Expérience de développement d'applications sur cluster de GPUs

Journée GPU
ONERA, Châtillon, France
12 janvier 2010

Stéphane Vialle

Stephane.Vialle@supelec.fr

SUPELEC équipe IMS & EPI AlGorille

Avec l'aide de L. Abbas-Turki, T. Jost , W. Kirshenmann, P. Mercier, S. Contassot-Vivier, L. Plagne

Plan

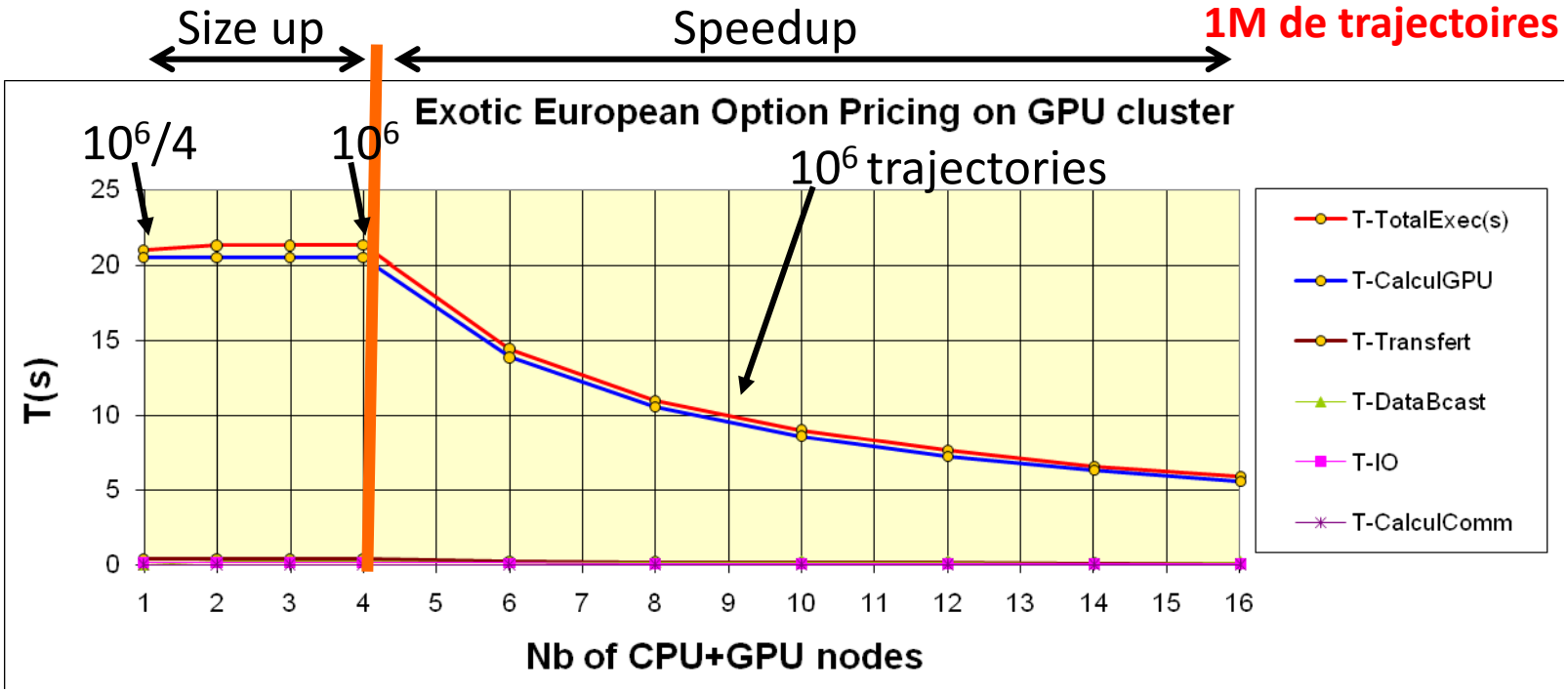
1. Principes de parallélisation sur cluster de CPU+GPU
2. Ex 1 : application distribuée *embarrassingly parallel*
3. Ex 2 : 1ère application distribuée fortement couplée
4. Ex 3 : 2ème application distribuée fortement couplée
5. Parallélisation simultanée sur CPUs et GPUs
6. Performances
7. Mesures et modélisation énergétiques
8. Conclusion

Principes de parallélisation sur cluster de CPU+GPU (1)

Un GPU permet d'aller plus vite mais possède une mémoire limitée (comparée à un CPU).

Nd GPU permettent d'aller encore plus vite (!) et de traiter un problème plus important :

Ex. de Monte-Carlo à 1M de trajectoires

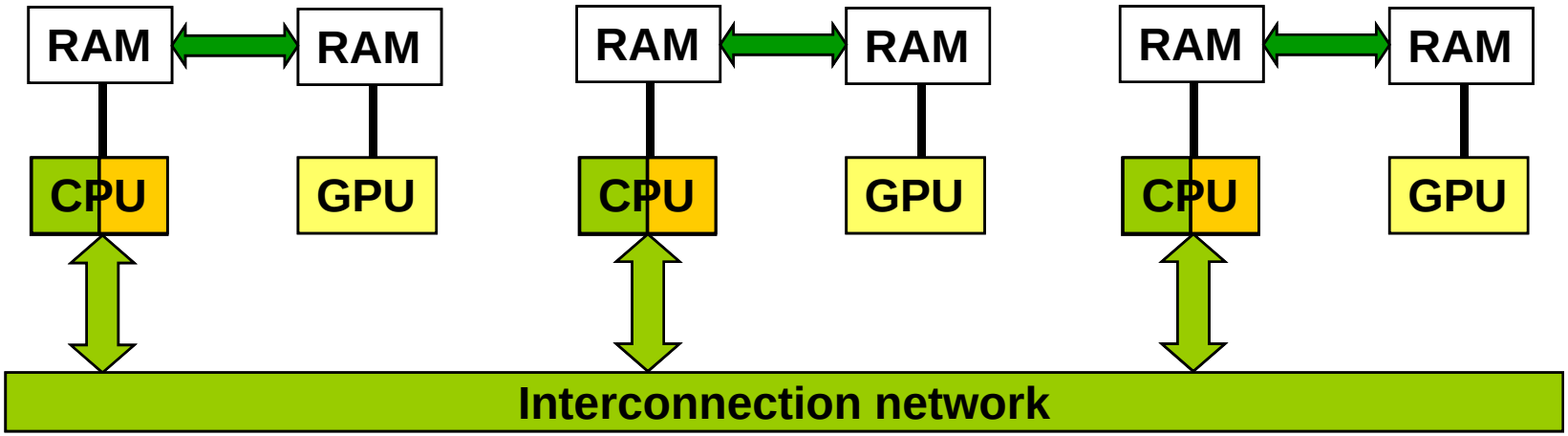


Principes de parallélisation sur cluster de CPU+GPU (2)

Un cluster de CPUs+GPUs est constitué :

- de CPUs multi-cœurs
- de GPUs (many-cœurs)
- d'un réseau d'interconnexion inter-nœuds
- de bus PCI-express pour les transferts entre CPU et GPU sur chaque nœud

Conceptuellement, tous ces composants peuvent fonctionner en parallèle…



Principes de parallélisation sur cluster de CPU+GPU (3)

Spécificités de la programmation des clusters de CPU+GPU :

- multi-grains :

  gros grain (cluster) + grain fin (GPU) + grain moyen (multi-cœurs CPU)

- multi-paradigms :

  messages + threads GPU + threads CPU

- multi-outils de développement :

  MPI + CUDA + OpenMP/IntelTBB/P-threads/…

… et donc :

- multi-bugs & multi-debugs
- multi-optimisations
- multi-surprises

Quelques difficultés algorithmiques

Cluster de CPUs – approche gros grain (processus et envois de messages)

1. Comment/où couper les calculs et les données ?
2. Equilibrer la charge sur les nœuds
3. Minimiser les communications
4. Recouvrir les communications et les calculs

+ optimisations sérielles

Cluster de CPUs – approche gros grain + grain moyen

5. Synchroniser les threads CPU pour l'accès aux données du nœud
6. Synchroniser les threads CPU avec les communications MPI
7. Eviter le *false sharing*

Cluster de GPUs – approche gros grain + grain fin

5. Minimiser les transferts CPU-GPU
6. Minimiser les accès à la RAM GPU globale – « réalisation du cache »
7. Réaliser des accès RAM globale « coalescent » (contigus et alignés)
8. Synchroniser les threads GPU
9. Synchroniser les threads GPU et les transferts CPU-GPU avec les communications MPI

Plan

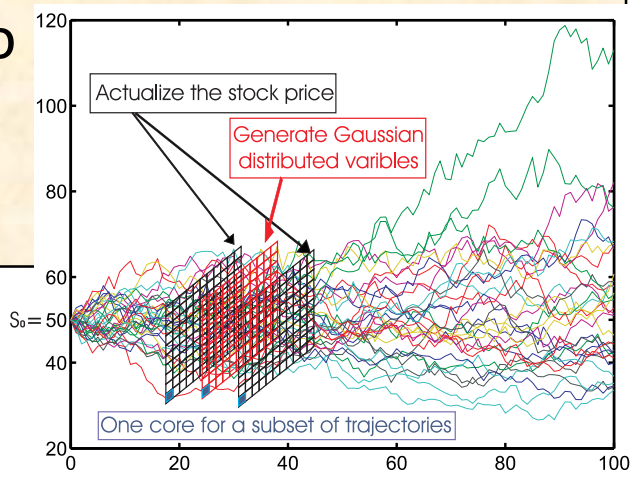
1. Principes de parallélisation sur cluster de CPU+GPU
2. Ex 1 : application distribuée *embarrassingly parallel*
3. Ex 2 : 1ère application distribuée fortement couplée
4. Ex 3 : 2ème application distribuée fortement couplée
5. Parallélisation simultanée sur CPUs et GPUs
6. Performances
7. Mesures et modélisation énergétiques
8. Conclusion

Ex 1 : application distribuée faiblement couplée

Objectifs :

Pricer d'options Européenne exotiques à grande vitesse (for hedging)

- calculant des trajectoires de Monte-Carlo
- utilisant un cluster de CPU multi-cœurs ou de GPU many-cœurs



Difficultés :

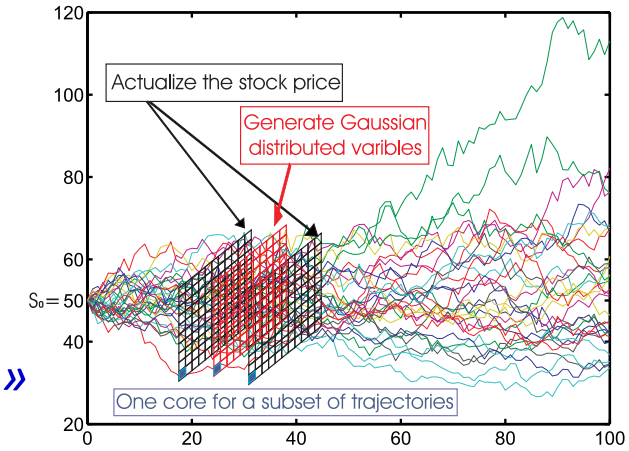
- Concevoir un algo parallèle gros-grain + grain moyen/grain fin
- Concevoir et implanter un générateur parallèle de nombres aléatoires **rigoureux et rapide**
- Evaluer les développements en termes de speedup, size up, précision, et consommation énergétique

