Avaliação de desempenho de um simulador numérico distribuído e tolerante a falhas baseado no modelo de atores

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

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

Abstract. *This paper presents and evaluates a multi-language approach based on the actor model to the development of fault-tolerant distributed simulators based on finite volume methods (FVMs). A study case is conducted with a FVM that has been employed for the simulation of blood flows and that has looser coupling than classical FVMs. The experimental results presented in this paper suggest the feasibility of this approach when compared with a traditional, single-language approach based on MPI, in scenarios of higher ratios of computation to communication. Besides, these results show the effectiveness of the implemented fault-recovery strategies.*

Resumo. *Este artigo apresenta e avalia uma abordagem multi-linguagem baseada no modelo de atores para o desenvolvimento de simuladores distribuídos tolerantes a falhas baseados em métodos de volumes finitos (MVFs). É usado como estudo de caso um MVF empregado na simulação de fluxos sanguíneos e que apresenta propriedades de menor acoplamento que MVFs clássicos. Os resultados experimentais preliminares com esse estudo de caso sugerem a viabilidade dessa abordagem quando comparada a uma abordagem tradicional mono-linguagem, baseada em MPI, em cenários com maior razão computação/comunicação. Além disso, tais resultados mostram a eficácia das estratégias de recuperação de falhas implementadas.*

1. Introdução

A simulação numérica é utilizada para entender fenômenos complexos, sendo amplamente utilizada em vários campos da engenharia, como por exemplo em problemas de transferência de calor e massa. Nesse cenário é bastante difundido o emprego de métodos de volumes finitos (MVFs). MVFs são métodos de discretização que podem ser usados em geometrias arbitrárias e levam a esquemas numéricos robustos. Uma característica dos MVFs segundo [Eymard et al. 2000] é a conservação local dos fluxos, tornando o método bastante atraente quando se modelam problemas de escoamento de fluidos, como por exemplo no caso do fluxo sanguíneo pelo sistema vascular [Blanco et al. 2014].

HO-FV-LTS (*High-Order Finite Volumes with Local Time Stepping*) [Müller et al. 2016] é um MVF desenvolvido para simulações de fluxo sanguíneo. Esse método propõe um esquema numérico para a solução aproximada de equações diferenciais parciais hiperbólicas em redes de domínios unidimensionais. Esse método é empregado no modelo ADAN (*Anatomically Detailed Arterial Network*) [Blanco et al. 2015], um modelo que incorpora quase todas as artérias reconhecidas pelo

campo da anatomia humana, resultando em uma rede de mais de 2.000 vasos com uma vasta gama de escalas espaciais incluídas.

O esquema numérico para o método HO-FV-LTS considera detalhes anatômicos da rede venosa ou arterial sendo modelada, bem como as características biológicas e mecânicas do sistema vascular. Dependendo do quão realistas precisam ser as propriedades mecânicas dos vasos (p.ex. considerando vasos elásticos ou viscoelásticos), uma formulação matemática mais sofisticada é necessária. Isso acarreta um maior esforço computacional para se obter aproximações numéricas precisas. Nesse sentido, um simulador – implementado em C++ com paralelismo inter-nó usando MPI e intra-nó usando OpenMP – é apresentado em [Müller et al. 2016] com o objetivo de ser usado em máquinas paralelas de memória distribuída, como clusters de HPC (*High Performance Computing*).

As características funcionais do código do simulador HO-FV-LTS são de acoplamento relativamente baixo entre os processos quando comparado a códigos baseados em MVFs clássicos (devido principalmente ao uso de passos de tempo locais), o que permite a exploração de tecnologias alternativas no que se refere à distribuição e comunicação de processos. Em particular, a exploração de aspectos como dinamicidade na alocação de recursos e suporte para tolerância a falhas são pontos importantes e que podem viabilizar o uso desse simulador em ambientes paralelos alternativos a clusters de HPC, como as nuvens computacionais. É interessante destacar que diversos ambientes comerciais de nuvem têm evoluído no sentido de oferecer melhor suporte a aplicações tradicionalmente executadas em ambientes de HPC – como por exemplo a Amazon[1] e a Microsoft Azure.[2] Pelo fato de oferecerem recursos diferenciados, em particular no nível de interconexão, esses ambientes têm custo também diferenciado. Assim, surge nesse contexto também o interesse econômico de se redesenhar aplicações para que possam explorar de forma mais eficiente nuvens não necessariamente customizadas para HPC.

Neste artigo propõe-se avaliar uma abordagem multi-linguagem para o desenvolvimento de simuladores baseados em esquemas numéricos pouco acoplados, como o do método HO-FV-LTS. Mais especificamente, este artigo experimenta a adoção do modelo de programação baseado em atores [Agha 1985] – por meio da linguagem Erlang – na coordenação da execução paralela de primitivas de computação – implementadas em C++ com paralelismo intra-nó usando OpenMP – derivadas do simulador original do HO-FV-LTS. Erlang foi escolhida devido a suas características de produtividade, escalabilidade e integração eficiente com C/C++.

Os resultados da experimentação enfocam primeiramente o *overhead* do uso de Erlang na implementação das ações de coordenação do simulador, quando comparada à implementação original baseada em MPI do simulador. Esses resultados mostram que, conforme esperado, o simulador baseado em MPI tem tempos de execução consistentemente menores que os do simulador baseado em Erlang, mas com uma vantagem que é reduzida conforme aumenta a razão computação/comunicação da simulação. Isso sinaliza a viabilidade do uso do modelo de atores combinado à execução em nuvens não customizadas para aplicações de HPC na resolução de problemas complexos em engenharia.

[1] `https://aws.amazon.com/hpc/`

[2] `https://azure.microsoft.com/en-us/solutions/big-compute/`

O segundo foco da experimentação foi na habilidade do simulador baseado em Erlang de tratar falhas nos recursos computacionais. Os resultados obtidos mostram que o simulador é capaz de se recuperar de falhas mesmo quando os processos Erlang responsáveis pelas ações de coordenação no simulador são parcialmente afetados.

