    for(j=...)
        t[i][j]= ...
```

l'adresse cible de t[i][j] est t + i * k + j où k correspond à la taille deuxième dimension de t. On remarque qu'on ne bénéficie plus du tout du cache si k est plus grand qu'une ligne de cache et que le tableau ne rentre entièrement dans le cache (remarque : en fortran les données sont stockées suivant la 1ere dimension par « colonnes »).

Application au jeu de la vie : on a intérêt a travailler par petit bloc (des *tuiles*) qui tiennent (exactement) dans le cache => introduction de boucles supplémentaires

```
for (I=...  I += TI) // pour toutes les tuiles …
 for(J=...  J+=TJ) // toute les tuiles
  for(i=I
   for(j=J
```



effet de la taille des tuiles sur les performances
http://www.ann.jussieu.fr/MathModel/polycopies/fxroux3.pdf

Remarque: les caches modernes sont capables de détecter des accès réguliers à la mémoire et peuvent ainsi faire du prefetching : mémoire -> L2 -> L1

La communauté a dégagé deux styles de programmation adaptés aux caches :
- ⇨ algorithmes *cache conscious,* tenant compte de la taille du cache
- ⇨ algorithmes *cache oblivious,* pensés pour optimiser la localité du travail (Dived & Conquer)

Les scientifiques utilisent des bibliothèques spécialisées caches-conscious les "basic linear algebra system" BLAS.
BLAS1 : vecteur O(N) O(N)
BLAS2 : vecteur matrice memoire O(N^2) pour O(N^2) opérations
BLAS3 : matrice matrice memoire O(N^2) pour O(N^3) opérations

Les BLAS de niveau 3 permettent de mieux exploiter les caches : on peut obtenir des facteurs 10 en performance. Ramener le problème à des opérations matrice / matrice quitte à faire plus d'opérations.

Machines multiprocesseurs à mémoire partagée
Deux types d'organisation dominent le marché SMP et NUMA (réseau de processeurs)

Type SMP (symetric multi pro) : Il y a d'un coté la mémoire et les processeurs de l'autre ; l'interconnexion est classiquement un bus ou un réseau devant la mémoire voire devant le cache. Un mécanisme d'arbitrage est chargé de gérer l'accès au bus et donc à la mémoire. Sur certains systèmes, les processeurs peuvent verrouiller le bus pour faire de la synchro faire des read/write atomiques. Une autre technique est de verrouiller le cache.

Il eut existé des système à bus unique comportant des dizaines de procs mais bon maintenant dès qu'on dépasse 4 sockets on utilise plusieurs bus en étoile (voir l'exemple du quad core 2 d'intel => filtre au niveau du chipset pour pouvoir servir en parallèle plusieurs requêtes). Mais cette approche en étoile est limitée. Une autre approche est de mettre le cache devant la mémoire et utiliser un crossbar pour y accéder comme sur le processeur Niagara (graup au CREMI) : 8 cores 4 cache (3 MB).

dudley1

graup

Cohérence de la mémoire sur SMP
Pb les données peuvent être dans la mémoire, dans des caches, dans des registres… quelle est la bonne version ? L'objectif des machines à mémoire commune est de donner au

programmeur la même vision de la mémoire que sur une machine monoprocesseur programmée à l'aide de threads (exécutés en séquence).

→ Consistance séquentielle (Lamport 1979), Sémantique de l'entrelacement :
« Un multiprocesseur est séquentiellement consistant si toute exécution résulte d'un entrelacement des séquences d'instructions exécutées par les processeurs et qui préserve chacune des séquences. En particulier tous les processeurs voient les écritures dans le même ordre. »

exemple : x = y = 0 → on ne doit pas voir 0 0

T1	T2
x = 1 print y	y = 1 print x

Du coup le programme suivant est mal programmé
ok = 0

resultat = 3.14 ok = 1	while(!ok) ; print resultat

Car le processeur ou le compilateur peuvent interchanger l'ordre d'exécution des instructions. Il faut utiliser des barrières mémoire ou des outils de synchronistation (mutex) qui empêchent les instructions de commuter, forcent l'écriture vers la mémoire des registres (barrière mémoire) et terminent les écritures pendantes.

Compromis cohérence / performance
Objectif : ne pas saturer le bus => minimiser les échanges avec la mémoire
Comme la stratégie "write through" monopolise le bus on préfère la stratégie write-back
il faut connaître et actualiser l'état des données dans le cache (a-t-on LA bonne version ?)

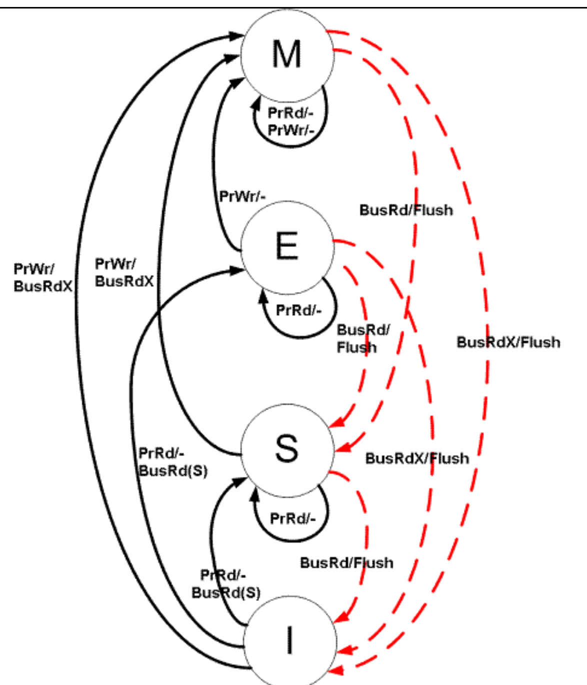
Protocoles de cohérence de cache

Evénements :
PrRd PrWr
BusRd BusRdX
Miss Flush Replace

MSI Modified Shared Invalid

MESI distinction entre shared et pas shared afin de ne pas écrire le busRdx si pas nécessaire, necessite un signale S (shared)

MOESI O = partagée mais sale (opteron) utile si on a la possibilité de lire la valeur sur le bus sans l'écrire dans la mémoire



Instructions atomiques

type __sync_fetch_and_add (*type* *ptr, *type* value, ...) type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)

PB du faux partage (false sharing)
Deux variables a priori indépendantes peuvent se retrouver dans la même ligne cache

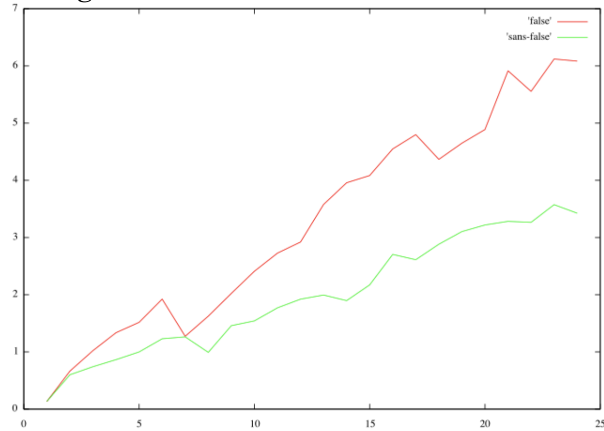
```
int x, y;
```

Cette ligne de cache peut alors faire du ping-pong entre deux processeurs.

Solutions : *padding* ou directive d'alignement (nécessite la connaissance de la taille des lignes de cache), introduction des TLS => allocation de pages par thread

