Experimentos no uso do modelo de atores para simulações numéricas distribuídas baseadas em elementos finitos

Antônio Tadeu Azevedo Gomes[1], **Franklin Zillmer**[1]

[1] Laboratório Nacional de Computação Científica (LNCC)
25.651-075 – Petrópolis – RJ – Brasil
{atagomes,fzillmer}@lncc.br

Abstract. *The numerical simulation of complex phenomena through the use of finite element methods (FEM) is widely disseminated in industry and academy. This paper is interested in a specific family of FEM called multiscale hybrid mixed (MHM) methods, given its potential applicability in cloud computing environments. To better explore this potential, however, more flexibility and reconfigurability than those provided by traditional strategies—notably those based on the MPI standard—is needed when implementing the orchestration of the numerical processes in a distributed simulator. This paper aims at presenting and assessing objectively an innovative multi-language approach, based on the actor model, to develop distributed MHM-based simulators. The experimental results presented herein show, through strong and weak scalability measures, the feasibility of this approach when compared with a traditional single-language approach based on MPI.*

Resumo. *A simulação numérica de fenômenos complexos usando métodos de elementos finitos (MEFs) é bastante disseminada na indústria e academia. O presente artigo se interessa por uma família específica de métodos denominada MEFs multiescala hibridizados mistos (MHM), dado seu potencial de aplicação em ambientes de nuvens computacionais. Para melhor explorar esse potencial, no entanto, são necessárias estratégias de implementação mais flexíveis e reconfiguráveis para a orquestração dos processos numéricos em um simulador distribuído do que aquelas usualmente empregadas em implementações de MEFs – notadamente aquelas baseadas no padrão MPI de troca de mensagens. Este artigo se propõe a apresentar e avaliar objetivamente uma abordagem multilinguagem inovadora, baseada no modelo de atores, para o desenvolvimento de simuladores distribuídos baseados no MHM. Os resultados experimentais apresentados neste artigo demonstram, por meio de medidas de escalabilidade forte e fraca, a viabilidade dessa abordagem quando comparada a uma abordagem tradicional, mono-linguagem, baseada em MPI.*

1. Introdução

A simulação numérica de fenômenos complexos, tais como o escoamento de fluidos em materiais porosos e a propagação de ondas em nanoestruturas, provê meios efetivos para resolver problemas relevantes em diversas áreas do conhecimento. Nesse contexto, o uso de métodos de elementos finitos (MEFs) é bastante disseminado na indústria e academia para simular numericamente esses fenômenos [Ciarlet 1978, Hughes 1987, Pironneau 1989]. O fundamento dos MEFs é a discretização de um domínio contínuo,

com uma certa geometria, em uma malha de subdomínios discretos, usualmente chamados de elementos. Através dessa discretização em malha, é possível obter com MEFs soluções aproximadas do sistema de equações diferenciais representando o fênomeno de interesse no domínio em questão. Em última instância, MEFs obtêm a partir desse processo de discretização um sistema linear da forma $\mathbf{Ax} = \mathbf{b}$ (sendo $\mathbf{A}$ uma matriz, $\mathbf{x}$ e $\mathbf{b}$ vetores). Tal sistema é resolvido por métodos de álgebra linear computacional, como os métodos diretos de decomposição LU, LLt e LDLt, por exemplo.

Apesar dos MEFs clássicos serem apropriados para diferentes problemas, sua acurácia pode ser comprometida seriamente quando usados em problemas multiescala – problemas que envolvem fenômenos em diferentes escalas de espaço e tempo e fortemente inter-relacionados. A qualidade da aproximação da solução nos MEFs depende da granularidade da malha; quanto mais fina a malha, melhor a aproximação, contudo o sistema linear resultante e a demanda por recursos computacionais tanto de processamento como de memória serão maiores. Independente de qual método de álgebra linear computacional seja usado, a característica altamente acoplada desses métodos impõe dificuldades na paralelização especialmente em ambiente distribuído, pois há continua troca de mensagens nos processos envolvidos. A fim de mitigar tais dificuldades computacionais novos MEFs vêm sendo desenvolvidos [Hou and Wu 1997, Efendiev et al. 2015], entre eles destacamos uma família de métodos denominada MEFs multiescala hibridizados mistos (MHM) [Harder et al. 2013, Araya et al. 2013]. Do ponto de vista matemático, os métodos MHM incorporam naturalmente múltiplas escalas (chamados de 'níveis MHM') e provêm soluções com precisão de alta ordem em malhas grossas. Do ponto de vista computacional, o nível MHM inferior é formado por um conjunto de problemas completamente independentes – ditos 'locais' – que podem ser facilmente resolvidos em paralelo. As soluções independentes desses problemas locais (que podem ser obtidas, por exemplo, por meio da resolução direta de sistemas lineares menores) alimentam o nível MHM superior, onde um sistema linear dito 'global', de complexidade muito menor (comparativamente ao que se obteria de forma equivalente usando um MEF clássico), pode ser então resolvido e cuja solução, ao ser combinada com as soluções dos problemas locais, gera a aproximação da solução para o problema em questão.

Os métodos MHM se tornam particularmente atraentes de serem aplicados em ambientes de nuvens computacionais, dado que, diferentemente dos MEFs clássicos, no MHM a atuação da tecnologia de interconexão do ambiente de execução subjacente tem um impacto muito menor no desempenho global das simulações, além de que a comunicação entre os processos ocorre somente no momento em que os processos responsáveis pelos problemas locais enviam suas soluções ao processo responsável pelo problema global. A execução dessas simulações em nuvens traz oportunidades relacionadas sobretudo à flexibilidade e dinamicidade de alocação de recursos, quando comparada a execução dessas mesmas simulações em ambientes de supercomputação. Uma oportunidade advém do fato do problema global só poder ser resolvido após os problemas locais terem terminado, o que permitiria a um simulador baseado no MHM liberar recursos computacionais associados à resolução dos problemas locais antes do término de uma simulação. Além disso, em função da maior ou menor disponibilidade de recursos computacionais (por exemplo devido a questões financeiras), os problemas locais podem ser redistribuídos durante uma simulação entre mais ou menos processadores sem nenhum impacto na implementação dos algoritmos numéricos associados ao MHM no simulador.

