Annexe: aide-mémoire, ces pages ne seront pas corrigées

Équations

Loi d'Amdahl

$$A(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

Probabilité k sur n fonctionnels

$$p = \binom{n}{k} \times p_d^k \times (1-p_d)^{n-k}$$

Probabilité p de deux événements **indépendants** p1 et p2: p = p1 x p2

Barrière Papillon

chaque thread communique avec un autre à chaque étape (i+1 % n, i+2 % n, i+4 % n...). Tous reçoivent l'information que tous sont prêts en même temps. Pas de phase d'annonce de fin!

Verrous RCU

Structures qui utilisent un pointeur en entrée (liste). Surtout des lectures. Lors d'un changement, atomiquement enlever le pointeur à l'ancienne donnée. Lorsque tous les lecteurs sortent, libère la mémoire.

Opération sur les matrices

$$A + B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a+e & b+f \\ c+g & d+h \end{pmatrix} = C$$

$$A - B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} - \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a-e & b-f \\ c-g & d-h \end{pmatrix} = C$$

$$k \times \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} ka & kb \\ kc & kd \end{pmatrix}$$

$$a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31} = c_{11}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$D = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix} \Rightarrow D^{-1} = \begin{bmatrix} 1/d_1 & 0 & \cdots & 0 \\ 0 & 1/d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/d_n \end{bmatrix}$$

Algorithme de Jacobi

itératif

$$x^{(k+1)} = D^{-1}(b - Ox^{(k)})$$

Où D diagonale de A, O=A-D

Mémoire

Endroits où l'ordre peut changer : compilateur, pipeline, multiprocesseurs

Modèle séquentiel: Attendre propagation du write, ou attendre que les write de partout soient effectifs avant de faire read ou write suivant. Trop contraignant.

Total Order: Read avant écriture

Partial Order: Accès à var différentes réordonnancés par rapport aux write

Weak ordering: accès à var différentes réordonnancés

Barrières mémoires

Pleine : smp_mb() accès effectués avant tous ceux qui suivent

Pour lecture : smp_rmb() read effectués avant read qui suivent

Pour écriture : smp_wmb() write effectués avant write qui suivent

Lectures dépendantes : smp_read_barrier_depends() Read effectués avant read dépendants qui suivent

Lorsque l'on utilise des barrières mémoires, il faut des barrières de chaque côté!!!

Valgrind

Memcheck: Enregistrement de l'état de chaque bit. 10 à 30x plus lent.

Helgrind : 100x plus lent.

Cachegrind : Deux niveaux de cache L1 et LL (last-level). 50x plus lent.

Callgrind: Permet de voir l'arbre d'appel ie A->B->...->Z

Massif : Head profiler

Google

Thread Sanitizer: Division en segment de synchronisation. 2 à 20x plus lent, 5 à 10x plus de mémoire. Garde les quelques derniers accès.

Address Sanitizer: Enregistrement de l'état de chaque bloc de 8 octets. 2x plus lent.

CPU profile : échantillonnage dans le temps de la chaine d'appel

Promela/Spin: simuler des programmes, de quelques dizaines de lignes, pour lesquels l'ordre d'exécution n'est pas prévisible.

Lockdep : graphe d'ordre de prise des verrous par variable et par structure (inode, page...). Vérification du contexte (normal ou interruption).

Gcov: nb de fois que chaque ligne est exécutée. Surcout 10 à 15%

GProf : nb d'appels à chaque fonction avec provenance. Échantillonnage dans le temps. Hypothèse: les différents appels prennent le même temps, donc le temps self+childs n'est pas très fiable

Oprofile: Échantillonnage des instructions avec les compteurs matériels. Surcoût < 0.5%

Algo Multiplication de matrices (envoi des données seulement lorsque nécessaire)

Si p nœuds, décomposer matrice nxn en matrices de taille $m = n/\sqrt{p}$

```
1  iup  = i+1 mod m;
2  idown = i-1 mod m;
3  for (k = 0; k < m; k++) {
4       km = (i+k) mod m;
5       broadcast(A[i,km]) to all nodes handling row i of C;
6       C[i,j] = C[i,j] + A[i,km]*B[km,j]
7       send B[km,j] to the node handling C[idown,j]
8       receive new B[km+1 mod m,j] from the node handling C[iup,j]
9  }
```

TBB

parallel_for(blocked_range<size_t>(0,n), MyClosure(a));
parallel_reduce(blocked_range<size_t>(0, n), MySum(a));
parallel_reduce Utilise une Structure en arbre
parallel_scan(blocked_range<int>(0,n), MyScan(a, b));
parallel_do(_rst, last, MyClosure);
parallel_sort(begin, end);

Exemple de code (parallel_reduce)

```
struct Sum {
   float value;
   Sum() : value(0) {}
   Sum(Sum& s, split) { value = 0; }
   void operator()(const blocked_range<float*>& r) {
      float temp = value;
      for(float* a=r.begin(); a!=r.end(); ++a) temp +=
*a;
         value = temp;}
   void join(Sum& rhs) {value += rhs.value;}
};
float ParallelSum(float array[], size_t n) {
   Sum total;
parallel_reduce(blocked_range<float*>(array,
array+n),total);
   return total.value;}
```

Exemple de code (parallel_pipeline)

```
float RootMeanSquare( float* first, float* last ) {
   float sum=0;
   parallel_pipeline(16,
      make_filter<void,float*>(filter::serial,
         [&](flow_control& fc)-> float*{
         if( first<last ) return first++;
         else { fc.stop(); return NULL; }
      }) &
      make_filter<float*,float>(filter::parallel,
      [](float* p){return (*p)*(*p);}) &
      make_filter<float,void>(filter::serial,
      [&](float x) {sum+=x;})
  );
  return sqrt(sum);}
```

OpenMP

Vars déclarées avant *omp parallel* sont globales
Vars déclarées après *omp parallel* sont locales
Omp single un seul thread execute cette partie
Barrière implicite à la fin des blocs, *nowait* enlève la barrière
Flush de la mémoire implicite à la fin des blocs

OpenCL

Synchronisation

Marqueur pour savoir lorsque les commandes précédentes d'une queue sont terminées.
Barrière pour assurer que toutes les commandes précédentes sont faites avant de commencer les suivantes.
void barrier (cl_mem_fence__ags _ags): synchroniser tous les workItem du workGroup.
void mem_fence (cl_mem_fence__ags _ags) barrières mémoires pour mémoire du workItem.

MPI

MPI_Wait est bloquant | MPI_Test est non bloquant

```
int MPI_Type_create_struct(int count, int blocklengths[],
   MPI_Aint displacements[], MPI_Datatype types[],
   MPI_Datatype *newtype);
Int MPI_Type_commit(MPI_Datatype *newtype);
```