



European Option Pricing on a GPU Cluster

Lokman Abbas-Turki, Stéphane Vialle

► **To cite this version:**

Lokman Abbas-Turki, Stéphane Vialle. European Option Pricing on a GPU Cluster. 2009. hal-00364242

HAL Id: hal-00364242

<https://hal-supelec.archives-ouvertes.fr/hal-00364242>

Preprint submitted on 25 Feb 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

European Option Pricing on a GPU Cluster

(ANR project « GCPMF »)



L. Abbas-Turki – ENPC

S. Vialle – SUPELEC & INRIA

With the help of P. Mercier (SUPELEC)
and B. Lapeyre (ENPC)

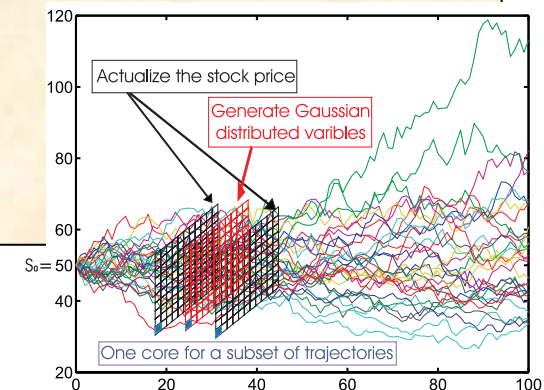
1 – Objectives

Main objectives and difficulties

Final objective:

High speed European contract pricing (for hedging)

- using Monte-Carlo computations,
- using a clusters of multi-cores
(GPUs or multi-core CPUs)



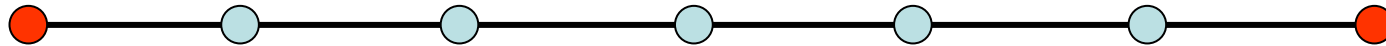
Scientific and technical locks:

- Design a parallel algorithm for an European option pricer on GPU and cluster of GPU,
- Design and implement a right and efficient parallel RNG,
- Find out the right compromise between speedup, size up, result accuracy and energy consumption

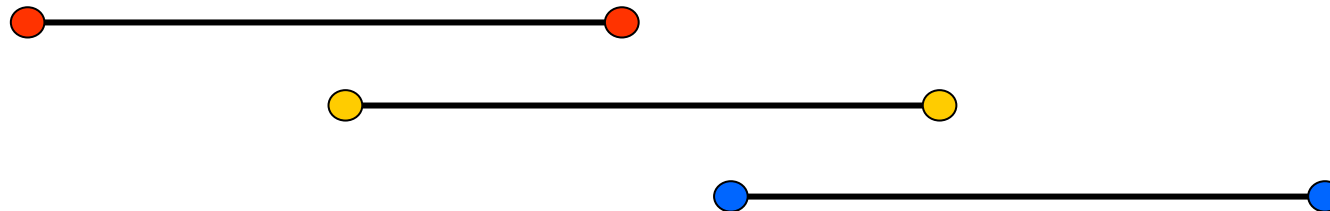
2 – RNG parallelization and comparison

Principles

- RNG on CPU vs RNG on GPU
 1. Data transfer
 2. Faster if Parallel
- Parallel RNGs: two efficient alternatives for GPU
 1. Period Splitting



2. Parametrization



Parallel RNG: PLCG

- Parallelization with parametrization

$$X_n = a.X_{n-1} \% m$$

- « a » is the parameter
- Chose an « a » for each sub-stream such that:
 1. Good parallel independence inter-streams
Michael Mascagni SPRNG
 2. Good sequential independence intra-stream
Donald E. Knuth Art of Computer Programming
 3. Total period: 2^{41} (int 32) and 2^{65} (float 64)

Parallel RNG: CMRG

- Pierre L'Ecuyer: Combination of two MRGs (total period: 2^{185})

$$X_n = a_1 * X_{n-1} + a_2 * X_{n-2} + a_3 * X_{n-3} \% m$$

$$X'_n = a'_1 * X'_{n-1} + a'_2 * X'_{n-2} + a'_3 * X'_{n-3} \% m'$$

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a_3 & a_2 & a_1 \end{pmatrix} \quad A' = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a'_3 & a'_2 & a'_1 \end{pmatrix}$$

For example, if we want:

- one RNG/trajectory,
- 2^{18} trajectories/GPU,
- maximum of $16 = 2^4$ GPUs.

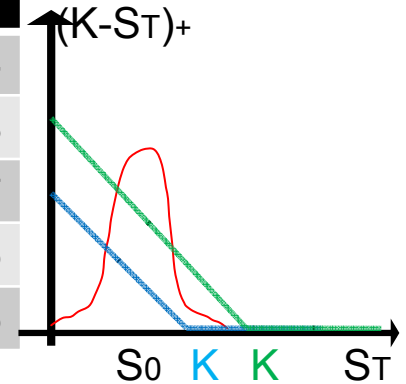
$$IP = 2^{185}/2^{22} = 2^{163} \text{ and } X_{_1} = (X_1, X_2, X_3)^T, X'_{_1} = (X'_1, X'_2, X'_3)^T$$

$$X_{_i} = (A^{IP})^i . X \% m \quad X'_{_i} = (A'^{IP})^i . X' \% m'$$

Parallel RNG: Results

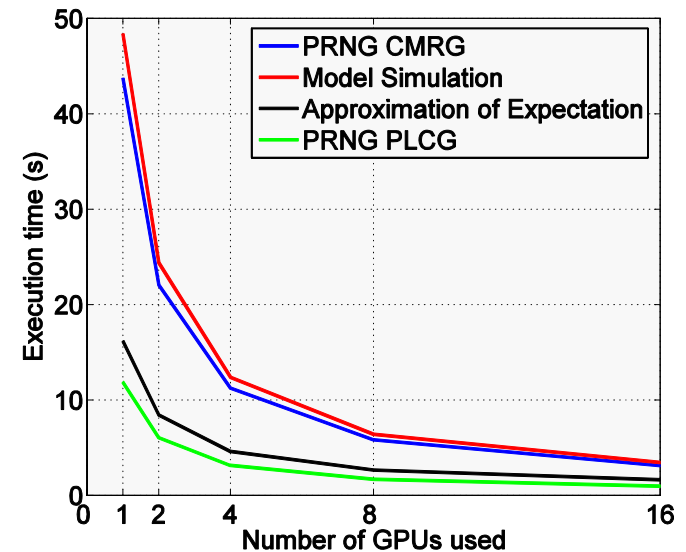
- Results Accuracy

Strike « K »	Real	PLCG MC	PLCG error	CMRG MC	CMRG error
80	0.0610	0.0621	0.0023	0.0611	0.0024
90	0.5068	0.5115	0.0078	0.5088	0.0008
100	2.1723	2.1862	0.0178	2.1740	0.0177
110	5.9208	5.9433	0.0299	5.9159	0.0296
120	11.8810	11.9023	0.0407	11.8775	0.0406



- Speedup Results for PRNG

RNG \ Machine	Gaussian sample for PLCG generation/second	Gaussian sample for CMRG generation/second
CPU	2.24 Kg/s	2.03 Kg/s
GPU	883.58 Kg/s	239.80 Kg/s



3 – Parallel algorithm and implementations

Parallel algorithm (1)

Parallel programming paradigms:

Coarse grain: message passing (MPI on PC cluster)