Por fim, em cenários de simulações de larga escala, o envolvimento de um conjunto maior de processadores aumenta a possibilidade de falhas durante a execução das simulações, de modo que a flexibilidade e dinamicidade de alocação de recursos deve também poder ser explorada com o objetivo de conferir ao simulador maior tolerância a falhas no ambiente de execução subjacente.

Contudo, o desenvolvimento de um simulador com as características de flexibilidade e dinamicidade acima demanda um ferramental apropriado de implementação. Tipicamente, simuladores baseados em MEFs são implementados sobre bibliotecas de passagem de mensagens – notadamente aquelas baseadas no padrão MPI[1] – especificamente adaptadas para uso em ambientes de supercomputação. Essas bibliotecas, pelo padrão, não são dotadas de primitivas que ofereçam facilidades de ampla reconfiguração da topologia de troca de mensagens entre os processos constituientes de um simulador. Extensões ao padrão MPI têm sido propostas para dar esse tipo de suporte, com algumas implementações como por exemplo AMPI [Huang et al. 2003]. No entanto, tais extensões ainda apresentam uma série de desafios no que se refere à sua implantação em nuvens [Acun et al. 2014].

Este artigo se propõe a apresentar uma nova abordagem, multi-linguagem, para o desenvolvimento de simuladores baseados em MEFs que possam executar de forma eficiente e flexível em nuvens computacionais, tendo como foco inicial os métodos MHM. Como objetivo específico, este artigo explora o uso do modelo de atores [Agha 1985] presente na linguagem Erlang para desenvolver um simulador MHM que seja flexível e dinâmico no que se refere a alocação de recursos em nuvens. A despeito de suas qualidades como linguagem para desenvolvimento de sistemas distribuídos altamente escaláveis e tolerantes a falhas, Erlang não foi projetada para computação numérica e são de fato poucas as instâncias de uso de Erlang para esse tipo de cenário documentadas na literatura [Scalas et al. 2008]. Porém, sua qualidade de integração permite que módulos Erlang possam se comunicar com relativa facilidade com módulos de *software* desenvolvidos em outras linguagens. Essa estratégia é explorada no presente trabalho, sendo C++ a linguagem escolhida para a implementação dos algoritmos numéricos associados ao MHM.

Uma série de experimentos foram conduzidos e seus resultados são apresentados neste artigo com relação ao *overhead* decorrente do uso de Erlang na implementação do simulador MHM, quando executado sobre uma tecnologia de interconexão Gigabit Ethernet (típica de nuvens), em comparação com uma implementação tradicional, baseada em MPI, quando executada sobre uma tecnologia de interconexão Infiniband. Esses resultados demonstram que, conforme esperado, a execução baseada em MPI sobre Infiniband é mais rápida, mas com uma vantagem pouco substancial em relação a execução baseada em Erlang sobre Gigabit Ethernet, o que sinaliza a viabilidade do uso do modelo de atores combinado à execução em nuvens para resolver problemas complexos em áreas onde tradicionalmente os ambientes de supercomputação são predominantes.

O restante deste artigo está estruturado como se segue. Na Seção 2 a linguagem Erlang e o modelo de atores são apresentados. A Seção 3 faz uma apresentação sucinta dos principais aspectos computacionais relacionados à família de métodos MHM. A Seção 4 descreve a arquitetura do simulador MHM baseado no modelo de atores. A

[1] `http://mpi-forum.org`

Seção 5 apresenta os resultados experimentais que aferem a escalabilidade do simulador MHM baseado no modelo de atores comparativamente a uma implementação baseada em MPI. Finalmente, a Seção 6 resume as contribuições obtidas e potenciais trabalhos futuros.

2. Erlang e o modelo de atores

A forma tradicional de se implementar concorrência em uma linguagem de programação é por meio de entidades chamadas *threads*, que podem ser vistas como comportamentos (fluxos de execução de instruções) operando sobre regiões compartilhadas de memória. O compartilhamento de memória entre *threads* e a consequente possibilidade de condições de corrida leva a necessidade de uso de semáforos, que por sua vez podem levar a outros problemas relacionados a concorrência, como *deadlocks*.

O modelo de atores [Agha 1985] propõe uma abordagem distinta para atacar problemas resultantes de concorrência. Nesse modelo, entidades chamadas atores também encapsulam comportamentos, mas estes não compartilham memória. Cada ator tem uma fila de mensagens que é usada para trocar mensagens com outros atores. Ao receber uma mensagem, um comportamento é disparado no ator, podendo ocasionar: (i) o envio de mensagens a outros atores; (ii) a criação de novos atores; ou (iii) a mudança no comportamento do ator para próximas mensagens a serem recebidas. A comunicação entre atores é assíncrona e não se pressupõe no modelo a ordenação ou mesmo garantia de entrega de mensagens. Disso decorre que sistemas baseados em atores precisam saber lidar explicitamente com reconfigurações e falhas, que são duas das principais características da linguagem Erlang.

Atores são denominados 'processos' em Erlang. Essa linguagem oferece recursos para criação e gerenciamento desses processos com o objetivo de simplificar a programação concorrente. Os processos Erlang geralmente seguem um padrão comum: uma vez instanciados eles chamam uma função recursiva para processar e produzir dados. A função recursiva geralmente segue a seguinte sequência de operações: (i) receber uma mensagem de um processo, (ii) tratar a mensagem, (iii) atualizar o estado do processo, e (iv) passar o estado atualizado para uma nova chamada da função através de sua chamada recursiva. Finalmente, uma mensagem de parada especial é geralmente definida para permitir que o processo receptor possa ser encerrado normalmente.

