

# Calcul parallèle pour la recherche des QTL

O.Filangi

Séminaire Rules & Tools - La rochelle  
24/09/2010

# Pourquoi?

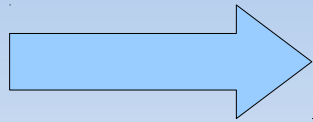
- **Utilisation des SNP**
  - Changement d'échelle [100 MS => 50 000 SNP]
  - Plus de données à traiter (démocratisation du séquençage)
  - Densité de l'information
  - Génération des données croît plus vite que la vitesse des processeurs
- => **Conséquences sur l'analyse**
  - Echantillonnage des analyses de liaisons élevé
  - Nouvelles analyses complexes (interaction, qtl additifs, lola)

**Objectif : Réduire le temps d'exécution (~ revenir à des temps convenables)**

# Critères de parallélisation

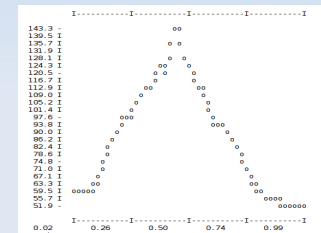
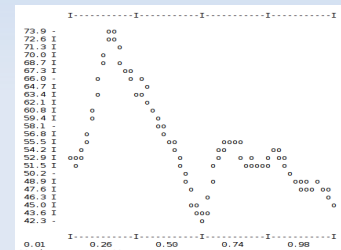
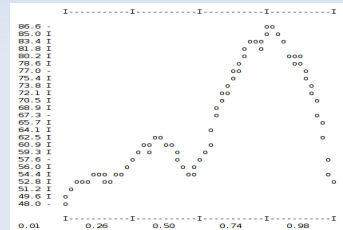
- **Gros grain**
  - Analyses uni-caractère
  - La position du calcul de vraisemblance
  - Analyses avec des données simulées pour l'estimation de seuil de rejet
  - Probabilités de transmissions de chaque F2
- **Grain fin**
  - Calcul matriciel
  - Resolution d'un systeme  $N[\text{position}] \times M[\text{simulation}]$

# Parallélisation (1)

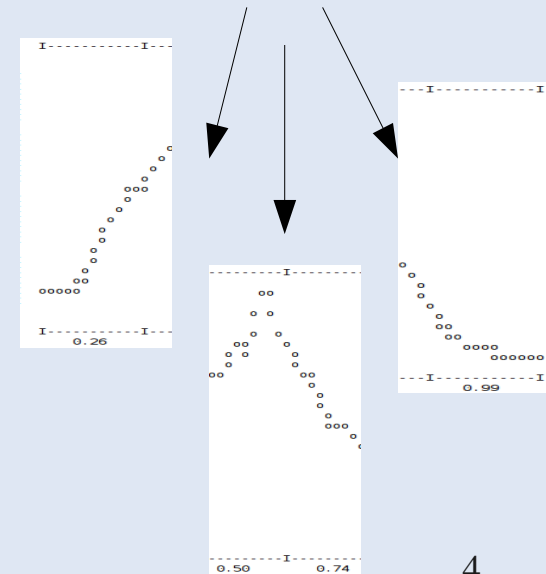


qtlmap

Parallélisation  
des analyses (eQTL)



Parallélisation  
du génome scan



Contexte multiprocesseurs – architecture à mémoire partagée  
=> OpenMP

# OpenMP-QtLMap

- **Parallélisation à gros grain**
  - Language Fortran
  - Adapté au code séquentiel
  - 1997 => implémenté par beaucoup de compilateur
- **Avantages**
  - Peu intrusif : l'utilisation de **pragma** seul (sans utilisation de l'API [use omp\_lib]) permet de compiler sans prise en charge de openMP
  - Très peu de modification du code existant
  - Support des variables globales (**threadprivate**) mais peu posés des problèmes
  - Transparents pour les utilisateurs (intervention faible)
    - Export OMP\_NUM\_THREADS = <nb proc>
    - Export OMP\_NESTED = <true/false>

# Retours d'exp./Problèmes

- Attention aux librairies ad-hoc truffé de variables globales => **non thread-safe**
- Résultats sensiblement différents avec les méthodes d'optimisation du calcul de vraisemblance
- Exemple problématique de l'imbrication de boucles parallèles et des variables globales **thread private** :

```
Subroutine analyse
!$omp parallel default(shared)
!$omp do
Do ic=1,ncar
  Call optimization_H0
  Call optimization_H1
End Do
!$omp end do
!$omp end parallel
End subroutine analyse
```

```
!valeurs des vraisemblances des familles de plein-freres
real ,dimension(:),allocatable, private :: fm0
!$omp threadprivate (fm0)

Subroutine optimization_H0
  <Initialisation de fm0 >
End Subroutine optimization_H0

Subroutine optimization_H1
!$omp parallel default(shared)
!$omp do
Do npos=1,nbpositions
  <Utilisation de fm0>
End do
!$omp end do
!$omp end parallel

End Subroutine optimization_H1
```

# Problèmes (suite)

Solution peu élégante mais qui marche....

```
!valeurs des vraisemblances des familles de plein-  
freres  
real ,dimension(:), private , pointer :: fm0  
!$omp threadprivate (fm0)  
  
Subroutine optimization_H0  
    <Initialisation de fm0 >  
End Subroutine optimization_H0  
  
Subroutine optimization_H1  
    real,dimension(:), pointer :: fm0_buf  
    fm0_buf => fm0  
    !$omp parallel default(shared)  
    fm0 => fm0_buf  
    !$omp do  
    Do npos=1,nbpositions  
        <Utilisation de fm0>  
    End do  
    !$omp end do  
    !$omp end parallel  
  
End Subroutine optimization_H1
```

# Performances OpenMP

- 1 Dell Precision – Intel Xeon (E5410) quad core 2,33Ghz (**machine personnelle**)
- 2 DGA8 – Intel Xeon (E7450) 24 coeurs 2,40Ghz (**serveur de calcul du ctig**)