```
int x __attribute__((__aligned__(64)));
```

Influence du false sharing sur une barrière en mode attente active

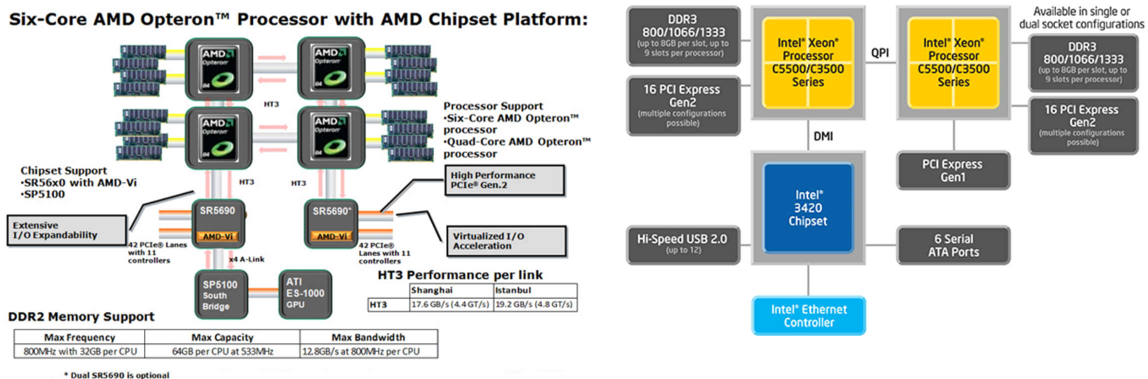


Conclusion sur SMP :

- \+ SMP facile a programmer, peu onéreuse
- \- contention sur le bus, augmentation latence mémoire, faible nombre de processeurs
- \- l'augmentation du nombre de cœur par processeur a rendu ce modèle obsolète pour le calcul haute performance.
- \-

Machines de type NUMA (Non uniform memory architecture)

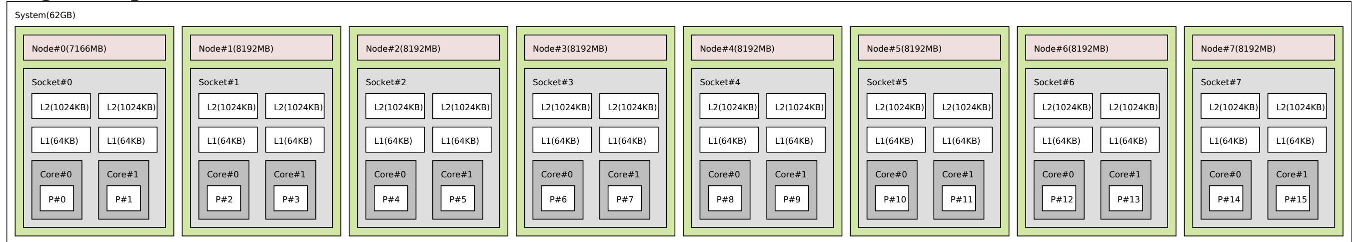
Il s'agit d'un ensemble de briques (mémoires + processeurs) communiquant au travers d'un réseau : cette technique permet d'assembler plus un grand nombre de processeurs (centaine voire des milliers) mais complexifie la compréhension des performances. En effet le temps d'accès aux données varie suivant la distance et surtout suivant la contention.



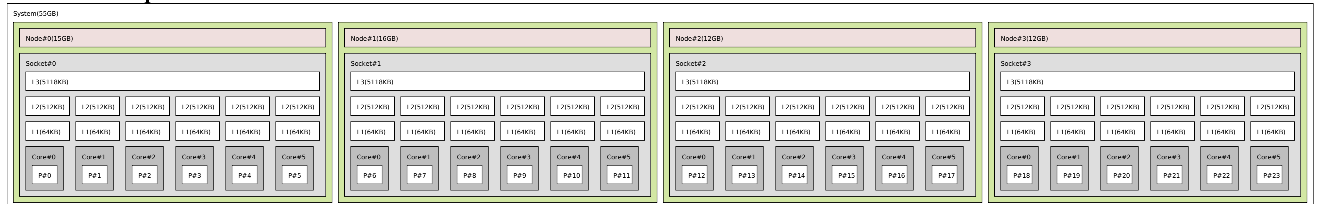
	Shanghai	Istanbul
HT3	17.6 GB/s (4.4 GT/s)	19.2 GB/s (4.8 GT/s)

Max Frequency	Max Capacity	Max Bandwidth
800MHz with 32GB per CPU	64GB per CPU at 533MHz	12.8GB/s at 800MHz per CPU

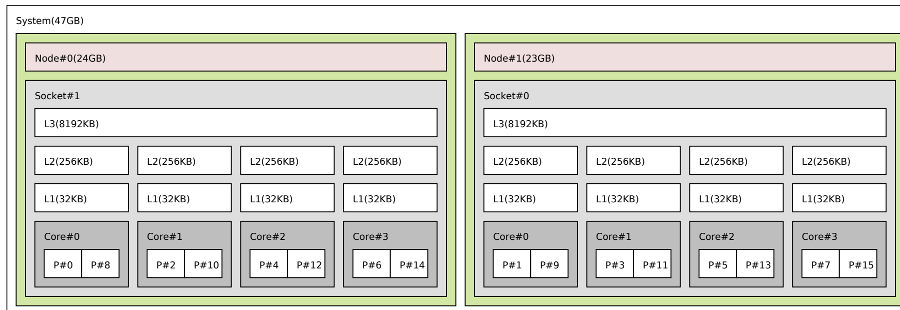
Hagrid : opteron 8000



fridulva : opteron Barcelona 4000



Infini1 : xeon nehalem



Bertha core2 xeon (machine de recherche)

Comment obtenir une ligne de cache ?
Machine DASH : un circuit spécialisé permet de mémoriser l'emplacement actuel des (groupes de) lignes de cache sur le Home Node (nœud où réside la mémoire contenant la donnée). On dispose d'un vecteur de bit par ligne permet de désigner les caches détenteurs de copies et un état de la ligne. : *shared* (au moins un cache l'a) *uncached* (aucun ne l'a) *exclusive* (un seul cache le détient et la donnée a été modifée).

Sur les opteron & nehalem on s'adresse au Home Node qui fait un broadcast si nécessaire. Voir http://www.nccs.gov/wp-content/uploads/2009/06/CrayORNL_120709.pdf

Difficulté de la maitrise des performances sur machines NUMA
Les performances d'un programme sur une machine numa dépendent de la localité des données (position respective des données et des traitement)

- *Latence* : Notion de facteur NUMA (Hagrid 1,1 -> 1,4) BULL novascale CEA 3.
- *Contention sur les bus*

Contrôler le placement des threads et de « leur » mémoire

Allocation "first touch" il s'agit de déterminer (ou de découvrir par benchmark) les dimensions où le programme à un comportement le plus régulier possible puis d'affecter de façon définitive des jobs aux threads en utilisant une politique de scheduling statique. Ensuite on fait une boucle d'initialisation à blanc pour fixer les pages sur les nœuds. Cette approche est performante lorsqu'on a thread par cœur.

Stratégie de rééquilibrage de charge "move-on-next-touch" bien adapté aux pbs pas trop irréguliers.

bibliothèques de placement de thread et de données (ex *libnuma*)
La libnuma permet de définir la politique d'allocation d'un thread et de ses fils (*default* : allocation locale, *bind* : allocation sur un ensemble fixé de nœuds, *interleaved* : répartie ,*prefered* : d'abord sur un noeud donné). On utilise malloc spécial : numa_alloc().

Possibilité d'utiliser la commande unix numactl pour paramétrer une exécution.

Conclusion : portabilité des performances
Pour obtenir des perfs on peut toujours "visser" un programme pour une architecture donnée. Mais on ne peut pas forcément le faire pour toutes les applis et pour toutes les archis…
Il s'agit donc de « programmer » en vue d'obtenir de très bonnes perfs de façon portable car la durée de vie d'une application peut être importante (plusieurs dizaines d'années).
Faire en sorte que les applications expriment les affinités threads / mémoire de façon portable.
C'est des sujets de travaux de recherche bordelais : on modélise la machine via une hiérarchie de runqueues ; de son coté l'application exprime l'affinité entre threads de façon portable lors de l'exécution et un ordonnanceur (programmable) plaque la hiérarchie de threads sur la hiérarchie des runqueues.

Parallélisme de données

1) DES MACHINES SMP AUX ARCHITECTURES NUMA

Il y a quelques années, on trouvait des PC multiprocesseurs (à 2, 4 ou 8 processeurs) avec un accès uniforme à la mémoire. C'était assez simple à programmer. On pouvait utiliser OpenMP sans arrière pensée. Bref, la vie était belle. À l'origine, OpenMP a justement été conçu pour ce genre de machine : des machines à accès uniforme à la mémoire. C'est sorti en 1996... C'est-à-dire un poil tard ! En effet, à cette époque, cela faisait longtemps que les constructeurs n'espéraient plus construire de "grosses" machines multiprocesseur à mémoire partagée avec un accès mémoire uniforme.

-> plus possible d'utiliser une technologie de type BUS pour l'accès à la mémoire (contentions)

-> l'utilisation de commutateurs (switch) atténue les problèmes de contention, mais augmente la latence en log(nb procs)

- Les constructeurs ont donc opté pour des architectures NUMA (Non Uniform Memory Access) afin d'augmenter le nombre de processeurs.

-> SGI (Silicon Graphics Inc.) est probablement le constructeur le plus connu aujourd'hui sur ce créneau (introduction d'une gamme de machines ccNUMAen 1996: Origin, puis Altix, etc.), mais il y en a eu bien avant: Cray X-MP (4 processeurs, 1984), Sun, DASH, FireFLy, etc.
-> Ces machines sont chère, peu extensibles, mais surtout sont bigrement difficiles à programmer ! Il faut placer les données près des processus qui les manipulent. Ce n'est pas si facile que ça, même quand on possède l'information. Alors, pour le système d'exploitation...

- Plusieurs constructeurs (dont Cray avec T3x, IBM avec SP, etc.)se sont orientés vers des machines où on abandonne la cohérence mémoire, voire même la mémoire partagée: on accède à sa mémoire rapidement, mais il faut utiliser des opérations explicites pour accéder à la mémoire des autres processeurs. C'est possible soit par accès mémoire distant (remote read, remote write), soit parenvoi de messages (send, recv).

- Le début des années 2000 a été fatal aux grosses machines multiprocesseur à mémoire partagée : les architectures de type "grappe de PC", au rapport performance/coût très supérieur, dominent le marché depuis...
-> En a-t-on pour autant fini avec les machines NUMA ? Non, mais vous saurez pourquoi au prochain épisode !

2) DES PROCESSEURS SCALAIRES AUX PROCESSEURS VECTORIELS... AUX PROCESSEURS SCALAIRES... AUX EXTENSIONS VECTORIELLES (MMX/SSE)...

- Dans le domaine du calcul scientifique, une large classe d'applications manipule des gros volumes de données sous forme de vecteurs (ou de matrices) pour appliquer des opérations similaires sur tous leurs éléments (addition de vecteurs, multiplication d'un scalaire par un vecteur, etc.). C'est ce que l'on appelle le "parallélisme de donnée".

- Pour accélérer ces applications, ce qu'il faut, ce n'est pas d'aligner plusieurs processeurs généralistes, mais plutôt d'offrir un support matériel spécifique pour appliquer en parallèle la même opération à plusieurs données simultanément. C'est le concept du fonctionnement SIMD (Simple Instruction - Multiple Data).

- Il y a plusieurs façon de concevoir des architectures mettant en oeuvre cette idée.
-> Machines Massivement Parallèles SIMD (i.e. DEC/MasPar)
-> Processeurs vectoriels

- Machine Massivement Parallèles SIMD