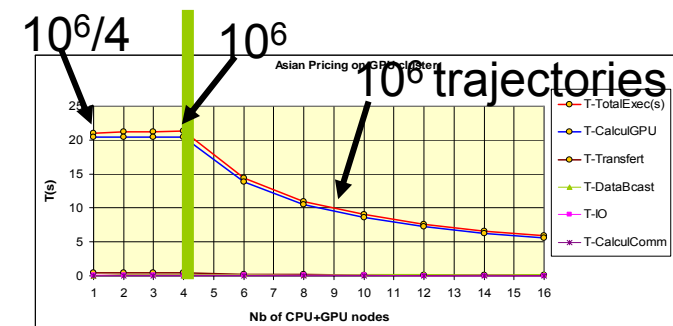
Medium grain: CPU-multithreading (OpenMP on multi-cores)

Fine grain: GPU-multithreading (CUDA on GPU)

Strategy:

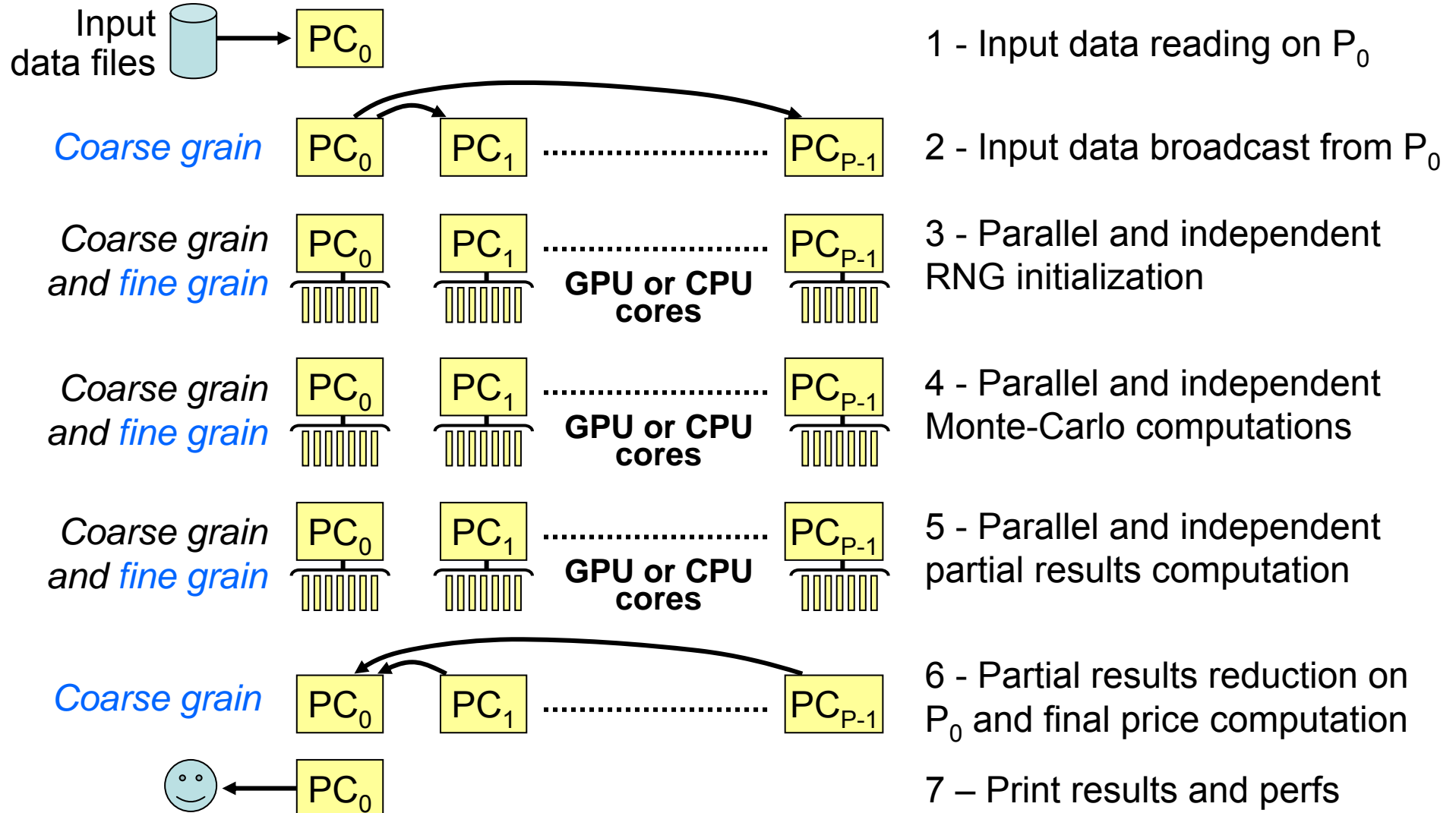
- Avoid concurrent input file accesses
- Minimize data transfer between CPU and GPU memories.
- Common algorithm for multi-core-CPU and GPU clusters.
- When limited by the GPU memory:

“size up + speedup”



Parallel algorithm and implementations

Parallel algorithm (2)



GPU/CPU code comparison (1)

OpenMP parallelization on multi-core CPU: split the external loop

```

void ActStock(double sqrttdt)
{
    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 = .....;
                    // - The passes that remain
                    for (int stock = 1; stock <= StkIdx ; stock++)
                        call = .....;
                    // Copy result in the global GPU memory
                    TabStockCPU[StkIdx][yIdx][xIdx] = call;
                }
        }
    }
}
float TabStockCPU[NbStocks][Ny][Nx]
    
```

GPU/CPU code comparison (2)

CUDA parallelization on GPU: one kernel work on one trajectory

```

__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 = .....;
    callBis = call;
    // The passes that remain
    for (int stock = 1; stock <= StkIdx; stock++) {
        GaussLine[xIdx] = TabGaussGPU[stock][yIdx][xIdx];
        call = callBis*.....;
        callBis = call;
    }
    // Copy result in the global GPU memory
    TabStockOutputGPU[StkIdx][yIdx][xIdx] = call;
}
float TabStockOutputGPU[NbStocks][Ny][Nx]
    
```

GPU/CPU code comparison (3)

CUDA parallelization on GPU: one kernel work on one trajectory

```

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
    }
}

float TabStockCPU[NbStocks][Ny][Nx];
float TabStockOutputGPU[NbStocks][Ny][Nx];
    
```

→ Identical data structures for CPU and GPU versions
when using one GPU-thread per trajectory

Compilation

OpenMPI + Cuda:

```
nvcc --host-compilation C++  
      -O3 -I/opt/openmpi/include  
      -DOMPI_SKIP_MPICXX -c X.cu  
  
nvcc -O3 -L/opt/openmpi/lib  
      -o pricer X.o Y.o .... -lmpi -lm
```

This is a basic C++ code.