2 caracteres simulés

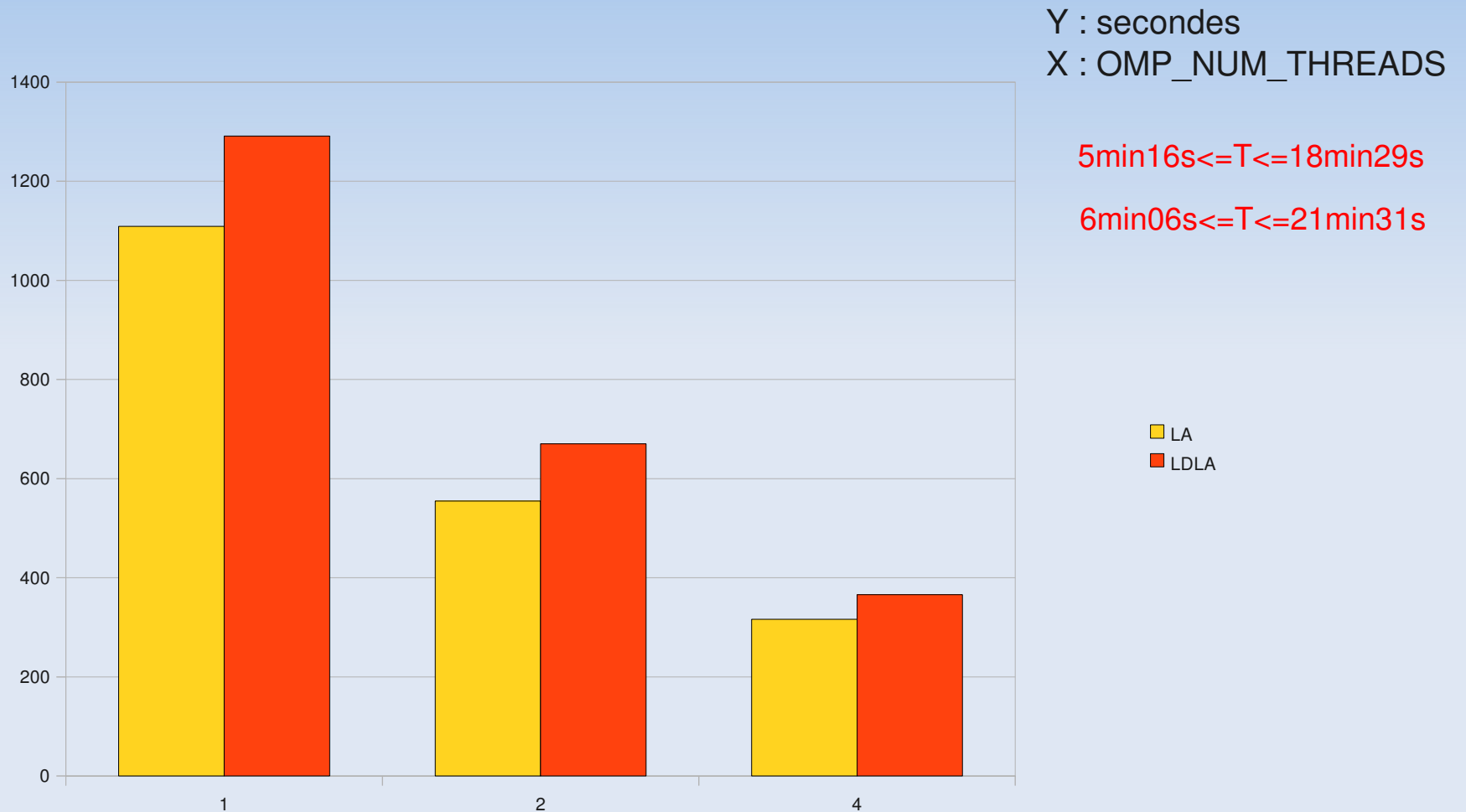
5000 markers

5 peres, 15 meres, 750 F2

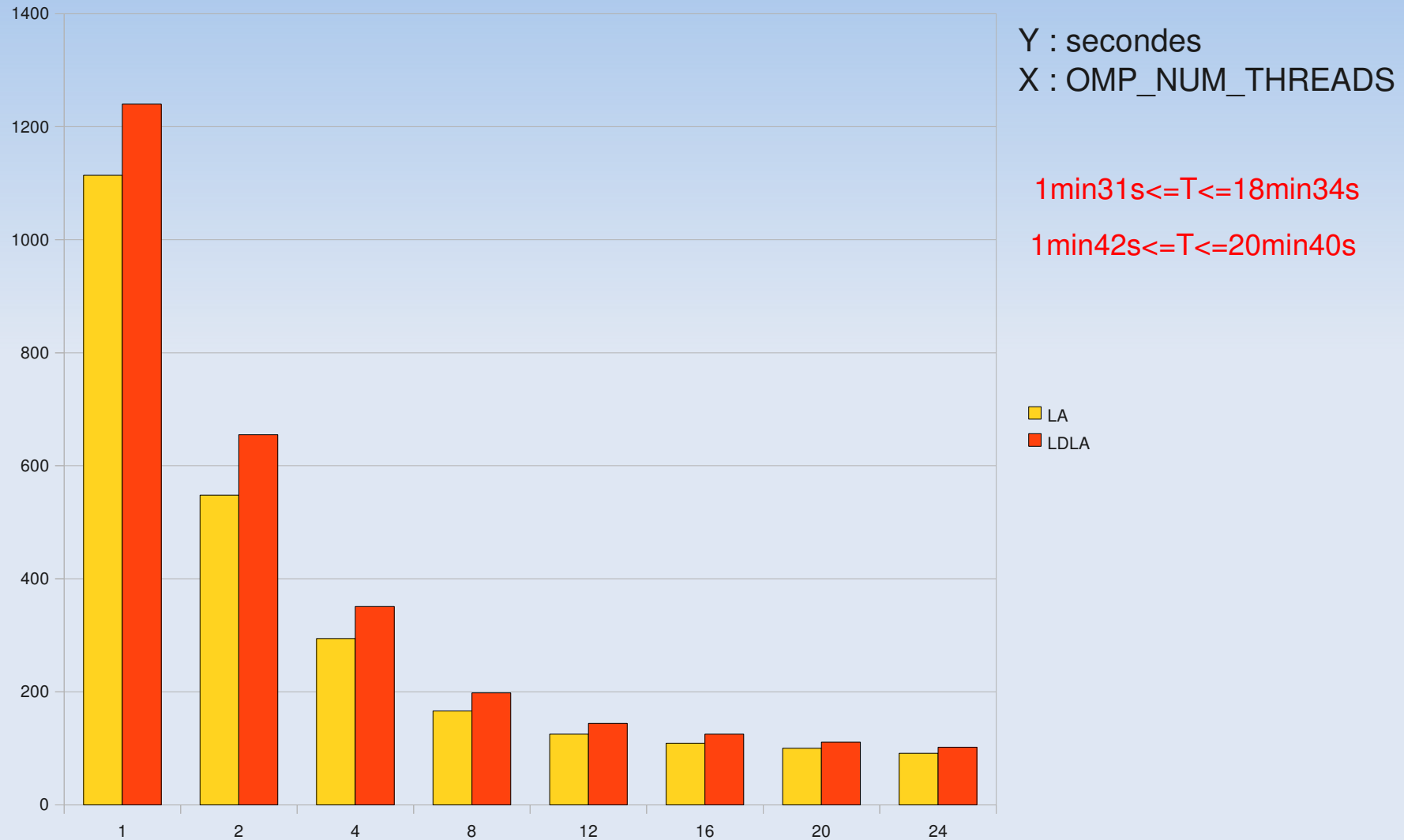
Pas 0,001 Morgan => 10000 positions à tester



# Analyse 1 caractere (1)



# Analyse 1 caractere (2)



2 DGA8 – Intel Xeon (E7450) 24 coeurs 2,40Ghz

# Analyse 2 caracteres (1)



Remarques :

