(Beginning of a) Framework for Galerkin methods on hybrid architectures

Christophe Prud'homme

Université Joseph Fourier

09/11/2010

Collaborators

V. Chabannes (Phd UJF/LJK)

N. Debit (UCB/ICJ)

M. Ismail (UJF/LSP)

G. Pena (U. Coimbra)

C. Prud'homme (UJF/LJK)

Sponsors

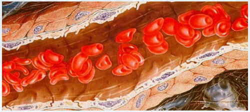
Cluster RA ISLE/CHPID, U. Coimbra, CNRS-INSMI

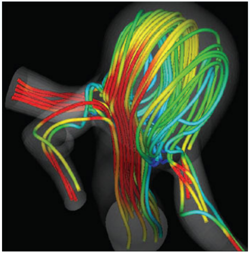
Sommaire

1. Introduction
2. Framework for Generalised Galerkin Methods
   - FEEL++: example of a DSEL
   - High order methods (h/p)
   - High order mesh
   - Function Spaces
   - Operators and Forms
   - A Language for PDEs
3. Extensions
   - Seamless interpolation tool
   - ALE framework
   - Exploit hybrid architectures
     - Exploit hybrid architectures : Multicore
     - Exploit hybrid architectures : GPU
4. Applications
   - Fluid-structure interaction
   - The fat boundary method
4. Conclusions and Perspectives

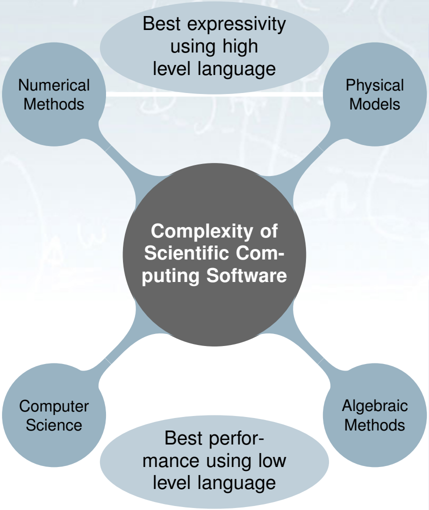
Motivations

- Rheology of blood flow
  - Interaction plasma/arterial wall
  - Simulation of a large number of blood cells
  - Spatio-temporal organization of entities
  - Mass transfer
- Spectral methods
  - Space
  - Time
  - Geometry
- High Performance Computing
  - Strategies for parallel computing (domain decomposition...)
  - Parallel/Hybrid Architectures : CPU(multicore) / (multi)GPU
- Integration to the library and language FEEL++





Generative Programming and DS(E)L

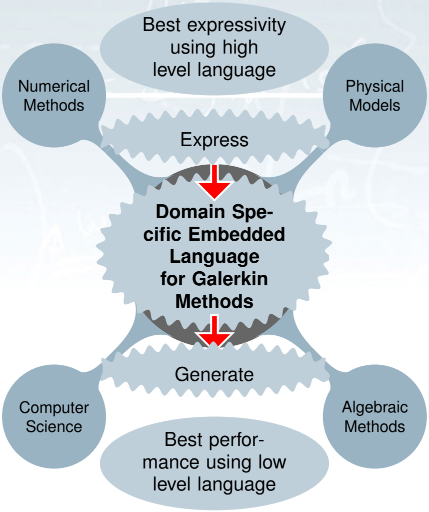


Complexity Types

- Algebraic
- Numerical
- Models
- Computer science

- Numerical and model complexity are better treated by a high level language
- Algebraic and computer science complexity perform often better with low level languages

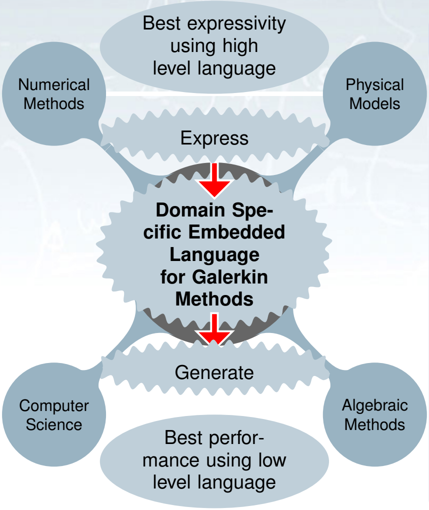
Generative Programming and DS(E)L



Generative paradigm

- distribute/partition complexity
- developer: The computer science and algebraic complexity
- user(s): The numerical and model complexity