Ex 1 : application distribuée faiblement couplée

Tirage des nombres aléatoires :

- « vite »
- de bonne qualité
- en parallèle

  → Risque de mauvaise qualité de la suite globale



« Ce n'est pas en faisant n'importe quoi que c'est aléatoire »

On génère des générateurs de nombres aléatoires en veillant :

- à ce qu'ils soient indépendants (génèrent des suites aléatoires différentes)
- à ce qu'ils puissent s'exécuter en parallèle (indépendamment les uns des autres)

On porte ces RNG sur GPUs :

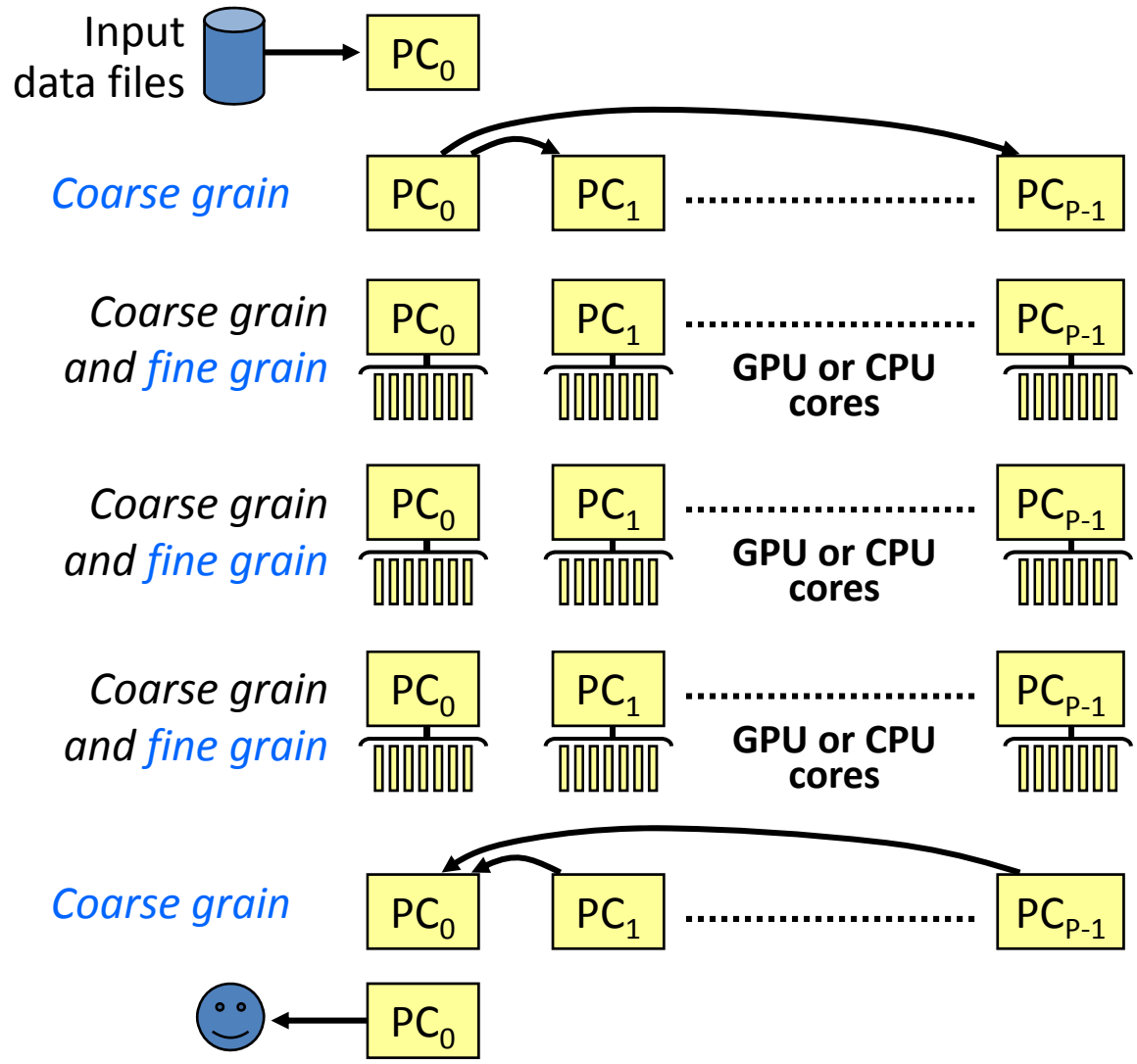
- pour rester sur le GPU pour tirer les nombres aléatoires
  (éviter de tirer les nombres aléatoires sur le CPU quand on utilise un CPU)

On teste différents RNG :

- de « qualités » différentes
- plus ou moins rapides

Ex 1 : application distribuée faiblement couplée

Algorithme parallèle :



1 - Input data reading on P_0

2 - Input data broadcast from P_0

3 - Parallel and independent RNG initialization

4 - Parallel and independent Monte-Carlo computations

5 - Parallel and independent partial results computation

6 - Partial results reduction on P_0 and final price computation

7 – Print results and perfs

Ex 1 : application distribuée faiblement couplée

MPI+OpenMP : calculs sur un nœud CPU multi-cœurs du cluster

```
void ActStock(double sqrtdt)
{
  int StkIdx, yIdx, xIdx;     // Loop indexes
  #pragma omp parallel private(StkIdx,yIdx,xIdx)
  {
  for (StkIdx = 0; StkIdx < NbStocks; StkIdx++) {
    Parameters_t *parPt = &par[StkIdx];
    // Process each trajectory
    #pragma omp for
    for (yIdx = 0; yIdx < Ny; yIdx++)
       for (xIdx = 0; xIdx < Nx; xIdx++) {
          float call;
          // - First pass
          call = ……;           // Calculs utilisant le RNG sur CPU
          // - The passes that remain
          for (int stock = 1; stock <= StkIdx ; stock++)
             call = ……;        // Calculs utilisant le RNG sur CPU
          // Copy result in the global GPU memory
          TabStockCPU[StkIdx][yIdx][xIdx] = call;
       }
  }
  }
}
```

Ex 1 : application distribuée faiblement couplée

MPI+CUDA : un thread GPU traite une trajectoire

```
__global__ void Actual_kernel(void)
{
  float call, callBis;
  // Computes the indexes and copy data into multipro sh. memory
  int xIdx = threadIdx.x + blockIdx.x*BlockSizeX;
  int yIdx = blockIdx.y;
  __shared__ float InputLine[Nx];
  __shared__ float BrownLine[Nx];
  InputLine[xIdx] = TabStockInputGPU[StkIdx][yIdx][xIdx];
  GaussLine[xIdx] = TabGaussGPU[0][yIdx][xIdx];
  // First pass
  call = ……;                    // Calculs utilisant le RNG sur GPU
  callBis = call;
  // The passes that remain
  for (int stock = 1; stock <= StkIdx; stock++) {
    GaussLine[xIdx] = TabGaussGPU[stock][yIdx][xIdx];
    call = callBis*……;           // Calculs utilisant le RNG sur GPU
    callBis = call;
  }
  // Copy result in the global GPU memory
  TabStockOutputGPU[StkIdx][yIdx][xIdx] = call;
}
```

Ex 1 : application distribuée faiblement couplée

MPI+CUDA : exécution des threads GPU (un par trajectoire)

```
void ActStock(double sqrtdt)
{
 // GPU thread management variables
 dim3 Dg, Db;
 // Set thread Grid and blocks features
 Dg.x = Nx/BlockSizeX; Dg.y = Ny; Dg.z = 1;
 Db.x = BlockSizeX; Db.y = 1; Db.z = 1;
 // Transfer a float version of the time increment on the GPU
 float sqrtdtCPU = (float) sqrtdt;
 cudaMemcpyToSymbol(sqrtdtGPU,&sqrtdtCPU,sizeof(float),0,
                    cudaMemcpyHostToDevice);
 // For each stock: transfer its index on the GPU and compute
 // its actualization (process all trajectories)
 for (int s = 0; s < NbStocks; s++) {
    cudaMemcpyToSymbol(StkIdx,&s,sizeof(int),0,
                       cudaMemcpyHostToDevice);
    Actual_kernel<<<Dg,Db>>>();  //Run the GPU computation
 }
}
```

Ex 1 : application distribuée faiblement couplée

MPI : routine principale

```
// MPI and appli inits
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD,&NbPE);
 MPI_Comm_rank(MPI_COMM_WORLD,&Me);
 ParseInit(argc,argv);
 if (Me == 0) { InitStockParaCPU(); }
 BroadcastInputData();
 // Application computations: use the multicore CPU ou GPU
 PostInitData(Me,NbPE);
 AsianSum(dt,0);
 for (jj = 1; jj <= N; jj++){                  // - time step loop
   for (k = 0; k < NbStocks; k++){             // - stock loop
     ComputeUniformRandom();                   //   + generation of uniform random numbers
     GaussPRNG(k);                             //   + turn uniform random nb to Gaussian random nb
   }
   ActStock(dt);                  // - compute the next stock value
   AsianSum(dt,jj);               // - compute the "average value of the stock" on each trajectory
 }
 ComputeIntegralSum();                         // - Ponderated sum of the average values
 ComputePriceSum();                            // - Ponderated sum of the stock prices
 ComputePayoffSum(&sum,&sum2);                 // - Compute the sum and square sum of the payoff
 CollectByReduction(&sum, &sum2, &TotalSum, &TotalSum2);
 // - Final computations on PE-0
 if (Me == 0) { ComputeFinalValues(&price,&error,TotalSum,TotalSum2,r,NbPE); }
MPI_Finalize();
```

Pas d'impact d'OpenMP

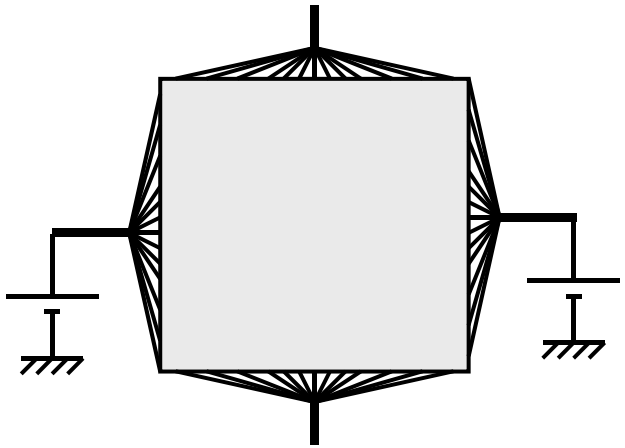
Pas d'impact de CUDA

Synchronisation légère des process MPI et des threads de calcul (CPU ou GPU)

Plan

1. Principes de parallélisation sur cluster de CPU+GPU
2. Ex 1 : application distribuée *embarrassingly parallel*
3. Ex 2 : 1ère application distribuée fortement couplée
4. Ex 3 : 2ème application distribuée fortement couplée
5. Parallélisation simultanée sur CPUs et GPUs
6. Performances
7. Mesures et modélisation énergétiques
8. Conclusion