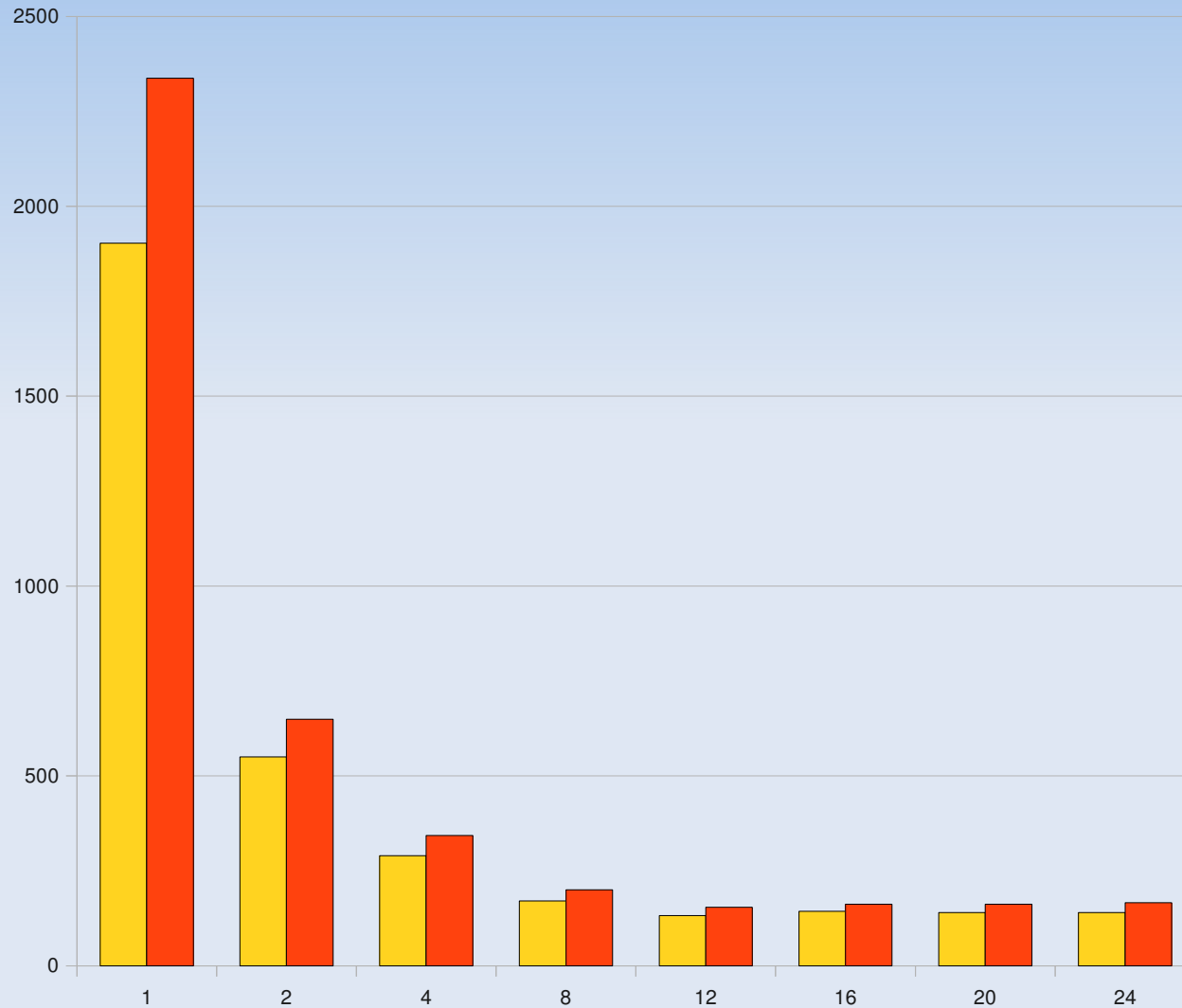
1) export OMP\_NUM\_THREADS=2

=> 2 analyses simultanées de deux caracteres

=> parallélisation du génome scan en 2<sup>1</sup>

Dell Precision – Intel Xeon (E5410) quad core 2,33Ghz

# Analyse 2 caracteres (2)



Y : secondes

X : OMP\_NUM\_THREADS

2min20s<=T<=31min43s

2min46s<=T<=38min57s

LA  
LDLA

2 DGA8 – Intel Xeon (E7450) 24 coeurs 2,40Ghz

Remarques :

1) export OMP\_NUM\_THREADS=2

=> 2 analyses simultanées de deux caractères

=> parallélisation du génome scan en 2

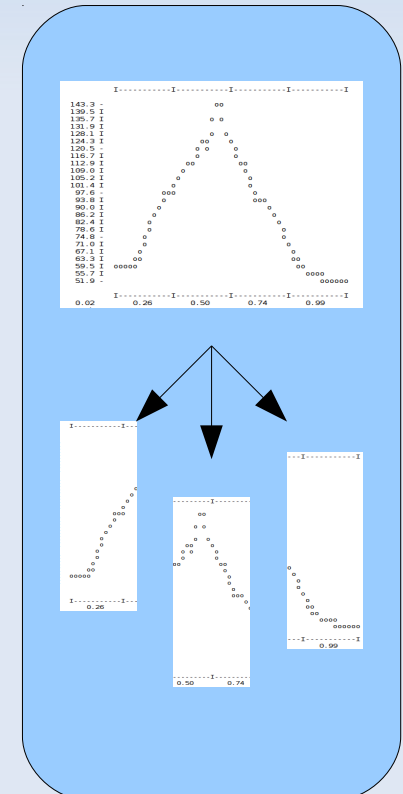
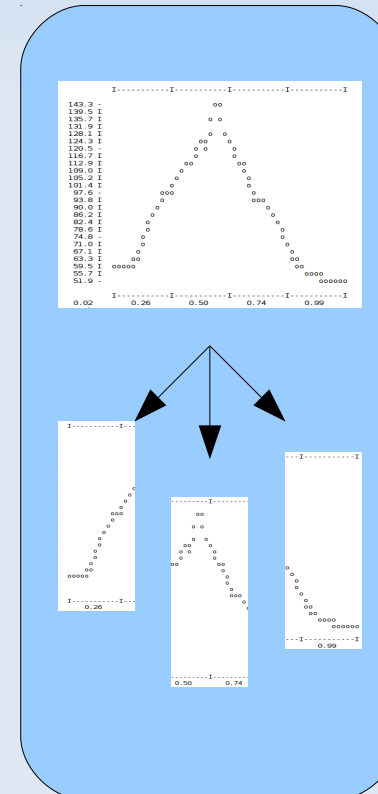
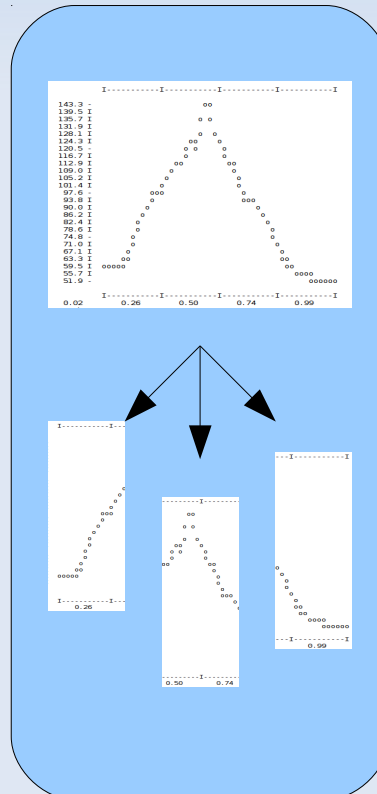
# Parallélisation (2)



qtlmap



Parallélisation  
des analyses  
de données simulées



architecture à mémoire distribuée  
+  
architecture à mémoire partagée

Parallélisation  
du génome scan

# MPI-QTLMap

- **Objectifs : exploiter la puissance de calcul des clusters des génopoles toulousaine et rennais et du ctig**
- **Contraintes / Inconvénients :**
  - Disponibilités des clusters
    - Au moment de l'exécution : Combien de noeuds sont disponibles ?
    - Puis-je raisonnablement prendre toutes les ressources ?
  - Politique d'administration du cluster
    - Environnement MPI : mpich+gfortran rennes, openmpi+ifort à toulouse
    - Politique d'allocation des noeuds, allocation erronés du à une surcharge
  - Pour des utilisateurs avertis
    - Ecriture de batch parallele avec execution de l'environnement mpi

**=> Relève plus de la demonstration de puissance de calcul que d'une utilisation en production**

# Implémentation

- Modification du code originale mais facilement adaptable si l'architecture logiciel a été bien pensé en couche
- Implémentation d'un gather :

```
call MPI_INIT (code)
call MPI_COMM_SIZE ( MPI_COMM_WORLD , nb_procs , code)
call MPI_COMM_RANK (MPI_COMM_WORLD, thread_rang , code)
```

```
nbsimulByThread = nsim / nb_procs
```

