Parallel Multigrid Solver for Finite Elements using B-Spline

Jalal Lakhlili[1], Yaman Güçlü[1], Ahmed Ratnani[1]
(Tran-Minh Tran[2])

ECCM-ECFD 2018
M.S. Mathematical aspects of isogeometric analysis
Glasgow - June 14, 2018

[1]Max-Planck-Institut für Plasmaphysik Garching, Germany.
[2]Suiss Plasma Center - École Polytechnique Fédérale de Lausanne, Switzerland.

Outline

I Multigrid (MG) using B-Splines

II 2D parallelization of the MG solver

Outline

I Multigrid (MG) using B-Splines

II 2D parallelization of the MG solver

Overview

Multigrid methods are efficient solvers/preconditioners for linear or nonlinear systems of equations derived from discretizations of elliptic PDEs, i.e. find $u \in U$ such that:

$$a(u, v) = f(v) \quad \forall v \in V,$$

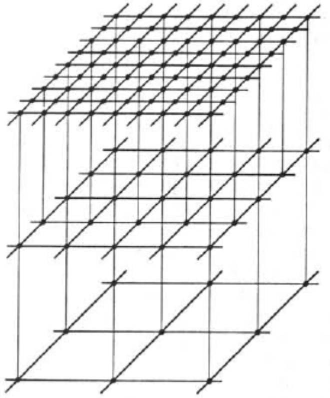
or equivalently:

$$Ax = b.$$

Theoretically, in optimal case, their numerical complexity is $\mathcal{O}(n)$.

Main idea

Geometric multigrid is based on a fine-to-coarse grid/FE-space hierarchy of the problem.



Ingredients

- Mesh hierarchy: fine to coarse.
  - ➠ We discretise the PDE in the fine mesh.
- Restriction operator: mapping from fine to coarse.
- Prolongation operator: mapping from coarse to fine.
- Smoother on each level.
- Coarse grid solver.

Inter-grid operations

▷ C. de Boor (2001)

Prolongation and Restriction are based on the knot insertion algorithm.

- We consider a nested sequence of knot vectors $T_0 \subset T_1 \subset \ldots \subset T_n$.
- We compute the knot insertion matrix P_i^{i+1} from T_i to T_{i+1}.
- We obtain the insertion matrix from T_0 to T_n by:

$$\mathcal{P} := P_0^n = P_0^1 \; P_1^2 \; \ldots \; P_{n-1}^n,$$

which correspands to the **Prolongation** operator.

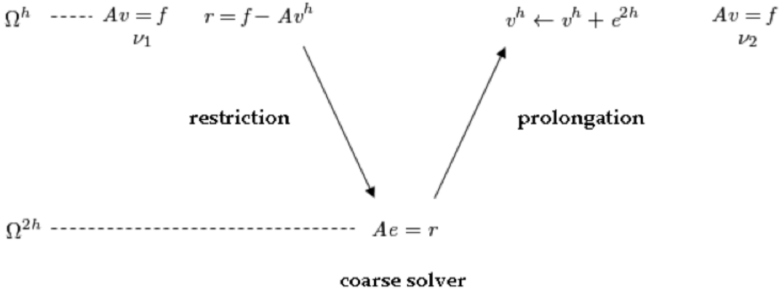
➠ In 2D, if $\mathcal{P}_1$, $\mathcal{P}_2$ denote the transfer operators for each direction, then the Prolongation operator is defined as the Kronecker product:

$$\mathcal{P} := \mathcal{P}_1 \otimes \mathcal{P}_2.$$

➠ The **Restriction** operator is then defined as: $\mathcal{R} := \mathcal{P}^T$.

V-cycle algorithm

1. Iterate: $\mathtt{x}_f := pcg^{\nu_1}(A_f, x_0, b_f)$
2. Get residual: $\mathtt{r}_f = b_f - A_f x_f$
3. Coarsen: $\mathtt{r}_c = R r_f$
4. Solve: $\mathtt{A}_c e_c = r_c$
5. Correct: $\mathtt{x}_f := x_f + P e_c$
6. Iterate: $\mathtt{x}_f := pcg^{\nu_2}(A_f, x_f, b_f)$



*$A_c = RA_f P$ is the coarse-grid version of A.

Post-smoothing

▷ Garoni, Manni, Pelosi, Serra-Capizzano and Speleers (2014)

The Stiffness matrix for a B-Splines discretization presents a pathology in high frequencies.

➠ Change the post-smoother in the Multigrid algorithm using **PCG** with the following preconditioner:

$$T[\mathfrak{m}_{p-1}] \otimes T[\mathfrak{m}_{p-1}].$$

$T[\mathfrak{m}_{p-1}]$ is the Toeplitz matrix associated to the symbol $\mathfrak{m}_{p-1}$ defined as

$$\mathfrak{m}_p(x, \theta) := \mathfrak{m}_p(\theta) = \phi_{2p+1}(p+1) + 2\sum_{k=1}^{p} \phi_{2p+1}(p+1-k)\cos(k\theta),$$

where ϕ_{2p+1} is the **cardinal spline** of degree $2p+1$.

