Universidade Federal de Juiz de Fora
Programa de Pós-Graduação em Modelagem Computacional

Ricardo Silva Campos

FERRAMENTAS *WEB* PARA DESCRIÇÃO E SIMULAÇÃO DE MODELOS DE CÉLULAS CARDÍACAS

Juiz de Fora
2011

Ricardo Silva Campos

Ferramentas *Web* para descrição e simulação de modelos de células cardíacas

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Orientador: Prof. D.Sc. Rodrigo Weber dos Santos

Coorientador: Prof. D.Sc. Marcelo Lobosco

Coorientador: Prof. D.Sc. Ciro de Barros Barbosa

Juiz de Fora

2011

Campos, Ricardo Silva.

Ferramentas *Web* para descrição e simulação de modelos de células cardíacas/Ricardo Silva Campos. – 2011.

120 f. : il.

Dissertação (Mestrado em Modelagem Computacional) – Universidade Federal de Juiz de Fora, Juiz de Fora, 2011.

1. engenharia biomédica. 2. eletrofisiologia. I. Título

CDU 61:573

Ricardo Silva Campos

Ferramentas *Web* para descrição e simulação de modelos de células cardíacas

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Aprovada em 29 de Agosto de 2011.

BANCA EXAMINADORA

Prof. D.Sc. Rodrigo Weber dos Santos - Orientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Marcelo Lobosco - Coorientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Ciro de Barros Barbosa - Coorientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Roque Luiz da Silva Pitangueira
Universidade Federal de Minas Gerais

Prof. D.Sc. Luis Paulo da Silva Barra
Universidade Federal de Juiz de Fora

AGRADECIMENTOS

Agradeço primeiramente a Deus.

Agradeço aos meus pais, minha irmã e minha noiva pelo apoio e carinho nesta caminhada.

Agradeço aos meus orientadores pela paciência e pela oportunidade dada a um voluntário.

Agradeço aos amigos do Fisiocomp e do mestrado pelo companheirismo sempre presente.

RESUMO

A modelagem da eletrofisiologia cardíaca é uma importante técnica para compreender e reproduzir o fenômeno de propagação de ondas elétricas no coração. Cada onda é chamada de potencial de ação e é responsável pela sincronização dos batimentos cardíacos. Este potencial depende de vários fatores, como a capacitância da membrana celular e concentrações de diferentes íons nos meios intra e extracelulares. Tipicamente, estes componentes podem ser representados por circuitos elétricos, que podem ser descritos por equações diferenciais ordinárias. Entretanto, o processo de geração do potencial de ação é complexo e de natureza não-linear. Para simulá-lo através de experimentos *in silico*, é necessário descrevê-lo através de dezenas de equações e parâmetros. Além disto, é necessário resolver as equações por meio de métodos numéricos eficientes. Visando auxiliar este processo de modelagem, este trabalho possui dois objetivos: 1) desenvolver uma ferramenta para descrever modelos computacionais que funcione através da *Web* e permita a edição de arquivos CellML – um padrão XML desenvolvido para descrever modelos celulares; 2) aprimorar os métodos numéricos utilizados pela ferramenta AGOS, que transforma CellML em um arquivo C++ que permite a simulação dos modelos. Diferentes métodos de passo de tempo adaptativo foram implementados e os algoritmos foram paralelizados via OpenMP. Esses métodos e técnicas computacionais foram comparados aos já então amplamente adotados pela área, métodos de Euler e BDF, avaliação parcial e Lookup-Tables, para a simulação de quatro diferentes modelos de células cardíacas. Os resultados mostraram que os métodos adaptativos combinados com as técnicas computacionais podem ser até 100 vezes mais velozes do que o método de Euler.

Palavras-chave: engenharia biomédica. eletrofisiologia.

ABSTRACT

Cardiac electrophysiology modeling is an important technique for studying and simulating the electrical wave propagation on cardiac tissue. The electrical wave initiates and propagates as a pulse that is known as action potential. The action potential is responsible for synchronizing the contraction and relaxation of the cardiac cells during a heartbeat. The cellular components and functions involved in the generation of an action potential are typically described by sets of ordinary differential equations. In-silico experiments of this complex phenomenon involve the description of the mathematical model and its numerical resolution. In this respect, this work targets two different goals. First, we have implemented a Web tool to create and edit cellular components and mathematical equations, based on a XML standard named CellML. The second goal is to improve the numerical resolution of models described in CellML. To this end we implemented different improvements to a previously published tool called AGOS. AGOS translates a CellML file to a C++ code that can be used for the numerical resolution of the model via the Euler or BDF methods. In this work, we have improved the numerical methods of AGOS by implementing and testing two different ways to adapt the time step. In addition, we have implemented a parallel version of the numerical solvers based on OpenMP directives, and added two numerical techniques known as Partial Evaluation and Lookup Tables to AGOS. We compared these computational techniques in terms of execution time, memory consumption and numerical error. Our preliminary results suggest that the adaptive time step methods combined with OpenMP or Lookup Tables and Partial Evaluation can be 100 times faster than the originally implemented Euler Method.

Keywords: biomedical egineering. electrophysiology.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AFD Autômato Finito Determinista, 66

AGOS Application program interface Generator for ODE Solution, 45

AP Avaliação Parcial, 48

API Application Program Interface, 46

BDF Backward differentiation formulas, 31

BDK Modelo de Bondarenko et al., 25

EDO Equacão Diferencial Ordinária, 25

ELR Método de Euler, 33

GHK Fórmula de Goldman-Hodgkin-Katz, 20

GRN Modelo de Garny et al., 28

LD Lado Direito, 32

LUT LookUp Tables, 50

NBL Modelo de Noble et al., 29

PA Potencial de Ação, 19

PVI Problema de Valor Inicial, 32

RK2 Método de Runge-Kutta de Segunda Ordem, 33

SI Sistema Internacional de Unidades, 42

TTP Modelo de ten Tusscher e Pafilov, 27

1 INTRODUÇÃO

A modelagem computacional permite a simulação de diversos fenômenos, com o objetivo de estudá-los, compreendê-los e até mesmo tentar prevê-los. Porém, o processo de representação e simulação destes fenômenos complexos é complicado e suscetível a erros, pois envolve sistemas com dezenas de equações diferenciais e centenas de parâmetros. Além disso, há uma demanda de grande poder computacional para resolver modelos realistas. O fenômeno abordado neste trabalho está relacionado a eletrofisiologia cardíaca, que estuda a atividade elétrica das células do coração e suas consequências para o funcionamento deste órgão. Existem várias pesquisas nesta área, motivadas pelo alto índice de mortalidade causado por problemas cardiovasculares [1]. Existem várias aplicações da modelagem computacional nesta área, por exemplo: a avaliação de novas drogas e sua influência na propagação do impulso elétrico no coração [2]; a modelagem de doenças, como a Doença de Chagas [3]; e a influência da atividade elétrica celular na contração mecânica do coração [4].

O primeiro objetivo deste trabalho é desenvolver uma ferramenta para auxiliar o processo de representação e descrição de um modelo celular. A ferramenta é um Editor para a linguagem CellML[5] e MathML[6], padrões XML (eXtensible Markup Language)[7] que representam componentes celulares e equações matemáticas, respectivamente. O Editor funciona pela *Web*, permitindo que usuários de diferentes sistemas operacionais possam acessá-lo. As metas são a simplificação do processo de modelagem, redução de erros e permitir que usuários com diferentes níveis de conhecimento em linguagem de programação possam desenvolver modelos computacionais da eletrofisiologia cardíaca. Esta última característica é importante em áreas multidisciplinares, onde há pesquisadores de diferentes áreas e consequentemente com diferentes habilidades em programação. Outro aspecto importante é que o modelo é criado independente da linguagem de programação, permitindo que ele seja resolvido pelas ferramentas baseadas em CellML, além de permitir o reuso de seus componentes.

Outra característica a ser considerada é a dificuldade de resolver as equações dos modelos da eletrofisiologia cardíaca, que são complexos, de natureza altamente não-linear e envolvem múltiplas escalas. O que significa que simular estes modelos demanda poder

de processamento computacional e pode levar muito tempo, de acordo com a complexidade do fenômeno. Por conseguinte, o outro objetivo deste trabalho é aprimorar técnicas computacionais para resolver as equações dos sistemas através do AGOS[8], que é uma ferramenta que transforma um arquivo CellML em código C++. Inicialmente, a ferramenta era capaz de executar simulações através de dois métodos numéricos: Euler com passo de tempo fixo e *Backward Differentiation Formulas*. Para que se melhore o desempenho do AGOS, foram implementados dois métodos que são capazes de adaptar o passo de tempo. Esta adaptação aumenta o passo de tempo em regiões estáveis da solução, com o intuito de diminuir o tempo de execução. Entretanto, em regiões instáveis é necessário que se mantenha o passo de tempo com valores menores, para que os percentuais de erro sejam considerados aceitáveis.

Além disto, foram utilizadas técnicas de computação paralela com OpenMP, que permite a utilização de recursos computacionais em ambientes multiprocessados com memória compartilhada. Também foram utilizadas técnicas avançadas para resolução de equações diferenciais, conhecidas como *Lookup Tables* (LUT) e Avaliação Parcial. Ambas são fornecidas pela ferramenta PyCML[9].

Os resultados foram avaliados considerando o tempo de execução, consumo de memória e percentual de erro. Os métodos adaptativos foram até 30 vezes mais rápidos do que o método de Euler. Ao se adicionar as técnicas de LUT ou OpenMP, o desempenho pode ser até 100 vezes melhor.

O texto se organiza da seguinte maneira: o Capítulo 2 apresenta alguns conceitos da eletrofisiologia cardíaca. O Capítulo 3 apresenta os métodos numéricos, a linguagem CellML, o AGOS, o PyCML e suas técnicas e o OpenMP. O Capítulo 4 contém a metodologia do trabalho, que explica como foram implementados o Editor, os métodos com passo de tempo adaptativo, a paralelização com OpenMP e a combinação do AGOS com o PyCML. Os resultados são apresentados no Capítulo 5 e discutidos no Capítulo 6. Finalmente, conclui-se o trabalho no Capítulo 7.

2 ELETROFISIOLOGIA CARDÍACA

O coração é o órgão responsável por impulsionar sangue para todo o corpo. Ele é composto por quatro câmaras, que se dividem em dois ventrículos e dois átrios. O fluxo de sangue chega ao coração pelo átrio direito, de onde é expelido para o ventrículo direito, que por sua vez impulsiona sangue para os pulmões, onde ocorre troca de gás carbônico por oxigênio. Então o sangue segue para o átrio esquerdo e depois para o ventrículo esquerdo, que impulsiona o fluxo para os órgãos periféricos. A Figura 2.1 mostra a anatomia do coração.

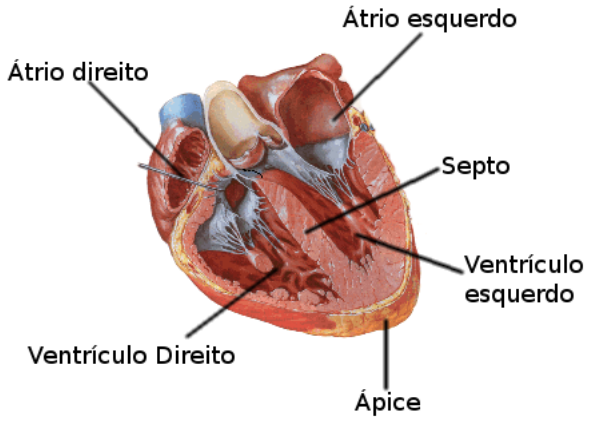

Figura 2.1: Anatomia do coração - Figura retirada de [10]

Para cumprir a sua função, o coração deve se contrair de forma sincronizada. Para tal, existe um sistema especial para gerar e conduzir impulsos elétricos, que sincronizam os movimentos de relaxamento e contração do tecido cardíaco. Diversas doenças atrapalham este sistema de propagação elétrica, que causam ritmos anormais de contração, resultando em bombeamento insuficiente de sangue, o que pode levar à morte.

No tecido cardíaco há alguns pontos capazes de gerar estímulos elétricos que serão propagados para o restante do tecido cardíaco. Estes pontos são: nó sinoatrial, que é o principal gerador de impulsos, também conhecido como marca-passo natural do coração; nó atrioventricular; e fibras de Purkinje. Este locais estão ilustrados na Figura 2.2.

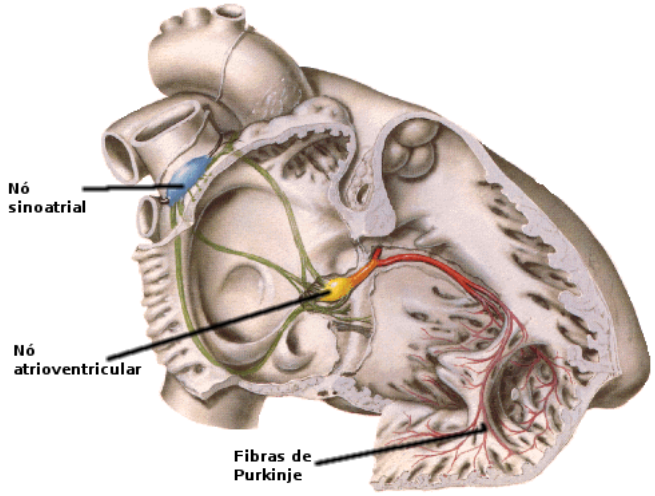

Figura 2.2: Sistema condutor do coração - visão do lado direito - Figura retirada de [10]

2.1 Potencial de Ação

O potencial de ação (PA) é uma rápida variação no potencial elétrico da membrana celular, e ocorre em diversas células, com diferentes finalidades. Por exemplo, em tecidos nervosos e musculares são gerados impulsos eletroquímicos que transmitem sinais por todo o tecido. Em células glandulares, macrófagas e ciliadas, mudanças no potencial da membrana ativam funções celulares. No tecido cardíaco, o PA é responsável por sincronizar o ritmo de contração e relaxamento do órgão.

Este mecanismo ocorre devido a diferença de concentração de substâncias entre os meios intra e extracelular, que são separados pela membrana. A membrana possui comportas que regulam a troca de substâncias entre os meios, chamados de canais protéicos.

Observa-se que naturalmente o meio intracelular é rico em potássio (K^+), que possui carga positiva, e o extracelular possui baixa concentração do mesmo íon. O potássio tenderá a se difundir do meio mais concentrado para o menos concentrado, levando consigo a carga positiva, deixando o meio extracelular carregado positivamente, e o intracelular, negativamente.

O aumento de carga positiva no exterior e de carga negativa no interior interfere no processo de difusão, fazendo que os íons de potássio voltem a entrar na célula, atenuando a voltagem transmembrânica. Porém, no meio extracelular há uma grande concentração de íons de sódio Na^+. Quando o meio intracelular torna-se negativo, esses íons movem-se para o interior da célula. A manutenção e o equilíbrio dos potenciais de membrana são controlados pelas bombas de sódio e potássio.

O potencial resultado do equilíbrio entre a difusão iônica e o campo elétrico gerado é chamado de potencial de Nernst. Este potencial é determinado pela concentração do íon nos dois lados da membrana. A tendência do íon se difundir para uma direção é proporcional à concentração. A fórmula que define o potencial de Nernst é:

$$v = \frac{RT}{zF} ln \frac{C_i}{C_e} \tag{2.1}$$

onde C_i e C_e representam a concentração iônica interna e externa, respectivamente, R é a constante dos gases, T é a temperatura absoluta, z é a valência do íon e F é a constante de Faraday.

Para calcular o potencial quando a membrana é permeável a vários íons, deve-se considerar três fatores para cada íon, a polaridade da carga elétrica, a permeabilidade (P) da membrana e as concentrações (C) interna (i) e externa (e). A Equação 2.2, chamada de Goldman-Hodgkin-Katz (GHK), calcula o potencial de repouso da membrana quando os íons Na^+, K^+ e Cl^- estão envolvidos:

$$v = -\frac{RT}{zF} ln \frac{C_{Na_i^+} P_{Na^+} + C_{K_i^+} P_{K^+} + C_{Cl_e^-} P_{Cl^-}}{C_{Na_e^+} P_{Na^+} + C_{K_e^i} P_{K^+} + C_{Cl_i^-} P_{Cl^-}} \tag{2.2}$$

O PA pode ser dividido em três etapas, descritas a seguir. Na primeira etapa, o PA está em repouso, que é o potencial normal de uma membrana. Nesta situação, a membrana está polarizada, já que o seu interior está carregado negativamente. O potencial de repouso é aproximadamente -90 mV em fibras nervosas, cujo modelo é descrito na Seção 2.3. Para estas células o PA é iniciado quando o meio intracelular recebe cargas positivas, o que causa uma rápida despolarização, que é a segunda etapa. Alguns milisegundos depois, ocorre a terceira etapa, que é a repolarização, quando o potencial tende a ficar em repouso.

Durante a despolarização, subitamente a membrana fica muito permeável ao íons de sódio, permitindo que estes se difundam para o interior da célula, levando consigo a sua carga positiva. A entrada desta carga faz com que o potencial cresça positivamente. Em algumas células o pico do PA é atingido próximo do potencial zero, porém em outras, em que há maiores concentrações de íons positivos, o potencial pode ultrapassar zero.

A repolarização começa após 0,1 ms da membrana ficar permeável ao sódio, quando os canais que transmitem estes íons começam a se fechar, enquanto os canais de potássio abrem-se abruptamente. Deste modo, a rápida difusão de potássio para o exterior resta-

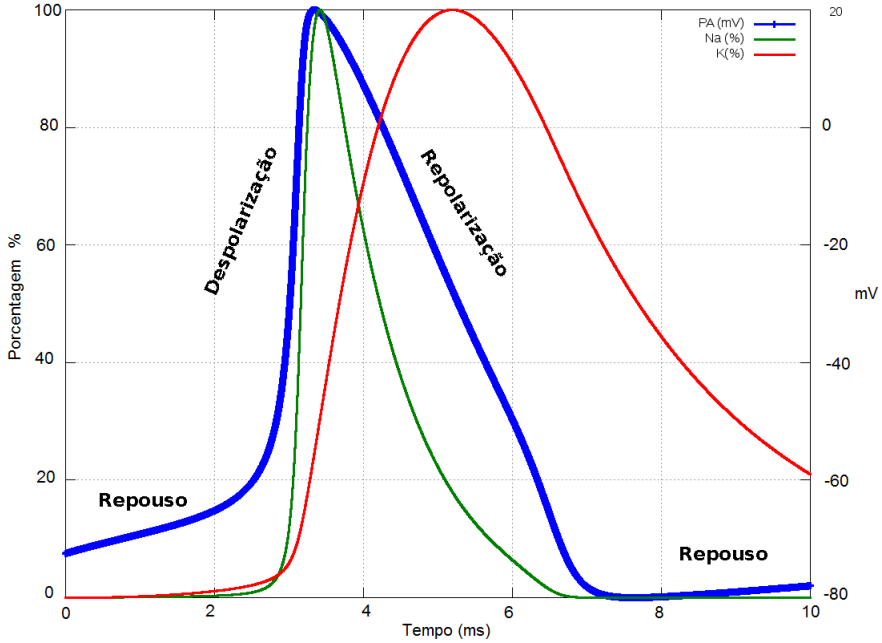

Figura 2.3: PA e condutâncias de sódio e potássio

belece o potencial de repouso.

A Figura 2.3 ilustra o PA e as condutâncias dos canais de sódio e potássio, em uma célula nervosa[11]. Os dados estão normalizados, para melhor visualização.

Maiores detalhes sobre este fenômeno podem ser encontrados na literatura[12].

2.2 Modelo para a membrana

Como descrito anteriormente, a membrana isola os meios, que estão eletricamente carregados. Deste modo, a membrana pode ser modelada como um capacitor. A capacitância é definida como a razão entre a carga armazenada e a voltagem necessária para armazenar esta carga:

$$C_m = \frac{Q}{V} \tag{2.3}$$

Onde C_m é a capacitância da membrana, Q é a carga armazenada e V é a diferença de potencial entre o meio interno e externo, ou seja, $V = V_i - V_e$. O fluxo de íons muda a quantidade de carga armazenada pelo capacitor, assim como o potencial transmembrânico, através da relação $Q = C_m V$ definida na Equação 2.3. Considerando a variação de carga no tempo, tem-se:

$$\frac{dQ}{dt} = C_m \frac{dV}{dt} \tag{2.4}$$

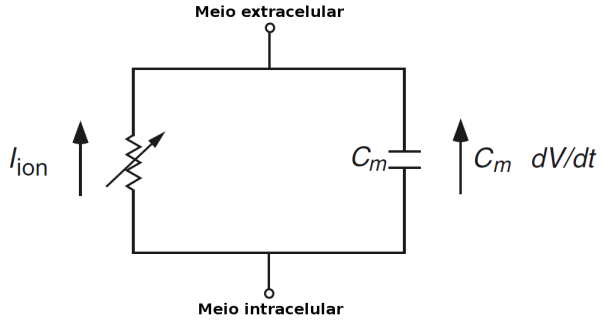

Figura 2.4: Modelo de circuito elétrico da membrana - adaptado de [13]

O termo $\frac{dQ}{dt}$ é a variação de carga no tempo, que corresponde a uma corrente, denominada de corrente capacitiva. A corrente total através da membrana celular é definida como a soma da corrente capacitiva com a corrente iônica que atravessa os canais,

$$I_m = I_{ion} + I_c \tag{2.5}$$

Substituindo a Equação 2.4 segue que

$$I_m = I_{ion} + C_m \frac{dV}{dt} \tag{2.6}$$

Uma vez que não pode haver acúmulo de carga nos dois lados da membrana, a soma das correntes iônica e capacitiva deve ser zero, e assim:

$$I_{ion} + C_m \frac{dV}{dt} = 0 \tag{2.7}$$

Esta equação é a base de todos os modelos utilizados neste trabalho. A principal diferença entre eles é a corrente iônica I_{ion}. Representar I_{ion} através de equações matemáticas é um grande desafio, pois envolve fenômenos complexos que variam muito de acordo com o tipo de célula a ser modelado. Uma forma de determinar I_{ion} é através da fórmula também denominada GHK, que estende a Equação 2.2. Maneiras mais complexas de representá-la são encontradas na literatura[13].

A Figura 2.4 representa um modelo elétrico de uma membrana. A membrana funciona como um capacitor em paralelo com uma resistência, que representa os canais iônicos.

2.3 Modelo de Hodgkin-Huxley

Na década de 1950, um modelo para o potencial de ação foi proposto por Hodgkin e Huxley[11], que receberam mais tarde o prêmio Nobel de Medicina. Eles observaram o fluxo de corrente elétrica pela membrana de um axônio de lula e desenvolveram uma descrição matemática do comportamento da membrana, a partir de dados experimentais.

O modelo tem dois elementos básicos, a membrana e os canais iônicos que a permeiam, ilustrados na Figura 2.5. A membrana é modelada eletricamente como um dielétrico de capacitância C_m. Os canais iônicos são modelados como resistências não lineares dependentes da diferença de potencial $V = V_i - V_e$ sobre a membrana e da concentração de diversos íons, onde os íons de sódio e potássio formam as correntes iônicas mais importantes para a mudança de potencial da membrana. O circuito resultante desta combinação está na Figura 2.5.

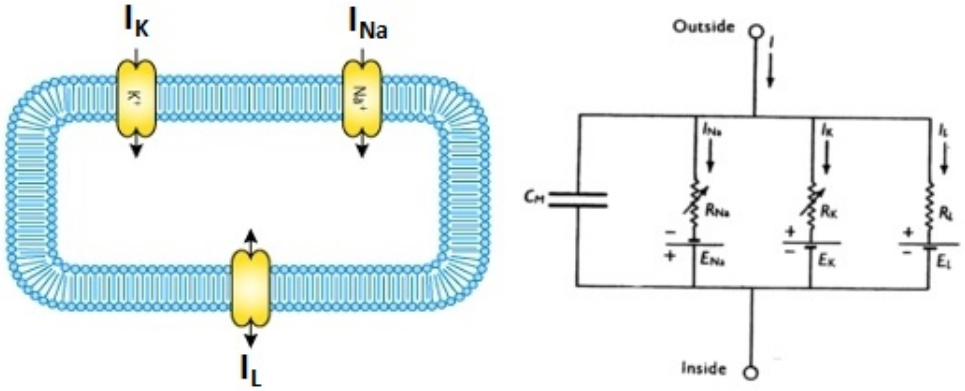

Figura 2.5: Representação da célula e o circuito. Figura retirada de [11]

No modelo Hodgkin e Huxley três correntes são assumidas: $I_{ion} = I_{Na} + I_K + I_l$. Denotando a capacitância da membrana por C_m, obtém-se, a partir de 2.7, que:

$$-C_m \frac{dV}{dt} = I_{Na} + I_K + I_l \tag{2.8}$$

I_{Na}, I_K são, respectivamente, as correntes de sódio e potássio e I_L é uma corrente de fuga constante. Estas correntes podem ser expressas pelas condutâncias iônicas (g_{Na}, g_K, g_l):

$$I_{Na} = g_{Na}(E - E_{Na}) = g_{Na}(V - V_{Na}) \tag{2.9}$$

$$I_K = g_K(E - E_K) = g_K(V - V_K) \tag{2.10}$$

$$I_l = g_l(E - E_l) = g_l(V - V_l) \tag{2.11}$$

Onde E_{Na} e E_K são os potenciais de equilíbrio para os íons de sódio e o potássio e E_l é o potencial em que a corrente de escape, causada por outros íons, como o de cloro, é zero. Os potenciais E foram substituídos por V por conveniência, onde

$$V = E - E_r \tag{2.12}$$

$$V_{Na} = E_{Na} - E_r \tag{2.13}$$

$$V_K = E_K - E_r \tag{2.14}$$

$$V_l = E_l - E_r \tag{2.15}$$

Com E_r sendo o valor absoluto do potencial de repouso e V, V_{Na}, V_K, e V_l podem ser medidos diretamente como deslocamentos do potencial de repouso.

A condutância do potássio é descrita pelas seguintes equações:

$$g_K = \overline{g}_K n_4 \tag{2.16}$$

$$\frac{dn}{dt} = \alpha_n(1-n) - \beta_n n \tag{2.17}$$

Onde $\overline{g}_K$ é a constante com dimensões de condutância por cm^2, α_n e β_n são taxas que variam com a voltagem e não com o tempo, e n varia entre 0 e 1. Estas taxas são dadas por:

$$\alpha_n = \frac{0,01(V+10)}{e^{\frac{V+10}{10}} - 1} \tag{2.18}$$

$$\beta_n = 0,125e^{\frac{V}{80}} \tag{2.19}$$

A variável n representa a proporção de partículas no interior da membrana, e $1-n$, a proporção no exterior. α_n determina a taxa de transferência de partículas do exterior para o interior, enquanto β_n determina a transferência no sentido contrário.

A condutância do sódio é modelada pelas seguintes equações:

$$g_{Na} = m^3 h \overline{g}_{Na} \tag{2.20}$$

$$\frac{dm}{dt} = \alpha_m(1-m) - \beta_m m \tag{2.21}$$

$$\frac{dh}{dt} = \alpha_h(1-h) - \beta_h h \tag{2.22}$$

Onde $\overline{g}_{Na}$ é uma constante e α_m, α_h, β_m e β_h são funções de V e não de t. m representa a proporção de moléculas ativadoras no meio intracelular e $1 - m$, a proporção no meio extracelular. h é a proporção de moléculas inativadoras no exterior da célula e $1 - h$, no interior. α_m ou β_h e β_m e α_h representam as constantes taxas de transferência de moléculas entre os meios, em ambas as direções, e são dadas por:

$$\alpha_m = \frac{0,1(V+25)}{e^{\frac{V+25}{10}} - 1} \tag{2.23}$$

$$\beta_m = 4e^{\frac{V}{18}} \tag{2.24}$$

$$\alpha_h = 0,07e^{\frac{V}{20}} \tag{2.25}$$

$$\beta_h = \frac{1}{e^{\frac{V+30}{10}} + 1} \tag{2.26}$$

Maiores detalhes da derivação destas equações e a comparação do resultado que elas produziram com experimentos podem ser conferidos no artigo de Hodgkin e Huxley [11].

Neste trabalho foram utilizados quatro modelos celulares, que serão sucintamente descritos a seguir.

2.4 O modelo de Bondarenko et al.

O modelo de Bondarenko et al.[14](BDK) é um modelo do PA em células do ventrículo de camundongo. Ao todo, o modelo possui 41 Equações Diferenciais Ordinárias (EDOs) que representam as correntes iônicas. O modelo descreve detalhadamente a dinâmica intracelular do cálcio (Ca^{2+}) e reproduz as propriedades das células das regiões do ápice e do septo, que estão indicadas na Figura 2.1.

Os canais iônicos são representados por cadeias de Markov. Estas cadeias são sistemas matemáticos em que se transita de um estado para o outro, onde há um número finito de estados. O próximo estado será escolhido aleatoriamente, dependendo apenas do estado atual.

Um exemplo básico de cadeias de Markov modelando correntes iônicas é o modelo de três estados, que representam os estados que um canal iônico pode assumir, podendo ser aberto (O), fechado (C) e inativo (I). A Figura 2.6 ilustra as transições existentes entre os estados. A corrente total de um íon S que passa por este canal é dada pelo seguinte

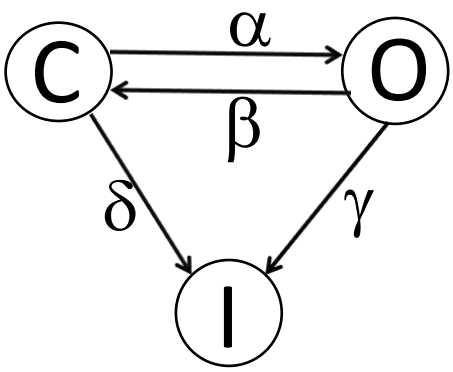

Figura 2.6: Cadeia de Markov para três estados

sistema de equações:

$$\begin{aligned} I_S &= g_S o(v - E_s) \\ \frac{dc}{dt} &= -(\alpha + \delta)c + \beta o \\ \frac{do}{dt} &= \alpha c - (\beta + \gamma)o \\ \frac{di}{dt} &= \gamma o + \delta c \end{aligned} \tag{2.27}$$

Onde c, o e i são as proporções em que os canais se encontram nos estados fechado, aberto e inativo, e α, β, γ e δ são as taxas de transição entre os estados.

A corrente iônica total deste modelo é a soma de dezesseis correntes transmembrânicas e é dada pela Equação 2.28. O diagrama esquemático da célula está descrito na Figura 2.7.

$$\begin{aligned} I_{ion} = I_{CaL}(O,V) + I_{p(Ca)}(Ca_i) + I_{NaCa}(V, Na_i, Ca_i) \\ + I_{Cab}(V) + I_{Na}(V, O_{Na}) + I_{Nab}(V) + I_{NaK}(NaK) \\ + I_{Kto,f}(ato_f, ito_f, V) + I_{Kto,s}(ato_s, ito_s, V) + I_{K1}(V) \\ + I_{Ks}(nKs) + I_{Kur}(aur, iur, V) + I_{Kss}(aKss, iKss) \\ + I_{Kr}(O_K, V, K_i, Na_i) + I_{Cl,Ca}(Ca_i, V) + I_{stim}() \end{aligned} \tag{2.28}$$

Onde I_{CaL} é a corrente de cálcio Ca^{2+} tipo L, $I_{p(Ca)}$ é a corrente máxima da bomba de cálcio($Ca2$), I_{NaCa} é o trocador sódio Na^+ e cálcio Ca^{2+}, I_{Na} é a corrente rápida de Na^+, I_{Nab} e I_{Cab} são as correntes de fundo de Na^+ e Ca^{2+}, I_{NaK} é a bomba Na^+/K^+, $I_{Kto,s}$ e $I_{Kto,f}$ são a inativação lenta e rápida da corrente $K+$ transiente de saída, I_{K1} é a corrente

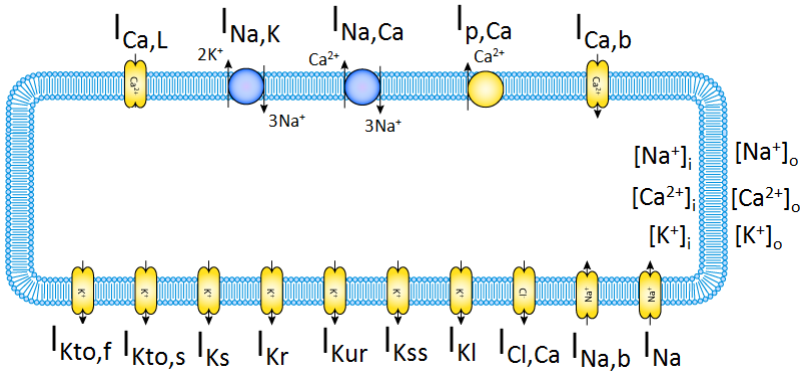

Figura 2.7: Representação da célula modelada em BDK. Figura adaptada de [5]

K^+ retificadora de entrada, I_{Kr} e I_{Ks} são as correntes rápidas e lentas de K^+, I_{Kur} é a corrente K^+ de retificação ultra-rápida, I_{Kss} IKss é a corrente $K+$ do estado estacionário, $I_{Cl,Ca}$ é a corrente cloro-cálcio e I_{stim} é a corrente de estímulo aplicada no tecido.