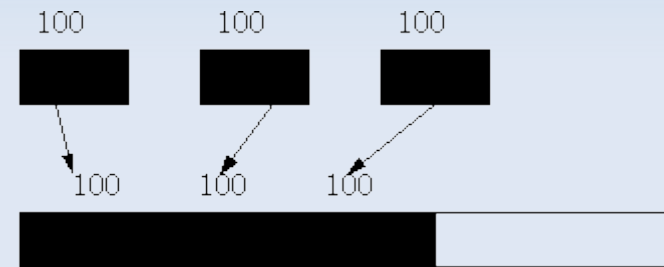
```
allocate (lrtmaxTab(nbsimulByThread))
!le processus Maitre (0)
If ( thread_rang == 0 ) then
  allocate (lrtmaxTab_final(nsim))
End if
```

```
do isim=1,nbsimulByThread
  call analyse(..,lrtmaxTab(isim)...)
End do
```

```
!Declaration des types a passer par message
call MPI_TYPE_CONTIGUOUS (nbsimulByThread, MPI_DOUBLE_PRECISION, dim3double,code)
call MPI_TYPE_COMMIT(dim3double,code)
```

```
! Ici le processus Maitre (0) recupere l'ensemble des resultats de la simulation....
```

```
deb_indice=nbsimulByThread * (thread_rang)+1
call MPI_GATHER (lrtmaxTab(1),1, dim3double ,lrtmaxTab_final(deb_indice),1,dim3double, 0 ,MPI_COMM_WORLD ,code) 15
call MPI_TYPE_FREE(dim3double,code)
```



# Problème

**Initialisation du germe pour la génération  
de nombre aléatoire via la date : unicité du germe dans un  
contexte parallèle ?**

! Initialize Random number Generator CoMmon (for Randlib)

```
call inrgcm()
```

! Init the random generator

```
call DATE_AND_TIME(date, time, zone, value)
```

```
phrase=date//time//zone
```

```
call phrtsd(phrase,iseed1,iseed2)
```

```
call setall(iseed1,iseed2)
```

« la génération de nombres aléatoires est trop importante pour être confiée au hasard »

*Robert R. Coveyou du Oak Ridge National Laboratory*



# BayesCPi

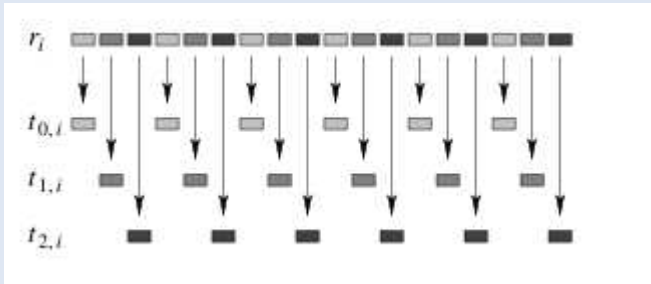
- Parallélisation du MCMC : exécuter N chaînes parallèles puis recoller les données générées (**GATHER**).
- Utilisation des PRNG (**Pseudo Random Number Generator**) bien adapté à une implantation informatique des MCMC
- 4 techniques : **le random seeding, la paramétrisation, le block splitting et le leapfrog.**

La random seeding et la paramétrisation empêche le "fair play"

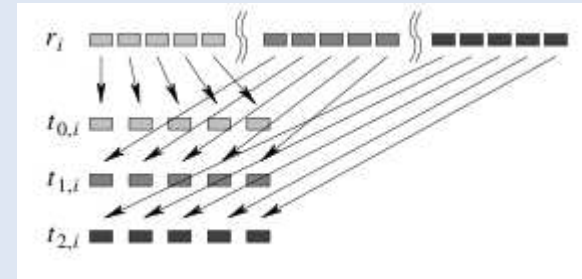
$$\begin{aligned}T(\theta, i) &= R(P*i) \\T(1, i) &= R(P*i+1) \\&\dots \\T(p-1, i) &= R(P*i+(p-1))\end{aligned}$$

$$\begin{aligned}T(\theta, i) &= R(i) \\T(1, i) &= R(i+M) \\&\dots \\T(p-1, i) &= R(i+M*(p-1))\end{aligned}$$

*M* nombre maximum d'appel au PRNG  
*P* le nombre de processus  
*R* sequence



leapfrog



Block splitting

# BayesCPi (2)

Tina's Random Number Generator Library (TRNG) is a state of the art C++ pseudo-random number generator library for sequential and parallel Monte Carlo simulations. (**Tina's RNG**)

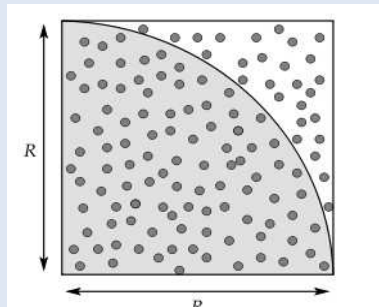
## - Collection de générateurs

- Linear congruential generator
- Multiple recurrence generator based on a linear feedback shift register sequence over  $F(2^{31}-1)$  of depth N
- Multiple recurrence generator based on a linear feedback shift register sequence over  $F(M)$  of depth N, with M being a Sophie Germain Prime
- Yet Another Random Number sequence based on a linear feedback shift register sequence over  $F(2^{31}-1)$  of depth N based on a multiple recursive generator
- Yet Another Random Number sequence based on a linear feedback shift register sequence over  $F(M)$  of depth N, with M being a Sophie Germain Prime

## - Collection de distributions : loi uniforme, gamma, normale....

### Example:

The numerical value of  $\pi$  can be estimated by throwing random points into a square.



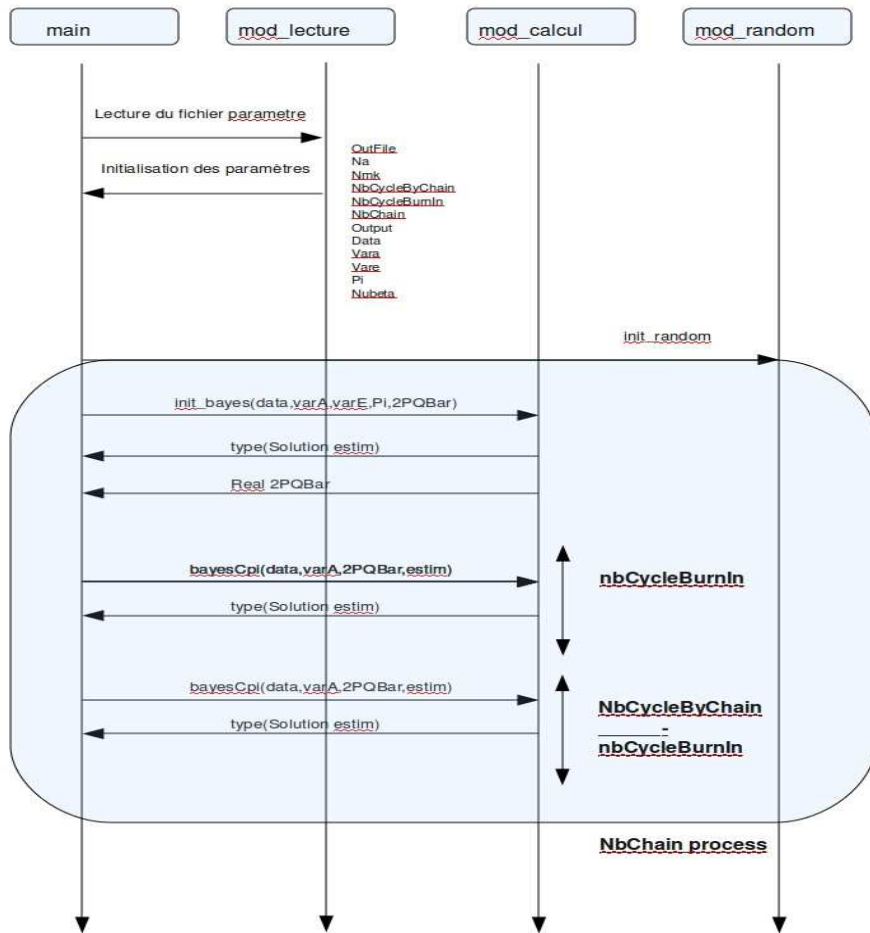
$$\frac{\text{number of points in circle}}{\text{number of points in square}} \approx \frac{\pi R^2 / 4}{R^2} = \frac{\pi}{4}$$

$$\pi \approx 4 \frac{\text{number of points in circle}}{\text{number of points in square}}$$

Listing 6.7: Parallel Monte Carlo calculation of  $\pi$  using leapfrog and OpenMP.