Cumpre destacar que este artigo apresenta semelhanças metodológicas com outro trabalho prévio dos autores [Gomes and Zillmer 2017], onde a abordagem multi-linguagem aqui sugerida foi originalmente apresentada. No entanto, em [Gomes and Zillmer 2017] outro método numérico baseado em elementos finitos e com acoplamento substancialmente menor que o HO-FV-LTS foi explorado, e nenhuma análise do tratamento de falhas foi conduzida. A solução e os resultados apresentados no presente artigo são dessa forma bem distintos.

O artigo está estruturado como se segue. Na Seção 2 o modelo de atores e a linguagem Erlang são brevemente introduzidos. Na Seção 3 o método HO-FV-LTS é apresentado com enfoque em seus aspectos computacionais. Na Seção 4 a arquitetura do simulador HO-FV-LTS baseado no modelo de atores é apresentada. Na Seção 5 são apresentados os resultados experimentais com as versões Erlang e MPI do simulador HO-FV-LTS e na Seção 6 são apresentadas as considerações finais e trabalhos futuros.

2. Erlang e o modelo de atores

A linguagem Erlang é baseada no modelo de atores proposto em [Agha 1985], tendo como principal característica a abstração de processos que não compartilham memória e que se comunicam por meio de troca assíncrona de mensagens. Notadamente, esse modelo de programação vem ganhando popularidade em ambientes de computação paralela e distribuída [Karmani and Agha 2011].

Erlang possui uma biblioteca conhecida como OTP (*Open Telecom Platform*), que simplifica a construção de sistemas distribuídos confiáveis [Armstrong 2010]. Um conceito central da OTP é o de 'árvore de supervisão', que introduz a ideia de trabalhadores e supervisores. Os trabalhadores são processos que implementam a funcionalidade fim da aplicação e os supervisores são aqueles que monitoram o funcionamento dos trabalhadores, sendo que eles podem reiniciar processos trabalhadores em caso de falha.

Árvores de supervisão são um tipo de 'comportamento' (*behaviour*) na OTP (chamado de comportamento `supervisor`). A OTP oferece outros comportamentos que reificam estilos arquiteturais comuns presentes em uma aplicação distribuída. Dentre eles temos o comportamento `application` que permite gerir a inicialização, a recuperação de falhas e a finalização distribuídas de uma aplicação, o comportamento `gen_server` que caracteriza uma aplicação cliente-servidor, e o comportamento `gen_statem` que dá suporte a criação e manipulação de máquinas de estados.

A linguagem oferece uma série de outras facilidades para programação distribuída (incluindo um banco de dados embarcado chamado `mnesia`, muito útil na implementação de estratégias de tolerância a falhas), mas que por questões de espaço não são apresentadas aqui. Devido a essas características Erlang tem recebido atenção recente no que se refere ao seu uso em computação científica. O trabalho de [Scalas et al. 2008] foi o primeiro a propor o uso de Erlang na construção de aplicações numéricas distribuídas. Mais recentemente, em [Trinder et al. 2017] é apresentada uma análise detalhada sobre a escalabilidade de Erlang, usando como estudos de caso o cálculo de órbitas

em álgebra simbólica e a otimização por meta-heurística de colônias de formigas. De forma análoga, em trabalho anterior dos autores do presente artigo um estudo foi conduzido explorando o uso de Erlang na implementação de métodos numéricos por elementos finitos [Gomes and Zillmer 2017]. No melhor do conhecimento dos autores este é o primeiro trabalho que conduz estudo semelhante sobre MVFs e que avalia o desempenho da aplicação também na presença de falhas.

3. O método numérico HO-FV-LTS

O método numérico HO-FV-LTS foi concebido originalmente para resolver equações diferenciais parciais hiperbólicas em redes de domínios unidimensionais. Sendo assim, grafos são usados para representar essas redes.

Dado um grafo G, considere $G_S = \{K_s\}_{s \in S}$ com $S \subset \mathbb{N} = \{0, \ldots, N_s - 1\}$ o conjunto resultante de uma partição do grafo G. Os subgrafos da partição formam a coleção de problemas locais, um para cada $K_s \in G_s$. A Figura 1 ilustra esses conceitos.

A simulação pelo método HO-FV-LTS do fluxo sanguíneo em um vaso se apresenta como uma equação de evolução que em sua forma matricial pode ser descrita como:

$$\partial Q + A(Q)\partial_x Q = Src(Q), \tag{1}$$

onde: (i) Q é o vetor de estado, mantendo informações sobre a taxa de fluxo sanguíneo e a força de fricção por unidade de comprimento no vaso, entre outras grandezas físicas de interesse do problema; (ii) $A(Q)$ e $Src(Q)$ representam a matriz de coeficientes físicos e o vetor de fonte, respectivamente; e (iii) x descreve a coordenada axial do vaso.

Para discretizar o domínio unidimensional de um vaso, o mesmo é dividido em células computacionais $T_i = [x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}]$ com tamanho denotado por Δx_i. A aproximação da solução pode então ser obtida integrando (1) em relação ao espaço e tempo no volume de controle $V_i^n = [x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}] \times [t^n, t^{n+1}]$.

