ON THE IMPLEMENTATION OF A SCALABLE SIMULATOR FOR MULTISCALE HYBRID-MIXED METHODS

ANTÔNIO TADEU A. GOMES, WESLLEY S. PEREIRA, FRÉDÉRIC VALENTIN
AND DIEGO PAREDES

Abstract. The family of Multiscale Hybrid-Mixed (MHM) finite element methods has received considerable attention from the mathematics and engineering community in the last few years. The MHM methods allow solving highly heterogeneous problems on coarse meshes while providing solutions with high-order precision. It embeds independent local problems which are responsible for upscaling unresolved scales into the numerical solution. These local contributions are brought together through a global problem defined on the skeleton of the coarse partition. Since the local problems are completely independent, they can be easily computed in parallel. In this paper, we present two simulator prototypes specifically crafted for the MHM methods, which adopt two different implementation strategies: (i) a multi-programming language approach, each language tackling different simulation issues; and (ii) a classical, single-programming language approach. Specifically, we use C++ for numerical computation of the global and local problems in a modular way; for process distribution in the simulator, we adopt the Erlang concurrent language in the first approach, and the MPI standard in the second approach. The aim of exploring these different approaches is twofold: (i) allow for the deployment of the simulator both in high-performance computing (with MPI) and in cloud computing environments (with Erlang); and (ii) pave the way for further exploration of quality attributes related to software productivity and fault-tolerance, which are key to Exascale systems. We present a performance evaluation of the two simulator prototypes taking into account their efficiency.

LNCC - Laboratório Nacional de Computação Científica, 25651-075 Petrópolis, RJ, Brasil

Instituto de Matematicas, PUCV - Pontificia Universidad Catolica de Valparaiso, Chile

E-mail addresses: {atagomes,weslleyp,valentin}@lncc.br, diego.paredes@pucv.cl.

1. Introduction

The numerical simulation of multiscale phenomena, such as fluid flows in porous media and wave propagation in nano-structures, provides effective ways to solve relevant problems in areas such as energy, climate/weather, and health. In this context, the use of Finite Element Methods (FEM) is particularly disseminated in industry and academia to simulate these phenomena. Despite the fact that classical FEM methods (e.g. the Galerkin method) are appropriate for many different problems, their accuracy may seriously decay when they face problems with highly multiscale features. Because of that, many modern FEM techniques have emerged [5, 10]. Among them, of particular interest to this paper, is the family of Multiscale Hybrid-Mixed (MHM) methods originally proposed in [2, 7]. From a mathematical viewpoint, the MHM methods naturally incorporate multiple scales (the MHM-levels) and provide solutions with high-order precision on coarse meshes. From a computational viewpoint, at a lower MHM-level a set of completely independent *local* problems can be easily computed in parallel to provide information to its upper MHM-level (the so-called *global* problem). As a result, the MHM methods are naturally shaped to be used in parallel computing environments. Thereby, the MHM methods appear as a highly competitive option to handle realistic multiscale boundary value problems (BVPs).

The MHM methods have been numerically validated for problems such as the Darcy equation [7], the Stokes/Brinkman models [3], the advective-reactive dominated problems [8], and the linear elasticity problem [6]. In this paper, we present and evaluate two simulator prototypes specifically crafted for the MHM methods, which adopt two different implementation approaches: (i) a multi-programming language approach with dynamic process scheduling; and (ii) a single-programming language approach with static process scheduling. Specifically, we use C++ for numerical computation of the global and local problems in a modular way. As for process distribution in the simulator, we employ the Erlang concurrent, actor-based language integrated with the C++ modules in the first approach, and a MPI-based C++ application in the second approach. The aim of exploring these different approaches is twofold: (i) to allow for the deployment of the simulator both in conventional high-performance computing (HPC) environments (with MPI) and in cloud computing environments (with Erlang); and (ii) to pave the way for further exploration of quality attributes related with software productivity and fault-tolerance, which are key to Exascale systems [4].

We present a preliminary performance evaluation of the two simulator prototypes, taking into account their strong and weak scalability efficiency. We show that, although not as fast as the MPI version, the Erlang version does provide fairly good speedups up to a few hundred cores, even with all node and link failover mechanisms being set up. Besides, despite the lack of formal evidence, it is worth mentioning Erlang's case for software productivity, since it took one of the authors 4 months worth of coding in the Erlang part, from learning a whole new, actor-based programming language to implementing a fully-fledged application with all node and link failover mechanisms enabled.

The remainder of this paper is structured as follows. Section 2 overviews the family of MHM methods. Section 3 presents the key computing primitives of the MHM simulator and how they relate to the algorithm implied by the MHM methods. Section 4 shows the architecture and key design principles that underlie the

two different implementations of the simulator. Section 5 offers a performance evaluation of these two simulator implementations. Finally, Section 6 presents some concluding remarks and perspectives for future work.

2. The MHM methods

To present the MHM methods in a compact way, we consider the BVP to find $u : \Omega \mapsto \mathbb{R}^d$, $d \in \{2,3\}$, so that $\mathcal{L}u = f$ in an open-bounded domain Ω with prescribed homogeneous Dirichlet boundary condition on the polygonal boundary $\partial\Omega$. Hereafter $\mathcal{L}$ is a bounded linear operator. Some definitions will be needed in the sequel.