Erlang é capaz de compilar em um formato intermediário (*bytecodes*) e pode ser executada em máquinas virtuais (nós Erlang). Normalmente, cada nó Erlang é executado em um *host* diferente (p.ex. uma máquina física em um *cluster* ou uma máquina virtual em uma nuvem).

Erlang é uma linguagem com estrutura funcional, projetada para facilitar a implementação de *software* distribuído devido a existência de bibliotecas com estruturas configuráveis, incluindo um sistema de gerenciamento de banco de dados embarcado eficiente e tolerante a falhas, particularmente útil para implementação de estratégias de *checkpointing* de aplicações. Outras bibliotecas possuem mecanismos de manipulação de erros e de monitoramento de exceções sendo conhecidas como OTP (*Open Telecom Platform*) e oferecem uma série de comportamentos (*behaviors*) genéricos, tais como servidores (`gen_server`), máquinas de estados finitos (`gen_fsm`), manipuladores de eventos entre outros.

Um desses comportamentos, denominado supervisor (`gen_supervisor`), tem como principal tarefa gerenciar, monitorar e, eventualmente, reiniciar processos em caso de falha. Os supervisores podem gerenciar qualquer tipo de processo em Erlang, incluindo outros supervisores. Esta organização cria uma estrutura hierárquica chamada árvore de supervisão. Finalmente, uma aplicação Erlang é um tipo especial de comportamento que permite a toda uma árvore de supervisão ser iniciada e interrompida como uma unidade. Para fornecer tolerância a falhas de *hardware*, uma aplicação Erlang pode ser controlada de maneira distribuída, através do emprego de dois ou mais nós Erlang executados em *hosts* diferentes. Este modo de organização configura uma aplicação distribuída. Se o nó Erlang onde uma aplicação distribuída executa falhar, por exemplo, devido a uma falha de rede ou de máquina, a aplicação pode ser reiniciada em outro nó Erlang.

Erlang oferece ainda um sistema de gerenciamento de banco de dados (SGBD) embarcado chamado *Mnesia*. Dentre as características do *Mnesia*, a tolerância a falhas e a possibilidade de manutenção dos dados geridos pelo SGBD em memória (para fins de desempenho) são duas das mais importantes, sendo largamente empregadas neste trabalho. Conforme [Mattsson et al. 1999], *Mnesia* tem a capacidade de replicar uma tabela entre vários nós Erlang. Todas as réplicas são iguais e não há nenhum conceito de cópia primária e de salvaguarda. Se uma tabela é replicada todas as operações de escrita são aplicadas a todas as réplicas para cada transação. Se houver falha em alguma das réplicas as operações de escrita ainda serão bem-sucedidas e os dados destas réplicas serão posteriormente atualizados.

3. A família de métodos numéricos MHM

Os métodos MHM são adaptados à resolução de modelos com múltiplas escalas ou com grandes contrastes, caracterizando-se por alta ordem de precisão (baixas taxas de erro) e por incorporar a granularidade típica de ambientes de execução massivamente paralelos. Para tanto, dada uma malha (grossa) $\mathcal{T}_H$ que discretiza o domínio em um conjunto de elementos $\{K_t\}_{t\in T}$, com $T \subset Z^+ = \{0, \ldots, N_t - 1\}$, o método MHM é composto de uma formulação global definida no esqueleto $\mathcal{E}_H$ da malha (que também é discretizada em um conjunto de faces $\{F_e\}_{e\in E}$, com $E \subset Z^+ = \{0, \ldots, N_e - 1\}$), e de uma coleção de problemas locais definidos para cada elemento K_t da malha, problemas estes dirigidos pelos dados do problema original e por um processo chamado de hibridização, que relaxa a continuidade da solução no esqueleto da malha. A Figura 1 ilustra esses conceitos de discretização da malha $\mathcal{T}_H$ e de seu esqueleto $\mathcal{E}_H$ no $\mathbb{R}^2$.

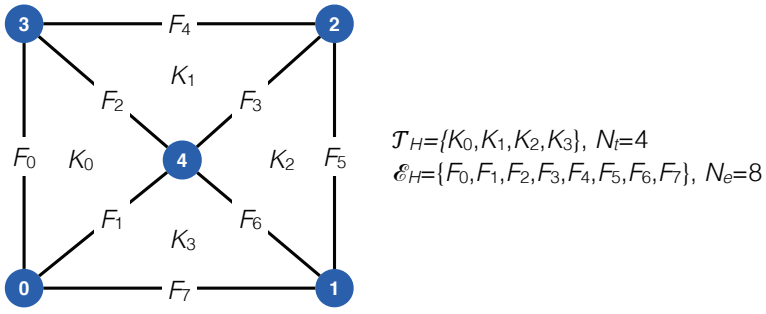


Figura 1. Exemplo de malha no $\mathbb{R}^2$.

A aproximação de solução entregue pelos métodos MHM tem a forma usual

$$u_H := u_0 + \sum_{K_t \in \mathcal{T}_H} \Big[\sum_{i=0}^{N_I - 1} k_i \eta_t^i + \eta_t^f \Big], \tag{1}$$

onde u_H é a solução aproximada do problema (o índice H refere-se ao nível de refinamento da malha grossa), u_0 e k_i são computados no problema global, e η_t^i e η_t^f são computados nos problemas locais. N_I é diretamente proporcional ao número de faces de um elemento da malha. Por questões de espaço e também de enfoque, outros detalhes da formulação matemática dos métodos MHM são omitidos neste artigo. O leitor interessado pode se remeter as referências [Harder et al. 2013] e [Araya et al. 2013]. Para um melhor entendimento dos aspectos computacionais desses métodos, é apresentado na Listagem 1 o pseudocódigo para o algoritmo numérico de um simulador baseado nos métodos MHM.

