Oil & Gas Science and Technology – Rev. IFP Energies nouvelles (2017) **72**, 12

DOI: 10.2516/ogst/2017007

OPEN ACCESS

Dossier

SimRace 2015: Numerical Methods and High Performance Computing for Industrial Fluid Flows
SimRace 2015 : Méthodes numériques et calcul haute performance pour la simulation d'écoulements complexes

ArcFVDSL, a DSEL Combined to HARTS, a Runtime System Layer to Implement Efficient Numerical Methods to Solve Diffusive Problems on New Heterogeneous Hardware Architecture

Jean-Marc Gratien*

IFP Energies nouvelles, 1-4 avenue de Bois-Préau, 92852 Rueil-Malmaison Cedex - France
e-mail: jean-marc.gratien@ifpen.fr

* Corresponding author

Abstract *— Nowadays, some frameworks like* Arcane *and* Dune *offer a number of advanced tools to deal with the complexity related to parallelism, meshes and linear solvers. However, they do not handle the high level complexity related to discretization methods and physical models. Generative programming and Domain Specific Languages (DSL) are key technologies allowing to write code with a high level expressive language and take advantage of the efficiency of generated code with low level services. DSL may be embedded in host languages like* Python *or* C++*. Such languages, named in that case Domain Specific Embedded Languages (DSEL), are applied for instance in frameworks like* Fenics *or* Feel++ *which are dedicated to the domain of Finite Element (FE) methods and Galerkin methods. ArcFVDSL is a DSEL developed on top of the Arcane framework, aiming to implement various lowest order methods (Finite-Volume (FV), Mimetic Finite Difference (MFD), Mixed Hybrid Finite Volume (MHFV), etc.) for diffusive problems on general meshes. In this paper, we present various implementations of different complex academic problems. We focus on the capability of the language to allow the description and the resolution of these problems with several lowest-order methods. We illustrate the benefits of such technology combined to runtime system tools like Heterogeneous Abstract RunTime System (HARTS) and its ability to handle seamlessly new heterogeneous architectures with multi-core processors enhanced by General Purpose computing on Graphics Processing Units (GP-GPU). We present the performance results of each implementation on different kinds of heterogeneous hardware architecture.*

Résumé — ArcFVDSL, un DSEL combiné à HARTS, un support d'exécution pour développer des méthodes numériques efficaces pour résoudre des problèmes diffusifs sur architectures hétérogènes — Les simulateurs industriels doivent intégrer à la fois des modèles physiques complets et des méthodes de discrétisation évoluées, tout en préservant de bonnes performances sur les diverses architectures matérielles. Leur mise au point nécessite donc de gérer de manière efficace (i) la complexité des modèles physiques sous-jacents, souvent exprimés sous la forme de systèmes d'Équations aux Dérivées Partielles (EDP) ; (ii) la complexité des méthodes numériques utilisées, qui continuent à évoluer avec l'émergence de nouveaux besoins ; (iii) la complexité des services numériques de bas niveau (gestion du parallélisme, de la mémoire, des interconnexions, GP-GPU) nécessaires pour tirer partie des architectures *hardware* modernes ; (iv) la complexité liée aux langages informatiques, dont l'évolution doit être maîtrisée sous peine d'obsolescence du code.

Tous ces requis doivent être remplis pour bénéficier pleinement des nouvelles architectures massivement parallèles et hiérarchiques. Idéalement, la complexité liée aux modèles physiques et aux méthodes numériques se gère mieux par des langages de haut niveau, qui permettent de cacher les détails informatiques. En revanche, l'efficacité des composantes de bas niveau demande un accès direct aux spécificités *hardware*. De nos jours, un certain nombre de *frameworks* comme *Arcane* et *Dune* proposent un nombre avancé de services pour gérer le parallélisme, les maillages ou les solveurs linéaires. Ils ne permettent pas en revanche d'appréhender la complexité des méthodes numériques. Les paradigmes de programmation générative et de langages spécifiques aux domaines (DSL) sont des technologies clé qui permettent d'écrire des codes avec un haut niveau d'expressivité et de pouvoir tirer partie de l'efficacité des codes générés avec des services bas niveau spécifiques aux architectures matérielles cibles. Nous présentons ArcFVDSL, un tel langage dédié aux méthodes de bas ordre (Volumes-Finis, Différences Finies Mimétiques, Volumes-finis Mixtes Hybrides,...) pour résoudre des problèmes diffusifs sur maillages généraux. Nous montrons comment, associé à des supports d'exécution tels que HARTS (*Heterogeneous Abstract RunTime System*), ce langage permet d'appréhender sans effort les nouvelles architectures à base de processeurs multi-cœurs, éventuellement accélérés avec des cartes GP-GPU (*General Purpose computing on Graphics Processing Units*). Nous présentons un certain nombre de cas académiques et des résultats de performances sur les nouvelles architectures.

INTRODUCTION

Industrial simulation softwares have to manage: (i) the complexity of the underlying physical models, usually expressed in terms of a Partial Differential Equation (PDE) system completed with algebraic closure laws, (ii) the complexity of the numerical methods used to solve the PDE system, and finally (iii) the complexity of the low level computer science services required to have efficient software on modern hardware. Robust and effective Finite Volume (FV) methods as well as advanced programming techniques need to be combined in order to fully benefit from massively parallel architectures (implementation of parallelism, memory handling, design of connections). Moreover, the above methodologies and technologies have become more and more sophisticated and too complex to be handled by physicists alone. Today, this complexity management becomes a key issue for the development of scientific software.

A number of existing frameworks already offer advanced tools to deal with the complexity related to parallelism. They hide hardware complexity and provide low level algorithms dealing directly with hardware specificities for performance reasons. They often offer services to manage mesh data services and linear algebra services which are key elements for efficient parallel software. However, all these frameworks often provide only partial answers to the problem as they only deal with hardware complexity and low level numerical complexity like linear algebra. The complexity related to discretization methods and physical models lacks tools to help physicists develop complex applications. New paradigms for scientific software must be developed to help them seamlessly handle the different levels of complexity so that they can focus on their specific domain.