First, let $\{\mathcal{T}_H\}_{H \geq 0}$ be a familiy of regular triangulations of Ω such that each mesh $\mathcal{T}_H$ decomposes in a set of globally-indexed elements $\{K_t\}_{t \in T}$, with $T \subset Z^+ = \{0, \ldots, N_t - 1\}$. This set defines a *mesh* with N_t elements. The label H refers to the *mesh size*, where H is the maximum diameter of any element in the mesh. Each element K has a boundary ∂K consisting of a set of faces F, and some of these faces (those who are internal to the mesh) may be shared between two elements. Denote by $\partial\mathcal{T}_H$ the set of ∂K, and by $\mathcal{E}_H$ the set in the triangulation of all globally-indexed faces $\{F_e\}_{e \in E}$, with $E \subset Z^+ = \{0, \ldots, N_e - 1\}$. $\mathcal{E}_H$ defines the *skeleton* of a mesh with N_e faces. Figure 1 illustrates these concepts for $\mathbb{R}^2$. We associate each face of E to a unitary normal vector $\mathbf{n}$, taking care to ensure this is facing outward on $\partial\Omega$. Also, the unitary outward normal vector on a face F of element K is represented by $\mathbf{n}_F^K$.
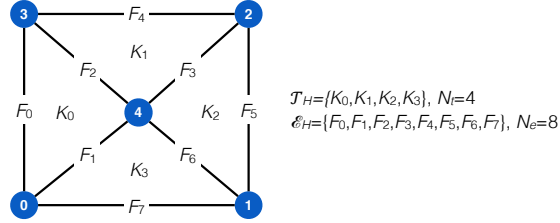

$\mathcal{T}_H = \{K_0, K_1, K_2, K_3\}$, $N_t = 4$
$\mathcal{E}_H = \{F_0, F_1, F_2, F_3, F_4, F_5, F_6, F_7\}$, $N_e = 8$

FIGURE 1. Mesh example in $\mathbb{R}^2$.

Second, we define some function spaces:
 (1) V_0 is the null space (with basis $\{v_0, \ldots, v_{N_{V_0}-1}\}$) of operator $\mathcal{L}|_K$ for all $K \in \mathcal{T}_H$ (e.g., for the Darcy equation V_0 is formed by piecewise constants in each K [7], and then $N_{V_0} = N_t$.);
 (2) $V_0^\perp := \bigoplus_{K \in \mathcal{T}_H} V_0^\perp(K)$ is the orthogonal complement of V_0 such that it yields the wellposedness of $\mathcal{L}|_K$ for all $K \in \mathcal{T}_H$;
 (3) $V := \bigoplus_{K \in \mathcal{T}_H} V_K$ is the space defined as the direct sum between V_0 and $V_0^\perp$ (e.g., for the Darcy equation $V_K = H^1(K)$ [8]);
 (4) Λ is the space containing the Lagrange multipliers defined over $\partial\mathcal{T}_H$ [2].

For these spaces, we also define the L^2 inner products $(\cdot,\cdot)_{\mathcal{T}_H} := \sum_{K \in \mathcal{T}_H}(\cdot,\cdot)_K$ and $(\cdot,\cdot)_{\partial\mathcal{T}_H} := \sum_{K \in \mathcal{T}_H}(\cdot,\cdot)_{\partial K}$, respectively.

Third, we define the following finite-dimensional space needed for approximating the exact field u:

(i) $\Lambda_H^{m,l} \subset \Lambda$ is the space of discontinuous polynomial functions on $F_e \in E$ of degree less than or equal to $l \geq 0$ and defined on a uniform partition of F_e, composed of m elements ($m \geq 1$), with basis $\{\psi_0, \ldots, \psi_{L_l-1}\}$.

Given these definitions, a BVP can be written within the MHM terminology in terms of a collection of local problems—each K being associated with a different local problem—brought together by a global problem defined on $\mathcal{T}_H$. Specifically, the global formulation is as follows: Find $(\lambda_H, u_0) \in \Lambda_H^{m,l} \times V_0$ such that

$$(1) \qquad \begin{cases} (T_h\lambda_H, \mu_l)_{\partial\mathcal{T}_H} + (u_0, \mu_l)_{\partial\mathcal{T}_H} & = & -(\hat{T}_h f, \mu_l)_{\partial\mathcal{T}_H} \\ (\lambda_H, q_0)_{\partial\mathcal{T}_H} & = & (f, q_0)_{\mathcal{T}_H} \end{cases}$$

for all $(\mu_l, q_0) \in \Lambda_H^{m,l} \times V_0$. L_l is the total number of degrees of freedom (DoFs) for $\Lambda_H^{m,l}$, and its value depends on the adopted degree l of the piecewise polynomials in $\Lambda_H^{l,m}$, the total number of partitions m in each face F_e, and the total number of faces N_e in the mesh. In the above formulation, f is given; T_h and $\hat{T}_h$ are linear bounded operators driven by the local problems. Figure 2(a) illustrates an example of the distribution of the DoFs for u_0 in the case of the Darcy equation and λ_H (with $l = 1$ and $m = 1$) in the mesh of Figure 1.

The local operators T_h and $\hat{T}_h$ in (1) are responsible for the upscaling process. To precise them, let $V_h(K_t) \subset V_0^\perp(K_t)$ be a finite-dimensional space (with basis $\{\phi_0^t, \ldots, \phi_{D_{K_t}-1}^t\}$) defined within an element K_t in the mesh. At the local level, each element K_t is considered a domain on its own and as such may have an internal triangulation, thus defining a *submesh* within K_t. Each of these submeshes may have a different characteristic size $h \geq 0$, depending on the high-contrast aspects of the coefficients involved in operator $\mathcal{L}$ for each K_t. The number of DoFs D_{K_t} in $V_h(K_t)$ will therefore depend on the degree of the polynomial functions in $V_h(K_t)$ and h. For the purposes of this paper, we assume the solution to each local problem is approximated using the continuous Galerkin method with a Lagrange basis for $\mathbb{P}_h^k$ (the space of piecewise polynomials of degree less than or equal to k); D_{K_t} is then determined by the degree k of the polynomials and the number of elements in the submesh. As such, we employ the bilinear form $a(u_h, w_h)_{K_t}$ to set up operators T_h and $\hat{T}_h$. It is defined (as usual) as the action of $\mathcal{L}u_h$ on w_h in K_t, with $u_h, w_h \in V_h(K_t)$. Figure 2(b) illustrates an example of $V_h(K_t)$ with quadratic interpolation $\mathbb{P}_h^2$ for element K_3 in the mesh of Figure 1. In the example, K_3 has been internally triangulated generating a submesh with 4 elements.