Listagem 1 Pseudocódigo para o algoritmo numérico de um simulador MHM.

```
Require: T_H = {K_t}_{t∈T}                          ▷ Malha com elementos indexados
  P_{T_H} ← SPLITPROBLEM(T_H)                       ▷ P_{T_H} := {(K_t, {F_t})}_{t∈T}
  S_{T_H} ← ∅
  for all p_t ∈ P_{T_H} do                          ▷ p_t := (K_t, {F_t})
      s_t ← SOLVELOCALPROBLEM(p_t)                  ▷ s_t := (η_t^f, {η_t^i}_{i∈{0,...,N_I−1}})
      S_{T_H} ← S_{T_H} ∪ s_t
  end for
  (u_0, k_i) ← REDUCELOCALPROBLEMS(S_{T_H})
  u_H ← COMPUTESOLUTION(u_0, k_i, S_{T_H})
```

Um simulador MHM é construído sobre um conjunto de primitivas que são diretamente mapeadas nos blocos matemáticos principais dos métodos MHM: SPLITPROBLEM, SOLVELOCALPROBLEM, REDUCELOCALPROBLEMS, e COMPUTESOLUTION.

A primitiva SPLITPROBLEM divide o problema original definido sobre uma malha (grossa) $\mathcal{T}_H$ em um problema global (o primeiro nível MHM) e um conjunto de problemas locais (o segundo nível MHM), um para cada elemento K_t da malha. Essa primitiva retorna um conjunto $P_{\mathcal{T}_H}$ de pares $(K_t, \{F_t\})$, onde $\{F_t\}$ representa o conjunto de todas as faces de K_t. (Todas as faces de todos os elementos, em conjunto, formam o esqueleto $\mathcal{E}_H$ da malha.) A primitiva SOLVELOCALPROBLEM computa as contribuições η_t^f e $\{\eta_t^i\}_{i\in\{0,\ldots,N_I-1\}}$ de um elemento específico K_t na malha, a serem usadas na montagem do problema global. A primitiva REDUCELOCALPROBLEMS combina as contribuições η_t^f e $\{\eta_t^i\}_{i\in\{0,\ldots,N_I-1\}}$ dos problemas locais para computar u_0 e k_i na Equação (1). Finalmente, na primitiva COMPUTESOLUTION, a aproximação u_H definida pela Equação (1) é computada.

4. Arquitetura do simulador MHM baseado no modelo de atores

A Figura 2 mostra como foi estruturada e criada a arquitetura do simulador MHM distribuído baseado no modelo de atores. Em nossa implementação empregou-se a arquitetura mestre-escravo referente ao modelo de comunicação adotado para os atores Erlang. Isso não limita, a princípio, o uso de estilos arquiteturais diversos induzidos pelos métodos numéricos. No caso do MHM, por exemplo, o estilo em questão é o *fan-out/fan-in*.

O processo `main_supervisor` implementa o comportamento `supervisor`. Esse processo gerencia os processos `main_server`, `regis_server`, e `fsm_supervisor`, sendo todos disparados na inicialização da aplicação. Todos esses processos executam em um mesmo nó Erlang; se o nó se torna indisponível, esses processos são reiniciados em outro nó Erlang com seus respectivos estados preservados graças ao uso do *Mnesia*. Esta é a primeira característica que permite ao simulador MHM ter suporte de tolerância a falhas.

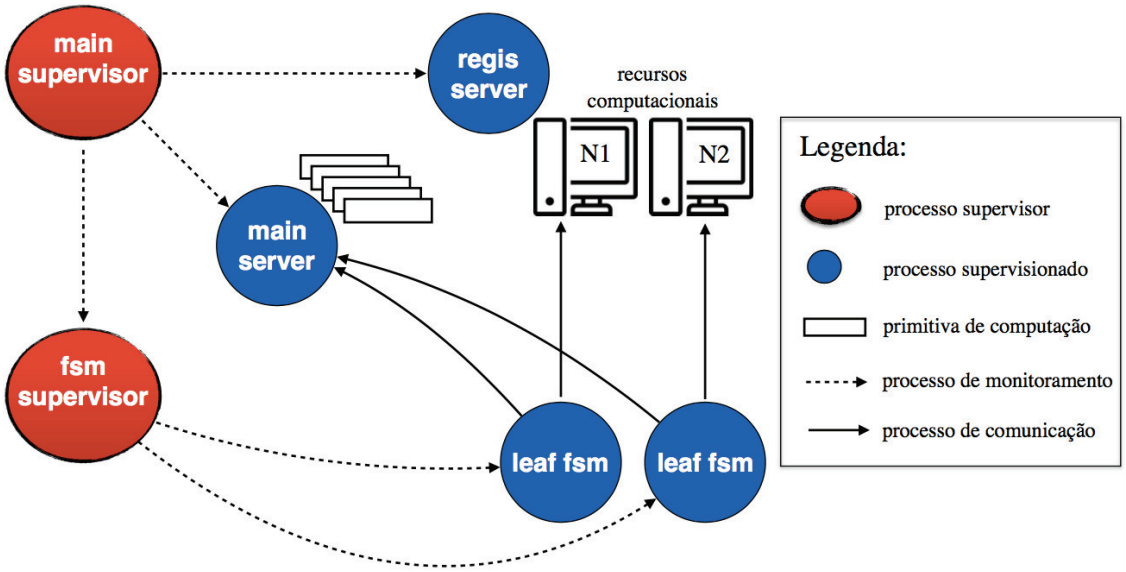


Figura 2. Arquitetura do simulador

O processo `main_server` mantém um conjunto de tarefas computacionais, que são dinamicamente alocadas nos recursos computacionais disponíveis. Cada tarefa é relacionada a um problema local ou global do MHM, e tem um estado associado – "Not Solved", "Being Solved", "Being Reduced" (para problemas globais somente) ou "Solved". Além disso, cada tarefa é mapeada em uma instância de uma das primitivas descritas na Seção 3. Essas primitivas são implementadas em C++ e executam em recursos computacionais geridos pelo `regis_server`, que implementa o comportamento `gen_server`. O `regis_server` coleta informações de status dos recursos computacionais no ambiente de execução subjacente (como máquinas virtuais em uma nuvem) e, para cada recurso, solicita ao `fsm_supervisor` (outra implementação do comportamento `supervisor`) a criação/destruição de processos 'escravos' associados ao recurso toda vez que o mesmo se torna disponível/indisponível. Esta é a segunda característica que permite ao simulador MHM ter suporte de tolerância a falhas.

