UNIVERSIDADE DE SÃO PAULO
Instituto de Matemática e Estatística
Departamento de Ciência da Computação

MAC0499 – Trabalho de Formatura Supervisionado

PARALELIZAÇÃO DE UM MODELO INTEGRADO DOS SISTEMAS TÉRMICO E RESPIRATÓRIO DO CORPO HUMANO

Alunos:
Fernando Fernandes Chaves
Leandro de Moraes

Orientador:
Prof. Dr. Marco Dimas Gubitoso

São Paulo
2009

Índice

I – Parte Objetiva

Capítulo 1
Introdução

A modelagem matemática de sistemas biológicos vem cada vez mais sendo utilizada no estudo dos mecanismos complexos que formam o ser humano. Essa abordagem apresenta vantagens em relação aos métodos experimentais tradicionais, pois pode ser aplicada no estudo de situações onde há riscos à segurança do indivíduo, como despressurização de cabines de avião ou hipotermia em cirurgias cardíacas.

Desde meados do século XX, vários modelos foram desenvolvidos para contemplar os sistemas respiratório e térmico do corpo humano. O modelo de Albuquerque-Neto (2009), um dos poucos a integrar os dois sistemas, vem sendo desenvolvido como tese de doutorado junto à Escola Politécnica da USP.

1.1 Motivação e Objetivo

Uma vez que a implementação original desse modelo visa atingir fidelidade na representação dos aspectos físico-químicos, mas não dá prioridade à eficiência computacional, que o deixa muito custoso para simulações complexas, julgou-se oportuno o desenvolvimento deste trabalho, cujo objetivo é reduzir o tempo da simulação numérica aplicando técnicas de paralelismo e otimização do código sequencial, permitindo que a simulação faça uso mais racional dos recursos computacionais disponíveis.

Capítulo 2
O Modelo

O objetivo do modelo é determinar a distribuição de calor, oxigênio e gás carbônico no corpo humano para diferentes situações de temperatura e pressão do ambiente.

O resultado apresenta a distribuição do calor como variação de temperatura em °C nos tecidos. A concentração dos gases é representada pela pressão parcial do oxigênio e do gás carbônico nos compartimentos onde são armazenados. Portanto, ao longo deste texto, bem como no código, a concentração dos gases é tratada muitas vezes apenas pelo termo “pressão”.

Para chegar ao objetivo, o modelo divide o corpo em segmentos, estes formados por vasos sanguíneos e pelas camadas que representam os tecidos e órgãos do corpo humano. Equações diferenciais são obtidas aplicando balanços de calor e massa aos elementos.

No caso da troca de gases nos vasos e tecidos e da troca de calor nos vasos, equações diferenciais ordinárias são obtidas e resolvidas pelo método de Euler implícito.

Na caso da troca de calor nos tecidos, são obtidas equações diferenciais parciais que são discretizadas pelo método dos volumes finitos (MVF) implícito. Para a resolução do método, a geometria de cada segmento é levada em consideração.

2.1 Geometria

No modelo, o corpo humano é dividido em 15 segmentos: cabeça, pescoço, tronco, braços, antebraços, mãos, coxas, pernas e pés. Estes são formados por camadas que representam os tecidos e órgãos: pele, gordura, músculo, osso, cérebro, pulmão, coração, e vísceras.

Os segmentos são representados por paralelepípedos (pés a mãos) ou cilindros (demais segmentos), sendo a distribuição dos tecidos uniforme em uma ou duas dimensões dos volumes. A figura 1 apresenta a geometria dos segmentos e a figura 2 mostra a distribuição dos tecidos em cada um deles.

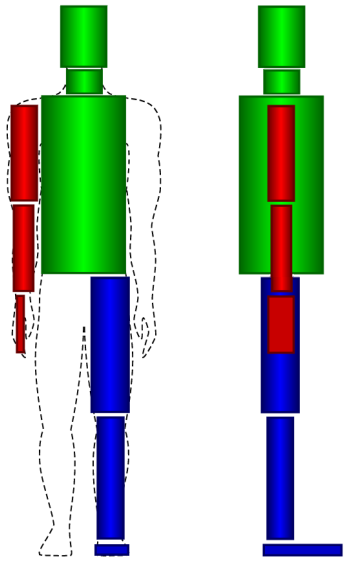

Figura 1: Representação dos segmentos no modelo.

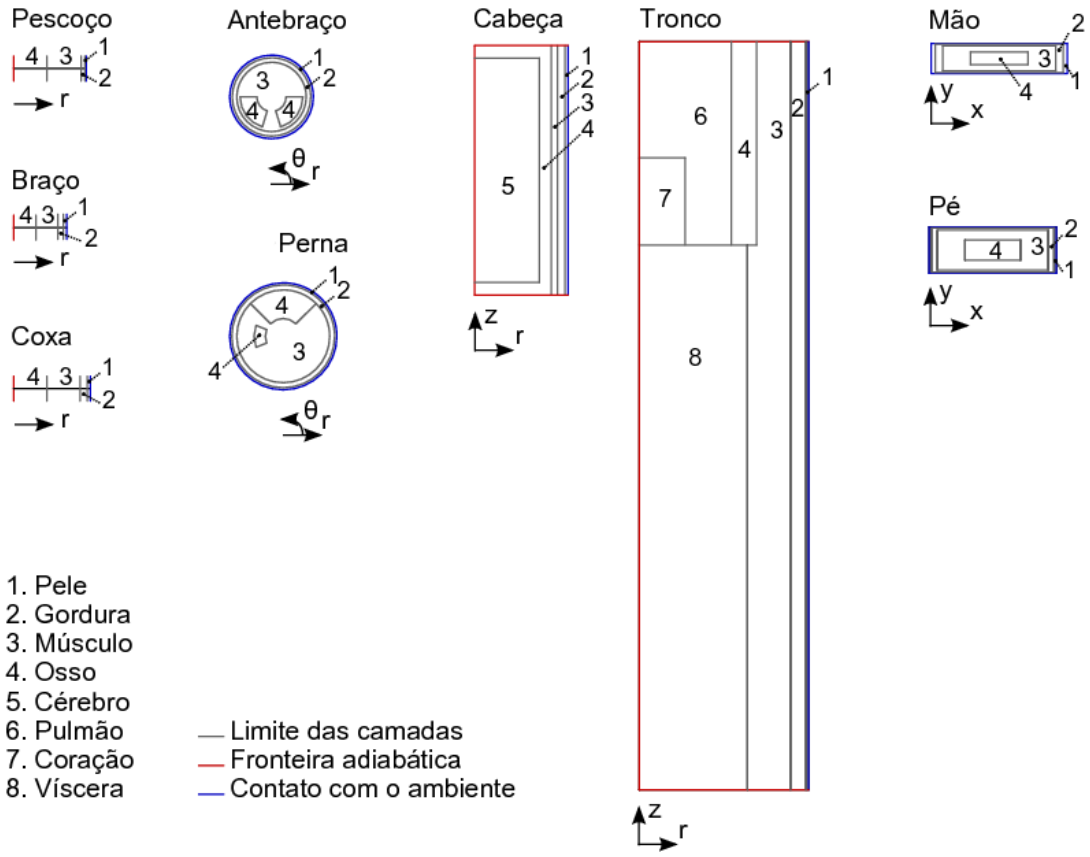

Figura 2: Distribuição das camadas nos segmentos.

2.2 Circulação

A circulação do sangue é o principal mecanismo de transporte de calor, O_2 e CO_2 no corpo. O principal método utilizado nessa modelagem é a divisão dos diversos reservatórios de sangue e gases dos segmentos em compartimentos, como mostrado na figura 3.

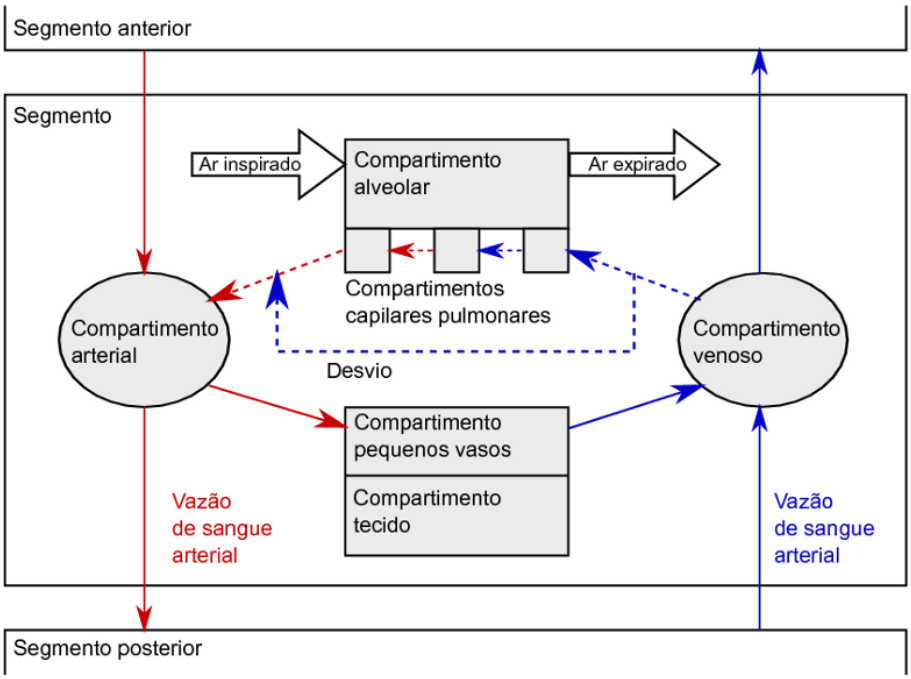

Figura 3: Circulação de sangue em um segmento.