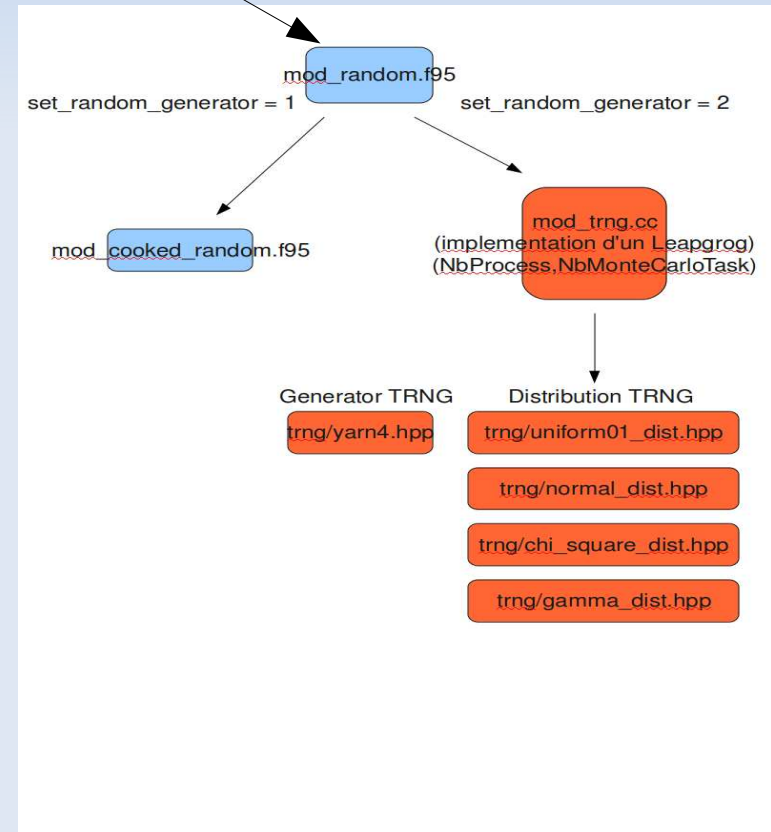
```
1 #include <cstdlib>
2 #include <iostream>
3 #include <omp.h>
4 #include <trng/yarn2.hpp>
5 #include <trng/uniform01_dist.hpp>
6
7 int main(int argc, char *argv[]) {
8     const long samples=10000001; // total number of points in square
9     long in=01; // no points in circle
10    // distribute workload over all processes
11    #pragma omp parallel
12    {
13        trng::yarn2 rx, ry; // random number engines for x- and y-coordinates
14        int size=omp_get_num_threads(); // get total number of processes
15        int rank=omp_get_thread_num(); // get rank of current process
16        // split PRN sequences by leapfrog method
17        rx.split(2, 0); // choose sub-stream no. 0 out of 2 streams
18        ry.split(2, 1); // choose sub-stream no. 1 out of 2 streams
19        rx.split(size, rank); // choose sub-stream no. rank out of size streams
20        ry.split(size, rank); // choose sub-stream no. rank out of size streams
21        trng::uniform01_dist<> u; // random number distribution
22        long in_local=01;
23        // throw random points in square
24        for (long i=rank; i<samples; i+=size) {
25            double x=u(rx), y=u(ry); // choose random x- and y-coordinates
26            if (x*x+y*y<=1.0) // is point in circle?
27                ++in_local; // increase thread-local counter
28        }
29        #pragma omp critical
30        in+=in_local; // increase global counter
31    }
32    // print result
33    std::cout << "pi = " << 4.0*in/samples << std::endl;
34    return EXIT_SUCCESS;
35 }
```

# BayesCPi(3)



```

subroutine bayesCpl(...)
...
!STEP 1 : tirage de la variance residuelle
!-----
!on tire une V.A (u) dans une loi de CHI2 à N+3 ddl
u = random_chisq(numRandom=id_chisq,ndf=data%na+3)
...
    
```



# BayesCPi (3)

```
#include <trng/yarn4.hpp>
#define GENERATOR_TYPE          trng::yarn4

static GENERATOR_TYPE *streamsChisq = NULL;
static GENERATOR_TYPE *streamsNormal = NULL;
static GENERATOR_TYPE *streamsUnif = NULL;
static GENERATOR_TYPE *streamsBeta = NULL;

//Directive openMP : les tableaux streamsXXX sont locaux a chaque processus OpenMP
#pragma omp threadprivate (streamsChisq,streamsNormal,streamsUnif,streamsBeta)
```

Déclaration des générateurs utilisés par le bayescpi

1

Initialisation

2

```
// LEAPFROG Method .
// allocation of a random number engine for each random we needs
if ( *nbChisq > 0 ) streamsChisq = new GENERATOR_TYPE[*nbChisq];
if ( *nbNorm > 0 ) streamsNormal = new GENERATOR_TYPE[*nbNorm];
if ( *nbUnif > 0 ) streamsUnif = new GENERATOR_TYPE[*nbUnif];
if ( *nbBeta > 0 ) streamsBeta = new GENERATOR_TYPE[2*(*nbBeta)]; // we used 2 gamma...

//Number of generator
int total = *nbChisq + *nbNorm + *nbUnif + 2*(*nbBeta) ;

//Generator for chi2 random
for (int j=0;j<*nbChisq;j++) {
    streamsChisq[j].split(total,j);
    streamsChisq[j].split(*nbProc,*whichProc);
}

//Generator for normal random
for (int j=0;j<*nbNorm;j++) {
    streamsNormal[j].split(total,*nbChisq + j);
    streamsNormal[j].split(*nbProc,*whichProc);
}

//Generator for uniform random
for (int j=0;j<*nbUnif;j++) {
    streamsUnif[j].split(total,*nbChisq + *nbNorm + j);
    streamsUnif[j].split(*nbProc,*whichProc);
}

//Generator for beta random (using 2 gamma random)
for (int j=0;j<2*(*nbBeta);j++) {
    streamsBeta[j].split(total,*nbChisq + *nbNorm + *nbUnif + j);
    streamsBeta[j].split(*nbProc,*whichProc);
}
```

utilisation

3

```
extern "C" void trng_normal_leap_(int *numRand,float*mean,float*sig,float*res) {
    trng::normal_dist<float> d(*mean, *sig);

    assert (numRand!=NULL);
    assert ((*numRand)>=0);

    *res = d(streamsNormal[*numRand]);
}
```

20

```
! ** TRNG Implementation **
! Interface with C++ developpement of mod_trng.cc
interface
    subroutine init_random_leap(nbProc,whichProc,nb_chisq,nb_norm,nb_unif,nb_beta)
        integer ,intent(in) :: nbProc ! number of processor
        integer ,intent(in) :: whichProc ! numero du processus OpenMP
        integer ,intent(in) :: nb_chisq,nb_norm,nb_unif,nb_beta ! nombre de generateur
    end subroutine init_random_leap

    subroutine trng_normal_leap(numRand,mean,sig,res)
        integer ,intent(in) :: numRand ! index du random
        real ,intent(in) :: mean
        real ,intent(in) :: sig
        real ,intent(out) :: res
    end subroutine trng_normal_leap
...
End interface
```