OpenMPI + OpenMP

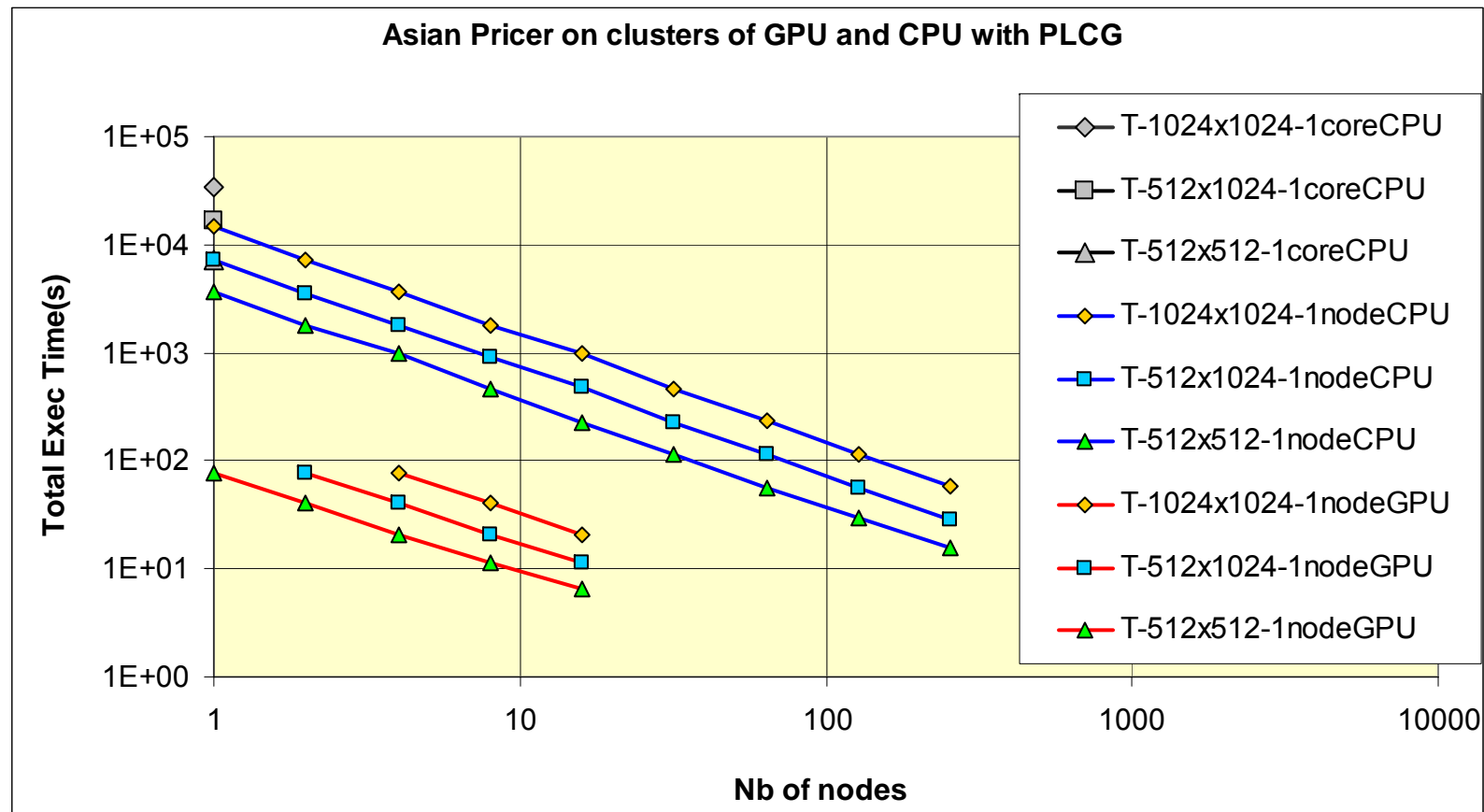
```
g++ -O3 -fopenmp -I/opt/openmpi/include  
     -c X.cc  
  
g++ -O3 -fopenmp -L/opt/openmpi/lib  
     -o pricer X.o Y.o ... -lmpi -lm
```

Compilation with CUDA code is easy.

4 – Experiments and performance analysis

Computing perf (1)

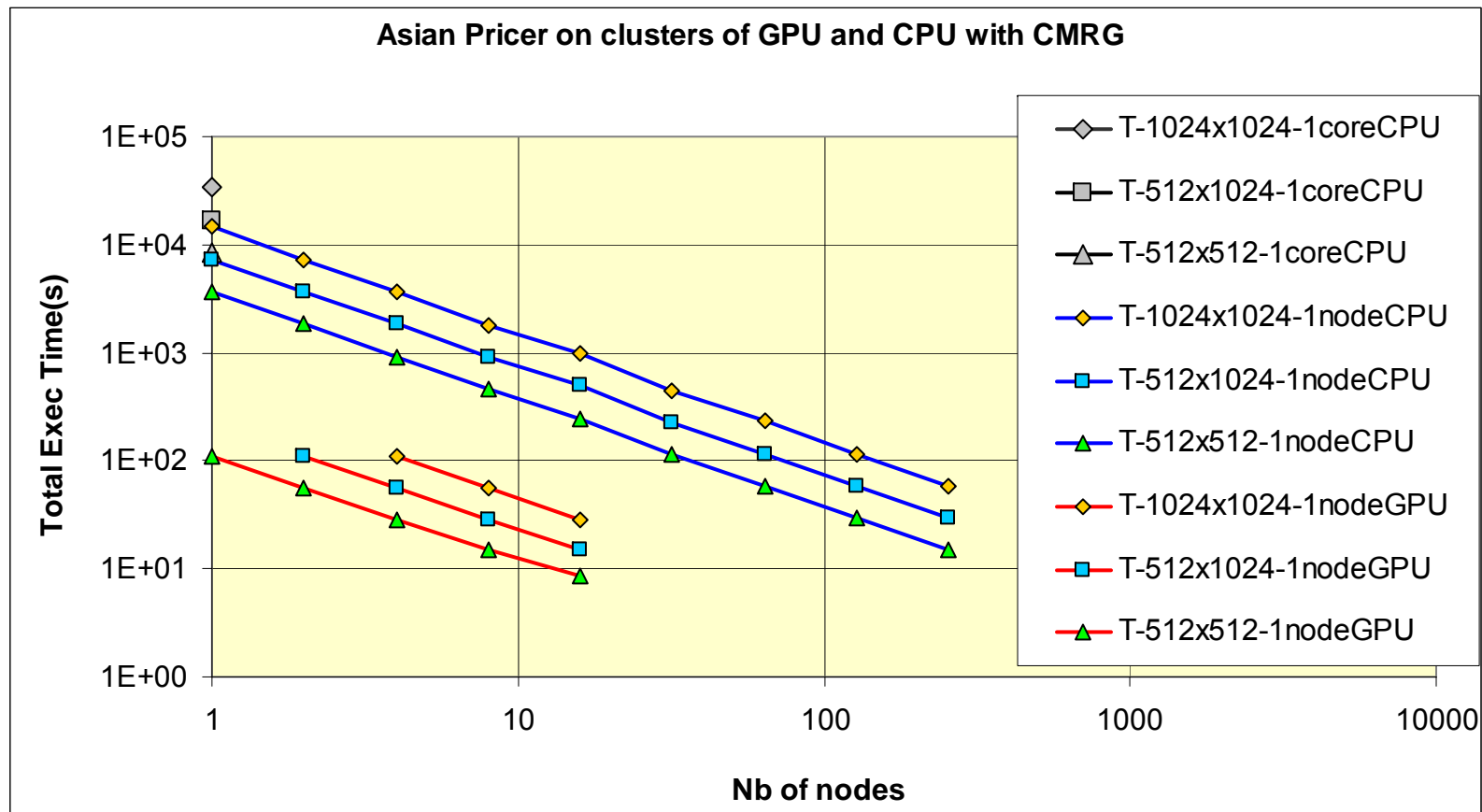
Pricing execution time with PLCG



Good scaling on both systems.