Em uma rede de vasos a solução clássica de cômputo de integrais sobre todo o domínio pode envolver um alto custo computacional devido ao uso de um passo de tempo $\Delta t = t^{n+1} - t^n$ único e global para todos os vasos. Resumidamente, o tamanho

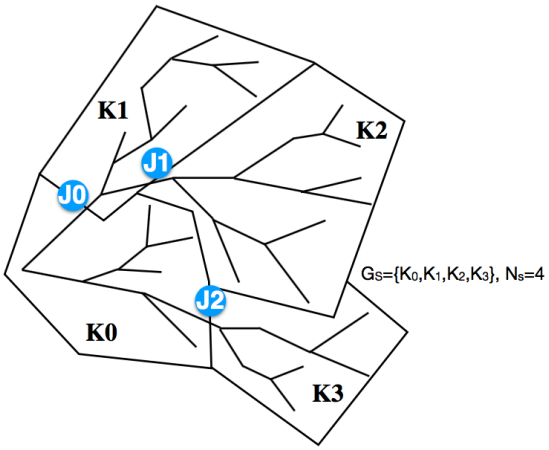

Figura 1. Exemplo de grafo particionado.

desse passo de tempo precisa respeitar uma condição de convergência (condição *Courant–Friedrichs–Lewy* – CFL) que o relaciona ao tamanho da maior célula computacional na rede de vasos, obrigando o uso de um Δt desnecessariamente pequeno em células de menor tamanho.

O esquema numérico proposto no HO-FV-LTS substitui esse processo de integração por um processo em três etapas que viabiliza o uso de diferentes passos de tempo locais por problema local K_s na obtenção da solução:

$$\begin{cases} \text{e1}: & Q_i^n & \rightarrow & w_i^n = w_i(x, t^n); \\ \text{e2}: & w_i^n & \rightarrow & Q_i^{ST}(x, t); \\ \text{e3}: & Q_i^n & \rightarrow & Q_i^{n+1}. \end{cases}$$

Etapa e1 **: reconstrução espacial** – a partir do vetor de estado Q_i^n no tempo t^n é obtido um conjunto de polinômios w_i^n para cada célula computacional i no problema local;

Etapa e2 **: predição local no espaço-tempo** – os polinômios obtidos na etapa anterior são usados para definir um problema de Riemann generalizado (PRG) na face $x_{i+\frac{1}{2}}$ de cada célula (localmente em $x = 0$):

$$\begin{cases} \partial Q + A(Q)\partial_x Q = Src(Q), \quad x \in \mathcal{R}, \quad t > t^n \\ Q(x,0) = \begin{cases} w_i(x,t^n) & \text{se} \quad x < 0 \\ w_{i+1}(x,t^n) & \text{se} \quad x > 0 \end{cases} \end{cases}$$

Sobre esse problema é usado o *solver* de *Dumbser-Enaux-Toro* (DET), que permite obter predições no espaço-tempo em cada face de célula para a solução do problema local.

Etapa e3 **: evolução dos dados** – a solução em um vaso de um problema local é atualizada para o tempo t^{n+1} com base nas predições obtidas na etapa anterior. O incremento do tempo local deve observar a condição CFL, mas somente nas células computacionais desse problema local.

Por questões de espaço e enfoque, detalhes da formulação matemática do método são omitidos neste artigo, podendo ser encontrados em [Müller et al. 2016].

3.1. Primitivas de computação

Para um melhor entendimento dos aspectos computacionais desse método, é apresentado na Listagem 1 o pseudocódigo para o algoritmo numérico de um simulador baseado no método HO-FV-LTS a partir de um conjunto de primitivas que são diretamente mapeadas nos blocos matemáticos principais do método: SPLIT_PROBLEM, INIT_VESSELS, RECONSTRUCT_AND_PREDICT e EVOLVE_DATA.

A primitiva SPLIT_PROBLEM divide o problema original definido sobre um grafo G em um conjunto de problemas locais G_S, um para cada partição do grafo. Essa primitiva usa o algoritmo de particionamento de grafos oferecido pelo pacote METIS,[3] sendo o número de partições de G definido pelo usuário. A primitiva retorna para cada $K_s \in G_S$ um par $(\{\mathcal{V}_{s,v}\}, \{\mathcal{J}_j\})$ com $v \in V \subset \mathbb{N} = \{0, \ldots, N_v - 1\}$ e $j \in J \subset \mathbb{N} = \{0, \ldots, N_j - 1\}$. $\{\mathcal{V}_{s,v}\}$ representa o conjunto de todos os vasos de K_s e $\{\mathcal{J}_j\}$ representa o conjunto de todas as 'junções' conhecidas de K_s. As junções são

[3] `http://glaros.dtc.umn.edu/gkhome/views/metis`

identificadas no momento do particionamento do grafo e compartilhadas com problemas locais adjacentes. Por exemplo, na Figura 1 temos três junções $\mathcal{J}_0$, $\mathcal{J}_1$ e $\mathcal{J}_2$, sendo que $\mathcal{J}_0$ é compartilhada entre os problemas locais K_0 e K_1. Essas junções permitem tratar as flutuações decorrentes de discrepâncias ocorridas na etapa de predição no espaço-tempo feita em problemas locais adjacentes e constituem-se no ponto focal onde ocorrem trocas de mensagens entre processos no simulador.

A primitiva INIT_VESSELS(K_s) é responsável por definir as condições iniciais de todos os vasos referentes ao problema local. A primitiva RECONSTRUCT_AND_PREDICT(K_s) é responsável por: (i) executar a reconstrução espacial e executar a predição no espaço-tempo para as faces das células de todos os vasos do problema local; e (ii) enviar predições parciais nas faces presentes na fronteira do problema local a problemas locais adjacentes por meio das junções, caso estejam disponíveis. O primeiro item se refere as etapas e1 e e2. O segundo item nos mostra que os problemas locais apresentam um certo nível de acoplamento por conta das junções, que é no entanto menor no HO-FV-LTS comparativamente a MVFs clássicos.

A primitiva EVOLVE_DATA(K_s) é responsável por: (i) utilizar as predições parciais recebidas nas junções pelos problemas locais adjacentes, caso estejam disponíveis; e (ii) atualizar a iteração e o tempo local. O primeiro item está relacionado ao último da primitiva RECONSTRUCT_AND_PREDICT(K_s), ou seja, é necessária uma consulta para verificar se resultados de predições parciais foram calculados por problemas locais adjacentes e usar tais predições em caso afirmativo. O segundo item contempla a etapa e3.