. Idée : une seule unité de contrôle (fetch-decode), et une grande quantité de processeurs élémentaires (des ALU quoi) qui exécutent tous en même temps la même instruction, sur des données différentes. Chaque processeur a un indice qui lui permet de savoir sur quel élément de tableau (vecteur, matrice) il doit travailler.
. Il n'y a pas d'exception à la règle "tout le monde fait la même chose" : une séquence if(conf) then A else B aura pour conséquence que certains processeurs exécuteront A pendant queles autres ne font rien, puis ensuite le complémentaire exécutera B pendant que les premiers de font rien.
. Il n'y a pas de mémoire "globale" partagée.
. Les communications avec les processeurs voisins proches se font très rapidement par des liens directs. En revanche, les communications "arbitraires" sont un peu plus chères via un full-crossbar switch.
. C'est marrant, mais en revanche les performances sont "à laramasse" sur les parties séquentielles du code ! En dans les applications parallèles, il en reste toujours...
. Peu de succès commercial (y'en avait quand même une à Lille au labo où j'ai fait mes études :)).
. Toutefois, cette technologie a connu un net regain d'intérêt assez récemment, grâce à un marché plus important que le HPC :les jeux... (on y reviendra plus tard)

- Processeurs vectoriels
. Idée : on agrandit les registres pour contenir plusieurs mots/réels, et on invente des instructions vectorielles (ADD,SCAL-MUL, etc.) qui appliquent la même opération sur plusieurs éléments à la fois (e.g. sur 64 mots à la fois dans un CrayX-MP).
. Deux manières de faire : soit on multiplie les ALUs (64additionneurs pour faire 64 additions en parallèle), soit on fait un additionneur super rapide et on pipeline lorsque plusieurs instructions sont appliquées successivement aux vecteurs. Ex: A =k.B + C. C'est la technique utilisée par les processeurs Cray.
. Notez que les processeurs vectoriels ont besoin d'un effort particulier pour être programmé. Typiquement, le langage Fortran, qui permet de manipuler des vecteurs et des matrices en les rendant visibles pour le compilateur (contrairement au C...) aété proposé pour ça. En Fortran, lorsqu'on additionne deux vecteurs, on écrit C = A + B, et le compilateur génére des instructions vectorielles... Hormis le compilateur, il reste la solution d'utiliser une bibliothèque spécialisée, que les concepteurs ont implémenté en assembleur en utilisant les bonnes instructions (ex: Basic Linear Algebra Subprograms: BLAS, FFT,etc.) Un peu de la même manière que les jeux utilisent DirectX au lieu de programmer la carte graphique directement...

- Les processeurs vectoriels au sens noble ont vu leursperformances dépassées par les processeurs généralistes, car :

. la complexité de ces processeurs est telle que les instructions scalaires "normales" sont pénalisées ;. la fréquence des processeurs scalaires a pu évoluer bien plus vite ;. le marché était un marché de niche, donc les processeurs étaient très chers.

- Les processeurs scalaires ont donc repris le dessus au début desannées 1990 (dans les machines parallèles).

- Toutefois, le marché du multimédia et des loisirs, qui est plus important que celui du HPC, a lui aussi des besoins similaires :traitement d'image, de la vidéo et du son. Pour accélérer le décodage d'un DVD ou certains jeux, les processeurs scalaires se sont vu adjoindre une extension pour effectuer des opérations vectorielles. Ex: le jeu d'instruction MMX (64 bits, uniquement sur les entiers, 1996, puis SSE, 128 bits (2 double ou 4float/int, 1999, PentiumIII) dans les processeurs x86.

- Là encore, il faut que le compilateur soit capable d'utiliser ces instructions, et donc de détecter où c'est possible dans un code C. Les compilateurs actuels (gcc -O3) ne sont pas mauvais à cet exercice, et les boucles vectorielles simple sont assez facilement détectées et optimisées.

- Note: l'émergence des cartes graphiques accélératrices a vite rendu le MMX peu utile pour les jeux.

2) VIVRE DANS UN UNIVERS EN ETERNEL RECOMMENCEMENT ?

- Les années paisibles: depuis le milieu des années 90, l'homme et les machines parallèles ont longtemps vécu en parfaite harmonie :les grappes de PC et les machines massivement parallèles sont des architectures à mémoire distribuée qui forcent les programmeurs à découper les applications en tâches qui communiquent explicitement au travers d'un réseau... Bien que moins naturel que le modèle à mémoire partagée, le modèle de programmation à mémoire distribuée a le mérite de forcer le programmeur à réfléchir aux coûts de communication (très élevés) et à écriredes algorithmes qui supportent plus facilement le "passage à l'échelle" sur des configurations de grande taille...

- Depuis le début des années 2000, l'univers du calcul parallèle a malheureusement subi encore plusieurs bouleversements :l'émergence des technologies "multicoeurs", et l'utilisation d'accélérateurs graphiques pour accélérer les calculs SIMD.

- Les raisons de l'arrivée des processeurs multicoeurs : plafonnement de la fréquence, vaines tentatives de "complexification" des processeurs actuels (les prédicateurs, par exemple, sont difficilement améliorables), et miniaturisation quine faiblit pas : il reste de la place sur les puces, alors autant y graver plusieurs coeurs juxtaposés.

- C'est désormais définitif : le parallélisme est partout, même dans les téléphones.

3) LE RETOUR DU NUMA, ET MÊME PIRE...

- Les processeurs multicoeur on ré-introduit les effets NUMA des machines multiprocesseur du temps jadis... Seulement,aujourd'hui, plus moyen d'y échapper : ces processeurs

sontpartout. Il faut donc faire un effort de programmation parallèlemême pour exploiter efficacement un vulgaire core 2 duo.

- Pire, certains constructeurs ont introduit des processeurs multicoeur dans lesquels les différents coeurs n'ont pas lesmêmes caractéristiques, voire ne partagent pas une mémoire cohérente !

- C'est l'approche choisie par IBM pour la conception du processeurCell/BE (2005, qui équipe la PS3).
. 1 Power Processing Element (PPEn dual-threaded) + 8 SPE(Synergistic Processing Elements, SIMD-capable co-processors).
. Les SPE n'ont que 256Ko de mémoire locale. Ils doivent utiliserdes DMA (très rapides, 3 cycles de latence) pour accéder à lamémoire principale ainsi qu'à la mémoire des autres SPE.
. Le problème de la cohérence mémoire est reporté au niveau logiciel !
. 100 Gflops sur LINPACK
. Le code exécuté par les SPE est compilé avec un compilateur annexe, et le binaire est chargé par le PPE sur le SPE.
. À noter que le Cell est utilisé dans la machine IBM Roadrunner(désormais n°2 au top 500 je crois) 1.7 Petaflops théoriques, 1.45Peta soutenus. 12000 PowerXCell + 6000 AMD Opteron Dual-core.
. IBM a annoncé récemment stopper le développement du processeur Cell, tout au moins dans le contexte du HPC. (mais la Playstation4 se sera apparemment pas construite autour d'un Cell, donc...)
. Ceci dit, le Cell préfigure probablement l'évolution prochaine des processeurs : quelques gros coeurs généralistes (pour exécuter vite les codes séquentiels) et plusieurs petits cœurs spécialisés (SIMD) pour les traitements parallèles.
. Intel a annoncé un processeur hybride Larrabee (48 coeurs) quiest composés de coeurs généralistes couplés à des unités SIMD...
. En quelque sorte, on intègre des mini-GPU tout près des cœurs généralistes. Des minis-quoi ? G.P.U. !

4) L'ARRIVEE DES GPU DANS LE CALCUL PARALLELE