Generative programming, component engineering and Domain Specific Languages (DSL) are key technologies to make the development of complex applications easier to physicists, hiding the complexity of numerical methods and low level computer science services. These paradigms allow to write code with a high level expressive language and take advantage of the efficiency of generated code for low level services close to hardware specificities. Their application to scientific computing has been limited so far to Finite Element (FE) methods, for which a unified mathematical framework has been existing for a long time. Such kinds of DSL have been developed for FE or Galerkin methods in projects like Freefem, Getdp, Getfem++, Sundance, Feel++ and Fenics. In these projects, they are embedded in host languages like *Python* or *C++* and are named Domain Specific Embedded Languages (DSEL).

Over the last few years, we have extended this kind of approach to lowest-order methods to solve the PDE systems resulting from geo modeling applications. Indeed, a recent consistent unified mathematical framework which allows a unified description of a large family of these methods has emerged. It enables then, as for FE methods, the design of a high level language inspired from the mathematical notation. Such languages help then physicist to implement their application writing the mathematical formulation at a high level, hiding the complexity of numerical methods and low level computer science services

guaranty of high performance. We have developed such a language, that we have embedded in the C++ language, on top of Arcane plateform [1]. We have used the Boost Proto library [2], a powerful framework providing tools to design this DSEL. In [3] we have presented ArcFVDSL, our DSEL aiming to implement various lowest-order methods (Finite-Volume, Mimetic Finite Difference, Mixed Hybrid Finite Volume, etc.) for diffusive problems on general meshes. In [4] we have presented technical details on how this language has been designed with the Boost Proto library. In this paper, we focus on the capability of the language combined to runtime system tools like Heterogeneous Abstract RunTime System (HARTS) [5], to handle seamlessly new heterogeneous architectures with multi-core processors enhanced by General Purpose computing on Graphics Processing Units (GP-GPU). We present the performance results of various implementations of different academic problems on different kinds of heterogeneous hardware architecture.

In Section 1, we briefly present the mathematical framework, our DSEL relies upon. We present the C++ concepts used to define the back end and the front end of the language and explain how to parse expressions representing bilinear and linear forms. We show how our framework enables to generate useful algorithms to solve the discrete problems. In Section 2, we present how by introducing HARTS, a runtime layer to handle new heterogeneous hardware architecture, we have improved the generated algorithms to take advantage of multi-core architectures.

In Section 4, we present some performance results on various academic test cases that we have implemented with our framework.

1 ARCFVDSL, A DSEL IN C++ TO SOLVE DIFFUSIVE PROBLEMS WITH LOWEST-ORDER METHODS

As for FE/DG (Finite Element/Discontinuous Galerkin) methods, a unified mathematical framework which allows a unified description of a large family of lowest-order methods has recently emerged [6]. The key idea is to reformulate the method at hand as a (Petrov)-Galerkin scheme based on a possibly incomplete, broken affine space. This is done by introducing a piecewise constant gradient reconstruction, which is used to recover a piecewise affine function starting from cell (and possibly face) centered unknowns.

For example, considering the following heterogeneous diffusion model problem:

$$-\nabla\cdot(\kappa\nabla u) = f \ \text{ in } \Omega, \quad u = 0 \ \text{ on } \partial\Omega \tag{1}$$

with source term $f \in L^2(\Omega)$, κ piecewise constant.

The continuous weak formulation reads: find $u \in H_0^1(\Omega)$ such that

$$a(u,v) = b(v) \ \ \forall v \in H_0^1(\Omega)$$

with

$$a(u,v) \stackrel{\text{def}}{=} \int_\Omega \kappa\nabla u\cdot\nabla v$$

$$b(v) \stackrel{\text{def}}{=} \int_\Omega fv$$

In this framework, for a given partition $\mathcal{T}_h$ of Ω, a specific lowest-order method is defined by (i) selecting a trial function space $U_h(\mathcal{T}_h)$ and a test function space $V_h(\mathcal{T}_h)$, (ii) defining for all $(u_h, v_h) \in U_h \times V_h$ a bilinear form $a_h(u_h, v_h)$ and a linear form $b_h(v_h)$. Solving the discrete problem consists then in finding $u_h \in U_h$ such that:

$$a_h(u_h, v_h) = b_h(v_h) \ \ \forall v_h \in V_h$$

The definition of a discrete function space U_h is based on four main ingredients:

- $\mathcal{T}_h$ the mesh representing Ω, $\mathcal{S}_h$ a submesh of $\mathcal{T}_h$ where $\forall S \in \mathcal{S}_h, \exists T_S \in \mathcal{T}_h, S \subset T_S$;
- $\mathbb{V}_h$ the space of vector of degrees of freedom with components indexed by the mesh entities (cells, faces or nodes);
- $\mathfrak{G}_h$ a linear gradient operator that defines for each vector $v_h \in V_h$ a constant gradient on each element of $\mathcal{S}_h$; and
- ∇_h the broken gradient operator.

Using the above ingredients, we can define for all $\mathbf{v}_h \in \mathbb{V}_h$ a piecewise affine function $v_h \in U_h \subset \mathbb{P}_d^1(\mathcal{S}_h)$ such that: $\forall S \in \mathcal{S}_h$, $S \subset T_S$, $T_S \in \mathcal{T}_h$, $\forall \mathbf{x} \in S$,

$$v_h(\mathbf{x})|_S = v_{T_S} + \mathfrak{G}_h(\mathbf{v}_h)|_S \cdot (\mathbf{x} - \mathbf{x}_{T_S}) \tag{2}$$

Usually three kinds of submesh $\mathcal{S}_h$ are considered: $\mathcal{T}_h$ the mesh itself, $\mathcal{P}_h$ the submesh with pyramidal subcells built regarding the faces of $\mathcal{T}_h$ and $\mathcal{N}_h$ the submesh with subcells built regarding the nodes of $\mathcal{T}_h$. We denote $\mathbb{T}_h$ the space of degrees of freedom with components indexed by cells and $\mathbb{F}_h$ the space of degrees of freedom with components indexed only by faces. Usually the following choices are considered: $\mathbb{V}_h = \mathbb{T}_h$ or $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$.