O, V, Ca_i, Na_i, O_{Na}, NaK, ato_f, ito_f, ato_s, ito_s, nKs, aur, iur, aKn, $iKss$, O_K, Ki são variáveis modeladas por EDOs.

2.5 O modelo de ten Tusscher e Panfilov

O modelo de ten Tusscher e Panfilov[15](TTP) é um modelo de uma célula do tecido ventricular humano, baseado em 19 EDOs que descrevem as correntes iônicas de sódio Na^+, potássio K^+ e cálcio Ca^{+2}. A descrição de I_{ion} é dada por:

$$\begin{aligned} I_{ion} = I_{Na}(m,h,j,V) + I_{K1}(V) + I_{to}(r,s,V) + I_{Kr}(Xr1,Xr2,V) \\ +I_{Ks}(Xs,V) + I_{CaL}(d,f,f2,fCass,Ca_ss,V) + I_{NaCa}(V,Na_i,Ca_i) \\ +I_{NaK}(Na_i,V) + I_{pCa}(Ca_i) + I_{pK}(V) + I_{bCa}(V) + I_{bNa}(V) + I_{stim}() \end{aligned} \tag{2.29}$$

Onde I_{Na} é a corrente rápida de sódio Na^+, I_{K1} é corrente de potássio K^+ de retificação de entrada, I_{to} é corrente K^+ de saída transiente, I_{Kr} e I_{Ks}são as correntes rápidas e lentas de K^+, I_{CaL} é o corrente Ca^{2+} tipo L, I_{NaCa} é o trocador Na^+ / Ca^{2+}, I_{NaK} é a bomba Na^+ / K^+, I_{pCa} e IpK são as correntes de platô de Ca^{2+} e K^+, I_{bCa} e $IbNa$ são as correntes de fundo Ca^{2+} e K^+ e I_{stim} é a corrente de estímulo aplicada no tecido.

m, h, j, V, r, s, $Xr1$, $Xr2$, d, f, $f2$, $fCass$, Ca_{ss}, Na_i, Ca_i e Xs são equações diferenciais ordinárias. A Figura 2.8 representa a membrana e os canais iônicos.

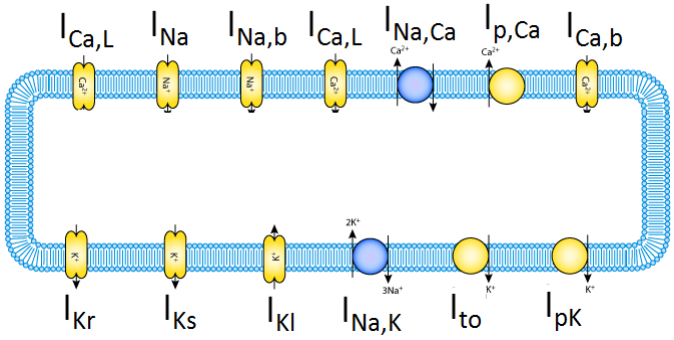

Figura 2.8: Representação da célula modelada em TTP. Figura adaptada de [5]

2.6 O modelo de Garny et al.

O modelo de Garny et al.[16](GRN) simula o comportamento de células cardíacas de coelho, reproduzindo a geração do potencial de ação do nó sinoatrial. O modelo utiliza 15 EDOs e a corrente iônica é dada pela Equação 2.30.

$$\begin{aligned} I_{ion} = I_{Na}(m,h,V) + I_{CaL}(f_L,d_L,V) + I_{CaT}(d_T,f_T,V) \\ +I_{to}(q,r,V) + I_{sus}(r,V) + I_{Kr}(P_i,V) + I_{Ks}(Xs,V) \\ +I_{f_{Na}}(y,V) + I_{f_K}(y,V) + I_{b_{Na}}(V) + I_{b_{Ca}}(V) \\ +I_{b_K}(V) + I_{NaCa}(V) + I_p() + I_{Cap}() \end{aligned} \tag{2.30}$$

Onde I_{Na} é corrente de sódio, I_{CaL} é a corrente de cálcio tipo L, I_{CaT} é a corrente de cálcio tipo T, I_{to} e I_{sus} são correntes sensíveis, I_{Kr} e I_{Ks} são as correntes rápidas e lentas de potássio, I_{fNa} e I_{fK} são as correntes ativas de hiperpolarização de sódio e potássio, I_{bNa}, I_{bCa} e I_{bK} são as correntes de fundo de sódio, cálcio e potássio, I_{NaCa} é a corrente de trocador sódio e cálcio, I_p é a corrente da bomba de sódio e potássio e I_{Cap} é a corrente de cálcio persistente.

m, h, V, f_L, d_L, d_T, f_T, q, r, y, P_i, Xs, f_Na, f_K, b_Ca são EDOs do sistema. A Figura 2.9 representa a célula modelada pela Equação 2.30.

2.7 O modelo de Noble et al.

O modelo de Noble et al.[17] (NBL) é um modelo matemático do potencial de ação ventricular, com 22 EDOs. Ele estuda a influência da troca dos íons de sódio e cálcio na

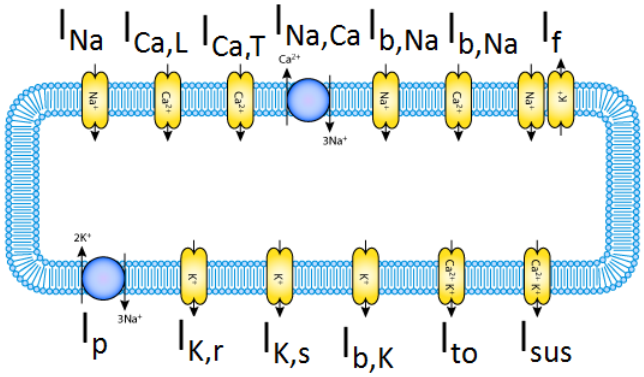

Figura 2.9: Representação da célula modelada em GRN. Figura adaptada de [5]

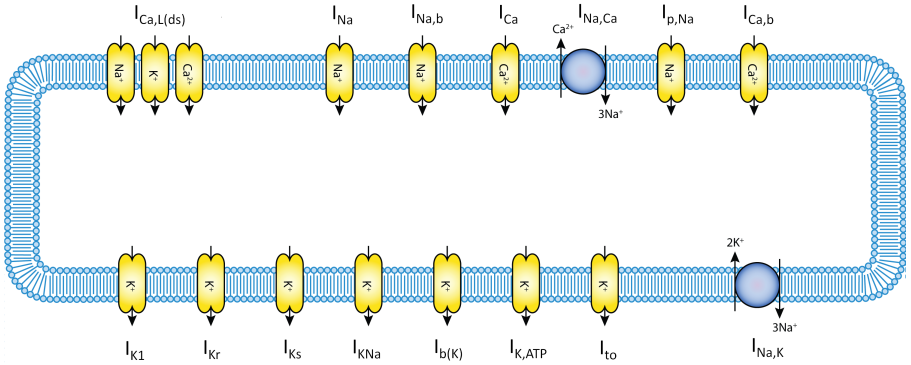

Figura 2.10: Representação da célula modelada em NBL. Figura adaptada de [5]

propagação do potencial de ação. A corrente iônica é dada pela Equação 2.31.

$$\begin{aligned} I_{Ion} &= I_{K1}(V) + I_{to}(s, r, V) + I_{Kr}(xr1, xr2, V) \\ &+ I_{Ks}(V) + I_{NaK}(Na_i) + I_{Na}(m, h, V) + I_{bNa}(V) \\ &+ I_{pNa}(V) + I_{CaL}(d, f, Ca_i, f2, K_i, Na_i, f2ds, V) \\ &+ I_{NaCa}(Na_i, Ca_i, Ca_ds, V) + I_{bCa}(V) + I_{stim} \end{aligned} \tag{2.31}$$

Onde I_{K1} é a corrente de potássio independente do tempo, I_{to} é a corrente transiente de saída, I_{Kr} e I_{Ks} são as correntes rápidas e lentas de potássio de retificação rápida, I_{NaK} é a corrente de bomba de sódio/potássio, I_{Na} é a corrente rápida de sódio, I_{bNa} e I_{bCa} são as correntes de fundo de sódio e cálcio, I_{pNa} é a corrente persistente de sódio, I_{CaL} é a corrente de cálcio tipo L, I_{NaCa} é a corrente de trocador sódio/cálcio e I_{stim} é a corrente de estímulo.

V, s, r $xr1$, $xr2$, Na_i, m, h, d, f, K_i, $f2ds$, Ca_i, Ca_ds são variáveis governadas por EDOs. A Figura 2.10 mostra a representação da célula.

3 FERRAMENTAS COMPUTACIONAIS

Os modelos matemáticos do potencial de ação (PA) de células cardíacas são tipicamente descritos por sistemas de Equações Diferenciais Ordinárias (EDOs). Modelos simples, menos realistas, podem ser resolvidos rapidamente. Porém, modelos complexos, com muitas equações, são computacionalmente custosos. Este problema se agrava quando são necessárias simulações longas, que reproduzem o funcionamento da célula por longos períodos de tempo. Este trabalho aborda somente modelos que reproduzem o funcionamento de uma única célula, porém existem simulações de tecidos, onde é necessário que vários modelos celulares sejam resolvidos, o que pode levar horas, ou até mesmo dias, dependendo do tamanho do tecido simulado.

Métodos numéricos e técnicas computacionais eficientes são de suma importância para a simulação de um modelo em tempo hábil e para produzir resultados com baixos índices de erro. Os métodos apresentados a seguir são o Método de Euler, Método de Runge-Kutta de Segunda Ordem e Método de *Backward differentiation formulas* (BDF). Logo após será feito um breve estudo sobre a acurácia e estabilidade de equações diferenciais ordinárias e os métodos utilizados para resolvê-las.

Também serão apresentadas a linguagem CellML - que é utilizada para representar modelos biológicos, a ferramenta AGOS, que resolve modelos descritos em CellML e é o foco deste trabalho, a ferramenta PyCML, que também resolve modelos descritos em CellML e fornece código-fonte com as técnicas de Avaliação Parcial e *LookUp Tables* e por último, OpenMP, que é um grupo de diretivas para programação paralela.

3.1 Métodos Numéricos

As células cardíacas são usualmente modeladas na forma de um sistema de Equações Diferenciais Ordinárias (EDOs), que são equações que envolvem derivadas das funções com apenas uma variável independente, geralmente a variável t, representando o tempo.

Os sistemas de EDOs têm a seguinte forma:

$$\begin{bmatrix} y_0'(t) \\ y_1'(t) \\ \vdots \\ y_n'(t) \end{bmatrix} = \begin{bmatrix} f_0(t, y(t)) \\ f_1(t, y(t)) \\ \vdots \\ f_n(t, y(t)) \end{bmatrix} \tag{3.1}$$

Onde t é uma variável independente real, $y : R \rightarrow R^n$ é uma função vetorial de t, $f : R^{n+1} \rightarrow R^n$ é chamada função "lado direito" (LD) do sistema, e y_i' é a derivada em relação a t, e também pode ser escrita como $\frac{dy_i(t)}{dt}$. A função LD é não-linear e na maioria dos casos não é possível determinar uma solução analiticamente. Portanto, métodos numéricos são necessários para obtenção de uma solução aproximada para y. Uma solução analítica é uma fórmula encontrada capaz de computar o valor da solução em qualquer ponto t. Já a solução numérica de uma EDO é uma tabela de valores aproximados da solução em um conjunto discreto de pontos. Estas soluções são encontradas através de simulações do comportamento do sistema de EDOs, onde as aproximações são obtidas iterativamente, ocorrendo incrementos discretos dentro do intervalo em que a solução é procurada.

O processo de incrementar a solução de um ponto discreto para o seguinte, pode acarretar em erro numérico, o que significa que a solução numérica não será exatamente igual à solução analítica. Para determinar se o erro numérico aumentará ou diminuirá, pode-se avaliar a estabilidade do sistema, tema que será abordado na Seção 3.1.7.

Outra caracterísca a ser destacada é que uma EDO não determina uma única solução, mas sim uma família de soluções. Para que seja encontrada uma solução específica, deve-se determinar uma condição inicial, na forma $y(t_0) = y_0$. Por isso, estas equações são conhecidadas como Problema de Valor Inicial (PVI) . Maiores detalhes sobre EDOs podem ser encontrados em [19, 18].

3.1.1 Equações diferenciais do tipo stiff

Um sistema de EDOs pode ser considerado *stiff* se o autovalores da matriz Jacobiana do lado direito das equações possui magnitudes muito diferentes. Como o Jacobiano varia com o decorrer do tempo, o sistema pode ser *stiff* em alguns intervalos e em outros, não. EDOs também podem ser classificadas qualitativamente como *stiff* quando o tamanho do

passo de tempo é determinado por questões de estabilidade e não por precisão[20].

Os métodos numéricos explícitos podem ser ineficientes para resolver equações *stiff*, sendo adequado escolher métodos implícitos, que são mais difíceis de programar e mais caros computacionalmente, porém mais estáveis.[18].

3.1.2 Método de Euler Explícito

O Método de Euler (ELR) é o mais simples e pode ser obtido a partir da Série de Taylor:

$$y(t+h) = y(t) + y'(t)h + \frac{y''(t)}{2}h^2 + \dots \tag{3.2}$$

Pela Equação 3.1, substitui-se $y'(t)$ e então:

$$y(t+h) = y(t) + f(t, y(t))h + \frac{y''(t)}{2}h^2 + \dots . \tag{3.3}$$

Para encontrar o método de Euler, consideram-se apenas os dois primeiros termos, descartando os termos com derivada igual ou superior à segunda, ou seja, $y(t+h) = y(t) + f(t, y(t))h$. Como $y(t)$ é a solução no instante t e $y(t+h)$ é a solução no instante seguinte, o método para resolver uma única equação pode ser escrito da seguinte maneira:

$$y_{k+1} = y_k + f(t, y_k)h \tag{3.4}$$

As soluções serão encontradas no conjunto discreto de pontos $[t_0, t_1, t_2, \dots, t_n]$, em que $t_{k+1} = t_k + h$, onde h é chamando de passo de tempo.

3.1.3 Método de Runge-Kutta de segunda ordem

O Método de Runge-Kutta de segunda ordem (RK2), também conhecido como Método de Heun, é obtido ao se considerar os termos da Série de Taylor até a segunda derivada:

$$y(t+h) = y(t) + y'(t)h + \frac{y''(t)}{2}h^2 \tag{3.5}$$

Para se determinar y'' utiliza-se a regra de cadeia, considerando que $y' = f(t, y)$:

$$y'' = f_t(t, y) + f_y(t, y)y' = f_t(t, y) + f_y(t, y)f(t, y) \tag{3.6}$$

Onde f_y e f_t são derivadas parciais de f em relação a y e a t, respectivamente. O cálculo das derivadas parciais pode ser complicado para equações de maiores ordens, porém é possível computar estes valores através de vários cálculos de f entre t_k e t_{k+1}. Para tal, deve-se obter a série de Taylor com duas variáveis para $f(t + h, y + hf)$:

$$f(t + h, y + hf) = f + hf_t + hf_y f + \dots \tag{3.7}$$

De onde encontra-se uma aproximação para y'' em função de duas avaliações de f:

$$f_t + f_y f \approx \frac{f(t + h, y + hf) - f(t, y)}{h} \tag{3.8}$$

Substituindo a Equação 3.8 em 3.5, finalmente encontra-se a equação do Método de Heun para uma única equação:

$$y(t + h) = y(t) + y'(t)h + \frac{f(t + h, y + hf) - f(t, y)}{2}h \tag{3.9}$$

Que pode ser escrito como:

$$\begin{aligned} y_{k+1} &= y_k + \frac{1}{2}(k_1 + k_2) \\ k_1 &= f(t_k, y_k)h \\ k_2 &= f(t_k + h, y_k + k_1)h \end{aligned} \tag{3.10}$$

O método em si não realiza o cálculo da derivada segunda, mas computa duas vezes o lado direito - a função f, para aproximar este valor.

3.1.4 Método de Euler Implícito

O método de Runge-Kutta de segunda ordem e o método de Euler são classificados como explícitos, o que significa que ambos avaliam a função f em t_k e y_k ($f(t_k, y_k)$) para encontrar o valor de y_{k+1}. Ou seja, y_k já foi computado na iteração anterior. Estes

métodos são simples de serem implementados, mas possuem uma região de estabilidade limitada, o que exige que o passo de tempo (h) seja menor.

Uma alternativa são os métodos implícitos, que são mais complexos de implementar, mas possuem a região de estabilidade maior.

O Método de Euler Implícito é dado por:

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1}) \tag{3.11}$$

O método é classificado como implícito pois é necessário avaliar a função f com o argumento y_{k+1} antes deste valor ter sido computado. Isso sigfica que para que a Equação 3.11 seja satisfeita, deve-se obter y_{k+1} resolvendo um sistema não linear.

3.1.5 Métodos Multi-passo

Métodos multi-passo também são conhecidos como métodos com memória, pois utilizam informações dos passos anteriores, $t_0, t_1, \ldots, t_n$ para encontrar a solução no instante t_{n+1}. Métodos multi-passo lineares possuem a seguinte forma:

$$y_{k+1} = \sum_{i=1}^{n} \alpha_i y_{k+1-i} + h \sum_{i=0}^{q} \beta_i f(t_{k+1-i}, y_{k+1-i}) \tag{3.12}$$

Onde os parâmetros α_i e β_i são determinados por interpolação polinomial. Se $\beta_0 = 0$, o método é explícito, caso contrário, é implícito.

Existe uma grande variedade destes métodos, com diferentes características de estabilidade e precisão. Os mais conhecidos são os Método de Adams-Moulton[18], adequados para problemas não *stiff*, e o método BDF (*Backward Differentiation Formulas*), muito utilizado para resolver problemas *stiff* [21].

3.1.5.1 O método BDF - *Backward Differentiation Formulas*

O método BDF é um método implícito e consequentemente, é mais caro computacionalmente. Porém, devido a sua estabilidade, é uma escolha comum para resolver sistemas *stiff* relacionados com a eletrofisiologia cardíaca [22]. O objetivo deste método é encontrar y_{n+1} usando valores passados, o que significa que $f(t_{n+1}, y_{n+1})$ é aproximada por uma

combinação linear das soluções: $(y_{n+1}, y_n, y_{n-1}, \dots)$[2]. O método possui a forma:

$$y'(t+h) = \frac{y_{n+1} - y_n}{h} \tag{3.13}$$

O método BDF de ordem n é obtido fazendo $q = 0$ na Eq. 3.12:

$$y_{k+1} = \sum_{i=1}^{n} \alpha_i y_{k+1-i} + h\beta_0 f(t_{k+1}, y_{k+1}) \tag{3.14}$$

Onde os parâmetros β_0 e $(\alpha_1, \dots, \alpha_n)$ podem ser determinados pelo método dos coeficientes indeterminados e maiores detalhes podem ser encontrados em [18]. Claramente, deve-se solucionar um sistema de equações não-lineares, para se que se encontre y_{n+1}. O sistema tem a seguinte forma:

$$G(y_{k+1}) = y_{k+1} - \sum_{i=1}^{n} \alpha_i y_{k+1-i} - h\beta_0 f(t_{k+1}, y_{k+1}) = 0 \tag{3.15}$$

O método de Newton é o mais utilizado para a busca das raízes deste sistema [2], o que aumenta o custo computacional dos métodos implícitos. Porém, por serem mais estáveis, os métodos implícitos aceitam passos de tempo maiores do que os métodos explícitos, o que diminui o tempo de execução.

Métodos multi-passo não são capazes de iniciar o processo de resolução sozinhos, pois nas primeiras iterações não há um histórico de soluções, sendo necessária a utilização de outro método até que exista um número suficiente de soluções para que o método multi-passo se inicie.

3.1.6 Acurácia

Os métodos numéricos produzem resultados diferentes da solução analítica. Esta diferença é chamada de erro, que pode ocorrer por duas razões. A primeira é o erro por arredondamento, que ocorre devido à precisão finita da aritmética de ponto flutuante. A outra razão é o erro de truncamento, também chamado de erro de discretização.

O erro de truncamento de um método numérico, que está no passo k, pode ser dividido em erro de truncamento local e erro de truncamento global. O erro local L_k é o erro

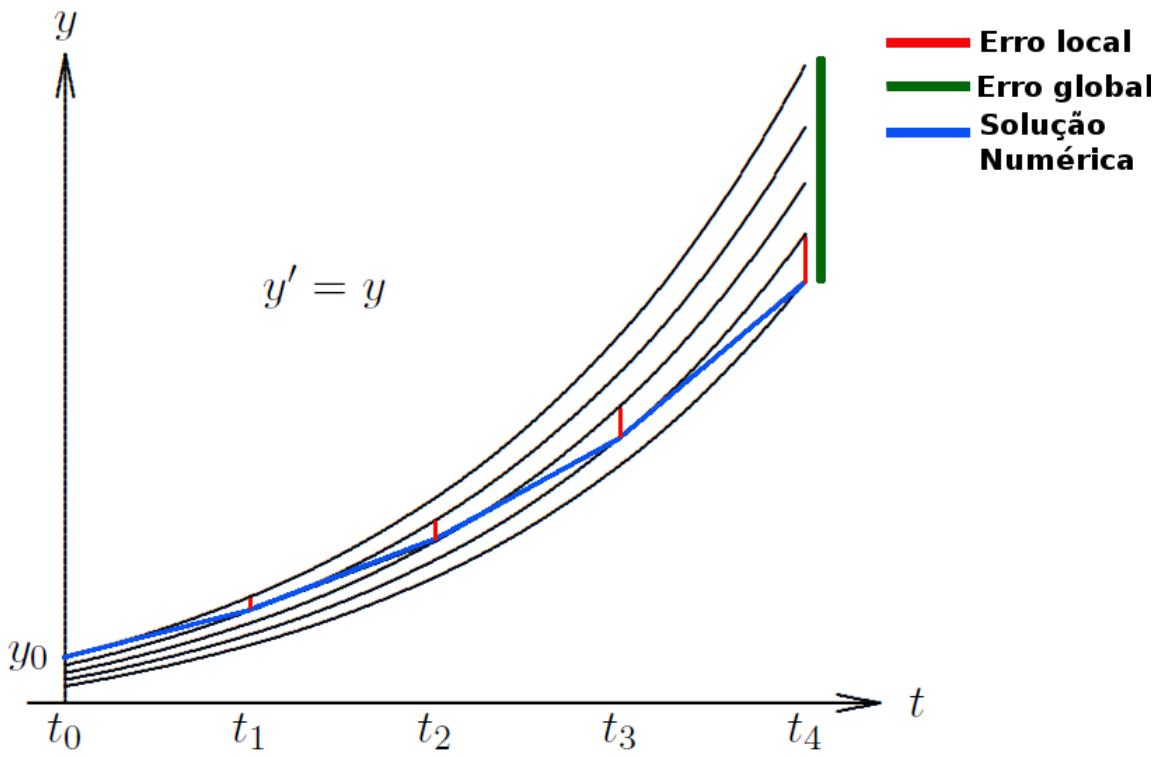

Figura 3.1: Erro local e global. Figura adaptada de [18]

causado em uma iteração do método numérico. L_k é dado por:

$$L_k = y_k - u_{k-1}(t_k) \tag{3.16}$$

Onde y_k é a solução computada pelo método no instante t_k e u_{k-1} é um membro da família de soluções da EDO que passa pelo ponto anterior $(t_{k-1},\ y_{k-1})$. O erro global de truncamento E_k é a diferença entre a solução computada e solução analítica, ambas determinadas com o valor inicial y_0 no instante t_0. É dado por:

$$E_k = y_k - u_0(t_k) = y_k - y(t_k) \tag{3.17}$$

A Figura 3.1 ilustra a diferença entre os erros. Por exemplo, no instante t_4, o erro global é a diferença entre a solução computada pelo método, em azul, e a solução iniciada em y_0. O erro local é dado pelo diferença entre a solução computada e o membro da família de soluções que passou por $(t_3,\ y_3)$, sendo que y_3 é a solução encontrada pelo método em t_3.

A acurácia de um método numérico é de ordem p se:

$$L_k = \mathcal{O}(h^{p+1}) \tag{3.18}$$

Isso ocorre porque o somatório de erros locais de t_0 até t_k, que é uma aproximação do erro global, é dado por:

$$\frac{t_k - t_0}{h}\mathcal{O}(h^{p+1}) = \mathcal{O}(h^p) \tag{3.19}$$

Onde h é o passo de tempo.

3.1.6.1 Acurácia do Método de Euler Explícito

Para calcular a ordem de acurácia do Método de Euler, deve-se subtrair o método da série de Taylor (Eq. 3.3), da qual ele é obtido ao se ignorar os termos de segunda ou maior ordem:

$$y_{k+1}^T - y_{k+1}^E = (y_k^T + f(t, y_k^T)h + \mathcal{O}(h^2)) - (y_k^E + f(t, y_k^E)h) \tag{3.20}$$

Ao se considerar que não houve erro até a t_k, ou seja, $y_k^T = y_k^E$, obtém-se que o erro local em t_{k+1} é $\mathcal{O}(h^2)$.

O erro global no intervalo [t_0,t_k] é dado por:

$$\frac{t_k - t_0}{h}\mathcal{O}(h^2) = \mathcal{O}(h) \tag{3.21}$$

O que significa que o Método de Euler é de primeira ordem.

3.1.6.2 Acurácia do Método de Runge-Kutta de segunda ordem

Para se obter a ordem de acurácia para o Método de Runge-Kutta de segunda ordem, deve-se proceder da mesma maneira em que a ordem do método de Euler foi encontrada. A diferença é que o Método de Runge-Kutta não descarta o termo com a segunda derivada da Série de Taylor, obtendo então, erro local $\mathcal{O}(h^3)$ e erro global $\mathcal{O}(h^2)$. Portanto, este método é de segunda ordem.

3.1.7 Estabilidade

As EDOs podem ser classificadas como estáveis, instáveis e neutras[18]. Quando os membros da família de soluções se afastam com o tempo, a equação é classificada como instável. Se as soluções se aproximam, a equação é estável. Caso as soluções não se afastem e nem se aproximem, ou seja, elas não convergem e nem divergem, a equação é neutra.

A estabilidade das equações está relacionada com a sensibilidade das soluções da EDO à pertubações. Em uma equação estável, uma perturbação a uma solução será diminuída com o tempo porque as soluções estão convergindo. Caso uma pertubação seja feita em um sistema instável, ela aumentará com o tempo, pois as soluções divergem. Se o sistema é neutro, uma pertubação moverá o sistema para uma nova posição. Uma equação pode apresentar diferentes comportamentos estáveis ou instáveis em diferentes intervalos. Este conceito qualitativo de estabilidade para uma EDO $y' = f(t, y)$ pode ser expresso quantitativamente pelo Jacobiano $J_f(t, y)$, que é dado por:

$$\{J_f(t, y)\}_{i,j} = \frac{\partial f_i(t, y)}{\partial y_j} \tag{3.22}$$

Onde $\{J_f(t, y)\}_{i,j}$ é o elemento da matriz Jacobiana com coordenadas (i, j), $f_i(t, y)$ é a i-ésima linha da função LD e y_j é o j-ésimo elemento $y(t)$.

Em sistemas de EDOs, se algum dos autovalores do Jacobiano possuir parte real positiva, a equação é instável. Se todos os autovalores possuírem partes reais negativas, a equação é estável. Caso existam um ou mais autovalores com partes reais iguais a zero e todos os demais autovalores possuírem partes reais negativas, a equação é neutra. Os elementos do Jacobiano dependem de y e t, e portanto, os autovalores podem variar no tempo. Para avaliar apenas uma equação, pode-se apenas observar o sinal do Jacobiano, que será escalar. Consequentemente, a estabilidade do sistema pode variar de acordo com a região.

A análise qualitativa da estabilidade de um método numérico pode ser feita de maneira análoga à análise feita para sistemas de EDOs. Uma equação é estável quando suas soluções não divergem com o tempo. Já um método numérico é classificado como estável se pequenas pertubações não causarem divergência entre as soluções. A divergência de soluções numéricas pode ocorrer devido à instabilidade da EDO que está sendo resolvida, ou por causa do próprio método numérico, independente da equação ser instável ou não.

3.1.8 Estabilidade do Método de Euler Explícito

O Método de Euler é derivado a partir de uma aproximação da Série de Taylor. Na Seção 3.1.6.1 foi demonstrado que o erro global é aproximado por um somatório dos erros locais. O erro global obtido ao se avançar do instante t_k para t_{k+1} é estimado ao se subtrair a Série de Taylor pelo Método de Euler:

$$\begin{aligned} E_{k+1} &= y_{k+1}^T - y_{k+1}^E = (y_k^T + f(t, y_k^T)h + \mathcal{O}(h^2)) - (y_k^E + f(t, y_k^E)h) \\ &= y_k^T - y_k^E + f(t, y_k^T)h - f(t, y_k^E)h + \mathcal{O}(h^2) \\ &= y_k^T - y_k^E + [f(t, y_k^T) - f(t, y_k^E)]h + \mathcal{O}(h^2) \end{aligned} \tag{3.23}$$

O Teorema do Valor Médio, ou Teorema de Langrange afirma que para uma função contínua f no intervalo $[a, b]$, existe um ponto $c \in (a, b)$ tal que $f'(c) = \frac{f(b)-f(a)}{b-a}$. Ao se aplicar este teorema à Equação 3.23, obtém-se:

$$\begin{aligned} J(\xi) &= \frac{f(t, y_k^T) - f(t, y_k^E)}{y_k^T - y_k^E} \\ f(t, y_k^T) - f(t, y_k^E) &= J(\xi)(y_k^T - y_k^E) \end{aligned} \tag{3.24}$$

Onde ξ é um valor desconhecido entre y_k^T e y_k^E. Substituindo a Eq. 3.24 em 3.23, encontra-se o erro global em t_{k+1}:

$$\begin{aligned} E_{k+1} &= y_k^T - y_k^E + (f(t, y_k^T) - f(t, y_k^E))h + \mathcal{O}(h^2) \\ &= y_k^T - y_k^E + J(y_k^T - y_k^E)h + \mathcal{O}(h^2) \\ &= E_k(1 + Jh) + \mathcal{O}(h^2) \end{aligned} \tag{3.25}$$

Pode-se considerar o erro local de ordem $\mathcal{O}(h^2)$ como desprezível. Portanto, o erro global é propagado a cada passo por um fator de $(1+J(\xi)h)$. Para uma única EDO, se $|1+Jh| < 1$, ou seja, $-2 < Jh < 0$, o erro não aumentará e o método será estável. Caso contrário, o método é instável e o erro aumentará.

O método pode ser instável caso a própria equação seja instável, o que ocorre quando o Jacobiano é positivo ($J > 0$), ou pode ser instável, quando a equação é estável ($J < 0$) mas $h > \frac{-2}{J}$.

3.1.9 Estabilidade do Método de Heun

A estabilidade do Método de Runge-Kutta de segunda ordem ou Método de Heun pode ser determinada considerando o caso linear:

$$y' = \lambda y \tag{3.26}$$

Onde λ é uma constante.

Ao se aplicar o método de Heun (Equação 3.10) a esta equação, obtém-se:

$$\begin{aligned} y_{k+1} &= y_k + \frac{1}{2}(k_1 + k_2) \\ k_1 &= \lambda y_k h \\ k_2 = \lambda(y_k + k_1)h &= \lambda h y_k + (\lambda h)^2 y_k \end{aligned} \tag{3.27}$$

De onde encontra-se:

$$y_{k+1} = y_k(1 + \lambda h + \frac{(\lambda h)^2}{2}) \tag{3.28}$$

Isto significa que a cada passo de tempo o erro é propagado por um fator de $\|1+\lambda h+\frac{(\lambda h)^2}{2}\|$. O método é considerado estável se:

$$\|1 + \lambda h + \frac{(\lambda h)^2}{2}\| < 1 \tag{3.29}$$

Da Equação 3.29 percebe-se que $-2 < \lambda h < 0$. Ou seja, ocorrem as mesmas condições de estabilidade do método de Euler.

3.2 CellML

A linguagem CellML[5] é um padrão aberto baseado em XML (eXtensible Markup Language) [7]. O objetivo é representar modelos matemáticos em computadores e permitir que eles sejam compartilhados, mesmo que sejam desenvolvidos e simulados por diferentes plataformas computacionais, além de permitir que componentes de um modelo sejam reaproveitados em outros, facilitando a construção de novos modelos.

Inicialmente, a linguagem foi criada apenas para a descrição de modelos biológicos,

```
<units name="microF_per_cm2">
        <unit prefix="micro" units="farad"/>
        <unit exponent="-2" prefix="centi" units="metre"/>
</units>
```

Figura 3.2: Declaração de unidades em CellML

porém ela pode ter aplicações mais genéricas e ser empregada em outros campos do conhecimento. CellML inclui informações sobre a estrutura do modelo, ou seja, como as suas partes estão organizadas e relacionadas entre si. Também inclui informações adicionais que descrevem o arquivo XML, chamadas de metadados. Tais descrições permitem que usuários façam buscas por um modelo específico em um banco de dados ou repositório. As equações matemáticas são descritas em uma outra linguagem chamada MathML[6], que também é baseada em XML e é embutida dentro do próprio CellML.