Listagem 1 Pseudocódigo para o algoritmo numérico de um simulador HO-FV-LTS.

```
Require: G, t_ini, t_end                          ▷ Grafo, Tempos inicial e final
  G_S = {K_s}_{s∈S} ← SPLIT_PROBLEM(G)            ▷ K_s := ({V_{s,v}}, {J_j})_{s∈S,v∈V,j∈J}
  for all K_s ∈ G_S do                            ▷ Executado em paralelo
      endOfSimulation ← 0
      K_s ← INIT_VESSELS(K_s)                     ▷ t_v^0 = t_ini, ∀v ∈ V
      while endOfSimulation ≠ 1 do
          K_s ← RECONSTRUCT_AND_PREDICT(K_s)
          endIteration ← 0
          while endIteration ≠ 1 do
              K_s ← EVOLVE_DATA(K_s)              ▷ t_v^{n_v} é atualizado, ∀v ∈ V
              if todos v ∈ V foram atualizados then
                  endIteration ← 1
              end if
              if endIteration = 0 then
                  K_s ← RECONSTRUCT_AND_PREDICT(K_s)
              end if
          end while
          if t_v^{n_v} ≥ t_end, ∀v ∈ V then
              endOfSimulation ← 1
          end if
      end while
  end for
```

Se ao final dessa primitiva o tempo final de simulação é alcançado, o simulador finaliza o problema local em questão.

4. Arquitetura do simulador baseado no modelo de atores

A arquitetura do simulador implementado em Erlang é ilustrada na Figura 2. Os elementos circulares representam processos Erlang e os elementos retangulares processos C++ com OpenMP que implementam as primitivas de computação do simulador. Essas primitivas são despachadas sob demanda para os recursos computacionais de um ambiente de execução. Esse ambiente de execução, representado na figura por dois nós processadores $N1$ e $N2$, pode ser um cluster de HPC ou uma nuvem computacional, por exemplo. Embora a figura possa sugerir que os processos Erlang executem fora do ambiente de execução, a localização ideal desses processos é dentro desse ambiente para minimizar a latência na troca de mensagens entre os processos Erlang e os processos C++.

O processo `app` possui o comportamento `application`, que inicia o simulador e dispara o processo `main_supervisor`. O processo `main_supervisor` implementa o comportamento `supervisor` e gerencia os processos `main_server`, `regis_server` e `fsm_supervisor`, dando origem a uma árvore de supervisão. Por fim, `fsm_supervisor` também apresenta o comportamento `supervisor` e gerencia os processos `leaf_fsm`. Todos esses processos executam em um mesmo nó chamado *master*; se o *master* se torna indisponível, esses processos são reiniciados em outro nó chamado *backup*. Erlang admite ainda múltiplos nós *backup* para aumentar a disponibilidade da aplicação. A presença de nós *master* e *backup* confere o primeiro nível de suporte de tolerância a falhas ao simulador.

O processo `main_server` mantém um conjunto de tarefas que são dinamicamente alocadas por intermédio dos processos `leaf_fsm` nos recursos computacionais disponíveis. Esses recursos computacionais são atribuídos aos processos `leaf_fsm` pelo processo `regis_server`, que pode recrutar ou liberar esses recursos sob demanda no ambiente de execução. Ambos os processos `main_server` e `regis_server` implementam o comportamento `gen_server`. Os processos `leaf_fsm` implementam o com-

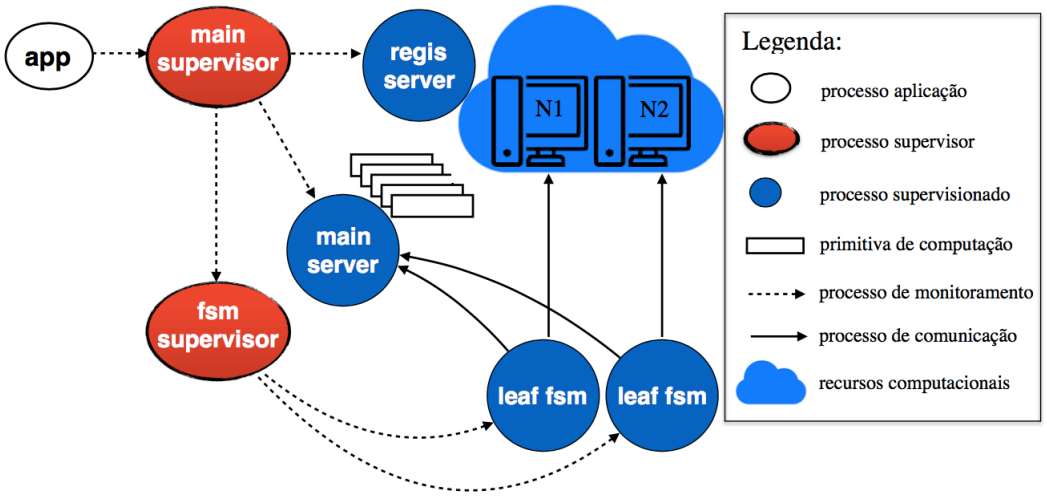

Figura 2. Arquitetura do simulador.

portamento gen_statem, implementando uma máquina de estados que evolui conjuntamente ao estado das tarefas mantidas pelo main_server.