With this framework, the model problem (1) can be solved with various methods:

- the cell centered Galerkin (ccG) method and the G-method with cell unknowns only;
- the hybrid finite volume method with both cell and face unknowns that recover the Mimetic Finite Difference (MFD) and Mixed Hybrid Finite Volume (MHFV) family.

For example, the hybrid finite volume method recovers the SUSHI scheme [7-10]. The discrete space with hybrid unknowns is then obtained with: (i) $\mathcal{S}_h = \mathcal{P}_h$,

(ii) $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$, (iii) $\mathfrak{G}_h = \mathfrak{G}_h^{\text{hyb}}$ with $\mathfrak{G}_h^{\text{hyb}}$ such that, for all $(\mathbf{v}_h^{\mathscr{T}}, \mathbf{v}_h^{\mathscr{F}}) \in \mathbb{T}_h \times \mathbb{F}_h$, all $T \in \mathscr{T}_h$ and all $F \in \mathscr{F}_r$,

$$\mathfrak{G}_h^{\text{hyb}}(\mathbf{v}_h^{\mathscr{T}}, \mathbf{v}_h^{\mathscr{F}})|_{\mathscr{P}_{T,F}} = \mathfrak{G}_h^{\text{green}}(\mathbf{v}_h^{\mathscr{T}}, \mathbf{v}_h^{\mathscr{F}})|_T + \mathfrak{r}_h(\mathbf{v}_h^{\mathscr{T}}, \mathbf{v}_h^{\mathscr{F}})|_{\mathscr{P}_{T,F}} \mathfrak{n}_{T,F} \tag{3}$$

where the linear residual operator $\mathfrak{r}_h : \mathbb{T}_h \times \mathbb{F}_h \to \mathbb{P}_d^0(\mathscr{P}_h)$ is defined as follows: for all $T \in \mathscr{T}_h$ and all $F \in \mathscr{F}_T$,

$$\mathfrak{r}_h(\mathbf{v}_h^{\mathscr{T}}, \mathbf{v}_h^{\mathscr{F}})|_{\mathscr{P}_{T,F}} = \frac{d^{\frac{1}{2}}}{d_{T,F}} \left[v_F - v_T - \mathfrak{G}_h^{\text{green}}(\mathbf{v}_h^{\mathscr{T}}, \mathbf{v}_h^{\mathscr{F}})|_T \cdot (\mathbf{x}_F - \mathbf{x}_T) \right]$$

This method with hybrid unknowns reads:

$$\text{Find } u_h \in V_h^{\text{hyb}} \text{ s.t. } a_h^{\text{sushi}}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in V_h^{\text{hyb}}$$

with

$$a_h^{\text{sushi}}(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega \kappa \nabla_h u_h \cdot \nabla_h v_h \tag{4}$$

and ∇_h broken gradient on $\mathscr{P}_h$.

This unified framework allows the design of a high level language close to the mathematical notation. Such a language enables to express the variational discretized formulation of PDE problem with various methods, each of them defining specific bilinear and linear forms. Algorithms are then generated to solve the problems, evaluating the forms representing the discrete problem. The front end of the language is based on concepts (mesh, function space, test trial functions, differential operators) close to their mathematical counterpart. They are linked to low level structures, the back end of the language, representing meshes, scalar arrays indexed by mesh entities, algebraic objects (vectors, matrices, linear operators). For theses structures we use frameworks like Arcane [6] or ALIEN a framework to handle various linear solver packages. Linear and bilinear forms are represented by expressions built with the terminals of the language linked with unary, binary operators (+, −, ∗, /, dot(.,.)) and with free functions like grad (.), div (.), integrate (.,.). The purpose of theses expressions is (i) to express the variational discretized formulation of the problem, (ii) to generate algorithms which consist in evaluating these expressions to build global linear systems which are solved to find the solution of the problem. The generative mechanism of our framework is based on the Boost Proto framework [2] and is described in detail [4].

For our diffusion model problem (1), such a DSEL will for instance achieve to express the variational discretized formulation (9) with the programming counterpart shown in Listing 1.

Listing 1
Diffusion problem implementation

```
MeshType Th;
Real K;
auto Vh = newSUSHISpace(Th);
auto u = Vh-> trial(''U'');
auto v = Vh-> test(''V'');
BilinearForm a =
integrate(allCells(Th), dot(K*grad(u),
grad(v)));
LinearForm b =
integrate(allCells(Th), f*v);

// Bilinear and linear form evaluation
Matrix M(/*...*/);
Vector rhs(/*...*/);
Vector sol(/*...*/);
LinearEvalContext ctx(M, rhs);
fvdsl :: eval(a, ctx);
fvdsl :: eval(b, ctx);

// Linear system resolution
Alien :: solve(M, rhs, sol);
```

The bilinear form a_h^{sushi} defined by (9) has the programming counterpart given in Listing 1 and the corresponding expression tree is detailed in Figure 1.

Listing 2 is a generic assembly algorithm consisting in iterating on each entity of the mesh group and in evaluating the test and trial expression on each entity. For such evaluation, different kinds of context objects are defined. The structure EvalContext<ItemT> enables to compute the linear combination objects, some generalized stencils or local vectors indexed by DOF. These objects are returned by the evaluation of test or trial expressions. When they are associated to a binary operator tag, they lead to a bilinear contribution, a local matrix contributing to the global linear system.