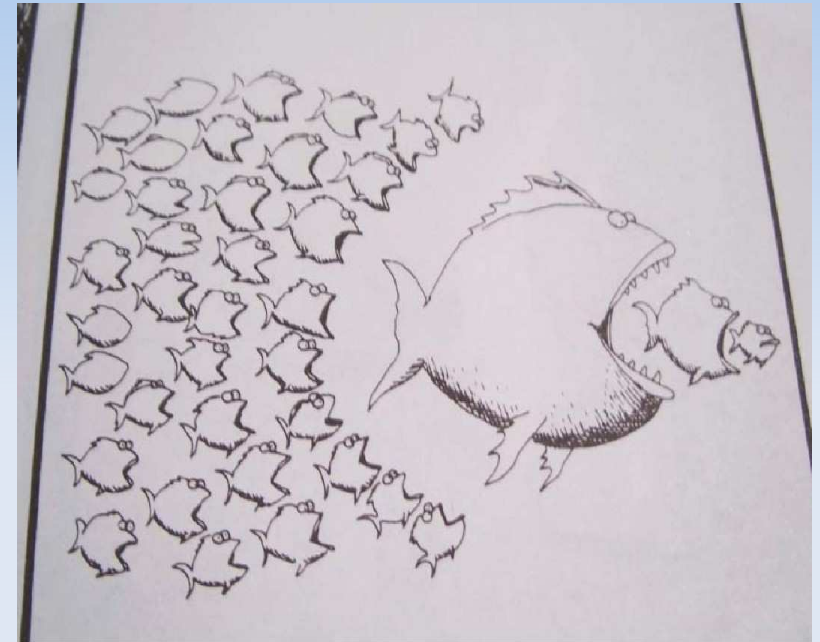
# GPU Computing

## GPU : Graphical Processing Unit

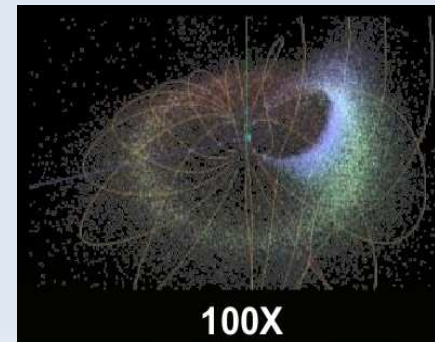
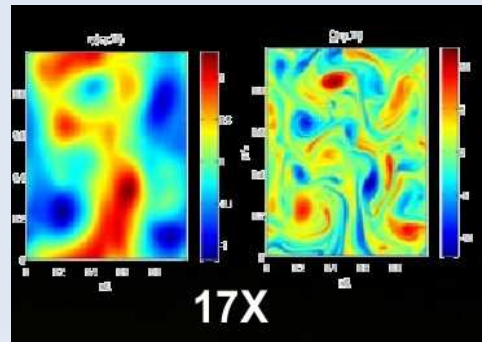
- Cartes vidéos, PlayStation3,XBox
- NVIDIA,ATI,AMD

**CUDA** (an acronym for Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA

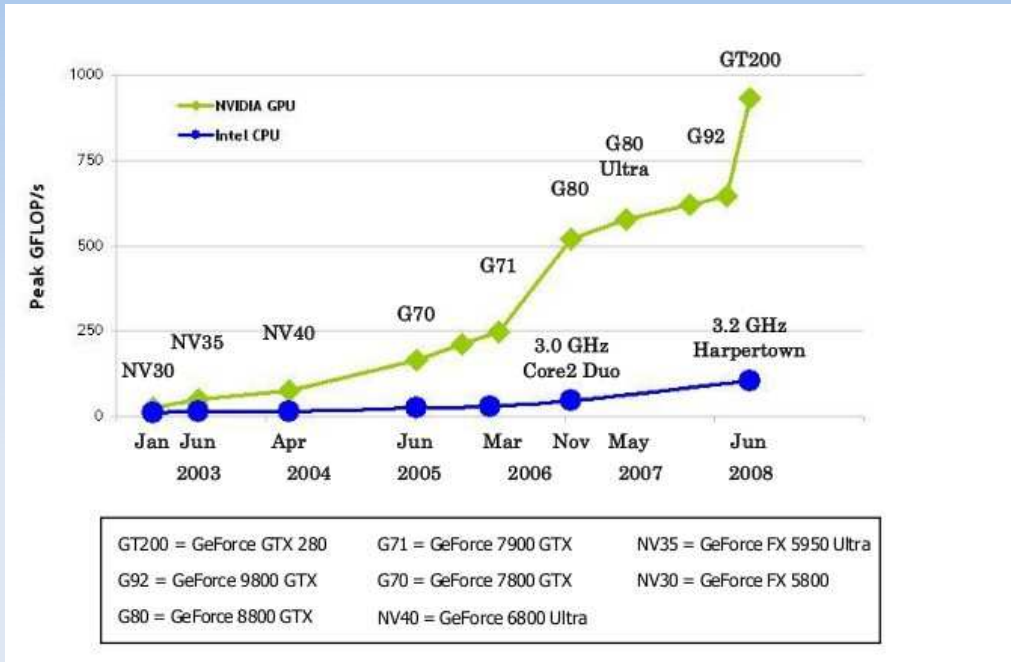
**GPGPU** : General-purpose computing on graphics processing units



## Pourquoi ?



# Comparaison/Performances

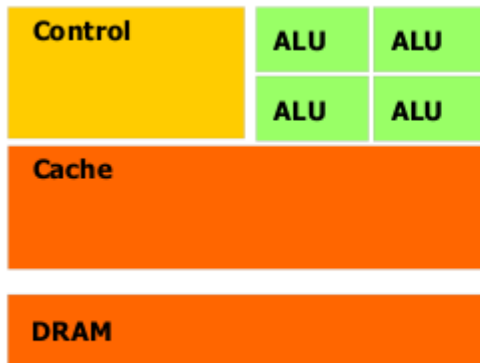


## Evolution des GPU / CPU

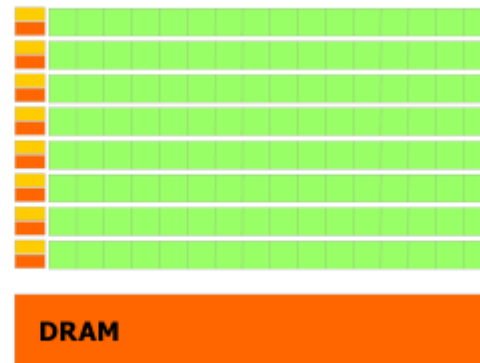
**G80 (Nov 2006 – GeForce 8800 GTX)**  
 128 flux processeurs  
 Jusqu'à 12 288 threads simultanés  
 Mémoire partagés (PBSM) accélère le calcul

**Fermi (2010 – Tesla C2050)**  
 448 coeurs  
 1,03 TFlops SP, 515 Gflops DP

”En comparaison avec les derniers CPU quad-core, les solutions Tesla C2050 et C2070 délivrent des performances de supercalcul équivalentes pour une consommation 20 fois moins importante et un prix 10 fois plus faible.”