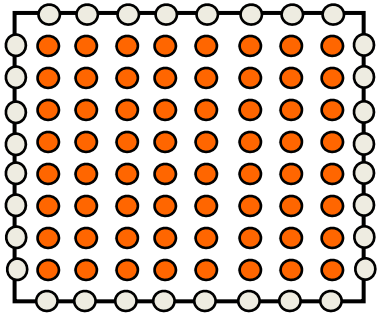
Ex 2 : application distribuée fortement couplée



Calcul des lignes de potentiel dans une plaque diélectrique :

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0$$

- Discrétisation et équation aux différences
- Itération jusqu'à la convergence ($V^{n+1}_{i,j} - V^{n}_{i,j} < \varepsilon$)



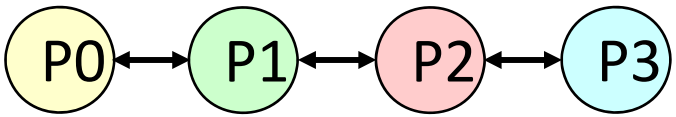
- Condition aux limites : V fixé
- $$V^{n+1}_{i,j} = \frac{V^{n}_{i-1,j} + V^{n}_{i+1,j} + V^{n}_{i,j-1} + V^{n}_{i,j+1}}{4}$$

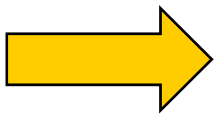
Ex 2 : application distribuée fortement couplée

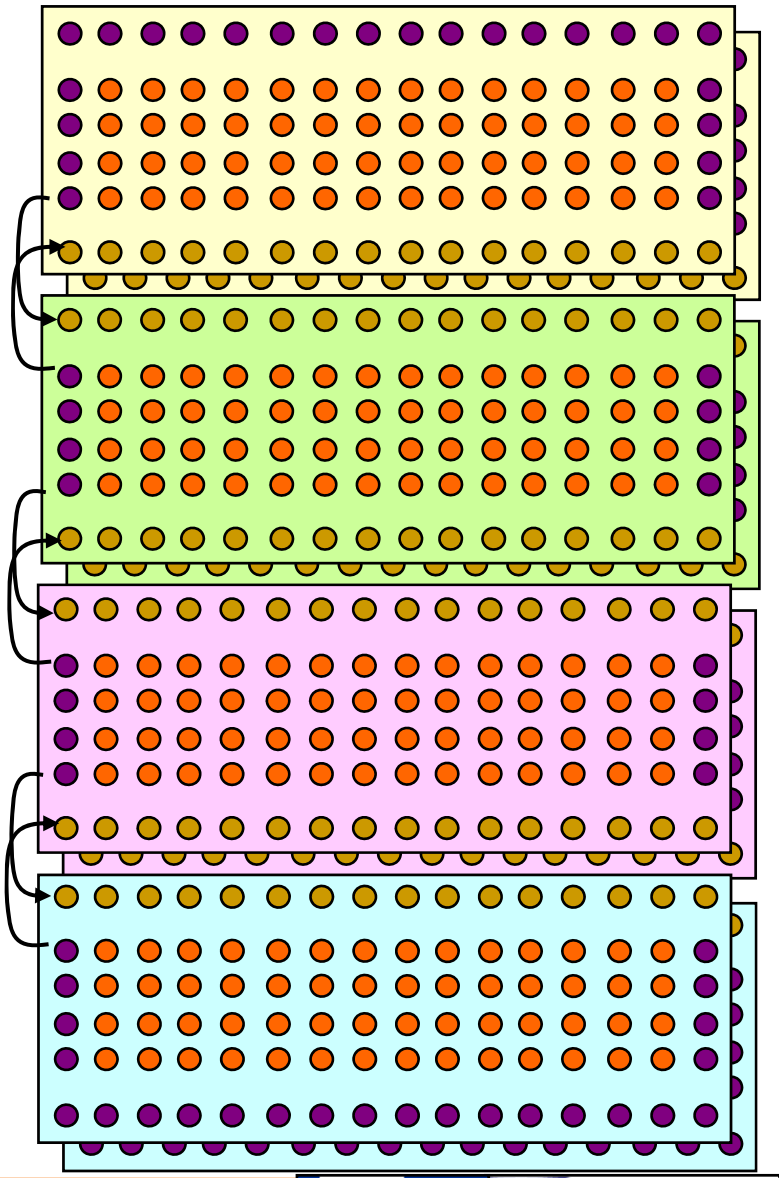
Parallélisation sur cluster par envoi de messages entre les nœuds de calcul :

- Partitionnement spatial des données et échange des frontières
- 2 extraits de tables (V^n et V^{n+1}) dans chaque mémoire locale + les frontières
- Parallélisation de la boucle sur les lignes de la grille

Ex : avec une ligne de 4 nœuds



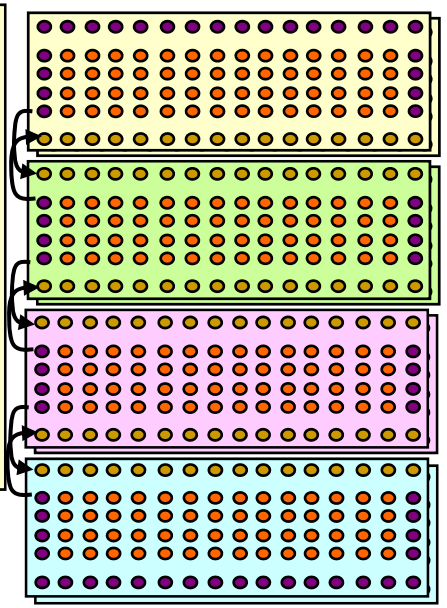




Ex 2 : application distribuée fortement couplée

Algorithme exécuté sur chaque proc. :

```
for (cycle = 0; cycle < NbCycle; cycle++) {
  calcul_local();
  //barriere();
  echange_frontieres(); // Avec synchronisation
                        // sur les comms.
  //barriere();
  permutation_indices_des_tables_Vn_et_Vnplus1();
}
```

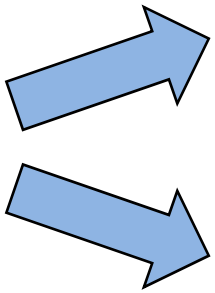


`Calcul_local()` :

- Nœud CPU mono-cœur : un process MPI
- Nœud CPU multi-cœurs : un process MPI + des threads OpenMP
- Nœud CPU+GPU : un process MPI + des threads GPU (+ des threads CPU pour les comms. asynchrones)

Ex. simple de code MPI+OpenMP (relax.)

Une *ligne de nœuds de calculs* pour exécuter une *relaxation de Jacobi*



Implantation en MPI+OpenMP sur CPUs multi-cœurs.

Implantation en MPI+CUDA sur GPUs

Impact d'OpenMP sur le code MPI ?

Impact de CUDA sur le code MPI ?

Ex. simple de code MPI+OpenMP (relax.)

MPI+OpenMP : calculs sur un nœud CPU multi-cœurs du cluster

```
#pragma omp parallel private(cycle,i,j,OI,NI)
 {
 OI = 0;
 NI = 1;

 for (cycle = 0; cycle < NbCycles; cycle++) {
   #pragma omp for
   for (i = 1; i <= USEFUL_SIDE_ROW; ++i) {        // - local grid computation
     for (j = 1; j <= USEFUL_SIDE_COL; ++j) {
       CPU_V[NI][i][j] = (CPU_V[OI][i-1][j] + CPU_V[OI][i+1][j] +
                          CPU_V[OI][i][j-1] + CPU_V[OI][i][j+1]) / 4.0;
     }
   }

   #pragma omp single
   {
     (*cpuCommFctPt)(NI,cycle);                    // - MPI comms.
   }

   OI = NI;                                        // - index switching
   NI = 1-NI;
 }
 }
```

Coordination des threads OpenMP et des comms. MPI

Ex. simple de code MPI+OpenMP (relax.)

MPI+OpenMP : Echange de frontières sur un anneau de CPU – v1-synchrone

```
if (Me == 0) {
        MPI_Sendrecv(&CPU_V[idx][SIDE_ROW-2][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                     &CPU_V[idx][SIDE_ROW-1][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                     MPI_COMM_WORLD,&status);
} else if (Me == NbPr - 1) {
        MPI_Sendrecv(&CPU_V[idx][1][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                     &CPU_V[idx][0][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                     MPI_COMM_WORLD,&status);
} else if (Me % 2 == 0) {
        MPI_Sendrecv(&CPU_V[idx][SIDE_ROW-2][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                     &CPU_V[idx][SIDE_ROW-1][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                     MPI_COMM_WORLD,&status);
        MPI_Sendrecv(&CPU_V[idx][1][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                     &CPU_V[idx][0][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                     MPI_COMM_WORLD,&status);
} else if (Me % 2 == 1) {
        MPI_Sendrecv(&CPU_V[idx][1][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                     &CPU_V[idx][0][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                     MPI_COMM_WORLD,&status);
        MPI_Sendrecv(&CPU_V[idx][SIDE_ROW-2][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                     &CPU_V[idx][SIDE_ROW-1][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                     MPI_COMM_WORLD,&status);
}
```

Pas d'impact d'OpenMP

Ex. simple de code MPI+OpenMP (relax.)

MPI+OpenMP : Echange de frontières sur un anneau de CPU – v2-asynchrone

```
if (Me == 0) {
   MPI_Irecv(&CPU_V[idx][SIDE_ROW-1][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                              MPI_COMM_WORLD,&RecvRequest[idx][1]);
   MPI_Issend(&CPU_V[idx][SIDE_ROW-2][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                              MPI_COMM_WORLD,&SendRequest[idx][1]);
   MPI_Wait(&RecvRequest[idx][1],&RecvStatus[1]);
   MPI_Wait(&SendRequest[idx][1],&SendStatus[1]);
 } else if (Me == NbPr - 1) {
   MPI_Irecv(&CPU_V[idx][0][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                              MPI_COMM_WORLD,&RecvRequest[idx][0]);
   MPI_Issend(&CPU_V[idx][1][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                              MPI_COMM_WORLD,&SendRequest[idx][0]);
   MPI_Wait(&RecvRequest[idx][0],&RecvStatus[0]);
   MPI_Wait(&SendRequest[idx][0],&SendStatus[0]);
 } else {
   MPI_Irecv(&CPU_V[idx][0][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                              MPI_COMM_WORLD,&RecvRequest[idx][0]);
   MPI_Irecv(&CPU_V[idx][SIDE_ROW-1][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                              MPI_COMM_WORLD,&RecvRequest[idx][1]);
   MPI_Issend(&CPU_V[idx][1][0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                              MPI_COMM_WORLD,&SendRequest[idx][0]);
   MPI_Issend(&CPU_V[idx][SIDE_ROW-2][0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                              MPI_COMM_WORLD,&SendRequest[idx][1]);
   MPI_Waitall(2,RecvRequest[idx],RecvStatus);
   MPI_Waitall(2,SendRequest[idx],SendStatus);
 }
```

Pas d'impact d'OpenMP

Ex. simple de code MPI+OpenMP (relax.)

MPI+OpenMP : Echange de frontières sur un anneau de CPU – v3-async-pers.