Computing perf (2)

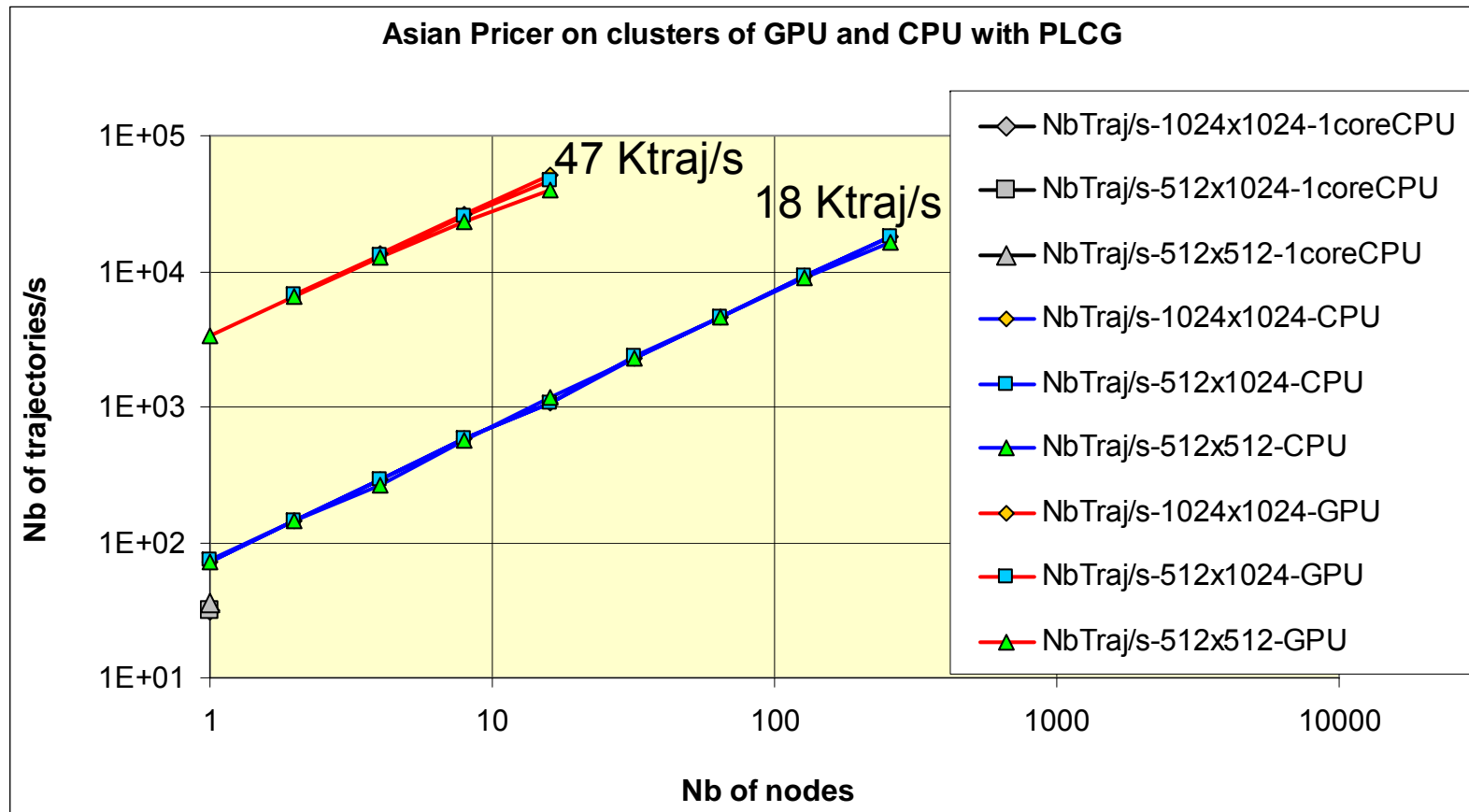
Pricing execution time with CMRG



Good scaling on both systems.
GPU time is impacted by the RNG.

Computing perf (3)

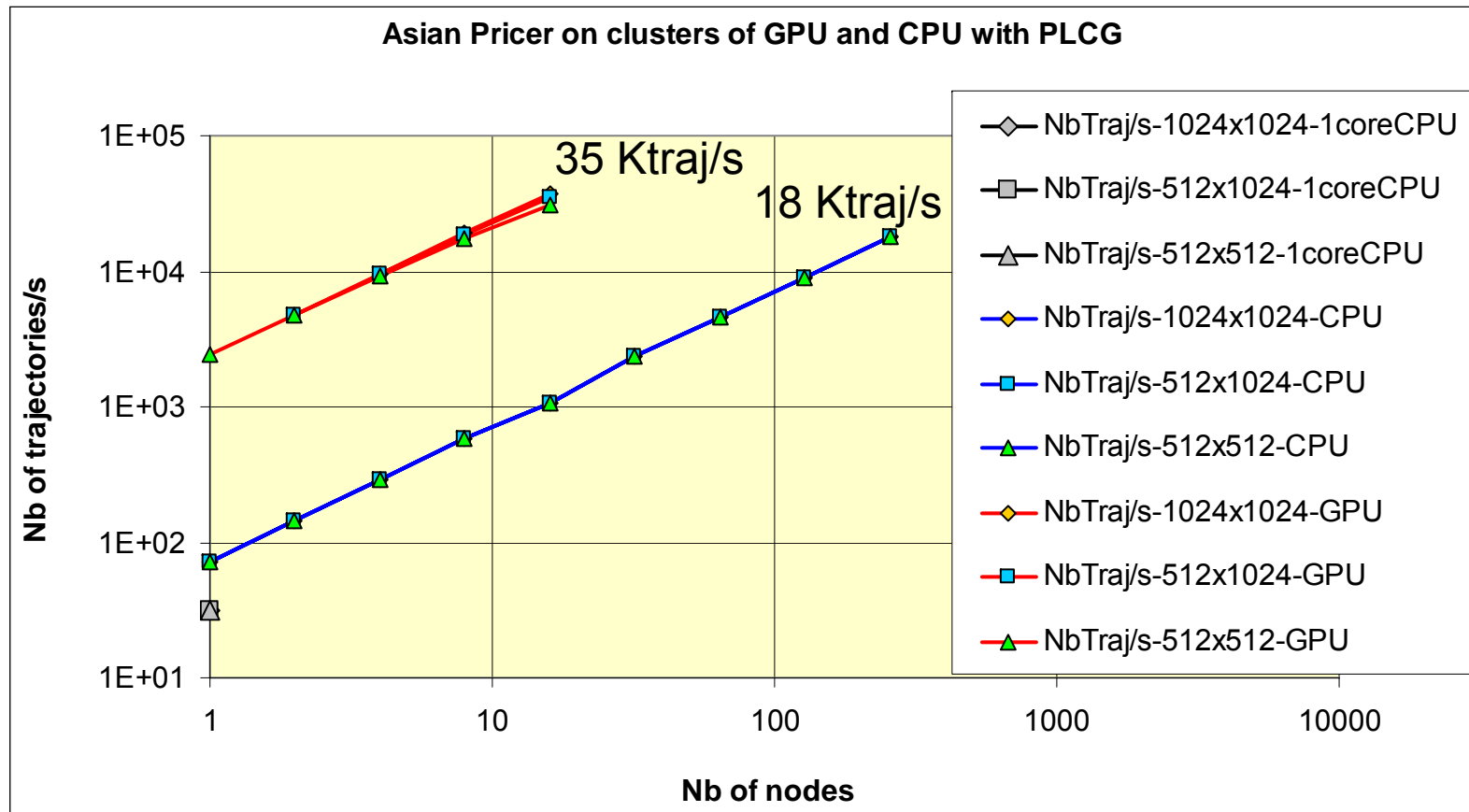
Pricing speed with PLCG



Computation speed is independent of the problem size.