Listing 2
Integration assembly algorithm

```
template<typename ItemT,
         typename TestExprT,
         typename TrialExprT,
         typename tag_op,
         typename BilinearContextT>
void integrate(Mesh const& mesh,
               GroupT<ItemT> const& group,
               TrialExprT const& trial,
               TestExprT const& test,
               BilinearContextT& ctx)
{
  static const Context :: ePhaseType phase =
  BilinearContextT :: phase_type;
```

```
    auto matrix = ctx.getMatrix();
    for(auto cell : group)
    {
      //! eval context on mesh item
      EvalContext<Item> ctx(cell);

      //! trial linear combination
      auto lu = proto :: eval(trial, ctx);

      //! test linear combination
       auto lv = proto :: eval(test, ctx));

      BilinearContribution<tag_op>
      uv(lu, lv);
      assemble<phase>(matrix,
                      measure(mesh, cell),
                      uv);
    }
  }
```

In the same way the evaluation of a linear form expression with a linear context leads to the construction of the right hand side of a global linear system.

Once the global linear system is built, it can be solved with a linear system solver provided by the linear algebra layer ALIEN.

2 HARTS, AN ABSTRACT OBJECT ORIENTED RUNTIME SYSTEM LAYER FOR HETEROGENEOUS ARCHITECTURE

In the previous section we have presented a generative framework, based on a DSEL that enables to describe numerical methods at a high level, and to generate C++ codes with back-end objects with a generative mechanism. In this section, we show how we can introduce various kind of parallelism in the generated codes with the runtime system layer HARTS presented in details [5]. This layer is aimed to handle, in a unified way, different levels of parallelism. As in most existing runtime system frameworks, it is based on: (i) an abstract architecture model that enables us to describe in a unified way most of present and future heterogeneous architectures with static and runtime information on the memory, network and computational units; (ii) an unified parallel programming model based on tasks that enables us to implement parallel algorithms for different architectures; (iii) an abstract data management model to describe the processed data, its placement in each memory level and the various ways to access it from the each computation unit. The use of HARTS as illustrated in Figure 2 has several advantages:

1. it enables to clearly separate the implementation of the numerical layer from the implementation of the runtime system layer;

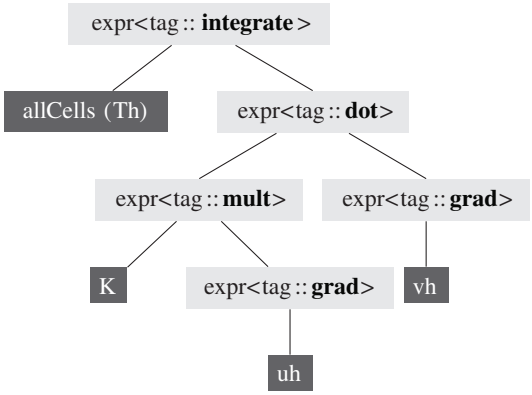

Figure 1

Expression tree for the bilinear form defined in Listing 1. Expressions are in *light gray*, language terminals in *dark gray*.

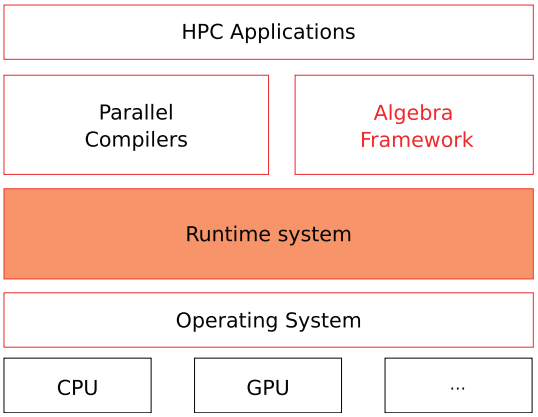

Figure 2

Layer architecture.

2. it enables to take into account the evolution of hardware architecture with new extensions and new concepts implementation, limiting in that way the impact on the numerical layer based on the DSEL generative layer.

Most of the algorithms that could be parallelized rely on this layer that bridges the gap between our DSEL and the low level Application Programming Interface (API) used to execute algorithms on various computational units. For example, this layer has been used:

1. to create an abstract API to manipulate algebraic objects like matrices and vectors with standard Basic Linear Algebra Subprograms (BLAS) 1 and 2 operations;
2. to enhance the assembly part of linear systems, while linear and bilinear expressions are evaluated on collections of mesh items.

Numerical low level algorithms can often been described as sequences of matrices and vectors algebraic operations.

Such sequences with a great number of floating points operations can be expensive, and the way to optimize them are diverse with respect to the hardware specificities. It is important for the generative framework to have a high level unified way to express such sequences independently of low level optimisations. We have for this purpose defined an abstract algebraic API detailed in Listing 3 aiming: (i) to hide hardware specificities, (ii) to manage memory allocation and locality, (iii) to manage parallel loops, (iv) to provide most BLAS 1 and 2 functionalities, (v) to provide tools to split vectors and manage vector views and range iterators.

```
class AlgebraKernel
{
    void allocate(Vector & v,
                  std :: size_t alloc_size);
    void assign(Vector & v, LambdaT op);
    void axpy(ValueT const &a,
              Vector const& x, Vector const&
              y);
    double dot(Vector const& x, Vector const& y);
    void mult(Matrix const& A,
              Vector const& x,
              Vector const& y);
    void exec(Precond const& P,
              Vector const& x,
              Vector const& y);
};
```

We have implemented this API with HARTS and with various other runtime system layers (XKaapi, StarSS, ...). With this API we can compare the following classical BiCGStab sequence to its programming counterpart in Listing 3.