CPU

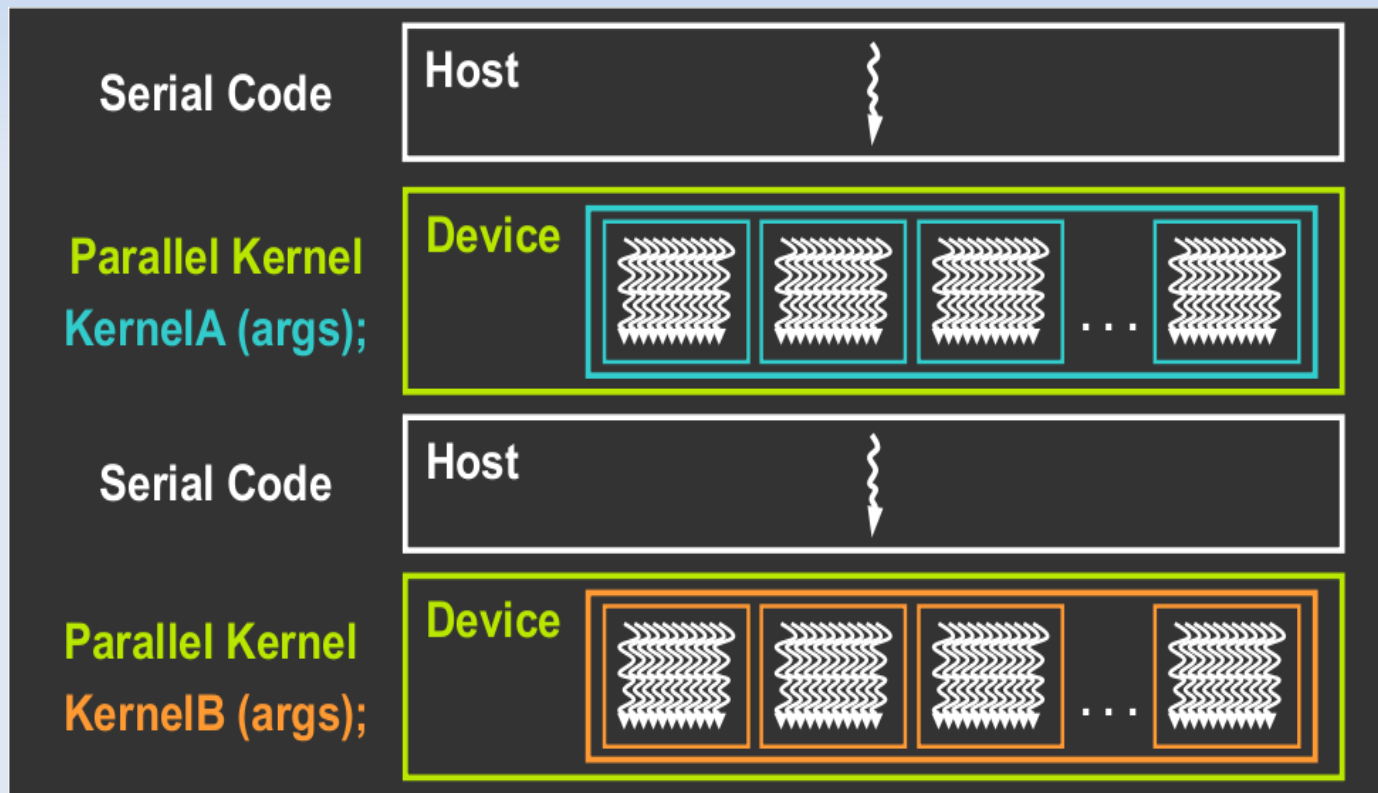


GPU

the GPU is specialized for compute-intensive, highly parallel computation

# Programmation hétérogène

- alternance de code séquentiel (fonctions **host**) / code parallèle (fonctions **kernel**)
- les fonctions **hosts** s'exécutent sur un seul thread (**CPU thread**)
- les fonctions **kernels** s'exécutent sur plusieurs threads (**GPU threads**)



Fonction **kernel** = nombre threads simultanés

# A Scalable Programming Model

CUDA est un modèle de programmation parallèle

Kernel = grille de block de thread

- Les blocks de threads ne sont pas synchronisés => **exécutions séquentielles ou simultanées** ce qui permet une évolution du modèle de programmation
- Block Id 1D ou 2D
- Thread Id 1D,3D

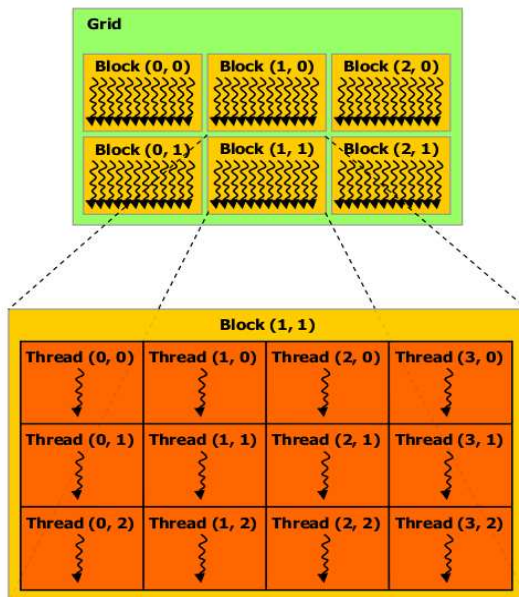
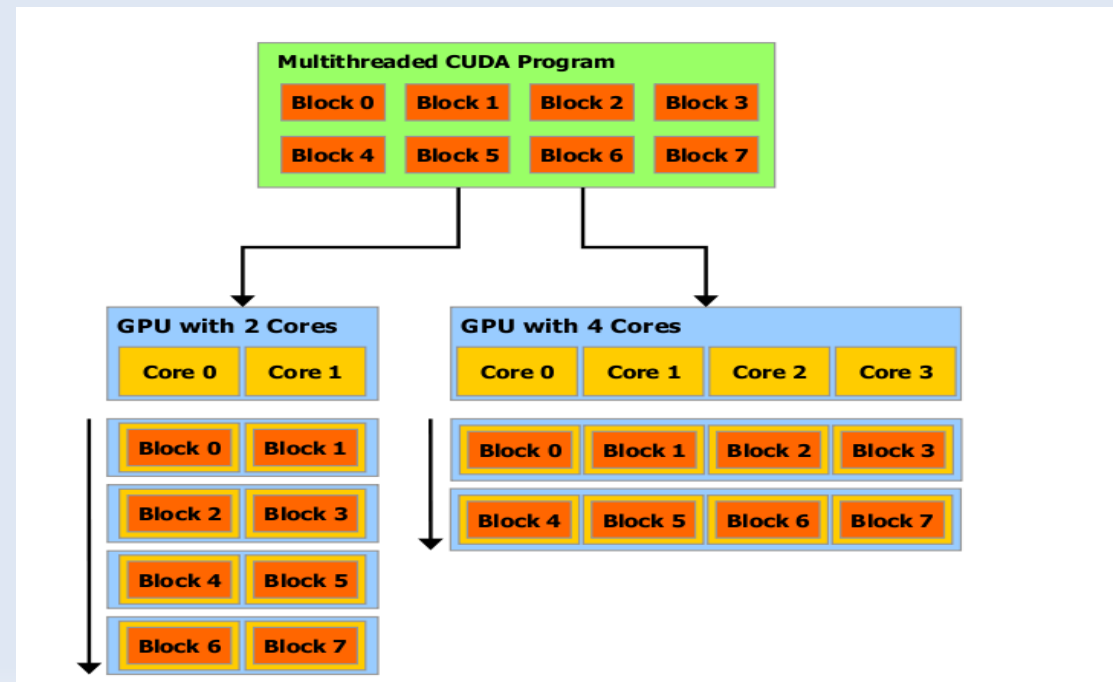
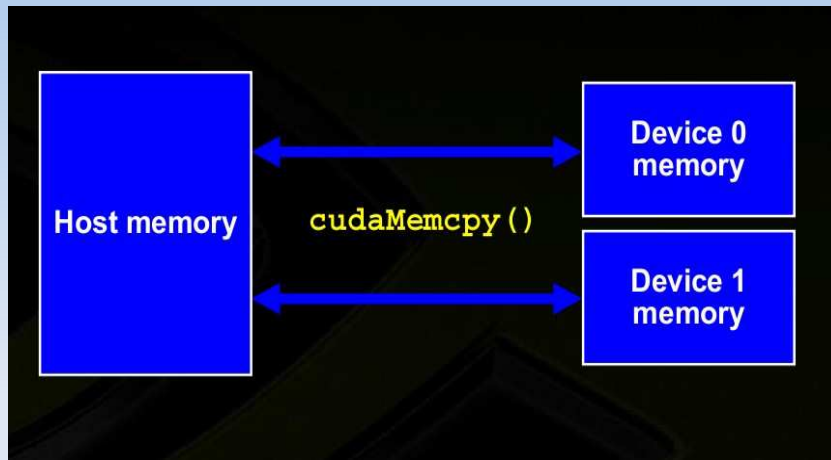