Computing perf (4)

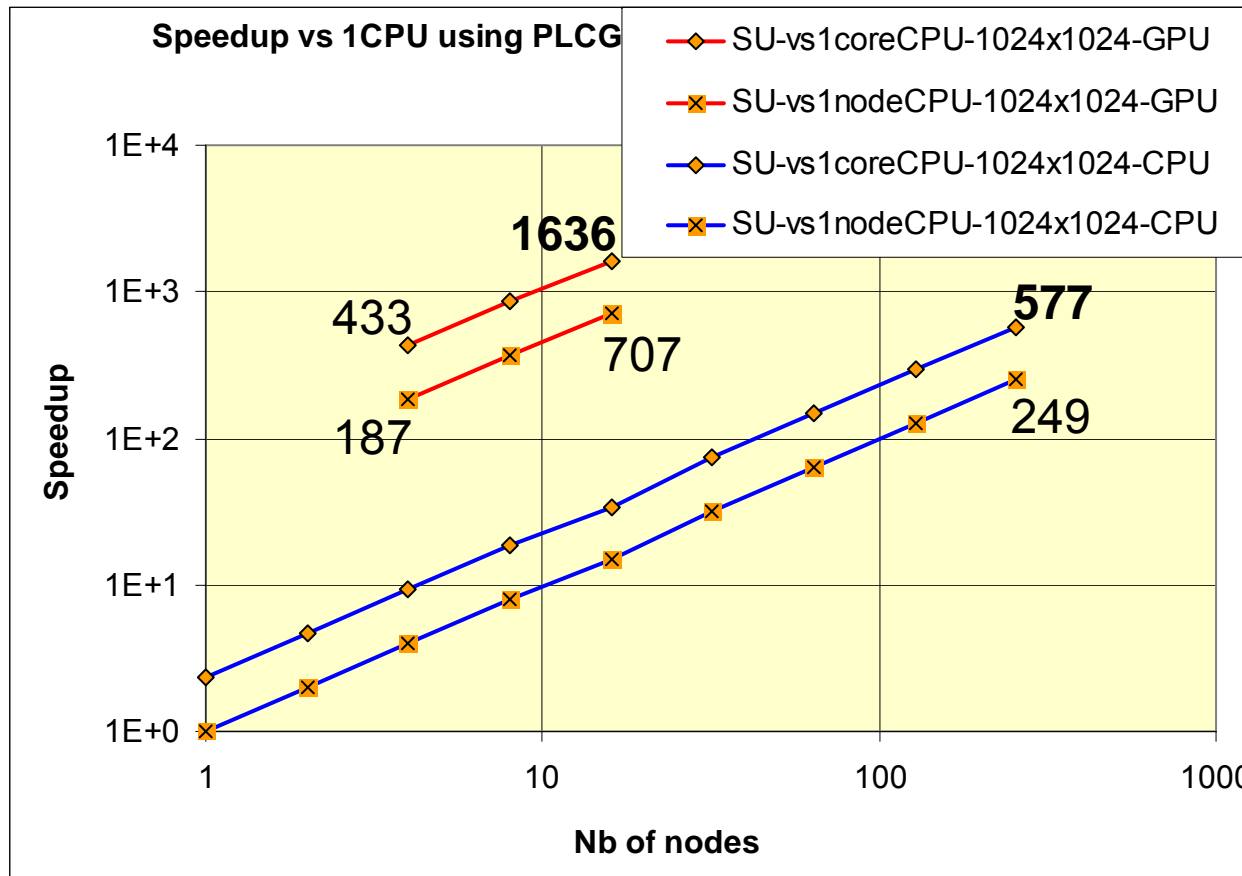
Pricing speed with CMRG



Computation speed is independent of the problem size.
 GPU computation speed is impacted by the RNG choice.

Computing perf (5)

Pricing speedup with PLCG

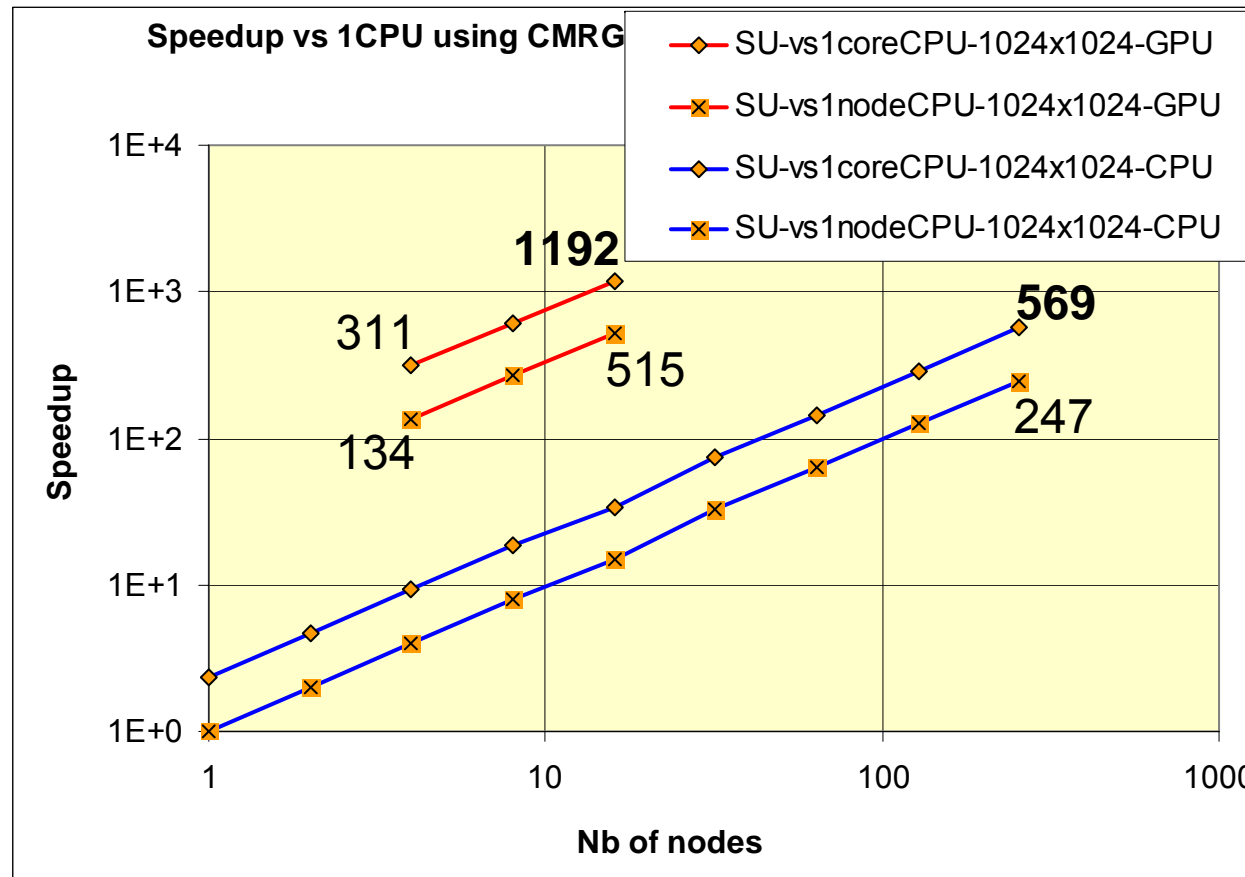


x 2.83

With 16 GPU nodes: speedup vs 1-core-CPU reaches 1636,
 speedup vs 1-node-CPU reaches 707,
 speedup vs 256-nodes-CPU cluster reaches 2.83.