Os grandes vasos estão representados pelos compartimentos arterial e venoso, cuja vazão de sangue conecta os segmentos. Os pequenos vasos (capilares) estão conectados aos tecidos. O sangue arterial proveniente do segmento anterior entra no compartimento arterial. Uma parte segue para os pequenos vasos, onde troca calor e gases com os tecidos, e a outra segue para o próximo segmento (com exceção dos segmentos das extremidades). Ao sair dos pequenos vasos, o sangue segue para o compartimento venoso, que também recebe sangue venoso do segmento posterior, e segue para o segmento anterior.

No caso do tronco, não existe segmento anterior. O sangue venoso segue para os capilares pulmonares, onde é realizada a troca de calor e gases com o compartimento alveolar.

2.3 Solução do Modelo

Tanto as equações diferenciais ordinárias, que variam no tempo, como as equações diferenciais parciais, que variam no tempo e espaço, são resolvidas por métodos que utilizam formulação totalmente implícita, ou seja, a cada passo do método numérico, uma relação entre valores antigos e atuais é calculada. Essa relação é então é comparada a uma precisão pré-estabelecida para determinar a ocorrência, ou não, de convergência. Isso significa que a simulação precisa de um processo iterativo a cada instante de tempo.

```
para tempo ← 0 até TempoDaSimulação                                    (I)
      para iteração ← 0 até NúmeroMaximoDeIterações                    (II)
            (…)
            paracada segmento
                  PassoCalculoTemperatura()
            (…)
            paracada segmento
                  PassoCalculoPressao()
            (…)
            se <atingiu precisão em todos os segmentos>                (III)
                  break;
```

Código 1: Fluxo de execução principal da simulação (função Corpo::Calcula()).

Como pode ser observado no código 1, a cada passo no laço do tempo (I), é executado um laço mais interno (II) que varre a lista de segmentos e dá um passo no cálculo dos valores (para aquele momento) até que haja convergência em todos os segmentos (III). Na verdade, se o contador `iteração` em algum momento atingir o parâmetro `NúmeroMaximoDeIterações`, a execução é interrompida com mensagem de erro.

Este texto irá constantemente se referir ao laço (II) como “laço das iterações” (não confundir com iterações do laço!).

2.4 Resultados

A título de ilustração, seguem alguns resultados obtidos pelo modelo.

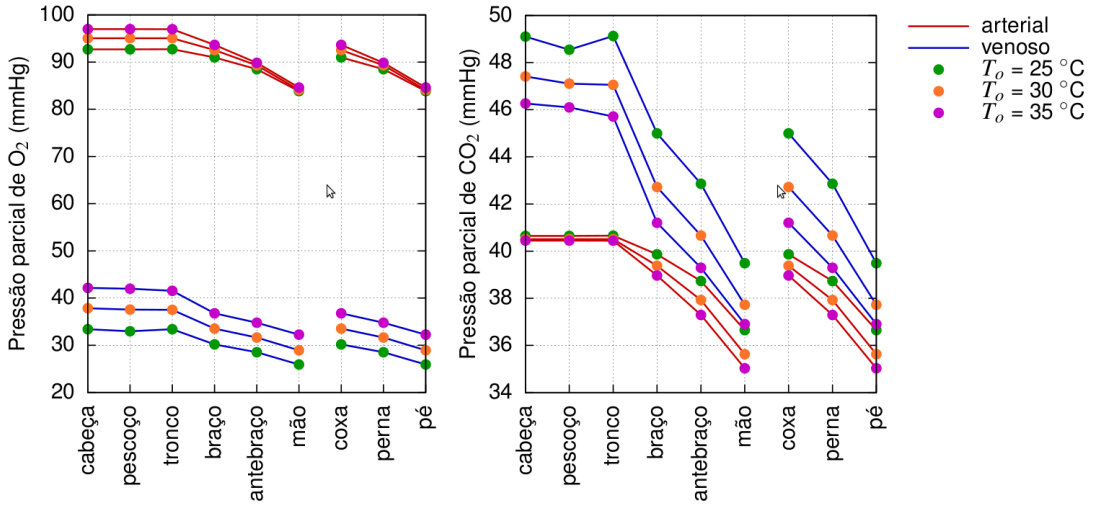

Figura 4: Concentração de O_2 e CO_2 no sangue arterial e venoso.

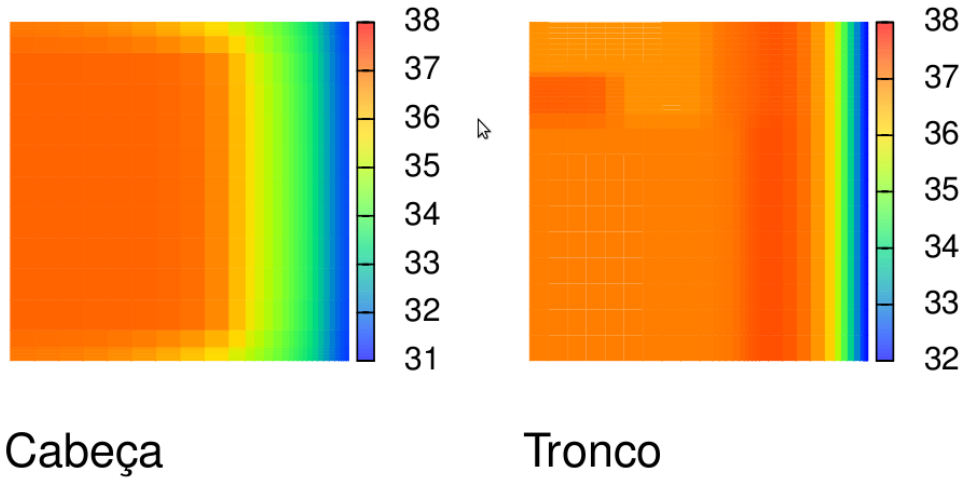

Figura 5: Distribuição de temperatura na cabeça e no tronco.

Capitulo 3
Metodologia

3.1 Biblioteca

A biblioteca Boost.Thread, versão 1.40, foi utilizada para a implementação de paralelismo na solução.

Decidiu-se utilizá-la pois se trata de uma biblioteca que recebe constantes atualizações e que tem se mostrado estável ao longo dos anos. Prova disso está no fato de que o futuro novo padrão de C++ (C++0x) utilizará partes desta biblioteca como base para suporte de funcionalidades básicas de concorrência.

Outro ponto importante vem do fato de que o padrão atual do C++ não aborda *threads*, sendo frequente o uso de bibliotecas fornecidas pelo sistema operacional para construir programas *multithreads*, o que pode resultar em um código não padronizado e não portável. Um dos objetivos dessa biblioteca é eliminar estes problemas, fornecendo uma interface flexível, cuja implementação trabalha com as plataformas POSIX, Win32 e Macintosh Carbon.

Os principais recursos fornecidos pela biblioteca são:

- Classe `thread`
  - Representa uma *thread* em execução. Recebe no construtor um ponteiro para função ou um *function object*, que será chamado quando a *thread* iniciar sua execução.

- Classe `thread_group`
  - Armazena um conjunto de *threads*. Entre outras funcionalidades, permite facilmente que se interrompa a execução até que todas as *threads* encapsuladas por ela tenham terminado suas execuções.

- Classe `mutex`
  - Fornece uma implementação de controle para acesso exclusivo à informação.

- Classe `unique_lock`
  - Através das instâncias desta classe, é efetuado o travamento das instâncias da classe `mutex`. Recebe no construtor uma instância da classe mutex, e imediatamente faz a requisição da trava, que será liberada em seu destrutor.

- Classe `barrier`
  - Implementação do conceito de barreira. Recebe no construtor o número de `threads` que devem fazer a sincronização nessa barreira. Quando esse número é atingido, libera as `threads` para prosseguirem com suas execuções.

3.2 Ambiente

Todas as análises realizadas foram feitas utilizando o seguinte ambiente:

3.2.1 *Software*

Sistema Operacional: Ubuntu 9.04
kernel: 2.6.28-16-generic
Compilador: g++ - 4.3.3-5ubuntu4

3.2.2 *Hardware*

Processador: Intel i7-920
Clock: 2.66 Ghz
Número de núcleos: 4
Cache: 8MB
Ram: 4GB
Hyper-Threading: Desabilitado

3.3 Medições

Para a análise da solução implementada e de todas as suas versões intermediárias, a metodologia descrita a seguir foi utilizada para medições de tempo tanto para a finalidade de *profiling* quanto para análise de desempenho.

Através do terminal dentro da interface gráfica, e somente com o *System Monitor* aberto, executa-se cinco vezes a simulação utilizando exatamente os mesmos parâmetros. As saídas contendo valores discrepantes (“pontos fora da curva”) são descartadas e as médias dos tempos utilizando as saídas remanescentes são calculadas. Essas médias serão então adotadas como resultado da medição.