1/3

```
if (Me == 0) {
   for (int idx = 0; idx < 2; idx++) {
     MPI_Ssend_init(&CPU_V[idx][SIDE_ROW-2][0],SIDE_COL,MPI_FLOAT,Me+1,0,
                    MPI_COMM_WORLD,&SendRequest[idx][1]);
     MPI_Recv_init(&CPU_V[idx][SIDE_ROW-1][0],SIDE_COL,MPI_FLOAT,Me+1,0,
                   MPI_COMM_WORLD,&RecvRequest[idx][1]);
   }
} else if (Me == NbPr-1) {
   for (int idx = 0; idx < 2; idx++) {
     MPI_Ssend_init(&CPU_V[idx][1][0],SIDE_COL,MPI_FLOAT,Me-1,0,
                    MPI_COMM_WORLD,&SendRequest[idx][0]);
     MPI_Recv_init(&CPU_V[idx][0][0],SIDE_COL,MPI_FLOAT,Me-1,0,
                   MPI_COMM_WORLD,&RecvRequest[idx][0]);
   }
} else {
   for (int idx = 0; idx < 2; idx++) {
     MPI_Ssend_init(&CPU_V[idx][1][0],SIDE_COL,MPI_FLOAT,Me-1,0,
                    MPI_COMM_WORLD,&SendRequest[idx][0]);
     MPI_Ssend_init(&CPU_V[idx][SIDE_ROW-2][0],SIDE_COL,MPI_FLOAT,Me+1,0,
                    MPI_COMM_WORLD,&SendRequest[idx][1]);
     MPI_Recv_init(&CPU_V[idx][0][0],SIDE_COL,MPI_FLOAT,Me-1,0,
                   MPI_COMM_WORLD,&RecvRequest[idx][0]);
     MPI_Recv_init(&CPU_V[idx][SIDE_ROW-1][0],SIDE_COL,MPI_FLOAT,Me+1,0,
                   MPI_COMM_WORLD,&RecvRequest[idx][1]);
   }
}
```

Initialisation des schémas de communications persistants

Ex. simple de code MPI+OpenMP (relax.)

MPI+OpenMP : Echange de frontières sur un anneau de CPU – v3-async-pers.

2/3

```
if (Me == 0) {
   MPI_Start(&RecvRequest[idx][1]);
   MPI_Start(&SendRequest[idx][1]);
   MPI_Wait(&RecvRequest[idx][1],&RecvStatus[1]);
   MPI_Wait(&SendRequest[idx][1],&SendStatus[1]);
} else if (Me == NbPr - 1) {
   MPI_Start(&RecvRequest[idx][0]);
   MPI_Start(&SendRequest[idx][0]);
   MPI_Wait(&RecvRequest[idx][0],&RecvStatus[0]);
   MPI_Wait(&SendRequest[idx][0],&SendStatus[0]);
} else {
   MPI_Start(&RecvRequest[idx][0]);
   MPI_Start(&RecvRequest[idx][1]);
   MPI_Start(&SendRequest[idx][0]);
   MPI_Start(&SendRequest[idx][1]);
   MPI_Waitall(2,RecvRequest[idx],RecvStatus);
   MPI_Waitall(2,SendRequest[idx],SendStatus);
}
```

Exploitation des schémas de communications persistants

Ex. simple de code MPI+OpenMP (relax.)

MPI+OpenMP : Echange de frontières sur un anneau de CPU – v3-async-pers.

3/3

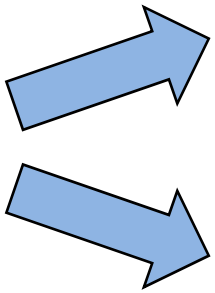
```
if (Me == 0) {
   for (int idx = 0; idx < 2; idx++) {
     MPI_Request_free(&SendRequest[idx][1]);
     MPI_Request_free(&RecvRequest[idx][1]);
   }
} else if (Me == NbPr-1) {
   for (int idx = 0; idx < 2; idx++) {
     MPI_Request_free(&SendRequest[idx][0]);
     MPI_Request_free(&RecvRequest[idx][0]);
   }
} else {
   for (int idx = 0; idx < 2; idx++) {
     MPI_Request_free(&SendRequest[idx][0]);
     MPI_Request_free(&RecvRequest[idx][0]);
     MPI_Request_free(&SendRequest[idx][1]);
     MPI_Request_free(&RecvRequest[idx][1]);
   }
}
```

Libération des schémas de communications persistants

Pas d'impact d'OpenMP

Ex. simple de code MPI+OpenMP (relax.)

Une *ligne de nœuds de calculs* pour exécuter une *relaxation de Jacobi*



Implantation en MPI+OpenMP sur CPUs multi-cœurs.

Implantation en MPI+CUDA sur GPUs

Impact d'OpenMP sur le code MPI ?

Impact de CUDA sur le code MPI ?

Ex. simple de code MPI+CUDA (relax.)

MPI+CUDA : calculs sur un nœud GPU

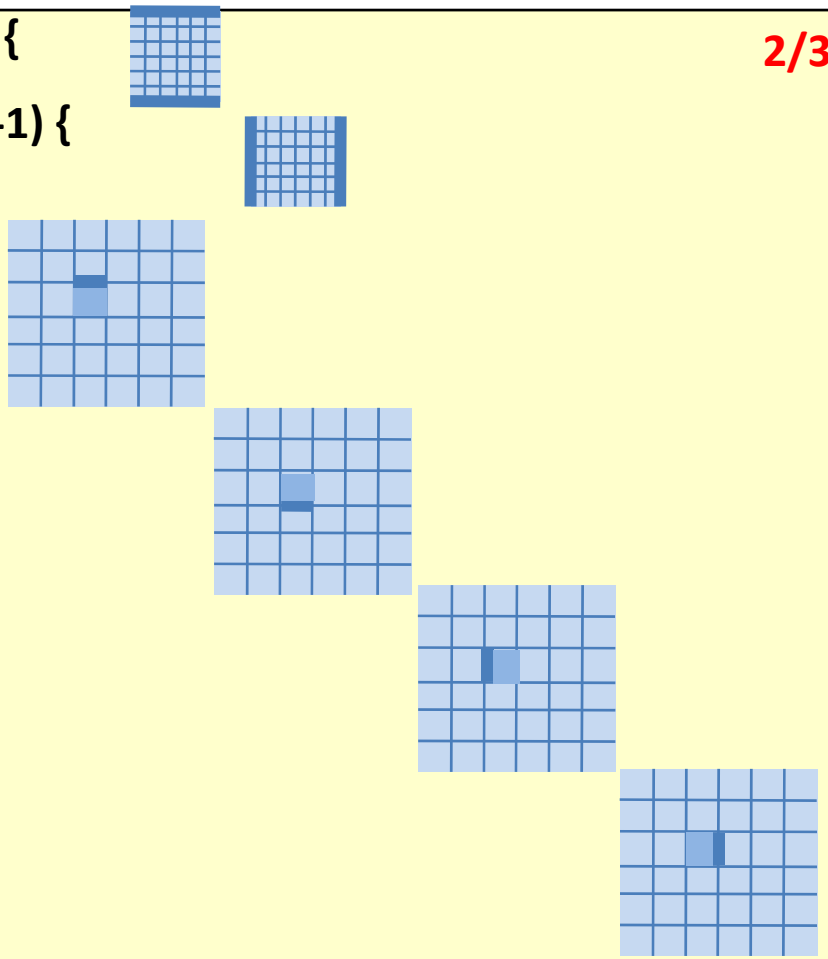
```
__global__ void GPUKernel_RelaxStepV2(int OI, int NI)      // This is a CUDA "kernel"      1/3
{
  int block_row, global_row;
  int block_col, global_col;
  float result;
  __shared__ float V[BLOCK_SIZE_Y][BLOCK_SIZE_X];

  // Compute the line and row processed by the thread
  block_row  = threadIdx.y;
  global_row = block_row + blockIdx.y*BLOCK_SIZE_Y;
  block_col  = threadIdx.x;
  global_col = block_col + blockIdx.x*BLOCK_SIZE_X;

  // If the associated grid element does not exist: end of the computation
  if (global_row >= SIDE_ROW || global_col >= SIDE_COL) {
    __syncthreads();

  // Get the value into the shared memory of the block and synchronize
  } else {
    V[block_row][block_col] = GPU_V[OI][global_row][global_col];
    __syncthreads();

                                                          // Next -->
```

Ex. simple de code MPI+CUDA (relax.)

MPI+CUDA : calculs sur un nœud GPU

2/3

```
    if (global_row == 0 || global_row == SIDE_ROW-1) {
      result = V[block_row][block_col];
    } else if (global_col == 0 || global_col == SIDE_COL-1) {
      result = V[block_row][block_col];
    } else {
      float up, bottom, left, right;
      if (block_row == 0)
        up = GPU_V[OI][global_row-1][global_col];
      else
        up = V[block_row-1][block_col];
      if (block_row == BLOCK_SIZE_Y-1)
        bottom = GPU_V[OI][global_row+1][global_col];
      else
        bottom = V[block_row+1][block_col];
      if (block_col == 0)
        left = GPU_V[OI][global_row][global_col-1];
      else
        left = V[block_row][block_col-1];
      if (block_col == BLOCK_SIZE_X-1)
        right = GPU_V[OI][global_row][global_col+1];
      else
        right = V[block_row][block_col+1];
      result = up + bottom + left + right;
      result /= 4.0;
    }
    GPU_V[NI][global_row][global_col] = result;
  }
}
```



Ex. simple de code MPI+CUDA (relax.)

MPI+CUDA : Exécution du *kernel*

3/3

```
int cycle;                          // Cycle counter.
dim3 Dg, Db;                        // Grid and Block configurations

// Set the grid and block configurations for the run of a kernel, and run the kernel.
Db.x = BLOCK_SIZE_X;
Db.y = BLOCK_SIZE_Y;
Db.z = 1;

Dg.x = SIDE_COL/BLOCK_SIZE_X;
if (SIDE_COL % BLOCK_SIZE_X != 0)
   Dg.x++;
Dg.y = SIDE_ROW/BLOCK_SIZE_Y;
if (SIDE_ROW % BLOCK_SIZE_Y != 0)
   Dg.y++;
Dg.z = 1;
```

Début de coordination threads-CUDA / comm-MPI

```
for (cycle = 0; cycle < NbCycles; cycle++) {
    GPUKernel_RelaxStepV2<<< Dg,Db >>>(cycle%2,(cycle+1)%2);  // - GPU computations
    (*gpuCommFctPt)((cycle+1)%2,cycle);                       // - MPI comms.
}
```

Ex. simple de code MPI+CUDA (relax.)

MPI+CUDA : Echange de frontières sur un anneau de GPU – v1- synchrone

1/3

```
if (Me == 0) {
  ...
} else if (Me == NbPr - 1) {
  ...
} else if (Me % 2 == 0) {
    cudaMemcpyFromSymbol(&BuffLineMinSend[0],GPU_V, sizeof(float)*(SIDE_COL),
                         (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                          (GPU_PADDED_SIDE(SIDE_COL))*(1))*sizeof(float),
                         cudaMemcpyDeviceToHost);
    cudaMemcpyFromSymbol(&BuffLineMaxSend[0],GPU_V, sizeof(float)*(SIDE_COL),
                         (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                          (GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW-2))*sizeof(float),
                         cudaMemcpyDeviceToHost);
    MPI_Sendrecv(&BuffLineMaxSend[0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                 &BuffLineMaxRecv[0],SIDE_COL,MPI_FLOAT,Me+1,tag,
                 MPI_COMM_WORLD,&status);
    MPI_Sendrecv(&BuffLineMinSend[0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                 &BuffLineMinRecv[0],SIDE_COL,MPI_FLOAT,Me-1,tag,
                 MPI_COMM_WORLD,&status);
    cudaMemcpyToSymbol(GPU_V,&BuffLineMinRecv[0], sizeof(float)*(SIDE_COL),
                       (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                        (GPU_PADDED_SIDE(SIDE_COL))*(0))*sizeof(float),
                       cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(GPU_V,&BuffLineMaxRecv[0], sizeof(float)*(SIDE_COL),
                       (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                        (GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW-1))*sizeof(float),
                       cudaMemcpyHostToDevice);
  } else if (Me % 2 == 1) {
    ...
  }
 }
}
```

Transferts données GPU → buffers CPU. quand les execs. de *kernels* sont finies.