Uma vez iniciado, um leaf_fsm continuamente solicita ao main_server novas tarefas por meio de requisições "get_next_problem". A tarefa recebida pelo leaf_fsm em resposta a uma dessas requisições é mapeada em uma das primitivas descritas na Seção 3. O leaf_fsm despacha e monitora a execução da primitiva como um processo C++ no recurso computacional a ele associado. Ao terminar, ele coleta os resultados da primitiva e os registra no main_server via outras requisições que podem mudar o estado da tarefa ou criar novas tarefas dependendo da primitiva em questão. É importante observar que nessa arquitetura não há comunicação direta entre os processos C++; assim, toda troca de dados relacionada às junções no método HO-FV-LTS – etapas e2 e e3 na Seção 3 – é intermediada pelos processos Erlang main_server e leaf_fsm. Esse arranjo simplifica consideravelmente o tratamento de falhas mas leva a um *overhead* importante na execução do simulador baseado em Erlang, como será visto na Seção 5.

Se o leaf_fsm detecta alguma falha associada à execução da primitiva (p.ex. falha no nó computacional ou partição na rede) ele notifica o main_server (que retorna a tarefa correspondente a seu último estado consistente) e o regis_server (que se encarrega de alocar um novo recurso computacional ao leaf_fsm). Esse mecanismo confere o segundo nível de suporte de tolerância a falhas ao simulador.

As Figuras 3 e 4 ilustram máquinas de estados na notação UML que representam a evolução das tarefas no simulador. Os estereótipos "<<Dynamic>>" e "<<Static>>" presentes nos estados dessas máquinas são descritos adiante.

Na máquina de estados HO_FV_LTS, ilustrada na Figura 3, é apresentado o estado da tarefa que representa o problema global G na Listagem 1. Essa tarefa é publicada no main_server com o estado "not_solved_global" no início da simulação. O estereótipo "<<Dynamic>>" na Figura 3 indica que qualquer leaf_fsm pode obter o problema global junto ao main_server via requisição "get_next_problem". Contudo, somente um leaf_fsm – o que tiver sua requisição tratada primeiro – receberá essa tarefa como resposta, despachando a primitiva SPLIT_PROBLEM em seu recurso computacional associado e mudando o estado da tarefa para "being_solved_global". Nesse estado, o leaf_fsm responsável por SPLIT_PROBLEM gerará um conjunto de requisições "insert_new_problem" ao main_server, uma para cada $K_s \in G_S$ produzido, desencadeando para cada K_s uma transição automática para o pseudoestado "interaction". O número de problemas locais a serem produzidos deve ser informado pelo usuário, sendo desejável para fins de eficiência computacional que esse número seja um múltiplo do número de nós computacionais disponíveis no ambiente de execução. Quando a primitiva SPLIT_PROBLEM é completada, a tarefa que estava no estado "being_solved_global" é finalizada e a máquina de estados HO_FV_LTS é encerrada, mas não a simulação, cujo comportamento será conduzido pelas máquinas de estado instanciadas a partir do pseudoestado "interaction", conforme descrito adiante.

Cada uma das transições para o pseudoestado "interaction" em HO_FV_LTS representa a criação de uma instância específica da máquina de estados LTS_SOLVER ilustrada na Figura 4. Essa máquina de estados representa o estado da tarefa associada ao problema local K_s, publicada inicialmente no main_server pela requisição "in-

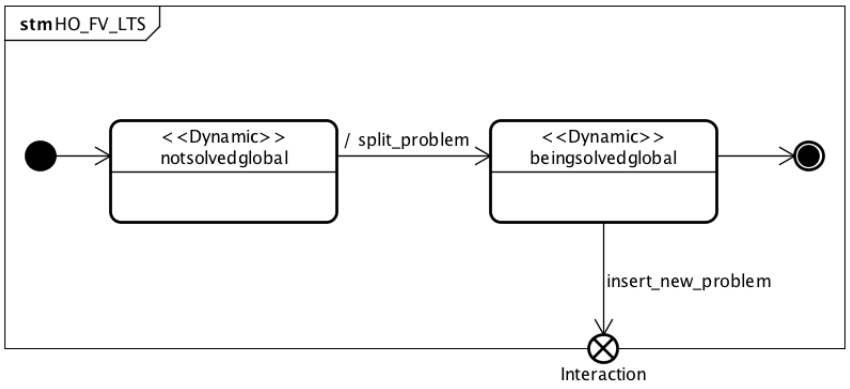

Figura 3. Evolução das tarefas na máquina de estados HO-FV-LTS.

sert_new_problem" como no estado "not_solved". O estereótipo "<<Static>>" indica que, uma vez que um `leaf_fsm` tenha obtido do `main_server` uma tarefa associada a um determinado problema local, nas próximas iterações envolvendo essa tarefa somente o `leaf_fsm` que a pegou na primeira vez poderá pegá-la novamente. Isso evita que em condições normais (isto é, na ausência de falhas) o processo C++ associado ao `leaf_fsm` em questão tenha que recarregar em memória os dados relacionados a essa tarefa a cada vez que uma primitiva envolvendo a mesma é disparada.

Sempre que um `leaf_fsm` recebe um problema local, o estado da tarefa correspondente muda de "not_solved" para "solved". Essa transição de estados é no entanto precedida de duas ações por parte do `leaf_fsm` modeladas como *entry* e *exit points* no estado "not_solved": (i) *entry* – o `leaf_fsm` requisita ao `main_server` tarefas cujos problemas locais compartilham junções com seu problema local e que tenham o atributo "sendReceive" definido para essas junções; e (ii) *exit* – o `leaf_fsm` requisita ao `main_server` o desligamento do atributo "sendReceive" nas junções identificadas em *entry*. O atributo "sendReceive" sinaliza assim a necessidade de uma ta-