Para a medição com o intuito de se analisar o desempenho, toda saída para tela é desabilitada. A simulação é executada com o seguinte comando:

```
$ time ./matreco
```

Para a medição com a finalidade de *profiling*, as saídas para tela são redirecionadas para um arquivo que será posteriormente analisado.

```
$ time ./matreco >> output.dat
```

Capitulo 4

Profiling

Primeiramente, foi realizado um *profiling* no código original com o objetivo de identificar as regiões com maior demanda por processamento. Numa primeira abordagem foi utilizada a ferramenta *gprof*, que identificou em alto nível o custo das principais funções.

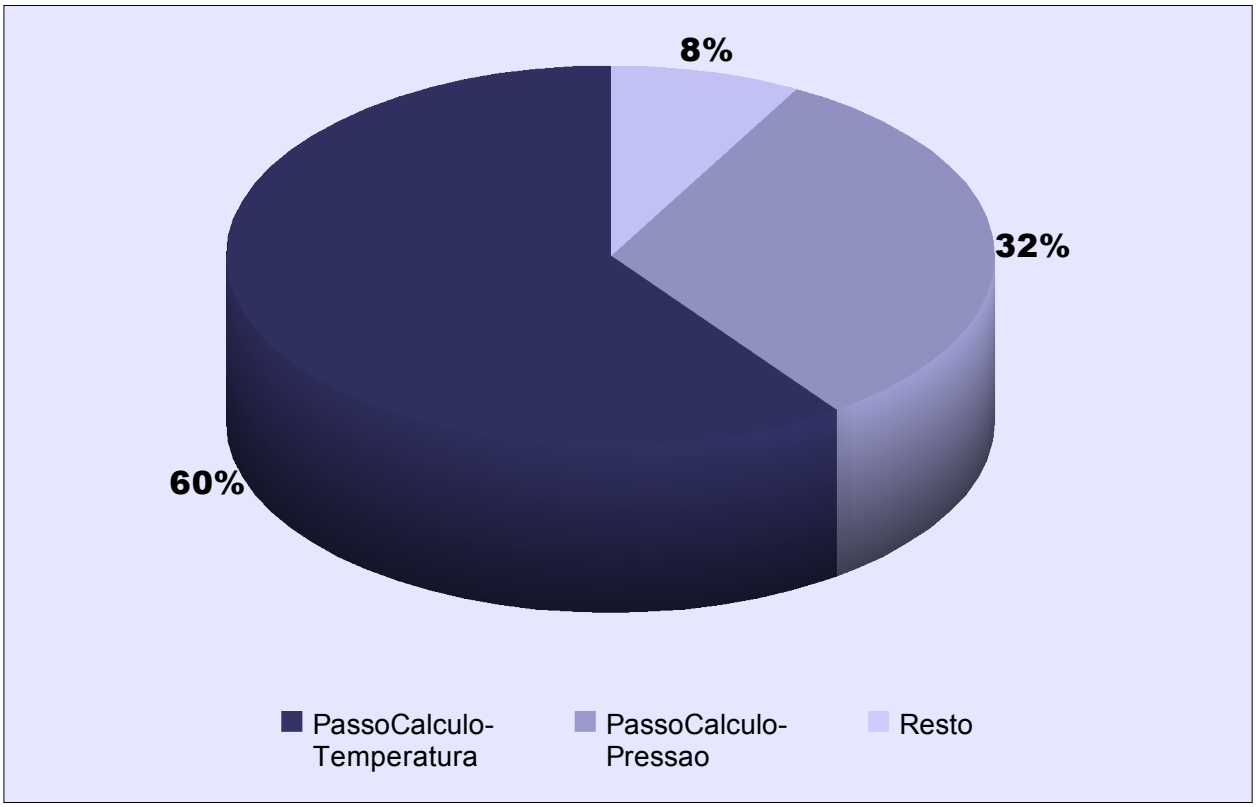

Gráfico 1: Resultado do *gprof*.

Como pode ser observado no gráfico 1, as funções `PassoCalculoTemperatura()` e `PassoCalculoPressao()` juntas consomem mais de 90% do tempo da simulação. Isso já era previsto, dado os métodos numéricos iterativos utilizados para resolver as equações diferenciais, citados no capítulo 2.

Procurou-se, portanto, um maior detalhamento no perfil de execução das funções. Para essa abordagem, as medições de tempo foram feitas diretamente no código, uma vez que o *gprof* não detalha o tempo de execução de trechos internos das funções.

Na função `PassoCalculoTemperatura()`, o cálculo da temperatura nas células dos

tecidos, que consiste na resolução de equações diferenciais parciais resolvidas pelo método dos volumes finitos (ou seja, a temperatura varia no tempo e espaço e sua discretização leva em conta a geometria dos segmentos) gasta 95% do tempo da função, contra 5% do cálculo da temperatura nos outros compartimentos (grandes e pequenos vasos), onde equações diferenciais ordinárias são resolvidas pelo método de Euler implícito.

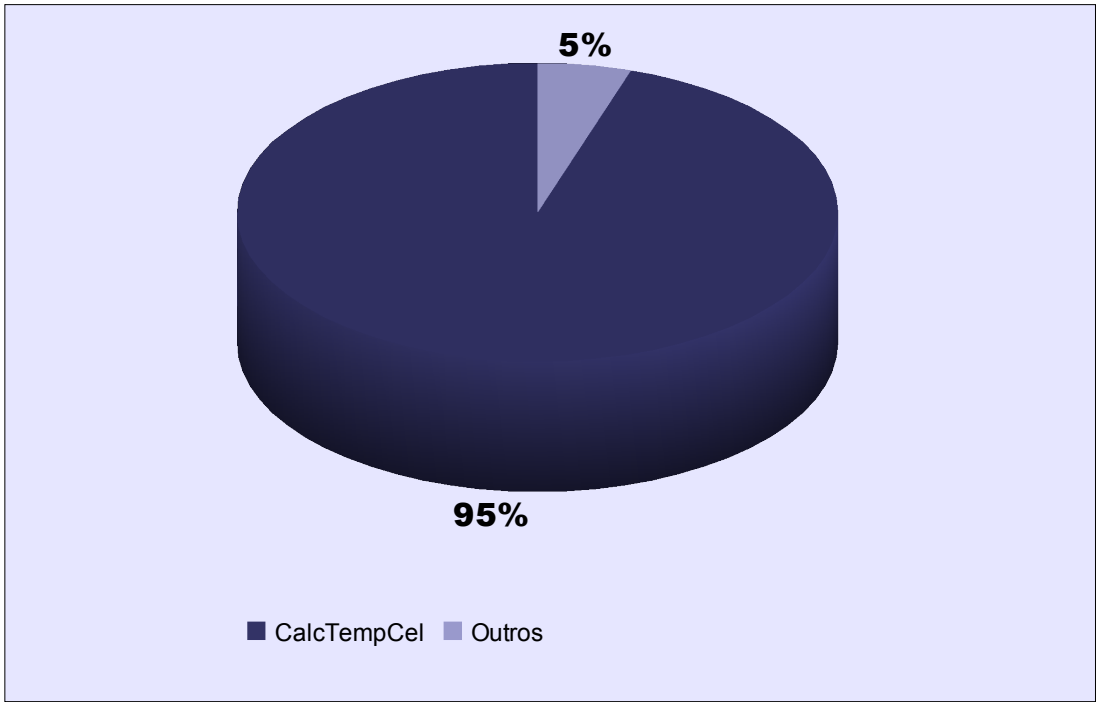

Gráfico 2: *Perfil de execução da funcão PassoCalculoTemperatura()*.

No caso da função `PassoCalculoPressao()`, somente a atualização das pressões [parciais de O_2 e CO_2] nos capilares pulmonares é responsável por consumir 25% do tempo gasto por essa função. O cálculo das pressões nos outros tecidos é responsável por consumir 45%. Nos grandes vasos, 30%. Vale ressaltar que o tecido pulmonar está presente apenas no segmento Tronco, portanto o tempo consumido para calcular as pressões nos capilares pulmonares é concentrado somente em um segmento, enquanto que os resultados apresentados para o cálculo nos outros tecidos e grandes vasos corresponde à somatória do tempo gasto para isso em todos os segmentos.

