Daniel Mendes Caldas

Implementações paralelas de um Simulador do Sistema Imunológico

Orientador:
Marcelo Lobosco

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Juiz de Fora

Monografia submetida ao corpo docente do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como parte integrante dos requisitos necessários para obtenção do grau de bacharel em Ciência da Computação

Prof. Marcelo Lobosco, D. Sc.
Orientador

Prof. Rodrigo Weber dos Santos, D. Sc.

Prof. Carlos Cristiano Hasenclever Borges,
D. Sc.

Agradecimentos

Agradeço a todos os meus professores, em especial aos meus orientadores Marcelo Lobosco e Rodrigo Weber dos Santos, que além de dividirem comigo a sua mais importante herança, seu conhecimento, não mediram esforços para contribuir com o meu crescimento intelectual e profissional, me dando a oportunidade de participar do projeto que resultou nesta monografia, de apresentar nosso trabalho e representar o nosso grupo em diversas conferências, e muito mais.

Agradeço aos meus amigos e colegas e do FISIOCOMP, em especial ao meu colega de projeto Bruno Gouvêa, pela oportunidade e honra de trabalhar ao seu lado, e a todos os outros colegas do laboratório, pelas idéias e sugestões, pelo incentivo, e pela oportunidade de fazer deste grupo maravilhoso.

A Todos grandes amigos que possuo e os que conquistei durante esses anos, com os quais sem eles nada faria sentido.

Por fim, agradeço de forma especial aos meus pais Denise e Elcio, e aos meus irmãos, por formarem a base sólida que são, e estarem sempre ao meu lado em todos os momentos.

Resumo

Observamos nos últimos anos um crescente interesse na modelagem matemática e computacional do sistema imunológico humano (SIH). Em particular, a disponibilidade de modelos computacionais para realizar testes e experimentos abre as portas para uma melhor compreensão do SIH, permitindo, por exemplo, o desenvolvimento de novas terapias contra diversas doenças. Apresentamos neste trabalho três implementações paralelas de um modelo computacional que representa o comportamento de duas células do SIH, os linfócitos B e T, assim como uma breve descrição do mesmo. As implementações paralelas foram desenvolvidas utilizando *pthreads* e OpenMP, e obtiveram sucesso em seu propósito de acelerar a execução da versão sequencial do código: ganhos de desempenho de até 4.7 vezes foram obtidos.

Sumário

Lista de Figuras

1 Introdução

O Sistema Imunológico Humano (SIH) consiste de uma rede complexa de células, tecidos e orgãos. O SIH tem um papel crucial na defesa do corpo contra doenças. Para alcançar esse objetivo, o sistema imunológico identifica e elimina uma grande gama de patógenos externos, como vírus e bactérias, e de células do próprio organismo que podem estar se comportando de forma anormal e que poderiam dar origem a tumores caso não fossem eliminadas. O sistema imunológico é ainda responsável pelo processo de eliminação de células mortas e renovação de algumas das estruturas do organismo. A compreensão do funcionamento do sistema imunológico é, portanto, essencial. Entretanto, a sua grande complexidade e a interação entre seus muitos componentes, nos mais diversos níveis, tornam a tarefa extremamente complexa. Alguns de seus aspectos, no entanto, podem ser melhor compreendidos se um modelo computacional é utilizado. Além disso, simulações *in silico* precisam de investimentos muito menores em tecnologia, recursos e tempo, comparadas com experimentos *in vivo*, permitindo aos pesquisadores testar um grande número de hipóteses em um curto período de tempo.

O desenvolvimento de um simulador do SIH não é uma tarefa trivial, envolvendo a discretização e a representação de diversos componentes do SIH, assim como a interação entre eles, em diferentes níveis, o que torna o programa extremamente complexo e computacionalmente caro. Mesmo com o poder computacional disponível nos dias de hoje, alguns modelos e cenários complexos podem precisar de muito tempo para serem processados. Para tratar este problema de desempenho, técnicas de programação paralela devem ser empregadas.