- Au départ, certains ingénieurs/chercheurs ont eu l'idéed 'utiliser ces accélérateurs graphique pour leur faire faire des calculs sur les vecteurs et les matrices. De nombreuses Success Stories sont apparues, avec des accélérations de x30 à x200(comparativement au même code sur un quad-core Opteron).

- vu le prix d'une carte graphique, ça a suscité un engouement terrible. Pourtant, on ne peut pas dire que c'est aussi aisé à programmer qu'une machine vectorielle !!!

- Architecture typique des GPU

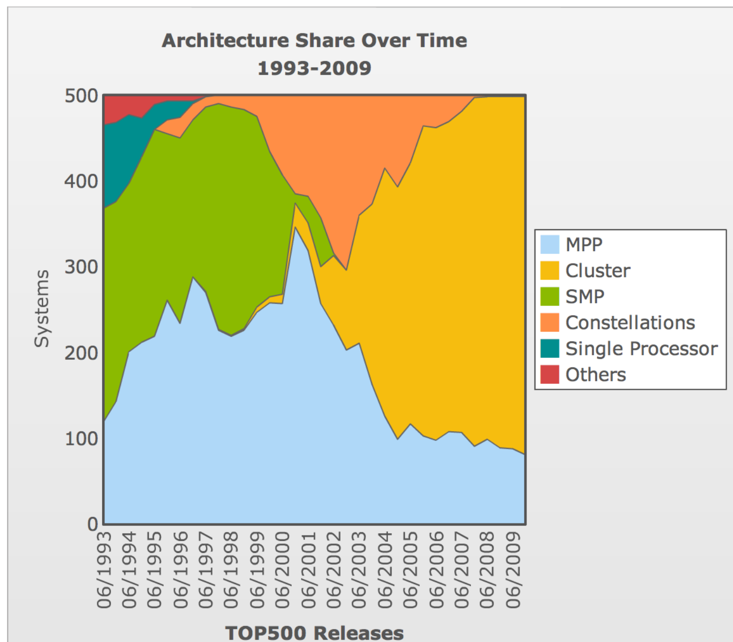
- Notion de noyau de calcul

- masquage de la latence

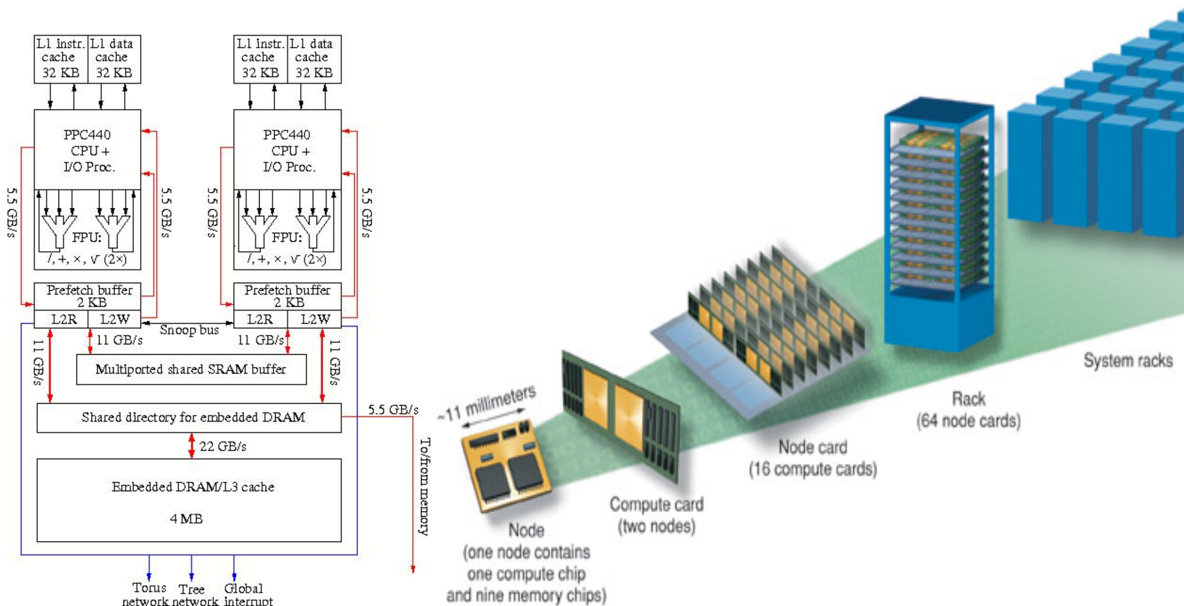
- programmation bas-niveau: CUDA, OpenCL
. Contexte, Programme, Noyau, Files.

Approche distribuée

Les architectures à mémoire commune sont limitées par la difficulté d'assembler efficacement un grand nombre de processeurs : plus le nombre de processeurs est important plus le fait de vouloir assurer la cohérence est coûteux en temps et en argent. Actuellement (et en attendant une révolution technologique) il est en fait plus rentable d'utiliser des machines en réseaux pour construire des plateformes de calculs conséquentes. Les plateformes de calculs actuelles sont soit des grappes (*cluster*) de SMP ou de petites NUMA (telles les machines infinis du CREMI) de smp, de numas, ou soit des clusters de machines spécialisées (IBM Bluegene, Roadrunner). Il y a aussi de plateformes de calcul basée sur internet : grille de calcul (cloud computing) et réseaux pair à pair.
Voir http://www.nwo.nl/files.nsf/pages/NWOA_7X8HXB/$file/overview09.pdf



TOP500



IBM Blue Gene : aucune cohérence de cache.

http://www.cerfacs.fr/globc/publication/technicalreport/2008/perfs_cgcm_bg.pdf

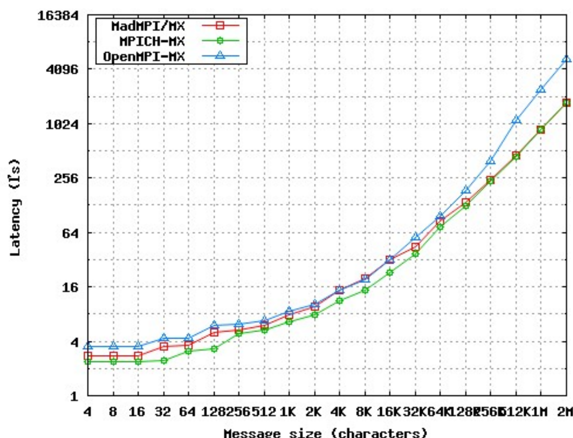
L'IBM Blue Gene est configuré de la manière suivante :
1 noeud de login quadri processeur Power5+ 1.5GHz 8 GO de mémoire
1 noeud de service quadri processeur Power5 2GHz 4 GO de mémoire
1024 noeuds de calcul soit 2048 processeurs Power PC440 16 noeuds d'IO (soit 1 noeud d'IO pour 64 noeuds de calcul)
2 serveurs GPFS des répertoires utilisateurs.
Les noeuds de calcul sont répartis dans 2 "midplanes". Chaque midplane" contient 16 "node cards". Chaque "node card" contient 32 noeuds et 0 à 2 cartes IO.
Le noeud possède les caractéristiques suivantes :
2 processeurs PowerPC440 700 Mhz (4 Flops x 700 Mhz =2.8 GFlops de performance crête par processeur,adressage mémoire 32 bits) 512 MO de mémoire par processeur soit 1GO par noeud Caches L1 (instruction et données) 32 KB par processeur, cache L2 4KB par processeur, cache L3 4MB par noeud
Trois réseaux d'interconnexion sont utilisés pour les calculs sur Blue Gene :
un tore 3D entre les noeuds de calcul, 2.1 GO/s de bande passante, entre 4 et 10 µs de latence, pour les communications MPI point à point un arbre entre noeuds de calcul et noeuds IO, 700 MO/s de bande passante, 5 µs de latence pour les IO, les communications globales, opérations de réduction
un réseau pour les synchronisations globales (MPI_BARRIER) et interruptions, 1.3 µs de latence
Deux modes de calcul sont disponibles sur blue gene :
Coprocesseur mode (CO) : le CPU0 gère le calcul, le CPU1 gère les communications point à point, La communication recouvre le calcul. Performance crête de 5.6/2=2.8 Gflops
Virtual node mode (VN) : CPU0 et CPU1 gère des taches indépendantes « virtuelles », le calcul et la communication ne se recouvrent pas. La performance est de 5.6 Gflops mais on divise par 2 : la mémoire disponible (512 MO au lieu de 1GO), le cache L3 (2MO au lieu de 4), la bande passante réseau.
Le mode coprocesseur bénéficie aux applications communiquant beaucoup, alors que le mode virtuel convient aux applications plus CPU intensive, n'ayant pas besoin de plus de 512 MO de mémoire.

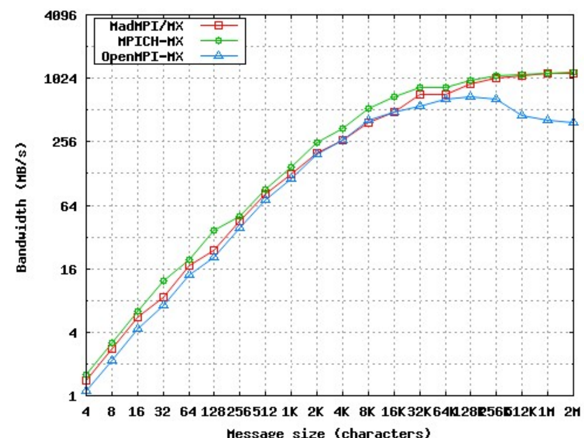
Il s'agit ici de faire travailler un ensemble de machines à la résolution d'un même problème, celles-ci devront donc communiquer pour collaborer. Deux approches majeures se distinguent pour programmer ces plateformes de calcul : soit le programmeur écrit explicitement les communications (communication par envoi de message), soit l'environnement de programmation les lui cache (communication implicite). Cela traduit un positionnement sur le compromis performance / facilité de développement : les applications sont plus ou moins sensible aux communications suivant le volume et la fréquence des données échangées et la nécessaire synchronisation des différentes machines. Pour fixer les idées voici les deux paramètres mis en avant par les fabricants de cartes pour évaluer le coût d'une communication :

→ la latence (le ping) : temps de transmission d'un octet (400 us – 1us)

→ le débit : nombre d'octets transmis par seconde en régime continu (1gb/s – 40 gb/s)

Les courbes des performances d'une carte spécialisée ressemblent à cela…





Pour calculer la latence au niveau de l'application, on utilise la technique du ping-pong qui calcule le temps d'un aller-retour : pour calculer le débit on peut soit envoyer un gros paquet et attendre la fin du send, soit utiliser la technique du ping-pong.

Quelques mot sur les différentes méthodes de communication :
3 techniques : PIO (jusque 128 octets) - DMA + copie (32 ko) - DMA zéro-copie (rdv)

Mode *Glouton* : petits messages PIO, pour les moyens DMA+copie
On suppose qu'il y a de la place pour stocker les données coté émetteur pour éliminer des synchronisations - Inconvénients : risque de saturation des buffers internes pouvant provoquer des deadlocks, consomme du CPU, passe mal à l'échelle

Mode *Rendez-vous*: moindre consommation de ressource et efficacité contre latence plus importante (il faut attendre l'acquitement du receveur).

Objectifs des communications dans le HPC :

1° transmission performante : favoriser la latence et le débit, peu solliciter le cpu pour communiquer, essayer de recouvrir les communications par du calcul. On peut utiliser différentes techniques pour découpler les appels à send/recieve de l'émission / réception physique des données. Dans ce cadre, deux notions ont été définies pour qualifier ce découplage : l'aspect bloquant et l'aspect synchronisant.

- Synchronisation avec le receveur : send ne se termine pas tant que le recv n'a pas débuté (asynchrones : l'émetteur continue son exécution).
- Envoi bloquant : send ne se termine pas tant que le message n'a pas été entièrement transmis (le buffer d'envoi n'est pas libre) en mode non bloquant l'émetteur dispose de procédures de test de fin de réception.

2° communications en parallèle (bissection bandwith : on fait communiquer la moitié de la grappe avec l'autre moitié)

3° transmission fiable (message pas altéré, contrôle de flux) : les piles de protocoles complexes comme TCP/IP/Ethernet peuvent être simplifiées car elles sont taillées pour internet ; elles ont donc des spécificités inutiles pour la communication internes aux clusters (ex. gestion de la congestion).

Peu de problème de sécurité jusqu'à présent car peu de mutualisation du matériel on a l'exclusivité des machines on gère la sécurité avant et après l'exécution de l'appli.

Approche communication par envois de message : le standard industriel : MPI Message Passing Interface (1994) Voir https://computing.llnl.gov/tutorials/mpi/
Basé sur des requêtes de type send / receive pour assurer la communication entre processus