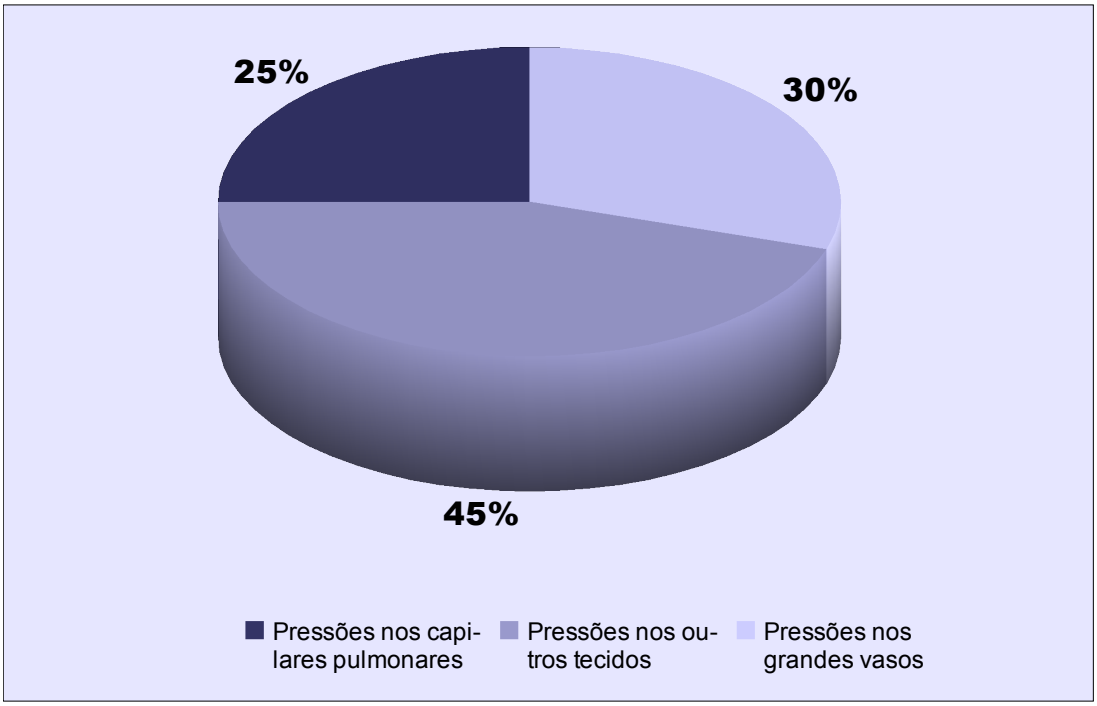

Gráfico 3: *Perfil de execução da função PassoCalculoPressao()*.

Os resultados obtidos neste capítulo indicaram os principais pontos do modelo a serem estudados para possíveis otimizações.

As soluções de paralelização e otimização do código sequencial serão descritas no capítulo a seguir.

Capítulo 5

Soluções

Para chegar ao resultado apresentado, a otimização foi dividida em partes, cada uma contemplando um ponto especifico do modelo a ser trabalhado. Este capítulo apresenta a evolução da solução final a partir de implementações intermediárias, representadas por versões.

5.1 Primeira versão

Conforme descrito no capítulo anterior, viu-se a necessidade de se aplicar alguma técnica de otimização nas funções de iteração (`PassoCalculoPressao()` e `PassoCalculoTemperatura()`).

A primeira abordagem adotada foi aplicar as alterações necessárias para que fosse possível o cálculo, em paralelo, da temperatura em cada segmento.

Como foi visto no capítulo 2, a cada iteração percorre-se a lista de segmentos chamando as funções que dão mais um passo nos cálculos de temperatura e pressão. O código então foi modificado para, a cada iteração do laço de iterações, criar uma *thread* para cada segmento, que será responsável pela execução do cálculo da temperatura.

Após a criação de todas as *threads*, a execução da simulação só deverá prosseguir quando todos os segmentos terminarem o cálculo da temperatura. Para atingir essa sincronização, foi criada uma barreira, como ilustrado no seguinte pseudo-código.

```
para tempo ← 0 até TempoDaSimulação
      para iteração ← 0 até NúmeroMáximoDeIterações
            paracada segmento
                  <cria thread para executar PassoCalculoTemperatura()>
            <barreira para sincronizar o cálculo da temperatura>
```

Código 2: Fluxo de execução da função Corpo::Calcula() com criação de threads.

A única variável compartilhada entre os segmentos durante o cálculo da temperatura que sofre alteração é o objeto que representa o compartimento pulmonar. O tronco atualiza a

temperatura desse compartimento. Contudo, nenhum outro segmento faz alteração ou leitura da variável. Logo não foi necessário fazer qualquer controle de concorrência.

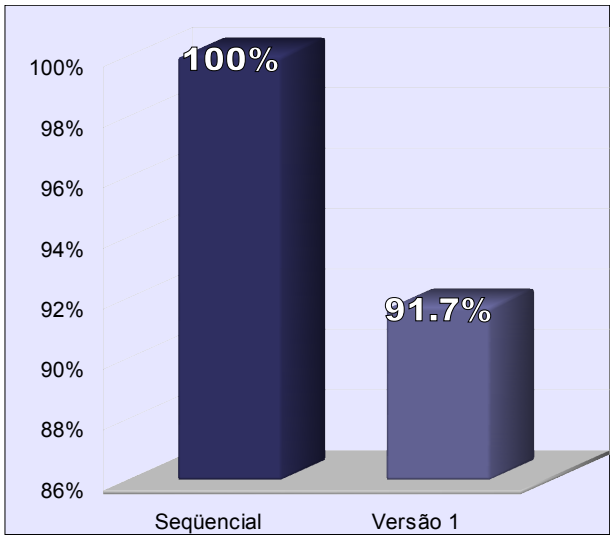

Gráfico 4: Tempo de execução em % do código sequencial.

Conforme pode ser visto no gráfico 4, a execução em paralelo das iterações do cálculo da temperatura diminuiu em 8.3% o tempo de execução, com base no código sequencial.

Contudo, essa abordagem introduz um *overhead* relativo ao ciclo de vida das *threads*, pois essas estão contidas dentro do escopo do laço de iteração (ver pseudo-código acima), ou seja, ao início de cada iteração as *threads* são criadas e, ao fim, destruídas. Como o número de iterações executadas durante toda simulação é grande, isso acarreta perda de desempenho.

Esse problema será abordado e resolvido na seção 5.3.

5.2 Segunda versão

Em seguida à implementação feita para o cálculo das temperaturas, repetiu-se a abordagem para o cálculo das concentrações dos gases. Da mesma forma, cria-se uma *thread* para cada segmento que é responsável por calcular as pressões [parciais dos gases].

Assim como na versão anterior, também utilizou-se de uma barreira para sincronizar o cálculo das pressões, permitindo que a simulação só continue ao fim de todos os cálculos, como ilustrado a seguir.

```
para tempo ← 0 até TempoDaSimulação
      para iteração ← 0 até NúmeroMáximoDeIterações
            paracada segmento
                  <cria thread para executar PassoCalculoTemperatura()>
            <barreira para sincronizar o cálculo da temperatura>
            (…)
            paracada segmento
                  <cria thread para executar PassoCalculoPressao()>
            <barreira para sincronizar o cálculo da pressão dos gases>
```

Código 3: Criação de threads *para os cálculos de temperatura e pressão e respectivas barreiras.*

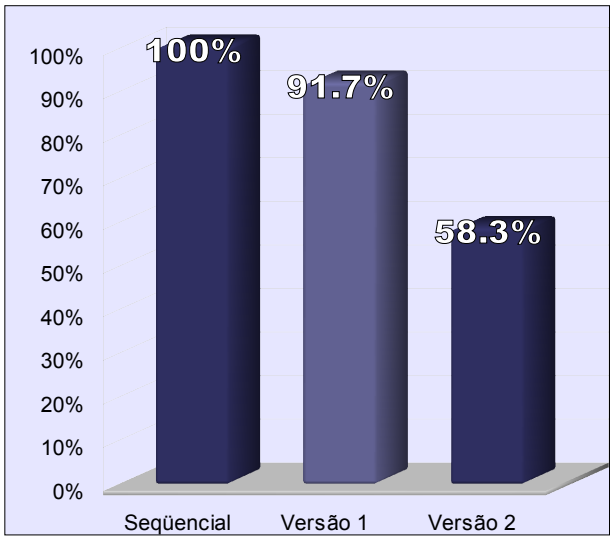

Gráfico 5: Tempo de execução em % do código sequencial.

No cálculo das pressões, a variável compartilhada que representa o compartimento pulmonar é alterada pelo segmento Tronco durante o cálculo das pressões nos capilares pulmonares, quando a pressão e a concentração nos mesmos são atualizadas. Porém, como os outros segmentos não utilizam essas informações, não foi necessário nenhum tipo de controle de concorrência para esse objeto.

A implementação permitiu uma redução de 41.7% no tempo de execução, com base no código sequencial. Já em relação à versão anterior, o ganho foi de 36%.

Assim como a versão anterior, essa versão introduz o mesmo *overhead* relacionado às *threads*, a ser abordado na próxima versão.

5.3 Terceira versão

Como visto nas versões anteriores, a implementação das soluções introduziu gastos excessivos com criação/destruição de *threads*. A principal motivação desta fase do desenvolvimento é a eliminação desses efeitos aumentando, consequentemente, a eficiência da paralelização.

Para atingir esse objetivo, dividiu-se a solução em duas etapas:

1. Agrupar numa mesma *thread* as chamadas de `PassoCalculoTemperatura()` e `PassoCalculoPressao()`;
2. Extrair a criação das *threads* para fora do laço das iterações.

5.3.1 Etapa 1

As chamadas para `PassoCalculoTemperatura()` e `PassoCalculoPressao()` foram agrupadas dentro de uma nova função, `PassoCalculo()`. Para tanto, foi necessário tratar atualizações de parâmetros entre as chamadas das funções.

Isso porque, durante o passo no cálculo das temperaturas, alguns parâmetros da simulação, como informações referentes à geração de calor, sofrem alteração mas devem ter seus valores originais restaurados para a execução do passo do cálculo das pressões.