Neste trabalho apresentamos um modelo computacional do SIH que simula a interação entre células B, células T e anticorpos, em diferentes cenários, baseado na teoria da Rede Imunológica (JERNE, 1974). Nosso modelo computacional é uma implementação e extensão de um conhecido modelo matemático da área (CARNEIRO et al., 1996). Em particular, focamos neste trabalho é sua implementação paralela utilizando a biblioteca de *threads* do padrão POSIX (*pthreads*)(BUTENHOF, 1997) e a API de programação paralela

OpenMP(CHANDRA et al., 2001).

Este trabalho está organizado como se segue. O capítulo 1 propicia uma visão geral do Sistema Imunológico. O capítulo 2 discorre sobre técnicas de programação paralela. O capítulo 3 descreve o modelo matemático original utilizado na implementação. Os detalhes da implementação e das versões paralelas do código são apresentadas no capítulo 4, enquanto os resultados obtidos são discutidos no capítulo 5. Finalmente, a seção conclusão concluí o trabalho.

2 O Sistema Imunológico Humano

2.1 Visão Geral

O sistema imunológico é um sistema de estruturas e processos biológicos de um organismo que o protege de doenças, identificando e eliminando células patogênicas e tumorais. Ele detecta uma grande variedade de agentes, desde vírus até vermes parasitas, e precisa distingui-los de células e tecidos sadios do próprio corpo para funcionar corretamente. Esta detecção é complicada pois os patógenos evoluem rapidamente, adaptando-se para evitar o sistema imunológico e infectar o hospedeiro.

Para sobreviver a este desafio, diversos mecanismos evoluíram para reconhecer e neutralizar patógenos. Mesmo simples organismos unicelulares como bactérias possuem sistemas de enzimas que os protegem contra infecções virais. Outros mecanismos imunes básicos que existiam em eucariotos ancestrais evoluíram e se mantêm em seus descendentes modernos, como plantas e insetos. Estes mecanismos incluem peptídeos antimicrobiais chamados defensinas, fagocitose, e o sistema complemento. Vertebrados, incluindo os humanos, possuem mecanismos de defesa ainda mais sofisticados (BECK; HABICHT, 1996).

O sistema imunológico típico dos vetebrados consiste de muitos tipos de proteínas, células, orgãos e tecidos que interagem em elaborada e dinâmica rede. Parte desta resposta imune mais complexa, o sistema imunológico humano se adapta com o tempo para reconhecer patógenos específicos mais eficientemente. Este processo de adaptação é chamado de ”imunidade adaptativa”ou ”imunidade adquirida”e cria uma memória imunológica. A memória imunológica, criada a partir de uma resposta primária específica a um patógeno, fornece respostas reforçadas mais eficientes para encontros secundários com aquele mesmo patógeno específico. Este processo de imunidade adquirida é a base da vacinação. A resposta primária pode demorar de 2 dias a 2 semanas para se desenvolver. Após o corpo obter imunidade a um certo patógeno, quando uma infecção por aquele mesmo patógeno

ocorre novamente, a resposta imunológica é chamada resposta secundária.

Transtornos no sistema imunológico podem resultar em doenças, incluindo doenças auto-imunes, doenças inflamatórias e câncer (COUSSENS; WERB, 2001). Doenças por imunodeficiência ocorrem quando o sistema imunológico está menos ativo que o normal, resultando em infecções recorrentes e sérias. A imunodeficiência pode ser resultado de uma doença genética, como a imunodeficiência combinada grave, produzida por produtos farmacêuticos, ou até mesmo por uma infecção, como a síndrome da imunodeficiência adquirida (AIDS), que é causada pelo vírus HIV. Em contraste, doenças auto-imunes são causadas por um sistema imunologico hiperativo, que ataca tecidos normais como se fossem organismos externos. Doenças auto-imunes comuns incluem tireoidite de Hashimoto, artrite reumatóide, diabetes mellitus tipo 1 e Lúpus eritematoso sistêmico.