On transfère des buffers de frontières alloués sur le CPU.

Transferts buffers CPU → données GPU. quand les communications sont finies.

→ Impact de CUDA (oui!)

Ex. simple de code MPI+CUDA (relax.)

MPI+CUDA : Echange de frontières sur un anneau de GPU – v2- asynchrone

1/3

```
if (Me == 0) {
  ...
} else if (Me == NbPr - 1) {
  ...
} else {
   MPI_Irecv(&BuffLineMinRecv[0],SIDE_COL,MPI_FLOAT,Me-1,tag,
             MPI_COMM_WORLD,&RecvRequest[0]);
   MPI_Irecv(&BuffLineMaxRecv[0],SIDE_COL,MPI_FLOAT,Me+1,tag,
             MPI_COMM_WORLD,&RecvRequest[1]);
   cudaMemcpyFromSymbol(&BuffLineMinSend[0],GPU_V,sizeof(float)*(SIDE_COL),
                        (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                         (GPU_PADDED_SIDE(SIDE_COL))*(1))*sizeof(float),
                        cudaMemcpyDeviceToHost);
   cudaMemcpyFromSymbol(&BuffLineMaxSend[0],GPU_V,sizeof(float)*(SIDE_COL),
                        (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                         (GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW-2))*sizeof(float),
                        cudaMemcpyDeviceToHost);
   MPI_Issend(&BuffLineMinSend[0],SIDE_COL,MPI_FLOAT,Me-1,tag,
              MPI_COMM_WORLD,&SendRequest[0]);
   MPI_Issend(&BuffLineMaxSend[0],SIDE_COL,MPI_FLOAT,Me+1,tag,
              MPI_COMM_WORLD,&SendRequest[1]);
   MPI_Waitall(2,RecvRequest,RecvStatus);
   cudaMemcpyToSymbol(GPU_V,&BuffLineMinRecv[0],sizeof(float)*(SIDE_COL),
                      (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                       (GPU_PADDED_SIDE(SIDE_COL))*(0))*sizeof(float),
                      cudaMemcpyHostToDevice);
   cudaMemcpyToSymbol(GPU_V,&BuffLineMaxRecv[0],sizeof(float)*(SIDE_COL),
                      (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                       (GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW-1))*sizeof(float),
                      cudaMemcpyHostToDevice);
   MPI_Waitall(2,SendRequest,SendStatus);
  }
```

On reçoit dans des buffers de frontières alloués sur le CPU.

données GPU → buffers CPU quand execs. *kernels* finies.

On envoie des buffers de frontières alloués sur le CPU.

Attente données reçues buffers CPU → données GPU. quand comms. terminées.

Attente données envoyées (pas optimal) → Impact de CUDA (oui!)

Ex. simple de code MPI+CUDA (relax.)

MPI+CUDA : Echange de frontières sur un anneau de GPU – v3-async-pers.

```
if (Me == 0) {
   MPI_Ssend_init(&BuffLineMaxSend[0],SIDE_COL,MPI_FLOAT,Me+1,0,
            MPI_COMM_WORLD,&SendRequest[1]);
   MPI_Recv_init(&BuffLineMaxRecv[0],SIDE_COL,MPI_FLOAT,Me+1,0,
            MPI_COMM_WORLD,&RecvRequest[1]);
 } else if (Me == NbPr-1) {
   MPI_Ssend_init(&BuffLineMinSend[0],SIDE_COL,MPI_FLOAT,Me-1,0,
            MPI_COMM_WORLD,&SendRequest[0]);
   MPI_Recv_init(&BuffLineMinRecv[0],SIDE_COL,MPI_FLOAT,Me-1,0,
                  MPI_COMM_WORLD,&RecvRequest[0]);
 } else {
   MPI_Ssend_init(&BuffLineMinSend[0],SIDE_COL,MPI_FLOAT,Me-1,0,
            MPI_COMM_WORLD,&SendRequest[0]);
   MPI_Ssend_init(&BuffLineMaxSend[0],SIDE_COL,MPI_FLOAT,Me+1,0,
            MPI_COMM_WORLD,&SendRequest[1]);
   MPI_Recv_init(&BuffLineMinRecv[0],SIDE_COL,MPI_FLOAT,Me-1,0,
                  MPI_COMM_WORLD,&RecvRequest[0]);
   MPI_Recv_init(&BuffLineMaxRecv[0],SIDE_COL,MPI_FLOAT,Me+1,0,
            MPI_COMM_WORLD,&RecvRequest[1]);
 }
```


Initialisation des schémas de communications persistants

On planifie des transferts de « buffers de frontières » alloués sur le CPU

→ Impact de CUDA

Ex. simple de code MPI+CUDA (relax.)

MPI+CUDA : Echange de frontières sur un anneau de GPU – v3-async-pers.

2/3

```
if (Me == 0) {
  ...
} else if (Me == NbPr - 1) {
  ...
} else {
  MPI_Start(&RecvRequest[0]);
  MPI_Start(&RecvRequest[1]);
  cudaMemcpyFromSymbol(&BuffLineMinSend[0],GPU_V,
                       sizeof(float)*(SIDE_COL),
                       (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                       (GPU_PADDED_SIDE(SIDE_COL))*(1))*sizeof(float),
                       cudaMemcpyDeviceToHost);
  cudaMemcpyFromSymbol(&BuffLineMaxSend[0],GPU_V,
                       sizeof(float)*(SIDE_COL),
                       (idx*(GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW) +
                       (GPU_PADDED_SIDE(SIDE_COL))*(SIDE_ROW-2))*sizeof(float),
                       cudaMemcpyDeviceToHost);
  MPI_Start(&SendRequest[0]);
  MPI_Start(&SendRequest[1]);
  MPI_Waitall(2,RecvRequest,RecvStatus);
  cudaMemcpyToSymbol(...);
  cudaMemcpyToSymbol(...);
  MPI_Waitall(2,SendRequest,SendStatus);
 }
```

Exploitation des schémas de communications persistants

Transferts données GPU → buffers CPU. quand les execs. de *kernels* sont finies.

Transferts buffers CPU → données GPU. quand les comms. sont finies.

→ Impact de CUDA (oui!)

Ex. simple de code MPI+CUDA (relax.)

MPI+CUDA : Echange de frontières sur un anneau de GPU – v3-async-pers.

3/3

```
if (Me == 0) {
    MPI_Request_free(&SendRequest[1]);
    MPI_Request_free(&RecvRequest[1]);
  } else if (Me == NbPr-1) {
    MPI_Request_free(&SendRequest[0]);
    MPI_Request_free(&RecvRequest[0]);
  } else {
    MPI_Request_free(&SendRequest[0]);
    MPI_Request_free(&RecvRequest[0]);
    MPI_Request_free(&SendRequest[1]);
    MPI_Request_free(&RecvRequest[1]);
  }
```

Libération des schémas de communications persistants

Plan

1. Principes de parallélisation sur cluster de CPU+GPU
2. Ex 1 : application distribuée *embarrassingly parallel*
3. Ex 2 : 1ère application distribuée fortement couplée
4. Ex 3 : 2ème application distribuée fortement couplée
5. Parallélisation simultanée sur CPUs et GPUs
6. Performances
7. Mesures et modélisation énergétiques
8. Conclusion

Ex 3 : application distribuée fortement couplée

Transport 3D d'espèces chimiques dans un cours d'eau : résolution d'un système d'EDP (advection, diffusion, réaction)

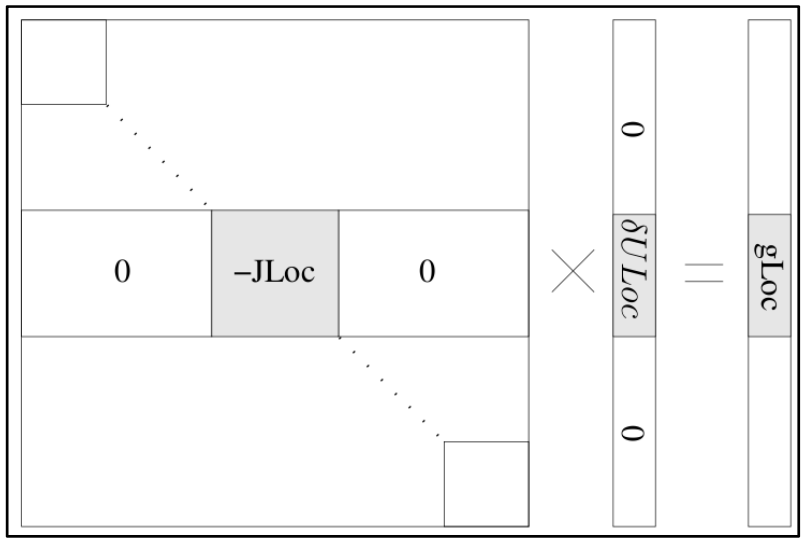
- Résolution :

  système discrétisé, linéarisé → un gros système linéaire « J.δU = g » à résoudre à chaque pas de temps (matrice J considérée cte.).

- Parallélisation :

  méthode *multisplitting Newton* (≈ décomposition par blocs) :

  - Un sous-domaine par nœud,
  - Résolution d'un système plus petit sur chaque noeud.
  - Solveur linéaire sur CPU ou GPU, sur chaque nœud.
  - Communications après la résolution de chaque petit système



Ex 3 : application distribuée fortement couplée

Implémentation *synchrone* MPI+CUDA :

1. CPU : Calcul de J et du vecteur initial sur CPU chaque nœud
2. Transfert : copie sur GPU sur chaque nœud
3. Pour chaque pas de temps :
   a. Transfert : transfert de vecteur CPU → GPU
   b. GPU : résolution du système linéaire sur GPU
   c. Transfert : transfert de vecteur GPU → CPU
   d. CPU : mise à jour du vecteur local sur CPU
   e. MPI: échange des dépendances (*Issend, Irecv*)
   f. CPU : mise à jour complémentaire sur CPU
   g. MPI: détection de convergence globale (*Gatherv*)

Implémentation a*synchrone* MPI+CUDA : en cours

Adaptation d'une méthode de calcul sur grille…

Plan

1. Principes de parallélisation sur cluster de CPU+GPU
2. Ex 1 : application distribuée *embarrassingly parallel*
3. Ex 2 : 1ère application distribuée fortement couplée
4. Ex 3 : 2ème application distribuée fortement couplée
5. Parallélisation simultanée sur CPUs et GPUs
6. Performances
7. Mesures et modélisation énergétiques
8. Conclusion

Exemple de code OpenMP+CUDA

Expérimentation sur un code d'analyse de sous-sol par FFT à la **CGGVeritas** sur un nœud CPU+GPU :

- un thread OpenMP appelle les routines CUDA, les autres traitent les tâches sur des cœurs CPU,
- équilibrage de charge dynamique,
- nouveau temps d'exécution beaucoup plus faible.