Os processos escravos (`leaf_fsm`) implementam o comportamento `gen_fsm`. Eles executam no mesmo nó Erlang que os demais processos da aplicação Erlang. Uma vez iniciado, um `leaf_fsm` continuamente solicita ao `main_server` novas tarefas computacionais. O `main_server` responde à requisição do `leaf_fsm`: (i) selecionando uma tarefa apropriada, (ii) mudando o estado da tarefa para "Being Solved"/"Being Reduced", e (iii) respondendo ao `leaf_fsm` com a primitiva MHM a ser executada. O `leaf_fsm` despacha então a execução da primitiva no recurso computacional a ele associado, e monitora essa execução. Quando o `leaf_fsm` detecta que a primitiva terminou,

ele envia os resultados da primitiva para o `main_server`, que pode mudar o estado da tarefa ou criar novas tarefas. Se o `leaf_fsm` detecta que a primitiva não completou por causa de uma falha no recurso computacional ou de uma partição de rede, ele notifica o `main_server` e o `regis_server`, que podem então realocar dinamicamente a tarefa para um recurso diferente.

A Figura 3 ilustra como o estado das tarefas evolui no simulador. O simulador inicia com uma única tarefa representando o problema global – $\mathcal{T}_H$ na Listagem 1 – sendo publicada no `main_server` com o estado "Not Solved". Nesse ponto, um único `leaf_fsm` será capaz de pegar essa tarefa, o que dispara o despacho da primitiva SPLITPROBLEM no recurso computacional associado a esse `leaf_fsm` e a mudança de seu estado para "Being Solved". O resultado dessa primitiva – $P_{\mathcal{T}_H}$ na Listagem 1 – é tratado pelo `leaf_fsm` como um conjunto de requisições – uma para cada $p_t \in P_{\mathcal{T}_H}$ – para a criação de novas tarefas no `main_server`. Cada uma dessas novas tarefas é relacionada a um problema local diferente e adicionada ao `main_server` com o estado "Not Solved". Para simplificar a ilustração da evolução desses estados, na Figura 3 só são apresentados dois problemas locais derivados da primitiva SPLITPROBLEM.

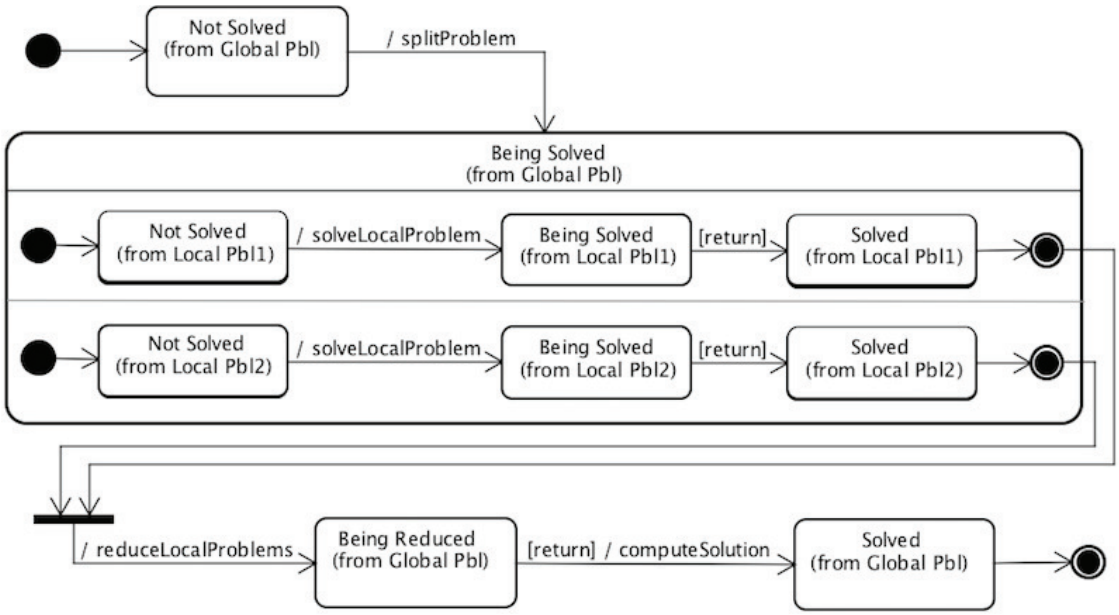


Figura 3. Evolução das tarefas.

A partir desse ponto cada `leaf_fsm` passa a obter diferentes tarefas relacionadas aos problemas locais, mudando seus estados para "Being Solved" e despachando a execução da primitiva SOLVELOCALPROBLEM em seu recurso computacional correspondente. Para computar η_t^f e $\{\eta_t^i\}_{i\in\{0,\ldots,N_I-1\}}$ em cada problema local, é usado um algoritmo sequencial de decomposição LU oferecido pela biblioteca Eigen,[2] de modo que o recurso computacional associado a um `leaf_fsm` possa resolver em paralelo tantos problemas locais quanto a quantidade de *cores* disponíveis nesse recurso. Quando essas invocações da primitiva SOLVELOCALPROBLEM são completadas, cada `leaf_fsm` envia seu conjunto de resultados – $s_t \in S_{\mathcal{T}_H}$ na Listagem 1 – para o `main_server`, que modifica o

[2] `http://eigen.tuxfamily.org`

estado das tarefas correspondentes para “Solved”. Cada `leaf_fsm` então prossegue sua execução obtendo outras tarefas e as despachando para seu recurso até que não haja mais tarefas no estado “Not Solved”.