Algorithm 1: BiCGStab Algorithm

```
Matrix A;
Vector b, p, pp, r, v
Scalar a;
do
  pp = inv(P).p;
  v = A.p;
  r += v;
  a = dot(p,r);
  if(a==0) break;
  ...;
while(|r|<tol*|b|);
```

Listing 3
BiCGStab sequence

```
AlgebraKernelType alg;
Matrix A; Vector p, pp, r, v;
double alpha;
// Declare the algorithm sequence
SequenceType seq = alg.newSequence();
alg.exec(precond, p, pp, seq);
alg.mult(A, pp, v, seq);
alg.axpy(1., r, v, seq);
alg.dot(p, r, alpha, seq);
alg.assertNull(alpha, seq);

while(!iter.stop())
{
  // execute the sequence
  alg.process(seq);
}
```

We have seen in Section 1 that the evaluation of bilinear and linear expressions with linear context objects leads to algorithms as in Listing 2. These algorithms consist in iterating on mesh entities, in computing local matrices and vectors which are assembled in a global matrix and right hand side vector. With new hardware architectures, such algorithms can be parallelized in a number of different ways as long as the concurrency on global data like the global linear system is managed. To handle the variety of low level systems, HARTS provides functionalities like HARTS::parallelForeach() (see Listing 3) to iterate on collection of mesh entities (nodes, faces, cells) and to apply lambda functions in parallel.

```
ItemGroupT<Item> items = ...;
parallelForeach(
  items,
  [&](ItemVectorView<Item> const& items)
{
  // Lambda function
  std :: for_each(items.begin(),
                  items.end(),
                  [](Item const item)
                  {
                   ...
                  });
});
```

Using graph coloring techniques, we can manage concurrency on shared data without locks. For that we partition the collections of mesh entities in sub collections of mesh entities with disjoined connectivities, then we apply on each sub collection any lambda functions, avoiding in that way any concurrent access to shared data (Shared degrees of freedom during the assembly phase). Listing 4 shows how the generic assembly algorithm of Listing 2 can be written delegating the parallelism to the runtime system layer.

Listing 4
Parallel integration assembly algorithm

```
template<typename ItemT,
         typename TestExprT,
         typename TrialExprT,
         typename tag_op,
         typename BilinearContextT>
void integrate(Mesh const& mesh,
               GroupT<ItemT> const& group,
               TrialExprT const& trial,
               TestExprT const& test,
               BilinearContextT& ctx)
{
  static const Context :: ePhaseType phase =
  BilinearContextT :: phase_type;
  auto matrix = ctx.getMatrix();

  ColorPartitioner<ItemT, ItemT> part
(mesh, group);
  for(std :: size_t color = 0;
    part.getNbColor();
    ++color)
{
  Harts :: parallelForeach(
    part.getPartition(color),
    [&](ItemVectorView<ItemT> const& items)
{
    for(auto cell : items)
    {
        //! eval context on mesh item
        EvalContext<Item> ctx(cell);

        //! trial linear combination
        auto lu = proto :: eval(trial, ctx);

        //! test linear combination
        auto lv = proto :: eval(test, ctx));

        //! bilinear contribution
        BilinearContribution<tag_op>
        uv (lu, lv);

        //! matrix assemble phase
        assemble<phase>(matrix,
                        measure(mesh,
                        cell), uv);
    }
    });
  }
}
```

3 EXAMPLE OF APPLICATIONS

In this section we present first three academic model problems, the heterogeneous diffusive problem, a linear elasticity problem and the Stokes problem. For each of them, we compare different mathematical formulations to their programming counterpart.

3.1 Diffusive Problem

The countinuous strong formulation reads:

$$\nabla \cdot (-\kappa \nabla u) = f$$

The countinuous variational formulation reads:
Find $u \in H_0^1(\Omega)$ such that

$$a(u,v) = b(v) \quad \forall v \in H_0^1(\Omega)$$

with

$$a(u,v) \stackrel{\text{def}}{=} \int_\Omega \kappa \nabla u \cdot \nabla v$$

$$b(v) \stackrel{\text{def}}{=} \int_\Omega f v$$

This problem can be solved with various lowest-order methods, the hybrid method presented in Section 1, the G-method [11] or the ccG method [12, 13].

In the G-method
The trial space for is obtained with (i) $\mathscr{S}_h = \mathscr{P}_h$, (ii) $\mathbb{V}_h = \mathbb{T}_h$, and (iii) $\mathfrak{G}_h = \mathfrak{G}_h^{\mathrm{g}}$ a gradient operator, piecewise constant on the elements $S \in \mathscr{P}_h$, base on the L construction, detailed in [11].
The method reads then:

$$\text{Find } u_h \in V_h^g \text{ s.t. } a_h^g(u_h, v_h) = \int_\Omega f v_h \quad \text{for all } v_h \in \mathbb{P}_{\mathrm{d}}^0(\mathscr{T}_h)$$

where $a_h^{\mathrm{g}}(u_h, v_h) \stackrel{\text{def}}{=} \Sigma_{F \in \mathscr{F}_h} \int_F \{\kappa \nabla_h v_h\} \cdot \mathrm{n}_f [\![v_{\mathrm{h}}]\!]$ with.
We can compare to the following programming counterpart:

```
MeshType Th(/*...*/);
VariableCellReal 3x3 K(/*...*/);
VariableCellReal f(/*...*/);

// FORMS DECLARATION
auto Uh = newGSpace(Th);
auto u = Uh-> trial();
auto v = Uh-> test();
```

```
Bilinear Form ah_g =
  integrate(
    internalFaces(Th),
    dot(N(Th), avr(K*grad(u)))*jump(v)))
+ integrate(
    boundaryFaces(Th),
    dot(N(Th), avr(K*grad(u)))*jump(v));

LinearForm bh_g =
    integrate(cells(Th), -f*v);

// FORMS EVALUATION
/*...*/
```

In the ccG method

We introduce the linear gradient operator $\mathfrak{G}_h^{\text{green}}: \mathbb{T}_h \times \mathbb{F}_h \to [\mathbb{P}_d^0(\mathscr{T}_h)]^d$ such that, for all $\left(\mathbf{v}^{\mathscr{T}}, \mathbf{v}^{\mathscr{F}}\right) \in \mathbb{T}_h \times \mathbb{F}_h$ and all $T \in \mathscr{T}_h$,

$$\mathfrak{G}_h^{\text{green}}(\mathbf{v}^{\mathscr{T}}, \mathbf{v}^{\mathscr{F}})|_T = \frac{1}{|T|_d} \sum_{F \in \mathscr{F}_T} |F|_{d-1}(v_F - v_T)\mathrm{n}_{T,F} \quad (5)$$