No código sequencial e nas versões anteriores, essas informações são restauradas após o passo do cálculo da temperatura de todos os segmentos. Contudo, na solução foi preciso implementar uma função que restaura os valores em questão levando em consideração somente as informações particulares de cada segmento, pois na execução de `PassoCalculoPressao()` de um segmento, nem todos os outros necessariamente terminaram suas execuções do passo da temperatura.

```
para tempo ← 0 até TempoDaSimulação
      para iteração ← 0 até NúmeroMáximoDeIterações
            paracada segmento
                  <cria thread para executar PassoCalculo>
            <barreira para sincronizar o cálculo da temperatura e pressão>
```

Código 4: Thread *única para o cálculo de temperatura e pressão.*

Onde,

```
PassoCalculo()
      PassoCalculoTemperatura()
      RestauraInformacoes()
      PassoCalculoPressao()
```

Código 5: Segmento::PassoCalculo().

5.3.2 Etapa 2

O objetivo dessa etapa é posicionar a criação das *threads* fora do laço das iterações. Para isso foi necessária a criação de um nova classe, `ThreadPassoCalculo`, que representará uma *thread* em execução, conterá um segmento e será responsável por invocar a função `PassoCalculo()` desse segmento. Esta classe também se fez necessária para encapsular a lógica de sincronização necessária para a execução. O fluxo de execução dessas *threads* é controlado pela simulação através de sincronizações utilizando três barreiras.

- `barreiraDefinicaoEscopoProximaIteracao`:
  - Ao fim de cada iteração, a simulação define o escopo para a próxima, que pode ser "em execução", quando a variável `tempo` ainda não atingiu `TempoDaSimulação`, ou "terminado", caso contrário. Após definir o escopo, a simulação libera as *threads*, e as mesmas fazem a verificação do escopo. Caso seja de término, as *threads* prosseguem até um ponto de sincronização de fim de execução. Caso contrário, cada *thread* invocará o método `PassoCalculo()` novamente para o segmento.
- `barreiraSincronizaFimDeExecucao`:
  - Quando as *threads* verificam que a simulação definiu o escopo para "terminado", elas seguem para este ponto de sincronização, que tem o objetivo de garantir que quando a execução da simulação terminar, todas as outras *threads* iniciadas não utilizarão mais as variáveis compartilhadas. Isso porque, como as variáveis compartilhadas estão no escopo da simulação, quando esta termina, as variáveis são destruídas e qualquer acesso a elas acarretarar erros.
- `barreiraSincronizacaoDeCalculo`:
  - Enquanto o escopo for de "em execução", esta barreira não permitirá o prosseguimento da simulação até que todos os segmentos tenham efetuado seus cálculos. Análoga às barreiras adotadas nas versões anteriores.

A implementação desta última barreira gerou a necessidade de ser verificada uma possível espera excessiva na mesma. Isso dado que os segmentos possuem diferentes geometrias e composições de tecidos, o que certamente implica em diferentes tempos de cálculo.

O gráfico 6 mostra a proporção do tempo de execução por segmento para toda a simulação. Supondo que esse tempo se distribui homogeneamente durante as iterações da simulação, ou seja, a proporção do gráfico se mantém a cada iteração, pode-se utilizar essa proporção para agrupar mais de um segmento em uma mesma *thread*, diminuindo assim *overhead* de criação e sincronização, e aumentando o uso efetivo de cada processador.

Para o balanceamento, buscou-se agrupar os segmentos em conjuntos cujo somatório do tempo de execução dos elementos se aproximasse o máximo do tempo do tronco, já que este é o limite mínimo de tempo de cada iteração.

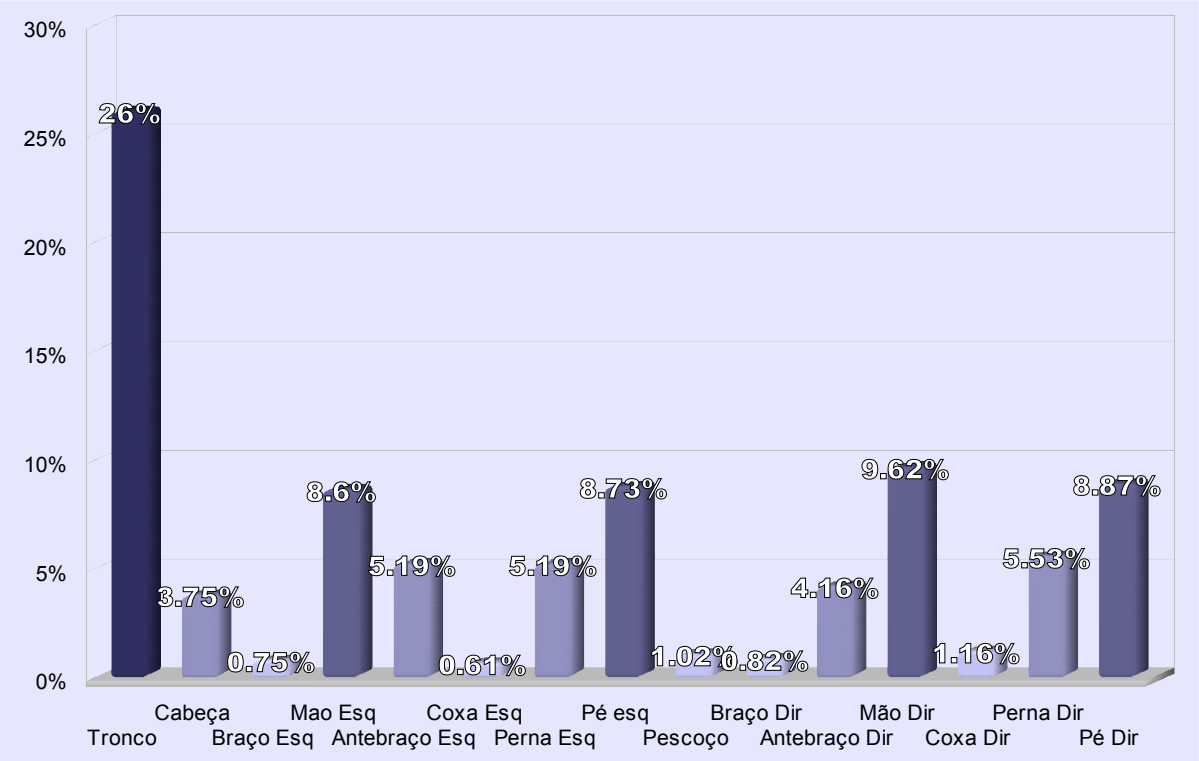

Gráfico 6: Comparativo do tempo de execução entre os segmentos.

Empiricamente, chegou-se a um agrupamento que forneceu o melhor desempenho:

- grupo 1: tronco;
- grupo 2: cabeça, pescoço, perna e coxa direita, perna e coxa esquerda;
- grupo 3: antebraço, braço, mão e pé direito;
- grupo 4: antebraço, braço, mão e pé esquerdo.

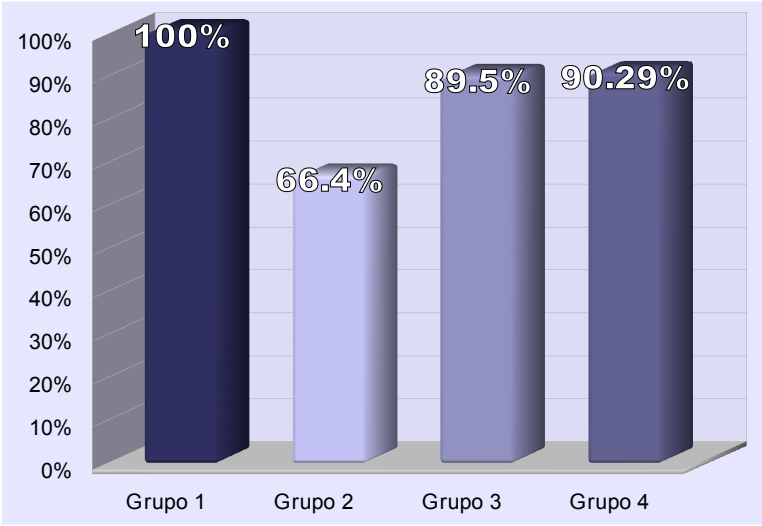

Gráfico 7: Tempo de execução das threads *de cada grupo por iteração (em % do grupo do tronco).*

Foi necessário fazer uma pequena alteração na classe `ThreadPassoCalculo` para que ela trabalhasse com um grupo de segmentos e não apenas com um.

Assim, criando apenas 4 *threads* e não mais uma *thread* por segmento, fez-se com que o uso das cpu's aumentasse, diminuindo em 41,7% o tempo em relação ao código sequencial (4,4% em relação à versão anterior).

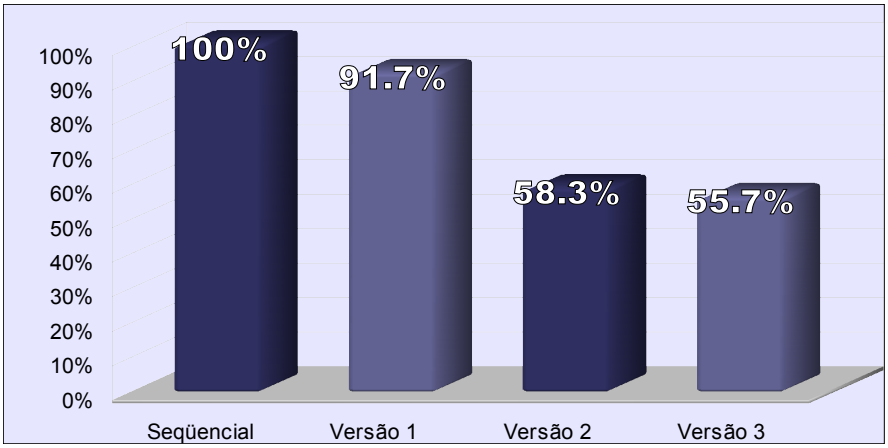

Gráfico 8: Tempo de execução em % do código sequencial.