Ça marche

Exemple de code MPI+Pthreads+CUDA

Expérimentation sur le solver d'EDP sur cluster de CPU+GPU :

- mesure de temps d'exécution sur 1 à 8 cœurs CPU,
- mesure de temps d'exécution sur le GPU,
- équilibrage de charge statique,
- mesure du nouveau temps d'exec. : gain très faible !
- recherche en cours de la source de perte de performance :
  - équilibrage de charge statique insuffisant ?
  - perturbation due à OpenMPI ?
  - …

Ça ne marche pas (encore) bien…

Plan

1. Principes de parallélisation sur cluster de CPU+GPU
2. Ex 1 : application distribuée *embarrassingly parallel*
3. Ex 2 : 1ère application distribuée fortement couplée
4. Ex 3 : 2ème application distribuée fortement couplée
5. Parallélisation simultanée sur CPUs et GPUs
6. Performances
7. Mesures et modélisation énergétiques
8. Conclusion

Performances – Pricer (1)

Comparaison en vitesse à un pgm séquentiel sur 1 cœur CPU

- Une première méthode de quantification les performances.
- En général on exhibe d'excellents résultats.
- Ex : pricer d'option exotiques Européennes

SU(16 nœuds GPU vs 1 cœur CPU) = 1636

Comparaison en vitesse à un pgm parallèle sur 1 nœud CPU multicoeurs

- Une méthode plus proche des préoccupations de l'utilisateur
- Les résultats chutent ☺
- Ex pour le pricer comparé à un CPU dual-cœur programmé en OpenMP:

SU(16 nœuds GPU vs 1 noeud CPU dual-coeur) = 707

Chute de plus de 50% (!)

Performances – Pricer (2)

Comparaison en vitesse à un pgm // sur 1 cluster de CPU multicoeurs

- La méthode qui intéresse l'utilisateur d'un cluster de CPUs
- Les résultats chutent ☺
- Ex pour le pricer comparé à un cluster de CPU dual-cœur avec MPI+OpenMP:

  SU(16 nœuds GPU vs 16 noeuds CPU dual-coeur) = 48

Comparaison de la précision des résultats

- Sur une nouvelle architecture, avec de nouvelles bibliothèques de calculs
  → il convient de comparer/vérifier la précision des calculs.
- Ex pour le pricer :

  Pas de différences (précision identique avec les mêmes calculs)
  Inutile de calculer plus de trajectoires de Monte-Carlo
- Ex pour un solver d'EDP :

  Des différences : problème sensibles aux arrondis, à l'enchaînement des op…
  Quel programme sur cluster de GPUs pour des résultats satisfaisants ?

Performances – Pricer (3)

Comparaison en énergie à un pgm // sur 1 cluster de CPU multicoeurs

- Celui qui paye les calculs peut y être très sensible ☺
- Le pgm le plus rapide est-il le plus économique ?
- Que prendre en compte pour mesurer la consommation énergétique ?
  - les nœuds utilisés
  - le switch du cluster …. Partiellement ou en totalité ?
  - la climatisation ? Si on peut isoler la partie concernant le cluster…
  - les serveurs de fichiers ?
- Ex pour le pricer comparé à un cluster de CPU dual-cœur avec MPI+OpenMP:

Cluster de 16 nœuds CPU dual-cœur (dans un cluster de 256 nœuds) : 934.8Wh
en *992s*

Cluster de 16 nœuds CPU+GPU : 16.4Wh
en *21s*

nœuds utilisés + switch en totalité (cluster utilisé en mode exclusif)
rmq : le switch dissipe 1720W

→ Gain énergétique d'un facteur 57*, et gain en temps d'un facteur 48*

Performances – Pricer (4)

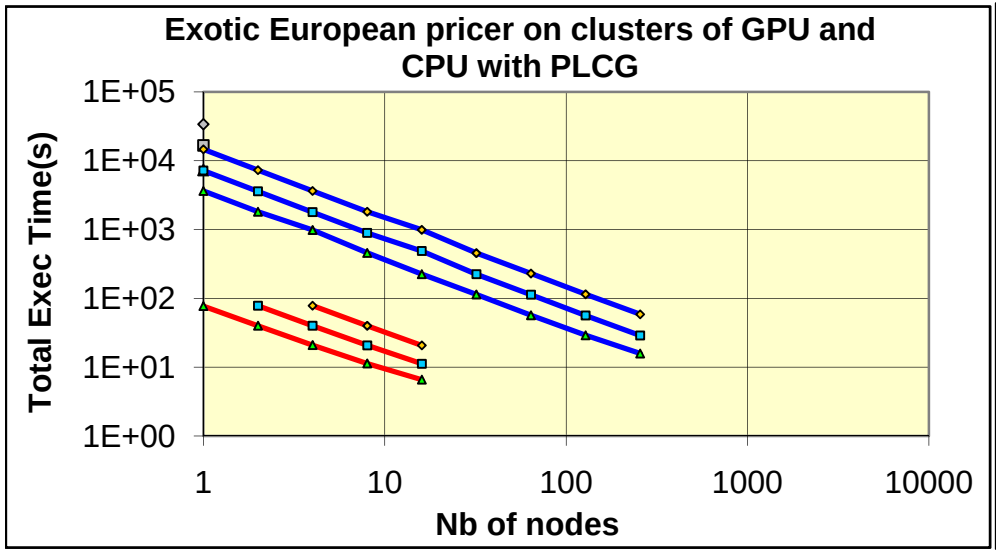
Point de fonctionnement optimal de chaque système (archi + pgm) ?

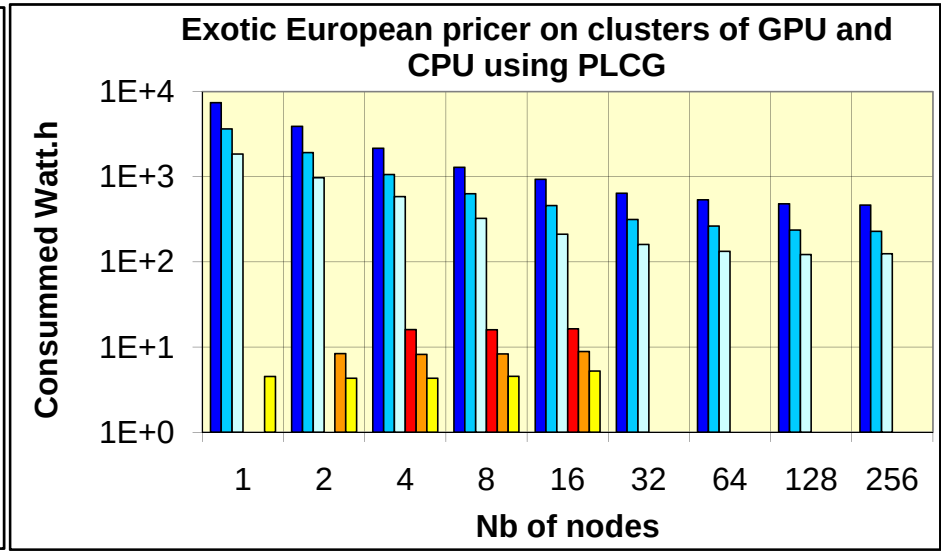
Faut-il comparer N nœuds CPU avec N nœuds GPU ?

Les points de fonctionnement optimaux pourraient être différents.

→ **Comparer les courbes de performances des deux clusters**

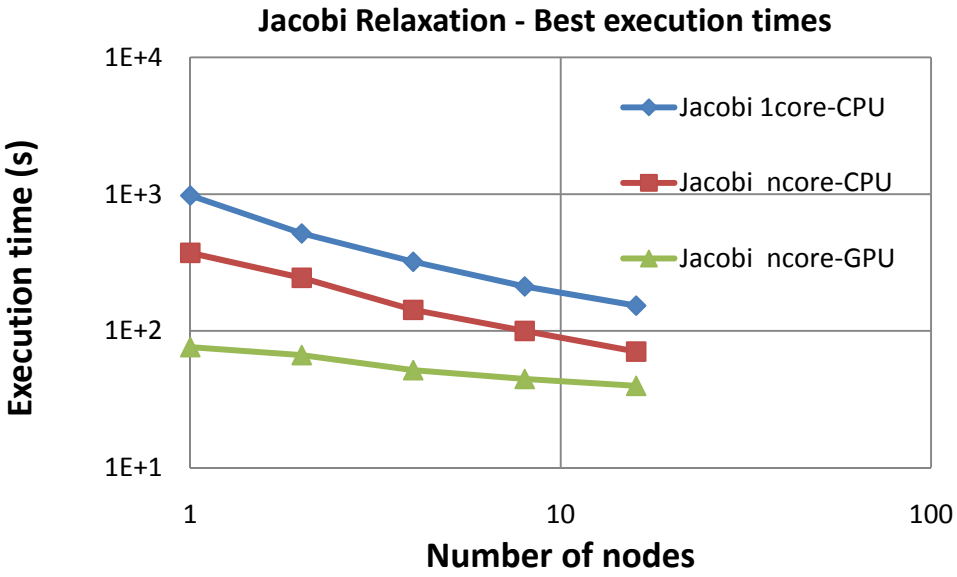
Ex du pricer : sans surprises car « *embarrassingly parallel* »





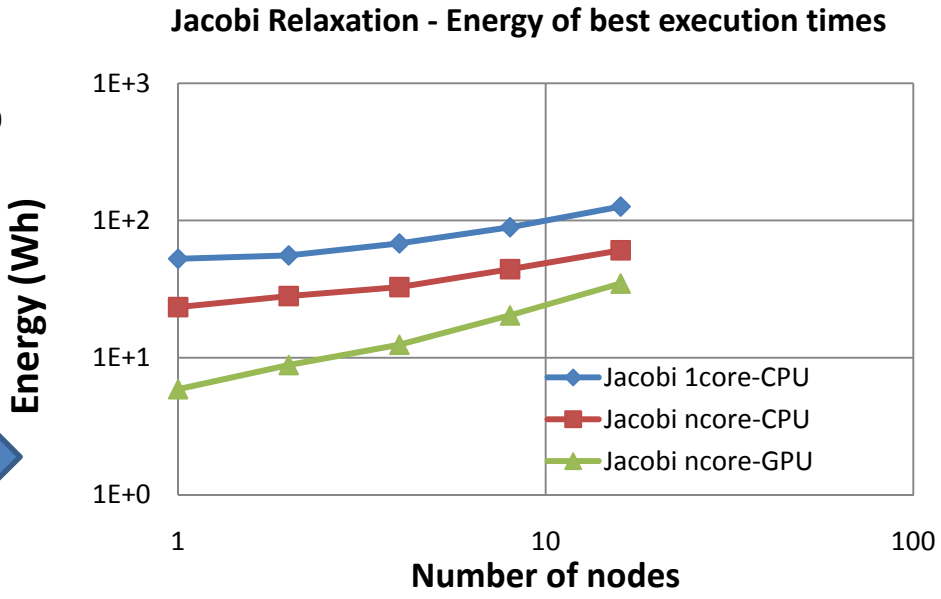
Performances – Relax. de Jacobi (1)