Quando todos os problemas locais estão no estado “Solved”, a próxima requisição de um `leaf_fsm` é respondida pelo `main_server` com um primitiva REDUCELOCALPROBLEMS, e o estado associado ao problema global é modificado de “Being Solved” para “Being Reduced”. Novamente, é empregada a decomposição LU nesta primitiva para resolver o sistema linear associado ao problema global. No caso da primitiva REDUCELOCALPROBLEMS, no entanto, é usada uma versão paralela, de memória compartilhada, oferecida pela suíte Pardiso,[3] uma vez que há um único sistema linear a ser resolvido no problema global. O resultado obtido a partir da primitiva REDUCELOCALPROBLEMS dispara a execução imediata da primitiva COMPUTESOLUTION e a mudança do estado da tarefa associada ao problema global para “Solved” no `main_server`. Após isso, a simulação é encerrada. A Figura 3 não contempla estados excepcionais pois estes são implicitamente tratados pelo mecanismo de tolerância a falhas de Erlang, inclusive o caso de parada total.

5. Experimentos

Para avaliar de forma comparativa o uso de Erlang na implementação do simulador MHM, foi implementada uma segunda versão do simulador MHM, chamada neste artigo de ‘implementação de referência’, empregando somente C++ e o padrão MPI. A Figura 4 ilustra os processos principais que compõem a implementação de referência. Nessa implementação, os R processos MPI que a compõem são onipotentes, em contraste com a versão Erlang, que diferencia mestre e escravos. Cada processo MPI $r \in R$ é responsável por executar parcialmente a primitiva SPLITPROBLEM sobre uma partição reduzida $\mathcal{T}_H^r$ de $\mathcal{T}_H$. Esse particionamento configura um escalonamento estático dos processos MPI no que se refere à solução dos problemas locais pela primitiva SOLVELOCALPROBLEM, escalonamento este que é obrigatoriamente mantido até o fim da simulação. Como observado na Figura 4, a implementação de referência é estruturalmente mais simples do que a versão Erlang, o que se deve sobretudo à ausência de um esquema de escalonamento dinâmico dos processos bem como de quaisquer mecanismos de tolerância a falhas.

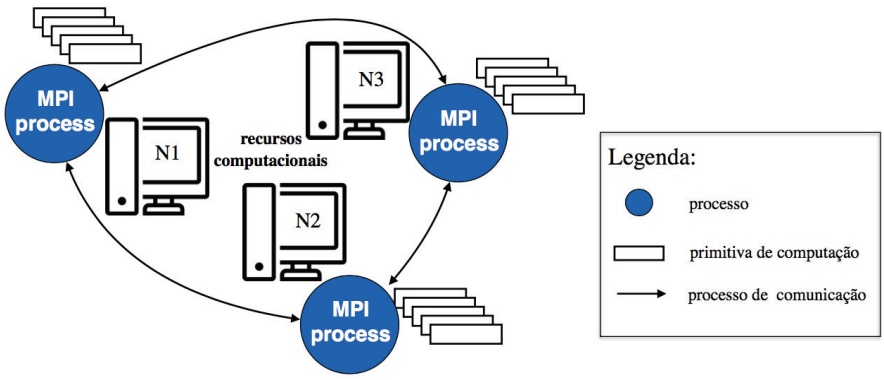


Figura 4. Arquitetura da implementação de referência.

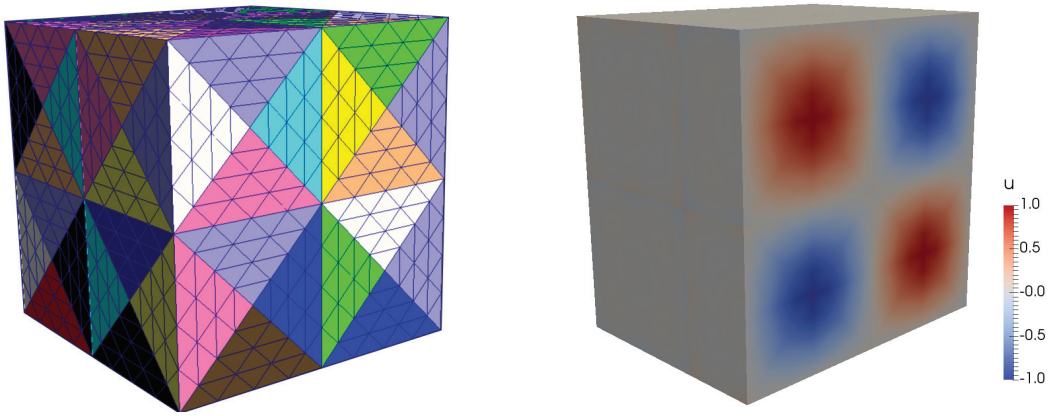
[3] `http://www.pardiso-project.org`

Para este artigo foram coletadas medidas de desempenho relacionadas a escalabilidade forte e fraca do simulador MHM baseado em Erlang e da implementação de referência com MPI. Para essas medidas, as duas versões foram executadas no supercomputador Santos Dumont (SDumont) do LNCC, que possui uma arquitetura de *cluster* com as seguintes configurações por nó de processamento: 2x CPU Intel Xeon E5-2695v2 Ivy Bridge (12 *cores* por CPU), 2.4GHZ, 64GB DDR3 RAM. Os nós do *cluster* são interconectados por meio de uma rede Infiniband FDR, com topologia *fat-tree*, e compartilham um sistema de arquivos distribuídos baseado no Lustre v2.1. As razões pelas quais foi utilizado o SDumont ao invés de uma nuvem para as simulações com a versão Erlang do simulador incluem a disponibilidade de um número maior de *cores* que viabilizassem as medidas de escalabilidade e o uso de um mesmo hardware e de uma mesma suíte de *software* para compilação e execução das duas versões do simulador. Para imitar os efeitos do uso de uma rede de interconexão mais típica de nuvens no caso da versão Erlang, para as simulações com esta versão do simulador foi utilizada a rede de interconexão 10 Gigabit Ethernet do SDumont, que é usualmente empregada para procedimentos de acesso remoto e de administração do *cluster*.

