

# PDE solver using asynchronous algorithms on a GPU cluster

Thomas Jost<sup>1</sup>

Sylvain Contassot-Vivier<sup>1,2</sup>

Stéphane Vialle<sup>1,3</sup>

<sup>1</sup>INRIA (ALGorille Team), Nancy, France

<sup>2</sup>University Henri Poincaré, Nancy, France

<sup>3</sup>Supélec, Metz, France

October 19, 2009

- 1 Introduction
- 2 Asynchronous iterative solver
  - PDE Solver
- 3 Sparse Linear Solver
  - Algorithm selection
  - Data structure
- 4 Results
  - Sparse linear solver
  - PDE solver
- 5 Conclusion

# Introduction

## Context

- Study of asynchronous iterative algorithms:
  - Suitability to hierarchical computing systems  
⇒ Solving PDEs on a cluster of GPUs
- Test case:
  - Application to an advection-diffusion-reaction problem
  - 3D transport of chemical species in shallow water

# Principle of the asynchronous PDE solver

## PDE solver scheme

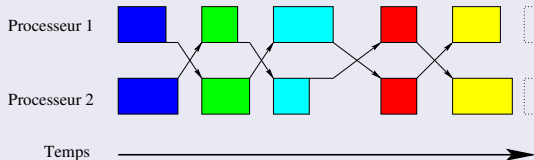
- Finite difference technique to solve the PDE
- Euler approximation of the derivatives
- Domain decomposition to extract parallelism

## Asynchronous scheme

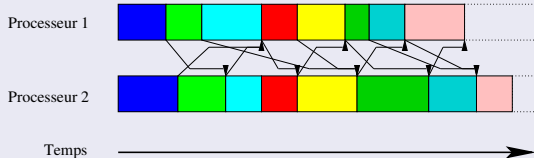
- The nodes do not wait for the last updates of their data dependencies
- ⇒ Provides an implicit overlapping of communications by computations

# Synchronous and asynchronous schemes

## Synchronous scheme



## Asynchronous scheme



More iterations but **no idle time** between them!!

# Motivation

## Asynchronism is interesting when

- the communications/computations ratio is large
- the communication costs are large

## In a cluster of GPUs

- Computations on GPUs are faster than on CPUs
  - There are two levels of communications:  
GPU ↔ CPU and CPU ↔ CPU
- ⇒ Similar context of *slow* heterogeneous communications according to the computations

**Asynchronism should provide interesting performances!**

# Multisplitting-Newton

## Algorithmic scheme

- We consider the fixed point problem  $x = T(x)$ ,  $x \in \mathbb{R}$  where  $T(x) = x - F'(x)^{-1}F(x)$
- We obtain something of the form  $F' \times DX = -F$  with  $F'$  sparse in most of the cases
- $F$  is distributed over the processors
- Each processor computes a different part of  $DX$  using the Newton algorithm within its sub-domain
- $F$  is updated by using the entire vector  $X$
- $X$  is itself updated via messages between the processors

# Proposed implementation

## Two levels of parallelism

- PDE solver at the CPUs cluster level
  - Use of the **CRAC** library (AND team / LIFC)
- Local linear systems solved on the GPUs
  - Use of the **CUDA** library (NVIDIA)

# Own implementation of the BiCG algorithm

## Sensitive points

- **Convergence speed**: number *and* duration of the iterations
- **Limited memory** → iterative algorithms are better
- **Parallelism**: simple, short kernels, or CUBLAS operations
- **Simplicity**: debugging tools do not yet allow for very complex implementations

## No matching solver exists

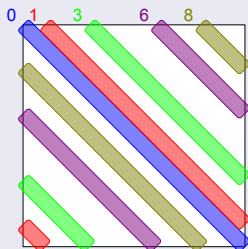
- Existing projects: MAGMA, cudaztec, GPUmatrix, CNC...
  - None of them perfectly fulfills our needs
- design of a new one!

**Biconjugate gradient is the winner!**

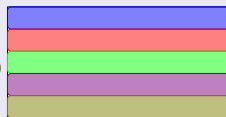
# Data structure

## Sparse structured matrices

- $n \times n$  matrix with several non-empty diagonals
- $n \times d$  array, each line is a diagonal of the matrix



AD



LA { 0 1 3 6 8 }

Each diagonal is identified by a number: diagonal  $\leftrightarrow$  line mapping

# Biconjugate gradient algorithm

## General scheme

- A few intermediate vectors are needed
- Each iteration: product by  $A$  and its transpose
  - fortunately, the data structure is adapted!

## GPU implementation

- CUBLAS for some vector operations (norm, dot product, SAXPY...)
- Two specific kernels
- Texture cache for memory reads
- `cudaMallocHost` for fast memory transfers
- ...and a lot of debugging!