Outline

I Multigrid (MG) using B-Splines

II 2D parallelization of the MG solver

Outline

Parallelization 2D

Goal: parallelization of the Multigrid solver (with GLT post-smoother) using distributed memory with MPI programming, in order to be run on clusters of multi-core processors.

Needs:

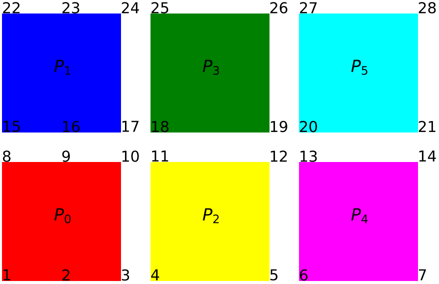
- Local data strucure.
- Partitioning of MPI processes.
- Communications patterns.

2D MPI Cartesian Topology

- Domain decomposition object

  Cart(npts, pads, periods, reorder, comm=MPI.COMM_WORLD)

  - Help to improve the *scalability of ghost cell exchanges*.
  - Acess to sub-communicators.



Vector Space Stencil

- Vector space for n-dimensional stencil format. Two different initializations are possible:

  - serial: **StencilVectorSpace(npts, pads, dtype=float)**

  - parallel: **StencilVectorSpace(cart, dtype=float)**

- Layer to which the local data structure belongs:
  - ➠ Vector Stencil.
  - ➠ Matrix Stencil.

Vector Stencil

Partition of *vectors* for the solution and RHS.

- The $N_i + p_i$ Splines on each dimension are partitioned by defining its starting and ending index $(s_i, e_i,\ i = 1, 2)$.
- The size of the local 2D array holding the Splines coefficients should be *augmented* with *ghost cells* of widths p_i at both ends of the vectors.
  - Creation: **v = StencilVector(space)**.
  - Local acess: $v[s_1 : e_1, s_2 : e_2]$

Matrix Stencil

Partition of the *matrix* for linear operators.

- A quadratic bilinear operator.
  - Creation: **M = StencilMatrix(space1, space2)**.
  - Local acess: $M[s_1 : e_1, s_2 : e_2, -p_1 : p_1, -p_2 : p_2]$
- The *local* matrix-vector product is thus defined simply by

$$v_{i_1,i_2} = \sum_{k_1=-p_1}^{p_1} \sum_{k_2=-p_2}^{p_2} M_{i_1,i_2,k_1,k_2} \; u_{i_1+k_1,i_2+k_2}$$

where: $s_1 \leq i_1 \leq e_1$ and $s_2 \leq i_2 \leq e_2$.

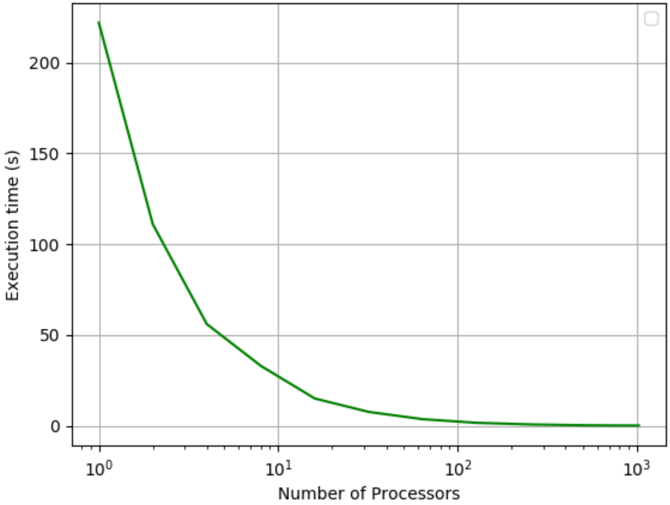
Solve a PDE equation:

$$\begin{cases} -\triangle u + u & = & f & \text{in } \Omega \\ \nabla u \cdot \mathbf{n} & = & 0 & \text{on } \partial\Omega \end{cases}$$

- Computational domain: $\Omega = [0, 1]^2$.
- Discretization: Finite Element using B-Splines.
- Fine grid: $128x128$, coarse grid: $64x64$.
- Smoother: PCG with Weighted Jacobi $(\omega = \frac{2}{3})$.
- Post-smoother: PCG with GLT $(iterations = p + 1)$.