Computing perf (6)

Pricing speedup with CMRG

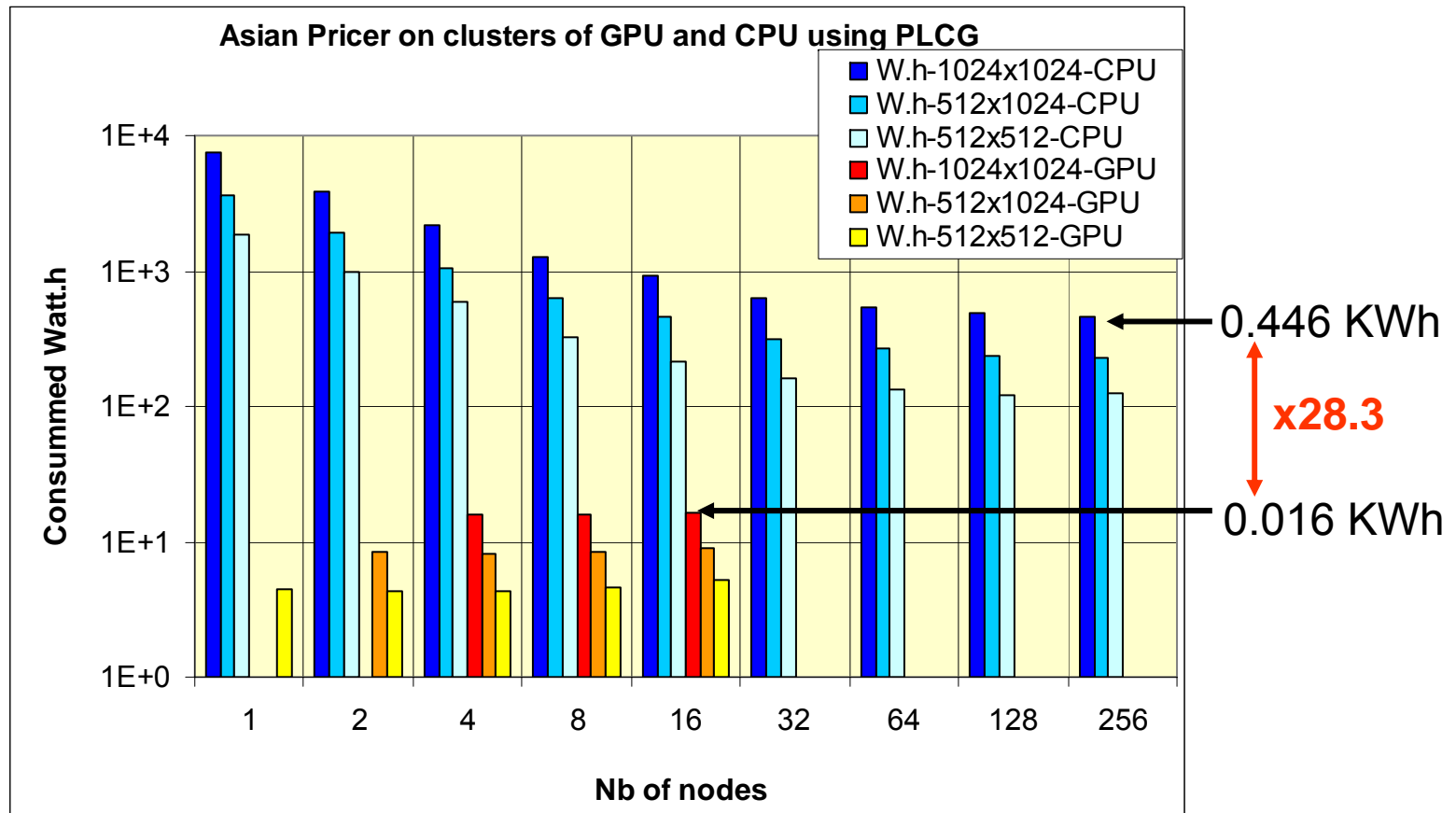


↑ x 2.09

With 16 GPU nodes: speedup vs 1-core-CPU reaches 1192,
 speedup vs 1-node-CPU reaches 515,
 speedup vs 256-nodes-CPU cluster reaches 2.09.

Energetic perf (1)

Energy consumption with PLCG

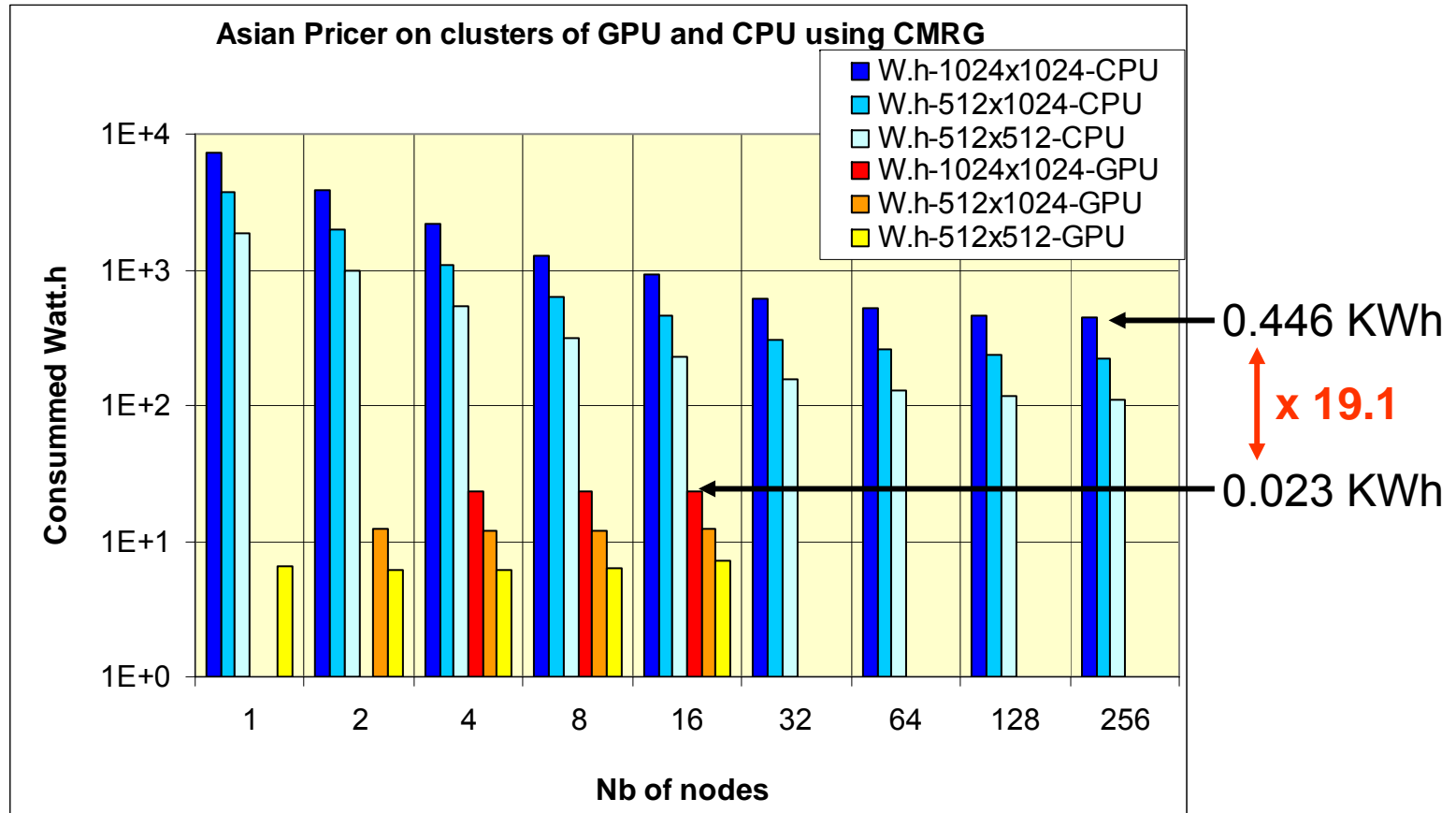


Air conditioned has not been considered.