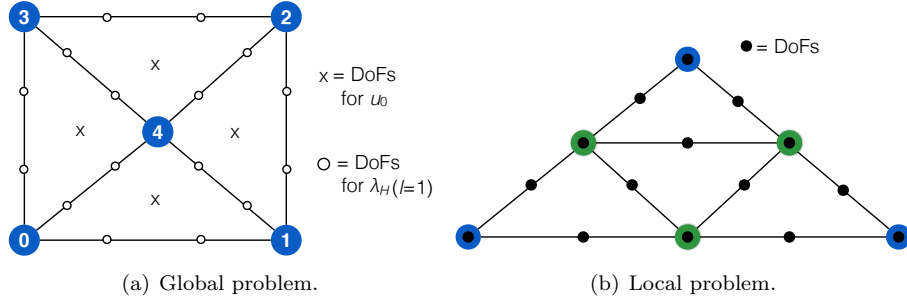(a) Global problem.   (b) Local problem.

FIGURE 2. DoFs distribution in MHM.

Each local problem comprises a set of local *subproblems*. One such subproblem defines $\hat{T}_h$ by computing $\eta_t^f := \hat{T}_h f \in V_h(K_t)$ in each $K_t \in \mathcal{T}_H$ as follows: Find $\eta_t^f \in V_h(K_t)$ such that

$$(2) \qquad a(\eta_t^f, w_h)_{K_t} = (f, w_h)_{K_t} \quad \text{for all } w_h \in V_h(K_t).$$

The other N_I subproblems define T_h by computing $\eta_t^i := T_h \psi_i$ from the following local problems: For $i \in \{0, \dots, N_I - 1\}$, find $\eta_t^i \in V_h(K_t)$ such that

$$(3) \qquad a(\eta_t^i, w_h)_{K_t} = -(\psi_i, w_h)_{\partial K_t} \quad \text{for all } w_h \in V_h(K_t),$$

where ψ_i changes its sign accordingly to $\mathbf{n} \cdot \mathbf{n}_F^{K_t}$, and N_I is defined by the number of faces in K_t and by the degree l of the polynomials and m in $\Lambda_H^{m,l}$. Finally, from the solution of Equations (1), (2) and (3), and using that $\lambda_H = \sum_{i=0}^{N_I - 1} k_i \psi_i$ with $k_i \in \mathbb{R}$, the exact field u is approximated by

$$u_{H,h} := u_0 + T_h \lambda_H + \hat{T}_h f$$

$$(4) \qquad = u_0 + \sum_{K_t \in \mathcal{T}_H} \Big[\sum_{i=0}^{N_I - 1} k_i \eta_t^i + \eta_t^f \Big].$$

3. The simulator primitives

The MHM simulator is built on top of key computing primitives that bear correspondence with the major mathematical building blocks of the MHM method: SPLITPROBLEM, SOLVELOCALPROBLEM, REDUCELOCALPROBLEMS, and COMPUTESOLUTION. These primitives are combined to implement the MHM simulator as depicted abstractly in Algorithm 1.

Algorithm 1 Pseudocode of *main* procedure for the MHM simulator.

Require: $\mathcal{T}_H = \{K_t\}_{t \in T}$ ▷ Mesh with indexed elements
   $P_{\mathcal{T}_H} \leftarrow$ SPLITPROBLEM$(\mathcal{T}_H)$ ▷ $P_{\mathcal{T}_H} := \{(K_t, \{F_t\}\}_{t \in T}$
   $S_{\mathcal{T}_H} \leftarrow \emptyset$
   for all $p_t \in P_{\mathcal{T}_H}$ **do**   ▷ $p_t := (K_t, \{F_t\})$
     $s_t \leftarrow$ SOLVELOCALPROBLEM(p_t) ▷ $s_t := (\eta_t^f, \{\eta_t^i\}_{i \in \{0, \dots, N_I - 1\}})$
     $S_{\mathcal{T}_H} \leftarrow S_{\mathcal{T}_H} \cup s_t$
   end for
   $(u_0, \lambda_H) \leftarrow$ REDUCELOCALPROBLEMS$(S_{\mathcal{T}_H})$
   $u_{H,h} \leftarrow$ COMPUTESOLUTION$(u_0, \lambda_H, S_{\mathcal{T}_H})$

The SPLITPROBLEM primitive divides the original problem posed over a mesh resulting from a triangulation $\mathcal{T}_H$ into a global problem (the 1st MHM-level) and a set of local problems (the 2nd MHM-level), one for each element K_t in the mesh. This primitive returns a set $P_{\mathcal{T}_H}$ of pairs $(K_t, \{F_t\})$, where $\{F_t\}$ represents the set of all faces of K_t.

The SOLVELOCALPROBLEM computes the contributions η_t^f and $\{\eta_t^i\}_{i \in \{0, \dots, N_I - 1\}}$ from a specific element K_t in the mesh to be used in the global problem. Functions η_t^f and η_t^i, $i \in \{0, \dots, N_I - 1\}$ are searched as the solution of (2) and (3), respectively,

in the following form:

$$(5) \qquad \eta_t^f = \sum_{j=0}^{D_{K_t}-1} c_j^t \phi_j^t \quad \text{and} \quad \eta_t^i = \sum_{j=0}^{D_{K_t}-1} d_{i,j}^t \phi_j^t.$$

The coefficients c_j^t and $\{d_{i,j}^t\}_{i\in\{0,\dots,N_I-1\}}$ are the actual numerical results computed by the SOLVELOCALPROBLEM primitive through the solution of a set of linear systems of the form $\underline{\mathbf{A}}\mathbf{C} = \mathbf{F}$ and $\{\underline{\mathbf{A}}\mathbf{D}_i = \mathbf{N}_i\}_{i\in\{0,\dots,N_I-1\}}$. It is worth noting that these systems share the same matrix $\underline{\mathbf{A}} := [a(\phi_j^t, \phi_i^t)]_{i,j\in\{0,\dots,D_{K_t}-1\}}$, albeit with different load vectors. This feature means that $\underline{\mathbf{A}}$ needs to be factorized only once (for linear problems), and therefore the increase in the value of degree l or m in the approximation space $\Lambda_H^{m,l}$ only increases the number of matrix-vector multiplications, which can be trivially parallelized even within the context of a single local problem by means of vectorized computations. Moreover, if the coefficients associated with the operator $\mathcal{L}$ are not time-dependent, $\underline{\mathbf{A}}$ needs to be assembled and factorized only during the first timestep in a time marching scheme.