Os documentos escritos em CellML são compostos por uma série de elementos que representam unidades de medida, componentes, variáveis, equações e conexões entre componentes[23, 24]. Em XML, as estruturas de marcação são chamadas de *tags*, e são caracterizadas por começarem com o caracter $<$ e terminarem com $>$. As principais *tags* da linguagem CellML serão descritas a seguir.

3.2.1 Units

As grandezas usadas em um modelo devem ser declaradas na seção de código delimitada pelas *tags* $< units >$ e $< /units >$. Isso permite que sejam feitas operações com variáveis de unidades diferentes, se estas forem compatíveis. Por exemplo, é possível que um programa que simule modelos converta e compare uma variável declarada como Volts com outra variável declarada como miliVolts. Esta característica garante robustez e reusabilidade dos componentes e dos modelos CellML. A linguagem fornece um conjunto de unidades padrão, composto pelas grandezas do Sistema Internacional de Unidades (SI) e por uma grandeza chamada *dimensionless*, que indica que a variável não possui unidade.

A Figura 3.2 mostra a definição de uma nova unidade que é criada a partir de unidades do SI. A grandeza a ser definida é microFarad por centímetro quadrado ($microF/cm^2$). Cria-se uma entrada $<unit>$ para cada unidade do SI com as informações necessárias para a conversão, como a unidade padrão (Farad e metro), os prefixos (micro e centi) e no caso do cm^2, deve-se colocar um expoente para conversão de metro para cm^2, que são respectivamente unidades de comprimento e de área.

3.2.2 Component

Os modelos são descritos como componentes interligados, onde um componente é uma unidade funcional que pode representar estruturas físicas, espécies, organelas celulares, eventos, entre outros. Componentes contêm variáveis e equações matemáticas e permitem que o desenvolvimento do modelo seja modularizado, ou seja, a construção de componentes é feita independentemente. Quando é necessário que dois componentes troquem informações entre si, as variáveis de cada uma são mapeadas através de uma interface bem definida de conexão.

3.2.3 Variable

Variáveis são declaradas dentro de componentes, devem possuir unidades, um valor inicial e devem especificar se seu valor vai ser utilizado ou calculado por outro componente.

3.2.4 Math

A descrição das equações matemáticas dos modelos é feita pela linguagem MathML[6], que contém *tags* para sinais aritméticos, relacionais e lógicos, equações diferenciais e funções diversas como funções trigonométricas, logaritmos, entre outros.

A Figura 3.3 mostra a definição do componente "membrana" que ocorre em todos os modelos utilizados neste trabalho. Todas as variáveis utilizadas nas equações de um componente devem ser declaradas, porém neste exemplo, para simplificação, apenas duas são exibidas. A primeira variável é o potencial de ação, e o atributo "*name*" indica que ela será identificada pelo nome V, "*initial_value*" é o valor inicial desta variável, "*public_inteface*"="*out*" determina que esta variável será computada dentro deste componente e será visível para outros e "*units*" é a unidade de grandeza, que deve ser declarada caso não faça parte do SI. A segunda variável, *time*, possui o atributo "*public_inteface*"="*in*", o que significa que ela é computada em outro componente e seu conteúdo será utilizado por este componente. Por último, dentro da *tag* <*math*> deve-se descrever as equações. Este exemplo mostra a equação $\frac{\partial V}{\partial t} = \frac{-(-I_{Stim}+I_{Na}+I_K+I_L)}{C_m}$

```
<component name="membrane">
    <variable initial_value="-75" name="V" public_interface="out" units="millivolt"/>
    <variable name="time" public_interface="in" units="millisecond"/>
    <!--  As outras variáveis serão omitidas    -->
    <math >
        <apply> <eq/>
            <apply> <diff/>
                <bvar><ci>time</ci></bvar>
                <ci>V</ci>
            </apply>
            <apply> <divide/>
                <apply> <minus/>
                    <apply> <plus/>
                        <apply> <minus/>
                            <ci>i_Stim</ci>
                        </apply>
                        <ci>i_Na</ci>
                        <ci>i_K</ci>
                        <ci>i_L</ci>
                    </apply>
                </apply>
                <ci>Cm</ci>
            </apply>
        </apply>
    </math>
</component>
```

$$\frac{\partial V}{\partial t} = \frac{-(-I_{Stim}+I_{Na}+I_K+I_L)}{C_m}$$

Figura 3.3: Declaração de um componente em CellML

```
<connection>
        <map_components component_1="membrane" component_2="environment"/>
        <map_variables variable_1="time" variable_2="time"/>
</connection>
```

Figura 3.4: Conexão de variáveis em CellML

3.2.5 Connection

Quando uma variável é calculada em um componente e utilizada por outro, deve-se criar uma conexão entre os componentes, indicando quais variáveis estarão enviando ou recebendo valores. O conjunto completo de mapeamentos entre variáveis de um par de componentes constitui uma conexão. Apenas uma conexão entre dois componentes é permitida. Após a especificação de quais componentes estão se conectando, deve-se definir quais pares de variáveis estão se ligando. As variáveis devem ser compatíveis com esta ligação, o que significa que em um par uma variável deve possuir a interface "*out*" e a outra "*in*".

Para o exemplo da Figura 3.3, a variável *time* do componente *membrane* é calculada em outro componente, então uma conexão é necessária. A Figura 3.4 ilustra este mapeamento com a variável *time* do componente *environment*.

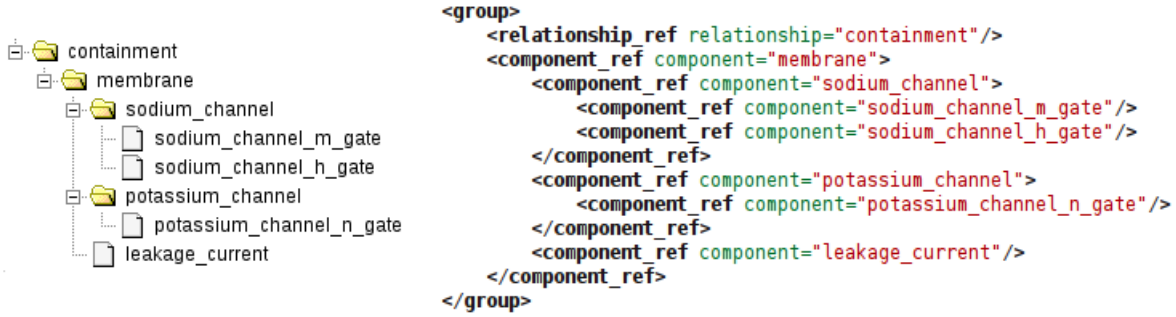

Figura 3.5: Hierarquia do modelo de Hodgkin e Huxley - representação gráfica e codificação em CellML

3.2.6 Group

Organiza os componentes de um modelo em uma estrutura hierárquica, podendo representar as relações lógicas ou a estrutura física. Os agupamentos são codificados como duas árvores de *tags* contendo o nome dos componentes. Em cada árvore um componente pode aparecer apenas uma vez. As relações lógicas são chamadas de *encapsulation* e permitem que um componente pai esconda detalhes de componentes filhos do restante do modelo. Isto simplifica o código, já que as variáveis de componentes filhos não precisam ser mapeadas através da *tag connection*. As relações físicas são chamadas de *containment* e são utilizadas para descrever, sem detalhes geométricos, a localização de componentes filhos dentro de um componente pai.

A Figura 3.5 mostra a representação gráfica e a codificação em CellML da árvore da estrutura física. A membrana é o componente raiz, onde estão localizados os canais de sódio e potássio, além da corrente de fuga. O canal de sódio possui dois componentes, que são as comportas “m” e “h”, enquanto o canal de potássio possui a comporta “n”.

3.3 Ferramentas

Existe uma crescente disponibilidade de ferramentas e técnicas para edição, validação, compartilhamento e execução de modelos CellML[23]. A seguir serão apresentadas as ferramentas utilizadas neste trabalho.

3.3.1 AGOS

O AGOS - *Application program interface Generator for Ordinary differential equation Solution* [25] é uma aplicação *Web* que possui ferramentas para simulação de modelos baseados em equações diferenciais ordinárias, descritas em CellML. O AGOS pode ser encontrado no Portal do Laboratório de Fisiologia Computacional - Fisiocomp[1]. Através deste portal, é possível submeter um modelo previamente escrito em CellML, que será transformado em uma API orientada a objetos em C++ [26, 8].

API significa *Application Program Interface*, que é um conjunto de regras e especificações que um *software* deve seguir para se comunicar com outro. Ou seja, API é uma interface entre dois programas que facilita a interação entre eles[27].

Para resolver o sistema de equações diferenciais ordinárias a API gerada pelo AGOS contém o código-fonte com as equações, métodos numéricos, inicialização de variáveis e escrita dos resultados em arquivos. A API pode ser copiada ou executada através do Portal Fisiocomp.

Durante o processo de transformação do arquivo CellML em código C++ são extraídas informações do modelo, como o nome das equações diferenciais, parâmetros, constantes e variáveis independentes, além das unidades. Com estas informações, monta-se dinamicamente uma página *Web* para a execução deste modelo, com campos para o lançamento de valores iniciais das equações e outros parâmetros para a simulação, como o passo de tempo, tempo de simulação, método numérico e como os resultados da simulação serão visualizados. Os campos são exibidos com os nomes de variáveis contidos no arquivo CellML, juntamente com a respectiva unidade.

Após a configuração da simulação, o modelo pode enfim ser executado e em seguida seus resultados podem ser mostrados em forma de gráficos. O portal fornece uma interface amigável e oculta técnicas de programação, métodos numéricos e visualização de resultados, com o propósito de facilitar e agilizar o processo de simulação computacional, que pode ser dispendioso e demanda conhecimento destas técnicas.

A Figura 3.6 ilustra o fluxograma do AGOS. Em (1) um modelo descrito em XML é enviado para o tradutor, que extrai dados das equações e gera a API. Em (2), os dados são utilizados para montagem da interface de usuário, e em (3) a API é integrada ao simulador, que invoca as funções necessárias para a simulação. Através da interface, o

[1]Disponível em $http://www.fisiocomp.ufjf.br$

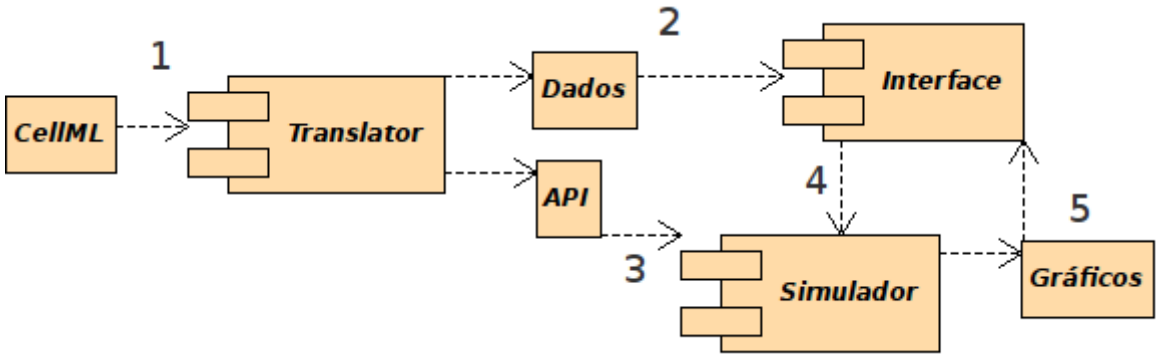

Figura 3.6: Fluxo de execução do AGOS

usuário dispara o simulador (4) que executa a API e gera gráficos com a resolução do sistema de equações, que são enviados de volta para a interface (5).

3.3.1.1 O tradutor

O tradutor do AGOS foi "implementado em C++ e faz uso de estruturas básicas e algoritmos para capturar as variáveis, parâmetros e equações que estão no arquivo CellML e traduzir para um código C++ executável"[26]. Ele foi desenvolvido na monografia de Amorim[26] e tem sido aprimorado através de vários trabalhos[8, 3, 25]. Os elementos extraídos das equações contidas no arquivo CellML são classificados como variáveis independentes, variáveis dependentes, variáveis auxiliares, parâmetros da equação, equações diferenciais e equações algébricas. Estes elementos são codificados e organizados na API, que contém métodos públicos - que são acessíveis por classes externas - "que retornam os valores de variáveis dependentes (getVariable), configuram o número de iterações e o intervalo de discretização (setup), configuram os parâmetros da equação (setParameter), calculam a solução numérica através do método Explícito de Euler (solveODE), ou através de métodos implícitos do módulo CVODE da biblioteca SUNDIALS"[26].

Já os métodos privados podem ser invocados apenas pela própria classe em que está contido. Um exemplo de método privado da API é o método que avalia as equações algébricas.

Na Figura 3.7 encontra-se o mapeamento entre elementos conceituais das EDOs e os métodos básicos da API. Os métodos da API são representados por retângulos e os elementos das EDOs, pelas formas semelhantes a uma elipse. A direção das setas define os relacionamentos de dependência. Por exemplo, "equações algébricas dependem de parâmetros, variáveis auxiliares, dependentes e independentes; o método SolveAE depende

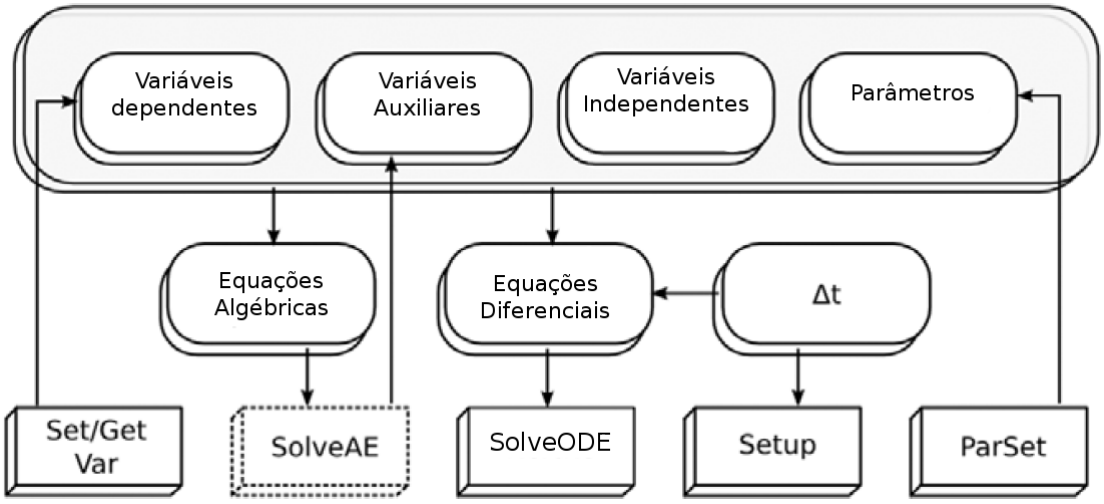

Figura 3.7: Mapeamento do sistema de EDOs para a API - Adaptada de [28]

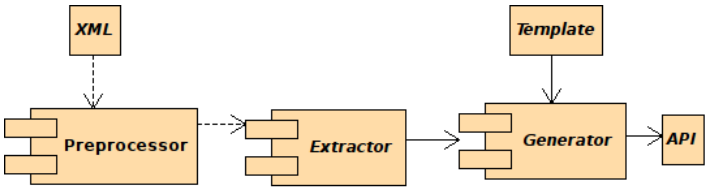

Figura 3.8: Arquitetura do tradutor

de equações algébricas; e por sua vez influencia as variáveis auxiliares"[26]. O módulo de tradução é subdividido em três partes: pré-processador, extrator e gerador. O pré-processador lê um arquivo XML e transforma o conteúdo do arquivo em um árvore, que então é enviada para a análise sintática e extrator, onde a árvore será percorrida, avaliada e terá os elementos das EDOs separados em estruturas de dados apropriadas. Na última etapa, o gerador de código usa um modelo genérico de código (*template*) e as informações extraídas para finalmente gerar a API. Este processo pode ser visualizado na Figura 3.8.

O AGOS realiza uma espécie de otimização em seu código, chamada de Avaliação Parcial (AP), que ocorre quando uma variável associada a uma equação é requisitada várias vezes por outras equações, no mesmo passo de tempo. Isso acontece principalmente quando o valor de uma equação é utilizado em diferentes componentes do CellML. Inicialmente, a equação seria recomputada todas as vezes que fosse acessada, em cada componente, produzindo o mesmo resultado, o que é desnecessário. Então, a otimização para este caso consiste em avaliar a equação apenas uma vez e armazená-la em uma variável, que será acessada toda vez que requisitada. Por exemplo, o potencial de equilíbrio do íon K^+ permanece inalterado dentro de um mesmo passo de tempo e é dado por $E_K = \frac{RT}{F} ln \frac{K_e}{K_i}$.

A variável E_k é utilizada pelo menos uma vez para cada uma das corrente iônicas que envolvem o íon K^+. Normalmente, E_K seria recalculada para cada corrente, reproduzindo o mesmo resultado de maneira desnecessária. A técnica de AP realiza apenas um cálculo desta variável para cada iteração, e cada corrente iônica acessa o valor de E_k na memória[2].

3.3.1.2 Sundials

Sundials (*SUite of Nonlinear and DIfferential/ALgebraic equation Solvers*) é uma biblioteca que contém diversos métodos numéricos para resolução de problemas de valor inicial de EDOs e sistemas algébricos não lineares que podem ser incorporados a um código existente[29]. O AGOS utilizou o módulo CVODE do Sundials que possui métodos para resolução de sistemas de EDOs dados na forma $y' = f(t, y)$. O método CVODE utilizado por este trabalho é o *Backward Differentiation Formulas* (BDF), que foi explicado na seção 3.1.5.1 e resolve problemas *stiff*. Também são oferecidos outros métodos como o de Adams-Moulton. A função que resolve equações através do método BDF utiliza passo de tempo adaptativo baseado no comportamento do erro local e permite a configuração de vários parâmetros para a resolução das equações, como a ordem do método e o número máximo de iterações do método de Newton, que é executado a cada iteração do método de integração. A cada iteração do método de Newton deve-se resolver um sistema linear onde é possível escolher entre os métodos direto para matrizes densas, direto para matrizes em banda, método de aproximação diagonal do Jacobiano e método iterativo do resíduo mínimo generalizado precondicionado(GMRES).

Maiores detalhes sobre a biblioteca Sundials podem ser encontrados em [29, 30]. A escolha dos métodos citados acima interfere fortemente no desempenho e acurácia na resolução de um sistema. Um estudo minucioso a respeito deste assunto foi realizado em [2].

3.3.2 PyCML

PyCml é uma aplicação desenvolvida em Python para validação de modelos e geração de código, disponível pela *Web* ou por linha de comando[9]. Ela é capaz de verificar a compatibilidade das unidades das variáveis, otimizar automaticamente o modelo com técnicas de Avaliação Parcial[31] e *LookUp Tables*[32, 33] e gerar código em C++. O código ge-

rado contém apenas as expressões do lado direito das equações não contendo métodos numéricos. O código gerado pelo PyCml foi projetado para ser uma parte integrante do ambiente de simulação Chaste[34], mas é possível utilizá-lo com métodos numéricos fornecidos pelo usuário. A seguir serão apresentadas as técnincas de Avaliação Parcial e *LookUp Tables* que serão utilizadas por este trabalho. A implementação destas técnicas no PyCml foram descritas em [35, 36].

3.3.2.1 Avaliação Parcial Fina

Avaliação parcial (AP) é uma técnica que pré-computa automaticamente partes de um dado programa em tempo de compilação, produzindo um novo programa especializado. Este novo programa deve produzir a mesma saída que o programa original produziria para uma mesma entrada[31]. Esta técnica analisa e classifica as expressões em estáticas ou dinâmicas. As expressões estáticas não mudam no decorrer da execução e podem ser computadas em tempo de compilação, enquanto as dinâmicas dependem de valores que mudam durante a execução do programa e devem ser computadas em tempo de execução. O objetivo é diminuir a frequência de cálculo de partes comuns no código, reorganizando-o[35].

Alguns compiladores são capazes de fazer otimizações, porém os programas de simulação de modelos podem envolver trechos de código grandes e complexos, impedindo o compilador de realizar melhorias em níveis desejáveis.

O PyCML pode melhorar códigos-fonte de equações de duas maneiras. A primeira forma de otimização de Avaliação Parcial do PyCML é a mesma do AGOS, e consiste em efetuar mudanças no código para evitar que expressões que geram resultados iguais sejam executadas repetidas vezes.

A segunda maneira de otimização é mais complexa e modifica a estrutura das equações e o AGOS não é capaz de realizar. Por exemplo, a equação $E_K = \frac{RT}{F} ln \frac{K_e}{K_i}$ utiliza as constantes R, T e F, o que sigfica que o resultado de $\frac{RT}{F}$ também será constante, então a avaliação parcial substitui esta conta pela constante que resulta destas operações. Esta mesma constante pode ser utilizada por outras equações, como E_{Na}, que também realiza o cálculo de $\frac{RT}{F}$. Como estas equações precisam ser computadas a cada iteração, esta redução no número de operações pode diminuir o tempo de execução total do modelo.

3.3.2.2 *Lookup Tables*

LookUp Tables (LUT)[32, 33] é um técnica computacional utilizada para pré-computar automaticamente expressões que seriam repetidamente calculadas. Esta técnica está implementada para expressões que apenas dependem do potencial transmembrânico V. O seguinte exemplo foi demonstrado em [36], para a equação:

$$\beta_m = 0.08e^{\frac{-V}{11}} \tag{3.30}$$

Em condições fisiológicas, o potencial V geralmente se encontra entre os valores de -100 mV e 50 mV. Assim, gera-se uma tabela T com os valores pré-computados de β_m dentro deste intervalo, a partir de interpolação linear entre duas entradas de T (T_i e T_{i+1}):

$$\beta_m = T_i + \frac{(T_{i+1} - T_i)(V - V_i)}{V_{i+1} - V_i} \tag{3.31}$$

Caso V saia do intervalo, pode-se considerar como erro e interromper a execução do programa ou então computa-se o valor com a equação original. O cálculo da equação do programa original é substituído por um acesso de memória na tabela T.

LUT aceleram a execução de modelos, porém, ao contrário de APs, as interpolações produzem erros, o que significa que a saída não será exatamente a mesma que o programa original produziria. Equações substituídas por LUT devem ser dependentes apenas da variável V e, para que esta troca seja eficiente, as equações devem possuir funções trigonométricas ou exponenciais que sejam computacionalmente caras para serem avaliadas.

3.4 Computação Paralela

Existem diversos problemas científicos que precisam realizar cálculos complexos sobre um grande volume de dados[37]. Tais problemas podem demorar muito tempo para que resultados sejam produzidos, tornando a utilização dos mesmos inviável. Um exemplo clássico são algoritmos de previsão de tempo que precisam gerar resultados a tempo de serem utilizados. Em alguns casos, como previsões mais detalhadas que envolvem grandes regiões, o cálculo pode demorar vários dias. Para que a execução do programa seja mais rápida e a previsão seja entregue a tempo, pode-se empregar computação paralela, que consiste em dividir problemas grandes em problemas menores para que as subdivisões

sejam executadas simultaneamente, com o objetivo de reduzir o tempo de execução. Esta técnica necessita de computadores paralelos que são aqueles que combinam múltiplos processadores em um único sistema[38].

Existem várias técnicas para desenvolver programas paralelos. Este trabalho está focado em sistemas de memória compartilhada e por isso foi escolhida uma API adequada para estes ambientes, chamada OpenMP. A seguir serão apresentados conceitos importantes para a exploração de paralelismo em programação.

3.4.1 Tipos de computadores paralelos

Originalmente computadores comuns possuíam um único processador que acessa um bloco de memória. Para que seja possível realizar execuções de um programa em paralelo, existem duas construções principais de ambientes de computação paralela.

O primeiro tipo são os computadores de memória distribuída, que são mais fáceis de serem construídos e mais difíceis de se programar. Eles são constituídos por vários computadores simples conectados por uma rede. Um exemplo de computadores deste tipo são os *clusters*. Cada nó do sistema possui seu próprio processador e sua própria memória e esta não pode ser acessada por outros processadores. Quando há necessidade de comunicação entre as máquinas existe um serviço na rede que permite a troca de mensagens. Nestes sistemas um problema pode ser dividido em subproblemas, onde cada subproblema é processado por um nó. Assim, quando cada nó terminar o seu trabalho os resultados dos subproblemas são unidos e formam a resposta do problema inicial.

O segundo tipo são computadores de memória compartilhada que consistem em vários processadores acessando o mesmo espaço de endereçamento de memória. Estes computadores são mais difíceis de se fabricar, pois necessitam de barramentos avançados e modelos de consistência de memória, mas são mais simples de se programar. Esta tecnologia tem se tornado cada vez mais comum com os processadores *multi-core* dominando o mercado atual, já que os processadores unitários estão próximos do limite físico que impede o aumento da frequência[39]. Exemplos de processadores deste tipo são o Intel i7 e o AMD Turion.

Os computadores de memória distribuída são considerados mais difíceis de programar por causa da necessidade de enviar e receber mensagens, o que obrigatoriamente deve ser explicitado no código. Por outro lado, em computadores de memória compartilhada não

há troca de mensagens e a troca de informações é feitas de forma semelhante ao que ocorre em programas sequenciais. Ou seja, são apenas feitas operações de leitura e escritas nas variáveis, o que é mais natural para os programadores não habituados com programação paralela.

3.4.2 Processos e Threads

Em sistemas operacionais, um processo é um programa em execução. Cada processo possui seu próprio espaço de endereçamento, que é uma lista de locais na memória que um processo pode acessar. Este espaço contém todas as informações necessárias para a execução de um programa, como as instruções e os seus dados, a pilha de execução e o conjunto de registradores[40].

Um processo pode criar uma cópia idêntica de si através da chamada de sistema denominada *fork*. As cópias criadas, também chamadas de processos filhos, se diferem do processo pai apenas por um identificador numérico, atribuído pelo sistema operacional. Ao término de suas execuções, os filhos podem se juntar ao processo original em uma operação conhecida como *join*. Cada processo criado possui o seu próprio espaço de endereçamento.

Para o usuário, os sistemas operacionais aparentam executar vários processos simultaneamente em apenas um processador, o que pode ser chamado de pseudoparalelismo. Isso acontece porque há uma rápida alternância de processos no processador, ou seja, um processo é executado durante uma fatia de tempo pelo processador, quando então é interrompido para que outro processo seja executado durante a outra fatia de tempo e assim sucessivamente. Isto é conhecido como multiprogramação[40]. Quando se interrompe a execução de um processo e aloca-se outro ocorre a troca de contexo, que armazena informações do espaço de endereçamento do primeiro processo para que ele possa ser restaurado mais tarde e retornar ao processador. E também deve-se restaurar o estado do próximo processo. Em contraste à multiprogramação que simula paralelismo existe o multiprocessamento, que é o paralelismo real que ocorre em sistemas com múltiplos processadores.

Threads, também chamadas de processos leves, são fluxos de execução similiares a processos, porém ao se criar novas *threads* não se cria um espaço de endereçamento para cada uma delas. Ao invés disto, elas compartilham o mesmo espaço com o processo pai.

Isto faz com que o custo de processamento e memória seja menor para manter *threads* do que para manter processos. Outra vantagem de *threads* é que a troca de contexto é mais barata. Esta troca é o procedimento de armazenar ou restaurar o estado de um processador que deve ocorrer para que vários processos compartilhem um único processador. Um processo pode subdividir-se em várias *threads* e estas podem ser executadas simultaneamente[40].

Em sistemas de memória distribuída cada máquina executa um processo independente. Quando é necessário trocar informações entre processos deve-se explicitamente enviar uma mensagem pela rede. Em sistemas de memória compartilhada são disparados processos que se dividem em *threads* que são executadas concorrentemente. Como o espaço de endereçamento é o mesmo, o envio de mensagens não é necessário, pois naturalmente as variáveis globais de um programa são compartilhadas entre todas as suas *threads*. Portanto, a troca de informações é feita por leituras e escritas na memória que devem ser sincronizadas.

3.4.3 Análise de desempenho

Na literatura existem medidas que ajudam a quantificar e avaliar o desempenho de uma aplicação em ambientes paralelos. As mais relevantes serão apresentadas a seguir.

3.4.3.1 Fator de *Speedup*

O fator de *speedup* ou de aceleração determina quão mais rápido é um programa paralelo comparado com o mesmo programa executado sequencialmente em um só processador. O *speedup* de um programa executado em p processadores é dado pela razão entre o tempo de execução sequencial T_{seq} e o tempo de execução em p processadores T_p.

$$S(p) = \frac{T_{seq}}{T_p} \tag{3.32}$$

Geralmente a aceleração máxima é p, com p processadores, o que é chamado de *speedup* linear. Esta aceleração é obtida quando o problema pode ser dividido em partes iguais onde cada processador recebe um processo e não há *overheads*.

Overheads são custos adicionais com operações que normalmente não seriam executa-

das em processos sequenciais. Por exemplo, existem custos computacionais para trocar mensagens pela rede, criar ou finalizar *threads* e sincronizar processos.

3.4.3.2 Eficiência

Outro aspecto a ser avaliado é a eficiência, que é uma estimativa do tempo em que os processadores realmente realizaram trabalho e é dado pela razão entre o *speedup* e o número de processadores:

$$E = \frac{T_{seq}}{T_p p} = \frac{S_p}{p} \tag{3.33}$$

Por exemplo, eficiência de 0.5 indica que os processadores permaneceram ocupados durante a metade do tempo de computação total.

3.4.3.3 Lei de Amdahl

Muitos fatores influenciam o *overhead*: a) o período de tempo em que processadores não realizam trabalho útil; b) operações extras que implementações em paralelo fazem que na versão sequencial não são necessárias e c) comunicação entre processos. Também é notável que algumas partes do código não podem ser paralelizadas e devem ser executadas sequencialmente, de maneira que os demais processadores ficam aguardando estas tarefas terminarem para poderem começar a sua execução. A influência da parte sequencial de um programa sobre o desempenho é dado pela Lei de Amdahl:

$$S(p) = \frac{p}{1 + (p-1)f} \tag{3.34}$$

Onde $S(p)$ é o *speedup* para p processadores e f é a fração de código que não pode ser divida em tarefas concomitantes. Se houver um número grande de nós, a aceleração será inversamente proporcional a f:

$$\lim_{p \to \infty} \frac{p}{1 + (p-1)f} = \frac{1}{f} \tag{3.35}$$

Por exemplo, caso 10% do código seja sequencial e muitos processadores estejam disponíveis, o *speedup* máximo será $\frac{1}{0.1} = 10$.

```
1 //comandos executados sequencialmente
2 #pragma omp parallel
3 {
4   //comandos executados paralelamente
5 }
6 //comandos executados sequencialmente
```

Figura 3.9: Diretiva *parallel*

3.4.4 OpenMP

OpenMP é uma API que possibilita a programação em ambientes de memória compartilhada em C/C++ e Fortran, para vários sistemas operacionais e arquiteturas de processadores[41], oferecendo uma interface simples e flexível para o desenvolvimento de aplicações paralelas.

A programação com OpenMP é feita através de diretivas do compilador combinadas com algumas chamadas de função e variáveis de ambiente. As diretivas possuem o seguinte formato: *#pragma omp nome_da_diretiva*. OpenMP é amplamente divulgado e aplicado na literatura[37, 42] e aqui serão reportados os aspectos mais relevantes para a implementação deste trabalho. Os comandos apresentados a seguir são utilizados por programas escritos em linguagem C/C++.

OpenMP gerencia *threads* de maneira semelhante ao modelo de *fork-join* de processos. Inicialmente a *thread* mestra é executada sozinha até encontrar a diretiva *parallel* que cria um time de *threads*. O bloco de código que será executado simultaneamente é delimitado por chaves - { e }. Cada *thread* do time executará o bloco inteiro. Quando o time terminar a mestra volta a ser executada sozinha. Há uma barreira implícita no final do bloco, ou seja, todos os membros do time devem terminar de executar todo o bloco para que a mestra volte a ser executada. Caso uma *thread* termine antes das demais, ela deve aguardar até que todas terminem. A Figura 3.9 mostra este esquema.

Por padrão, o número de *threads* criadas é o número de processadores disponíveis no sistema. Caso o programador queira especificar o número de *threads* a serem criadas é possível invocar a função $omp\_set\_num\_threads(numeroThreads)$. Cada *thread* possui um número identificador que pode ser conhecido através da função $omp\_get\_thread\_num()$. A *thread* mestra é sempre identificada por 0.

3.4.4.1 Escopo de dados

Uma questão importante é determinar o escopo das variáveis. Por padrão, as variáveis declaradas antes da diretiva *parallel* são consideradas compartilhadas, o que significa que elas poderão ser lidas ou escritas por todas as *threads* do time. Para declarar explicitamente uma variável como compartilhada, deve-se adicionar a cláusula *shared* à diretiva *parallel*.