```
MPI_Send(buffer,count,type,dest,tag,comm)
MPI_Recv(buffer,count,type,source,tag,comm,status)
```

Portable, grand nombre de fonctionnalités, disponible sur toutes les plateformes : incontournable. Par exemple le système d'exploitation blue-gene d'ibm a été conçu pour bien fonctionner avec MPI (au dépit de certains ingénieurs perfectionnistes).
→ 2 type de communications : point à point / collective
→ des outils pour faciliter la programmation distribuée (barrière, réduction, type dérivé)

HelloWord

```
#include <stdio.h>
#include "mpi.h"
int main( int argc, char *argv[]){
int rank;int size;
MPI_Init( &argc, &argv );
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf( "Hello world from process %d of %d\n", rank, size );
MPI_Finalize();
return 0;
}
> mpicc –o hello hello.c
> mpiexec –machinefile les-machines –n 10 hello
```

Communication d'un jeton en anneau

```
    if (rank == 0)
     {
        printf( "Jeton lance par le maitre (%d participants) \n", size );
        MPI_Send(&token, 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
        MPI_Recv(&token, 1, MPI_CHAR, size-1, tag, MPI_COMM_WORLD, &etat);
        printf( "Jeton reçu par le maitre \n");
     }
    else
     {
        MPI_Recv(&token, 1, MPI_CHAR, rank-1, tag, MPI_COMM_WORLD, &etat);
        printf( "Jeton chez  %d  \n", rank);
        MPI_Send(&token, 1, MPI_CHAR, (rank+1) % size, tag, MPI_COMM_WORLD);
     }
```

Jeton centralisé

```
if (rank == 0) {
        for(i = 0; i < 3*(size-1); i++) {
          MPI_Recv(&token, 1, MPI_CHAR, MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &etat);
             MPI_Send(&token, 1, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD);
          MPI_Recv(&token, 1, MPI_CHAR, etat.MPI_SOURCE, 2, MPI_COMM_WORLD, &etat);
        }
        printf( " done \n");
    } else
        for(i = 0; i < 3; i++){
          MPI_Send(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);
          MPI_Recv(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &etat);
          printf( "Jeton chez  %d  \n", rank);
          //sleep(1);
          MPI_Send(&token, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);
    }
```

Schéma de calcul maître esclave : on remplace le token par le travail à réaliser

---

Les principales primitives de communications de MPI

Objectifs : exprimer les communication de façon efficace, donner le moyen favoriser le recouvrement calcul / communication

1/ Appels bloquants :

```
Send / Recv : termine une fois que buffer peut-être réutilisé / utilisé
Ssend / Recv : Ssend termine lorsque le buffer est réutilisable et que recv a commencé
Bsend / Recv : Bsend termine après recopie des données dans un buffer
```

On utilise Buffer_attach(void*,size_t) / Buffer_detach pour allouer *le* buffer du processus.

```
MPI_Probe(source,tag,comm,&status) : attente d'une réception sans la débuter sa réception
MPI_Get_Count(&status, datatype, &count) :
```

```
status.source   status.tag    status.error : variable status
```

2/ Appels non bloquants

Découpler la soumission d'une requête de son acquittement pour réaliser du recouvrement calcul / communication mais aussi pour poster plusieurs requêtes (émission et/ou réception à la fois). Il faut bien faire attention à ce que le buffer ne soit pas être réutilisé trop tôt pour cela on utilise une fonction de *complétion* wait / test permettant d'attendre/ de vérifier l'état de chaque requête. Nécessite d'associer un objet du type MPI_Request à chaque requête de Isend / Irecv.

```
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
MPI_Irecv (&buf,count,datatype,dest,tag,comm,&request)
```

```
MPI_Wait (&request,&status)
MPI_Test (&request,&resultat,&status)
```

NB. Irsend Ibsend Issend : influe sur le résultat test/wait qui indique la fin de la réception

→ possibilité de tester / attendre un tableau de requête à la fois any some all

```
MPI_Waitall (count, &array_of_requests,&array_of_status)
```

Exemple communication dans le jeu de la vie

```
For(etape=0 ; etape < NB ; etape++)
{
calculer_bordure
poster les requêtes
calculer_interieur
attendre les requêtes
echanger in et out
}

cellule[2][DIM][DIM] → cellule[2][XDIM][DIM]    avec XDIM = DIM/size + 2

MPI_Request req[4] ;
MPI_Status sta[4] ;
…
Isend(cellule[out][XDIM+1], MPI_CHAR, DIM,  haut, tag, MPI_COMM_WORLD, &req[0]) ;
Isend(cellule[out][0], MPI_CHAR, DIM,  bas, tag,  MPI_COMM_WORLD, &req[1]) ;
Irecv(cellule[in][XDMI+1], MPI_CHAR, DIM, haut, tag,  MPI_COMM_WORLD, &req[2]) ;
Irecv(cellule[in][0], MPI_CHAR, DIM, bas, tag, MPI_COMM_WORLD, &req[3]) ;
…
MPI_Waitall(4,req,sta) ;
```

Quelques optimisations

➔ Requêtes persistantes (15% de gain sur les petites requêtes)

```
MPI_Isend_init( …, &request)
MPI_Start / MPI_Startall
MPI_Request_free(request) ;
```

➔ Datatype (définir des types dynamiquement pour agréger des données)

Exemple : au lieu de découper le tableau du jeu de la vie en bande on le découpe en carré. Envoyer un à un les octets des colonnes est inefficace (messages de 1 octet), une amélioration est de regrouper « à la main » les colonnes, les datatypes permet d'automatiser cette opération :

Construction du type vertical

```
MPI_Datatype colonne_t;
MPI_Type_vector(XDIM, 1, YDIM + 2, MPI_CHAR, &colonne_t);
MPI_Type_commit(&colonne_t);

// transmettre les lignes
…
// transmettre les colonnes
MPI_Isend(&tab[out][1][1],1,colonne_t,Gauche,1,MPI_COMM_WORLD);
MPI_Isend(&tab[out][1][YDIM],1,colonne_t,Droite,1,MPI_COMM_WORLD);
MPI_Irecv(&tab[in][1][1],1,colonne_t,Gauche,1,MPI_COMM_WORLD,&etat);
MPI_Irecv(&tab[in][1][YDIM],1,colonne_t,Droite,1,MPI_COMM_WORLD,&etat);
// transmettre les coins
…
```

➔ Opérations collectives (réduction, distribution / regroupement de données)
pb calculer le nombre de cellules vivantes
Réduction

```
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
```

Root recevra le cumul cependant l'aspect bloquant peut entraîner une synchronisation entre les processus, afficher le cumul avec du retard...

Synchro d'un ensemble de processus

```
MPI_Barrier(communicator)
```

Diffusion d'une valeur

```
Bcast(&buffer,count,datatype,root,comm)

Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm)
Gather
Alltoall
```

➔ Le Ready send : lorsque l'on sait que le recv correspondant est déjà en attente
Exemple demande du jeton : Maitre / Esclave

Maitre :

```
Recv(&demande,1,MPI_CHAR,MPI_ANY_SOURCE,tag,
MPI_COMM_WORLD,&status)

Rsend(&jeton, 1, MPI_CHAR, status.MPI_SOURCE,
tag, MPI_COMM_WORLD)
```

Esclave :