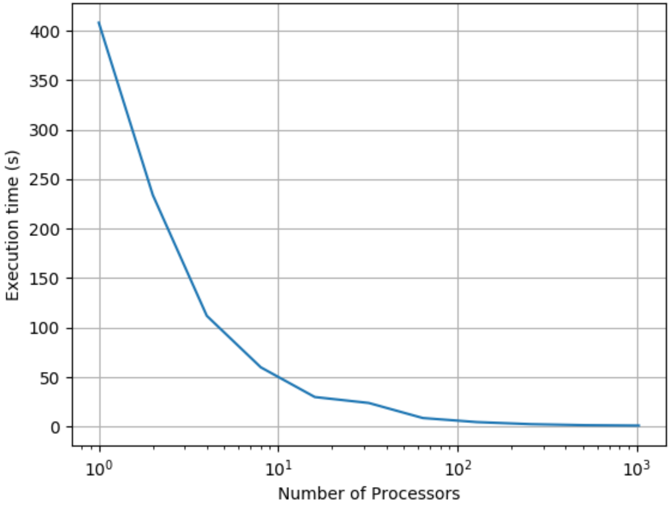
Scaling Result

- Assembly matrix $-\triangle + Id$ (Spline degree = 3, grid = 128x128):



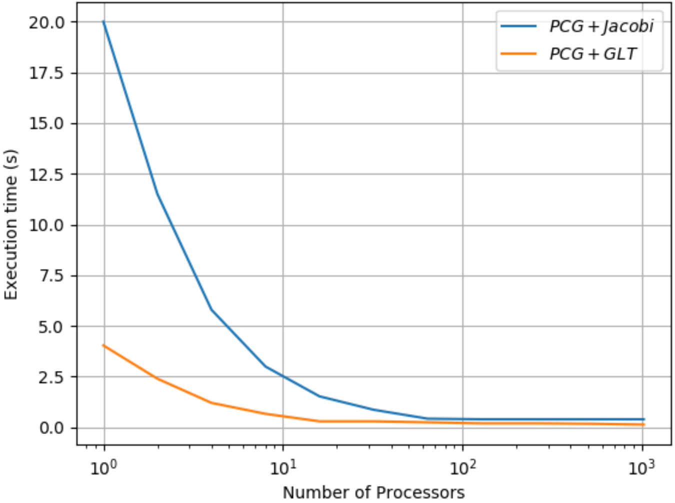
Scaling Result

- PCG + weighted Jacobi (iterations = 6, tolernce = 10^{-6}):



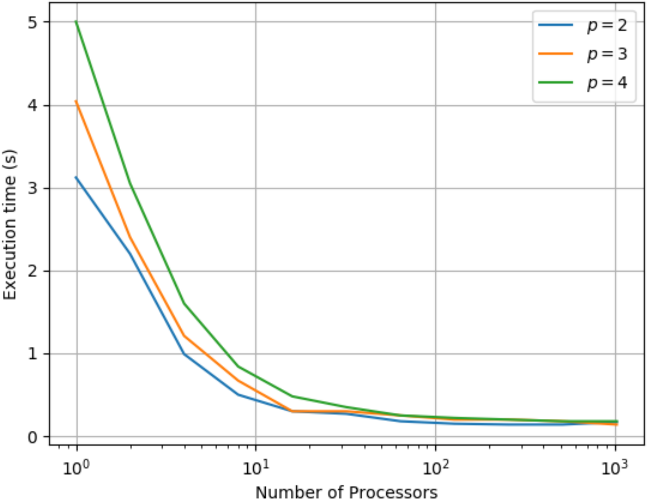
Scaling Result

- Smoother PCG + weighted Jacobi **vs.** Post-smoother PCG + GLT:



Scaling Result

- Post-smoother with different splines dregrees:



Conclusion and remarks

- List of implemented and tested components:
  - Inter-grid prolongation and restriction.
  - Parallel Jacobi relaxation.
  - Preconditioned PCG required for the GLT *post-smoother*.
  - Prallelization patterns within isogeometric context
    - Ghost cell exchange.
    - Local matrix-vector product.
    - Computation of residues.

- Future works:
  - Perform more tests with various levels and discretizations.
  - Develop other parallel smoothers (Red-Black Jacobi, GMRES, ...).
  - Convert loops from Python to Fortran and accelerate the code using *Pyccel* (tool like Pythran in C++).

Benchmarks

	Pure Python	Cython	PyPy	Pythran	Pyccel
Timing	0.12095	0.022737	0.017099	0.0012869	**0.00096607**
Speedup					

Table: Benchmark result on the Black-Scholes program.

	Pure Python	Cython	PyPy	Pythran	Pyccel
Timing	0.277	0.001811	0.059782	0.0024909	**0.0007259**
Speedup					

Table: Benchmark result on the Grow-cut program.

	Pure Python	Cython	PyPy	Pythran	Pyccel
Timing	0.040508	0.0136299	0.147573	0.0294818	**0.0048789**
Speedup					

Table: Benchmark result on the Rosen der program.