Por outro lado, pode ser necessário que as variáveis sejam visíveis apenas para uma única *thread*. Para tal, basta acrescentar a cláusula *private*, da mesma maneira que a cláusula anterior. Desse modo, cada *thread* do time possuirá uma nova variável do mesmo tipo, que será apenas visível e acessada pela *thread* que a possui. Variáveis privadas não podem ser inicializadas pela mestra, porque elas serão declaradas novamente dentro de cada *thread*, e não podem ser acessadas fora da região paralela.

Há ainda outras cláusulas especiais para determinar o escopo de variáveis, explicadas a seguir. A primeira cláusula é a *firstprivate*, que declara as variáveis como privadas, porém estas podem ser inicializadas pela mestra. Cada *thread* do time começa a execução com o valor atribuído anteriormente, e então realiza leituras e escritas na sua variável independentemente.

A cláusula *lastprivate* também declara variáveis como privadas. Porém, as variáveis mantêm-se visíveis após o término da execução do bloco paralelo. Uma variável deste tipo possuirá o valor que foi atribuído a ela na *thread* que foi executada por último.

A claúsula *reduction* é uma maneira de reunir o trabalho de todas as *threads*. Este processo funciona da seguinte maneira: uma cópia privada da variável é criada para cada *thread*. Ao final do bloco paralelo, todas as cópias são reduzidas em uma só variáviel. Por exemplo, cada *thread* realiza um cálculo com uma variável x, e ao final deseja-se que os valores obtidos individualmente sejam todos somados e armazenados em uma só variável. Para tal, a declaração da diretiva deve ser: $\#pragma\ omp\ parallel\ reduction(+ : x)$.

3.4.4.2 Compartilhamento de trabalho

A diretiva *parallel*, apresentada anteriormente, é utilizada para criar um time de *threads*. Nesta seção serão apresentados comandos que dividem a execução de um bloco de código entre as *threads* do time.

A cláusula *for* é utilizada para paralelismo de dados o que significa que serão execu-

```
1  #include "omp.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4  int main(){
5    int vec[100];
6    for (int i=0; i<100;i++)
7      vec[i] = i + 1;
8
9    int soma;
10   #pragma omp parallel reduction(+:soma)
11   {
12     soma = 0;
13     #pragma omp for schedule(static)
14     for(int i=0; i<100; i++){
15       soma +=vec[i];
16     }
17   }
18   printf("A soma e %d\n", soma);
19 }
```

Figura 3.10: Somatório de um vetor em paralelo

tadas as mesmas tarefas, porém sobre dados diferentes. A cláusula deve aparecer sempre em conjunto com o comando de repetição *for* da linguagem de programação. Ela divide um *loop* em partes que serão distribuídas para as *threads* do time. A divisão pode ocorrer de duas maneiras, estática ou dinâmica, e é declarada através da cláusula *schedule*(tipo_-divisão, tamanho). A divisão estática consiste em dividir as iterações em "pedaços" do tamanho especificado e então cada bloco é atribuído estaticamente a cada *thread*. Se o tamanho não for declarado, as iterações são divididas igualmente entre as *threads*. Por exemplo, considera-se um algoritmo que soma todos os elementos de um vetor com 100 posições em um sistema com quatro processadores. A implementação deste problema pode ser visualizada na Figura 3.10. O primeiro laço apenas inicializa o vetor com valores de 1 a 100. Logo após inicializa-se a seção paralela, criando-se quatro *threads*, onde a variável "soma" é declarada como do tipo *reduction*. Consequentemente, serão feitas cópias dela e estas serão calculadas individualmente por cada *thread*. Ao final da seção paralela, como está especificado pelo sinal + na cláusula *reduction*, as cópias serão somadas e este resultado será armazenado em "soma". Para cada *thread* deve-se inicializar a variável "soma", o que é feito na linha 12. Na linha 13, determina-se que as iterações do laço a seguir, que soma os valores do vetor, serão divididas entre as quatro *threads* do time. Pela divisão estática a primeira *thread* somará da posição 0 à posição 24; a segunda de 25 a 49; a terceira de 50 a 74; e a última de 75 a 99. Ao final o resultado da soma é

exibido.

Para divisões dinâmicas o padrão é atribuir uma iteração para cada *thread*. Neste exemplo será considerada que a quantidade de iterações é definida como dez. A cláusula *for* ficará assim: $\#pragma\ omp\ for\ schedule(dynamic,\ 10)$. Desta forma as quatro *threads* receberão dez iterações contínuas cada. Quando a primeira terminar o OpenMP atribui mais dez iterações a esta *thread*, e assim sucessivamente, até que as 100 iterações sejam computadas.

Existem vantagens e desvantagens entre as duas divisões. A divisão dinâmica é mais flexível, mas claramente adiciona custo extra para gerenciar e sincronizar as *threads*, pois deve-se observar a todo momento qual delas já terminou o seu trabalho, para que se possa enviar mais tarefas e deve-se garantir que cada iteração seja executada apenas uma vez[43].

A divisão estática não possui esse *overhead*, mas pode gerar mal balaceamento de carga, ou seja, uma *thread* pode receber mais trabalho que as demais. Em tarefas que podem ser divididas em blocos que possuem aproxidamente o mesmo custo computacional, a divisão estática é mais adequada. Por outro lado, existem problemas que não podem ser divididos em blocos com o mesmo custo. Dividindo estaticamente uma tarefa deste tipo é possível que algumas *threads* recebam um bloco com tarefas muito mais custosas do que as demais. Portanto, aquelas que receberam tarefas mais leves terminarão seu trabalho antes daquelas com carga maior e deverão esperar ociosamente até que todas terminem de executar. Isto causa desperdício de recursos computacionais, prejudicando o desempenho. Neste caso é melhor dividir as tarefas dinamicamente, pois as *threads* que receberem tarefas mais leves terminarão primeiro e logo em seguida receberão mais trabalho. Este processo manterá os processadores ocupados por mais tempo, o que compensará o tempo gasto com sincronizações.

Outra claúsula para divisão de trabalho é a cláusula *sections*. A sua estrutura é ilustrada pela Figura 3.11. Esta cláusula indica que cada *thread* receberá uma tarefa diferente. As tarefas estão contidas nos blocos de código iniciados por $\#pragma\ omp\ section$.

A terceira e última cláusula é a cláusula *single*. Durante a execução de tarefas paralelas pode ser necessário que uma parte do código seja executada sequencialmente, por apenas uma única *thread*. Para tal, o bloco de comandos a ser executado por somente uma *thread* deve ser iniciado por $\#pragma\ omp\ single$.

```
1  #pragma omp parallel
2  {
3    #pragma omp sections
4    {
5      #pragma omp section
6      {
7        //bloco de comandos
8      }
9      //outros blocos section
10   }
11 }
```

Figura 3.11: Cláusula *sections*

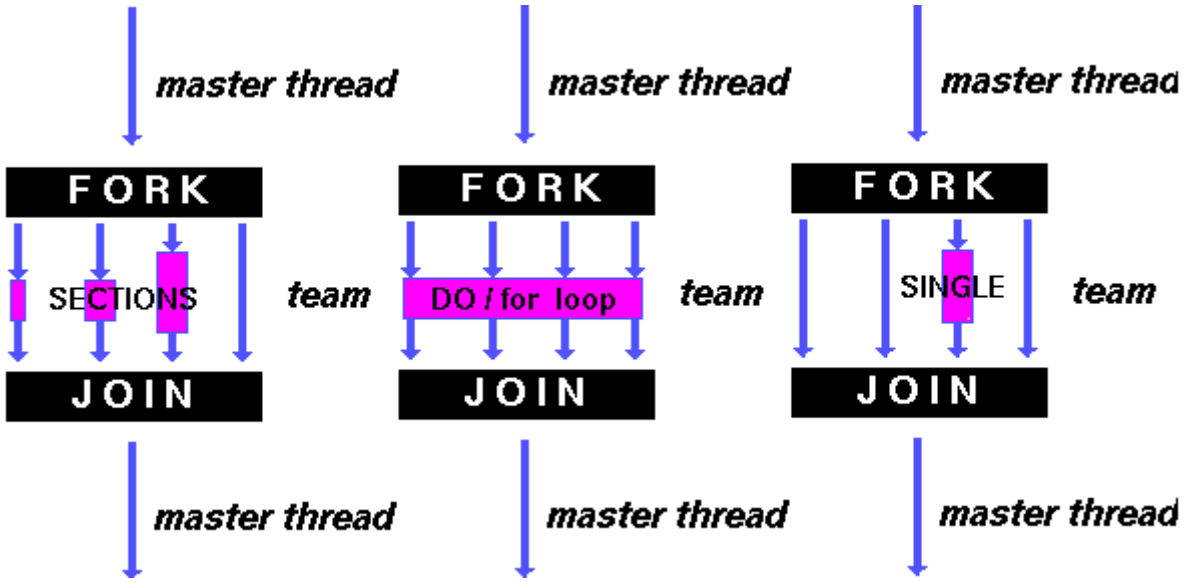

Figura 3.12: Divisões de trabalho - retirada de [42]

Estes comandos podem ser combinados para formarem blocos complexos de programas paralelos. Eles não iniciam um time de *threads*, e por isso devem aparecer dentro de blocos com a diretiva *parallel*. Por padrão, todos eles possuem uma barreira implícita no final do bloco, ou seja, o bloco seguinte só será executado quando todas as *threads* terminarem o seu trabalho. Para retirar esta barreira, deve-se adicionar a cláusula *nowait* no início do bloco. A Figura 3.12 ilustra o comportamento das três cláusulas de divisão de trabalho.

3.4.4.3 Sincronização

As cláusulas de sincronização são utilizadas para coordenar variáves compartilhadas que são acessadas por múltiplas *threads*, pois pode acontecer que várias *threads* precisem atualizar uma certa variável, ou então uma *thread* tente ler uma variável que está sendo alterada por outra. Uma situação que cause acessos conflitantes pode ocasionar resultados

incorretos. Os comandos apresentados a seguir evitam estes problemas.

#pragma omp critical: a cláusula *critical* permite que somente uma *thread* execute um bloco de código por vez. Quando uma ou mais *threads* alcançam uma seção crítica, elas esperam até que nenhuma outra esteja executando o bloco da seção crítica para executá-lo.

#pragma omp barrier: quando uma *thread* alcança uma barreira, indicada pela cláusula *barrier*, ela aguarda até que todas as outras também a alcançem, quando então todas elas continuam a execução.

#pragma omp atomic: esta cláusula define uma área de exclusão mútua, do mesmo modo que a cláusula *critical*. Porém, a cláusula *atomic* só pode ser aplicada a um único comando de atribuição, que atualize uma variável escalar. Por exemplo $x = expressão$. Por outro lado, o comando *critical* pode ser aplicado a um bloco qualquer de código.

#pragma omp $flush(lista\_de\_variáveis)$: esta sincronização faz com que as *threads* tenham um visão consistente de uma variável na memória. Para alcançar isto todas as operações de leitura e escrita na memória, que se encontram nos trechos de códigos anteriores ao local onde está o *flush*, devem ser executados antes do ponto de sincronização. Da mesma maneira todas as operações de memória que se encontram depois desse ponto só podem ser executadas após a execução da sincronização. Esta cláusula é automaticamente executada no início e no fim das diretivas *parallel* e *critical*, e somente no fim das cláusulas *for*, *section* e *single*, se a cláusula *nowait* não estiver presente.

ordered: usada em conjunto com a cláusula *for*, para que as iterações sejam executadas na ordem em que seriam executads se o código fosse sequencial.

#pragma omp master: esta cláusula determina que somente a *thread* mestra executará um bloco de código. Neste caso não há barreira implícita no início ou no fim do bloco. As *threads* do time, exceto a mestra, ignoram o bloco desta diretiva e continuam a execução do código seguinte.

4 METODOLOGIA

Este capítulo apresenta o trabalho realizado para alcançar os dois objetivos propostos. São eles o desenvolvimento de uma ferramenta para edição de modelos descritos em CellML e a utilização de técnicas computacionais para resolver os sistemas de equações com desempenho satisfatório.

4.1 Editor

Os modelos computacionais são ferramentas importantes para a compreensão dos fenômenos relacionados à eletrofisiologia cardíaca e têm sido utilizados para auxiliar testes de novas drogas, desenvolvimento de equipamentos e técnicas de diagnósticos não-invasivos. Entretanto, o desenvolvimento de modelos realistas pode ser complicado e exige conhecimentos de programação, matemática e fisiologia. Além disto, a diversidade de plataformas computacionais e linguagens de programação dificultam o compartilhamento e reuso de modelos, restringindo a sua utilização a poucos grupos de pesquisa.

Para amenizar este problema foi criado o padrão CellML de descrição de modelos celulares, baseado em XML (*eXtensible Markup Language*). Porém, os modelos podem possuir dezenas de equações com centenas de parâmetros, o que dificulta a sua construção nesta linguagem. Por isto este trabalho propõe uma ferramenta para edição e criação de modelos, através de uma interface *Web*. A idéia é que o usuário interaja com o Editor sem se preocupar com o CellML, principalmente ao descrever as equações. No final do processo o Editor transforma o modelo descrito pelo usuário em um arquivo CellML. O Editor também foi desenvolvido integrado ao AGOS - que permite o armazenamento e a simulação dos modelos. Outra ferramenta que edita arquivos CellML chama-se COR [23]. Entretanto, a sua instalação é restrita a sistemas operacionais Microsoft Windows. Em contrapartida, a proposta deste trabalho é construir uma aplicação para a *Web* de forma a não depender do sistema operacional do usuário, de não haver a necessidade de instalação e a possibilidade de integração com o Portal Fisiocomp.

O Editor cria uma interface, onde é possível editar as unidades, componentes e variáveis dos modelos. As equações devem ser escritas em forma de texto e ao final do processo

todas estas informações são reunidas em um arquivo CellML. A seguir serão apresentados maiores detalhes da construção desta ferramenta.

4.1.1 Implementação

O intuito desta ferramenta é facilitar a descrição de modelos CellML através de uma interface *Web* integrada ao AGOS. A linguagem de programação PHP foi utilizada para o desenvolvimento do Editor. A principal motivação para esta escolha foi que a interface do AGOS já havia sido construída com esta linguagem. Além disto, o Editor exige um maior dinamismo em sua interface, o que foi atingido com Ajax - *Asynchronous JavaScript and XML*[44], um conjunto de técnicas que permite atualizar páginas *Web* de maneira assíncrona. Em outras palavras, Ajax permite que dados sejam trocados entre o *browser* e o servidor de modo que não seja necessário recarregar toda a página para se exibir a informação obtida. Isto proporciona mais dinamismo e usabilidade nas aplicações *Web*[45], além de possuir outras vantagens como diminuir a carga nos servidores e o tráfego na rede[46]. As tecnologias que compõem o Ajax são: *Cascading Style Sheets* (CSS), uma maneira de definir estilos visuais para páginas *Web*; *Documment Object Model* (DOM), que organiza a estrutura de uma página *Web* em um conjunto de objetos programáveis, permitindo que a aplicação Ajax modifique a interface de usuário; e *XMLHttpRequest Object*, que permite a troca de dados entre o cliente e o servidor através de requisições HTTP. Uma aplicação Ajax é controlada pela linguagem JavaScript, que manipula a interface de usuário através do DOM e processa as interações entre a interface e o usuário através de eventos do *mouse* e teclado. O CCS é utilizado para apresentar os resultados e *XMLHttpRequest Object* é responsável pela comunicação, enviando e recebendo dados enquanto a interface está sendo utilizada. Ou seja, não há recarregamento de página, permitindo que a mesma continue a ser visualizada ao mesmo tempo em que ocorrem atualizações em sua interface.

O Editor foi construído com os recursos apresentados acima e funciona da seguinte maneira: inicialmente o usuário escolhe entre editar um modelo existente ou criar um novo modelo. No primeiro caso, o programa lê o arquivo fornecido pelo usuário, e no segundo lê um arquivo modelo, que apenas contém as estruturas básicas - algumas unidades e componentes com uma equação que servem de exemplo para a construção do novo modelo. O programa percorre todo o arquivo de entrada e identifica todas as *tags* que o compõem,

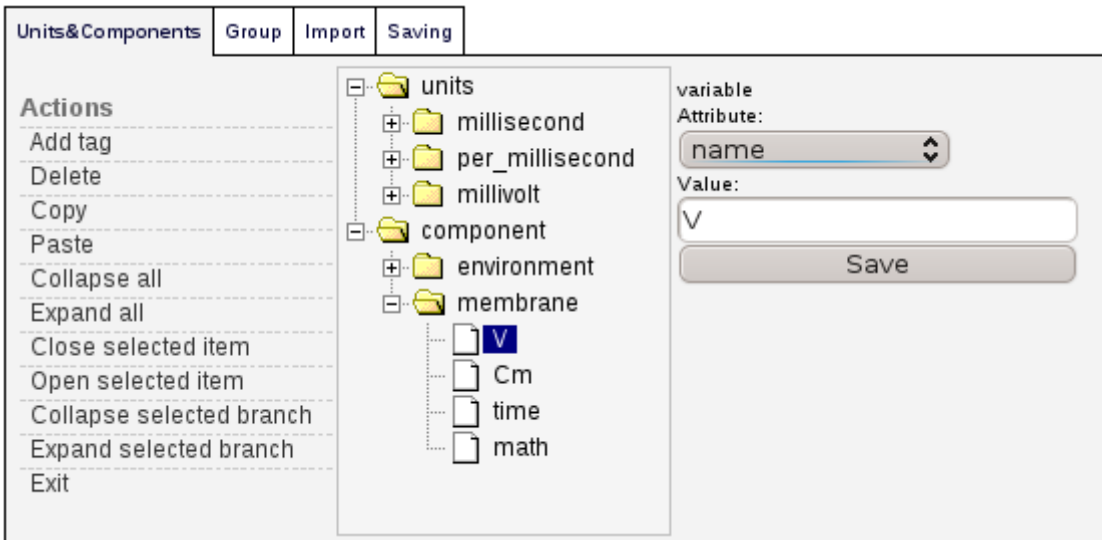

Figura 4.1: Editor - Interface

afim de montar a interface de usuário. Primeiro, o arquivo é transformado em uma estrutura de árvore que é armazenada em um banco de dados. A partir destes dados cria-se uma interface de usuário, onde as unidades e os componentes são organizados em uma estrutura semelhante a uma árvore de diretórios. Dentro da pasta chamada "component" estão as variáveis e suas equações, como exibido na Figura 4.1. No lado esquerdo do Editor, há um painel com as possíveis operações para manipular o modelo, como adicionar novos items, apagá-los, copiar e colá-los, além de expandir e recolher os itens de um nó, função que também está disponível nos sinais de + e − próximo aos nós da árvore - mais uma vez assemelhando-se com a interface que alguns sistemas operacionais oferecem para a navegação em diretórios.

No painel direito aparecem dois campos ao se selecionar um item da árvore, um com os atributos válidos para o objeto selecionado e outro com o valor do atributo. Por exemplo, ao selecionar uma variável o código JavaScript caputra o evento do *mouse*, que então aciona o *XMLHttpRequest Object* que envia uma requisição para o servidor com a mensagem "listar atributos de variáveis". O servidor realiza uma consulta no banco de dados e retorna o resultado para o cliente que fez a requisição. Este, por sua vez, modifica os atributos do campo *Attribute* através do DOM, na página formatada com CSS. Este processo é mostrado na Figura 4.2. O mesmo procedimento se repete ao se escolher um atributo - por exemplo, ao se escolher o atributo "*initial_value*", um pedido é enviado ao servidor, que retornará o valor inicial da variável e a aplicação preencherá o campo "*Value*".

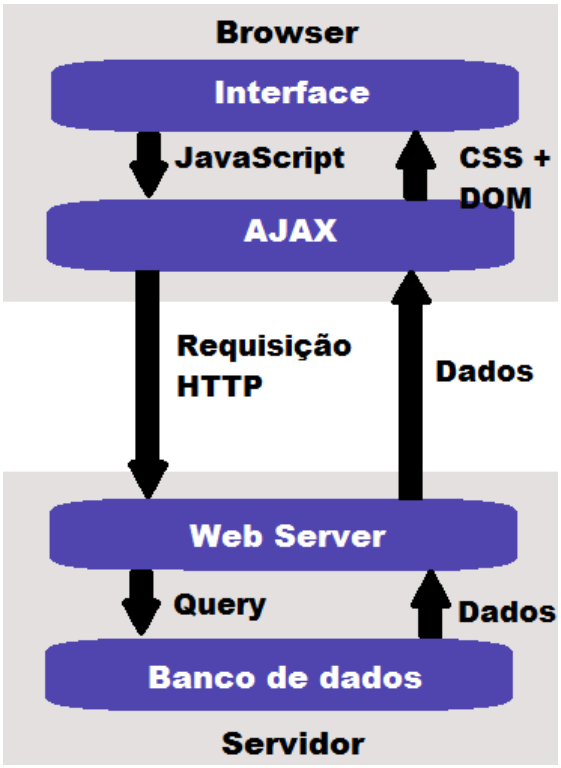

Figura 4.2: Fluxo de dados - AJAX

Para a alteração de equações foi implementada uma inteface separada. Elas também são lidas juntamente com o arquivo inicial, transformadas em árvore e salvas em um banco de dados. Entretanto, o usuário altera equações como texto, como se estivesse escrevendo equações em linguagem C o que facilita muito a sua manipulação pois claramente equações com muitas operações e variáveis são difíceis de serem lidas e escritas em CellML ou MathML.

A primeira etapa para a construção da interface é simples, basta que a árvore com as equações seja percorrida a fim de obter a *string* com as equações matemáticas. Cada operação matemática é composta de uma *tag* chamada "*apply*" que possui dois ou três nós filhos, como visto na Seção 3.2. Um deles, obrigatoriamente, informa a operação matemática. Se esta for uma operação unária, haverá um outro nó filho que pode ser um numeral, uma variável ou uma expressão. Nos dois primeiros casos o programa imprime o valor encontrado. Já no terceiro o procedimento é executado recursivamente: encontra-se a operação, e os outros filhos são percorridos em profundidade. Caso seja uma operação binária, o método percorre o termo à esquerda primeiro e depois o que está à direita.

Após a impressão da *string*, a equação pode ser editada normalmente. Porém, o processo oposto, que transforma a *string* em árvore, é mais complicado. Deve-se determinar

o que são números, variáveis e sinais matemáticos, além da precedência de operadores, que são problemas tradicionalmente resolvidos por compiladores de linguagens de programação[47, 48]. Portanto, foi criada uma linguagem de programação simplificada, para que seja possível escrever equações matemáticas e depois transformá-las em CellML. Como sugerido pela literatura na área de compiladores, a primeira etapa de construção é a análise léxica, na qual a equação em forma de *string* será lida caracter a caracter de maneira a agrupá-los em *tokens*, que são sequências de caracteres com significado. Por exemplo, ao se avaliar a equação $Vm = 5.2 * b$, o analisador léxico determinará que Vm e b são identifcadores de variáveis, $=$ e $*$ são os sinais de atribuição e multiplicação, respectivamente, e 5.2 representa um número.

O analisador léxico é composto por um autômato finito determinista (AFD), que tem a função de reconhecer os *tokens* da equação a ser interpretada. O AFD é uma máquina de estados M que é dada por:

$$M = (Q, \Sigma, \delta, q_0, F), \tag{4.1}$$

onde Q é um conjunto finito e não vazio de estados, Σ é o alfabeto que são os caracteres reconhecíveis pelo automâto, δ é o conjunto de transições entre os estados, o que indica como o AFD muda de um estado para o outro, a partir de um símbolo que pertença ao alfabeto Σ, $q_0 \in Q$ é o ponto de partida, chamado de estado inicial, e F é um subconjunto de Q que contém os estados finais, ou seja, os estados em que a máquina classifica a palavra de entrada e finaliza o seu funcionamento.

O AFD funciona em conjunto com a tabela de símbolos que é responsável por armazenar todas as palavras reservadas da linguagem, além dos identificadores e números presentes no programa de entrada. A Figura 4.3 ilustra o AFD desenvolvido. Nela o estado inicial é $q0$, indicado com um triângulo branco, e os estados finais são os estados com dois círculos circunscritos. Por exemplo, a expressão $xa2 = 5.3$ será avaliada da seguinte maneira: o primeiro caracter é "x", e inicialmente o estado é $q0$. A única transição possível para este caracter leva para $q31$, pois é única que aceita letras. A transição para os segundo e terceiro caracteres ("a" e "2") mantém o AFD no mesmo estado e então encontra-se "=" que é classificado como qualquer outro caracter não alfa-numérico, levando o AFD para o estado final $q33$. Ou seja, foi encontrado o identificador $xa2$, que é enviado para a tabela de símbolos, onde consulta-se se é uma palavra reservada. Caso

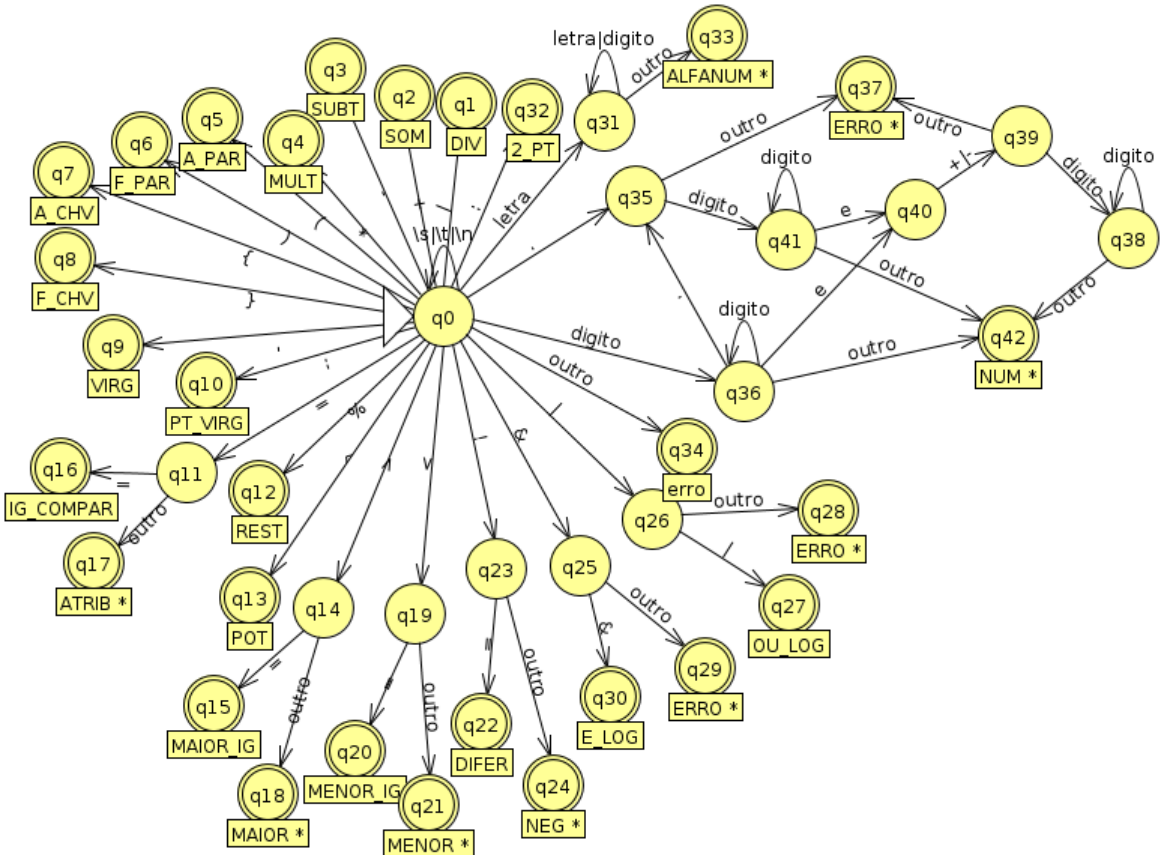

Figura 4.3: Autômato finito determista do Editor

não seja, é um nome de variável ou de função. Então volta-se para o estado inicial, onde se lê o sinal "=", que causa uma transição para o estado $q11$. Como não há outro sinal de igual na frente, o que seria uma comparação, o AFD reconhece que este é um *token* de atribuição, no estado final $q17$, e então o mesmo processo se repete com o número. Caso encontre caracteres que não perteçam ao alfabeto, o AFD retorna uma mensagem de erro para avisar o usuário.

A segunda etapa deste compilador é composta pelo analisador sintático, árvore de sintaxe e gerenciamento de erros. O analisador sintático implementa a gramática da linguagem que determina as formas de organizar os *tokens*, a fim de que eles formem comandos ou sentenças reconhecidas pela linguagem. O objetivo é verificar se o conjunto de *tokens* obtidos através do analisador léxico forma uma sentença que pode ser reconhecida pela gramática da linguagem fonte. O código que reconhece a linguagem é recursivo descendente, isto é, os *tokens* são percorridos recursivamente, do nó raiz para os nós folha e as produções são verificadas da esquerda para a direita. A sintaxe de uma linguagem de programação pode ser representada através de uma notação chamada gramática livre

expr -> *attrib*
attrib -> *or attrib'*
attrib' -> **=** *or attrib'* | ε |**=** *select*
or -> *and or'*
or' -> **||** *and or'* | ε
and -> *equaldiff and'*
and' -> **&&** *equaldiff and'*| ε
equaldiff -> *lessthan equaldiff*
equaldiff' -> **==** *lessthan equaldiff'* | **!=** *lessthan equaldiff'* | ε
lessthan -> *sum lessthan'*
lessthan' -> **<** *sum lessthan'* | **<=** *sum lessthan'* | **>=** *sum lessthan'* | **>** *sum lessthan'* | ε
sum -> *mult sum'*
sum' -> **+** *mult sum'* | **-** *mult sum'* | ε
mult -> *not mult'*
mult' -> ***** *not mult'* | **/** *not mult'* | **%** *not mult'* | **&** *not mult'* | ε | **^** *not mult'*
not -> *not'* | *value*
not' -> **!** *value* | **+** *value* | **-** *value* | ε
value -> **ID** | **NUM** | **ID ()** | **ID (** *exprlist* | **(** *expr* **)** | **reservedWords**
exprlist -> **)** | *expr exprlisttail* **)**
exprlisttail -> **,** expr exprlisttail | ε
select -> **SELECT{** *case*+ *otherwise***}**
case -> *value* **CASE:** *orr;*
otherwise -> **OTHERWISE(***orr***);**

Figura 4.4: Gramática livre de contexto do Editor

de contexto[47], que é composta por quatro componentes: um conjunto de *tokens* ou símbolos terminais, um conjunto de não terminais, um conjunto de produções, que são compostas por um lado direito e um esquerdo.

O lado esquerdo é um não-terminal e o lado direito possui uma sequência de *tokens* e não-terminais. O último componente da gramática é um não-terminal que é escolhido como ponto de partida. A gramática desenvolvida para este trabalho pode ser visualizada na Figura 4.4. É ela que define a estrutura hierárquica das construções da linguagem de programação proposta neste trabalho. Os itens em negritos são os *tokens* terminais, a primeira produção é o símbolo de partida, os itens em itálico são não-terminais, ε é a palavra vazia, e não-terminais com mais de uma produção possuem seus lados direitos separados pelo caracter |, lido como "ou". As setas separam os lados direito e esquerdo da produção. Por exemplo, o comando *SELECT* para descrever equações condicionais é definido pelo não terminal *select*, que concantena a palavra-chave *SELECT* com o *token* "{", e em seguida com dois não terminais, *case* e *otherwise*, logo em seguida está o *token* "}". O sinal + indica que pode haver uma ou mais ocorrências deste não-terminal. Por sua vez, *case* concantena o não terminal *value* com a palavra-chave *CASE*, seguida de ":",

```
y = SELECT{
x^2  CASE:  x > 3;
0       CASE:  x>0 && x<=3;
OTHERWISE(x);
}
```

Figura 4.5: Comando SELECT

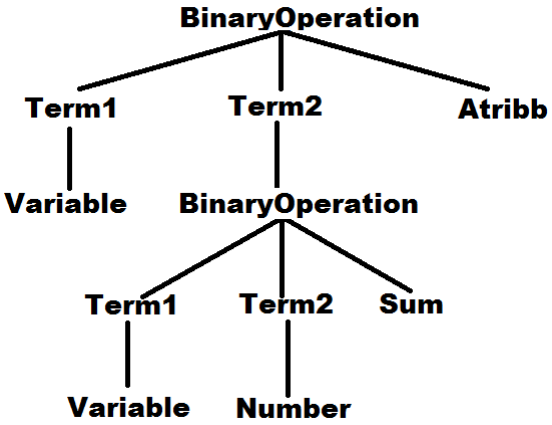

Figura 4.6: Árvore de sintaxe

outro não terminal e então ";". A Equação 4.2 será descrita como na Figura 4.5.

$$y = \begin{cases} x^2 & \text{se} \quad x > 3 \\ 0 & \text{se} \quad 0 < x \leq 3 \\ x & \quad \textit{caso contrário} \end{cases} \tag{4.2}$$

A árvore de sintaxe é a estrutura sintática dos comandos da linguagem que neste caso são as equações do modelo. As equações são percorridas pelo analisador sintático que monta a árvore de sintaxe ao encontrar padrões definidos pela gramática. Os nós da árvore são compostos de instâncias de classes criadas para representar as expressões e atribuições da linguagem criada. Por exemplo, ao se avaliar a expressão $a = x + 2$, o analisador sintático avaliará os *tokens* e montará a árvore de sintaxe ilustrada pela Figura 4.6. Inicialmente, considera-se que a expressão é uma operação binária, com os termos a e $x+2$ e o operador $=$. O segundo termo também é uma operação binária, que então é dividida nos termos x e 2, com o operador $+$. A partir desta estrutura obtém-se facilmente o código CellML, bastando percorrer a árvore e associar os operadores da árvore com os comandos CellML equivalentes.