The REDUCELOCALPROBLEMS primitive gathers the contributions η_t^f and $\{\eta_t^i\}_{i\in\{0,\dots,N_I-1\}}$ from the subproblems to compute u_0 and λ_H from Equation (1). The underlying linear system associated to (1) reads:

$$(6) \qquad \begin{pmatrix} \underline{\mathbf{A}} & \underline{\mathbf{B}}^{\mathrm{T}} \\ \underline{\mathbf{B}} & \underline{\mathbf{0}} \end{pmatrix} \begin{pmatrix} \mathbf{L} \\ \mathbf{P} \end{pmatrix} = \begin{pmatrix} \mathbf{E} \\ \mathbf{F} \end{pmatrix},$$

with:

$$\underline{\mathbf{A}} := [(\psi_j, \eta_t^i)]_{i,j\in\{0,\dots,L_l-1\}},$$
$$\underline{\mathbf{B}} := [(\psi_j, \upsilon_i)]_{i\in\{0,\dots,N_{V_0}-1\},j\in\{0,\dots,L_l-1\}},$$
$$\mathbf{E} := [-(\psi_j, \eta_t^f)]_{j\in\{0,\dots,L_l-1\}},$$
$$\mathbf{F} := [(\upsilon_j, f)]_{j\in\{0,\dots,N_{V_0}-1\}}.$$

The computed entries $[k_j]_{j\in\{0,\dots,L_l-1\}}$ in $\mathbf{L}$ define the solution to λ_H in $\Lambda_H^{m,l}$; likewise, the computed entries $[b_j]_{j\in\{0,\dots,N_{V_0}-1\}}$ in $\mathbf{P}$ define the solution u_0 in V_0 through

$$(7) \qquad u_0 = \sum_{j=0}^{N_{V_0}-1} b_j \upsilon_j.$$

Finally, owing to the COMPUTESOLUTION primitive, the two-level numerical solution $u_{H,h}$ reads

$$u_{H,h} = \sum_{j=0}^{N_{V_0}-1} b_j \upsilon_j + \sum_{K_t\in\mathcal{T}_H} \sum_{j=1}^{D_{K_t}-1} \Big[\sum_{i=0}^{N_I-1} k_i d_{i,j}^t \phi_j^t + c_j^t \phi_j^t \Big].$$

where we used (7) and (5) in (4).

4. IMPLEMENTATION DETAILS

The MHM simulator is a distributed system. It exploits either Erlang's or MPI support for scalable computing and C++'s efficiency for the numerical computations of the global and local problems. The overall architecture of the MHM simulator is depicted in Figures 3(a) and 3(b) for the Erlang and MPI versions, respectively. The following subsections describe the two "layers" (distribution and computation) of the architecture.
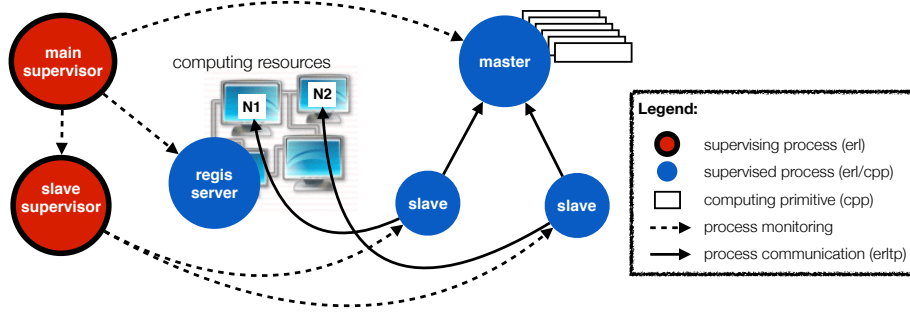
(a) Erlang version.
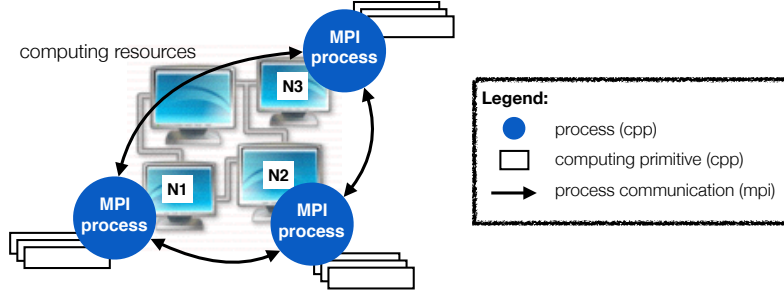
(b) MPI version.

FIGURE 3. Architecture of the MHM simulator.

4.1. The distribution layer.

4.1.1. *The Erlang version.* Before describing this version of the distribution layer, we give a short introduction to the Erlang language and its runtime, including its fault-tolerance mechanisms.

An Erlang prime. Erlang is a concurrent language with functional programming features. It is a language that compiles to an intermediate format (*byte codes*) and runs in virtual machines (the Erlang *nodes*). Typically, each Erlang node runs on a different physical machine.

Erlang programs are built on functions that can be called from the same process of the calling function or dispatched (*spawned*) to another process residing in the same or a different Erlang node. Processes in Erlang run concurrently and communicate with each other by message passing. They usually follow a common pattern: once spawned they call a tail-recursive function to process and produce data. This data is referred to as the process *state*. The tail-recursive function usually takes the following sequence of operations: (i) receive a message from another process, (ii) handle the message, (iii) update the process state, and (iv) pass the updated state to a new invocation of the function through its tail-recursive call. Finally, a special *stop* message is usually defined to allow the receiving process to be terminated gracefully.

The Erlang runtime provides templates—called *behaviors*—that implement the above pattern for processes acting as servers (the `gen_server` behavior) and finite state machines (FSMs – the `gen_fsm` behavior), among others. Building on its messaging mechanism, Erlang offers process monitoring capabilities that allow a process to detect whether another process has failed and act accordingly. This is explored in an specific behavior called `supervisor`. A supervisor is a process whose main task is to manage—spawn, monitor, and eventually re-spawn—other processes in case of failure. Supervisors may manage any type of Erlang process, including other supervisors. This organization creates what is called a *supervision tree*. Finally, an Erlang *application* is a special kind of behavior that allows an entire supervision tree to be started and stopped as a unit. To provide tolerance to hardware faults, an Erlang application can be controlled in a distributed manner, by employing two or more Erlang nodes running in different machines. This setup configures a *distributed application*. If the Erlang node where a distributed application goes down (e.g. because of a network partition or machine failure), the application can be restarted at another node.