Performances de la relaxation de Jacobi (pb fortement couplé)



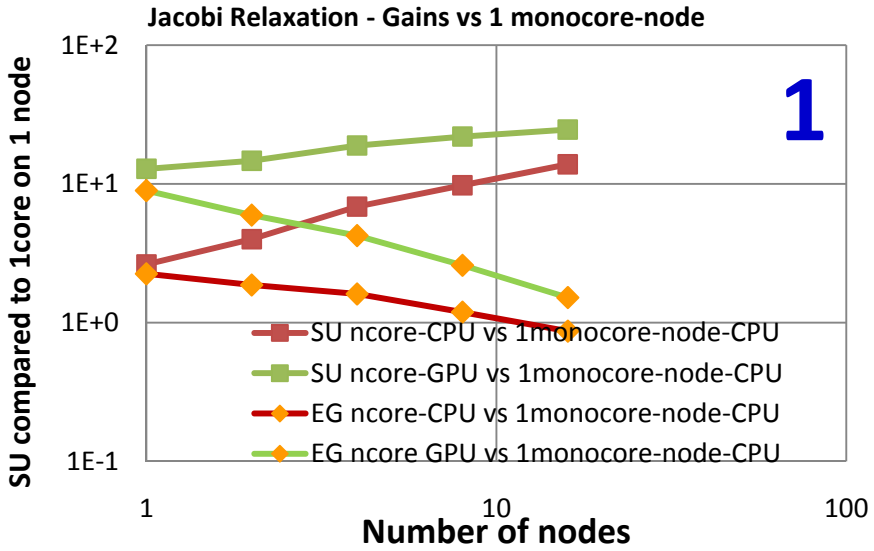
Mesure du temps d'exécution

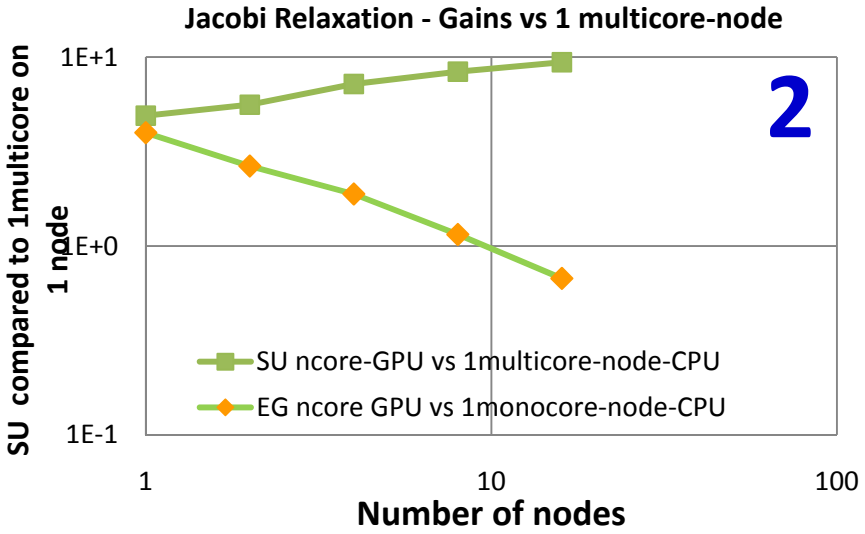
Mesures de l'énergie consommée

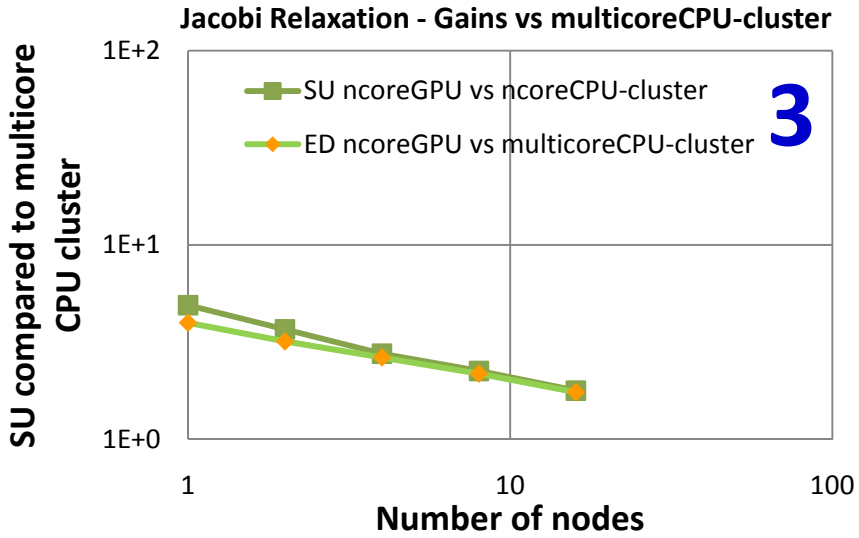


Performances – Relax. de Jacobi (2)

Performances de la relaxation de Jacobi (pb fortement couplé)







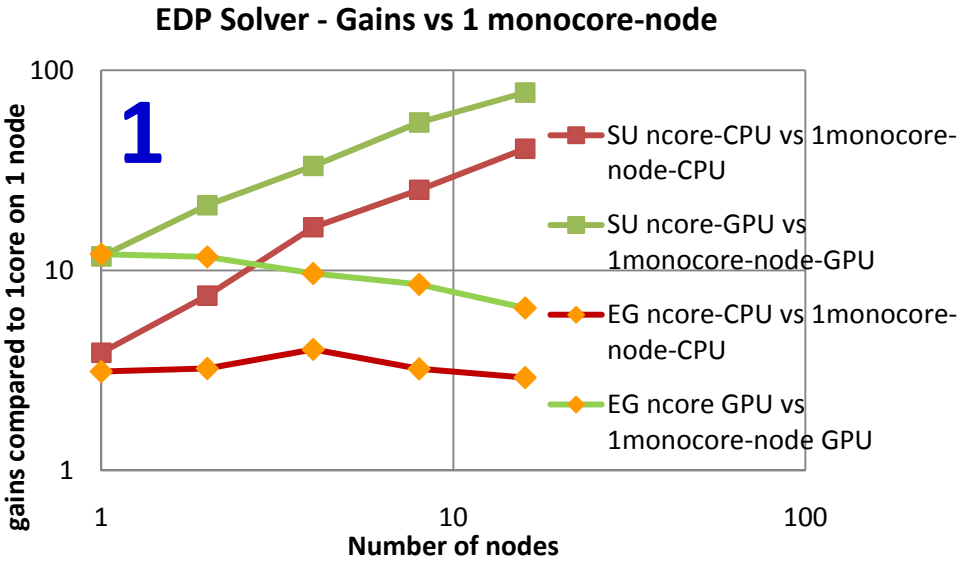
Bilan :

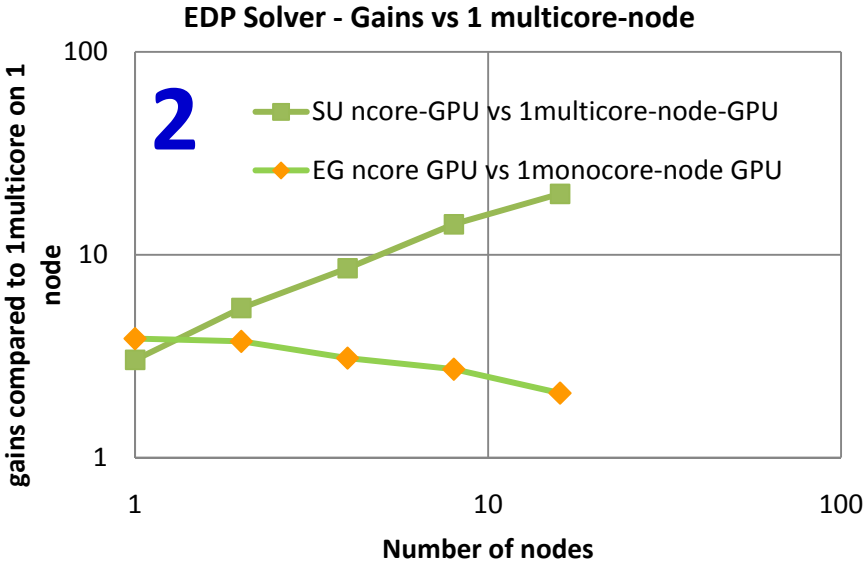
- problème distribué fortement couplé,
- bon comportement sur cluster de CPUs,
- bon comportement sur cluster de GPUs,
- comportements de plus en plus proches quand le nombre de nœuds augmente.

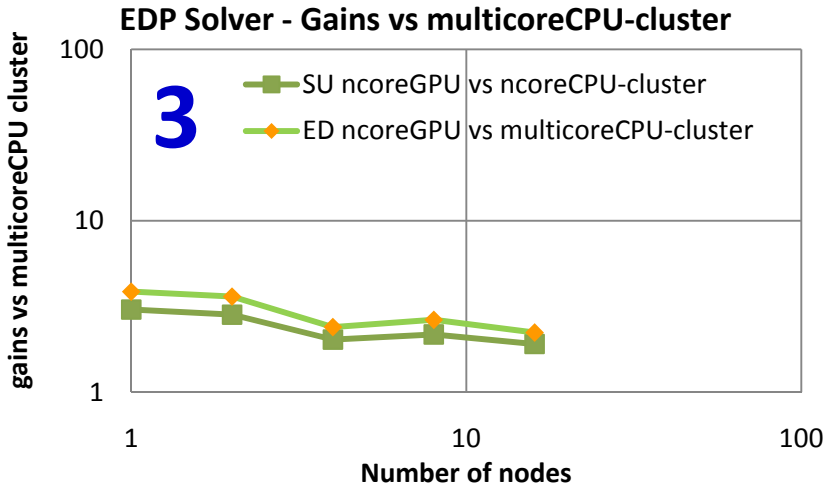
→ **Résultats très différents du cas faiblement couplé.**

Performances – Solveur EDP (1)

Performances du solveur d'EDP (pb fortement couplé)







Domaine 30x30x30, 2 espèces chimiques…

Au final, sur 16 nœuds le cluster de GPU (*GT285*) n'est que 1.85 fois plus rapide que le cluster de CPU (*Nehalem 4 cœurs hyperthreadés*)

→ **Motivation pour utiliser à la fois les CPUs et les GPUs!**

Plan

1. Principes de parallélisation sur cluster de CPU+GPU
2. Ex 1 : application distribuée *embarrassingly parallel*
3. Ex 2 : 1ère application distribuée fortement couplée
4. Ex 3 : 2ème application distribuée fortement couplée
5. Parallélisation simultanée sur CPUs et GPUs
6. Performances
7. Mesures et modélisation énergétiques
8. Conclusion