O gerenciamento de erros é responsável por avisar o usuário que há algum problema

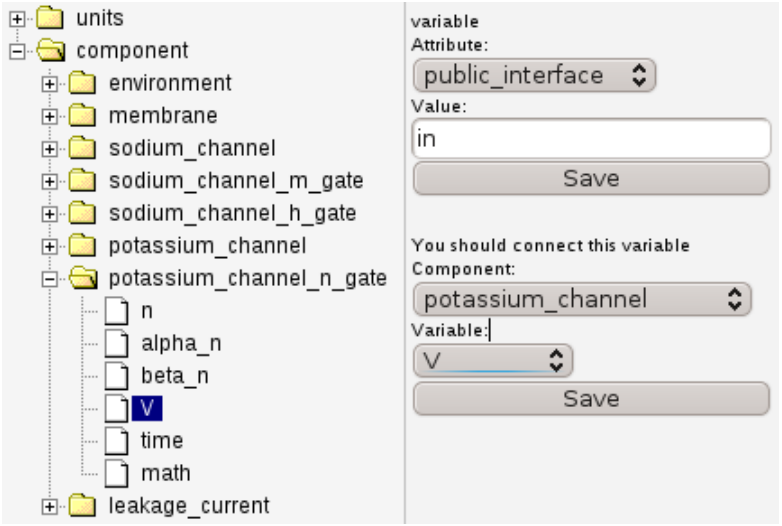

Figura 4.7: Variável calculada em outro componente

na sintaxe do seu programa informando qual o *token* é esperado mas não foi encontrado, o que é importante para que apenas sejam gerados arquivos CellML com equações válidas, que serão corretamente interpretadas por aplicações como o AGOS e o PyCML.

O editor possui ainda outras funcionalidades como conexões entre variáveis. Por exemplo, as variáveis que são utilizadas em um componente mas são calculadas em outro devem ser marcadas com o atributo "*public_interface*"=*in*.

Então, ao se selecionar o atributo em questão de uma variável que esteja com interface igual a "*in*", uma lista aparece com todos outros componentes do modelo. Ao se escolher um componente, todas as suas variáveis serão listadas. A Figura 4.7 ilustra esta situação, onde a variável V do componente *potassium_channel_n_gate* é conectada à variável V de *potassium_channel*.

Outra possibilidade é importar componentes de outros modelos que estejam no repositório do usuário. Na aba "*Import*", é possível copiar qualquer parte de outro modelo apenas selecionando os itens desejados, como na Figura 4.8.

Por último, é possível copiar o modelo alterado ou então salvá-lo no repositório do AGOS, para que este então possa ser simulado.

4.2 Técnicas para solução de equações

O outro objetivo deste trabalho é a implementação e a comparação de várias técnicas computacionais para resolver os sistemas de equações dos modelos da eletrofisiologia cardíaca,

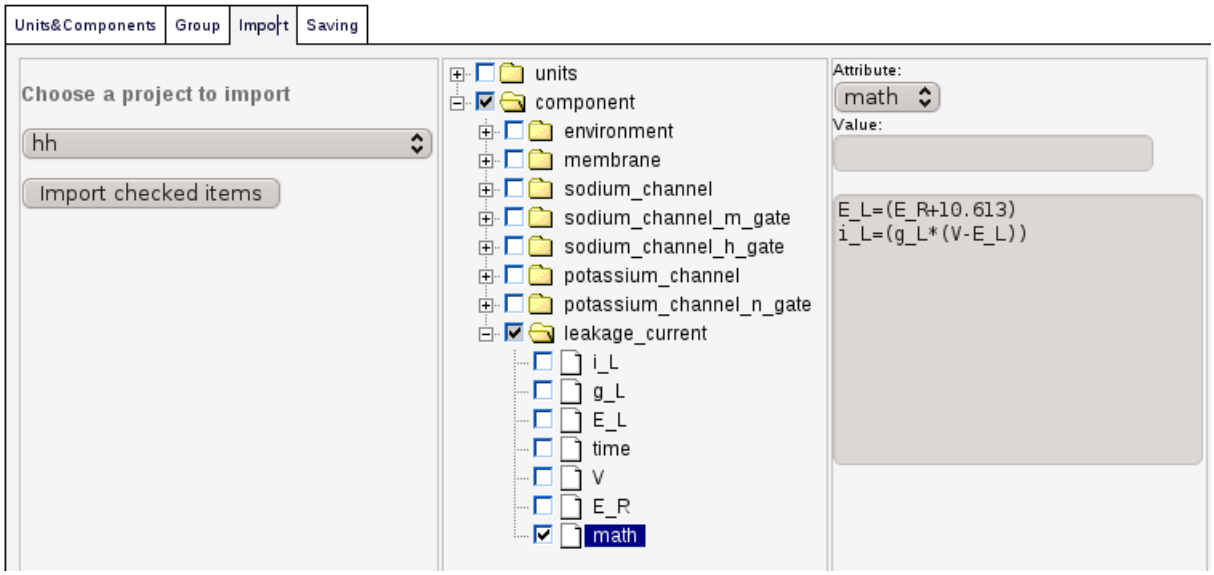

Figura 4.8: Importação de outros modelos

que são complexos de natureza altamente não-linear e envolvem múltiplas escalas. Isto significa que simular estes modelos demanda poder de processamento computacional e pode levar muito tempo, de acordo com a complexidade do modelo. Para amenizar este problema foram utilizadas técnicas de computação paralela com OpenMP, que permite a utilização de recursos computacionais em ambientes multiprocessados com memória compartilhada. Também foram utilizados diferentes algoritmos e técnicas para resolver as equações diferenciais ordinárias com desempenho satisfatório e com percentuais de erro aceitáveis. Os métodos apresentados a seguir foram implementados no AGOS, fazendo-se necessário modificar o seu *template* de código para que quaisquer modelos descritos em CellML possam ser simulados com estes métodos.

4.2.1 Método adaptativo

Um dos métodos numéricos do AGOS é o método de Euler, que necessita de um passo de tempo muito pequeno para obter baixo percentual de erro e convergir. A utilização de um passo de tempo pequeno causa a necessidade de um grande número de iterações que resulta em demora para resolver o sistema. Em modelos mais realistas, que descrevem com mais detalhes os processos fisiológicos, a necessidade de poder de processamento aumenta. A simulação da atividade de uma célula cardíaca durante um segundo pode levar alguns minutos para ser executada. Quando se torna necessário simular um tecido, que é um conjunto de células, o método pode levar muitas horas ou até mesmo alguns dias

para simular uma única batida do coração, por exemplo. Para amenizar esta demora, este trabalho implementou um método numérico que apresenta melhor desempenho e pouco erro numérico. O objetivo do método é adaptar o passo de tempo, a fim de reduzi-lo em regiões mais instáveis, onde realmente há necessidade de passos de tempo pequenos e aumentá-lo em regiões estáveis. Foram adotadas duas maneiras de implementar o passo de tempo, através de uma fórmula e através de uma heurística. Ambas são baseadas na comparação entre os métodos de Euler e de Runge-Kutta Explícito de segunda ordem (RK2), apresentados na Seção 3.1. O método tem a seguinte forma, para um sistema $\overrightarrow{y}$ com k equações:

$$\begin{aligned} y_{n+1}^j &= y_n^j + h_n f^j(t_n, \overrightarrow{y_n}) \ , \ j = 1, \ \ldots, \ k \\ \widetilde{y}_{n+1}^j &= y_n^j + \frac{h_n}{2}(f^j(t_n, \overrightarrow{y_n}) + f^j(t_{n+1}, \overrightarrow{y_{n+1}})) \ , \ j = 1, \ \ldots, \ k \\ erro_n &= \|\overrightarrow{y_{n+1}} - \overrightarrow{\widetilde{y}_{n+1}}\|_\infty \\ h_{n+1} &= obtem\_novo\_h(erro_n, tol, h_n) \end{aligned} \tag{4.3}$$

Onde y_{n+1} é o resultado computado pelo método de Euler, $\widetilde{y}_{n+1}$ é o resultado obtido com o método de RK2 e *obtem_novo_h* é uma função que encontra o novo passo de tempo e $erro_n$ é a diferença entre as soluções obtidas pelos dois métodos, que pode ser simplificada para a seguinte forma:

$$\begin{aligned} erro_n &= \|\overrightarrow{y_{n+1}} - \overrightarrow{\widetilde{y}_{n+1}}\|_\infty \\ &= \|(y_n^j + h_n f^j(t_n, \overrightarrow{y_n})) - (y_n^j + \frac{h_n}{2}(f^j(t_n, \overrightarrow{y_n}) + f^j(t_{n+1}, \overrightarrow{y_{n+1}})))\|_\infty \\ &= \|\frac{h_n}{2}(f^j(t_n, \overrightarrow{y_n}) - f^j(t_{n+1}, \overrightarrow{y_{n+1}}))\|_\infty \end{aligned} \tag{4.4}$$

A primeira forma de estimar o novo passo de tempo é obtida através da observação que o $erro_n$ é da ordem de h_n^2, ou seja, $erro_n = \alpha h_n^2$. Isto deriva do fato do método de Euler ser de $\mathcal{O}(h^2)$ (erro local) e o de Heun, de $\mathcal{O}(h^3)$. Dessa forma, a diferença das duas soluções é da ordem de h_n^2.

Deseja-se encontar o novo passo de tempo h de maneira que o erro seja limitado a

uma tolerância tol:

$$erro_{n+1} = \alpha h_{n+1}^2 \leq tol \tag{4.5}$$

Assim, fazendo $erro_{n+1} = tol$:

$$\frac{erro_n}{erro_{n+1}} = \frac{\alpha h_n^2}{\alpha h_{n+1}^2}$$
$$h_{n+1} = h_n \sqrt{\frac{tol}{erro_n}} \tag{4.6}$$

Caso o erro obtido em uma iteração seja maior do que a tolerância a solução é descartada e a iteração n é recomputada com o h_{n+1}. Caso contrário, o método obtém o novo passo de tempo e segue para a iteração $n + 1$.

A segunda maneira de adaptação é obtida a partir de uma heurística inicialmente proposta por Minkoff e Kridler[49] e testada com modelos da eletrofisiologia cardíaca em [50, 51]. Nesta heurística é calculada uma iteração com o método de Euler e logo em seguida calcula-se a mesma iteração com o método de RK2. O passo de tempo é obtido através do algoritmo apresentado na Figura 4.9 que utiliza uma tolerância e o erro entre as soluções de cada método, na mesma iteração. A heurística é muito simples e consiste em aumentar o passo de tempo h se o erro for menor que metade da tolerância tol. Se o erro está entre $\frac{tol}{2}$ e tol, mantém-se h inalterado. Se o erro está entre tol e $2tol$ diminui-se h e quando o erro é maior que o dobro da tolerância, h é dividido por 2 e a solução computada nesta iteração y_{n+1} é descartada e computada novamente com o novo h. Foram testados diversos valores para multiplicar h, com o objetivo de aumentá-lo ou diminui-lo para a próxima iteração. Respectivamente, os valores que resultaram melhores desempenhos foram 1,5 e 0,65.

Ao se resolver um sistema de equações é necessário computar o erro para cada uma das variávies e selecionar um dos erros para adaptar o passo de tempo, tanto pela fórmula quanto pela heurística. Como as variáveis possuem diferentes escalas de grandeza, o erro absoluto pode ser uma má escolha, pois a variáveis com maiores valores produzirão maiores erros, comparando-se com as variáveis de menor valor. Estas variáveis com valores menores podem produzir baixos erros absolutos, mas altos percentuais de erro relativo, o que indica que está havendo uma instabilidade maior com elas, e não com as variáveis

```
1  SE ( erro < 0.5 * tolerância ) FAÇA
2       Aceitar y_{n+1}⃗;
3       Aumentar h;
4  SENÃO SE ( erro ≥ 0.5 * tolerância E erro < tolerância ) FAÇA
5       Aceitar y_{n+1}⃗;
6       Manter h;
7  SENÃO SE ( erro ≥ tolerância E erro < 2 * tolerância ) FAÇA
8       Aceitar y_{n+1}⃗;
9       Diminuir h;
10 SENÃO FAÇA
11      Descartar y_{n+1}⃗;
12      h = h/2;
13      Refazer o passo;
14 FIM–SE
```

Figura 4.9: Heurística do método adaptativo

maiores. Por outro lado, se apenas o erro relativo for considerado, pode ocorrer que todas as variáveis produzam níveis de erro muito próximos de zero, o que também pode ser prejudicial para escolha ideal da variável que está sobre maior instabilidade. Para amenizar esta questão, este trabalho utiliza um esquema de tolerância variável. A cada iteração é escolhida uma tolerância para cada variável que deve ser o maior valor entre a tolerância absoluta e o produto da tolerância relativa com a solução y^j_{n+1}. Estes valores são armazenados no vetor $\overrightarrow{tols_{n+1}}$. Após isto, determina-se a tolerância tol_{n+1}, que deve ser o maior valor em $\overrightarrow{tols_{n+1}}$.

$$\overrightarrow{tols_{n+1}} = max(relTol\overrightarrow{\|y_{n+1}\|}, absTol)$$
$$tol_{n+1} = max(tols_{n+1}) \tag{4.7}$$

Onde relTol e absTol são as tolerâncias relativas e absolutas determinadas pelo usuário, e $\overrightarrow{y_n}$ é solução de todas as variáveis na iteração n.

Para ambos os métodos adaptativos, intuitivamente, computa-se o lado direito duas vezes em cada iteração, uma para o Método de Euler no instante t_n, e outra para o Método RK2, no instante t_{n+1}. Porém, isto dobraria o custo computacional destes métodos. Uma maneira de resolver este problema é reutilizar $f(t_{n+1})$ da iteração n, como $f(t_n)$ na iteração seguinte $n+1$. O método modificado, que utiliza apenas uma avaliação do lado direito por iteração e utiliza a tolerância da Equação 4.7 pode ser encontrado na Figura 4.10. Na linha 1 desta figura o lado direito das equações é computado pela primeira vez e o

```
1  LD1 = lado_direito( t0, y0 );
2  ENQUANTO (tn < tempoFinal) FAÇA
3     y(n+1) = LD1 * hn + yn;
4     LD2 = lado_direito(tn+1, y(n+1));
5     tols(n+1) = max(relTol * ||y(n+1)||, absTol);
6     tol(n+1) = max(tols(n+1));
7     erro(n+1) = max(0.5 * ||LD1 - LD2||);
8     hn+1 = obtém_novo_h(erro(n+1), tol(n+1));
9     SE(erro(n) < tol(n+1) ) FAÇA
10       yn = y(n+1);
11       LD1 = LD2;
12       tn = tn + hn;
13    SENÃO FAÇA
14       //nao incrementa o tempo,
15       //para que este passo seja refeito.
16    FIM–SE
17    hn = hn+1;
18 FIM–ENQUANTO
```

Figura 4.10: Método adaptativo

resultado é armazenado no vetor LD_1. Após isto, o laço do método iterativo se inicia e então é computado o método de Euler que é armazenado no vetor y_{n+1}, o que ocorre na linha 3. Na linha seguinte, é computado o novo lado direito LD_2 que será utilizado nesta equação para calcular o erro e será utilizado na iteração seguinte para computar o método de Euler. Nas linhas 5 e 6 é encontrada a tolerância tol_{n+1} e na linha 7, o $erro_{n+1}$. Na linha 8 é encontrado o novo passo de tempo h_{n+1}, onde a função obtém_novo_h calcula h_{n+1} pela fórmula ou pela heurística apresentadas anteriormente. Na linha 9 verifica-se se o erro é aceitável, em caso positivo, realizam-se duas trocas de ponteiro de vetores. A primeira faz com que o vetor y_n receba os valores de y_{n+1} e a segunda coloca o conteúdo do vetor com o lado direito LD_2 no vetor LD_1. Então a variável do tempo é incrementada e depois a variável do passo de tempo h_n é atualizada para o valor h_{n+1}, quando enfim o método segue para a próxima iteração. Caso o erro não seja aceitável, não se trocam os vetores de solução e passo de tempo, pois estes devem ser descartados e computados novamente. Também não se incrementa a variável do tempo e após h ser atualizado repete-se a iteração.

Os desempenhos dos métodos adaptativos propostos nesta seção serão apresentados no capítulo seguinte.

4.2.2 AGOS + OpenMP

Uma maneira de diminuir o tempo de execução das simulações é explorar os recursos de ambientes multiprocessados, através de técnicas de computação paralela. Este trabalho propõe a utilização do OpenMP para paralelizar a execução dos métodos numéricos propostos acima, através de uma adaptação na ferramenta AGOS. A ideia é que o AGOS, ao ler um arquivo CellML para transformá-lo em C++, automaticamente gere os métodos adaptados para serem executados em paralelo. Ele deve dividir as equações entre os processadores de maneira que elas sejam executadas simultaneamente. Entretanto, a divisão de tarefas para equações diferenciais ordinárias é uma tarefa complicada, por causa da avaliação parcial usada pelo AGOS. Como visto nas seções anteriores, os métodos numéricos que resolvem estas equações são iterativos e sempre dependem do passo de tempo anterior. Esta característica impossibilita que o laço que incrementa a variável do tempo seja paralelizado, pois uma iteração n + 1 sempre precisará dos valores que foram calculados na iteração n. Então, este trabalho propõe uma maneira de paralelizar a avaliação do lado direito das equações, dentro de uma única iteração. A idéia é agrupar as expressões de maneira que uma expressão só dependa de variáveis que estejam no seu mesmo grupo. Assim, um grupo tem suas expressões avaliadas sequencialmente, porém vários grupos podem ser executados paralelamente.

Para alcançar este objetivo foi implementado no AGOS um grafo para representar a dependência entre as equações. Um grafo é um conjunto de objetos, chamados vértices, conectados por ligações denominadas arestas [52]. Neste trabalho os vértices são as equações do modelo - diferenciais ou algébricas e as arestas representam a dependência que há entre as equações. O algoritmo apresentado na Figura 4.11 ilustra a montagem do grafo de dependências, onde para cada equação, percorre-se todas as suas variáveis. Se uma variável algébrica é encontrada, adiciona-se no grafo uma aresta entre a equação e esta variável, indicando que a equação EQ depende da variável VAR para ser computada. O algoritmo avalia equações algébricas e diferenciais para encontrar as variáveis que elas precisam para serem computadas. Porém, dentro de uma única iteração foi considerado que apenas variáveis algébricas causam dependência. Isso ocorre porque em uma iteração $t + 1$ todas as equações utilizam valores das variáveis diferenciais que foram computadas na iteração t. Ou seja, as variáveis diferenciais computadas em t não são utilizadas nesta iteração, mas apenas serão utilizadas em $t + 1$. Portanto, é possível afirmar que dentro

```
1 PARA CADA (equação EQ) FAÇA
2   adiciona_vértice(EQ);
3   PARA CADA (variável VAR pertencente a EQ) FAÇA
4       SE (VAR é algébrica) FAÇA
5         adiciona_arco(EQ, VAR);
6       FIM–SE
7   FIM–PARA
8 FIM–PARA
```

Figura 4.11: Montagem do grafo de adjacências das equações

de uma única iteração nenhuma equação, algébrica ou diferencial, depende de variáveis diferenciais, pois sempre serão utilizados os valores que foram computados na iteração anterior. Todavia, as variáveis devem ser sincronizadas ao final de cada iteração, pois no próximo passo os seus valores serão utilizados pelas equações. Também foram desconsiderados os parâmetros, que não alteram seus valores no decorrer da simulação. A implementação foi feita através de uma lista de adjacências que consiste em um arranjo que contém todos os vértices. Para cada vértice V, existe uma lista de adjacências que contém ponteiros para os vértices que possuam uma aresta com V. Ou seja, para cada equação, existe uma lista com as variáveis que esta equação depende.

Após a montagem do grafo que determina a dependência entre as equações, deve-se agrupá-las de modo que em um grupo exista apenas equações que possuam alguma dependência entre si. Para tal, o primeiro vértice V_1 é marcado por um rótulo R_1. Com este mesmo rótulo marcam-se todas as adjacências do vértice, significando que a equação do vértice V_1 e todas as variáveis que ela depende estão no mesmo grupo. Também deve-se marcar com R_1 as equações que dependem de V_1, então todos os vértices são percorridos e se algum deles possuir V_1 em suas adjacências marca-se este com R_1 também. Este algoritmo é executado recursivamente para cada equação marcada com R_1, desde que o vértice em questão não esteja rotulado, até que todos os vértices que dependam de V_1, ou que V_1 dependa, sejam marcados. Então o algoritmo deve encontrar o próximo vértice que não tenha sido rotulado, para então repetir este procedimento até que todos os vértices possuam algum rótulo. Este algoritmo pode ser visualizado na Figura 4.12.

Por exemplo, considerando o sistema de Equações 4.8, o algoritmo avalia as dependências que são: X depende de I; I e B não dependem de nenhuma variável e Y depende de B. O próximo passo é agrupar estas equações e para tal, atribui-se um rótulo que identifica um grupo à primeira variável. Neste exemplo, marca-se X com o rótulo 1 e

```
1  RTL = 1;                          //escolhe o rótulo inicial
2  PARA CADA (vértice V) FAÇA //percorre cada vértice do grafo
3     rotular_adjacências(V);
4     RTL += 1;
5  fim-para
6
7  //rotula as adjacências de V
8  FUNÇÃO rotular_adjacências(rótulo RTL, vértice V) FAÇA
9     SE (V não possui rótulo) FAÇA
10       v.rótulo = R1; //marca vértice
11    FIM-SE
12    PARA CADA (vértice VV adjacente de V) FAÇA
13       SE (VV não possui rótulo) FAÇA
14          rotular_adjacências(RTL, VV);
15       FIM-SE
16    FIM-PARA
17    rotular_vértices_V_adjacente(RTL, V);
18 FIM-FUNÇÃO
19
20 //rotular vértice em que V está na lista de adjacências
21 FUNÇÃO rotular_vértices_V_adjacente(rótulo RTL, vértice V1) FAÇA
22    PARA CADA (vértice V) FAÇA
23       SE (V posui V1 em sua lista de adjacências) FAÇA
24          rotular_adjacências(RTL, V);
25       FIM-SE
26    FIM-PARA
27 FIM-FUNÇÃO
```

Figura 4.12: Agrupamento de equações dependentes através de rotulação em grafo

depois marca-se a sua lista de adjacências, que contém apenas a variável I. Então verifica-se se X e I estão nas adjacências de outras equações. Como não estão, incrementa-se o rótulo e avalia-se a próxima expressão, repetindo este processo. Claramente há dois grupos: um contém X e I e o outro, B e Y.

$$\begin{aligned} \frac{\partial X}{\partial t} &= \frac{I}{5} \\ I &= Y^3 X \\ B &= \frac{-0.1X}{X+50} \\ \frac{\partial Y}{\partial t} &= \frac{B+3}{10} \end{aligned} \tag{4.8}$$

Depois das equações terem sido agrupadas, deve-se adicionar ao código C++ as dire-

```
1 while(tempo < tempoFinal){
2   #pragma omp parallel for
3   {
4     tempo += h;
5     for(i=0;i<numGrupos;i++)
6       executarGrupo(i);
7   }
8 }
```

Figura 4.13: Paralelização dos grupos de equações - seção paralela interna ao laço

tivas OpenMP que permitem que estes grupos de equações executem paralelamente. Na Seção 3.4.4 foram apresentadas diversas maneiras de dividir o trabalho que um programa deve realizar entre processadores. Foram realizadas várias tentatizas de paralelização que serão apresentadas a seguir. A primeira consiste na mais intuitiva maneira de paralelização deste problema, porém a mais ineficiente.

Ela consiste em criar a seção paralela dentro do laço que incrementa o tempo da simulação. Isto é prejudicial para o desempenho, pois as *threads* são criadas e destruídas a cada iteração, o que aumenta muito o custo de manutenção das mesmas. Este código pode ser visto na Figura 4.13.

Para que não ocorra criação de *threads* a cada iteração deve-se colocar o comando *#pragma omp parallel* acima do laço do tempo. Porém, esta modificação implica em outras mudanças no código que devem garantir que as partes sequenciais do programa sejam de fato executadas sequencialmente. Por exemplo, o incremento da variável tempo deve ser colocado dentro de uma diretiva para garantir que ela seja incrementada apenas uma vez por iteração (*#pragma omp single*). Outro problema é a sincronização, pois deve-se garantir que a iteração seguinte somente inicie após que todas as variáveis diferenciais da iteração atual já tenham sido computadas. Para resolver o problema da variável do tempo foi criada uma variável privada para cada *thread*, para que cada uma delas atualize a sua própria variável. Esta solução foi escolhida porque uma variável global exigiria um comando de sincronização extra, como o *single*, para garantir que o tempo seja incrementado apenas uma vez por iteração. Existe também o problema de invalidação de *cache*, que ocorre ao se atualizar uma variável global. Quando uma *thread* faz isto é preciso que todas as outras atualizem a sua *cache*, para que a coerência de memória seja mantida. Este procedimento atrapalha o desempenho, pois a atualização de memória consome alguns ciclos de *clock* e neste trabalho a variável será atualizada a cada iteração,

o que produziria uma atualização para cada *thread*, a cada iteração. Já o problema de sincronização foi resolvido com uma barreira. Todos os processadores recebem o seu trabalho, executam-no e aguardam todos os outros terminarem para iniciar uma nova iteração. O novo código está ilustrado na Figura 4.14. Na linha 2 declara-se a variável privada prv_tempo, e esta recebe o valor inicial que está na variável global tempo. A partir de então a variável global não é mais acessada. Não foi necessário escrever o comando da barreira, pois há uma barreira implícita no final do comando *for*.

Com esta implementação foram avaliadas duas maneiras de divisão do trabalho disponíveis com o comando *for*. A primeira maneira é a divisão estática (*schedule(static)*), que divide igualmente o número de tarefas entre os processadores, independente do tempo que cada uma gaste para ser computada. Para este problema a divisão estática não foi uma boa escolha, pois as tarefas possuem diferentes tamanhos e esta divisão pode concentrar muito trabalho em um só processador, fazendo com que os demais fiquem ociosos. Então foi testada a divisão dinâmica que envia uma tarefa de cada vez para os processadores. Esta estratégia de divisão do trabalho foi capaz de distribuir melhor as tarefas. Entretanto, ela adiciona custos extras de processamento, pois deve verificar se uma *thread* terminou de computar o seu trabalho e escolher uma tarefa para enviar a um processador. Além disto, este procedimento de divisão pode não ser eficiente quando as tarefas a serem processadas não estão ordenadas por custo computacional. Por exemplo, um grupo de quatro tarefas deve ser enviado para 2 processadores. As tarefas demoram 5, 1, 2, e 6 segundos (s) para serem computadas. Pela divisão dinâmica as tarefas de 5 e 1 s são enviadas para os processadores. A segunda *thread* terminará seu trabalho e deverá receber a próxima tarefa que é de 2 s. Quando esta terminar, a primeira *thread* ainda estará computando a sua tarefa de 5 s, e então a segunda receberá a tarefa de 6s. Ou seja, a primeira *thread* ficará ocupada por 5 s, enquanto a segunda, por 9 s. Como este procedimento se repete a cada iteração, o resultado final será desastroso para o desempenho do método. Também foi testada a divisão de tarefas através da diretiva *section*, mas o problema da divisão de tarefas se repetiu.

Portanto, se fez necessário um algoritmo para dividir as tarefas da maneira mais justa possível, mas adicionando o mínimo de custo extra. Este trabalho propõe uma divisão de trabalho *a priori*, que dividirá as tarefas entre as *threads* antes que o laço do tempo se inicie, mas considerando o tempo de computação necessário para executá-las. Para

```
1 #pragma omp parallel
2   double tempo_local = tempo;
3   while(tempo_local < tempoFinal){
4   #pragma omp for
5     tempo_local += h;
6     for(i=0;i<numGrupos;i++)
7       executarGrupo(i);
8 }
```

Figura 4.14: Paralelização dos grupos de equações - seção paralela externa ao laço

isso, foi feito um passo adicional antes do laço, que computa o tempo gasto por cada tarefa. Para isso, cada grupo de equações foi executado 100 vezes e o tempo de cada um foi medido. A partir de então haverá um vetor com o tempo gasto por cada tarefa, e deve-se dividir as tarefas de maneira mais justa possível entre n *threads*. Este problema é semelhante ao problema da partição, que consiste em dividir igualmente um conjunto de número inteiros em dois subconjuntos disjuntos, onde ambos devem ter a mesma soma. Neste trabalho, os elementos do conjunto a ser dividido são os tempos de cada tarefa, e os subconjuntos disjuntos são as *threads*, que devem receber o mesmo somatório de tempo de execução. Entretanto, ao contrário do problema clássico da partição, o algoritmo precisa que o número de subconjuntos seja variável, pois é desejável que se distribua tarefas para qualquer número de *threads*.

Os problemas de otimização como este acima são resolvidos por algoritmos que podem ser classificados de acordo com a sua tratabilidade. Esta característica está relacionada com o tempo de execução do algoritmo. Para os problemas mais simples, dada uma entrada de tamanho n, ele será resolvido em tempo $\mathcal{O}(n^k)$, onde k é uma constante, o que é chamado de tempo polinomial. Os problemas deste tipo pertencem à classe P. Já os problemas intratáveis, que são mais difíceis, são resolvidos em tempo superpolinomial. A segunda classe de problemas é chamada de NP. Ela contém os problemas em que é simples certificar-se que uma solução é correta. Para que um problema esteja contido nesta classe o tempo de verificação da solução deve ser polinomial e não é considerado o tempo de encontrar a solução. Um subconjunto importante de NP é a classe NP-completo. Para que um problema pertença a esta classe ele deve estar contido em NP e ser "tão difícil quanto qualquer problema em NP"[52]. Ou seja, um problema deve ser tão custoso computacionalmente quanto os problemas contidos em NP. Uma peculiaridade desta classe é que se um problema NP-completo pode ser resolvido em tempo polinomial,

todos os problemas pertencentes a este conjunto também poderão ser resolvidos em tempo polinomial[53]. Entretanto, há fortes indícios de que tais problemas são intratáveis, pois ainda não foram encontrados algoritmos com tempo polinomial para resolvê-los[52]. O problema de divisão de tarefas proposto neste trabalho é classificado como NP-completo, o que significa que não há solução em tempo polinomial para resolvê-lo.

Para dividir as tarefas entre as *threads* de maneira eficiente, este trabalho utilizou uma estratégia gulosa. Esta estratégia é uma heurística simples de se implementar e geralmente é computacionalmente barata[52]. Ela pode ser aplicada em algoritmos que resolvem problemas de otimização. O algoritmo desta estratégia encontra soluções parciais iterativamente até que a solução final seja encontrada. A cada iteração é escolhida a solução ótima até aquele ponto. Isto significa que o algoritmo faz uma escolha que considera ótima em uma iteração, porém não leva em conta as consequências desta escolha nas próximas iterações. Ele jamais pode voltar atrás para tentar reencontrar uma solução melhor para uma iteração passada, o que quer dizer que as decisões dele são definitivas. Esta estratégia nem sempre encontra a solução ótima. Embora existam heurísticas mais robustas como programação dinâmica, a estratégia gulosa foi escolhida por possuir menor custo computacional[53]. De qualquer maneira, os experimentos realizados sugeriram que o algoritmo dividiu as tarefas satisfatoriamente.

O algoritmo proposto pode ser encontrado na Figura 4.15. Na linha 1 é criado um vetor para armazenar o tempo de cada tarefa e na linha 2, um vetor que associa uma tarefa a uma *thread*. Na linha 3 é feita a medição do tempo e na linha 4 o vetor com os tempos é ordenado em ordem decrescente. Em seguida, é criado um vetor de subconjuntos, com tamanho igual ao número de processadores desejado. Então executa-se a estratégia gulosa, que consiste em atribuir as tarefas começando da maior para a menor, sendo que o subconjunto escolhido para receber uma tarefa é aquele que possui o menor somatório, o que está sendo mostrado nas linhas 6 a 10.