The simulator distributed application. Figure 3(a) illustrates the chief processes of the Erlang-based simulator distributed application. The *main supervisor* implements the `supervisor` behavior. It manages a *master*, a *regis server*, and a *slave supervisor*, which are all spawned at the application bootstrap. All these processes run in a single Erlang node; if this node becomes unavailable, they are resumed in another Erlang node with their state preserved thanks to the use of the Mnesia fault-tolerant database embedded in the Erlang runtime. This is the first key feature that allows the Erlang-based simulator to have a very fine-grained fault tolerance support.

The master implements the `gen_server` behavior. It keeps track of a set of *computing tasks*, which are dynamically scheduled to the available computing resources, as described below. Each task is related to either a global or a local problem in the MHM methods, and has an associated state—"not solved", "being solved", "being reduced" (for global problems only) or "solved". Besides, each task maps onto an instance of the simulator primitives described in Section 3. These primitives, which are implemented in C++ (see Section 4.2), run on computing resources managed by the regis server, which also implements the `gen_server` behavior. The regis server collects status information about the computing resources in the execution platform (e.g. computing nodes in an HPC cluster or virtual machines in a cloud) and, for each such resource, asks the slave supervisor (another implementation of the `supervisor` behavior) to dynamically create/destroy new *slaves* associated with the resource whenever the resource becomes available/unavailable. This is the second key feature for the fine-grained fault tolerance support in the Erlang-based simulator.

The slaves implement the `gen_fsm` behavior. They run in the same Erlang node as the other processes of the Erlang application. Once started, an slave keeps on asking the master about new computing tasks. The master replies to a request from a slave by: (i) selecting an appropriate task, (ii) changing the state of the task to "being solved"/"being reduced", and (iii) replying to the slave with the simulator primitive to be executed. The slave then dispatches the execution of the primitive onto its associated computing resource, and keeps track of the primitive execution. When the slave detects that the primitive is finished, it sends the results

of the primitive to the master, which may change the state of the task or create new tasks. If the slave detects that the primitive hasn't completed because of a failure in a computing resource, it notifies the master and the regis server, which dynamically reallocate the task to a different resource.

Figure 4 illustrates how the state of the tasks evolves in the simulator. The simulator starts with a single task representing a global problem—$\mathcal{T}_H$ in Algorithm 1—being published in the master with the "not solved" state. At this point a single slave will be able to get this task, which triggers the dispatching of the SPLITPROBLEM primitive to its associated computing resource and the changing of its state to "being solved". The result of this primitive—$P_{\mathcal{T}_H}$ in Algorithm 1—is taken by the slave as a set of requests—one for each $p_t \in P_{\mathcal{T}_H}$—for the creation of new tasks in the master. Each of these new tasks is related to a different local problem and added to the master with the "not solved" state. To simplify the illustration, in Figure 4 we only present two local problems being derived from the SPLITPROBLEM primitive.
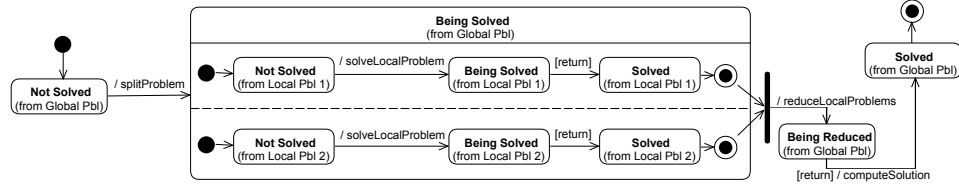

FIGURE 4. Evolution of task states.

From this point on the slaves start to get different tasks related with local problems, changing their states to "being solved" and dispatching the execution of the SOLVELOCALPROBLEM primitive to their associated computing resources. We employed the sequential LU factorization implementation of the Eigen library[1] within this primitive for solving the linear systems that compute c_j^t and $\{d_{i,j}^t\}_{i\in\{0,\ldots,N_I-1\}}$ in Equation (5), with the intention to make as many local problems be solved in parallel in a single computing resource as the amount of cores the resource provides. When these primitive invocations are finished, the slaves send their results—$s_t \in S_{\mathcal{T}_H}$ in Algorithm 1—to the master, which changes the state of their corresponding tasks to "solved". The slaves then proceed with getting other tasks and dispatching them to their resources until there are no more tasks in the "not solved" state.

Once all local problems are in the "solved" state, the next request from an slave is replied with the REDUCELOCALPROBLEMS primitive, and the state associated with the global problem is changed from "being solved" to "being reduced". Again, we employed the LU factorization within this primitive to solve the linear system in Equation (6). For the REDUCELOCALPROBLEMS primitive, however, we used the shared-memory parallel Pardiso solver.[2] The result from the REDUCELOCAL-PROBLEMS primitive triggers the immediate execution of the COMPUTESOLUTION primitive and a changing of the task's state to "solved" in the master. After that, the simulator finishes.

---

[1]http://eigen.tuxfamily.org
[2]http://www.pardiso-project.org

4.1.2. *The MPI version.* Figure 3(b) illustrates the chief processes of the MPI-based application. The R MPI processes that make part of the application are omnipotent; in contrast with the Erlang version, there's no master or slave process. Each MPI process $r \in R$ is responsible for running its own *share* of the SPLITPROB-LEM primitive. This share corresponds to the execution of SPLITPROBLEM over a partition $\mathcal{T}_H^r$ of $\mathcal{T}_H$. Such a partitioning configures a static process scheduling that is kept until the end of the simulation.

The result of the SPLITPROBLEM primitive at each MPI process ($P_{\mathcal{T}_H^r}$) is taken by this process as a set of local problems to be solved—one for each $p_t \in P_{\mathcal{T}_H^r}$. From this point on the MPI processes start to execute the SOLVELOCALPROBLEM primitive for all p_t they are responsible for, in the same manner as the Erlang version.