Generative Programming and DS(E)L



Definitions

- A Domain Specific Language (DSL) is a programming or specification language dedicated to a particular domain, problem and/or a solution technique
- A Domain Specific Embedded Language (DSEL) is a DSL integrated into another programming language (e.g. C++)

FEEL++: example of a DSEL – Navier-Stokes

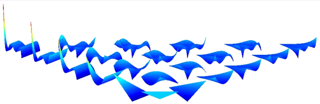
- FEEL++: C++ library for partial differential solves developed at U. Grenoble(LJK)
- The variational formulation of Navier-Stokes equation :

$$\int_\Omega \alpha \mathbf{u} \cdot \mathbf{v} + 2\nu D(\mathbf{u}) : D(\mathbf{v}) + \beta \nabla \mathbf{u} \cdot \mathbf{v} - \nabla \cdot \mathbf{v}\, p + \nabla \cdot \mathbf{u}\, q$$

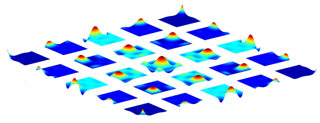
```
auto def =  0.5*(grad(v)  + trans(grad(v)));
auto deft = 0.5*(gradt(u) + trans(gradt(u)));
form2(_test=Xh, _trial=Xh, _matrix=M) =
  integrate( elements(Xh->mesh()),
             alpha*trans(idt(u))*id(v)
             + 2.0*nu*trace(trans(deft)*def)
             + trans(gradt(u)*idv(beta))*id(v)
             - div(v)*idt(p) + divt(u)*id(q) );
```

High order methods (h/p)

Polynomial basis



(a) Dubiner polynomials of degree ≤ 5 on triangles



(b) Legendre polynomials of degree ≤ 4 on quadrangles

Express the polynomials of $\mathbb{P}_k(K)$, or $\mathbb{Q}_k(K), K \subset \mathbb{R}^d, d = 1, 2, 3, \ldots$ in the Dubiner/Legendre basis $\{\phi_i\}_{i=1,\dim \mathbb{P}_k(K)}$, (see Kirby, Sherwin/Karniadakis)

$$p = \sum_i (p, \phi_i)_K \phi_i, \quad p \in \mathbb{P}_k(K)$$

- Hierarchical L_2 orthonormal basis
  - Trivial to extract a basis of $\mathbb{P}_l(K) \subset \mathbb{P}_k(K), l \leq k$
  - Ease construction of polynomial families
- Algebraic representation
- Provide policies for polynomials

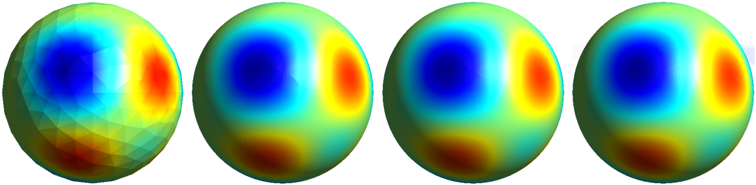
High order mesh

Motivations

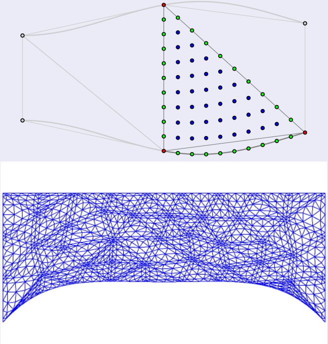
High order accuracy on complex geometries requires high order meshes (geometric transformation of order ≥ 2)

Difficulties

- Mesh generation (gmsh)
- Robust interpolation (non linear solves ...)
- Very expensive (Quadratures, Interpolation,...)
- Visualisation



High Order Mesh



- Convexes and associated geometric transformation ($\mathbb{P}_N, \mathbb{Q}_N, N = 1, 2, 3, 4, 5$)
- Support for high order ALE maps [Pena et al., 2010]
- Geometric entitites are stored using Boost.MultiIndex
- Element-wise partitioning using Scotch/Metis, sorting over process id key

Example

```
elements(mesh [, processid]);
markedfaces(mesh, marker [, processid]);
```