2.2 Sistema Inato

O sistema inato é composto por mecanismos de defesa não-específicos, que constituem uma resposta indiferenciada ao agente invasor. Constituem as estratégias de defesa mais antigas, sendo algumas destas formas encontradas nos seres multicelulares mais primitivos, nas plantas e fungos (MAYER, 2007).

2.2.1 Barreiras físicas

A pele é a principal barreira. A sua superfície lipofílica é construida por células mortas ricas em queratina, uma proteína fibrilar, que impede a entrada de microorganismos. As secreções ligeiramente ácidas e lípidicas das glândulas sebácea e sudorípara criam um microambiente cutâneo hostil ao crescimento excessivo de bactérias. O ácido gástrico é uma poderosa defesa contra a invasão por bactérias do intestino. Poucas espécies são capazes de resistir ao baixo pH e enzimas destruidoras que existem no estômago. A saliva e as lágrimas contêm enzimas bactericidas, como a lisozima, que destroem a parede celular das bactérias. No intestino, as numerosas bactérias da microbiota normal competem com potenciais patógenos por nutrientes e locais de fixação, diminuindo a probabilidade de estes últimos se multiplicarem em número suficiente para causar uma doença. É por isso que o consumo demasiado de antibióticos orais pode levar à depleção da microbiota benigna normal do intestino e, com cessação do tratamento, espécies perigosas podem multiplicar-se sem competição, causando, posteriomente, diversas doenças. O muco é outra defesa, revestindo as mucosas. Ele sequestra e inibe a mobilidade dos corpos invasores, sendo a

sua composição hostil para muitos microorganismos. Além disso, contém anticorpos do tipo IgA.

2.2.2 Fagócitos

Os fagócitos são células, como os neutrófilos e macrófagos, que têm a capacidade de estender porções celulares (pseudópodes) de forma direcionada, englobando uma partícula ou microorganismo estranho. Este microrganismo é contido num vacúolo, o fagossoma, que depois é fundido com lisossomas, vacúolos ricos em enzimas e ácidos, que digerem a particula ou organismo. Os fagócitos reagem a citocinas produzidas pelos linfócitos, mas também fagocitam, ainda que menos eficazmente, de forma autônoma sem qualquer estimulação. Naturalmente esta forma de defesa é importante contra infecções bactérianas, já que vírus são demasiado pequenos e a maioria dos parasitas demasiado grandes para serem fagocitados. A fagocitose também é importante na limpeza dos detritos celulares após infecção ou outro processo que leve a morte celular nos tecidos. No entanto, os fagocitos morrem após algumas fagocitoses, e se o número de invasores e de detritos for grande, poderão ambos, fagócitos e bactérias, ficar presos num líquido pastoso e rico em proteínas estruturais, que se denomina pus. Estas células produzem também radicais livres, formas altamente reactivas de oxigênio, que danificam as bactérias e outros invasores, além dos tecidos a sua volta. Algumas bactérias, como o Mycobacterium tuberculosis, agente da tuberculose, têm mecanismos de defesa contra a digestão após fagocitose, e sobrevivem dentro do fagócito, parasitando-o e escondendo-se aí dos linfócitos (MAYER, 2007).

Fagócitos e células relacionadas:

Neutrófilo: granulócito, fagocítico móvel; é o mais abundante e sempre o primeiro a chegar ao local da invasão. Sua morte no local da infecção forma o pus. Ele ingere, mata e digere patógenos microbianos. É derivado dos mastócitos e basófilos.

Macrófago: célula gigante, sendo a forma madura do monócito. Tem capacidade de fagocitar e destruir microorganismos intracelulares. A sua diferenciação é estimulada por citocinas. É mais eficaz na destruição dos microorganismos. Tem vida longa, ao contrário do neutrófilo. É móvel e altamente aderente quando em atividade fagocítica. Macrófagos especializados incluem: células de Kupffer (fígado), células de Langerhans (pele) e células da Glia (sistema nervoso central).