Considérations énergétiques

- Les GPUs consomment autant/plus de *puissance* (Watts) que les CPUs

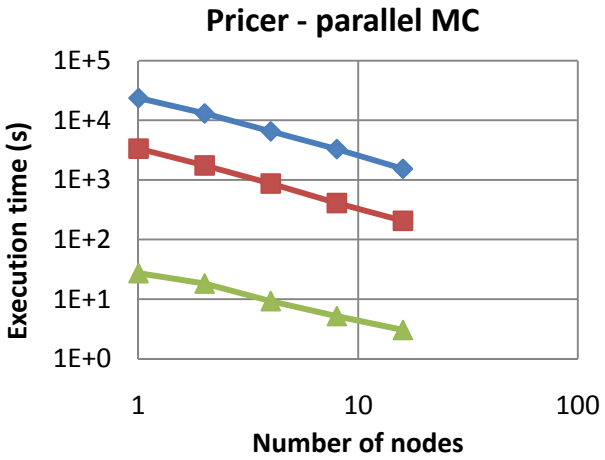
  → *la puissance consommée d'un cluster de CPU+GPU augmentera (vs un cluster de CPUs)*

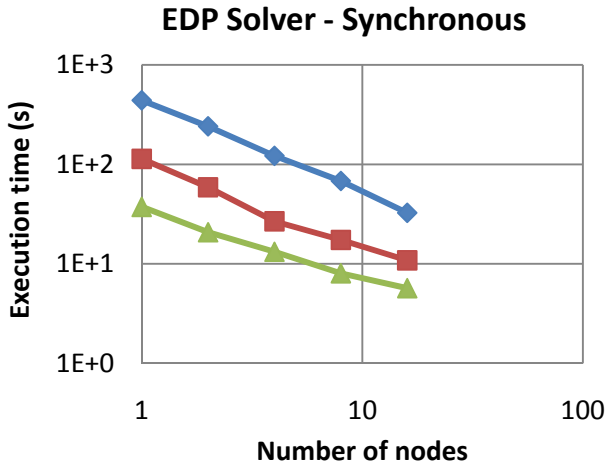
  → *attention : souscrire plus de puissance coûte plus cher!*

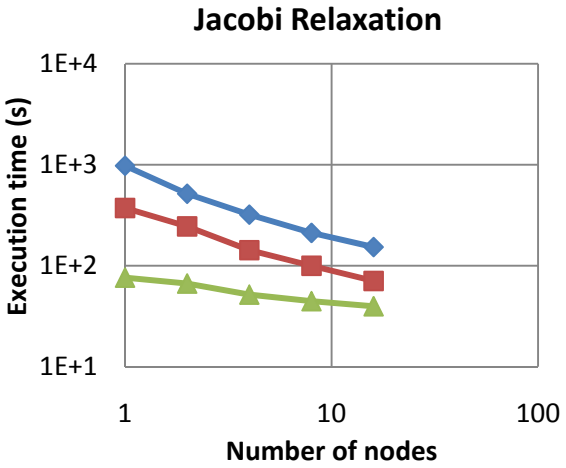
- En réduisant les temps d'exécution on peut consommer :
  - ✓ *moins d'énergie* (WattHeures) : plus rapide & plus éco
  - ✓ *autant d'énergie* (WattHeures) : plus rapide
  - ✓ *plus d'énergie* (WattHeures) : plus rapide & moins éco

- Les performances relatives d'un cluster de CPU multi-cœurs et d'un cluster de GPUs peuvent varier avec le nombre de nœuds, la taille du problème, l'algorithme…

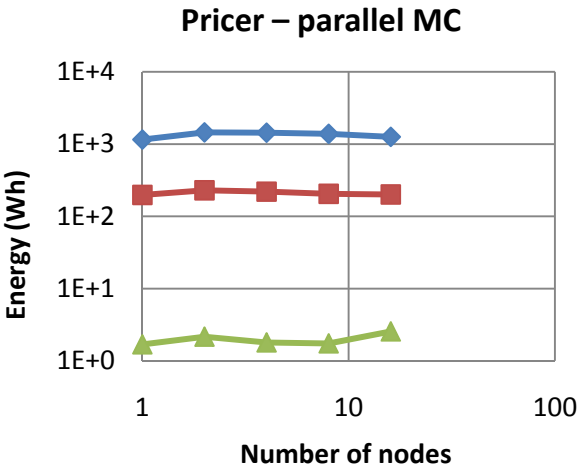
Considérations énergétiques

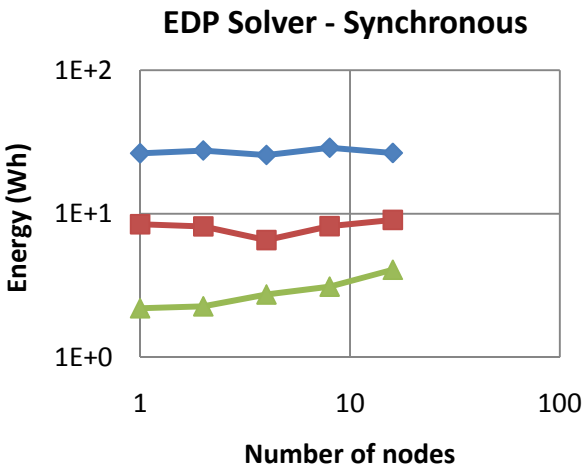


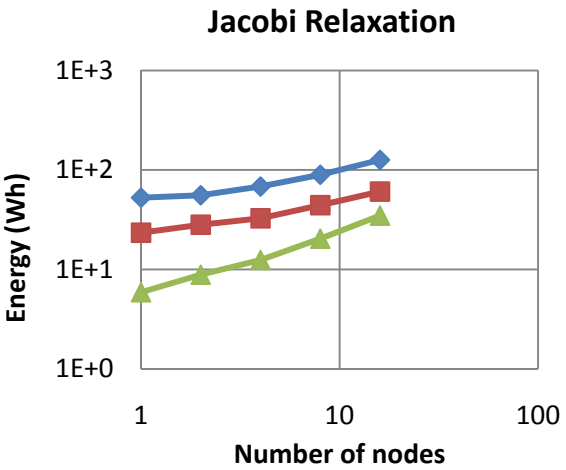




Monocore-CPU cluster Multicore-CPU cluster Manycore-GPU cluster



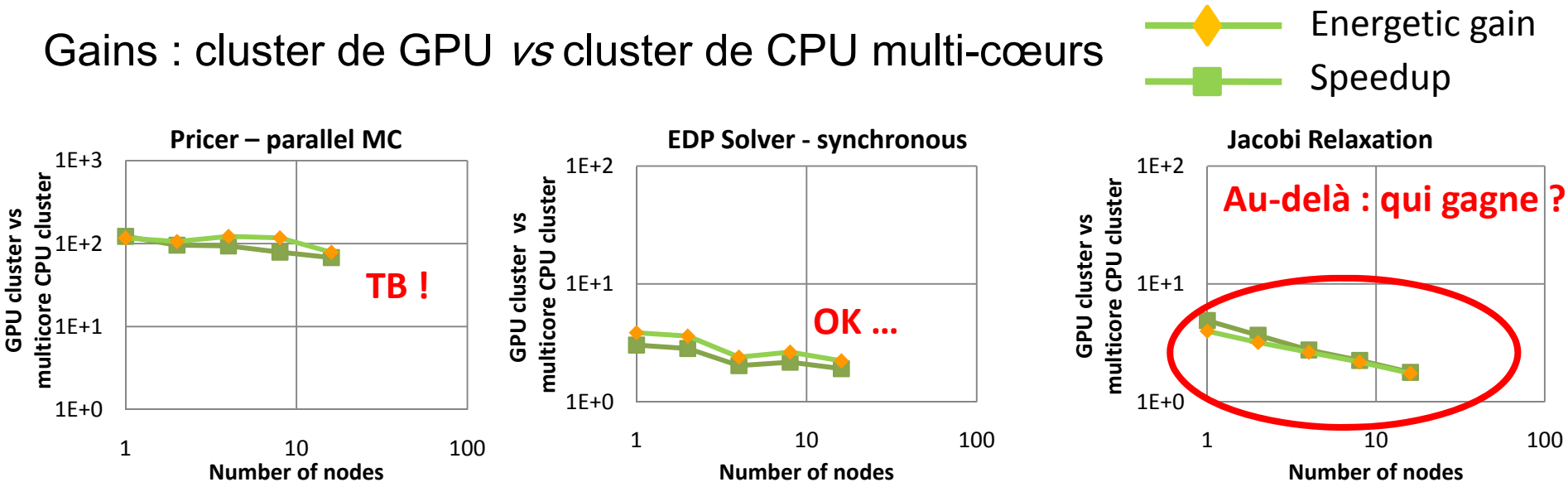




Considérations énergétiques

Quel est le cluster le plus intéressant ?

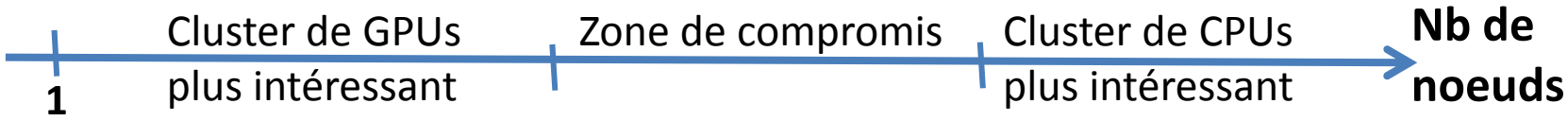
Gains : cluster de GPU *vs* cluster de CPU multi-cœurs



Hypothèse : si on suppose qu'il y a « scalabilité »

→ Modélisation … *(publication en cours)*

→ Il existe des seuils de basculement d'intérêt entre clusters de CPUs et de GPUs : il faut choisir la solution optimale.



Plan

1. Principes de parallélisation sur cluster de CPU+GPU
2. Ex 1 : application distribuée *embarrassingly parallel*
3. Ex 2 : 1ère application distribuée fortement couplée
4. Ex 3 : 2ème application distribuée fortement couplée
5. Parallélisation simultanée sur CPUs et GPUs
6. Performances
7. Mesures et modélisation énergétiques
8. Conclusion

Conclusion

Cluster de CPUs mono-coeurs → MPI

Cluster de CPUs multi-coeurs → MPI ou
MPI + threads-CPU

Cluster de CPUs multi-coeurs + GPUs *many*-coeurs → MPI + threads-GPU ou
MPI + threads-CPU + threads-GPU

Plus génériquement :

Clusters hétérogènes (ou hybrides) : noeuds "CPU+accélérateurs" (accélérateurs : unités SSE, GPU, Larabee, Cell, FPGA...)

→ Algorithmique parallèle multi-grains
→Programmation parallèle multi-paradigmes

A ce jour : complexité de développement croissante, et performances pas toujours « aussi croissantes »

Perspectives

- Le « TOP5 » contient des machines hétérogènes
- "On ne passera pas l'Exaflops sans des architectures hétérogènes"...

  → Développer l'algorithmique multi-grains
  → Maîtriser la programmation multi-paradigmes
  → Développer de nouveaux environnements de programmation

- 2ème machine Top500 : 1.0 Pflops et 2.3 Mwatts (avec accélérateurs)
  1ère machine Top500 : 1.7 Pflops et 6.9 Mwatts (sans accélérateurs)
- Difficile d'imaginer passer l'Exaflops sans optimisations énergétiques

  → Recherche d'architectures, de bibliothèques et de codes optimisés en vitesse, ou en énergie, ou visant un compromis.

Expérience de développement d'applications sur cluster de GPUs

Questions ?

Stéphane Vialle

SUPELEC équipe IMS & EPI AlGorille

Avec l'aide de L. Abbas-Turki, T. Jost , W. Kirshenmann, P. Mercier, S. Contassot-Vivier, L. Plagne