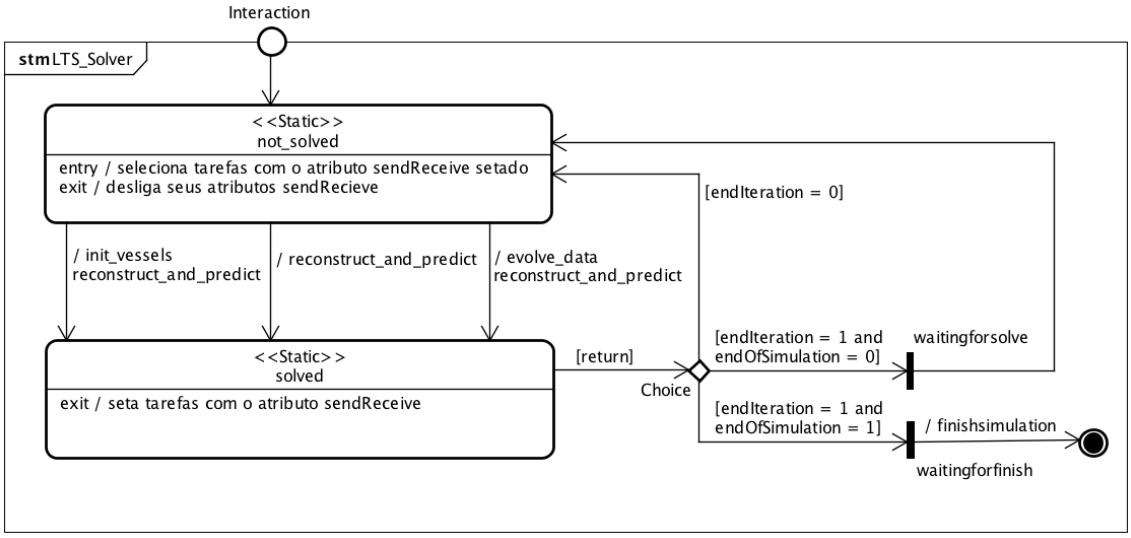

Figura 4. Evolução das tarefas na máquina de estados LTS Solver.

refa trocar informações entre problemas locais adjacentes e está ligado às predições parciais na fronteira desses problemas locais adjacentes tratadas nas primitivas RECONSTRUCT_AND_PREDICT e EVOLVE_DATA na Listagem 1. Além de "sendReceive", as tarefas associadas aos problemas locais possuem outros dois atributos – "endIteration" e "endOfSimulation" – que têm correspondência direta com as variáveis de mesmo nome na Listagem 1.

As primitivas a serem disparadas pelo `leaf_fsm` após as duas ações acima podem ter as seguintes configurações: (i) INIT_VESSELS e RECONSTRUCT_AND_PREDICT – caso seja a primeira vez que o problema local tenha sido recebido pelo `leaf_fsm`; (ii) RECONSTRUCT_AND_PREDICT – caso o estado anterior da tarefa tenha sido o pseudoestado "waitingforsolve"; e (iii) EVOLVE_DATA e RECONSTRUCT_AND_PREDICT – caso o atributo "endIteration" associado à tarefa seja igual a 0.

Como resultado de qualquer uma das primitivas, um conjunto de requisições pode ser enviado pelo `leaf_fsm` ao `main_server` para definir o atributo "sendReceive" em outras tarefas, dependendo da necessidade de atualização das informações nas junções dos problemas locais adjacentes. Além disso, há a mudança do estado relacionado ao problema local de "solved" para o pseudoestado "Choice", que determinará o próximo estado da tarefa dependendo do valor dos atributos "endIteration" e "endOfSimulation", como ilustrado na Figura 4. Em particular, quando "endIteration" é igual a 1, o próximo estado da tarefa será um entre dois possíveis pseudoestados que atuam como barreiras de sincronização com as demais tarefas: (i) "waitingforsolve" – as tarefas têm os tempos de simulação de seus problemas locais sincronizados antes de mudarem seus estados para "not_solved"; e (ii) "waitingforfinish" – as tarefas têm seu encerramento sincronizado, permitindo ao `main_server` encerrar a simulação como um todo.

A Figura 4 não contempla estados excepcionais pois estes são implicitamente tratados pelos mecanismos de tolerância a falhas de Erlang. Isso inclui o caso de parada total, no qual a simulação não precisa ser reiniciada – caso este não tratado no simulador original baseado em MPI. Para tanto, o simulador baseado em Erlang usa os estados das tarefas como *checkpoints*; quando uma falha ocorre em um recurso computacional (independente do mesmo hospedar um nó *master* ou não), todos os estados das tarefas envolvidas retornam para o último estado consistente no simulador. A Seção 5 analisa o *overhead* adicional no tempo total de execução decorrente do uso dessa estratégia.

5. Experimentos

Para avaliar o uso de Erlang na implementação do simulador HO-FV-LTS, a versão Erlang foi comparada à versão original baseada em MPI apresentada em [Müller et al. 2016], chamada nesta seção de 'implementação de referência'. O código C++ foi compilado em ambas as implementações usando o gcc-6.2 com as mesmas diretivas de otimização. Na implementação baseada em Erlang, foi utilizado o Erlang/OTP 17 (erts-6.1) com HiPE (compilação *just-in-time* para código nativo) ativado. Na implementação de referência, foi utilizada a biblioteca OpenMPI-2.1. Todas as simulações foram executadas no cluster Santos Dumont do LNCC, que apresenta a seguinte configuração por nó computacional: 2 processadores Intel Xeon E5-3695v2 Ivy Bridge com 12 cores cada (totalizando 24 cores por nó), 2.4 GHz, 64Gb de memória RAM DDR3. Os nós são interconectados por uma rede Infiniband FDR e têm acesso a um sistema de arquivos paralelo Lustre-2.1.

Em ambos os simuladores um único processo C++ foi mapeado por nó computacional, executando 24 threads OpenMP em cada nó (1 thread por core). Variou-se a quantidade de nós empregada nos experimentos de 1 a 9 (24 a 216 cores). As configurações de experimentação com 1 único nó usaram a implementação de referência tendo somente o OpenMP ativado para paralelização, sendo denominadas neste artigo de *baselines*.