Function Spaces

- Product of N-function spaces (a mix of scalar,vectorial, matricial and different basis types)
- Get each function space and associated “component” spaces
- Associated elements/functions of N products and associated components, can use different backend (gmm,petsc/slepc,trilinos)

Example

```
typedef FunctionSpace<Mesh,bases<Lagrange<2,Vectorial>,
                       Lagrange<1,Scalar> > > space_t;
auto Xh = space_t::New( mesh );
auto Uh = Xh->functionSpace<0>();
auto x = Xh->element();
auto p = x->element<1>(); // view
```

Operators and Forms

- Linear Operators/Bilinear Forms represented by full, blockwise matrices/vectors
  - Full matrix $\begin{pmatrix} A & B^T \\ B & C \end{pmatrix}$, Matrix Blocks A, B^T, B, C
- Don't throw away the functional information for the algebraic representation

Example

```
auto Xh = Xh_type::New(mesh); Vh = Vh_type::New(mesh);
auto u = Xh->element(); auto v = Vh->element();
// operator T : Xh -> Vh'
auto T = LO( Xh, Vh [, backend] );
T = integrate(elements(mesh), id(u)*idt(v) );
// linear functional f : Vh -> R
auto f = LF( Vh [, backend] );
T.apply( u, f ); f.apply( v );
```

A Language for PDEs

Enablers and Features

- Meta/Functional - programming (Boost.MPL...): high order functions, recursion, ...
- Crossing Compile-time to Run-time (Boost.fusion...)
- Lazy evaluations (multiple evaluation engines) use `Expr<...>` (expression) and `Context<...>` (evaluation) (e.g. Boost.Proto)

Features: Use the *C++* compiler/language optimizations

- Optimize away redundant calculations (*C++*)
- Optimize away expressions known at compile time(*C++*)

Example

// $a : X_1 \times X_2 \to \mathbb{R} \quad a = \int_\Omega \nabla u \cdot \nabla v$

form (_text=X_1,_trial=X_2,_matrix=M)=

```
  integrate( elements(mesh), gradt(u)*trans(grad(v)));
```

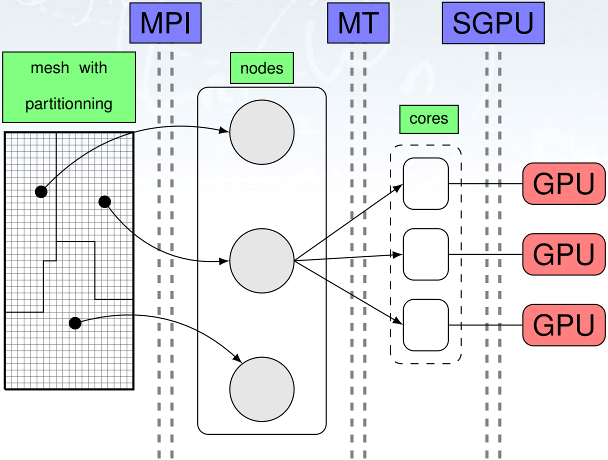
A Language for PDEs

Example: a Linear-Elasticity code

```
FunctionSpace<mesh_type,bases<Lagrange<1>>> space_type;
auto Xh = space_type::New(mesh);
auto u = Xh->element(), v = Xh->element();
// strain tensor  .5*(∇u + ∇u^T)
auto deft = 0.5*( gradt(u)+trans(gradt(u)) );
auto def = 0.5*( grad(v)+trans(grad(v)) );
auto D = backend->newMatrix( Xh, Xh );
form( _test=Xh, _trial=Xh, _matrix=D ) =
 integrate( elements(mesh),
     lambda*divt(u)*div(v)  +
     2*mu*trace(trans(deft)*def) ) +
 on( markedfaces(mesh,"clamped"), u, F, 0*one() );
// solve
backend->solve( _matrix=D, _solution=u, _rhs=F );
// apply displacement to the mesh
movemesh( mesh, u );
```

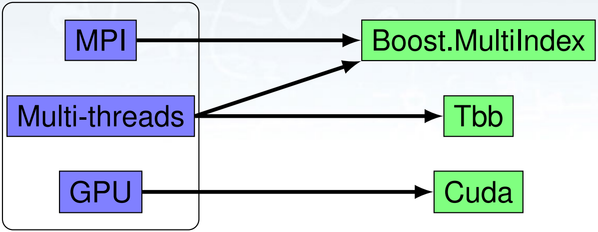
Exploit hybrid architectures

- many nodes, many cores, hybrid nodes
- MPI, Multi-Thread, Cuda/OpenCL



Exploit hybrid architectures : strategy

- Implementation realized using several libraries/frameworks :