O código C++ foi compilado com o compilador Intel icpc versão 16.0.2 *build* 20160204, com as *flags* de otimização '$-O2$' e '$-ipo$'. Foram usadas a biblioteca Eigen versão 3.3-rc1 e a implementação do resolvedor de sistemas lineares Pardiso presente na biblioteca MKL da suíte Intel's Parallel Studio XE 2016.2.062. O código Erlang foi executado com o sistema aberto Erlang/OTP da Ericsson, versão 17 erts-6.1, com a compilação em código nativo HiPE (*High Performance Erlang*) habilitada.

Para as simulações, foi usado um modelo de fluxo difusivo (equação de Darcy: $\kappa \nabla u(x, y, z) = f(x, y, z)$) como problema e um cubo unitário como domínio do problema, a ser discretizado em uma malha grossa. O coeficiente de difusão na equação foi setado como $\kappa = 1$ e seu termo de fonte como $f(x, y, z) = 12\pi^2 \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$, e as condições de fronteira do domínio foram estabelecidas como $u = 0$ (Dirichlet homogêneo). A solução exata para esse problema é $u(x, y, z) = \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$. As simulações foram setadas com 24, 192 e 1.536 elementos na malha de tetraedros que discretiza o domínio. Em cada um desses elementos uma submalha com 59 ou 6.046 tetraedros é gerada em tempo de execução como parte das operações efetuadas pela primitiva SOLVELOCALPROBLEM. A malha de 192 tetraedros com 59 tetraedros por submalha é ilustrada na Figura 5(a), e a aproximação numérica correspondente ao modelo escolhido sobre esta mesma malha é ilustrada na Figura 5(b).

A configuração descrita acima gera 6 parametrizações diferentes para o simulador MHM, mas só foram experimentadas 5 delas para os propósitos deste artigo, uma vez que a configuração com 1.536×59 tetraedros gera problemas pequenos e sujeitos a variações estatísticas importantes devido ao ambiente de execução. As 5 parametrizações apresentadas foram testadas em 3 situações de disponibilidade de recursos distintas: 24, 192 e 768 *cores*. O valor de 24 *cores* corresponde ao uso de um único nó do SDumont, sendo usado como *baseline* de comparação. O valor de 192 *cores* equivale a 8 nós do SDumont, e esse número de *cores* foi utilizado para mostrar a escalabilidade fraca comparando as malhas grossas de 1.536 e 12.288 tetraedros, com mesmo nível de refinamento nas malhas locais. O valor de 768 *cores* corresponde ao uso de 32 nós do SDumont, número máximo



(a) Malha grossa com 192 elementos, cada um deles de uma cor diferente e incluindo suas respectivas submalhas de 59 elementos cada, num total de 724.992 elementos.

(b) Isolinhas da aproximação numérica obtida com MHM para $x = 0{,}75$. Note que nas fronteiras do domínio o valor da aproximação para u é zero, como estabelecido pela condição de fronteira de Dirichlet homogêneo.

Figura 5. Exemplo de modelo usado nas simulações.

possível de se alocar neste *cluster* no momento dos experimentos.

Para cada uma das 15 configurações, foram feitas 3 rodadas de simulação e escolhida, para cada configuração, a rodada com o menor tempo total de execução para apresentação neste artigo. (É importante destacar que não houve variação estatística significativa entre rodadas de uma mesma configuração.) No caso da situação de disponibilidade de 24 *cores*, como a simulação ocupa somente um nó do SDumont, foi usada uma versão mais simples do simulador MHM que usa somente *multithreading*, sem nenhum envolvimento de Erlang ou MPI nas execuções. O propósito dessa simplificação foi capturar melhor o impacto da distribuição dos processos nas versões Erlang e MPI do simulador MHM.

A Tabela 1 consolida todas as medidas de escalabilidade obtidas nos experimentos. Analisando a escalabilidade forte – isto é, o quão eficiente é o simulador em ter reduzido seu tempo de execução com um aumento na quantidade de recursos disponíveis para o mesmo –, pode-se observar na tabela que a eficiência de ambos os simuladores melhora à medida que os problemas ficam maiores. Certamente são necessárias medidas de desempenho em cenários com maior quantidade de recursos disponíveis para confirmar essa tendência, mas de toda forma tratam-se de resultados promissores: à medida que os problemas crescem em tamanho, a razão entre processamento e comunicação cresce consideravelmente nos dois simuladores em função das características de baixo acoplamento entre as fases local e global dos métodos MHM. Outra observação é que a versão Erlang do simulador é menos eficiente do que a versão MPI – o que era esperado –, mas por uma margem pequena. Esse é um outro resultado interessante porque demonstra a viabilidade de se executar simulações com os métodos MHM em ambientes de nuvem e, de forma mais geral, de se utilizar Erlang como linguagem de coordenação para aplicações científicas com necessidades de tolerância a falhas e estruturas de execução semelhantes à induzida pelos métodos MHM.

Tabela 1. Medidas de escalabilidade.