An MPI barrier is used for making all MPI processes wait for each other as they finish solving all their corresponding local problems. Once all local problems are solved, each MPI process $r \in R$ is responsible for running its own share of the REDUCELOCALPROBLEMS primitive. This is where the MPI version of the MHM simulator mostly differs from the Erlang counterpart. A share of the REDUCE-LOCALPROBLEMS primitive corresponds to the distributed assembly of the global linear system shown in Equation (6), and its solution through LDL^T factorization using the distributed interface of the parallel PaStiX library [9]. The result from the REDUCELOCALPROBLEMS primitive triggers at each MPI process the immediate execution of its own share of the COMPUTESOLUTION primitive. After that, the simulator finishes.

As it can be seen from Figure 3(b), this implementation is structurally simpler than the one of the Erlang-based simulator. Most of such simplicity is due to the lack, in this implementation, of a dynamic process scheduling scheme and of any finer-grained fault-tolerance mechanisms. This means that a failure in a single computing resource running an MPI process crashes the whole application. In addition, checkpoints for the fast restart of a crashed application are only available in two points: at the end of the SPLITPROBLEM primitive, and at the end of all invocations of the SOLVELOCALPROBLEM primitive. Therefore, a failure during the solution of a single local problem results in *all* local problems being solved again, but if all local problems have been solved, the application can be restarted from the point of the REDUCELOCALPROBLEMS primitive.

4.2. **The computation layer.** C++ is used for implementing the numerical computations that ultimately solve the global and local problems. The C++ layer is divided into 2 packages.

The first package implements key concepts present in any FEM-based simulator as C++ classes: (i) geometric elements (`Element`) and meshes (`Mesh`); (ii) quadrature rules for numerical integration (`Quadrature`); (iii) spaces of approximate functions (`Space`); (iv) basis function definitions for elements (`Brick`) and meshes (`Block`); (v) coefficient and source term definitions (`SimpleData`); (vi) contribution computations (`LocalContribSet`); and (vii) a wrapper to linear solver libraries (`LinearSolver`). Figure 5 shows a UML diagram with the main relationships between these classes.

The `Block` class is an abstract class; it must be specialized in classes depending on where the data comes from. The `GalerkinLagrangeBlock` class shown in
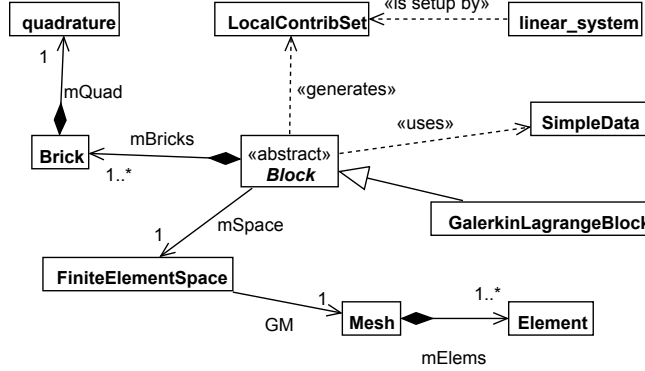
FIGURE 5. UML diagram of main C++ classes.

Figure 5, for instance, compute its values from the definitions of basis functions in `FiniteElementSpace`.

The second package comprises a set of C++ classes that implement the framework for MHM methods: (i) global problem definitions (`Problem`); (ii) local problem definitions (`LocalProblem` and its direct subclasses `StationaryLocalProblem` and `TransientLocalProblem`); (iii) definitions of spaces V_0 and $\Lambda_H^{m,l}$ (`MHMSpace` and subclasses of `LocalSpace`); and (iv) a subclass of Block (`MHMBlock`). Figure 6 presents a UML diagram that depicts the main relationships between these classes.

The `GlobalProblem` class implements the simulator for Equation (1). Its methods `GlobalProblem::split()`, `GlobalProblem::reduce()`, and `GlobalProblem::solve()` correspond to the primitives SPLITPROBLEM, REDUCELOCALPROBLEMS, and COMPUTESOLUTION in Algorithm 1.

`LocalProblem` is an abstract class responsible for: (i) defining a submesh within an element $K_t \in \mathcal{T}_H$, (ii) solving a local problem in this submesh, and (iii) computing the local contributions related with this submesh to the global problem. Its main method—`LocalProblem::solveLocalProblem()`—corresponds to primitive SOLVELOCALPROBLEM in Algorithm 1. The subclasses `StationaryLocalProblem` and `TransientLocalProblem` specialize the behavior of this method to stationary and transient problems. Specializations of these two subclasses then override `solveLocalProblem()` to provide concrete behavior. For instance, the `DiffusionSubproblem` class specializes `StationaryLocalProblem` to solve the following local subproblems, which particularize the general Equations (2) and (3) for the Darcy equation [7]: Find $\eta_t^f \in V_h(K_t)$ such that

$$(8) \qquad (\kappa\nabla\eta_t^f, \nabla w_h)_{K_t} = (f, w_h)_{K_t} \quad \text{for all } w_h \in V_h(K_t),$$

and for $i \in \{0, \ldots, N_I - 1\}$, find $\eta_t^i \in V_h(K_t)$ such that

$$(9) \qquad (\kappa\nabla\eta_t^i, \nabla w_h)_{K_t} = -(\psi_i, w_h)_{\partial K_t} \quad \text{for all } w_h \in V_h(K_t),$$