Basófilo e Mastócito: são granulócitos polimorfonucleados que produzem citocinas em defesa contra parasitas. Também são responsáveis pela inflamação alégica mediadas por

IgE.

Eosinófilo: são granulócitos polimorfonucleados que participam na defesa contra parasitas, também participando de reações de hipersensibilidade via mecanismo de citotoxidade. Envolvido em manifestações de alergia e asma, via espeficidade por antígeno IgE.

Os neutrófilos, eosinófilos e basófilos também são conhecidos como polimorfonucleados (devido aos seus núcleos lobulados) ou granulócitos.

2.2.3 Sistema complemento

O sistema complemento é um grupo de proteínas produzidas pelo fígado, presentes no sangue. Elas reconhecem e ligam-se a algumas moléculas presentes em bactérias(via alternativa), ou são ativados por anticorpos ligados a bactérias (via clássica). Então inserem-se na membrana celular do invasor e criam um poro (chamdo de MAC, ou Complexo de Ataque à Membrana), pelo qual entra água excessiva, levando à lise (rompimento osmótico da célula). Outras proteínas não específicas incluem a proteína c-reativa, que também é produzida no fígado e se liga à algumas moléculas comuns nas bactérias, mas inexistentes nas pessoas, ativando o complemento e a fagocitose.

2.2.4 Resposta inflamatória

A resposta inflamatória é fundamentalmente uma reação inespecífica, apesar de ser na prática controlada pelos mecanismos específicos (pelos linfócitos). Caracteriza-se por cinco sintomas e sinais, definidos na antiguidade greco-romana: calor, rubor, tumor (edema), dor e, em último caso (crônicos), perda da função. A inflamação é desencadeada por fatores libertados pelas células danificadas, mesmo se por danos mecânicos. Esses mediadores (bradicinina, histamina) sensibilizam os receptores da dor, e produzem vasodilatação local (rubor e tumor), mas também atraem os fagócitos, principalmente neutrófilos (quimiotaxia). Os neutrófilos que chegam primeiro fagocitam invasores presentes e produzem mais mediadores que chamam linfócitos e mais fagócitos. Entre as citocinas produzidas, as principais são InterLeucina 1 (IL-1) e TNF (Fator de necrose Tumoral) (MAYER, 2007).

2.3 Sistema Adaptativo

O sistema imunológico adaptativo compreende as respostas específicas aos antígenos. Uma vez que o antígeno foi reconhecido, o sistema adaptativo cria um exército de células imunes especificamente desenvolvidas para atacar aquele antígeno. A resposta adaptativa incluí também mecanismos de memória, que tornam respostas futuras para um antígeno específico mais eficientes. O sistema adaptativo é composto por células de um tipo especializado de leucócito, chamado linfócito. Células B e células T são os principais tipos de linfócitos.

2.3.1 Linfócitos B e produção de anticorpos

Os linfócitos B são células que fazem parte de 5 a 15% dos linfócitos circulantes, se originam na medula óssea e se desenvolvem nos órgãos linfóides. O nome linfócito B é devido a sua origem na cloaca das aves na Bursa de Fabricius. São células de núcleo grande e que possuem o retículo endoplasmático rugoso e o complexo de Golgi extremamente desenvolvidos em seu citoplasma, e especialistas em síntese de gamaglobulinas quando ativadas. Porém, em repouso, estas organelas não estão desenvolvidas (BROWNLEE, 2007). Os linfócitos B têm como função própria a produção de anticorpos contra um determinado agressor. Anticorpos são proteínas denominadas de imunoglobulinas ou gamaglobulinas que exercem várias atividades de acordo com o seu isótipo (IgG, IgM, IgA) Estes anticorpos realizam diversas funções como: opsoninas, ativadores de complemento, neutralizadores de substâncias tóxicas, aglutinação, neutralização de bactérias, entre outras. Os linfócitos B possuem como principal marcador de superfície a IgM monomérica, que participa do complexo receptor de antígenos. Esta imunoglobulina entra em contato com o antígeno quando lhe é apresentado diretamente ou indiretamente pelos macrófagos. A IgM se ligando ao epítopo, internaliza o complexo IgM-epítopo. Este complexo realiza diversas modificações na célula, que tem a finalidade de induzi-la a produção de imunoglobulinas. Os linfócitos B em repouso não produzem imunoglobulinas, mas quando estimulados por interleucinas (como a IL-4 e a IL-1) vão sofrer expansão clonal e se transformar numa célula ativa denominada de plasmócito. Os plasmócitos possuem na sua ultra-estrutura o REG e o complexo de Golgi desenvolvido, e o núcleo com aspecto de roda de carroça. Secretam ativamente anticorpos específicos na resposta imune humoral (RIH).