16-nodes GPU cluster consumes 28.3 times less than a 256-nodes CPU cluster.

Energetic perf (2)

Energy consumption with CMRG

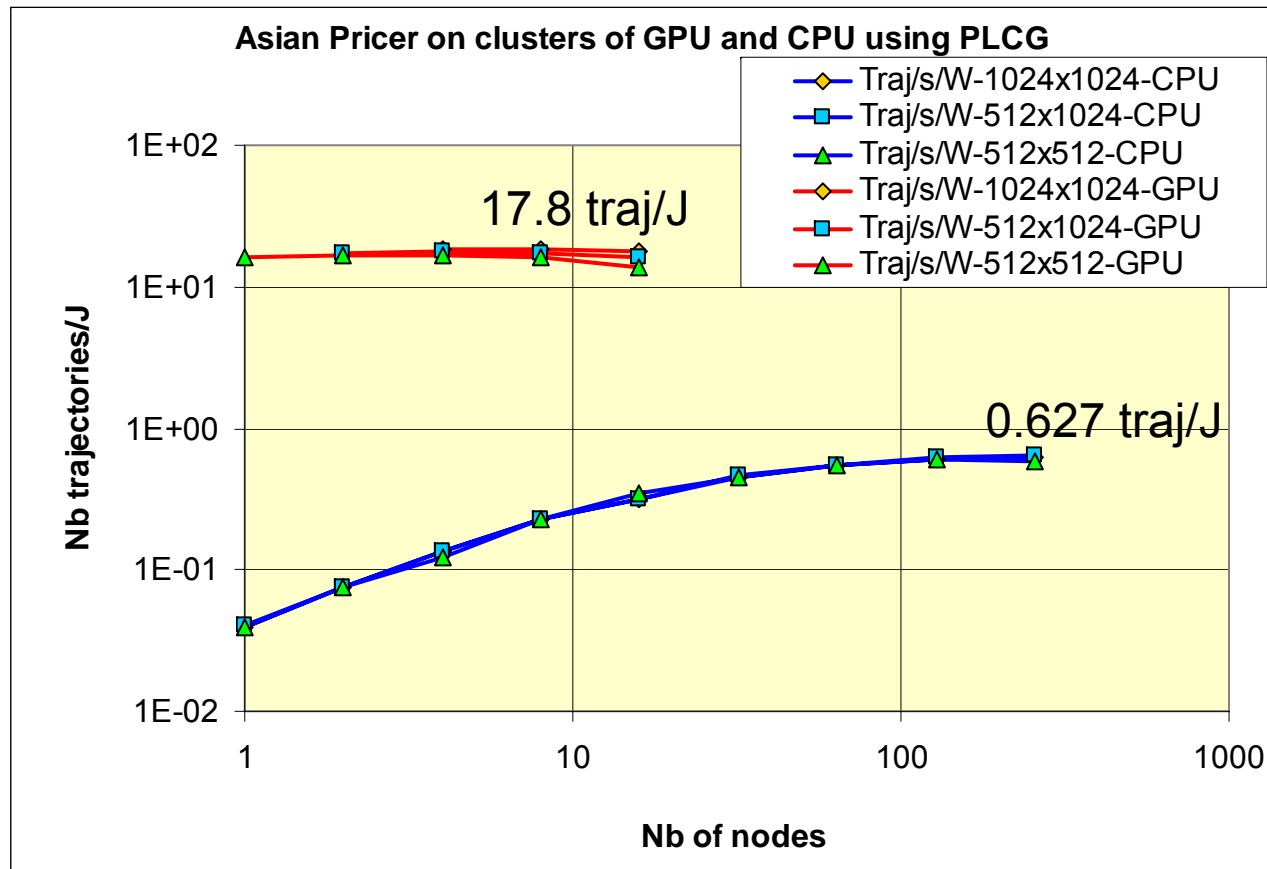


Air conditioned has not been considered.

16-nodes GPU cluster consumes 19.1 times less than a 256-nodes CPU cluster.

Energetic perf (3)

Effectiveness of computing energy with PLCG

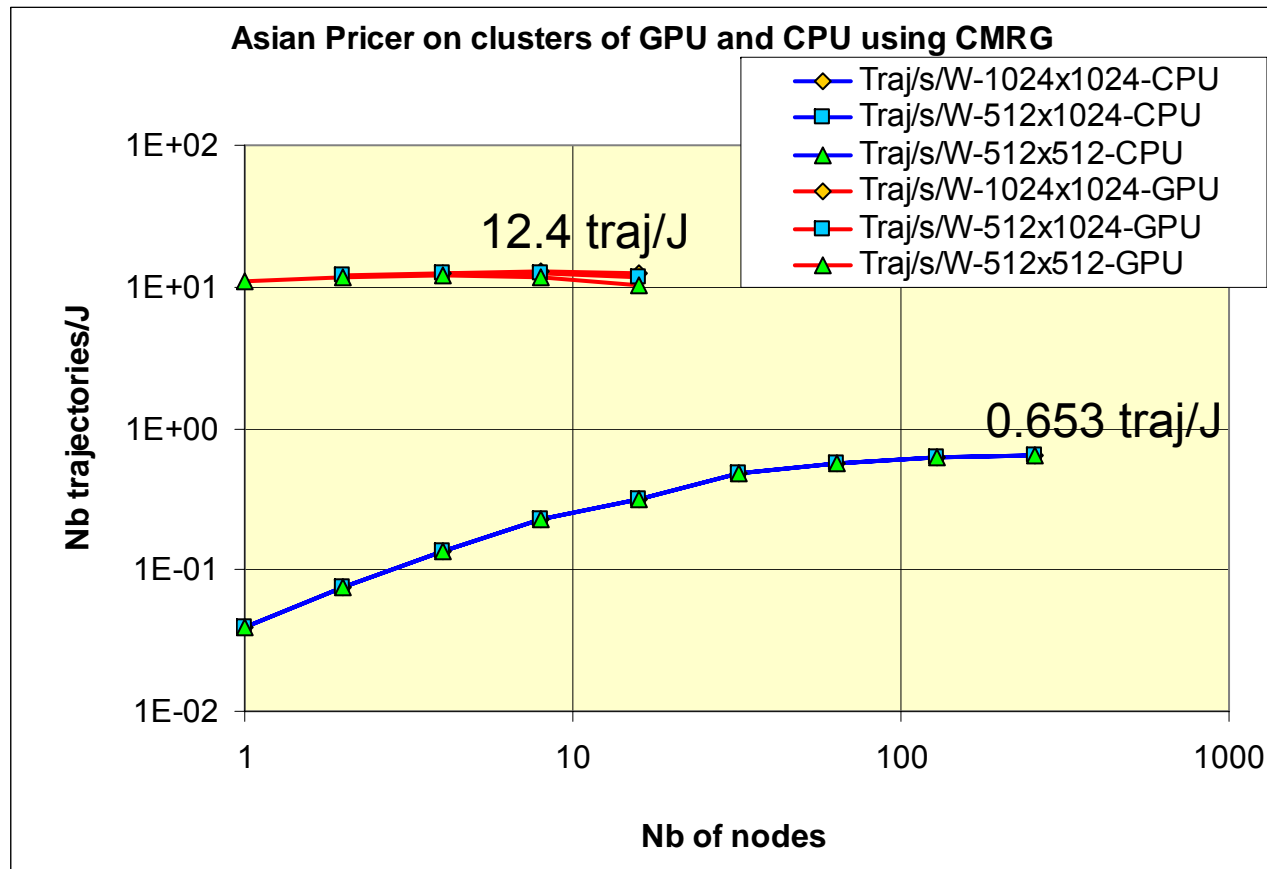


Currently, air conditioning has not been considered.