The discrete space for the ccG method is obtained with: (i) $\mathscr{S}_h = \mathscr{T}_h$, (ii) $\mathbb{V}_h = \mathbb{T}_h$, and (iii) $\mathfrak{G}_h = \mathfrak{G}_h^{\text{ccg}}$ with $\mathfrak{G}_h^{\text{ccg}}$ such that

$$\forall \mathbf{v}_h \in \mathbb{V}_h, \quad \mathfrak{G}_h^{\text{ccg}}(\mathbf{v}_h) = \mathfrak{G}_h^{\text{green}}\left(\mathbf{v}_h, \mathbf{T}_h^{\text{g}}(\mathbf{v}_h)\right) \quad (6)$$

where $\mathbf{T}_h^{\text{g}}$ is a linear trace reconstruction operator on the faces of $\mathscr{T}_h$.

Let for all $(u_h, v_h) \in V_h^{\text{ccg}} \times V_h^{\text{ccg}}$,

$$a_h^{\text{ccg}}(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega \kappa \nabla_h u_h \cdot \nabla_h v_h - \sum_{F \in \mathscr{F}_h} \int_F [\kappa \nabla_h u_h \cdot \mathrm{n}_F [\![v_h]\!] + [\![u_h]\!] \kappa \nabla v_h \cdot \mathrm{n}_F] + \sum_{F \in \mathscr{F}_h} \eta \frac{\gamma_F}{h_F} \int_F [\![u_h]\!][\![v_h]\!] \quad (7)$$

The method reads:

$$\text{Find } u_h \in V_h^{\text{ccg}} a_h^{\text{ccg}}(u_h, v_h) = \int_\Omega f v_h \quad \text{for all } v_h \in V_h^{\text{ccg}} \quad (8)$$

We can compare to the following programming counterpart:

```
MeshType Th(/*...*/);
VariableCellReal 3x3 K(/*...*/);
VariableCellReal f(/*...*/);
auto Uh = newCCGSpace(Th);
auto u = Uh-> trial();
auto v = Uh-> test();

// FORMS DECLARATION
BilinearForm ah_ccg =
  integrate(all Cells(Th),
        dot(K*grad(u), grad(v)))
+ integrate(allFaces(Th),
    -K*jump(u)*dot(N(Th), avr(grad(v)))
    -K*dot(N(Th), avr(grad(u)))*jump(v))
+ integrate(internalFaces(Th),
    eta /H(Th)*jump(u)*jump(v));

LinearForm bh_ccg =
    integrate(cells(Th), -f*v);

// FORMS EVALUATION
/*...*/
```

3.2 Linear Elasticity

The countinuous strong formulation reads:

$$-\nabla \cdot \sigma(\mathbf{u}) = \mathbf{f}$$

where $u : \Omega \to \mathbb{R}^d$ is the vector-valued displacement field.

The countinuous variational formulation reads:

Find $u \in [H_0^1(\Omega)]^d$ such that

$$a(u, v) = b(v) \quad \forall v \in [H_0^1(\Omega)]^d$$

with

$$\epsilon(\mathbf{v}) \stackrel{\text{def}}{=} \frac{1}{2}(\nabla \mathbf{v} + \nabla \mathbf{v}^T)$$
$$\sigma(\mathbf{v}) \stackrel{\text{def}}{=} 2\mu\epsilon(\mathbf{v}) + \lambda \nabla \cdot \mathbf{v} I_d$$
$$a(\mathbf{u}, \mathbf{v}) \stackrel{\text{def}}{=} \int_\Omega \sigma(\mathbf{u}) : \epsilon(\mathbf{v})$$
$$b(\mathbf{v}) \stackrel{\text{def}}{=} \int_\Omega \mathbf{f} \cdot \mathbf{v}$$

The discrete variational hybrid method reads [14]:

$$\text{Find } u_h \in [V_h^{\text{hyb}}]^d \text{ s.t. } a_h^{\text{sushi}}(u_h, v_h) = \int_\Omega f v_h \quad \text{for all } v_h \in [V_h^{\text{hyb}}]^d$$

with

$$a_h^{\text{sushi}}(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega \epsilon_h(u_h) : \epsilon_h(v_h) + \lambda \nabla_h \cdot (u_h) \nabla_h \cdot (v_h) \quad (9)$$

We can compare to the following programming counterpart:

```
MeshType Th(/*...*/);
auto Uh = newHybridSpace(Th);
auto u = Uh-> trialArray(''U'', Th::dim);
auto v = Uh-> testArray(''V'', Th::dim);

// BILINEAR AND LINEAR FORMS
BilinearForm ah =
```

```
  integrate(allCells(Th),
         m_2mu*ddot(eps(u), eps(v)))
+ integrate(allCells(Th),
          m_lambda*(div(u)*div(v)));
LinearForm bh =
  integrate(allCells(Th),
          dot(m_f, v));

// Boundary conditions DIRICHLET + NEUNMANN
ah +=
  on(boundary(Th, ''Dirichlet'',
     trace(u) = g);
ah +=
  integrate(boundary(Th, ''Neunmann''),
      alpha*dot(SigmaN(u), v));
bh +=
  integrate(boundary(Th, ''Neunmann''),
          alpha*dot(t, v)););
```

3.3 Stokes Problem

The countinuous strong formulation reads:

$$-\Delta \mathbf{u} + \nabla p = \mathbf{f} \text{ and } \nabla \cdot \mathbf{u} = 0$$

where $u : \Omega \to \mathbb{R}^d$ is the vector-valued velocity field, $p : \Omega \to \mathbb{R}$ is the pressure, and $f : \Omega \to \mathbb{R}^d$ is the forcing term.