Figure 2-1. Grid of Thread Blocks



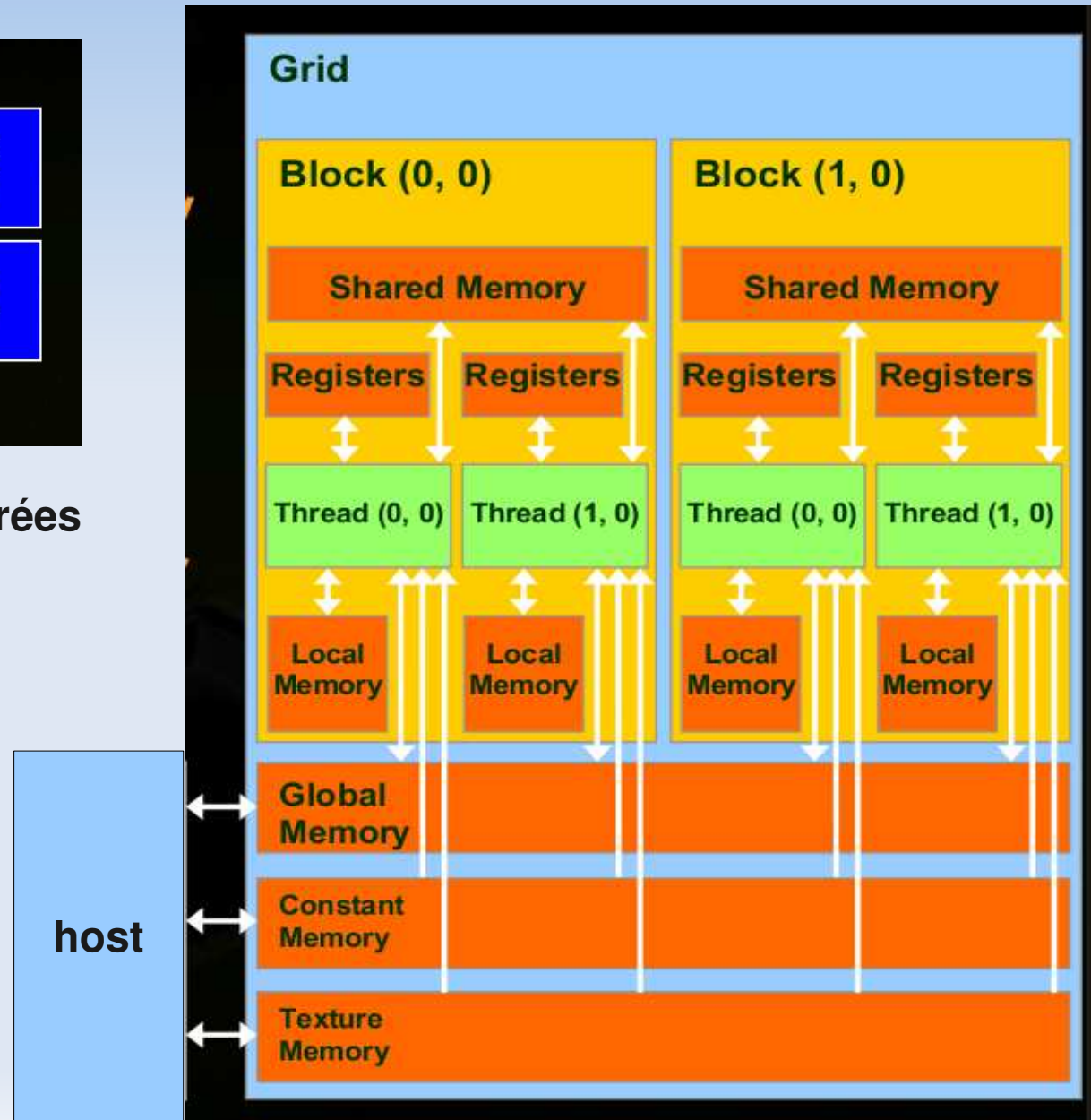


# Utilisation des données



Les données doivent être transférées sur le matériel

Hiérarchie mémoire



# Exemple

## CPU program

```
void increment_cpu(float *a,
                  float b,
                  int N)
{
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}

void main()
{
    .....
    increment_cpu(a, b, N);
}
```

## CUDA program

```
__global__ void increment_gpu(float *a,
                              float b,
                              int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}

void main()
{
    .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

# Analyse QTLMMap - Cuda

Première implémentation naive => on écrit l'algorithme du modèle lineaire homoscedastique

- 1) Transfert de la matrice de contingence sur le device
- 2)  $X.X^{-1}$  (CUBLAS::cublasSsyrk)
- 3) Estimabilité des effets du modèle (décomposition cholesky)
- 4) Construction de la solution RHS =  $Xr'.Y$  (CUBLAS::cublasSgemv)
- 5) Résolution LU (Etape de descente)
- 6) Résolution LU (Etape de remonté)
- 7)  $XB = XINC * \text{Solution}$  (CUBLAS::cublasSgemv)
- 8)  $RHS = Y - XB$  (CUBLAS::cublasSscal,cublasSaxpy)
- 9) Variances résiduelles (CUBLAS::cublasSdot)
- 10) Récupération des données stoquées sur le device

Remarques:

- Les données statiques (Y,CD,...) sont prealablement transférées sur le device.
- Utilisation de la librairie CUBLAS
- implémentation ad-hoc

=> Perte de performance.....

- >Matrice trop petite (peu de niveau à estimer=>pas intensif).
- >Le calcul à la position implique beaucoup trop de transfert sur le matériel

Conclusion/retour d'exp. :

=> calculer les solutions pour toutes les positions en même temps

- 1 seul transfert
- beaucoup plus de threads vont être utilisés

=> Pour une utilisation de calcul matriciel "simple" => implémentation en fortran simple avec CUBLAS sans besoin de connaître les rouages de CUDA

# Synthèses

	Prise en main	Difficultés	Deploiement	Retour d'ex.
OpenMP	++		<ul style="list-style-type: none"><li>- Serveur de calcul MP</li><li>- Machine personnel</li></ul>	+
MPI	+	<ul style="list-style-type: none"><li>- Environnement d'exécution</li><li>- Passage des informations/résultats entre processus</li></ul>	<ul style="list-style-type: none"><li>- Serveur de calcul MP, MD</li><li>- Machine personnelle</li></ul>	-
CUDA	-	<ul style="list-style-type: none"><li>- Réécritures des algos</li><li>- Connaissance du C</li></ul>	<ul style="list-style-type: none"><li>- Machine personnelle avec carte GPU</li><li>- Cluster GPU</li></ul>	-/+

MP : mémoire partagée

MD : mémoire distribuée (cluster)

Questions ?