# Performances comparison

## Platform

Linux OS, Intel Nehalem, GeForce GTX 285

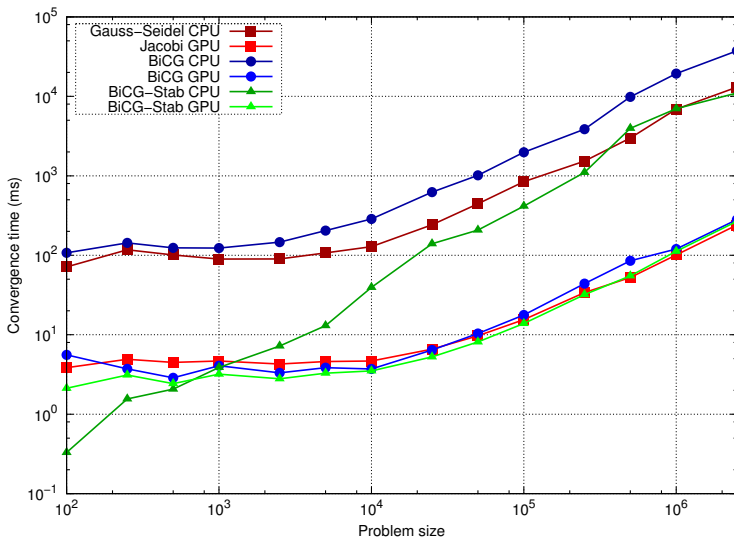
## Measurements

- Time → total execution time, including memory transfers
- Accuracy →  $\|b - Ax\|_\infty$  (maximum of the error vector in absolute value)

## Test data

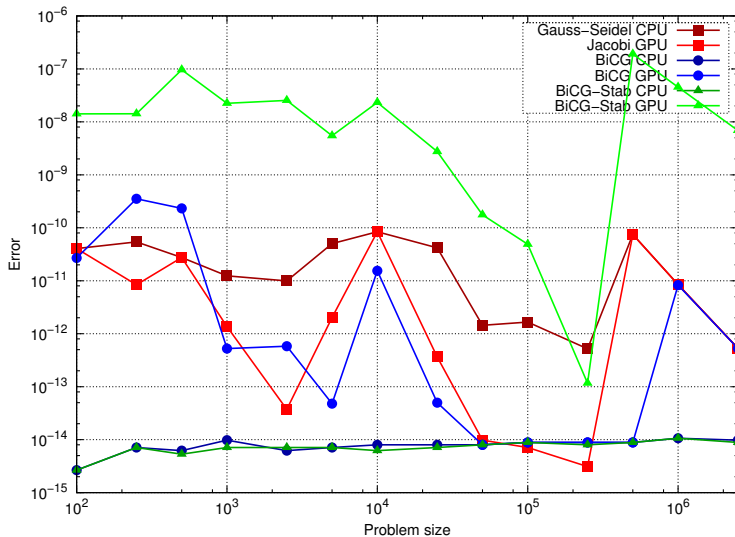
- Automatic data generation → reproducible
- Sizes between  $n = 100$  and  $n = 3,000,000$
- Real-world matrices from the Sparse Matrices Collection of the University of Florida

## Time

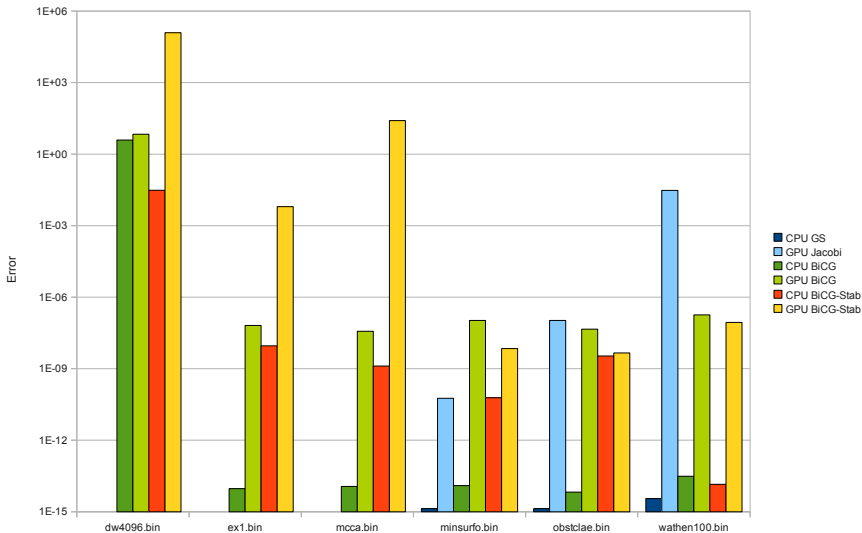




# Accuracy



# Real matrices benchmarks



# Results overview

## In summary

- GPU much faster than CPU
- ...but far from being as easy to use
- BiCG is faster better than Jacobi/Gauss-Seidel, but with a lower accuracy
- BiCG-Stab convergence is good, but accuracy is worse
- most interesting method: **BiCG on GPU**

# GPU vs CPU

## Platform

Linux OS, Intel Core2 Duo 2.66 GHz, GeForce 8800GT

## Performances

NbProcs	GPU (s)	CPU (s)	SU $\left(\frac{t_{parCPU}}{t_{parGPU}}\right)$
1	292.8	2,325.0	7.94
2	178.9	1,384.2	7.74
4	105.5	743.5	7.05
8	71.9	422.7	5.88
12	58.6	305.3	5.21

# Asynchronous vs Synchronous versions

## Platform

Linux OS, Intel Nehalem, GeForce GTX 285

## Performances

- Only preliminary results ☹️
- Between 10 and 15 times faster with the **CRAC** environment
- Smaller ratios between asynchronous **CRAC** and synchronous **MPI**

# Conclusion

## GPU sparse linear solver

- Taking advantage of the matrices structure is essential
- Fast, but not as accurate as their CPU counterparts
- The chosen algorithm is not adapted to all kinds of matrices

## PDE solver on GPU cluster

- Very interesting results between GPU and CPU versions
- Asynchronism also brings a substantial speed up!
- Should be interesting to **fully exploit** the cluster:
  - Use *all* the Nehalem cores as well as the available GPUs
  - Need for adapting the code to that architectural heterogeneity!