5.4 Quarta versão

Tendo em vista o resultado das marcações de tempo por segmento realizadas para o balanceamento de carga na versão anterior (gráfico 6), verificou-se que o segmento do tronco tem um grande impacto no tempo da simulação, pois sozinho consome 20% do total, quantidade similar à de grupos com vários segmentos.

Fazendo uma análise mais detalhada do segmento tronco, verificou-se que o compartimento pulmonar é responsável por consumir 60% do tempo gasto pelo mesmo. Isso se deve ao fato de que o compartimento pulmonar concentra toda a troca de gases entre o corpo humano e o ambiente. Esse processo ocorre entre o compartimento alveolar e os capilares pulmonares, onde há muita variação das pressões parciais dos gases nos capilares, tornando mais lenta a convergência dos cálculos.

Para melhorar o processamento no tronco, foram adotadas duas estratégias: otimizar o cálculo das pressões paralelizando alguns compartimentos e paralelizar o cálculo das pressões nos capilares.

A primeira estratégia consiste em deixar o cálculo das pressões no pulmão paralelo ao cálculo das pressões nos outros compartimentos do tronco. Para isso, quando o compartimento do tronco inicia o cálculo das pressões, é criada uma *thread* somente para o cálculo das pressões no pulmão. A *thread* do tronco continua a execução calculando as pressões nos outros compartimentos. Após isso, a *thread* do tronco é sincronizada à *thread* do pulmão através de uma barreira, e a *thread* do tronco só poderá prosseguir quando a *thread* do pulmão terminar.

A segunda estratégia demanda a apresentação do conceito de regime permanente e transitório. No escopo da simulação, o regime permanente se dá quando as equações não estão variando no tempo, o que ocorre no cálculo das condições iniciais da simulação. Em geral, nesse caso, as equações são mais simples.

No fluxo da simulação, não é dado incremento ao passo do tempo em regime permanente.

Complementarmente, o regime transitório acontece quando a variação de tempo é levada em consideração nos cálculos. Esse regime passa a vigorar desde o término do cálculo das condições

iniciais até o termino da simulação.

Tanto em regime permanente quanto em transitório, há uma dependência entre os cálculos das pressões dos capilares pulmonares. Para a execução do cálculo, o capilar depende do valor da concentração no capilar anterior. Dado que os capilares pulmonares estão representados em série no modelo, e na implementação estão armazenados em uma lista. Assim, para o cálculo do capilar na posição "i" é necessário o valor da concentração do capilar na posição "i – 1". Caso o capilar seja o primeiro da lista, ou seja, i vale zero, a concentração de entrada desse capilar será igual à do compartimento venoso.

Em regime permanente, o capilar altera suas concentrações durante o cálculo, fazendo com que o capilar seguinte acesse um valor atualizado como entrada da concentração. Logo, verifica-se a natureza sequencial dessa implementação, pois a ordem de execução deve ser respeitada, não sendo viável sua paralelização.

Em regime transitório, a atualização da concentração nos capilares só é realizada após todos terem efetuado seus cálculos. Logo, é possível a paralelização desse cálculo.

Portanto, foi implementada uma solução híbrida, de cálculo sequencial em regime permanente e paralelo em regime transitório.

Para tanto, na função `PassoCalculoPressao()`, a chamada para a função de cálculo das pressões dos capilares foi substituída por uma chamada a uma nova função, `AtualizaCapilaresPulmonares()`, que faz identificação do tipo de regime, executando o código de maneira sequencial, no caso de regime permanente, ou paralela, se transitório.

Para execução do cálculo sequencial, foi criada a função `AtualizaCapilaresPulmonarSeq()`, onde o cálculo é feito levando em consideração o regime permanente.

Para execução do código em paralelo, é criada uma *thread* para cada capilar pulmonar, que executará a função `AtualizaCapilaresPulmonarPar()`, implementada para o processamento em paralelo.

Ao fim do cálculo, as *threads* dos capilares são sincronizadas com a *thread* do pulmão, para

que seja feita a atualização da concentração dos gases nos capilares.

```
AtualizaCapilaresPulmonares()
      se regimePermanente = true
            então AtualizaCapilaresPulmonarSeq()
      senão
            paracada capilar
                  <cria thread para executar AtualizaCapilaresPulmonarPar()>
            <barreira para sincronizar o cálculo dos capilares>
            AtualizaConcentracoes()
```

Código 6: Paralelização dos capilares pulmonares.

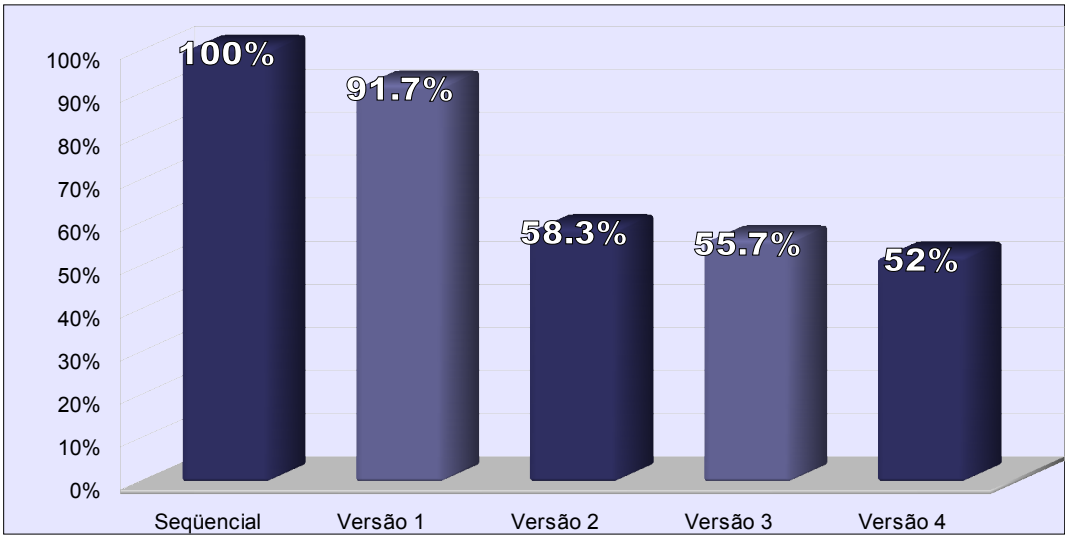

Gráfico 9: Tempo de execução em % do código sequencial.

Ao fim da paralelização dos capilares, tempo de execução em relação ao código sequencial caiu 48% (6.64% em relação à versão anterior).

Capítulo 6

Análise da Limitação de Desempenho

Pode-se notar no gráfico 6 do capítulo anterior, que os seguimentos da mão e pé direito e da mão e pé esquerdo juntos consomem aproximadamente 40% da simulação.

O gráfico 10 representa as parcelas de tempo destinadas aos cálculos da temperatura e pressão nesses segmentos. Por ela, pode-se observar que o cálculo de temperatura ocupa quase completamente o tempo de execução dos mesmos. De fato, o cálculo da temperatura nesses segmentos consome aproximadamente 32% do tempo total da execução.

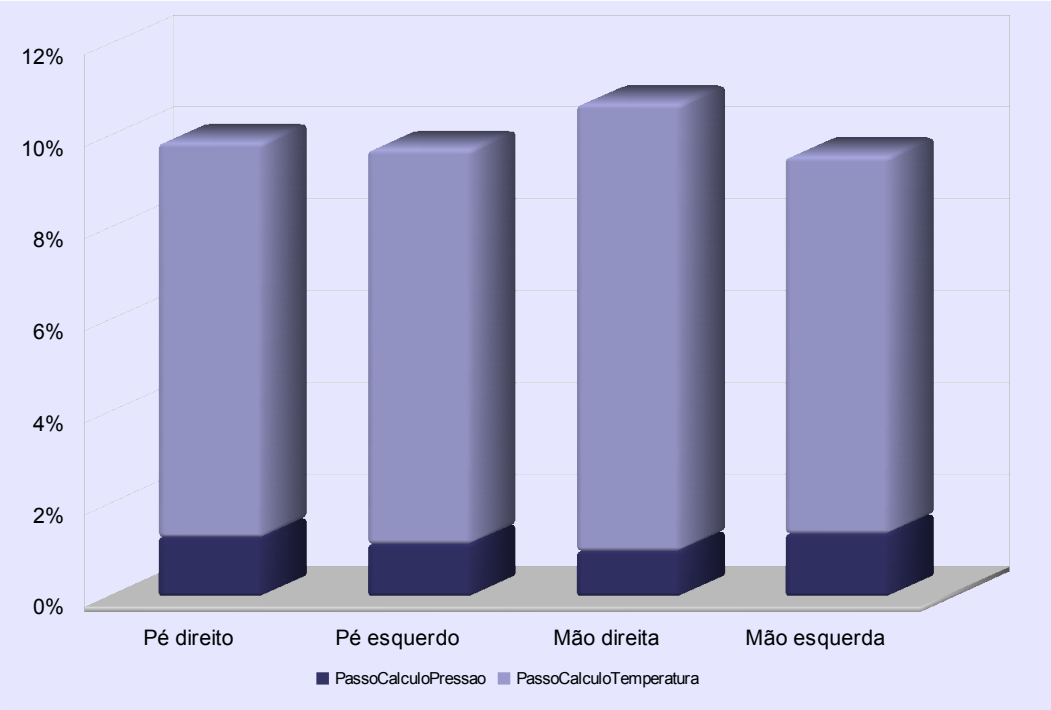

Gráfico 10: tempo destinado aos cálculos da temperatura e pressão nos segmentos retangulares (em % do total).