2.3.2 Linfócitos T e suas funções

Os linfócitos T são células que tem diversas funções no organismo, e todas são de extrema importância para o sistema imune. O nome linfócito T derivada das células serem dependentes do timo para o seu desenvolvimento, sendo então o T de Timo-dependentes. Funcionalmente os linfócitos são separados em linfócito T-helper, linfócito T-citotóxico, linfócito T-supressor e linfócito T de memória. Cada um deles possui receptores característicos (além do TCR, que é padrão para as células T), que são identificáveis por técnicas imunológicas e que tem funções específicas. Entretanto, todas as células T possuem os receptores TCR e o CD3. O linfócito T-helper possui receptor CD4 na superfície, que tem a função de reconhecer macrófagos ativados. É o principal alvo do vírus HIV. Esta célula é o mensageiro mais importante do sistema imune. Ele envia mensagens de ataque para diversos leucócitos para realizar a guerra imunológica contra o agente agressor. O linfócito T-helper é a célula que interage com os macrófagos, reconhecendo o epítopo que lhe é apresentado. A IL-1 estimula a expansão clonal de linfócito T-helper monoclonais que vão secretar diversas interleucinas, sendo portanto, dividido em LT helper 1 e LT helper 2. Esses subtipos de LT helper secretam interleucinas distintas, cada uma com uma função específica (JANEWAY et al., 2001).

Algumas funções principais dos linfócitos T-helper:

- estimulação do crescimento e proliferação de linfócito T-citotóxicos e supressoras contra o antígeno;
- estimulação do crescimento e diferenciação dos linfócitos B em plasmócitos para produzir anticorpos contra o antígeno;
- ativação dos macrófagos;
- auto estimulação (um linfócito T-helper pode estimular o crescimento da população de linfócito T- helpers).

Linfócitos T supressores são linfócitos que tem a função de modular a resposta imune através da inibição da mesma. Ainda não se conhece muito a respeito desta célula, mas sabe-se que ele age através da inativação dos linfócitos T citotóxicos e helpers, limitando a ação deles no organismo numa reação imune. Sabe-se que o linfócito T-helper ativa o linfócito T-supressor que vai controlar a atividade destes linfócito T- helpers, impedindo que eles exerçam sua atividades excessivamente (W, 2005). Os linfócitos T-supressores

também participam da chamada tolerância imunológica, que é o mecanismo que o sistema imune usa para impedir que os leucócitos ataquem as próprias células do organismo. Portanto, se houver deficiência na produção ou ativação dos linfócitos T supressores, poderá haver um ataque auto-imune ao organismo.

O linfócito T-citotóxico apresenta receptores TCR, especializado para o reconhecimento de antígenos associados ao complexo MHC-I na superfície de outras células. Produz perforinas e outras proteínas que matam células estranhas, células infectadas por vírus e algumas células cancerosas.

O linfócito T de memória apresenta receptores TCR, e é uma célula preparada para responder mais rapidamente e com maior intensidade, diante de nova exposição ao mesmo antígeno.