```
Irecv(&jeton,maitre,&req)
Send(&demande,maitre)
Wait(&req,&status)
```

Permet de gagner une demande de rendez-vous

Les limites de l'interface MPI

→ trop de fonctionnalités (600 pages de documentation) : plus le modèle est complexe plus les implémentations sont erronées, peu performante
→ mais manque d'expressivité :

- QOS (priorité des messages, fiabilité)
- Construction incrémental limité (agrégation)
- Pas d'information quant aux seuil de performances : les utilisateurs sont obligés de découvrir eux mêmes les modes d'envois… => codage en dur rarement portable.

Limites des implémentations

→ Progression des communications => performances peu compréhensible par les utilisateurs.
- Pousser les paquets en appelant mpi_test si l'implémentation ne dispose pas de thread de progression ou n'embarque pas de code sur les cartes réseaux
→ Intégration problématique des threads

1 - Environnement
Initialiser MPI et quitter MPI :
int MPI_Init(int *argc,char*** argv)
int MPI_Finalize(void)
Quitter brutalement MPI :
int MPI_Abort(MPI_Comm comm)
Savoir si un processus à fait un MPI_Init :
int MPI_Initialized(int *flag)
Récupérer la chaine de caractères associée au code d'erreur err:
int MPI_Error_string(int errcode,char *chaine,int *taille_chaine)

2 - Communications point à point bloquantes
Envoyer un message à un processus :
int MPI_[R,S,B]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)
Recevoir un message d'un processus :
int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Status *status)
Envoyer et recevoir un message :
int MPI_Sendrecv(void *sendbuf,int sendcount,MPI_Datatype sendtype,int dest,int sendtag,void *recvbuf,int recvcount,MPI_Datatype recvtype,int source,int recvtag,MPI_Comm comm,MPI_Status *status)
Compter le nombre d'éléments reçus :
int MPI_Get_count(MPI_status *status,MPI_Datatype datatype,int *count)
Tester l'arrivée d'un message :
int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status)

3 - Communications point à point non bloquantes
Commencer à envoyer un message :
int MPI_I[r,s,b]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request * request)
Commencer à recevoir un message :
int MPI_Irecv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request)
Compléter une opération non bloquante :
int MPI_Wait(MPI_Request *request,MPI_Status *status)
Tester une opération non bloquante :
int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status)
Libérer une requête avant de la réutiliser :
int MPI_Request_free(MPI_Request *request)
MPI_Test checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.
int MPI_Testany(int count,MPI_Request *array_of_requests,int *index, int *flag,MPI_Status *status)
int MPI_Testall(int count,MPI_Request *array_of_requests, int *flag,MPI_Status *array_of_statuses)
int MPI_Testsome(int incount,MPI_Request *array_of_requests,
int *outcount,int *array_of_indices, MPI_Status *array_of_statuses)
int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag, MPI_Status *status)
int MPI_Cancel(MPI_Request *request);
MPI_Waitall(), MPI_Cancel MPI_Iprobe(),
MPI_Test_cancelled()

4 - Communications persistantes
Décrire un schéma persistant :
int MPI_[R,S,B]send_init(void *sendbuf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)
int MPI_Recv_init(void *recbuf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)
Démarrer une communication persistante :
int MPI_Start(MPI_Request *request)
Autres fonctions : MPI_Startall(), MPI_Request_free()

5 - Communications collectives
Diffusion générale d'un message :
int MPI_Bcast(void *buf,int count,MPI_Datatype datatype,int root,MPI_Comm comm)
Collecte de données :

int MPI_Gather(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype ,int root,MPI_Comm comm)
Diffusion sélective d'un message :
int MPI_Scatter(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype ,int root,MPI_Comm comm)
Collecte de données et rediffusion :
int MPI_Alltoall(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm)
Calcul d'une réduction :
int MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op operation,int root,MPI_Comm comm)
Calcul d'une réduction et rediffusion du résultat :
int MPI_Allreduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op operation,MPI_Comm comm)
Synchronisation de processus :
int MPI_Barrier(MPI_Comm comm)

6 - Types dérivés
Construire un type de données contigües :
int MPI_Type_contiguous(int nbre,MPI_Datatype ancien_type,MPI_Datatype *nouveau_type)
Type de données distantes d'un pas constant :
int MPI_Type_[h]vector(int nbre,int taille_bloc,MPI_Aint pas,MPI_Datatype ancien_type,MPI_Datatype *nouveau_type)
Type de données distantes d'un pas variable :
int MPI_Type_[h]indexed(int nbre,int *taille_bloc,MPI_Aint *pas,MPI_Datatype ancien_type,MPI_Datatype * nouveau_type)
Contruire un type structuré :
int MPI_Type_struct(int nbre,int *taille_bloc,MPI_Aint *pas,MPI_Datatype *anciens_types,MPI_Datatype *nouveau_type)
Valider un type :
int MPI_Type_commit(MPI_Datatype *dataype)
Routine portable pour retourner l'adresse d'une variable :
int MPI_Address(void *variable, MPI_Aint *adresse)
Autres fonctions :
MPI_Type_free, MPI_Type_extent, MPI_Type_size(), MPI_Type_[u,l]b()

7 - Constantes
Jokers :
MPI_ANY_TAG, MPI_ANY_SOURCE
Datatypes élémentaires :
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_UNSIGNED, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT ,MPI_UNSIGNED_LONG, MPI_LOGICAL, MPI_BYTE, MPI_PACKED
Constantes reservées :
MPI_PROC_NULL, MPI_UNDEFINED
Communicateurs reservés :
MPI_COMM_WORLD, MPI_COMM_SELF
Opérateurs de MPI_Reduce et MPI_Allreduce :
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR

8 – MPI_Status : status.source status.tag status.error status.length status.size status.bytes

Programmation distribuée implicite
DSM Distributed Shared Memory – MVP Mémoire Virtuellement Partagée

Une DSM est un système logiciel donnant l'illusion au programmeur de disposer d'une machine à mémoire commune au-dessus d'un système à mémoire distribuée. L'objectif des DSM est de faire de gagner sur le temps de développement, sur la lisibilité des programmes voire sur la qualité du programmeur. Il est clair que l'approche implicite ne permet pas d'atteindre les performances de l'approche explicite lorsque les communications représentent une part non négligeable du temps de calcul.

Différents types de DSM : DSM à base de pages – DSM d'objets - SSI (openMOSIX, kerrighed). Intel a lancé Cluster OpenMP qui permet d'utiliser OMP sur une grappe via une DSM. Notons aussi qu'une machine NUMA est une DSM réalisée matériellement.

Dans ce chapitre, on va s'intéresser aux mécanismes et algorithmes nécessaires à la réalisation d'une DSM fort simple mais peu performante puis on va relâcher certaines contraintes de cohérence pour faire quelques optimisations.

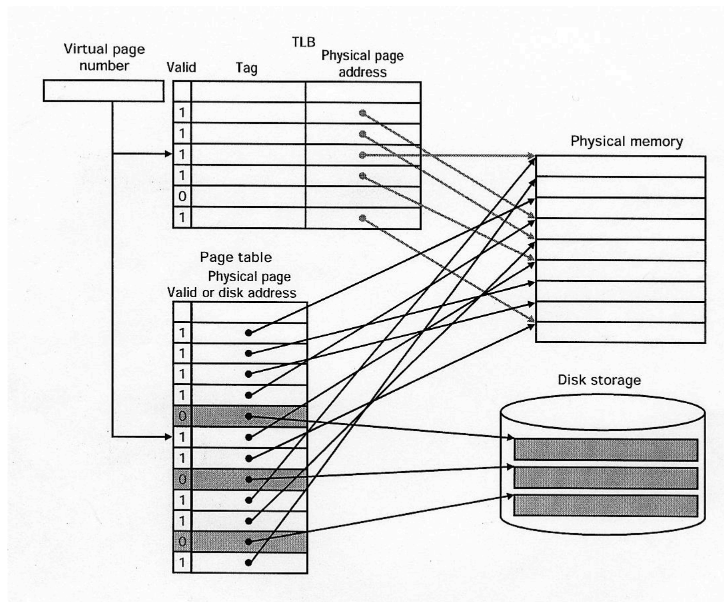
DSM à base de pages (SVM 1986)
Une DSM peut être implantée avec ou sans support matériel dédié au niveau du noyau (performance) ou au niveau d'une bibliothèque (souplesse).

La solution la plus simple est d'utiliser le système de pagination disponible sur les machines… c'est en effet l'unité mémoire gérée par au niveau du matériel. Par exemple l'appel système mmap permet de placer la mémoire où l'on veut (si l'espace mémoire visée est virtuellement libre) et donc il est facile d'avoir un système d'adressage virtuel cohérent sur un ensemble de machine: il suffit d'utiliser de façon cohérente les mmaps. (Dessin : à une adresse virtuelle correspond autant de case mémoire qu'il y a de machines.)

Le déterminisme du code généré par le compilateur fait que les adresses des variables globales sont les mêmes d'une machine à une autre, il suffit alors d'instrumenter la bibliothèque d'allocation pour faire en sorte que chaque machine alloue dans un espace propre (chacune à son tas). Donc le seul vrai problème est d'assurer la cohérence de la mémoire du système…

Une version simple : faire en sorte qu'une page ne soit disponible que sur une seule machine à la fois. Lorsqu'une machine accède à une donnée, il faut déterminer si elle en est le propriétaire et autrement qu'elle le devienne.

Idée de mise œuvre : utiliser la mmu pour invalider les pages dont la machine n'est pas le propriétaire. L'appel système mprotect() permet de gérer les droits rwx des pages virtuelles. Il suffit donc de supprimer les droits rw des pages dont on n'est pas propriétaire. Lorsque le processus accèdera à une page invalide, il recevra un signal syssegv. Rappelons que lorsqu'on a une interruption matérielle l'instruction fautive est retentée à la fin du traitement de l'interruption… donc si on traite le signal en allant chercher la page chez son propriétaire (qui invalidera à son tour sa copie) on a gagné. En fait, c'est exactement le mécanisme du *swap* (c'est même une façon de faire du swap haute performance).



TLB à un niveau
http://users.ece.gatech.edu/~dblough/3055/

Dans ce cadre il suffit donc d'ajouter un mécanisme (données + protocole) pour pouvoir déterminer le propriétaire de la page, voici différentes techniques :

→ diffusion des requêtes : on « crie » sur le réseau et la machine détentrice répond
- risque saturation du réseau et occupation inutile des cpu distants

→ centralisation : la connaissance du placement des pages :
- un gestionnaire dispose de toutes les informations
- risque de saturation du serveur

→ distribution de la connaissance du placement (notion de home node)
- distribuer de façon cyclique (une fois pour toute) la responsabilité des pages
- permet en deux requêtes d'atteindre la page

Optimisations
Les performances peuvent se dégrader lorsque plusieurs processus se « disputent » une même page. Une première optimisation est de s'adresser au propriétaire « probable » : on mémorise le nom du processus à qui l'on a envoyé la page. On peut chaîner les propriétaires probables… cependant il y a un risque d'avoir une requête non satisfaite (il faut que la requête rattrape la page). Une possibilité est de renvoyer la requête au gestionnaire. Une autre optimisation concerne les effets ping-pong pour de simples lectures : il s'agit de protéger en « lecture seule » les pages à l'aide de l'appel système mprotect(). Avant de changer de modifier une page, il faut prévenir les autres détenteurs exactement comme dans MSI.

Affaiblir le protocole de cohérence de la mémoire distribuée
Une limite de la technique précédente est celle du faux partage : les problèmes d'écritures en // sur une même variable ne nous intéressent pas ici – c'est au programmeur d'optimiser son programme. Dans les DSM basée sur des pages la granularité est fixée par le système d'exploitation, pour régler les pb de faux partage il s'agit donc de les réduire au niveau de l'application (ou de sa compilation) ou bien de ne pas imposer la consistance séquentielle au niveau des pages, d'utiliser des modèles de consistance moins forte. Notre objectif, c'est d'éviter le faux partage pour optimiser des programmes très bêtes du genre

for(i=0 ; i < 100 ; i++) A[2*i] = A[2*i] + 1;

for(i=0 ; i < 100 ; i++) A[2*i+1] = A[2*i+1] + 1;

En cohérence séquentielle on peut observer un ping–pong… Pour assurer une consistance plus lâche on va utiliser les événements de synchronisation pour rendre la mémoire cohérente au bon moment. En effet il existe des modèles de consistance moins forte que la consistance séquentielle mais bien adaptés aux programmes // non bogués (sans erreur de synchronisation).

Ordre causal (Lamport) est l'ordre lâche le plus facile à mettre en œuvre : il s'agit de respecter l'ordre entre les événements *dépendants.*

b dépends directement de a

- si b est exécuté après a par le même thread
- si b est une réception correspondant à l'émission a

La relation « dépend » fermeture réflexive et transitive de « dépends directement »

Exemple x = y = 0