Este algoritmo é executado apenas uma vez por simulação, antes do laço dos métodos de resolução. Ele acrescenta um pequeno custo, mas é vantajoso em relação às estratégias descritas acima. Após a divisão de tarefas, cada grupo de equações estará associado a uma *thread* e somente será executado por ela. O grupo de equações contém as expressões do lado direito e o cálculo do método de Euler em si é dividido de acordo com a respectiva expressão. O método de Euler com OpenMP está descrito na Figura 4.16. A função

```
1  tempos[numTarefas];                  //vetor com os tempos de cada tarefa
2  tarefas[numTarefas];                 //vetor com o número da thread a ser
       associada a uma tarefa
3  medir_tempo_tarefas(tempos);         //mede o tempo das tarefas
4  ordenar_vetor(tempos);               //ordena os vetores em ordem decrescente
5  subConjuntos[numeroThreads];         //cria um vetor com o tamanho do número de
       threads
6  subConjuntos = 0;                    //inicializa as posições do vetor com zero.
7  PARA i de 0 ATÉ numTarefas FAÇA
8    min = encontrar_mínimo(subConjuntos); //encontra o índice do subconjunto
         que possui menor somatório
9    subConjuntos[min] += tempos[i];        //soma a tarefa no subconjunto
10   tarefas[i] = min;                  //tarefa i associada a thread de número min
11 FIM-PARA
```

Figura 4.15: Estratégia gulosa para a divisão de trabalho

omp_get_thread_num() na linha 7, retorna o identificador de uma *thread* e o comando SE nas linhas 7 e 13, garante que um determinado grupo de expressões do lado direito só será executado pela *thread* que foi associada pelo algoritmo guloso. Uma barreira se fez necessária para sincronização já que o *parallel for* foi retirado. Os índices numéricos sobrescritos nos vetores $\overrightarrow{LD}$, $\overrightarrow{y_n}$ e $\overrightarrow{y_{n+1}}$ indicam que estes vetores estão associados ao grupo com o mesmo número.

Os métodos adaptativos funcionam de maneira semelhante, mas exigem uma barreira adicional, pois após o cálculo do lado direito, deve-se sincronizar os resultados para se calcular o novo passo de tempo e logo depois deve haver uma nova sincronização para que a próxima iteração inicie com todas as *threads* possuindo o mesmo valor de h. O esquema dos métodos adaptativos está ilustrado na Figura 4.17.

Os resultados da combinação do AGOS com OpenMP serão apresentados no Capítulo 5. Este trabalho implementou a avaliação de expressões do lado direito em paralelo com OpenMP na ferramenta AGOS. Como trabalho futuro pretende-se adaptar a ferramenta PyCML para que esta também realize a avaliação do lado direito em paralelo, possibilitando a combinação de OpenMP com a técnica de *Lookup Tables*.

4.2.3 AGOS + PyCML

A junção do AGOS com a ferramenta PyCML foi proposta e avaliada em [2, 54]. Entretanto, a junção proposta foi feita manualmente, copiando e colando o código necessário, o que é uma tarefa dispendiosa, principalmente para modelos com muitas equações. Este trabalho propõe uma forma de unir as duas ferramentas automaticamente de maneira que

```
1  tarefas[numTarefas]
2  divisao_gulosa(tarefas);
3  #pragma omp parallel
4    t_n = tempo;     //t_n é uma variável privada
5    ENQUANTO(t_n < tempoFinal) FAÇA
6      t_n += h;
7      SE (tarefa[0]== omp_get_thread_num()) FAÇA
8        //so computa o lado direito e o método de Euler
9        //das variáveis associadas a esta thread
10       LD^0 = executarGrupo(0);
11       y^0_{n+1} = LD^0 * h + y^0_n;
12     FIM-SE
13     SE(tarefa[1]== omp_get_thread_num())  FAÇA
14       LD^1 = executarGrupo(1);
15       y^1_{n+1} = LD^1 * h + y^1_n;
16     FIM-SE
17     // ... repete-se este SE para cada grupo
18     #pragma omp barrier
19   FIM-ENQUANTO
20 FIM-PARALLEL
```

Figura 4.16: Método de Euler com OpenMP

o PyCML gere os lados direitos utilizando as técnicas de avaliação parcial e *Lookup Tables* e o AGOS seja capaz de executá-las. Inicialmente, a junção era feita apenas substituindo o código da função $f(t, y)$ do AGOS que computa o lado direito. Então o PyCML foi alterado para apenas imprimir o lado direito das equações. Isso foi necessário porque esta ferramenta foi desenvolvida como parte integrante do ambiente Chaste[9, 34, 55] e existem algumas partes do código gerado que não são necessárias para o AGOS. O código da função $f(t, y)$ do AGOS foi mantido e o construtor da classe, que contém os métodos do AGOS, recebe um parâmetro que indica qual lado direito ele deve computar, ou seja, o lado direito do AGOS ou do PyCML com avaliação parcial ou com *Lookup Tables*.

O resultado desta junção pode ser visualizado através do Portal do Fisiocomp, onde os códigos-fonte podem ser copiados ou executados através da interface *Web*. Os desempenhos das técnicas adicionadas ao AGOS serão avaliadas no próximo capítulo.

```
1  tarefas[numTarefas]
2  divisao_gulosa(tarefas);
3  LD_1 = lado_direito( t_0, y_0 );
4  #pragma omp parallel
5    t_n = tempo;     //t_n     uma variável privada
6    ENQUANTO(t_n < tempoFinal) FAÇA
7      SE(tarefa[0]== omp_get_thread_num()) FAÇA
8        //computa o lado direito e o método de Euler
9        //somente das variáveis associadas
10       y^0_{n+1} = LD^0_1 * h_n + y^0_n;
11       LD^0_2 = executarGrupo(0);
12     FIM-SE
13     // ... repete-se este SE para cada grupo
14     #pragma omp barrier
15     tols_{n+1} = max(relTol * ||y_{n+1}||, absTol);
16     tol_{n+1} = max(tols_{n+1});
17     erro_{n+1} = max((0.5 * ||LD_1 - LD_2||));
18     h_{n+1} = obtém_novo_h(erro_{n+1}, tol_{n+1});
19     SE (erro_n < 1 ) FAÇA
20       y_n = y_{n+1};
21       LD_1 = LD_2;
22       tempo = t_n + h_n;
23     SENAO FAÇA
24       //nao incrementa o tempo,
25       //para que este passo seja refeito
26     FIM-SE
27     h_n = h_{n+1};
28     #pragma omp barrier
29   FIM-ENQUANTO
30 FIM-PARALLEL
```

Figura 4.17: Método com passo de tempo adaptativo com OpenMP

5 RESULTADOS

Neste capítulo serão apresentados os resultados das soluções propostas anteriormente. Primeiro, são apresentados uma análise qualitativa do Editor e um novo modelo computacional criado através da utilização desta ferramenta. A segunda parte do capítulo faz uma análise quantitativa do desempenho dos métodos e técnicas de resolução de equações diferenciais, avaliando a velocidade em que estes foram executados, além de ponderar a respeito da relação custo-benefício entre os métodos considerando desempenho, o uso de memória e o erro numérico.

5.1 Editor

O Editor de CellML proposto neste trabalho está disponível na *Web*, através do *site* do Fisiocomp[1]. Ele contribui para facilitar a edição de modelos descritos em CellML, que pode ser uma tarefa dispendiosa, principalmente para modelos compostos por muitas equações. Portanto, a parte do Editor que transforma as equações descritas em CellML para a linguagem proposta neste trabalho e vice-versa é o item que mais contribuiu para facilitar a edição de modelos. Será feita uma análise qualitativa através de um exemplo de utilização da ferramenta.

Esta ferramenta já está funcional e foi utilizada no desenvolvimento do artigo de Campos et al.[28] e na dissertação de Oliveira[4], onde um novo modelo computacional foi proposto através da combinação de componentes de dois outros modelos. Inicialmente, deve-se efetuar o *download* destes dois modelos no repositório do CellML ou no *site* do Fisiocomp. O primeiro deles é o modelo de ten Tusscher et al. [2] [56], que simula o comportamento eletrofisiológico de células do coração. O segundo é o modelo de Rice et al. [3] [57], que representa o fênomeno de contração mecânica das células cardíacas. A partir da junção de ambos os modelos será possível simular a interferência que a propagação do potencial de ação tem sobre o desenvolvimento da força mecânica que ocorre nos miócitos para que o coração se contraia. O processo de contração das células é iniciado quando

[1] Disponível em $http://www.fisiocomp.ufjf.br$
[2] Disponível para *download* em $http://www.fisiocomp.ufjf.br/modelos/ten\_et\_al.xml$
[3] Disponível para *download* em $http://www.fisiocomp.ufjf.br/modelos/rice\_et\_al.xml$

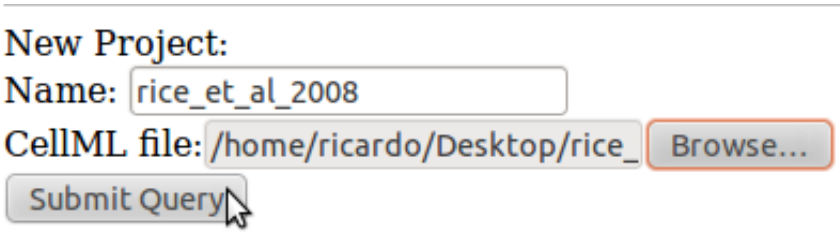

Figura 5.1: Submissão de arquivos CellML

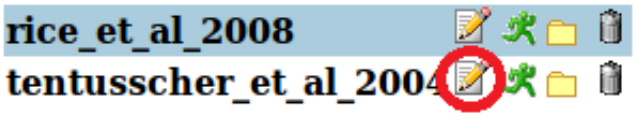

Figura 5.2: Repositório de modelos do AGOS

a concentração intracelular de cálcio aumenta, o que faz com que este íon se ligue a certas proteínas, como Troponina C e esta reação química, juntamente com o consumo de energia, proporciona a geração de força mecânica.

O primeiro passo para a criação do novo modelo é acessar o *link* "*FisioTools Login*", disponível no *site* do Fisiocomp. É necessário um breve cadastro do usuário para que se tenha acesso às funcionalidades do AGOS. Após acessar o repositório, o usuário pode submeter os arquivos com extensão XML que descrevem os modelos de Rice et al.[57] e de Ten Tusscher et al.[56]. A Figura 5.1 ilustra a submissão de um arquivo.

Após o envio dos arquivos, o AGOS os transforma em código C++ e já é possível simulá-los pela interface. A Figura 5.2 mostra o repositório com os dois modelos enviados.

Para iniciar a criação do novo modelo, deve-se editar o modelo ten Tusscher et al.[56] através do ícone de edição, que está destacado em vermelho na Figura 5.2.

A partir deste momento, a interface do editor estará visível com os componentes e unidades do modelo, o que pode ser visualizado na Figura 5.3.

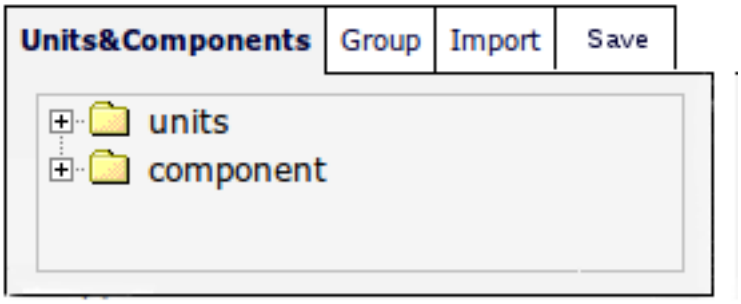

Figura 5.3: Interface do editor

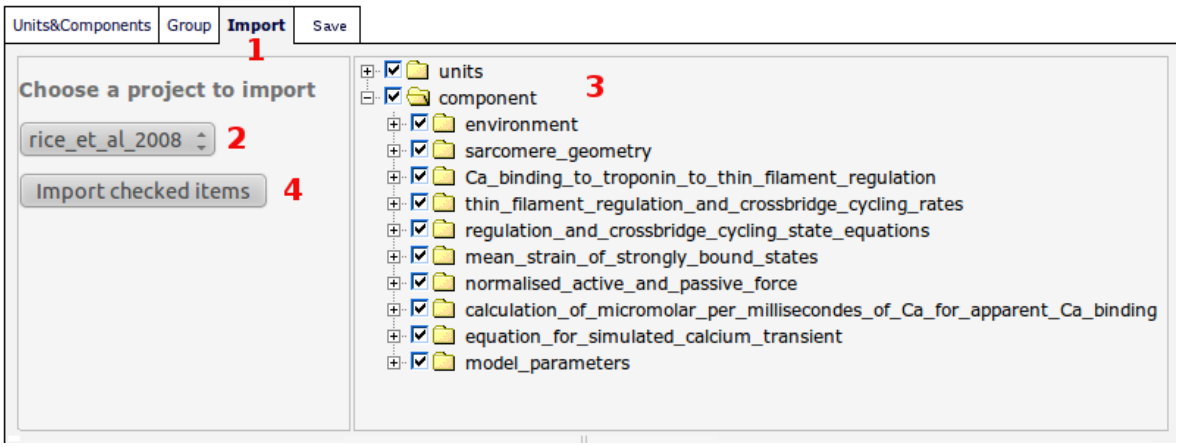

Figura 5.4: Importação de Rice et al.

O próximo passo é copiar todos os componentes e unidades do modelo de Rice et al.[57], o que pode ser feito pela aba "*Import*" (1), na Figura 5.4. A partir desta janela deve-se escolher o modelo a ser importado (2) que então será exibido em forma de árvore. Em seguida é necessário selecionar todos os componentes e unidades que serão copiados (3). Feito isto, deve-se efetuar a cópia através do botão "*Import checked items*" (4).

Após uma mensagem de confirmação avisando que os itens selecionados foram copiados, deve-se voltar para a aba "*Units & Components*", para que se possa dar prosseguimento à criação do modelo. O próximo passo consiste em alterar as equações, com intuito de fazer com que a corrente de cálcio do modelo eletrofisiológico seja influenciada pela ligação de cálcio à troponina. Esta ligação é descrita pelo modelo mecânico.

Primeiro é preciso alterar um componente chamado "*equation_for_simulated_calcium_transient*", que originalmente pertencia ao modelo de Rice et al.[57]. Ele possui a seguinte equação:

$$Cai = \begin{cases} \frac{Ca_{amplitude} - Ca_{diastolic}}{beta}(e^{-\frac{time - start_{time}}{tau2}}) + Ca_{diastolic} & \text{se} \quad time > start_{time} \\ Ca_{diastolic} & \quad caso\ contrário \end{cases} \tag{5.1}$$

Que deve ser alterada para:

$$Cai = 1000 Ca_i \tag{5.2}$$

Para que isto seja feito pelo editor basta procurar pelo componente em questão (Figura 5.5). Dentro dele há um nó chamado "*math*" (1) que ao ser selecionado exibe uma janela com as equações deste componente (2). Então basta identificar a equação Ca_i, alterá-la

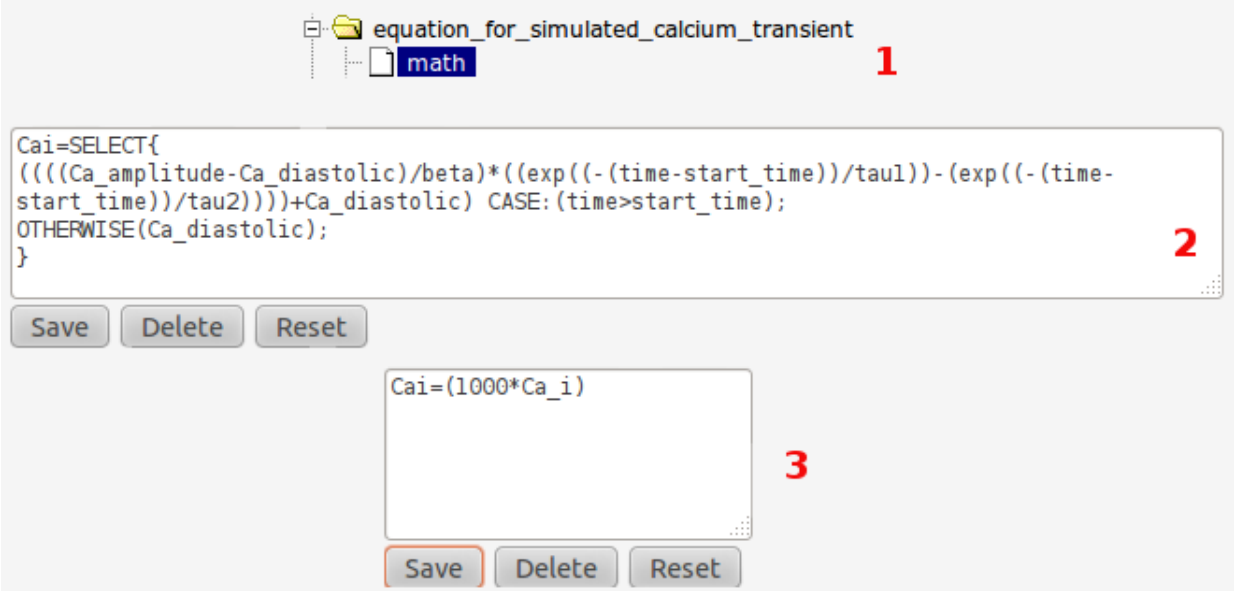

Figura 5.5: Modificação de Equações

e salvá-la (3).

Também é necessário alterar algumas equações do modelo eletrofisiológico que estão no componente "*calcium_dynamics*". Originalmente elas eram dadas por:

$$\frac{\partial g}{\partial time} = \begin{cases} 0 & \text{se} \quad g_{inf} > g \ e \ V > -60 \\ d_g & caso\ contrário \end{cases} \tag{5.3}$$

$$\frac{\partial Ca_i}{\partial time} = Ca_{i\_bufc}(i_{leak} - i_{up} + i_{rel} - \frac{i_{CaL} + i_{b\_Ca} + i_{p\_Ca} - 2i_{NaCa}}{2V_cF}Cm) \tag{5.4}$$

Estas equações devem ser alteradas para:

$$\frac{\partial g}{\partial time} = \begin{cases} 0 & \text{se} \quad 0.01d_g > 0 \ E \ V > -60 \\ d_g & caso\ contrário \end{cases} \tag{5.5}$$

$$Ca_i = \frac{-bc + (bcbc - 4cc)^{0.5}}{2} \tag{5.6}$$

Após isto, é necessário adicionar mais três equações ao sistema:

$$\frac{\partial Ca_{tot}}{\partial time} = i_{leak} - i_{up} + i_{rel} - \frac{i_{CaL} + i_{b\_Ca} + i_{p\_Ca} - 2i_{NaCa}}{2V_cF}Cm \tag{5.7}$$

$$cc = (\frac{TropTot}{1000} - Ca_{tot})K_{buf\_c} \tag{5.8}$$

$$bc = K_{buf\_c} + Buf_c + \frac{TropTot}{1000} - Ca_{tot} \tag{5.9}$$

Figura 5.6: Declaração de variável no componente

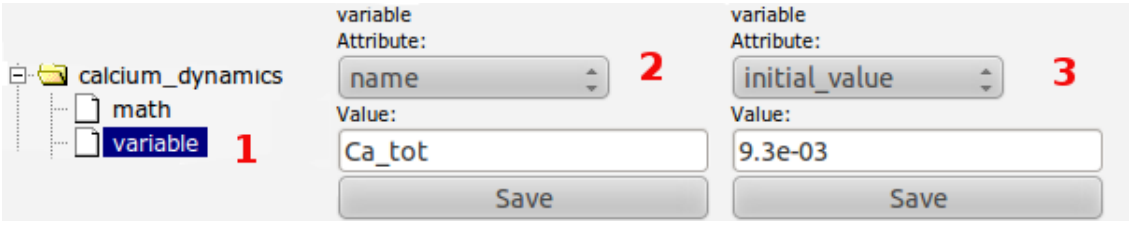

Figura 5.7: Alteração dos atributos de uma variável

Estas equações podem ser alteradas de maneira semelhante ao que foi feito com a equação *Cai*. Para adicionar novas equações existe um *link* "*Add new equation*" ao final da janela de edição das equações.

É recomendável que as variáveis adicionadas nas equações sejam declaradas no componente para que o arquivo CellML possua os valores iniciais, unidades e outras informações. Para adicionar a variável deve-se seguir os passos descritos na Figura 5.6. Primeiro escolhe-se o componente em que a variável está sendo utilizada (1). Em seguida, deve-se acionar o *link* "*Add tag*", que está localizado no painel de ações ao lado esquerdo da interface do editor (2). Após isto aparecerá um painel no lado direito da interface, onde é necessário escolher o tipo de *tag* a ser inserida. Neste caso, deve-se escolher o tipo "*variable*" (3).

Uma vez que a variável foi adicionada ao componente é possível alterar o seus atributos. A Figura 5.7 ilustra este procedimento. Primeiro seleciona-se a variável a ser editada (1). Então será exibido no lado direito do editor uma lista de opções com os atributos disponíveis para variáveis. O primeiro atributo a ser alterado é nome da variável. Deve-se se escolher a opção "*name*", escrever o nome Cai_tot no campo "*value*" e salvar a operação através do botão "*Save*" (2). Para alterar o valor inicial, basta escolher a opção "*initial_value*", lançar o valor e salvar a operação (3). Neste caso, o valor inicial da variável é "$9.3e-3$".

Finalmente, o modelo eletromecânico está criado e o próximo passo é salvá-lo no repositório (Figura 5.8). Esta opção está disponível na aba "*Saving*" (1), onde é possível

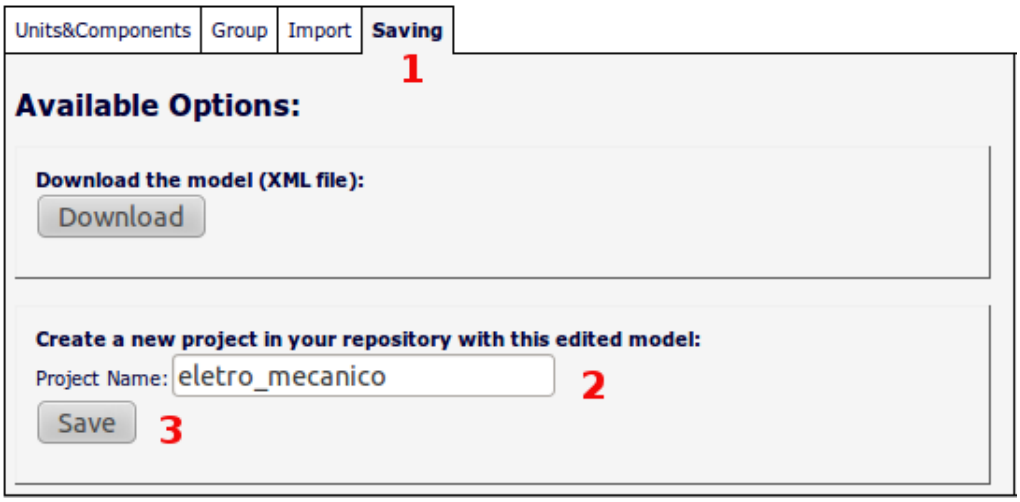

Figura 5.8: Salvando o novo modelo no repositório

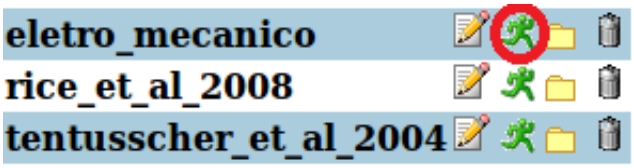

Figura 5.9: Simulação do modelo

escolher o nome do projeto no repositório (2) e salvá-lo (3).

Para simular o modelo eletro-mecânico é preciso clicar no ícone indicado na Figura 5.9 que está disponível para todos os modelos armazenados no repositório do AGOS.

A partir de então será aberta uma janela para configuração dos parâmetros e valores iniciais das equações (1), assim como a escolha do método numérico, tolerância, passo de tempo, tempo de simulação, entre outros (2). Na Figura 5.10 estão ilustrados os componentes da janela que configura uma simulação através do AGOS.

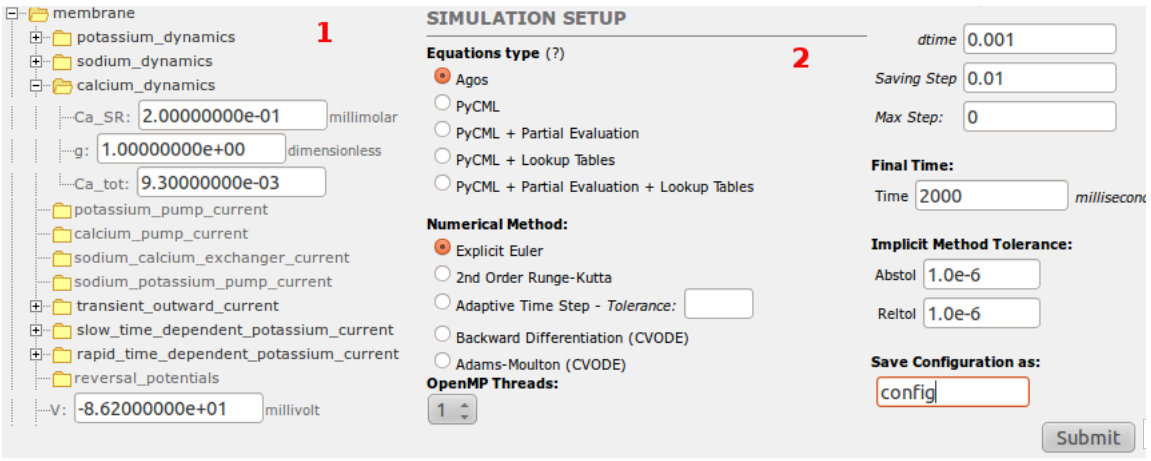

Figura 5.10: Simulação do modelo

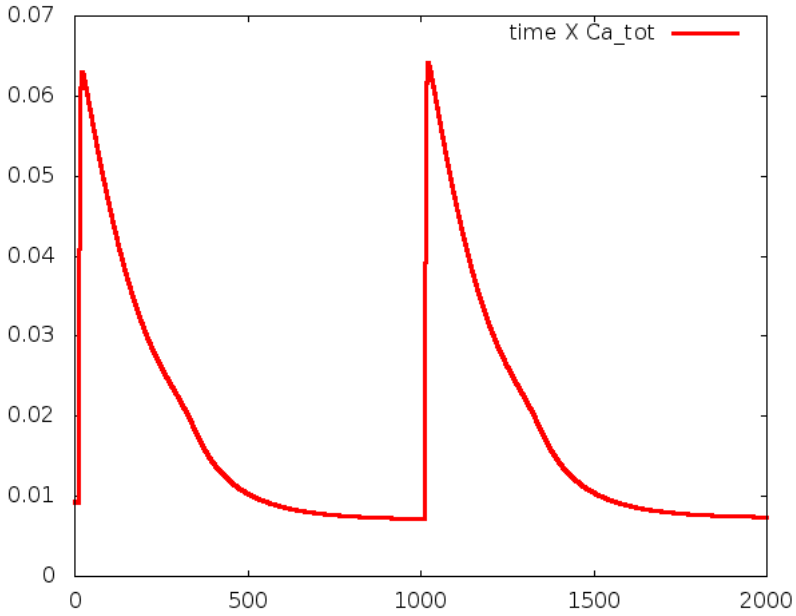

Figura 5.11: Corrente de cálcio

Enfim, serão exibidos na tela os resultados da simulação. Por exemplo, a Figura 5.11 exibe o comportamento da corrente de cálcio.

Analisando o processo de criação de um novo modelo gerado a partir da união de outros modelos previamente existentes percebe-se que o Editor é uma ferramenta promissora. Claramente, a alteração de equações através de comandos MathML seria mais trabalhosa e suscetível a erros. Entretanto, algumas modificações podem ser feitas para melhorar a usabilidade e exploração de recursos oferecidos pelo CellML. Tais sugestões estão descritas na Seção 6.5. O escopo deste trabalho é o desenvolvimento do Editor e não serão aprofundadas discussões a respeito do novo modelo gerado. Maiores detalhes sobre este assunto podem ser encontrados em [28, 4].

5.2 Métodos numéricos

Serão apresentados os resultados dos métodos numéricos propostos anteriormente, comparando a relação custo-benefício entre eles, considerando o tempo de processamento, a memória consumida e o erro numérico. Cada método foi executado cinco vezes e o tempo médio de execução será apresentado em segundos. O desvio-padrão, em todos os casos, foi menor que 1% do tempo médio de execução.

Para avaliar o erro produzido pelos métodos foi considerado que a solução ideal é

aquela produzida pelo Método de Runge-Kutta de seguda ordem (RK2), com passo de tempo fixo. Foi considerada apenas a variável V dos sistemas de equações que representa o potencial de ação. Ela foi escolhida pois é a variável mais importante dos modelos e desperta maior interesse nos pesquisadores no campo da eletrofisiologia. A comparação entre um método e a solução foi feita através do erro relativo RRMS (*relative root-mean-square error*), que é dado por:

$$erro^*_{total} = 100\frac{\sqrt{\sum_{i=1}^{n}(V_i^{RK2} - V_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(V_i^{RK2})^2}} \tag{5.10}$$

Onde V_i^{RK2} é o potencial de ação obtido pelo método de RK2, V_i^* é o potencial de ação obtido pelo método a ser avaliado no instante de tempo i e n é o número total de iterações salvas. Para o cálculo deste erro os resultados foram armazenados em arquivo a cada 1 ms da simulação. Como alguns métodos alteram o passo de tempo, pode ocorrer que o tempo da simulação não coincida com o tempo de armazenar os resultados. Para se armazenar os resultados corretamente, foi feita uma avaliação do lado direito das equações a mais, no momento em que é necessário salvar. Para tal, o passo de tempo é alterado temporariamente, apenas para que o tempo seja incrementado de maneira que o resultado seja computado exatamente no momento em que deveria ser salvo.

Para cada modelo foram testados diferentes passos de tempo e diferentes tolerâncias. Serão reportados os maiores passos de tempo e tolerâncias que produziram menos de 1% de erro, calculado pela Equação 5.10.

As execuções foram realizadas em uma máquina Intel(R) Core(TM)2 Quad, 2.33GHz de frequência, com 4Gb de memória RAM. O sistema operacional é o Linux Ubuntu Server 2.6.35-22-server e o compilador utilizado é o gcc 4.4.5. Todos os resultados apresentados a seguir foram obtidos através de executáveis compilados com a opção de otimização -O3. Em média, os resultados com a *flag* -O3 foram 30% mais rápidos do que os resultados sem otimização.

Cada um dos quatro modelos apresentados no Capítulo 2 foram simulados com todos os métodos apresentados, com as seguintes configurações: o modelo de Noble et al.[17] (NBL) foi estimulado a uma frequência de 1Hz, simulando a atividade celular durante 100 segundos. O estímulo foi realizado durante $1,5 \times 10^{-3}$ s. O passo de tempo fixo utilizado pelos métodos de Euler e de RK2 foi de $1,0 \times 10^{-5}$ s.

Tabela 5.1: Máximo passo de tempo permitido

Modelo	MEAH e MEAF	BDF
NBL (s)	$1,5 \times 10^{-3}$	$4,0 \times 10^{-4}$
TTP (s)	$1,5 \times 10^{-3}$	$5,0 \times 10^{-5}$
GRN (s)	-	-
BDK (s)	$1,5 \times 10^{-3}$	$5,0 \times 10^{-5}$

O modelo de ten Tusscher & Panfilov[15] (TTP) foi estimulado com uma frequência de 2Hz, com o estímulo durando $1,5 \times 10^{-3}$ s em uma simulação de 100s da atividade celular. Os métodos de Euler e de RK2 foram executados com passo de tempo fixo de $1,0 \times 10^{-6}$ s.

O modelo de Garny et al.[16] (GRN) foi simulado por 100s. Este modelo não permite ser estimulado e portanto, não há frequência ou duração do estímulo. O passo de tempo fixo utilizado para executar os métodos de Euler e RK2 é de $5,0 \times 10^{-6}$ s.

O modelo de Bondarenko et al.[14] (BDK) foi estimulado a uma frequência de 14Hz, simulando a atividade celular durante 10 segundos. O estímulo foi realizado durante $1,5 \times 10^{-3}$ s. Os métodos de Euler e RK2 foram executados com passo de tempo fixo de $2,0 \times 10^{-7}$ s.

Os modelos de Bondarenko et al.[14], Tentusscher & Panfilov[15] e Noble et al.[17] devem ser estimulados por um instante de tempo e para que esse estímulo não seja perdido quando houver um passo de tempo muito grande, foi atribuído ao passo de tempo máximo o valor da duração do estímulo. Para os modelos que não precisam ser estimulados, não há passo de tempo máximo. Para o método BDF, que também possui passo de tempo adaptativo, não foi possível utilizar o mesmo passo de tempo máximo que foi utilizado para os métodos de Euler Adaptativo, que foi 1,5 ms para todos os modelos. Isso ocorreu porque o BDF produzia erros maiores que o critério estabelecido, que determina que erro seja sempre menor que 1%. Diminuir a tolerância não foi suficiente para alcançar este critério, assim a solução encontrada foi diminuir o h máximo permitido. Os passos de tempo máximos permitidos, para o método BDF e para os métodos de Euler Adaptivo, pela heurística (MEAH) e pela fórmula (MEAF), são apresentados na Tabela 5.1. Mesmo assim, o erro produzido pelo BDF foi um pouco maior em três dos quatro modelos testados, o que será mostrado a seguir. Este trabalho utilizou o método BDF disponível através da biblioteca Sundials[29]. Foi utilizado o BDF de ordem cinco.