2.3.3 Cooperação entre Células T e B

A cooperação entre células T e B é fundamental para a resposta humoral e celular, e esta cooperação é essencial para a homeostase da rede imunológica na ausência de antígenos. A teoria de rede imunológica (JERNE, 1974) apresentou um novo ponto de vista sobre essa cooperação, e vem sendo extensivamente estudada (COUTINHO, 2003; SULZER; WEISBUCH, 1995; BOER; HOGEWEG, 1989; BOER; PERELSON; KEVREKIDIS, 1990) desde então. Um destes estudos (CARNEIRO et al., 1996) descreve o comportamento dos linfócitos no SIH. De acordo com pressupostos e postulados apresentados, o destino de cada clone de célula T e B depende da cooperação com o resto do sistema. Eles podem estar ativos, inativos ou induzidos (células B apenas). Este comportamento é apresentado esquematicamente na Figura 1. A indução de células B acontece pelo acoplamento com epítopes de antígenos nativos (indicado por J na figura), e por moléculas Ig solúveis produzidas por outras células B (K). Sua ativação requer cooperação com células T ativadas (L). Células B ativadas produzem moléculas Ig, dividem-se, e/ou tornam-se células de memória. Células T inativas são ativadas por pepitídeos de antígenos presentes (M). A ativação pode ser inibida por moléculas Ig produzidas por células B (N). Células T ativas podem ativar células B induzidas, se dividir, e/ou se tornar células de memória.

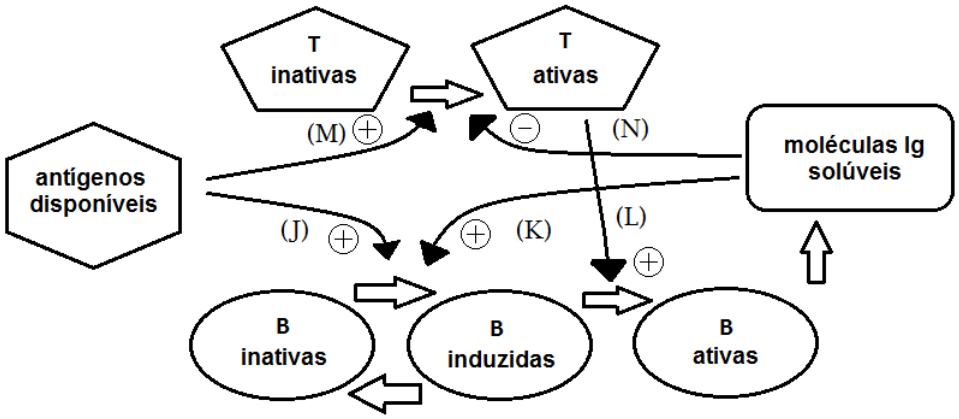

Figura 1: Rede Imunológica (adaptado de (CARNEIRO et al., 1996)).

3 Técnicas de Paralelização

Existem muitas maneiras de se organizar computadores para se realizar computações paralelas. As principais construções paralelas utilizadas nos dias de hoje se encontram na categoria chamada de MIMD na classificação de Flynn (PATTERSON, 2003). Esta categoria pode ser basicamente divididas entre 2 subcategorias: arquiteturas de memória compartilhada, e arquiteturas de memória distribuída. Existe ainda a subcategoria de memória compartilhada distribuída, que possui características mistas das duas anteriores.

3.1 Arquiteturas Paralelas

3.1.1 SMP

Em computação, SMP envolve a arquitetura de computadores multiprocessadores onde dois ou mais processadores idênticos estão conectados a uma única memória principal compartilhada e são controlados por um único Sistema Operacional. Os sistemas multiprocessadores mais comuns hoje usam uma arquitetura SMP. No caso de processadores *multi-core*, a arquitetura SMP se aplica aos cores, tratando-os como processadores separados. O gargalo na escalabilidade dos SMP está no consumo de banda e energia de se interconectar vários processadores, a memória, e os discos. (AL., 2009)

Sistemas SMP permitem qualquer processador executar qualquer tarefa, desde que cada tarefa não esteja em execução em algum outro processador ao mesmo tempo. Com suporte do sistema operacional, sistemas SMP podem mover tarefas entre os processadores facilmente, permitindo-se balancear a carga de trabalho eficientemente.