O domínio do problema é o grafo da Figura 5, que representa uma rede arterial simplificada com 55 vasos. Cada um dos vasos da rede arterial é segmentado em um certo número de células computacionais de tamanho Δx cujo valor é definido pelo usuário. O passo de tempo é computado automaticamente pelo simulador para cada problema local em função de Δx de modo a respeitar a condição CFL. O tempo final de simulação e o número de CFL também são definidos pelo usuário, tendo sido usado nos experimentos $t_{end} = 0.1$s e $\mathrm{CFL} = 0.9$.

Quanto menor o Δx, maior a quantidade de células computacionais por vaso e, por conseguinte, maior a acurácia da solução e também maior a intensidade computacional de resolução do problema em cada vaso. A solução do problema global envolvendo os 55 vasos foi calculada para as duas versões do simulador usando valores para Δx variando de 0,12 a 0,015 unidades de distância. Cada uma dessas configurações foi exercitada 3 vezes, selecionando-se o menor tempo de execução entre elas,[4] para cada cenário de quantidade de nós empregados, num total de 105 rodadas de experimentos.

Os resultados das configurações exercitadas são apresentados na Tabela 1. Os tempos de execução apresentados estão em segundos e a eficiência é computada como

$$100\frac{T_b P_b}{T_p P_p},$$

onde T_p representa o tempo de execução obtido com P_p cores e T_b o tempo obtido na configuração *baseline* ($P_b = 24$).

Os resultados mostram que, a despeito do menor acoplamento do método HO-FV-LTS comparativamente a MVFs clássicos, sua escalabilidade é restrita mesmo na

[4]O número pequeno de amostras deveu-se a diferença insignificante entre os tempos de execução coletados, no máximo da ordem de 1%.

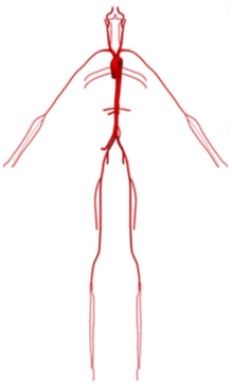

Figura 5. Rede arterial simplificada com 55 vasos sanguíneos (extraída de [Müller et al. 2016]).

Tabela 1. Medidas de tempo de execução e eficiência.

Δx	*Baseline*	MPI			Erlang		
	24 cores	72 cores	144 cores	216 cores	72 cores	144 cores	216 cores
0.12							
T (seg)	296	225	263	225	985	1187	1160
Eficiência	—	43,85%	18,76%	14,62%	10,02%	4,16%	2,84%
0.09							
T (seg)	926	795	641	597	2077	2168	2136
Eficiência	—	38,83%	24,08%	17,23%	14,86%	7,12%	4,82%
0.06							
T (seg)	2692	1770	1899	1707	4148	4120	4268
Eficiência	—	50,70%	23,63%	17,52%	21,63%	10,89%	7,01%
0.03							
T (seg)	21481	13976	14992	13421	24296	22692	22411
Eficiência	—	51,23%	23,88%	17,78%	29,48%	15,78%	10,65%
0.015							
T (seg)	159720	103370	109335	102640	141795	150985	149275
Eficiência	—	51,50%	24,35%	17,29%	37,55%	17,63%	11,89%

implementação de referência. Os resultados também mostram que no simulador baseado em Erlang o uso de múltiplos nós computacionais só começa a resultar em menores tempos de execução do que o *baseline* no conjunto de experimentos com maior razão computação/comunicação ($\Delta x = 0.015$). Pode-se observar ainda que, se por um lado a implementação de referência tem resultados consistentemente superiores aos do simulador baseado em Erlang, por outro lado a diferença entre a eficiência dos simuladores diminui consideravelmente à medida que aumenta-se a razão computação/comunicação dos problemas devido à redução de Δx.

Para avaliar como o simulador baseado em Erlang se comporta na presença de falhas, dois cenários foram criados para experimentos de curta duração ($\approx$ 120s por execução) usando o mesmo problema com $\Delta x = 0.12$ unidades de distância e $t_{end} = 0.02$ unidades de tempo. Falhas do tipo *fail-stop* foram artificialmente injetadas nos nós computacionais, segundo uma distribuição de probabilidade uniforme entre os nós. No primeiro cenário uma falha é injetada no intervalo $[20\text{s}, 60\text{s}]$ após o início da execução, e no segundo cenário duas falhas são injetadas nos intervalos $[20\text{s}, 60\text{s}]$ e $[60\text{s}, 80\text{s}]$ após o início da execução. Tais intervalos foram escolhidos de modo a garantir que as falhas aconteçam durante a execução do simulador e que não sejam muito no início da simulação para que o relançamento da mesma não tenha um custo negligenciável. Além disso, para fins de comparação um terceiro cenário sem falhas também foi utilizado. Todos esses cenários foram executados sobre 8 nós computacionais. Cada cenário foi executado 100 vezes, à exceção do cenário sem falha, executado somente 3 vezes como nos testes apresentados na Tabela 1. Em todos esses três cenários, 2 nós computacionais hospedam os processos Erlang de coordenação do simulador, sendo o primeiro nó o *master* e o segundo o *backup*: originalmente o nó *master* é o responsável pela coordenação, que é assumida pelo nó *backup* quando o *master* tem sua execução interrompida por falha.

O simulador foi capaz de se recuperar de todas as falhas em nós, exceto quando ambos os nós *master* e *backup* sofrem falhas no segundo cenário; tal situação ocorreu uma única vez nas 100 execuções desse cenário (o que é compatível com a probabilidade teórica de 1.11% de ambos os nós *master* e *backup* falharem). O simulador não é capaz de

se recuperar desse tipo de falha, mas a probabilidade de falha de todos os nós responsáveis pela coordenação pode ser facilmente reduzida aumentando o número de nós *backup* na simulação, o que é trivialmente obtido via configuração da aplicação distribuída Erlang sem necessidade de recodificação da aplicação.