Para que o tempo de execução fosse computado corretamente, os resultados não foram

Tabela 5.2: Desempenho - Método de Euler com h fixo e BDF

Modelo	NBL		TTP		GRN		BDK	
Método	Euler	BDF	Euler	BDF	Euler	BDF	Euler	BDF
TME (s)	28,0	6,2	317,0	40,8	47,4	2,0	165,4	11,5
MEM (kB)	180	316	180	312	152	288	196	356
ER (%)	0,07	0,9	0,02	0,8	0,9	0,2	0,003	0,6
h_{max} **(s)**	-	4,0e-4	-	5,0e-5	-	5,2e-3	-	5,0e-5
h_{min} **(s)**	-	4,2e-7	-	2,7e-8	-	6,1e-5	-	5,0e-9
h_{med} **(s)**	1,0e-5	3,9e-4	1,0e-6	4,9e-5	5,0e-6	1,6e-3	2,0e-7	4,8e-5
it_{total}	1,0e7	1,3e6	1,0e8	2,0e6	2,0e7	1,2e5	5,0e7	2,3e5
it_{desc}	-	2,5e3	-	5,4e3	-	7,8e3	-	3,7e3
Equações algébricas	60		70		75		72	
Equações diferenciais	22		19		15		41	

salvos em arquivo, a fim de que o tempo de acesso a disco não fosse contado. Para o cálculo de erro, onde é necessário armazenar o resultado em arquivo, foi feita uma nova execução onde a contagem de tempo foi desconsiderada.

Inicialmente, foram executados os métodos que o AGOS já possuía, que são o método de Euler com passo de tempo fixo e o BDF. Os resultados podem ser vistos na Tabela 5.2. Os dados apresentados são tempo médio de execução, memória e erro, doravante denominados TME, MEM e ER. Também são apresentados os passos de tempo máximo, mínimo e médio (h_{max}, h_{min} e h_{med}), além do número total de iterações que o método computou (it_{total}), o número de iterações que o método teve que descartar porque o erro foi maior que o permitido (it_{desc}), assim como a quantidade de equações diferenciais e algébricas.

O método BDF executou mais rápido do que o método de Euler com h fixo em todos os modelos. O pior caso ocorreu com o modelo NBL, em que o BDF foi 5 vezes mais rápido. Por outro lado, o melhor caso aconteceu com o modelo GRN, que foi 25 vezes mais rápido.

Em média, o consumo de memória do BDF foi 80% maior, pois este método precisa montar as matrizes necessárias para a resolução do método de Newton. O erro do método de Euler foi muito menor na execução de três modelos (NBL, TTP e BDK) porque este método exige passos de tempo muito pequenos por questões de estabilidade, o que consequentemente causará menos erro. O número de iterações do BDF foi de 8 a 217 vezes menor, respectivamente, com os modelos NBL e BDK. A redução do número de iterações não foi proporcional ao tempo obtido, já que o melhor desempenho foi obtido com o mo-

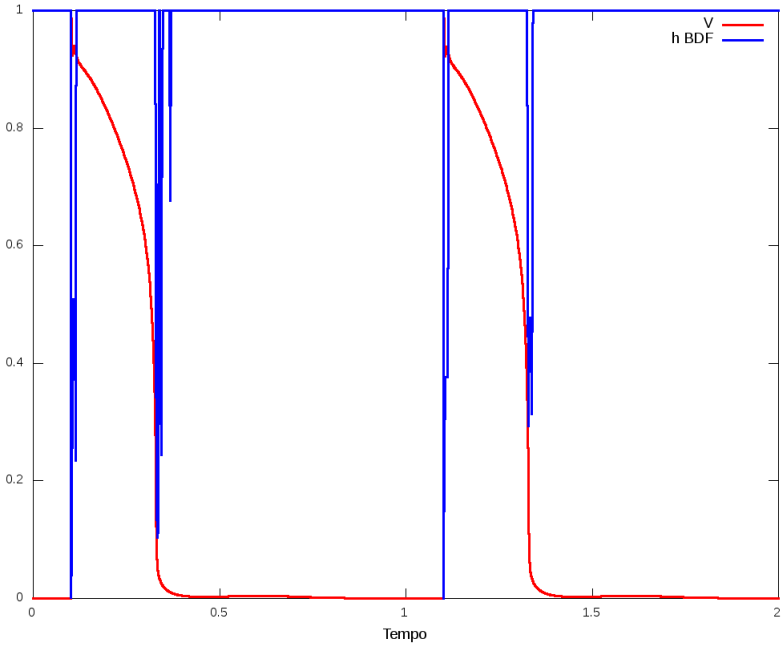

Figura 5.12: Passo de tempo BDF e potencial de ação - Noble et al.

delo GRN, que reduziu suas iterações em 167 vezes. Isto aconteceu pois o método de Newton executado pelo BDF foi mais custoso para o modelo BDK, que possui 41 EDOs, contra apenas 15 EDOs do GRN. As Figuras 5.12, 5.13, 5.14 e 5.15 ilustram a evolução do passo de tempo do método BDF e do potencial de ação (V) de cada modelo. As figuras mostram que o passo de tempo oscilou pouco, ficando em quase toda a simulação com o módulo máximo permitido. O passo de tempo apenas foi reduzido no início e no término do potencial de ação, onde as variáveis mudam bruscamente seus valores. A única exceção ocorreu com o modelo GRN (Figura 5.14). Este modelo possui dinâmica mais lenta que os demais e não possui passo de tempo máximo, o que influenciou o comportamento do passo de tempo.

5.2.1 Métodos Adaptativos

Os desempenhos obtidos pelos métodos MEAH e MEAF são apresentados nas Tabelas 5.3 e 5.4, de onde é possível concluir que os métodos MEAF e MEAH diminuíram significamente o tempo de execução, comparados ao método de Euler com h fixo, sendo de 5 a 32 vezes mais rápidos. Estes métodos consumiram a mesma quantidade de memória e os erros cresceram. O tempo de execução dos métodos MEAF e MEAH foram semelhantes aos do BDF, porém uma maior quantidade de memória foi necessária para o método implícito.

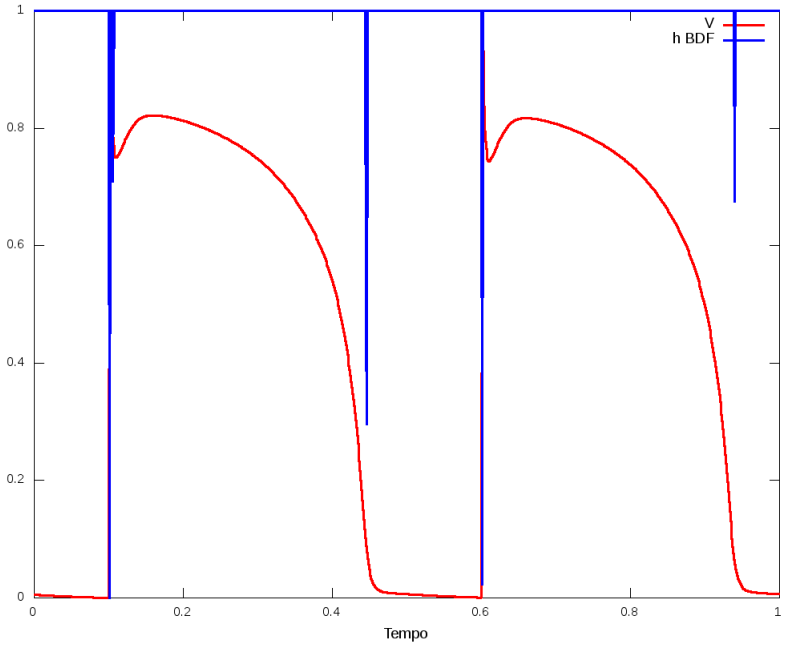

Figura 5.13: Passo de tempo BDF e potencial de ação - Ten Tusscher e Panfilov

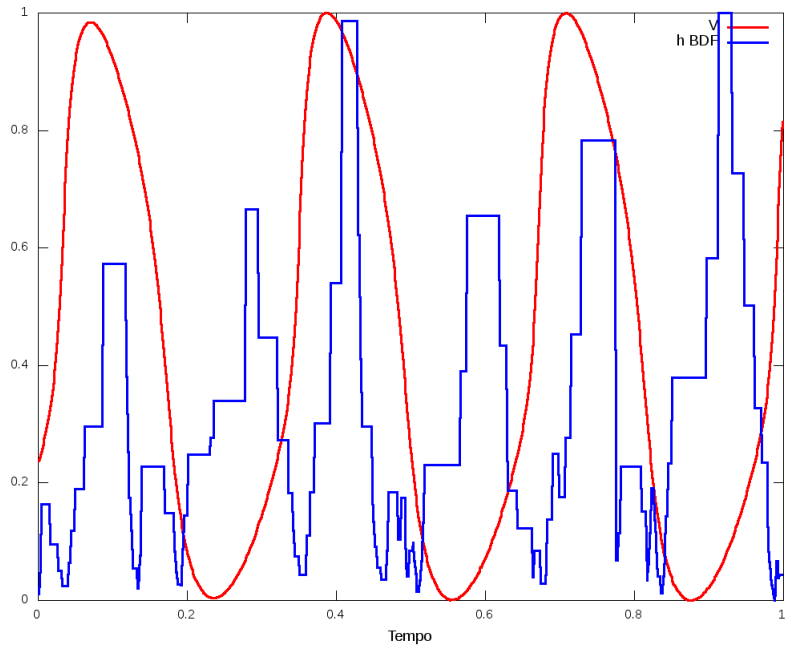

Figura 5.14: Passo de tempo BDF e potencial de ação - Garny el al.

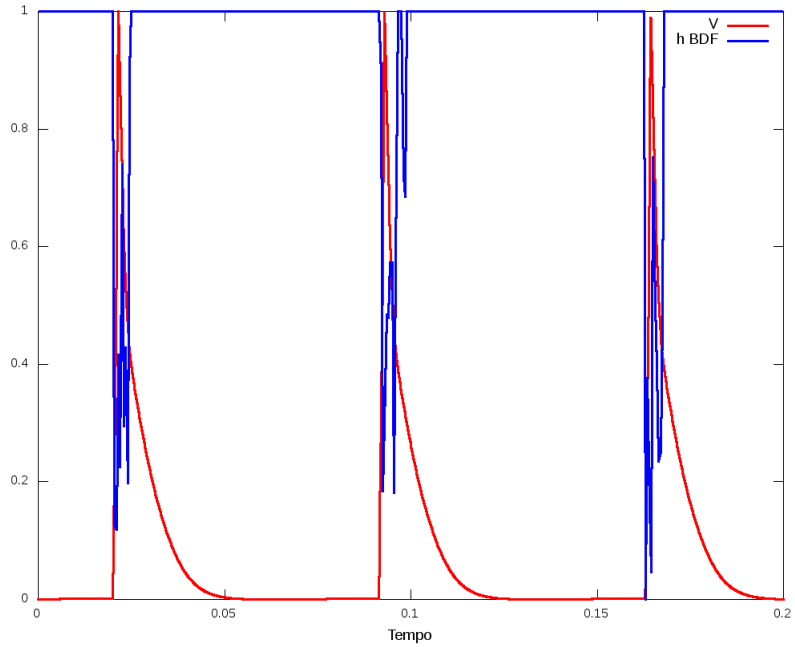

Figura 5.15: Passo de tempo BDF e potencial de ação - Bondarenko et. al

Tabela 5.3: Desempenho - Métodos Adaptativos - Noble et al. e ten Tusscher e Panfilov

Modelo	NBL		TTP	
Método	MEAH	MEAF	MEAH	MEAF
TME(s)	5,0	5,6	48,7	50,4
MEM(kB)	180	180	180	180
ER(%)	0,17	0,15	0,5	0,4
h_{max} **(s)**	1,1e-3	7,0e-4	1,0e-3	8,6e-4
h_{min} **(s)**	8,3e-7	1,0e-6	8,3e-7	1.0e-6
h_{med} **(s)**	7,5e-5	6,7e-5	8,3e-6	7,5e-6
it_{total}	1,4e6	1,5e6	1,3e7	1,3e7
it_{desc}	6,1e4	3,4e4	5,1e5	5,3e3
Tolerância relativa	1,0e-3		1,0e-3	
Tolerância absoluta	1,0e-3		1,0e-3	

Tabela 5.4: Desempenho - Métodos Adaptativos - Garny et al e Bondarenko et al.

Modelo	GRN		BDK	
Método	MEAH	MEAF	MEAH	MEAF
TME(s)	1,5	1,5	10,0	10,6
MEM(kB)	152	152	196	196
ER(%)	0,9	0,8	0,6	0,6
h_{max} **(s)**	3,9e-3	9,7e-4	1,5e-3	1,5e-3
h_{min} **(s)**	5,0e-6	5,0e-6	1,3e-7	2,0e-7
h_{med} **(s)**	1,9e-4	1,7e-4	5,9e-6	5,3e-6
it_{total}	5,4e5	5,6e5	1,7e6	1,9e6
it_{desc}	2,8e4	1,4e3	8,3e5	2,5e3
Tolerância relativa	1,0e-4		1,0e-2	
Tolerância absoluta	1,0e-4		1,0e-2	

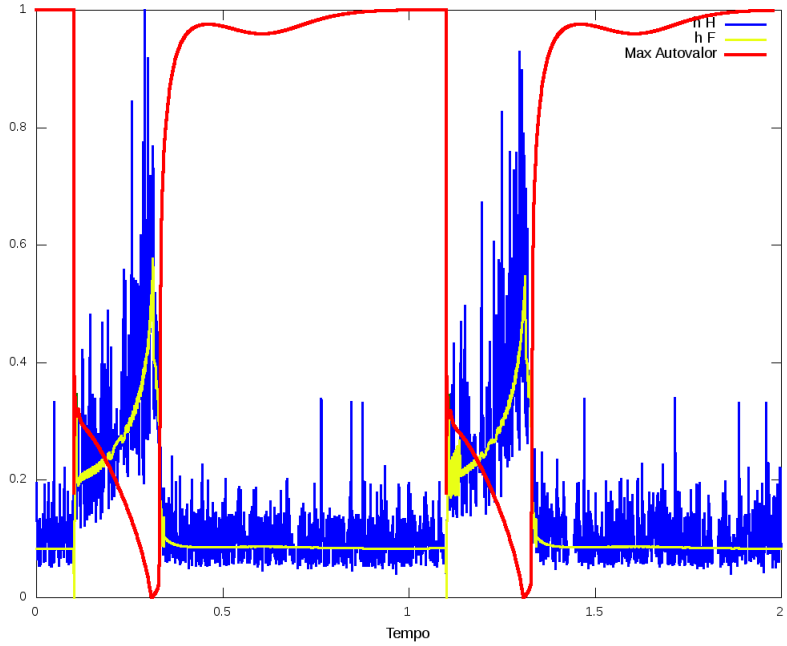

Figura 5.16: Passo de tempo adaptativo e maior autovalor do jacobiano - Noble et al.

Tabela 5.5: Desempenho - MEAH e BDF

Modelo	NBL		TTP		GRN		BDK	
Método	MEAH	BDF	MEAH	BDF	MEAH	BDF	MEAH	BDF
TME(s)	5,0	6,2	48,7	40,8	1,5	2,0	10,0	11,5
MEM(kB)	180	316	180	312	152	288	196	356
ER(%)	0,17	0,9	0,5	0,8	0,9	0,2	0,6	0,6
h_{max} **(s)**	1,1e-3	4,0e-4	1,0e-3	5,0e-5	3,9e-3	5,2e-3	1,5e-3	5,0e-5
h_{min} **(s)**	8,3e-7	4,2e-7	8,3e-7	2,7e-8	5,0e-6	6,1e-5	1,3e-7	5,0e-9
h_{med} **(s)**	7,5e-5	3,9e-4	8,3e-6	4,9e-5	1,9e-4	1,6e-3	5,9e-6	4,8e-5
it_{total}	1,4e6	1,3e6	1,3e7	2,0e6	5,4e5	1,2e5	1,7e6	2,3e5
it_{desc}	6,1e4	2,5e3	5,1e5	5,4e3	2,8e4	7,8e3	8,3e5	3,7e3

A Figuras 5.16, 5.17, 5.18 e 5.19 ilustram a evolução do passo de tempo dos métodos adaptativos MEAH e MEAF e do maior valor dos módulos das partes reais dos autovalores do jacobiano dos sistemas de equações. As derivadas parciais necessárias para compor a matriz jacobiano foram calculadas pelo AGOS através de diferenças finitas. O cálculo dos autovalores foi feito pelo *software* MATLAB[58].

Ambos os métodos adaptativos repetiram o mesmo comportamento para todos o modelos simulados neste trabalho.

Os métodos que apresentaram melhores desempenhos foram o BDF e o MEAH, e a partir de então apenas estes serão considerados para avaliação das técnicas apresentadas a seguir. Os desempenhos destes métodos podem ser comparados através da Tabela 5.5.

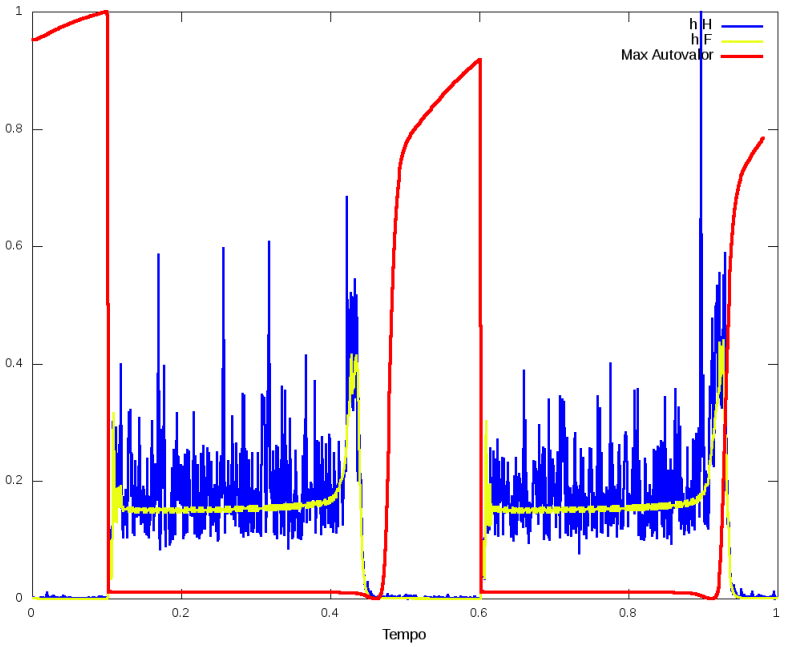

Figura 5.17: Passo de tempo adaptativo e maior autovalor do jacobiano - Ten Tusscher e Panfilov

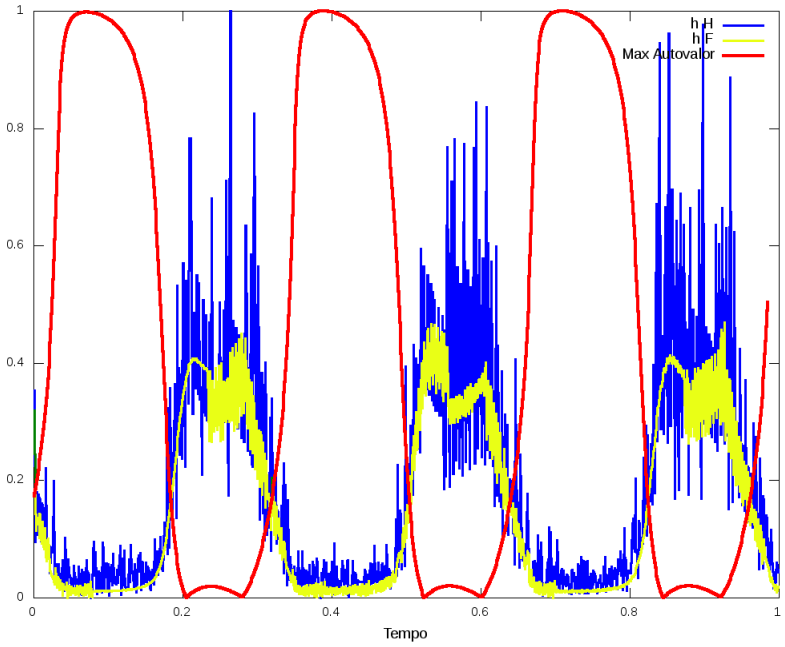

Figura 5.18: Passo de tempo adaptativo e maior autovalor do jacobiano - Garny el al.

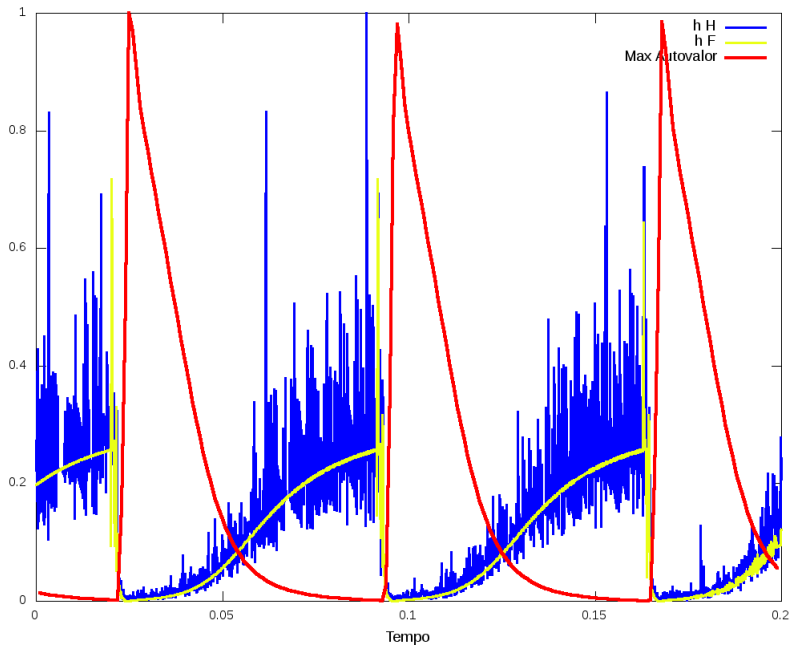

Figura 5.19: Passo de tempo adaptativo e maior autovalor do jacobiano - Bondarenko et. al

Tabela 5.6: Desempenho - Técnica computacional de Avaliação Parcial Fina PyCML

Modelo	NBL		TTP		GRN		BDK	
Método	MEAH	BDF	MEAH	BDF	MEAH	BDF	MEAH	BDF
TME(s)	5,0	6,1	44,3	40,0	1,5	1,9	9,8	11,6
MEM(kB)	180	316	180	312	152	288	196	356
ER(%)	0,17	0,9	0,5	0,9	0,9	0,17	0,6	0,6
Aceleração	1,0	1,0	1,1	1,0	1,0	1,0	1,0	1,0

5.2.2 Técnicas Computacionais

Nesta seção serão apresentados os resultados das técnicas computacionais de Avaliação Parcial, *Lookup Tables* e de ambas combinadas. As Tabelas 5.6, 5.7, 5.8 mostram o tempo médio de execução, memória gasta, erro produzido e aceleração, que é a razão entre o método sem técnica alguma e o método com a técnica em questão. Nas tabelas que contém *Lookup Tables* foi relatado o número de tabelas geradas, que determinará o gasto de memória e a melhora no desempenho.

O método de Avaliação Parcial do PyCML não foi capaz de melhorar o tempo de execução das simulações, como visto na Tabela 5.6. Este método produz a mesma saída que o método original. Por conseguinte, os erros são os mesmos do que aqueles produzidos sem esta técnica. O consumo de memória também foi o mesmo.

Os tempos de execução da técnica de *Lookup Tables* estão na Tabela 5.7. A técnica

Tabela 5.7: Desempenho - Técnica computacional de *Lookup Table*

Modelo	NBL		TTP		GRN		BDK	
Método	MEAH	BDF	MEAH	BDF	MEAH	BDF	MEAH	BDF
TME(s)	4,3	5,9	24,3	36,2	0,7	1,6	7,4	11,0
MEM(kB)	1.928	2.068	4.060	4.196	6.276	6.420	4.208	4.360
ER(%)	0,2	0,9	0,5	0,9	0,9	0,2	0,6	0,6
Aceleração	1,2	1,1	2,0	1,1	2,3	1,2	1,4	1,1
Nº de tabelas	13		30		48		31	

Tabela 5.8: Desempenho - Técnica computacional de *Lookup Tables*+Avalição Parcial

Modelo	NBL		TTP		GRN		BDK	
Método	MEAH	BDF	MEAH	BDF	MEAH	BDF	MEAH	BDF
TME(s)	3,3	5,5	19,2	35,0	0,4	1,4	6,8	10,8
MEM(kB)	4.188	4.320	3.808	3.944	4.032	4.176	4.716	4.864
ER(%)	0,2	0,9	0,5	0,9	0,9	0,2	0,7	0,6
Aceleração	1,6	1,1	2,5	1,2	3,7	1,3	1,5	1,1
Nº de tabelas	31		28		30		35	

acelerou a execução do método BDF em até 1,2 vezes e a do método MEAH, em até 2,3 vezes. No entanto, as tabelas geradas por esta técnica aumentam muito o consumo de memória, que foi até 41 vezes maior para o MEAH e 22 vezes para o BDF. Ambos os casos ocorreram com o modelo GRN, em que a técnica de LUT foi capaz de gerar mais tabelas, o que resulta em uma maior necessidade de memória, mas em contrapartida significa que mais expressões deixaram de ser computadas e foram substituídas por acessos às LUT, o que melhora o desempenho. O erro obtido com a técnica foi insignificatemente maior.

O desempenho da junção das técnicas de avaliação parcial PyCML e LUT pode ser visto na Tabela 5.8. Pode-se perceber que as técnicas reunidas resultaram em melhores desempenhos que as técnicas separadas. Ocorreram algumas variações na memória, pois em alguns modelos o número de tabelas foi reduzido e em outros, aumentado. As razões para isto serão explicadas no capítulo seguinte. Os erros praticamente permaneceram inalterados em comparação com a técnica de LUT.

5.2.3 Programação Paralela

A seguir serão apresentados os desempenhos obtidos com OpenMP para os métodos de Euler com h fixo e Euler Adaptativo pela Heurística, respectivamente nas Tabelas 5.9 e 5.10. Ambas as tabelas apresentam os tempos médios de execução, memória e o fator de aceleração (*speedup*) com 2, 3 e 4 *threads*, comparados com os respectivos métodos

Tabela 5.9: Desempenho - AGOS com OpenMP - Método de Euler com h fixo

Threads	**Modelo**	NBL	TTP	GRN	BDK
2	**TME (s)**	23,0	205,5	37,0	110,5
	Memória (kB)	220	224	220	244
	Speedup	1,2	1,5	1,3	1,5
3	**TME (s)**	23,0	155,5	24,3	85,5
	Memória (kB)	232	240	232	252
	Speedup	1,2	2,0	2,0	1,9
4	**TME (s)**	22,4	153,0	25,0	82,7
	Memória (kB)	244	248	244	264
	Speedup	1,3	2,0	1,9	2,0

Tabela 5.10: Desempenho - AGOS com OpenMP - Método de Euler Adaptativo com Heurística

Threads	**Modelo**	NBL	TTP	GRN	BDK
2	**TME (s)**	4,2	38,0	1,4	6,4
	Memória (kB)	240	236	224	260
	Speedup	1,2	1,3	1,0	1,6
3	**TME (s)**	4,3	32,0	1,1	5,4
	Memória (kB)	256	252	236	272
	Speedup	1,2	1,5	1,3	1,9
4	**TME (s)**	4,7	32,0	1,2	5,6
	Memória (kB)	268	264	248	284
	Speedup	1,0	1,5	1,2	1,8

executados sequencialmente, cujos desempenhos foram relatados nas seções anteriores deste capítulo. Os erros não foram relatados pois os métodos paralelos apresentaram os mesmos resultados que o método sequencial.

Com duas *threads*, o método de Euler com h fixo obteve desempenho de 20 a 50% mais rápido. Com 3, o desempenho com o modelo de Noble el al. permaneceu o mesmo, equanto os demais modelos executaram duas vezes mais rápido, aproximadamente. Com quatro processadores, os desempenhos praticamente permaneceram os mesmos que foram obtidos com 3 *threads*. No pior caso, o consumo de memória aumenta até 60% com 4 *threads*. O aumento ocorre com o acréscimo de processadores, pois as variáveis privadas devem ser replicadas para cada *thread*.

Os *speedups* obtidos com o método Adaptativo foram menores do que aqueles obtidos com o método de Euler. No melhor caso, o desempenho foi de até 1,9 vezes mais rápido. O pior caso obteve no máximo 20% de aceleração. Existem vários fatores que atrapalham o desempenho de programas paralelos. No Capítulo 6 serão apresentados alguns experimentos que explicam os motivos desta diferença de desempenho.

6 DISCUSSÃO

Neste capítulo serão discutidos importantes aspectos que influenciaram os desempenhos das técnicas apresentadas neste trabalho.

6.1 Métodos Adaptativos

Comparando-se os métodos MEAF e MEAH com o método de Euler com h fixo conclui-se que os métodos adaptativos diminuíram significamente o tempo de execução, sendo de 5 a 32 vezes mais rápidos e todos eles consumiram a mesma quantidade de memória. Os percentuais de erro cresceram, já que o método aumenta o passo de tempo. O número de iterações foi de 7 a 37 vezes menor, respectivamente para os modelos NBL e GRN. Ao contrário do método BDF, os métodos adaptativos obtiveram desempenhos proporcionais ao número de iterações, pois estes métodos não realizam computações significativas além do lado direito.

Percebe-se que o método MEAH obteve desempenho ligeiramente mais rápido do que o MEAF, pois o mesmo obteve o passo de tempo médio um pouco maior.

Analisando a evolução do passo de tempo h e do maior módulo do autovalor do jacobiano, percebe-se que quando o autovalor aumenta, h diminui. Por outro lado, se o autovalor diminui, h aumenta. Isso ocorre porque o método deve satisfazer a equação $\|1 + J * h\| < 1$, demonstrada na Seção 3.1.7, para que o erro não seja propagado. Para isto, o método aumenta ou diminui o valor de h. Portanto, se o sistema de equações possui regiões maiores com maiores autovalores, h será menor, fazendo que mais iterações sejam necessárias. Consequentemente, o método será mais lento. Inversamente, se a solução possui autovalores menores, h será maior e o método possuirá melhor desempenho.

O método BDF é a escolha padrão para sistemas *stiff* e é amplamente utilizado em modelos da eletrofisiologia cardíaca. Comparando-se o BDF com os métodos adaptativos propostos neste trabalho (Tabelas 5.2, 5.3 e 5.4), percebe-se que os adaptativos foram um pouco mais rápidos que o BDF em três modelos, NBL, GRN e BDK. Em média, o MEAH executou com 1 s a menos. A única exceção ocorreu com o modelo TTP, em que o BDF foi 19% mais rápido.

O método BDF adaptou o passo de tempo com menos oscilações que os adaptativos (Figuras 5.12, 5.13, 5.14 e 5.15), o que não significou que este método foi mais rápido ou que resultou em menores percentuais de erro.

Uma vantagem dos métodos adaptativos sobre o BDF é a memória gasta. O BDF precisa montar o jacobiano e resolver um sistema linear, o que torna uma iteração mais custosa em termos de memória. Todavia, os computadores atuais possuem memória muito além do necessário para as simulações apresentadas neste trabalho, onde os métodos adaptativos consumiram de 152 a 196 kB, enquanto o BDF consumiu de 288 a 356 kB, ou seja, entre 53% e 80% a mais do que os adaptativos. Não obstante, em simulações de tecido, onde é preciso executar milhares de modelos celulares simultaneamente, a questão de memória pode ser importante para o desempenho do programa e até mesmo impedir o uso de técnicas implícitas como o BDF.

Os erros numéricos produzidos pelos métodos foram todos menores que 1%. O método BDF obteve erros maiores que os adaptativos para três modelos, sendo que o modelo GRN foi a exceção. A porcentagem de erro para este modelo foi mais alta devido à ausência de estímulo. Quando há estímulo, há também o passo de tempo máximo permitido, que impede que o método alcance passos de tempo muito grandes, o que auxilia no controle do erro.

6.2 *Lookup Tables* e Avalição Parcial Fina

A técnica de Avaliação Parcial Fina do PyCML não produziu melhorias no desempenho. Isso ocorreu porque o AGOS já possui uma espécie de avaliação parcial simplificada. Por exemplo, na Seção 3.3.2.1 foram apresentadas duas maneiras de melhorar um código com avaliação parcial. A primeira, que o AGOS e o PyCML são capazes de fazer, ocorre quando uma variável associada a uma equação é requisitada várias vezes por outras equações, no mesmo passo de tempo. Isso acontece principalmente quando o valor de uma equação é utilizada em diferentes componentes do CellML. Inicialmente, a expressão seria recomputada todas as vezes que fosse acessada, em cada componente, produzindo o mesmo resultado desnecessariamente. Então a otimização para este caso consiste em computar a expressão apenas uma vez e armazená-la em uma variável, que será acessada toda vez que requisitada. A outra maneira de melhorar o desempenho é mais complexa e ape-

nas o PyCML é capaz de realizar. Consiste em avaliar se subexpressões das equações resultam em valores constantes. Por exemplo, algumas equações possuem o cálculo $\frac{RT}{F}$. Entretanto, R, T e F são constantes, o que significa que o resultado desta conta também será constante. Ou seja, não será necessário recomputá-la no próximo passo, será apenas necessário gravá-la em uma variável e acessá-la sempre que necessário.