The countinuous variational formulation reads:

$$a((\mathbf{u},p),(\mathbf{v},q)) \stackrel{\text{def}}{=} \int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v} + \int_\Omega p \nabla \cdot v + \int_\Omega \nabla q \cdot \mathbf{u}$$
$$b((\mathbf{v},q)) \stackrel{\text{def}}{=} \int_\Omega \mathbf{f} \cdot \mathbf{v}$$

Following [15], we consider a discretization based on the spaces

$$U_h \stackrel{\text{def}}{=} [V_h^{\text{ccg}}]^d, \quad P_h \stackrel{\text{def}}{=} \frac{\mathbb{P}_d^0(\mathscr{T}_h)}{\mathbb{R}} \tag{10}$$

The momentum diffusion is discretized by the bilinear form $a_h \in \mathscr{L}(U_h \times U_h, \mathbb{R})$ such that

$$a_h(u_h, v_h) = \sum_{i=1}^{d} a_h^{\text{sip}}(u_{h,i}, v_{h,i}) \tag{11}$$

where, for all $w_h \in U_h$, the Cartesian components of w_h are denoted by $(w_{h,i})_{i\in\{1,\ldots,d\}}$. The velocity-pressure coupling hinges on the bilinear form $b_h \in \mathscr{L}(U_h \times P_h, \mathbb{R})$:

$$b_h(v_h, q_h) = -\int_\Omega (\nabla_h \cdot v_h) q_h + \sum_{F\in\mathscr{F}_h} \int_F [\![v_h]\!] \cdot n_F \{q_h\} \tag{12}$$

The discrete divergence operator associated to b_h is not surjective with choice of spaces (10). The stability of the velocity-pressure coupling can be recovered by penalizing pressure jumps via the bilinear form $s_h \in \mathscr{L}(P_h \times P_h, \mathbb{R})$ such that

$$s_h(p_h, q_h) = \sum_{F\in\mathscr{F}_h^i} \int_F h_F [\![p_h]\!][\![q_h]\!] \tag{13}$$

The discrete problem reads: Find $(u_h, p_h) \in U_h \times P_h$ such that, for all $(v_h, q_h) \in U_h \times P_h$,

$$a_h(u_h, v_h) + b_h(v_h, p_h) - b_h(u_h, q_h) + s_h(p_h, q_h) = \int_\Omega f \cdot v_h \tag{14}$$

We can compare to the following programming counterpart:

```
MeshType Th;
auto Uh = newCCGSpace(Th);
auto Ph = newPOSpace(Th);
auto u = Uh-> trialArray(Th :: dim);
auto v = Uh-> testArray(Th :: dim);
auto p = Ph-> trial();
auto q = Ph-> test();
FVDomain :: algo :: Range<l> _i(dim);
BilinearForm ah =
  integrate(allCells(Th),
    sum(_i)[dot(grad(u(_i)), grad(v(_i))])
+ integrate(Internal <Face > :: items(Th),
  sum(_i)[
      -dot(N(Th), avg(grad(u(_i))))*jump(v(_i))
      -jump(u(_i))*dot(N(), avg(grad(v(_i))))
      +eta/H(Th)*jump(u(_i))*jump(v(_i))
         ]);
BilinearForm bh =
  integrate(allCells(Th),
          -id(p)*div(v))
+ integrate(allFaces(Th),
         avg(p)*dot(fn, jump(v)));
BilinearForm bth =
  integrate(allCells(Th),
         div(u)*id(q))
+ integrate(allFaces(Th),
          -dot(N(Th), jump(u))*avg(q));
BilinearForm sh =
  integrate(internalFaces(Th),
         H(Th)*jump(p)*jump(q));
LinearForm fh =
integrate(allCells(Th),
        sum(_i)[f(_i)*v(_i)]);
```

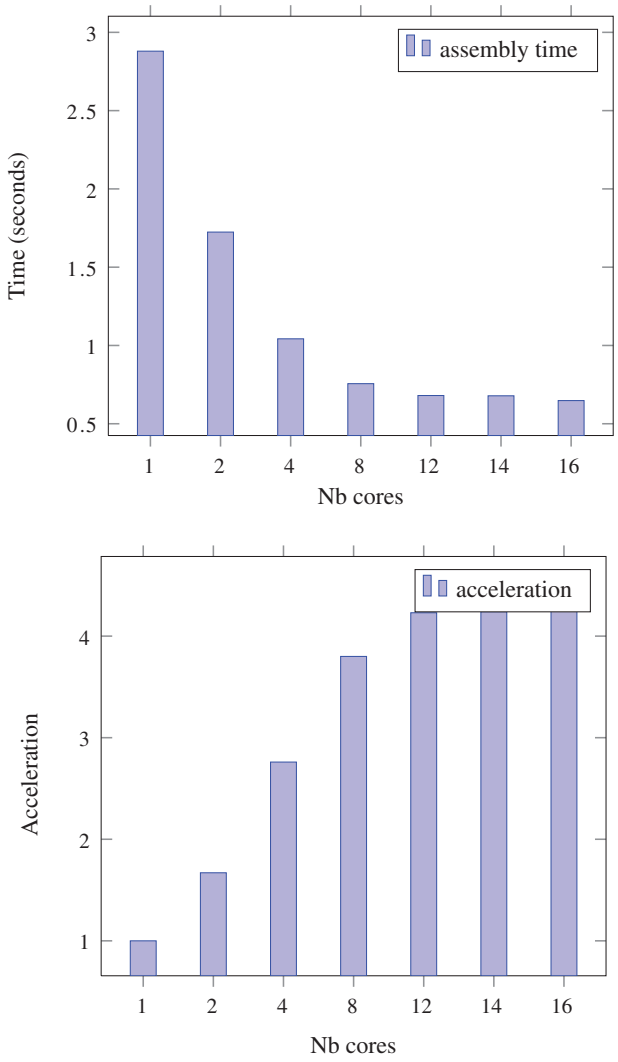

Figure 3

Linear system assembly performance.