		Baseline	MPI		Erlang	
Número de tetraedros		24 *cores*	192 *cores*	768 *cores*	192 *cores*	768 *cores*
$12.288 \times 59 = 724.992$						
	Tempo (seg)	$\mathbf{129{,}81}^*$	57,74	72,88	97,00	97,00
	Eficiência	100,00%	28,10%	5,57%	16,62%	4,16%
$1.536 \times 6.046 = 9.286.656$						
	Tempo (seg)	489,00	$\mathbf{82{,}57}^*$	51,72	$\mathbf{108{,}00}^*$	65,00
	Eficiência	100,00%	74,03%	29,54%	56,60%	23,51%
$12.288 \times 6.046 = 74.293.248$						
	Tempo (seg)	4.753,00	641,61	$\mathbf{220{,}20}^*$	667,00	$\mathbf{240{,}00}^*$
	Eficiência	100,00%	92,60%	67,45%	89,70%	61,89%

As entradas na Tabela 1 que estão em negrito e demarcadas com um '$*$' também podem ser usadas para uma análise das duas versões do simulador MHM no que se refere à sua escalabilidade fraca – isto é, o quão eficiente é o simulador em manter o seu tempo de execução constante à medida que o tamanho dos problemas e a capacidade computacional disponível para resolvê-los aumentam numa mesma razão. Entre a primeira e terceira configurações, usando a versão Erlang, houve um aumento no tempo de execução por um fator de $\approx 1{,}84$. Para a versão MPI, esse aumento foi de um fator de $\approx 1{,}69$. Esses fatores servem de base para medir a eficiência de escalabilidade fraca dos simuladores, tomando o número total de tetraedros em cada configuração como tamanho do problema:[4] da primeira para a terceira configuração há um aumento em $\approx 102{,}47$ vezes o número de tetraedros, contra um aumento em apenas 32 vezes o número de *cores* utilizados. Percebe-se, portanto, que as duas versões de simuladores MHM escalam de forma excepcional, com um aumento muito pequeno no tempo total de execução quando comparado a um aumento considerável no tamanho dos problemas, proporcionalmente ao aumento na quantidade de recursos computacionais envolvidos.

6. Conclusões

Este artigo apresentou uma nova abordagem para o desenvolvimento de simuladores baseados em uma família específica de métodos numéricos baseados em elementos finitos, chamada MHM. Nessa abordagem, a linguagem Erlang é usada na implementação de um algoritmo de coordenação, baseado no modelo de atores, de um conjunto de processos responsáveis pela execução distribuída do algoritmo numérico induzido pelo MHM. Os resultados experimentais apresentados neste artigo demonstram a viabilidade da solução proposta e, em particular, a possibilidade de se executar de forma eficiente métodos numéricos baseados em elementos finitos em nuvens.

O fato de Erlang não ser adequada para a computação numérica e da solução adotada neste trabalho para mitigar essa limitação envolver a integração com outras linguagens mais apropriadas para esse tipo de computação abre oportunidades para que outras famílias de métodos numéricos se beneficiem de seus atributos de qualidade. Como perspectiva de pesquisa futura, temos intenção de mostrar que outros

[4]Outras formas de medir o tamanho do problema incluem a quantidade total de variáveis a serem computadas e a qualidade da aproximação em termos de um conjunto específico de normas de erro, mas como a apresentação dessas medidas envolve uma apresentação mais detalhada dos aspectos matemáticos dos métodos MHM, essas outras formas são omitidas no presente artigo

métodos com características distintas do MHM também podem se beneficiar do modelo de atores. Em particular, estamos no momento investigando um outro método também desenvolvido no LNCC, baseado em volumes finitos e com maior nível de acoplamento [Müller et al. 2016], o que induz simuladores com estilo arquitetural distinto do *fan-out/fan-in* presente no MHM. Em seguida, pretendemos definir um ou mais *behaviors* Erlang suficientemente genéricos, desenhados na forma de um *framework* de aplicação, para atender demandas de diferentes estilos arquiteturais induzidos por diferentes métodos numéricos.

Agradecimentos

A pesquisa apresentada neste artigo foi parcialmente financiada pelo *EU H2020 Programme* e pela RNP por meio do projeto HPC4E (`http://www.hpc4e.eu`). Os experimentos descritos neste artigo empregaram os recursos computacionais providos pelo supercomputador SDumont (`http://sdumont.lncc.br`).

Os autores agradecem a Weslley Pereira e Frédéric Valentin do LNCC, e a Diego Paredes da PUC Valparaiso, Chile, pela revisão de versões anteriores do conteúdo presente neste artigo.

Referências

Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., et al. (2014). Parallel programming with migratable objects: Charm++ in practice. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 647–658. IEEE.

Agha, G. A. (1985). Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document.

Araya, R., Harder, C., Paredes, D., and Valentin, F. (2013). Multiscale hybrid-mixed method. *SIAM Journal on Numerical Analysis*, 51(6):3505–3531.

Ciarlet, P. (1978). *The finite element method for elliptic problems*. North-Holland.

Efendiev, Y., Lazarov, R., Moon, M., and Shi, K. (2015). A spectral multiscale hybridizable discontinuous Galerkin method for second order elliptic problems. *Computer Methods in Applied Mechanics and Engineering*, 292:243 – 256. Special Issue on Advances in Simulations of Subsurface Flow and Transport (Honoring Professor Mary F. Wheeler).

Harder, C., Paredes, D., and Valentin, F. (2013). A family of multiscale hybrid-mixed finite element methods for the Darcy equation with rough coefficients. *Journal of Computational Physics*, 245:107–130.

Hou, T. Y. and Wu, X.-H. (1997). A multiscale finite element method for elliptic problems in composite materials and porous media. *Journal of Computational Physics*, 134(1):169 – 189.

Huang, C., Lawlor, O., and Kalé, L. V. (2003). Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas.

Hughes, T. J. R. (1987). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey.

Mattsson, H., Nilsson, H., and Wikström, C. (1999). Mnesia—a distributed robust DBMS for telecommunications applications. In *Practical Aspects of Declarative Languages*, pages 152–163. Springer.

Müller, L. O., Blanco, P. J., Watanabe, S. M., and Feijóo, R. A. (2016). A high-order local time stepping finite volume solver for one-dimensional blood flow simulations: application to the ADAN model. *International Journal for Numerical Methods in Biomedical Engineering*, 32(10):e02761–n/a. e02761 cnm.2761.

Pironneau, O. (1989). *Finite Element Methods for Fluids*. John Wiley,New York.

Scalas, A., Casu, G., and Pili, P. (2008). High-performance technical computing with Erlang. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 49–60. ACM.