```
W(x)1
W(y)2
        R(X)1  R(Y)0
        R(Y)2  R(X)0
```

Un tel résultat est impossible en cohérence séquentielle… autre exemple :

```
W(x)1
        R(x)1 W(y)2
                R(y)2  R(x) 1  (imposé)
        R(x) 1                 R(y) 0 (non imposé)
```

Mise en œuvre de la cohérence faible

Quand synchroniser ?

Pour mettre en œuvre une DSM basée sur l'ordre causal on profite des synchronisations introduites par le programmeur pour resynchroniser la mémoire. NB cette technique ne marche pas pour les *lock free algorithms* qui reposent sur l'utilisation des protocoles de cohérence de cache à la MESI.

Au niveau des barrières

P1	P2

…
For()… A[2*i] =
Barriere(b) ;
For()… x = A[2*i+1]
	…
	For() … A[2*i+1] =
	Barriere(b) ;

Ici on profite de la barrière pour rendre cohérente la mémoire, en plus de la barrière, il faut que chaque processus envoie ses propres modifications et attende d'avoir reçu toutes les modifications pour continuer.

Que faire pour des mécanismes de synchronisation plus souples ? (lock / unlock) Une idée simple faire comme dans CVS – SVN : individualiser les événements, distinguer la phase de lecture et celle d'écriture.

Acquire() : mettre à jour sa mémoire / *Release*() : diffuser ses modifications
Par exemple, il est indispensable d'appeler Acquire() en entrée d'une section critique et de bon ton d'appeler Release() en sortie de la section critique (comme les méthodes synchronized en java).

Quand envoyer concrètement les données ?
approche gloutonne « Eager Release Consistency » → release diffuse les données systématiquement des communications inutiles génèrent un trafic important.

approche paresseuse « Lazy Release Consitency » → acquire va chercher les pages, latence importante. (solution généralement adoptée)

Comment détecter et prendre en compte les modifications ?

→ Mettre les pages de la DSM en Read Only et passer en RW lors de l'écriture. On utilise le traitement du signal sigsegv pour maintenir à jour une table des pages modifiées et n'envoyer que celles-là. (voir TreadMarks http://infoscience.epfl.ch/record/55539/files/computer96.pdf)

Cela peut coûter cher d'envoyer beaucoup de pages alors que seuls quelques octets par page ont été modifiés… une optimisation possible : envoyer seulement les différences… dans le traitant on enregistre une copie de la page que l'on va modifier (technique des *patchs*).

Mais il s'agit de calculer une bonne version… Pour CVS, c'est facile car système centralisé et disposant d'un numéro de version de on gère les conflits à la main avant d'envoyer ses modifications.

On ne peut pas attendre que chacun ait terminé ses acquires/release ou que chacun soit sorti de toutes les sections critiques. Ce qui est important c'est que la cohérence causale soit respectées modulo les release réalisés (Si P1 sait que P2 a fait un release ce release doit apparaître) et que la cohérence séquentielle soit imposée par le programmeur au moyen des verrous.

→ Solution centralisée : le gestionnaire de la page réalise le acquire en allant chercher les modifications chez les différents clients de la page, réalise la synthèse puis diffuse la page. (Cela peut faire un truc joli en RDMA)

→ Solution décentralisée : le processus crie sur le réseau… tout le monde doit répondre, il s'agit de déterminer la dernière version de chaque information…

Idée : un processus ne doit transmettre ses modifications qu'une seule fois… il s'agit de faire le cumul des modifications… On peut conserver un numéro de version pour chaque

information… le processus diffuse le numéro de version qu'il a pour chaque information afin de minimiser les envois…

Bref ici on a un problème de granularité… Le compilateur pourrait nous aider…c'est le cas pour le langage Orca (Pays Bas) où la granularité est l'objet… voir http://pdos.csail.mit.edu/~kaashoek/papers/tse.ps

UPC unified parallel C (Famille *Partitioned Global Address Space Languages)*

Approche basée sur threads et mémoire partagée distribuée

```
int main() {  printf("Hello from thread %i/%i\n", MYTHREAD, THREADS);  upc_barrier;  return 0;}
```

Propose des variables partagées :

```
shared int i ;
shared int t[THREADS] ;
shared int tt[10][THREADS] ; // répartition cyclique
#define N 100*THREADS
shared int a[N], b[N], c[N];
void main()
{
int i;
upc_forall(i=0; i<N; i++; i)
    a[i]=b[i]+c[i];
}

…
upc_forall (i=0; i<100; i++; &a[i])
    a[i] = b[i] + c[i];
…

shared int a[THREADS][THREADS] ;
shared int b[THREADS], c[THREADS] ;
void main (void) {
int i, j;
upc_forall( i = 0 ; i < THREADS ; i++; i) `
 {
    c[i] = 0;
    for ( j= 0 ; j
    c[i] += a[i][j]*b[j];
 }
}
shared [THREADS] int a[THREADS][THREADS];
// répartition cyclique par bloc de THREADS

exemple plus compliqué: shared [3] int A[R][THREADS]
```

Cohérence mémoire dans UPC

Pour un bloc → pragma UPC strict/relaxed   Pour une variable

strict shared → cohérence séquentielle
relaxed shared → cohérence relâchée

la mémoire est synchronisée lors

- d'une écriture ou d'une lecture d'une variable partagée à cohérence séquentielle (lecture = acquire , écriture = release)
- d'une barrières upc_barrier expr ; upc_notify expr ;
- d'une demande de synchronisation upc_fence ;

Comment ça marche ?
Pointeur : phase, thread, adresse virtuelle
Prétraitement du source :
utilisation de macro pour transformer les références aux variables partagées par des appels de fonctions,

Coût important → upc_memcpy() memput() memget() pour minimiser le coût des traductions

PB Optimisation à la main → ressemble à mpi
Pour conclure notons que les DSM et les PGAS ont du mal à trouver leur marché entre les OpenMP sur des machines multicœurs et MPI sur grappe. Pourtant il y a clairement un marché à prendre : MPI + OpenMP ne semble pas s'imposer pour la programmation des grappes de multicoeurs.

Programmation des Architectures Parallèles
Examen du mardi 15 avril 2008 Durée : 1h30 – Sans document

Prédiction de branchement
Rappeler les différentes stratégies utilisées dans les microprocesseurs modernes pour tenter de prédire si un branchement sera pris ou non. Comparer leur efficacité sur l'exemple suivant :