Exploit Multicore Architectures using TBB

FEEL++ is using modern *C++*, it is difficult to obtain easily interesting performances using OpenMP. Try using Intel Threading Building Blocks (Intel TBB)

Intel TBB

- Version 3.0. Open Source (and Commercial version)
- Not yet another threading framework, but "higher level task-based parallelism that abstracts platform details and threading mechanisms for scalability and performance"
- Use C++0x (lambda functions, ...)
- Enforce clean design in library

Intel TBB

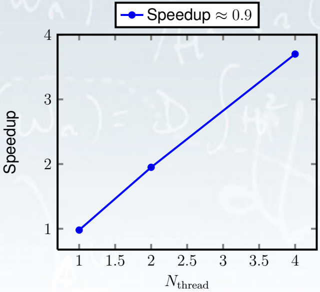
- Partition mesh elements, faces... on computational node among the cores using `blocked_range` and `simple_partitioner` (other partitioners **auto**, `affinity` not adapted)
- Task based `parallel_for` and `parallel_reduce`

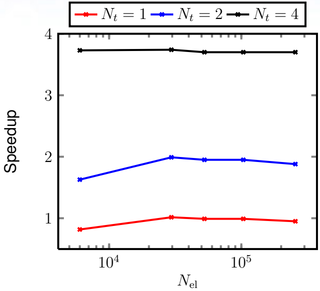
```
mesh_element_iterator it = this->beginElement();
mesh_element_iterator en = this->endElement();
// boost::multi_index structure is not using random
// access indices: create a view
typedef boost::reference_wrapper<const
                             mesh_element_type> ref_type;
std::vector<ref_type> _v;
for( auto _it = it; _it != en; ++_it )
  _v.push_back(boost::cref(*_it));
tbb::blocked_range<decltype(_v.begin())>
                             r( _v.begin(), _v.end(),
                              tbb:simple_partitioner());
Context context;
tbb::parallel_reduce( r,  context);
```

Context Evaluation

```
//  integration context
class Context
{ // ...
typedef typename std::vector<boost::reference_wrapper<
        const mesh_element_type> >::iterator elt_iterator;
void operator() ( const tbb::blocked_range<elt_iterator>& r
{ // loop over the sub-range [r.begin,r.end]
  for( auto _elt = r.begin(); _elt != r.end(); ++_elt )
   {
     geot_c.update( _elt ); // geo trans context
     expr.update( geot_c ); // expr context
     im.update( c ); // integration context
     ret += M_im( M_expr );
    }
}
void join( ContextEvaluate const& other )
{
  ret += other.ret;
} };
```

Scalability





- Compute some integrals over a 3D domain
- Compare serial version with multi-thread(m-t) version
- For a given number of elements, plot Speedup vs Number of threads (1 to 4): very good speedup
- For a given number of threads, plot Speedup vs Number of elements
- Use `tbb::task_scheduler_init` to loop over number of threads

Exploit hybrid architectures : GPU

Local assembly

- Dense matrix of size $N_{\mathrm{localDof}} \times N_{\mathrm{localDof}}$
- Construction using reference element
- Computation time depends on complexity of integrand expression and approximation choices
- Matrix entries can be independently computed

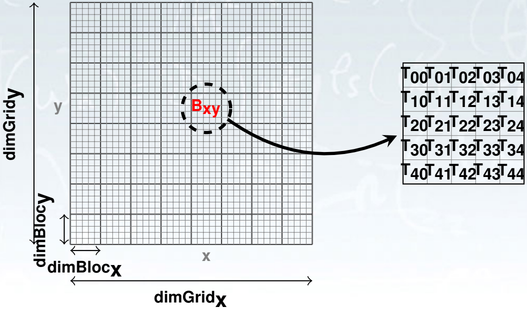
Global assembly

- Dispatch elementary matrices into global matrix

Strategy

Do part of the local assembly on GPUs and global assembly on CPU.