Assim, de todos os pontos passíveis de otimização, esse é, junto com o cálculo da pressão no tronco, um dos mais importantes.

A seguir, detalhes sobre o cálculo serão apresentados.

Analisando mais a fundo o cálculo da temperatura nestes segmentos, notou-se que todos partilham o mesmo problema, ou seja, o ponto que demanda maior tempo de processamento é comum entre eles, que ocorre quando estes estão atualizando a temperatura de suas células.

O motivo para que esses segmentos gastem nesse cálculo mais do que os outros segmentos está relacionado às suas geometrias. Como mencionado na descrição do modelo, para o cálculo da temperatura nos tecidos são utilizadas equações diferenciais parciais discretizadas pelo método de volumes finitos, ou seja, o cálculo leva em consideração a geometria de cada segmento. Os elementos das malhas geradas são convenientemente designados por "células".

Os segmentos dos pés e mãos são os únicos representados no modelo com geometria retangular. A grande variação de temperatura nas duas direções desses segmentos (figura 6) faz com que o cálculo demore mais para convergir. Diferentemente, os segmentos de geometria não retangular, como o segmento do antebraço, que possui geometria polar, a variação da temperatura é grande somente na direção do raio, mas em θ é muito pequena (figura 7), resultando numa convergência mais rápida do cálculo.

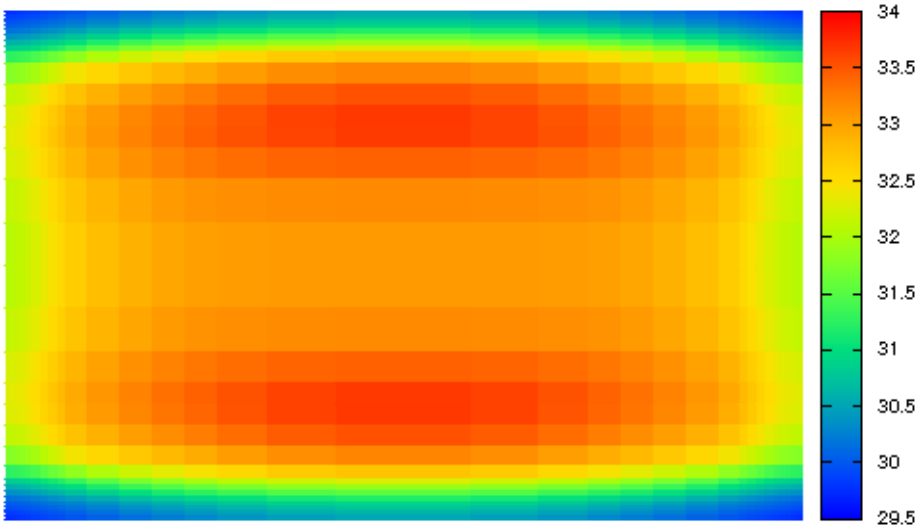

Figura 6: Variação de temperatura na malha da mão.

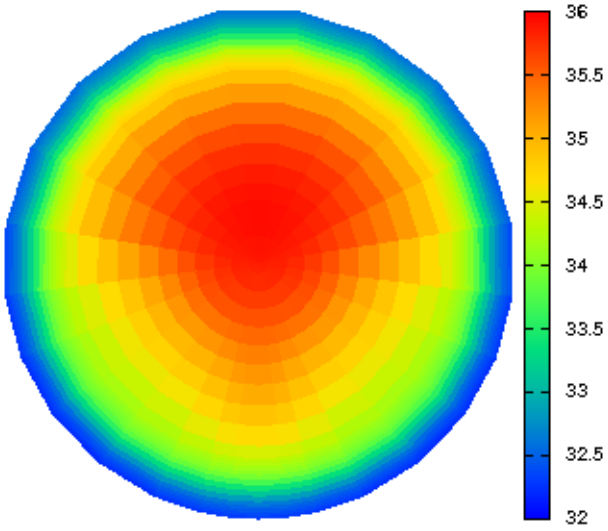

Figura 7: Variação de temperatura na malha do antebraço.

A malha ortogonal é representada por uma matriz bidimensional (pois não é considerada variação no eixo *z*). Na aplicação do método dos volumes finitos implícito, a temperatura de cada célula num determinado instante depende da temperatura de suas vizinhas e de sua própria temperatura em um instante anterior.

Na solução do modelo, a matriz é percorrida sequencialmente linha a linha e depois coluna a coluna, e a temperatura de cada célula é alterada no momento em que é visitada. Assim, como há dependência entre as células, a alteração desse valor durante a varredura afetará o valor a ser calculado nas células vizinhas.

Ou seja, os valores gerados na saída estão intrinsecamente ligados à maneira como o método é aplicado, não sendo possível fazer qualquer tipo de alteração na forma como a matriz é percorrida sem que se altere esses valores.

Portanto nenhuma otimização sequencial ou paralela que altere o modo como a matriz é percorrida pode ser implementada sem inserção de perturbações na saída.

Assim, este problema representa um gargalo para a otimização da simulação, pois consome cerca de 32% do tempo de execução, o que limitará o ganho de desempenho.

Capítulo 7

Conclusões

A partir de frequentes medições realizadas na simulação, foi possível identificar os pontos críticos e desenvolver soluções, quando possível, para otimização.

A solução final consta da união das seguintes implementações:

- A paralelização dos segmentos nos cálculos de temperatura e pressão;
- União das chamadas das funções de cálculo de temperatura e pressão;
- Balanceamento através do agrupamento de mais de um segmento numa mesma *thread*;
- Paralelização do cálculo da concentração de gases nos capilares.

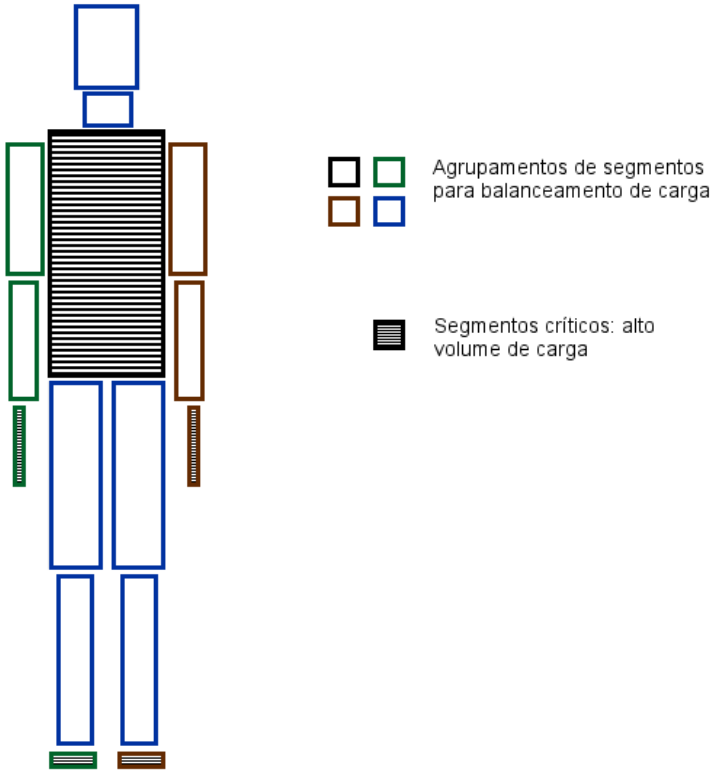

Figura 8: Segmentos críticos e balanceamento de carga.

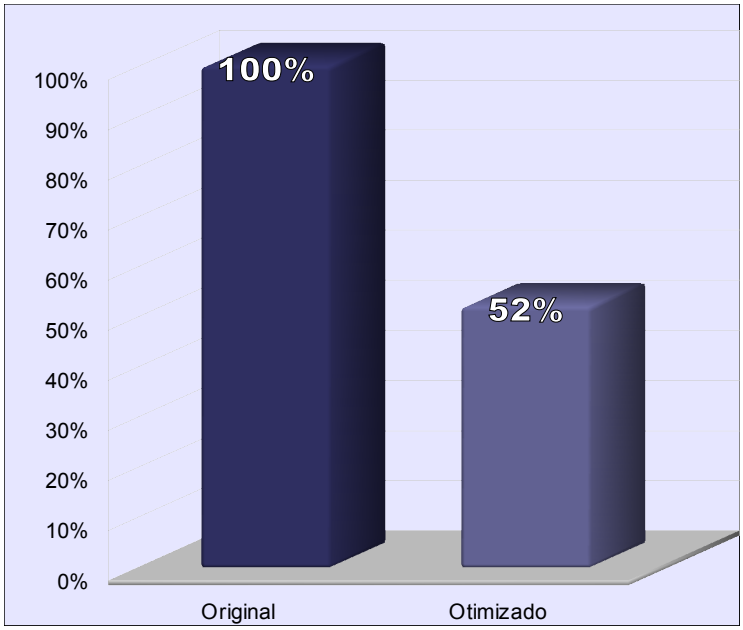

Gráfico 11: Tempo de execução da versão final em % do código original.