```
for(i=0 ; i < 1000 ; i++)
    for(j=0 ; j < 10 ; j++)
        if (j & 1) // i est impair
          k++ ;
```

Décrire une expérience permettant d'apprécier la qualité de l'unité de prédiction d'un processeur.

Programmation OpenMP
Il s'agit de paralléliser le plus efficacement possible la boucle suivante (en modifiant au besoin le code):

```
for(i=0 ; i < 1000 ; i++)
  s += f(i) ;
```

3. En supposant que le temps de calcul de f(i) ne dépends pas de la valeur de i ;
4. En supposant que le temps de calcul de f(i+1) est toujours (très) supérieur à celui de f(i).

Programmation MPI
On désire paralléliser *efficacement* le traitement d'une séquence d'images en utilisant une grappe de pc et en adoptant un schéma de programmation de type maître/esclave. Le maître distribue la séquence image par image aux esclaves inactifs ; le travail (répétitif) d'un esclave consiste à réceptionner une image, la traiter et la retourner au maître ; le maître remplace la version originale de chaque image par celle traitée.
Écrire le pseudo code du programme en supposant qu'une image est codée par 3 tableaux (RVB) de 1024*768 octets et que l'on dispose de N esclaves. On précisera bien les appels MPI.

Programmation des architectures distribuées
Expliquer pourquoi la communauté des programmeurs parallèles préfère actuellement l'approche de programmation de type MPI pure aux approches hybrides telles MPI + OpenMP ou MPI + pthread en matière de programmation des grappes de multiprocesseurs.

Programmation des Architectures Parallèles

Examen du jeudi 23 avril 2009 Durée : 1h30 – Sans document

Instructions atomiques

Le jeu d'instructions des processeurs contemporains multicores contient une version atomique de quelques instructions simples comme ADD, AND, CMPXCHG, DEC, INC, SUB, XOR, XCHG,… Grâce à ces instructions on peut réaliser des verrous en espace utilisateur, comme l'illustre le verrou suivant basé sur la fonction `GETLOCK()` qui elle même repose sur une version atomique de CMPXCHG.

```
if (GETLOCK(lock) != 0)
{
  while (1)
 {
   if (*(volatile int*)lock) == 0)
   {
     if (GETLOCK(lock)) != 0)
     {
       goto PROTECTED_CODE;
     }
   }
 }
 BACK_OFF_LOCK(lock); // perdre du temps ou passer la main
}
PROTECTED_CODE:
```

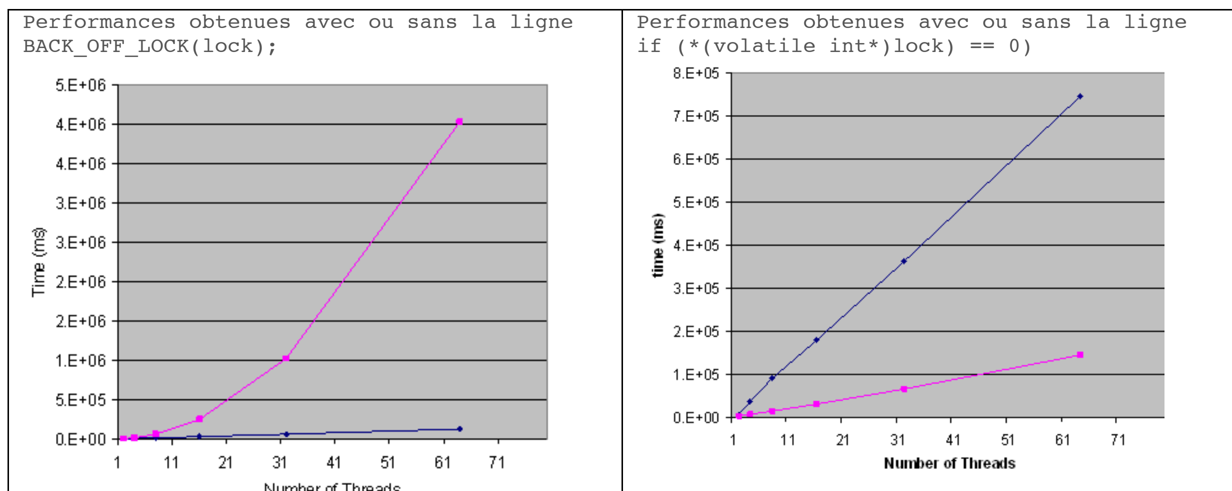
1) Comment peut-on indiquer au compilateur OpenMP d'utiliser une instruction atomique ?

2) Exposer les techniques permettant la mise en œuvre matérielle des instructions atomiques.

En vous appuyant sur les deux courbes suivantes expliquer l'origine des gains de performance obtenus par une application utilisant intensivement un verrou implémenté comme ci-dessus :

3) Via l'introduction de la procédure `BACK_OFF_LOCK()` qui fait *à priori* perdre du temps (tours de boucle à vide ou passe la main à un autre thread si les appels à cette fonction sont nombreux et rapprochés).

4) Via l'introduction d'un test à priori inutile sur la variable pointée par lock.



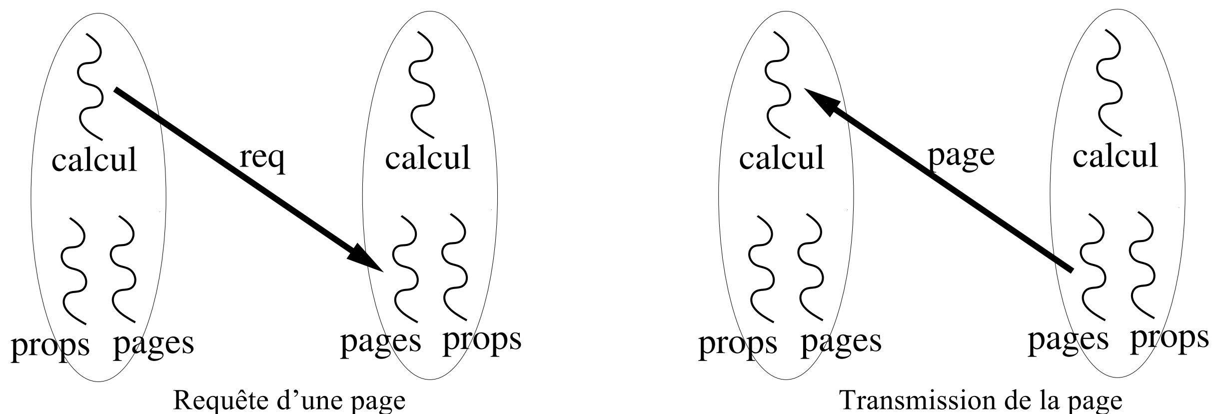
Programmation OpenMP

Il s'agit de proposer un programme parallèle OpenMP qui place dans une matrice d'entiers C[N][N] la somme de deux matrices A[N][N] et B[N][N] et qui place dans la variable NULS le nombre d'éléments nuls de la matrice résultat C.

5) Donner deux versions de code parallèle : l'une simple et l'autre la plus optimisée possible.

Programmation MPI

Il s'agit de réaliser les communications nécessaires à la réalisation d'une DSM à base de pages. Cette DSM sera fort simple puisque les pages ne seront pas dupliquées (un seul propriétaire à un moment donné) et on supposera qu'il n'y a qu'un thread de calcul par processus (un seul défaut de page à la fois par processus). En plus du thread de calcul chaque processus est équipé de deux threads de service : un premier chargé de la communication des pages et un second chargé de la gestion des propriétaires des pages. Lors d'un défaut de page, le thread de calcul est interrompu et le traitant `dsm_handler_readwrite()`avec l'adresse fautive. Le thread de calcul émet alors une requête de page au processus propriétaire de la page. Cette requête est réceptionnée par le thread serveur des pages qui, en réponse, émet la page réclamée vers le processus. C'est le thread de calcul qui assurera la réception de la page. Une fois sorti du traitant, le thread de calcul peut poursuivre son travail.



Requête d'une page

Transmission de la page

On dispose des fonctions suivantes :

```
#define PAGE_SIZE 4096
typedef char* page ;

int dsm_mon_rang ;
// rang du processus par rapport à MPI_COMM_WORLD

page *dsm_adresse_de_page(void *adresse_fautive) ;
//retourne l'adresse de base de la page contenant l'adresse fautive

void sys_proteger(page p, int droits) ;
// NUL = aucun droit – RO = lecture seule – RW = lecture ecriture

// Fonctions gérant de façon distribuée le propriétaire d'une page donnée
// ces fonctions utilisent le tag MPI DSM_TAG_PROPRIETAIRE de valeur 555
// et n'interfèrent donc pas avec les communications sur les autres TAG

int dsm_obtenir_proprietaire(page p) ;
// retourne le rang du propriétaire officiel de la page par rapport à MPI_COMM_WORLD

int dsm_definir_proprietaire(page p, int proprietaire) ;
// positionne le rang du propriétaire officiel de la page par rapport à MPI_COMM_WORLD
```