A técnica de LUT foi até 1,2 vezes mais rápida com o BDF e até 2,3 vezes com o MEAH, mas o consumo de memória chega a ser 41 vezes maior para o MEAH e 22 vezes para o BDF. Ambos os casos ocorreram com o modelo GRN, em que a técnica de LUT foi capaz de gerar mais tabelas, o que resulta em mais necessidade de memória. Mas significa que mais equações deixaram de ser computadas e foram substituídas por acessos às LUT, o que melhora o desempenho. O erro obtido com a técnica foi insignificantemente maior.

A discrepância entre o desempenho dos métodos BDF e MEAH ocorre porque as técnicas de avaliação parcial e LUT apenas melhoram o desempenho do lado direito das equações. Como a tarefa mais dispendiosa do método adaptativo é computar o lado direito, as melhorias ficam mais evidentes do que no caso do método BDF, pois este, além do lado direito, precisa computar o jacobiano do sistema de equações e resolver o método de Newton, o que não será melhorado pelas técnicas apresentadas aqui. Por isto, o ganho das técnicas no caso do BDF é menor do que no adaptativo. Um aspecto negativo de LUT é o consumo de memória, no pior caso, aumenta de 152 kB para aproximadamente 6 MB. Portanto, deve-se considerar a relação custo-benefício entre desempenho e memória. Por exemplo, simular apenas uma célula com LUT, consome até 6 MB, o que não é relevante para as máquinas atuais. Entretanto, para simulações de tecido, é necessário utilizar centenas ou até milhares de células, o que exige técnicas que consomem menos memória.

As técnicas de avaliação parcial e LUT combinadas resultaram nos melhores desempenhos. Com o método BDF, o desempenho foi até 1,35 vezes mais rápido. Com o método adaptativo, a simulação foi até 3,7 vezes mais rápida. Ao se combinar as duas técnicas, alguns modelos aumentaram ou diminuíram o número de tabelas geradas, o que determina o consumo de memória. Isso ocorre porque a avaliação parcial modifica algumas equações, o que influencia no número de tabelas. Um exemplo que pode ser encontrado em [2] é a equação a seguir, que inicialmente é dada por:

$$I = \frac{1}{1 + e^{v(F/RT)}} \tag{6.1}$$

Onde I é uma corrente elétrica, R é a constante dos gases, T é a temperatura absoluta e F é a constante de Faraday. Desta maneira, não é possível gerar uma LUT para esta equação, visto que a mesma não depende apenas de v. Entretanto, ao se utilizar a técnica de avaliação parcial, a expressão $\frac{F}{RT}$, formada por constantes, é substituída pelo valor numérico do resultado, para que este não seja computado desnecessariamente. Desta forma, a equação passa a depender apenas de v, possibilitando que uma nova LUT seja gerada.

Por outro lado, LUT podem deixar de existir. Isso occoreu em equações condicionais, que possuem a seguinte característica:

$$g(V,x) = \begin{cases} f(V) & \text{se} \quad x > 3 \\ f2(V) & \text{se} \quad x \leq 3 \end{cases} \tag{6.2}$$

Onde $g(V,x)$ é uma função qualquer, $f(V)$ e $f2(V)$ são funções que apenas dependem de V e x é uma variável. Caso seja aplicada a técnica de LUT à função $g(V,x)$, serão geradas duas tabelas de substituição, uma para $f(V)$ e outra para $f2(V)$. Entretanto, se x for constante, a técnica de avaliação parcial eliminará uma equação. Por exemplo, se $x = 4$ durante toda a simulação, a equação $f2(V)$ pode deixar de existir e então a equação $g(V,x) = f(V)$. Ao se combinar as técnicas, a LUT de $f2(V)$ não será mais gerada, pois a avaliação parcial a eliminará sendo apenas gerada a LUT de $f(V)$. Portanto, haverá menos consumo de memória. Isto ocorreu principalmente com o modelo de Garny el al. [16].

6.3 *OpenMP*

Os métodos de Euler e MEAH foram adaptados para ambientes multiprocessados com OpenMP, onde o melhor desempenho de ambos ocorreu quando o trabalho foi dividido para três *threads*. Neste caso, os métodos foram duas vezes mais rápidos que os sequenciais e consumiram até 50% a mais de memória. Como algumas variáveis são privadas, a quantidade de memória utilizada aumentará com o acréscimo de processadores, pois estas variáveis são declaradas novamente em cada *thread*. O método adaptativo consumiu mais memória, pois são necessários mais vetores privados para armazenar os valores do lado direito das iterações atual e anterior, entre outras variáveis extras necessárias.

Foi verificado que o OpenMP se comportou de maneira diferente para cada modelo, não houve melhoria do desempenho dos métodos ao se aumentar de 3 para 4 *threads* e o MEAH apresentou menores *speedups* que o método de Euler. A seguir serão discutidos os experimentos que explicam os resultados desta implementação.

Um dos fatores que pode diminuir a velocidade de execução de aplicações com OpenMP é o custo de criação e manutenção de *threads*. Percebe-se que tarefas muito pequenas que precisam de pouco tempo de computação, não são adequadas para se aplicar técnicas de computação paralela, pois o custo extra que existe ao se utilizar *threads* sobrepõem-se ao benefícios de utilizar múltiplos processadores. Para demonstrar esta situação foi implementado um programa semelhante ao proposto na Seção 4.2.2, que consiste em um laço para variar o tempo de simulação e dentro deste existe um trabalho a ser feito. Para que fosse possível realizar testes com diferentes custos, o lado direito foi substituído pelo cálculo de um somatório, onde o total de números a ser somado é determinado pelo usuário. Este somatório pode ser paralelizado, dividindo-se igualmente o trabalho entre as *threads*, a fim de evitar interferências no desempenho por causa do desbalanceamento de carga. Foi colocada uma barreira após o somatório, para que todas as tarefas sejam iniciadas ao mesmo tempo na próxima iteração. O desempenho deste programa foi avaliado com diversos tamanhos de somatório, divididos entre um ou mais processadores, e o resultado deste teste pode ser visualizado na Figura 6.1. Foram avaliados os desempenhos com 1.200, 12.000 e 120.000 operações de soma. Estas operações foram divididas igualmente entre 2, 3 e 4 *threads*. É possível concluir que as menores acelerações foram obtidas quando há menos trabalho, resultando em desempenhos de 1,45 a 2,08 vezes mais rápidos. Com 12.000 operações, o programa foi capaz de acelerar a execução de 1,94 a 3,60 vezes, desempenho superior ao obtido com 1.200 operações. Com 120.000 operações, a aceleração foi praticamente linear, indicando que o método proposto por este trabalho possui melhor desempenho quando há mais operações.

Nos modelos avaliados anteriormente, existem muitas tarefas com poucas operações, principalmente quando há mais *threads*. Consequentemente, pode-se afirmar que o desempenho do OpenMP é prejudicado nestes casos, pois o custo das tarefas não é suficiente para que o *overhead* seja superado, fazendo com que o desempenho seja reduzido.

Outro problema a ser considerado é a má divisão do trabalho, ou desbalanceamento de carga. Isto ocorre porque as tarefas indivisíveis possuem tamanhos muito discrepantes, o

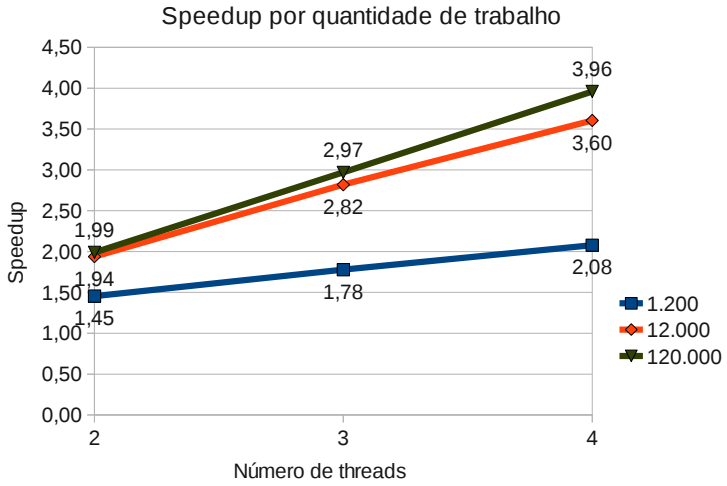

Figura 6.1: *Speedup* por quantidade de trabalho

que faz com que as tarefas maiores limitem o desempenho. Por exemplo, o melhor desempenho obtido com OpenMP aconteceu com o modelo de Bondarenko et al.[14], que possui 16 tarefas, ou seja, 16 conjuntos de equações, que não dependem uns dos outros, mas possuem dependências internas. O maior conjunto consome 32,9% do tempo de processamento real do modelo, enquanto os demais consomem de 0,2 a 20,6%. Neste contexto, o processamento real significa o tempo em que a aplicação de fato esteve sendo computada no processador. Ou seja, o tempo gasto com espera em barreira, para sincronização, foi desconsiderado. As Figuras 6.2 e 6.3 exibem a porcentagem de trabalho que cada thread recebeu para os modelos de Bondarenko et al.[14] e Noble et al.[17]. Ao se analisar a Figura 6.2, percebe-se que com duas e três *threads*, o algoritmo guloso apresentado na Seção 4.2.2 conseguiu agrupar as tarefas de maneira que a distribuição delas entre os processadores fosse equilibrada. Entretanto, para quatro *threads*, o ideal seria distribuir 25% das tarefas para cada processador, o que não é possível, pois um conjunto de equações, que é indivisível, possui quase 33% do trabalho total. Então esta tarefa é enviada para um processador e as demais tarefas menores são divididas entre os outros três processadores. Claramente a divisão de tarefas ficará desbalanceada, o que varia de acordo com a dependência que existe entre as equações de cada modelo.

No modelo de Noble et al., que obteve o pior desempenho, o problema da má distribuição de trabalho fica mais evidente. Este modelo possui quinze árvores, das quais a maior delas consome quase 70% do tempo de computação. Fica claro que a implementação

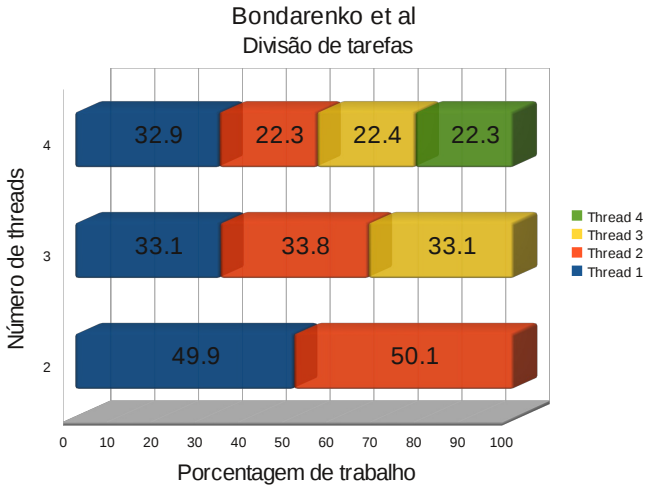

Figura 6.2: Divisão de Trabalho - Bondarenko et. al

em paralelo só é vantojosa com duas *threads*. Ao se introduzir mais processadores, não será possível diminuir o tempo de computação, pois 70% das tarefas não pode ser paralelizada e ficará em apenas um processador, enquanto o restante das tarefas será distribuído entre os outros processadores. Tal situação é ilustrada na Figura 6.3.

Uma das consequências do desbalanceamento de trabalho é o desperdício de processadores, já que as *threads* que recebem tarefas menores devem aguardar as que receberam tarefas maiores, devido à sincronização. A Figura 6.4 ilustra a porcentagem do tempo do total em que as *threads* permaneceram ociosas. Percebe-se que a divisão das tarefas na simulação no modelo de Noble et al. causou maior ociosidade, onde o método de Euler com h fixo permaneceu de 28 a 43% do tempo total de computação em espera, enquanto o método de Euler com h adaptativo com heurística oscilou entre 28 e 44%. O modelo de Bondarenko et al. permaneceu menos tempo ocioso, onde o método de Euler aguardou entre 16 e 30%, e o adaptativo, 20 e 33%. A partir destes experimentos, pode-se constatar que o modelo que ficou menos ocioso obteve melhor desempenho, o que ratifica a importância de uma divisão de trabalho equilibrada entre as *threads*. Porém, os experimentos realizados neste trabalho possuem tarefas com tamanhos discrepantes. Estas tarefas não podem ser subdivididas por questões de dependência, o que prejudica o desempenho em ambientes de computação paralela. Pode-se afirmar que em alguns modelos só há algum ganho com 2 *threads*, enquanto outros modelos, com tarefas que possuem tamanhos mais próximos, podem obter ganhos com mais *threads*. Outra característica a

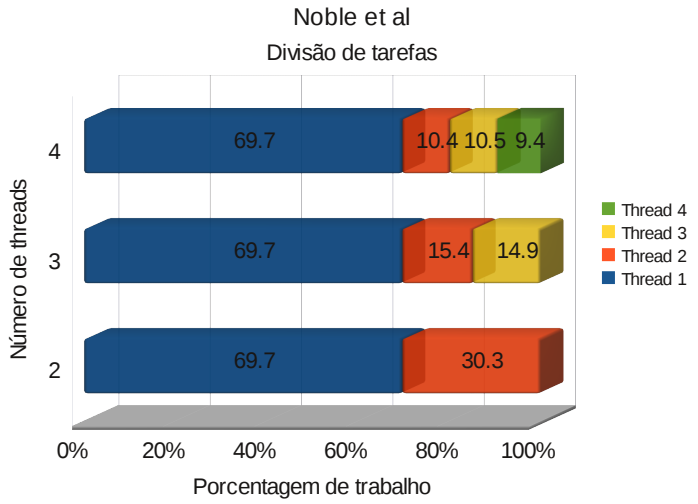

Figura 6.3: Divisão de Trabalho - Noble et. al

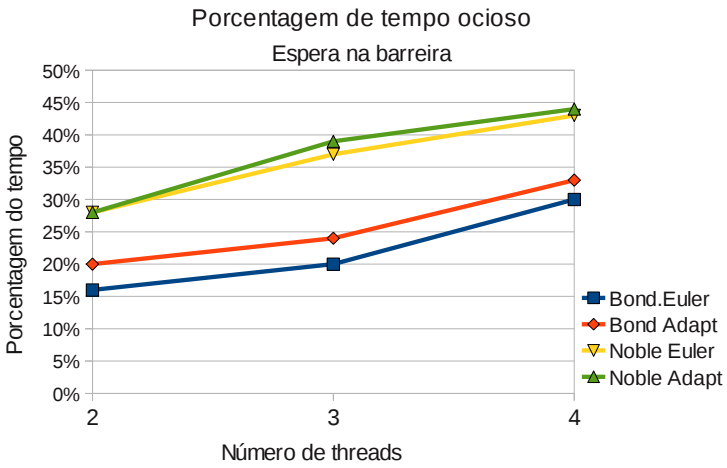

Figura 6.4: Porcentagem de tempo ocioso

ser notada é que o método adaptativo possui maior tempo ocioso. Isto ocorre porque a cada iteração do tempo este método calcula o lado direito. Para que se calcule o novo passo de tempo, é necessário que todas as *threads* tenham terminado de computar o lado direito, o que torna uma barreira necessária. Após encontrar o novo passo de tempo, é necessária outra sincronização para que todas as *threads* iniciem a próxima iteração com os valores corretos de h. Em contrapartida, o método de Euler com h fixo só necessita de uma sincronização por iteração, após o cálculo do lado direito. Logo, a maior espera que ocorre devido à barreira extra causa um pior desempenho, que pode ser notado nas Tabelas 5.9 e 5.10.

6.4 AGOS e PyCML

Ao se avaliar os métodos apresentados anteriormente, percebe-se que os melhores desempenhos em termos de tempo de execução foram alcançados pelo Método de Euler com h Adaptativo pela Heurística (MEAH). Comparando as técnicas computacionais aplicadas a este método aquelas que produziram maiores reduções no tempo de execução foram o OpenMP com três *threads* e a combinação de avaliação parcial com *Lookup Tables* do PyCML

As técnicas do PyCML foram mais velozes do que o AGOS com OpenMP na execução de três modelos, NBL, TTP e GRN. A exceção foi o modelo BDK. Isto ocorreu porque o BDK possui mais equações. Entre equações diferenciais e algébricas, ele possui 113, contra 82 do NBL, 89 do TTP e 90 do GRN. Como foi demonstrado na Seção 6.3, se a quantidade de trabalho for pequena, pode não ser vantajoso utilizar *threads* por causa do custo extra que há para mantê-las. Além disto, este modelo teve as suas tarefas agrupadas de maneira mais equilibrada, o que resultou em tempos menores de espera nas barreiras. Estes fatores sugerem que o algoritmo de resolução de EDOs com OpenMP proposto neste trabalho é promissor para sistemas que possuam números maiores de tarefas, desde que haja pouca dependência entre as equações.

Um aspecto que favorece o OpenMP é o consumo de memória. Considerando o algoritmo com três processadores, foi necessário até 1,6 vezes a mais de memória do que o respectivo método sequencial. Por outro lado, o técnica AP + LUT precisou de até 27 vezes a mais de memória.

6.5 Trabalhos futuros

Como trabalhos futuros, são sugeridas melhorias na interface do Editor, para aumentar a sua usabilidade. Por exemplo, atualmente os componentes estão sendo organizados em uma árvore separada. Uma melhoria que pode ser feita é permitir que os componentes sejam organizados hierarquicamente na interface principal, onde são exibidas as unidades, variáveis e equações. Outra melhoria é implementar a análise semântica do compilador, que realiza verificações entre as operações do programa de entrada, verificando se os tipos dos dados são compatíveis. Pode-se utilizar esta técnica para conferir se as equações escritas no Editor possuem operações com unidades compatíveis.

A respeito das técnicas computacionais, é possível unir o OpenMP a *Lookup Tables*, porém isto exige que o grafo de adjacências seja implementado no código do PyCML. Os resultados não sugerem que o OpenMP seja executado com avaliação parcial. A técnica de avaliação parcial reduz o número de equações, substituindo uma equação que seria computada várias vezes por uma equação que é computada apenas uma vez. Esta característica aumenta a dependência entre as equações, o que atrapalha o balanceamento de carga e consequentemente, o desempenho.

Outra melhoria é modificar a maneira que se computa o tempo dos grupos de equações, para que o método guloso os divida entre os processadores. Atualmente cada grupo é executado 100 vezes, e o resultado é descartado. O método numérico poderia ser executado sequencialmente até que se obtenha os tempos necessários para que o método guloso seja executado e após isto o método começaria a execução em paralelo. Isto diminuiria o custo extra para se dividir os grupos, pois os resultados não seriam descartados.

7 CONCLUSÃO

Uma ferramenta *Web* foi implementada para manipular modelos descritos em CellML, o que envolveu a criação da interface e a implementação de um compilador para transformar *strings* com equações em MathML. Demonstrou-se um exemplo de utilização desta ferramenta, onde foi criado um novo modelo através da combinação de outros dois modelos previamente existentes. O novo modelo simula a influência da atividade elétrica do coração sobre a atividade mecânica.

Foram implementados métodos numéricos com passo de tempo adaptativo no AGOS, que gera código C++ automaticamente a partir de modelos descritos em CellML. Foram feitas comparações dos métodos numéricos, considerando o tempo de execução, consumo de memória e percentual de erro numérico. Ao se comparar os métodos propostos com aqueles que inicialmente existiam no AGOS, o método adaptativo pela heurística (MEAH) foi executado em até 32 vezes mais rápido que o método de Euler com passo de tempo fixo. O outro método que já existia no AGOS é o *Backward Differentiation Formulas* (BDF), escolha padrão para sistemas *stiff* relacionado a eletrofisiologia cardíaca. O tempo de execução do MEAH foi até 1,3 vezes mais rápido do que o BDF, além do BDF ser implícito e precisar resolver operações adicionais que consomem até 80% a mais de memória do que os métodos explícitos.

Adicionalmente, foram implementadas técnicas avançadas para resolução de equações diferenciais, chamadas de Avaliação Parcial e *Lookup Tables* (LUT), ambas fornecidas pela ferramenta PyCML. Também foram utilizadas técnicas de computação paralela, implementadas via OpenMP.

Comparando o desempenho do MEAH com MEAH + OpenMP percebe-se que o OpenMP foi capaz de executar até 2 vezes mais rápido. Por outro lado, o MEAH com as técnicas de LUT e avaliação parcial unidas foi até 3,7 vezes mais rápido do que o MEAH sem técnica alguma. No entanto, a técnica LUT podem consumir até 17 vezes mais memória.

Ao se comparar o método BDF com o MEAH com as técnicas, observa-se que o MEAH com OpenMP foi até 2,1 vezes mais ágil que o BDF. Já o MEAH com LUT e avaliação parcial foi até 5 vezes mais rápido.

Como trabalho futuro são sugeridas melhorias na usabilidade do Editor para que seja mais simples organizar hierarquicamente os componentes em sua interface. Também se faz necessário melhorar a maneira de dividir as tarefas para que estas sejam executadas em paralelo, além de unir a técnica de LUT com OpenMP.

Para o desenvolvimento deste trabalho foram estudados vários assuntos de diferentes áreas, como eletrofisiologia cardíaca, métodos numéricos para resolução de equações diferenciais e alguns campos da ciência da computação, como compiladores e programação paralela. A diversidade de áreas do conhecimento em único trabalho é uma característica deste programa multidisciplinar de pós-graduação. Apesar das ferramentas e técnicas propostas terem sido apenas testadas com modelos de células cardíacas, elas podem ser utilizadas para descrever e resolver equações de modelos computacionais que representem fênomenos de diferentes naturezas, desde que descritos através de equações diferenciais ordinárias.

REFERÊNCIAS

[1] BRASIL, "MINISTÉRIO DA SAÚDE - Uma análise da mortalidade no Brasil e regiões", http://portal.saude.gov.br/SAUDE/visualizar_texto.cfm?idtxt=24421, Agosto 2011.

[2] CAMPOS, F. O., *Modelagem computacional da eletrofisiologia cardíaca: O desenvolvimento de um novo modelo para células de camundongos e avaliação de novos esquemas numéricos*, Dissertação (mestrado em modelagem computacional), Universidade Federal de Juiz de Fora - Mestrado em modelagem computacional, 2008.

[3] COSTA, C. M., CAMPOS, R. S., CAMPOS, F. O., DOS SANTOS, R. W., "Web Applications Supporting the Development of Models of Chagas Disease for Left Ventricular Myocytes of Adult Rats", *Lecture Notes in Computer Science*, v. 5103, pp. 120–129, 2008.

[4] DE OLIVEIRA, B. L., *Modelagem quantitativa da eletromecânica do tecido cardíaco humano*, Dissertação (mestrado em modelagem computacional), UNIVERSIDADE FEDERAL DE JUIZ DE FORA, 2011.

[5] "The CellML Project", http://www.cellml.org/, Março 2011.

[6] "W3C Math Home", http://www.w3.org/Math/, Março 2011.

[7] "XML Tutorial", http://www.w3schools.com/xml/default.asp, Março 2011.

[8] BARBOSA, C. B., SANTOS, R. W., AMORIM, R., CIUFFO, L. N., MANFROI, F., OLIVEIRA, R. S., CAMPOS, F. O., "A transformation tool for ODE based models", *Lecture Notes in Computer Science*, v. 3991, pp. 69–75, 2006.

[9] "PyCml - CellML Tools in Python", https://chaste.comlab.ox.ac.uk/cellml/, Abril 2011.

[10] NETTER, F. H., *Atlas de anatomia humana*. Artes Médicas, 1998.

[11] HODGKIN, A., HUXLEY, A., "A quantitative description of membrane current and its application to conduction and excitation in nerve", *J Physiol*, v. 117, pp. 500–544, 1952.

[12] GUYTON, A. C., HALL, J. E., *Textbook of Medical Physiology*. 11th ed. Elsevier, 2006.

[13] KEENER, J., SNEYD, J., *Mathematical Physiology I: Cellular Physiology*. Segunda ed., v. Volume 8/I. Springer, 2009.

[14] BONDARENKO, V. E., SZIGETI, G. P., BETT, G. C. L., KIM, S. J., RASMUSSON, R. L., "A computer model of the action potential of the mouse ventricular myocytes", *American Journal of Physiology*, v. 287, pp. H1378–H1403, 2004.

[15] TEN TUSSCHER, K. H. W. J., PANFILOV, A. V., "Alternans and spiral breakup in a human ventricular tissue model", *American Journal of Physiology, Heart and Circulatory Physiology*, v. 291 3, pp. H1088–1100, 2006.

[16] GARNY, A., KOHL, P., HUNTER, P. J., BOYETT, M. R., NOBLE, D., "One-dimensional rabbit sinoatrial node models: benefits and limitations", *The Journal of Cardiovascular Electrophysiology*, v. 14, pp. S121–S132, 2003.

[17] NOBLE, D., VARGHESE, A., KOHL, P., NOBLE, P., "Improved guinea-pig ventricular cell model incorporating a diadic space, IKr and IKs, and length- and tension-dependent processes", *an J Cardiol*, v. 14, pp. 123–134, 1998.

[18] HEATH, M. T., *Scientic Computing - An Introductory Survey*. 1997.

[19] RUGGIERO, M. A. G., DA ROCHA LOPES, V. L., *Cálculo Numérico - Aspectos Teóricos e Computacionais*. Segunda ed. Makron Books, 1996.

[20] MACLACHLAN, M. C., SUNDNES, J., SPITERI, R. J., "A comparison of nonstandard solvers for odes describing cellular reactions in the heart", *Computer Methods in Biomechanics and Biomedical Engineering*, v. 10(5), pp. 317–326, 2007.

[21] HAIRER, E., WANNER, G., *Solving Ordinary Differential Equations II, Stiff and Differential Algebraic Problems*. Springer, 1991.

[22] SUNDNESS, J., LINES, G. T., NIELSEN, B. F., MARDAL, K.-A., TVEITO, A., *Computing the electrical activity in the heart*. 2006.

[23] GARNY, A., NICKERSON, D. P., COOPER, J., DOS SANTOS, R. W., MILLER, A. K., MCKEEVER, S., NIELSEN, P. M. F., HUNTER, P. J., "CellML and associated tools and techniques", *Philosophical Transactions of the Royal Society A.*, v. 366, pp. 3017–3043, 2008.

[24] CUELLAR, A., NIELSEN, P., HALSTEAD, M., BULLIVANT, D., NICKERSON, D., HEDLEY, W., NELSON, M., LLOYD, C., *CellML 1.1 Specification*, Bioengineering Institute, University of Auckland.

[25] MARTINS, D., CAMPOS, F. O., CIUFFO, L. N., OLIVEIRA, R. S., AMORIM, R. M., DA FONSECA VIEIRA, V., EBECKEN, N. F. F., DE BARROS BARBOSA, C., DOS SANTOS, R. W., "A Computational Framework for Cardiac Modeling Based on Distributed Computing and Web Applications", *Lecture Notes in Computer Science*, v. 4395, pp. 544–555, 2007.

[26] AMORIM, R. M., *Ferramenta de Transformação para Equações Diferenciais Ordinárias descritas em XML e sua aplicação em Modelagem Computacional*, Trabalho de conclusão de curso (graduação em ciência da computação), Universidade Federal de Juiz de Fora, 2007.

[27] ORENSTEIN, D., "QuickStudy: Application Programming Interface (API)", http://www.computerworld.com/action/article.do? command=viewArticleBasic&articleId=43487, Abril 2011.

[28] CAMPOS, R. S., AMORIM, R. M., COSTA, C. M., DE OLIVEIRA, B. L., DE BARROS BARBOSA, C., SUNDNES, J., DOS SANTOS, R. W., "Approaching cardiac modeling challenges to computer science with CellML-based web tools", *Future Generation Computer Systems*, 2009.

[29] "Sundials", https://computation.llnl.gov/casc/sundials/main.html, Abril 2011.

[30] COHEN, S. D., HINDMARSH, A. C., "CVODE, A Stiff/Nonstiff ODE Solver in C", *Computers in Physics*, v. 10(2), pp. 138–143, 1996.

[31] JONES, N. D., GOMARD, C. K., SESTOFT, P., *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[32] VIGMOND, E. J., HUGHES, M., PLANK, G., LEON, L. J., "Computational tools for modeling electrical activity in cardiac tissue", *Journal of Electrocardiology*, v. 36, pp. 69–74, 2003.

[33] DEXTER, F., SAIDEL, G. M., LEVY, M. N., RUDY, Y., "Mathematical model of dependence of heart rate on tissue concentration of acetylcholine", *Am J Physiol.*, v. 256, pp. H520–6, 1989.

[34] PITT-FRANCIS, J., PATHMANATHAN, P., BERNABEU, M. O., BORDAS, R., COOPER, J., FLETCHER, A. G., MIRAMS, G. R., MURRAYB, P., OSBORNE, J. M., WALTER, A., CHAPMAN, S. J., GARNY, A., VAN LEEUWEN, I. M. M., MAINIB, P. K., RODRÍGUEZ, B., WATERS, S. L., WHITELEY, J. P., BYRNE, H. M., GAVAGHAN, D. J., "Chaste: A test-driven approach to software development for biological modelling", *Computer Physics Communications*, v. 180, pp. 2452–2471, 2009.

[35] COOPER, J., MCKEEVER, S., GARNY, A., "On the application of partial evaluation to the optimisation of cardiac electrophysiological simulations", *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2006.

[36] COOPER, J. P., *Automatic validation and optimisation of biological models*, Ph.D. Thesis, Oxford University, 2009.

[37] WILKINSON, B., ALLEN, M., *Parallel Programming*. Segunda ed. Prentice Hall International, 2005.

[38] MATTSON, T. G., SANDERS, B. A., MASSINGILL, B. L., *Patterns For Parallel Programming*. Pearson Education, 2005.

[39] GALLINA, L. Z., *Avaliação de Desempenho do OpenMP em Arquiteturas Paralelas*, Trabalho de conclusão de curso (graduação em ciência da computação), UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2006.

[40] TANENBAUM, A. S., WOODHULL, A. S., *Sistemas operacionais*. Segunda ed. Bookman, 1999.

[41] "OpenMP.org", http://openmp.org/wp/, Abril 2011.

[42] BARNEY, B., "OpenMP Tutorial", https://computing.llnl.gov/tutorials/openMP/, Abril 2011.

[43] CHANDRA, R., DAGUM, L., KOHR, D., MCDONALD, D. M. J., MENON, R., *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001.

[44] "Ajax Tutorial", http://www.w3schools.com/ajax/default.asp, Julho 2011.

[45] CRANE, D., PASCARELLO, E., WITH DARREN JAMES, *Ajax in action*. Manning, 2006.

[46] ROODT, Y. O., *The effect of Ajax on performance and usability in web environments*, Master's Thesis, Universiteit van Amsterdam, 2006.

[47] AHO, A. V., SETHI, R., ULLMAN, J. D., *Compiladores: Princípios, Técnicas e Ferramentas*. 1995.

[48] APPEL, A. W., PALSBERG, J., *Modern Compiler Implementation in Java*. Second edition ed. Cambridge University Press, 2002.

[49] MINKOFF, S. E., KRIDLER, N. M., "A comparison of adaptive time stepping methods for coupled flow and deformation modeling", .

[50] CAMPOS, R. S., LOBOSCO, M., DOS SANTOS, R. W., "Adaptive Time Step for Cardiac Myocyte Models", *Procedia Computer Science*, v. 4, pp. 1092 – 1100, 2011, Proceedings of the International Conference on Computational Science, ICCS 2011.

[51] CAMPOS, R. S., DOS SANTOS, R. W., "Esquema adaptativo para a resolução numérica de modelos da eletrofisiologia cardíaca", *Anais do IV MCSUL / IX ERMAC*, 2010.

[52] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C., *Algoritmos Teoria e Prática*. Elsevier, 2002.

[53] WILF, H. S., *Algorithms and Complexity*. University of Pennsylvania, 1994.

[54] CAMPOS, F. O., CAMPOS, R. S., SANTOS, R. W., "Evaluation of Numerical Methods and Computational Techniques for Solving Cardiac Cell Models", In:

MAGJAREVIC, R., DOSSEL, O., SCHLEGEL, W. C. (eds), *World Congress on Medical Physics and Biomedical Engineering, September 7 - 12, 2009, Munich, Germany*, v. 25/4, pp. 2287–2290, *IFMBE Proceedings*, Springer Berlin Heidelberg, 2010.

[55] "Chaste - Cancer, Heart and Soft Tissue Environment", http://www.cs.ox.ac.uk/chaste/, Agosto 2011.

[56] TEN TUSSCHER, K. H. W. J., NOBLE, D., NOBLE, P. J., PANFILOV, A. V., "A model for human ventricular tissue", *Am J Physiol Heart Circ Physiol*, v. 286(4), pp. H1573–H1589, 2004.

[57] RICE, J., WANG, F., BERS, D., TOMBE, P., "Aproximate model of cooperative activation and crossbridge cycling in cardiac muscle using ordinary differential equations", *Biophysical Journal*, v. 95, pp. 2368–2390, 2008.

[58] "MATLAB - The Language Of Technical Computing", http://www.mathworks.com/products/matlab/index.html, Setembro 2011.