4 PERFORMANCE RESULTS

In this section, we present some performance results obtained on the heterogeneous diffusive problem. We focus on the two most expensive parts, the global linear system assembly and the linear system resolution. In Figure 3, we can notice that, even if the linear system assembly is not trivial to parallelize due to concurrent access matrix and vector entries, we can obtain not so bad accelerations. For the linear system resolution, our generic layer ALIEN gives us access to various linear solver packages that can perform on multi-core nodes or on GP-GPU. In Figure 4, we see the solver performance up to 16 cores on multi-cores architecture. We can also see how we can even take advantage of

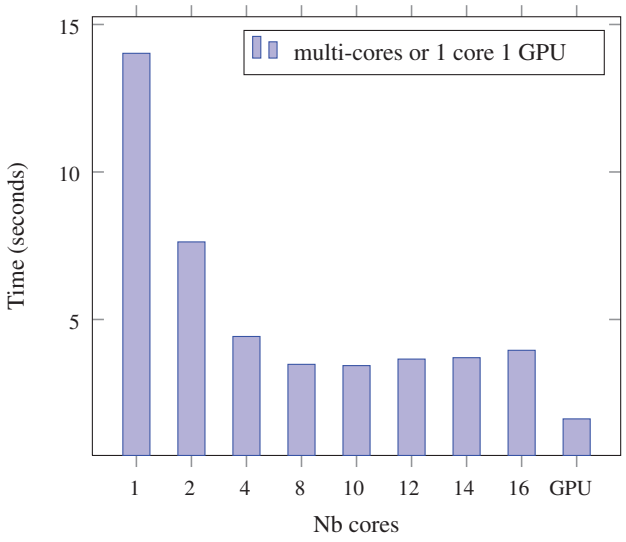

Figure 4

Linear solver performance.

the power of GP-GPU with the dedicated solver package for GP-GPU.

CONCLUSION AND PERSPECTIVE

We have presented ArcFVDSL, a DSEL developed on top of the Arcane framework. This high level language enables to implement various lowest-order methods for diffusive problems on general meshes. It hides low level optimisations for new heterogeneous architecture. We have shown how this generative framework combined to the HARTS layer can generate efficient codes and allows to handle seamlessly architectures with multi-core processors enhanced by GP-GPU. This combination turns to be a good solution to provide high level tools to implement new complex numerical methods preserving the performance of the generated code on heterogeneous hardware architectures. In the future, we plan to introduce at the numerical level, new features in the language to handle for example non linear models. At the hardware level, we plan to introduce new optimisations through the HARTS layer to take advantage on new many integrated cores architectures using for instance Intel Xeon Phi processors.

REFERENCES

1 Grospellier G., Lelandais B. (2009) The arcane development framework, *Proc. of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing, POOSC '09*, ACM, New York, NY, USA, pp. 4:1-4:11, ISBN: 978-1-60558-547-5.

2 Niebler E. (2011) Boost::proto documentation. Available at http://www.boost.org/doc/libs/1_47_0/doc/html/proto.html

3 Di Pietro D.A., Gratien J.-M., Prud'homme C. (2013) A domain specific embedded language in C++ for lowest-order methods for diffusive problem on general meshes, *BIT Numer. Math.* **53**, 111-152.

4 Gratien J.-M. (2012) *Implementing a domain specific embedded language for lowest-order variational methods with Boost Proto*, Available at http://hal.archives-ouvertes.fr/hal-00788281.

5 Gratien J.-M. (2013) *An abstract object oriented runtime system for heterogeneous parallel architecture*, Available at https://hal-ifp.archives-ouvertes.fr/hal-00788293. Working paper or preprint.

6 Di Pietro D.A., Gratien J.-M. (2011) Lowest order methods for diffusive problems on general meshes: a unified approach to definition and implementation, *FVCA6 Proc.*, Available at http://hal.archives-ouvertes.fr/hal-00562500/fr/.

7 Brezzi F., Lipnikov K., Shashkov M. (2005) Convergence of mimetic finite difference methods for diffusion problems on polyhedral meshes, *SIAM J. Numer. Anal.* **43**, 1872–1896.

8 Brezzi F., Lipnikov K., Simoncini V. (2005) A family of mimetic finite difference methods on polygonal and polyhedral meshes, *Math. Models Methods Appl. Sci.* **15**, 1533-1553.

9 Droniou J., Eymard R., Gallouet T., Herbin R. (2010) A unified approach to mimetic finite difference, hybrid finite volume and mixed finite volume methods, *Math. Models Methods Appl. Sci.* **20**, 2, 265-295.

10 Eymard R., Gallouet Th., Herbin R. (2010) Discretization of heterogeneous and anisotropic diffusion problems on general nonconforming meshes SUSHI: a scheme using stabilization and hybrid interfaces, *IMA J. Numer. Anal.* **30**, 1009-1043.

11 Agélas L., Di Pietro D.A., Droniou J. (2010) The G method for heterogeneous anisotropic diffusion on general meshes, *M2AN Math. Model. Numer. Anal.* **44**, 4, 597-625.

12 Di Pietro D.A. (2010) Cell-centered Galerkin methods, *C. R. Math. Acad. Sci. Paris* **348**, 31-34.

13 Di Pietro D.A. (2011) A compact cell-centered Galerkin method with subgrid stabilization, *C. R. Acad. Sci. Paris, Ser. I.* **348**, 1-2, 93-98.

14 Lemaire S. (2013) Nonconforming discretizations of a poromechanical model on general meshes. *PhD Report*, Available at http://www.theses.fr/2013PEST1168/document

15 Di Pietro D.A. (2012) Cell centered Galerkin methods for diffusive problems, *M2AN Math. Model. Numer. Anal.* **46**, 6, 111-144.

Manuscript submitted in December 2015
Manuscript accepted in February 2017
Published online in April 2017

Cite this article as: J.-M. Gratien (2017). ArcFVDSL, a DSEL Combined to HARTS, a Runtime System Layer to Implement Efficient Numerical Methods to Solve Diffusive Problems on New Heterogeneous Hardware Architecture, *Oil Gas Sci. Technol* **72**, 12.