No que se refere à eficiência do simulador na presença de falhas, quando ocorre uma só falha o tempo médio de execução cresce 13.4% em comparação com o tempo médio sem falhas. Quando duas falhas ocorrem o tempo médio de execução (para as 99 simulações completadas com sucesso) cresce 28.2% em comparação com o tempo médio sem falhas. Tais resultados demonstram que, na presença de falhas – ou, alternativamente, na presença de variações na disponibilidade de recursos durante a execução –, a abordagem proposta é mais conveniente do que o relançamento desde o início do simulador baseado em MPI, tanto para usuários como para provedores de recursos.

6. Conclusões

Este artigo propôs e avaliou objetivamente um abordagem multi-linguagem para o desenvolvimento de um simulador paralelo e tolerante a falhas para o método numérico HO-FV-LTS, empregando para tal o modelo de atores fornecido na linguagem Erlang. Os resultados obtidos mostram que conforme os problemas simulados aumentam sua razão computação/comunicação, a diferença entre a eficiência das soluções baseadas em MPI e Erlang diminui de forma importante, sugerindo que para problemas muito complexos o uso da abordagem proposta pode ser viável, com benefícios tanto no que se refere ao suporte de tolerância a falhas como na possibilidade de uso de ambientes onde o dinamismo na alocação de recursos é desejável (p.ex. em nuvens computacionais).

A despeito dos resultados apontarem para a viabilidade da abordagem, uma reavaliação da estratégia de implementação precisa ser conduzida. Como já dito, o fato de os processos Erlang intermediarem toda a comunicação entre processos C++ simplifica o tratamento de falhas mas impõe um *overhead* significativo no simulador. Esse é o principal ponto a ser atacado em implementações futuras do simulador baseado em Erlang, onde possivelmente seria uma melhor opção do ponto de vista de desempenho a adoção de uma arquitetura híbrida, empregando o estilo mestre-escravo no escalonamento dinâmico dos problemas locais nos nós computacionais e o estilo *peer-to-peer* na troca direta de dados relacionados às junções dos problemas locais. Arquitetura semelhante foi adotada em [Trinder et al. 2017] para o cálculo de órbitas em álgebra simbólica, com bons resultados. De todo modo, uma perfilação do código do simulador baseado em Erlang usando ferramentas específicas para essa linguagem[5] se faz necessária.

O número restrito de nós usado nos experimentos deve-se ao fato de que a rede arterial utilizada é pequena demais para ser particionada entre 10 ou mais nós. Outras redes com quantidade significativamente maior de vasos estão disponíveis (como citado na introdução) mas demandam a utilização de primitivas adicionais associadas ao método HO-FV-LTS que não puderam ser introduzidas na versão Erlang do simulador até o momento da escrita deste artigo. O emprego dessas redes no simulador baseado em Erlang explorando uma quantidade maior de nós foi deixado como trabalho futuro.

Nos experimentos deste trabalho foi adotado um percentual de falhas que viabilizasse o estudo da eficácia do mecanismo de recuperação de falhas. Como trabalho futuro

[5] `http://erlang.org/doc/efficiency_guide/profiling.html`

pretende-se investigar as taxas de falhas usualmente presentes nos ambientes de execução alvos deste trabalho e seu impacto na eficiência da abordagem aqui apresentada.

Agradecimentos

Os autores agradecem a Pablo Javier Blanco do LNCC e a Lucas Müller da Universidade Norueguesa de Ciência e Tecnologia pela disponibilização de acesso ao código fonte do simulador HO-FV-LTS original e pelas explicações acerca do método numérico.

Os autores agradecem ao Laboratório Nacional de Computação Científica (LNCC) por prover os recursos de HPC do supercomputador Santos Dumont (SDumont), que contribuíram para os resultados reportados neste artigo. (`http://sdumont.lncc.br`)

Referências

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

Armstrong, J. (2010). Erlang. *Communications of the ACM*, 53(9):68–75.

Blanco, P. J., Watanabe, S. M., Dari, E. A., Passos, M. A. R. F., and Feijóo, R. A. (2014). Blood flow distribution in an anatomically detailed arterial network model: criteria and algorithms. *Biomechanics and Modeling in Mechanobiology*, 13(6):1303–1330.

Blanco, P. J., Watanabe, S. M., Passos, M. A. R. F., Lemos, P. A., and Feijóo, R. A. (2015). An anatomically detailed arterial network model for one-dimensional computational hemodynamics. *Biomedical Engineering, IEEE Transactions on*, 62(2):736–753.

Eymard, R., Gallouët, T., and Herbin, R. (2000). Finite volume methods. *Handbook of numerical analysis*, 7:713–1018.

Gomes, A. T. A. and Zillmer, F. (2017). Experimentos no uso do modelo de atores para simulações numéricas distribuídas baseadas em elementos finitos. *XV Workshop em Clouds e Aplicações*, pages 144–157.

Karmani, R. K. and Agha, G. (2011). *Actors*, pages 1–11. Springer US, Boston, MA.

Müller, L. O., Leugering, G., and Blanco, P. J. (2016). Consistent treatment of viscoelastic effects at junctions in one-dimensional blood flow models. *Journal of Computational Physics*, 314:167 – 193.

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.

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.

Trinder, P., Chechina, N., Papaspyrou, N., Sagonas, K., Thompson, S., Adams, S., Aronis, S., Baker, R., Bihari, E., Boudeville, O., Cesarini, F., Stefano, M. D., Eriksson, S., fördős, V., Ghaffari, A., Giantsios, A., Green, R., Hoch, C., Klaftenegger, D., Li, H., Lundin, K., Mackenzie, K., Roukounaki, K., Tsiouris, Y., and Winblad, K. (2017). Scaling reliably: Improving the scalability of the erlang distributed actor platform. *ACM Trans. Program. Lang. Syst.*, 39(4):17:1–17:46.