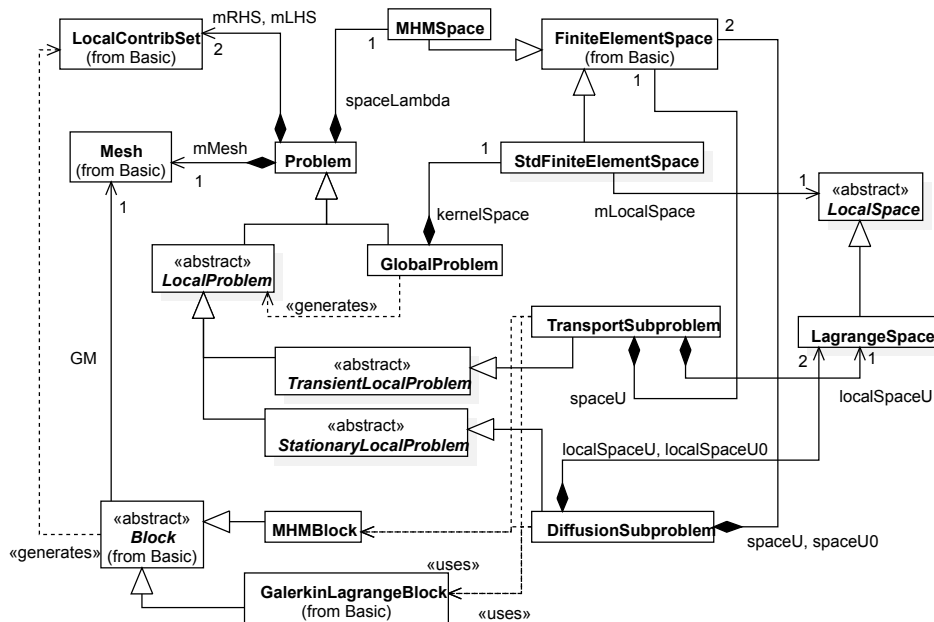where κ is the diffusion coefficient.

FIGURE 6. UML diagram of the C++ classes that implement the framework for MHM methods.

5. PERFORMANCE EVALUATION

We collected strong and weak scaling performance measurements for the two simulator prototypes. For these measurements, we ran the two prototypes in the Santos Dumont HPC cluster at LNCC in Brazil, which has the following per-node configuration: 2x CPU Intel Xeon E5-2695v2 Ivy Bridge (12 cores each CPU), 2.4GHZ, 64GB DDR3 RAM. The nodes of the cluster are interconnected through an Infiniband FDR network with a fat-tree topology, and they share a distributed file system based on Lustre v2.1. The C++ code was compiled with Intel's icpc compiler, version 16.0.2 build 20160204, with the optimization flags '$-O2$' and '$-ipo$' enabled. We used Eigen version 3.3-rc1, PaStiX version 5.2.2.22, and the Pardiso implementation provided with the MKL library that comes with Intel's Parallel Studio XE 2016.2.062. The Erlang code was run with Ericsson's Open Source Erlang/OTP system, version 17 erts-6.1, with HiPE (High Performance Erlang) native code compilation enabled. The MPI implementation used was the one that also comes with Intel's Parallel Studio XE 2016.2.062.

We solve the Darcy equation defined in the unit cube as a model problem. The diffusion coefficient is set as $\kappa = 1$ and $f(x, y, z) = 12\pi^2 \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$. As such, the exact solution reads $u(x, y, z) = \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$. We setup the MHM simulators with 24, 192 and 1,536 elements in a global tetrahedral mesh and within each such element a submesh of either 59 or 6,046 tetrahedra dynamically generated by the TetGen library [11] during runtime, as part of the computation within the SOLVELOCALPROBLEM primitive. We adopted degree $l = 0$

and $m = 4$ (59 tetrahedra case) or $m = 16$ (6,046 tetrahedra case) for the polynomials in $\Lambda_H^{m,l}$, and degree $k = 2$ for the polynomials in the submesh. We depict the 192 tetrahedra mesh along with its submeshes (59 tetrahedra per element) in Figure 7(a), and the corresponding numerical solution to our model problem in Figure 7(b).

(a) Mesh with 192 elements including the submeshes.

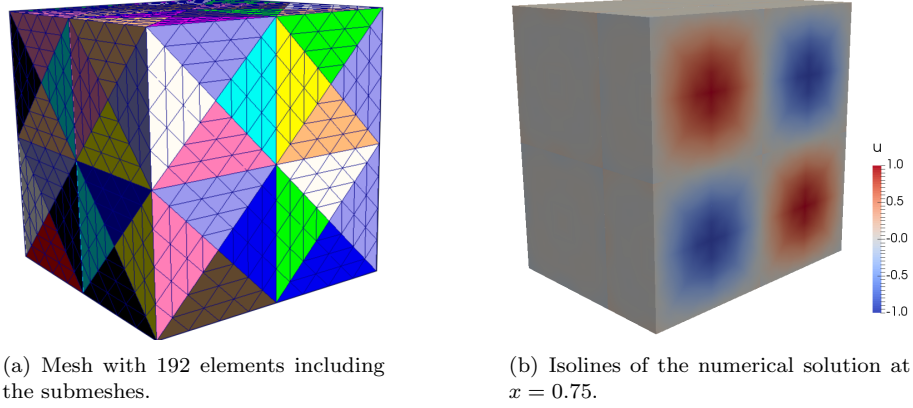(b) Isolines of the numerical solution at $x = 0.75$.

FIGURE 7. A model problem.

For the sake of illustrative comparison, we present in Table 1 the number of DoFs that the standard finite element (Galerkin) method should have to reach a similar approximation order (in terms of accuracy) compared to the one using the MHM methods. As shown in [7], since we are using $l = 0$ and $k = 2$ with the MHM methods, the numbers of DoFs in the table correspond to the use of the linear space $\mathbb{P}_h^1$ with the Galerkin method. We emphasize that, different from what happens within the MHM methodology, the standard approximation using Galerkin needs to compute all such DoFs at once. This is achieved by solving a single linear system which is typically much larger than those generated by the global and local problems of the MHM methods.

The setup described above renders 6 different MHM parameterizations, but we do not present any results associated to the configuration with $1,536 \times 59$ tetrahedra. The reason is that they are much smaller problems and therefore much more subject to statistical variations due to the execution environment. We tested the other 5 configurations in 3 resource settings: 24, 192 and 768 cores. For each of the 15 configurations, 3 runs were conducted. We selected the run with the best overall wall-clock time for presentation herein. (It is worth noticing that no significant statistic variations were identified in the results.) In the case of the resource setting

TABLE 1. Correspondence between MHM parameterization and Galerkin DoFs for $\mathbb{P}_h^1$.