Exploit hybrid architectures : GPU



GPU Memories

- Global memory :
  - accessible by all GPU cores
  - accessible by the host (CPU)
- Shared memory :
  - defined on each block
  - very fast access

Process topology

The block size depends on the hardware architecture and parallel program
=> Find the optimal choice

CPU/GPU Communications

- Very expensive: to be minimized
- Limited capacity: batch communication (packets of elements)

Using modern architectures : GPU

Elementary mass matrix M and stiffness matrix S

$$M(i,j) = \int_K \Phi_i(x)\Phi_j(x)dx = \sum_{q=1}^{N_q} w_q \hat{\Phi}_i(\hat{x}_q)\hat{\Phi}_j(\hat{x}_q)|J|$$

$$\begin{aligned} S(i,j) &= \int_K \nabla\Phi_i(x)\cdot\nabla\Phi_j(x)dx \\ &= \sum_{q=1}^{\tilde{N}_q} \tilde{w}_q \left[(\nabla_{\hat{\mathbf{x}}}\varphi_K)^{-T}\ \nabla_{\hat{\mathbf{x}}}\left(\hat{\phi}_i\left(\hat{\mathbf{x}}_q\right)\right)\right]\cdot\left[(\nabla_{\hat{\mathbf{x}}}\varphi_K)^{-T}\ \nabla_{\hat{\mathbf{x}}}\left(\hat{\phi}_j\left(\hat{\mathbf{x}}_q\right)\right)\right]|J| \end{aligned}$$

Local assembly

Locally assemble $M+S$ on GPU using cuda 2.3 with parameters $d=2,3$, $N=1,...10$, N_{el}, BlockSize (default: 6), BatchSize (default: 100), measure not only computing time but also commmunications between CPU-GPU

Influence of interpolation degree I

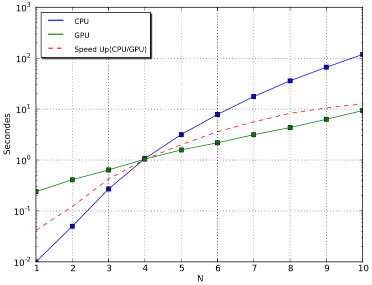
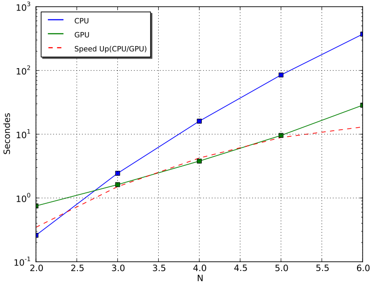


Figure: d=2, Nel=10000, BlockSize=6

N	CPU	GPU	Speed Up
1	0.01	0.24	0.0416667
2	0.05	0.41	0.121951
3	0.27	0.64	0.421875
4	1.07	1.04	1.02885
5	3.16	1.58	2
6	7.85	2.18	3.60092
7	17.68	3.15	5.6127
8	35.95	4.34	8.28341
9	66.53	6.31	10.5436
10	118.71	9.37	12.6692

Influence of interpolation degree II



N	CPU	GPU	Speed Up
2	0.26	0.75	0.346667
3	2.44	1.62	1.50617
4	16.04	3.79	4.23219
5	84.97	9.57	8.87879
6	371.06	28.52	13.0105

Figure: d=3, Nel=10000, BlockSize=6

Influence of block size I

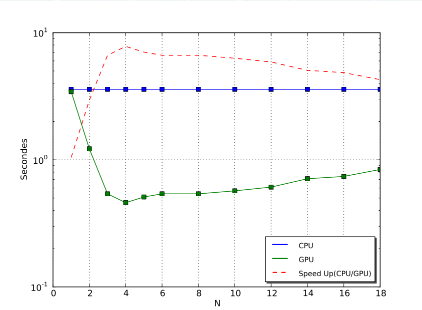


Figure: d=2, N=8, Nel=1000

Size	CPU	GPU	Speed Up
2	3.59	1.22	2.94262
3	3.59	0.54	6.64815
4	3.59	0.46	7.80435
5	3.59	0.51	7.03922
6	3.59	0.54	6.6481
8	3.59	0.54	6.6481
10	3.59	0.57	6.2982
12	3.59	0.61	5.88525
14	3.59	0.71	5.05634
16	3.59	0.74	4.85135
18	3.59	0.84	4.27381