As implementações permitiram que o tempo de simulação fosse reduzido quase à metade do original. Trabalhando com 4 processadores, obteve-se um *Speedup* $S_4 \approx 2$, o que foi considerado satisfatório, dado o gargalo identificado no capítulo 6.

Como pode ser observado na figura 9, a simulação otimizada passou a utilizar os recursos computacionais de maneira mais racional, distribuindo a execução dos cálculos entre os processadores. Em contraste, na simulação original há sobrecarga em um dos processadores enquanto os demais permanecem ociosos.

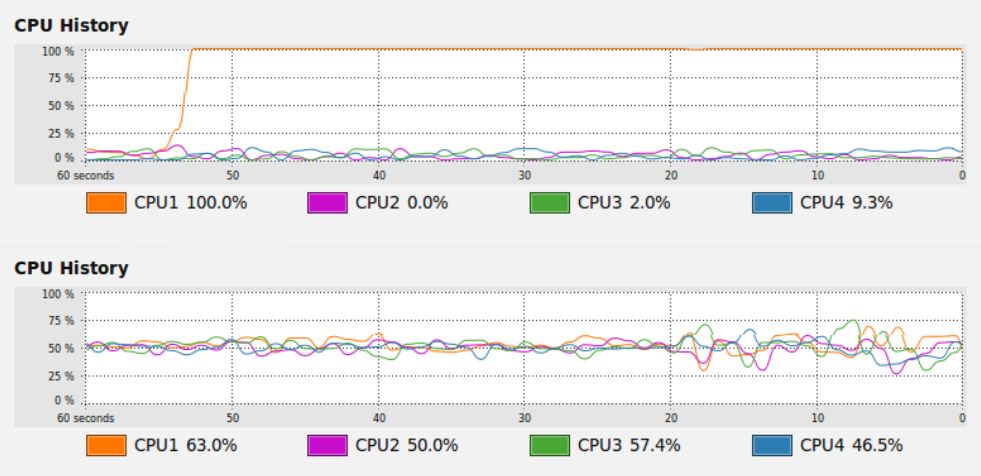

Figura 9: Uso dos processadores com execução sequencial (em cima) e paralelizada (em baixo).

Assim, a otimização da simulação apresentou bons resultados, possibilitando obter uma resposta em menor tempo, e possibilitando também sua aplicação para problemas mais complexos, o que antes era mais custoso devido a ineficiência computacional que acarretava em um longo tempo de execução.

Acredita-se então, que a otimização da simulação agregou valor até mesmo ao modelo, pois o mesmo poderá ser aplicado em estudos de situações que necessitavam de uma abordagem mais complexa, podendo assim ser mais utilizado e difundido pela comunidade científica.

Referências

ALBUQUERQUE-NETO, C.; **Modelo Integrado dos Sistemas Térmico e Respiratório do Corpo Humano.** Qualificação (Doutorado) – Escola Politécnica, Universidade de São Paulo, 2009.

Boost C++ Libraries
http://www.boost.org/

Dr. Dobb's - What's New in Boost Threads?
http://www.ddj.com/cpp/211600441

II – Parte Subjetiva

Fernando

Foi muito gratificante concluir um trabalho cujo tema foi proposto por mim, em conjunto com meu ex-colega de Poli que está desenvolvendo o modelo. Meu orientador, o Professor Gubi, logo observou que a carga de trabalho seria tão grande o quanto determinássemos, e então o Leandro se juntou a nós.

Como eu já havia cursado Programação Concorrente (obrigatória) e Computação Paralela e Distribuída (optativa), já estava familiarizado com os conceitos que iria precisar no trabalho. O grande desafio mesmo, o maior, sem dúvida, foi estudar e entender a tese de doutorado do Cyro [Albuquerque-Neto - autor do modelo]. Para isso contamos com a ajuda do autor da tese, que deu aulas e tirou dúvidas semanalmente. De minha parte, acompanhei o desenvolvimento do trabalho do Cyro desde que me decidi pelo tema, ainda em março. Assistimos (eu e o Leandro) à qualificação do doutorado e sempre tentamos relacionar o que víamos no código ao modelo teórico. Digo isso porque seria virtualmente impossível concluir esse trabalho sem o entendimento do modelo.

Logo na primeira tentativa de paralelização (a versão 1 do capítulo 5), já percebemos que o esforço não seria trivial. Vimos que deveríamos nos aprofundar no entendimento do modelo à medida que fossemos evoluindo em nosso trabalho.

Algumas disciplinas cursadas no BCC foram mais relevantes para o trabalho, cada uma à sua maneira. Posso citar:

- Introdução à Computação Paralela e Distribuída
- Programação Concorrente
- Sistemas Operacionais
- Análise de Algoritmo
- Organização de Computadores
- Estatística II

Duas disciplinas cursadas na poli também ajudaram:

- Métodos Numéricos para Engenharia Mecânica

- Introdução a Teoria de Controle I e II

Obviamente, os conhecimentos adquiridos em Computação Paralela e Programação Concorrente foram os relevantes, sendo diretamente aplicados à solução do problema de paralelizar o modelo. O conhecimento adquirido em S.O. foi muito importante para entender coisas que aconteciam por trás da execução. O mesmo vale para Organização de Computadores. Sendo uma disciplina fundamental, Análise de Algoritmos é relevante para qualquer trabalho desse tipo. Utilizamos também conhecimentos de Estatística II para elaborar métodos de verificar o tamanho do erro gerado entre implementações com perturbação na saída e os valores de referência, do código sequencial (no intuito de verificar se eventualmente estaríamos no caminho certo de produzir uma saída sem erros).

É virtualmente ilimitada a demanda por trabalhos de paralelização em modelos matemáticos desse tipo. É certo que não falta demanda de modelos de combustão e outras aplicações de mecânica dos fluidos, termodinâmica, fenômenos de transporte de calor e massa etc. Isso certamente requer a uma série de aprofundamentos bastante específicos.

Leandro

Desafios e frustrações

No início do primeiro semestre, eu ainda não havia definido um tema. Conversando com o professor Gubi e com o colega Fernando, descobri a possibilidade de desenvolver este trabalho.

O principal motivo que me levou a escolhe-lo, foi a possibilidade de desenvolver um trabalho realmente aplicado, coisa que pouquíssimas vezes aconteceu durante a graduação. Vi neste trabalho a oportunidade de concretizar todo o conhecimento adquirido no curso.

O maior desafio que eu encontrei no desenvolvimento do trabalho foi ter que entender a fundo algumas partes do modelo para identificar qual era a otimização adequada a ser aplicada.

Como o código era muito pouco documentado, para entender qual era o objetivo de certas funções implementadas, tive que ler a parte relacionada na descrição do modelo e fazer a ligação com a parte implementada, o que foi realmente uma tarefa difícil.

Um problema que enfrentei no desenvolvimento, foi a dificuldade de realizar *debug* no código paralelizado utilizando a interface gráfica do eclipse. Algumas vezes, os *breakpoints* eram simplesmente ignorados, ou funcionavam para algumas *threads* e para outras não.

A única frustração que tive durante o trabalho foi quando descobrimos o problema do gargalo, descrito no capítulo X. Este problema limitou muito o ganho de desempenho.

Matérias

As matérias que mais contribuíram para que eu realizasse este trabalho foram:

MAC 441 - Programação Orientada a Objetos

A simulação foi desenvolvida em C++, utilizando orientação a objeto. Portanto, conhecer o paradigma de programação orientada a objetos foi essencial para o desenvolvimento do trabalho.

MAC0342/5716 - Laboratório de Programação Extrema

Foram utilizadas algumas técnicas de XP durante o desenvolvimento. A que mais foi útil para nós foi a programação pareada, o que proporcionou, na maioria das vezes, um rápido desenvolvimento.

MAC 438 - Programação Concorrente

Sem dúvidas, a matéria de programação concorrente foi a mais importante para que eu conseguisse realizar o trabalho proposto. Através dela, obtive todo o fundamento teórico sobre concorrência que precisei aplicar.

MAC 323 - Estruturas de Dados

MAC 122 - Princípios de Desenvolvimento de Algoritmos

Estas matérias fornecem o conhecimento básico indispensável para qualquer tipo de trabalho em computação. Influenciaram, ainda que indiretamente, esse trabalho e todos os que desenvolvi durante a graduação.

Futuro

Os modelos científicos estão em constante evolução. Penso que não será diferente com o modelo representado pela simulação, até porque o mesmo é uma evolução e integração de dois outros modelos. Assim, acho que a adaptação das otimizações implementadas e a implementação de novas serão necessárias a medida que o modelo evoluir.