		Global mesh size	
		$1,536$	$12,288$
Submesh size	59	$16,689$	$127,073$
	$6,046$	$1,493,697$	$11,850,113$

with 24 cores, since it occupies a single node in the cluster used for the experiments, we used a baseline version of the MHM simulator that only uses multithreading, without any Erlang or MPI overhead. The purpose is to better capture the impact of distribution in the Erlang and MPI versions of the simulator.

Table 2 consolidates all the scalability measurements. Analyzing the strong scalability, we can observe in the table that the efficiency of both simulators improves as the problems get bigger. We certainly need to make measurements with larger scenarios, but we believe this is a promising result. Another observation is that the Erlang version of the simulator is less efficient than the MPI one—which was expected—, but not by a large margin. This is another interesting result because it opens opportunities for the efficient use of the MHM approach in cloud computing environments, and more generally, for the use of Erlang as an appropriate coordination language for fault-tolerant HPC applications.

TABLE 2. Scalability measurements.

Problem size		Baseline 24 cores	MPI 192 cores	768 cores	Erlang 192 cores	768 cores
$12,288 \times 59 =$						
$724,992$ tetrahedra	Time (sec)	**129.81***	57.74	72.88	97.00	97.00
	Efficiency	100.00%	28.10%	5.57%	16.62%	4.16%
$1,536 \times 6,046 =$						
$9,286,656$ tetrahedra	Time (sec)	489.00	**82.57***	51.72	**108.00***	65.00
	Efficiency	100.00%	74.03%	29.54%	56.60%	23.51%
$12,288 \times 6,046 =$						
$74,293,248$ tetrahedra	Time (sec)	$4,753.00$	641.61	**220.20***	667.00	**240.00***
	Efficiency	100.00%	92.60%	67.45%	89.70%	61.89%

The entries in Table 2 that are in bold and marked with a '$*$' can be also used for a weak scalability analysis. We first point out that from the 1st to the 3rd configuration, using the Erlang version, there was an increase of the time of execution by a factor of ≈ 1.84, so the weak scalability efficiency was of $54,34\%$. As for the MPI version, there was an increase of this time by a factor of ≈ 1.69, with an efficiency of 58.95%. We must nevertheless compare these efficiency assessments with 3 different ways to observe the increase in the size of the corresponding problems:

(1) Taking the total number of tetrahedra in each configuration: from the 1st to the 3rd configuration there is an increase of ≈ 102.47 times the number of tetrahedra;

(2) Taking the correspondence with Galerkin DoFs for $\mathbb{P}_h^1$ as presented in Table 1: from the 1st to the 3rd configuration there is an increase of ≈ 61 times the number of DoFs;

(3) Taking the approximation error in the L^2-norm of the computed solution: from the 1st to the 3rd configuration there is a reduction of ≈ 7 times the error to the exact solution (from 0.151×10^{-2} to 0.214×10^{-3}).

From the numbers above we can verify that the two simulator prototypes scaled well, with a small increase in the time of execution when compared with the large increase in the size of the problems.

6. Conclusions

In this paper we presented our ongoing work on the implementation of scalable simulators specifically crafted for the MHM methods. We presented a preliminary evaluation of our two simulator prototypes, whose results are promising with regard to its scalability. These results also show that Erlang's overhead was small in comparison with its benefits in terms of productivity and extensive support for fault tolerance.

For future work, we will be investigating the use of coprocessors (GPGPUs and MICs) to reduce the time taken by the execution of the large amount of SOLVELO-CALPROBLEM primitive instances in the simulators. We also intend to experiment with other parallel sparse linear solvers than PaStiX to solve Equation 6 in the REDUCELOCALPROBLEMS primitive for the case of the MPI version. In particular, the hybrid approaches presented in [1,12] combine direct and iterative methods and seem fit to solve linear systems similar to the one illustrated in Equation (6).

7. Acknowledgement

References

[1] E. Agullo, L. Giraud, A. Guermouche, A. Haidar, and J. Roman. Parallel algebraic domain decomposition solver for the solution of augmented systems. *Advances in Engineering Software*, 6061:23 – 30, 2013. CIVIL-COMP: Parallel, Distributed, Grid and Cloud Computing.

[2] R. Araya, C. Harder, D. Paredes, and F. Valentin. Multiscale hybrid-mixed method. *SIAM J. Numer. Anal.*, 51(6):3505–3531, 2013.

[3] R. Araya, C. Harder, A. Poza, and F. Valentin. Multiscale hybrid-mixed method for the Stokes and Brinkman equations – the method. HAL 01347517, 2016.

[4] F. Cappelo, A. Geis, B. Gropp, L. Kale, B. Kramer, and M. and Sinir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[5] Yalchin Efendiev, Raytcho Lazarov, Minam Moon, and Ke Shi. A spectral multiscale hybridizable discontinuous galerkin method for second order elliptic problems. *Computer Methods in Applied Mechanics and Engineering*, 292:243 – 256, 2015. Special Issue on Advances in Simulations of Subsurface Flow and Transport (Honoring Professor Mary F. Wheeler).

[6] C. Harder, A.L. Madureira, and F. Valentin. A hybrid-mixed method for elasticity. *ESAIM: Math. Model. Num. Anal.*, 50(2):311–336, 2016.

[7] Christopher Harder, Diego Paredes, and Frédéric Valentin. A family of multiscale hybrid-mixed finite element methods for the darcy equation with rough coefficients. *Journal of Computational Physics*, 245:107 – 130, 2013.

[8] Christopher Harder, Diego Paredes, and Frédéric Valentin. On a multiscale hybrid-mixed method for advective-reactive dominated problems with heterogeneous coefficients. *Multiscale Modeling & Simulation*, 13(2):491–518, 2015.

[9] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.

[10] Thomas Y. Hou and Xiao-Hui Wu. A multiscale finite element method for elliptic problems in composite materials and porous media. *Journal of Computational Physics*, 134(1):169 – 189, 1997.

[11] Hang Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11:1–11:36, February 2015.

[12] Ichitaro Yamazaki, Xiaoye S. Li, Francois-Henry Rouet, and Bora Ucar. On partitioning and reordering problems in a hierarchically parallel hybrid linear solver. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 00(undefined):1391–1400, 2013.