Influence of block size II

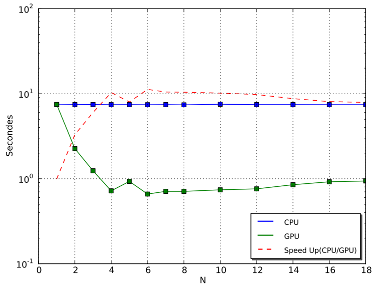


Figure: d=3, N=6, Nel=200

Size	CPU	GPU	Speed Up
2	7.45	2.26	3.2964
3	7.46	1.24	6.01613
4	7.43	0.72	10.3194
5	7.44	0.93	8.0
6	7.42	0.66	11.2424
7	7.44	0.71	10.4789
8	7.4	0.71	10.4225
10	7.52	0.74	10.1622
12	7.44	0.76	9.7894
14	7.44	0.85	8.75294
16	7.43	0.92	8.07609
18	7.43	0.94	7.90426

Influence of number of elements I

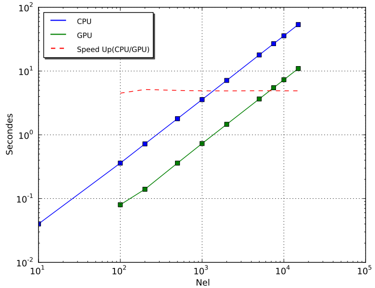


Figure: d=2, N=8

Nel	CPU	GPU	Speed Up
10	0.04	0	inf
100	0.36	0.08	4.5
200	0.72	0.14	5.14286
500	1.79	0.36	4.97222
1000	3.57	0.73	4.89041
2000	7.13	1.46	4.88356
5000	17.9	3.65	4.90411
7500	26.86	5.48	4.90146
10000	35.72	7.3	4.89315
15000	53.57	10.95	4.89224

Influence of number of elements II

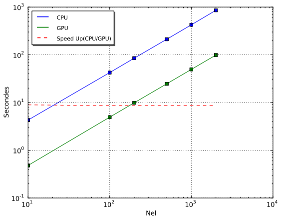


Figure: d=3, N=8

Nel	CPU	GPU	Speed Up
10	4.29	0.48	8.9375
100	42.33	4.93	8.58621
200	85	9.88	8.60324
500	211.62	24.67	8.57803
1000	424.52	49.38	8.597
2000	855.91	98.8	8.66306

Influence of size batch

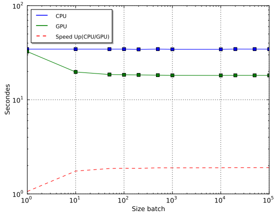


Figure: d=2, N=5, Nel=300000

Size	CPU	GPU	Speed Up
1	34.53	32.54	1.06116
10	34.53	19.75	1.74835
50	34.53	18.55	1.86146
100	34.54	18.43	1.87412
200	34.33	18.37	1.86881
500	34.56	18.26	1.89266
1000	34.41	18.19	1.8917
10000	34.39	18.15	1.89477
20000	34.58	18.19	1.90104
50000	34.55	18.18	1.90044
100000	34.52	18.15	1.90193

Conclusions and Perspectives I

Some Conclusions

- Decoupling complexities through DS(E)L is now necessary and is already available in many frameworks (FEEL++, Freefem++, Fenics, Sundance,...)
- TBB offers (so far) very good scaling on multicore at a very small implementation price (the implementation in C++ is actually fun) and (almost) fully integrated to the FEEL++ language (still many opportunities to exploit multi-threading)
- As local computations are getting more and more complex and if accuracy is needed, GPU computing will be indeed even more very interesting

Conclusions and Perspectives II

Some Perspectives

- GPU implementation not integrated yet in the embedded language, porting to OpenCL necessary
- Full framework (MPI,TBB,OPENCL) for large scale hybrid architectures
- Other numerical methods are being investigated (ANR HAMM)

FEEL++ (formerly known as Life)

http://www.feelpp.org, http://forge.imag.fr/projects/feelpp

- LJK/EDP, Université Joseph Fourier Grenoble 1
- Dept. of Mathematics, U. Coimbra
- CMCS, EPF Lausanne (Switz.)

Developers

- Grenoble: C. Prud'homme, M. Ismail, V. Chabannes, V. Doyeux, S. Veys
- Coimbra: G. Pena

References I

Pena, G., Prud'homme, C., and Quarteroni, A. (2010).
High order methods for the approximation of the incompressible navier-stokes equations in a moving domain.
Submitted.