Effectiveness of computing energy is 28.3 times higher on the 16-nodes GPU cluster.

Energetic perf (4)

Effectiveness of computing energy with CMRG

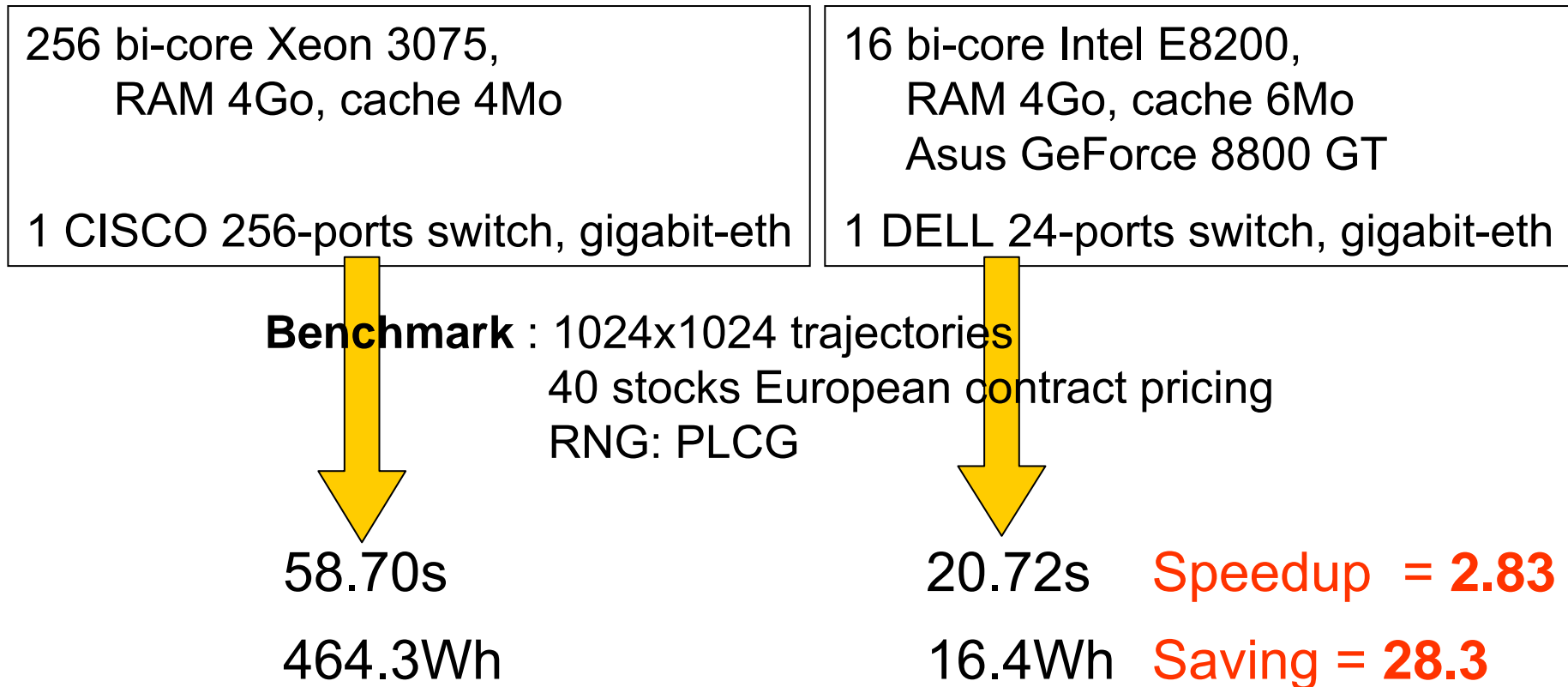


Currently, air conditioning has not been considered.

Effectiveness of computing energy is 19.0 times higher on the 16-nodes GPU cluster.

Bi-core CPU cluster vs GPU cluster

Global comparison of GPU and CPU clusters



GPU cluster is $2.83 \times 28.3 = 80.1$ times more efficient

5 – Conclusion and perspectives

Conclusion & Perspectives

Developments:

- OpenMPI + CUDA + “C+” were compatible.
- Learning CUDA + all dev \approx 3 weeks (2 people).
- Debug on GPU was hard.

Performances:

- Good scaling
- GPU cluster computes faster
- GPU cluster consumes less energy
- Too high quality RNG is harmful
- PLCG + 16-node GPU cluster
 - 1636 times faster than 1-core CPU
 - 2.8 times faster than 256-node CPU cluster
 - consumes 28.3 times less than 256-node CPU cluster

Next steps:

- Experiment others Monte-Carlo simulations
- Design algorithm with better perf & mixed perf (speedup x energy saving)

European Option Pricing on a GPU Cluster

(ANR project « GCPMF »)

Questions ?

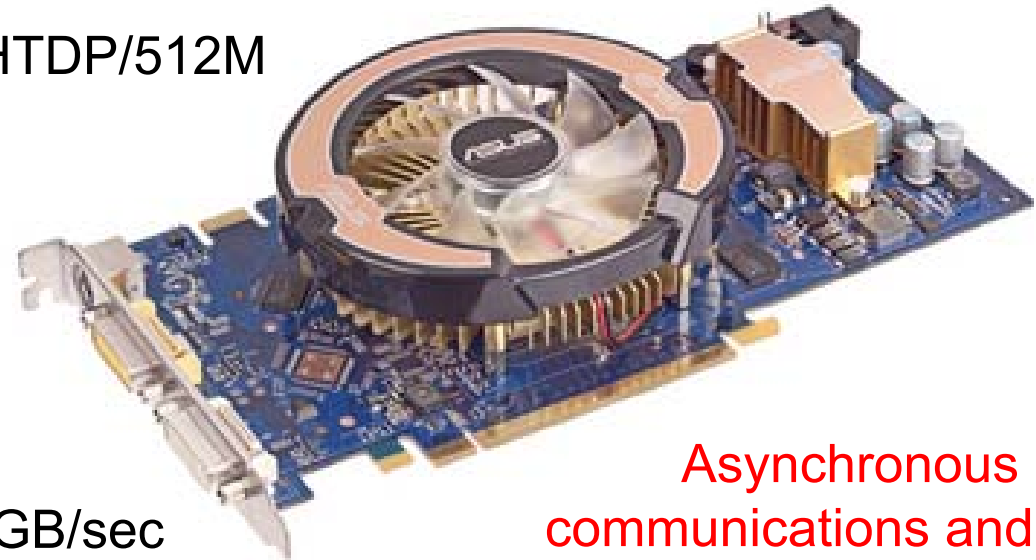


Hardware choice

GPU on each node: ASUS GeForce 8800 GT

Product model: EN8800GT/G/HTDP/512M

- Multiprocessors: 14
- Stream processors: 112
- Core clock: 600 MHz
- Memory clock: 900 MHz
- Memory amount: 512 MB
- Memory interface: 256-bit
- Memory bandwidth: 57.6 GB/sec
- Texture fill rate: 33.6 billion/sec



Asynchronous
communications and
Cuda 1.1 supported

CPU on each node: 1 processor dual-cores Intel E8200, 2.66 GHz
front side bus:1333MHz
RAM : 4Go DDR3, cache : 6Mo