As principais arquiteturas paralelas que fazem uso de memória compartilhada existentes hoje se enquadram nesta categoria.

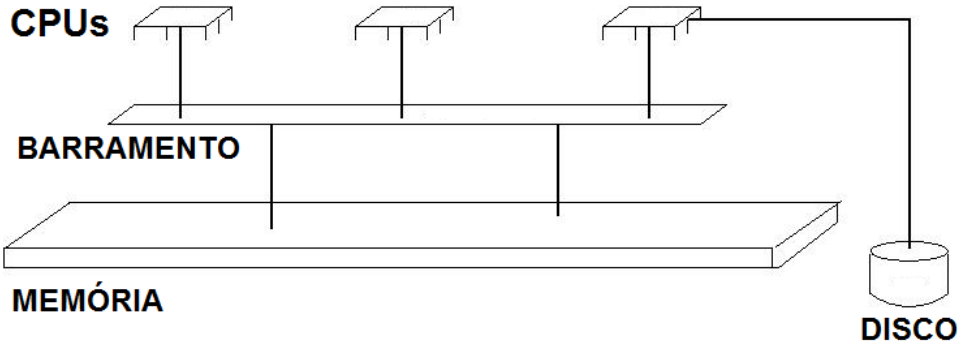

Figura 2: SMP - *Symmetric Multi-Processing*.

3.1.2 MPP

Na arquitetura MPP (*Massive Parallel Processing*), os processadores possuem maior independência entre si, havendo pouco ou nenhum compartilhamento de recursos. Cada nó de um sistema MPP é um computador independente, com memória e disco próprios. Neste tipo de arquitetura o controle de paralelismo é feito pela aplicação. (KALB, 1998)

MPP é controlado por um escalonador que determina quais aplicações executarão em quais nós. Ou seja, não se pode utilizar um nó que não tenha sido alocado à aplicação pelo escalonador.

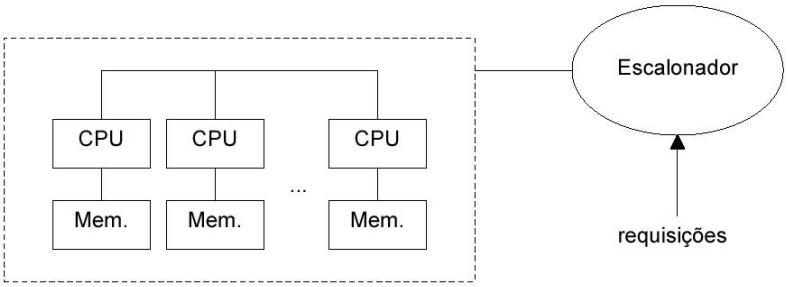

Figura 3: MPP - *Massive Parallel Processing*.

As arquiteturas paralelas que usam memória distribuída são construções deste tipo.

3.2 Programação com Memória Compartilhada

Em computação, memória compartilhada é uma memória que pode ser acessada simultaneamente por múltiplos programas com a intenção de prover comunicação entre eles ou para evitar cópias redundantes. Dependendo do contexto, os programas podem ser executados em somente um processador ou por pelo menos dois processadores distintos.

No *hardware*, a memória compartilhada se refere tipicamente a grandes blocos de RAM que podem ser acessados por diferentes unidades centrais de processamento (CPU) em um sistema de multiprocessamento. Um sistema de memória compartilhada é simples de ser programado já que todos os processadores compartilham a mesma visão dos dados, e a comunicação entre processadores pode ser tão rápida quanto o acesso à memória na mesma posição. O problema com sistema de memória compartilhada é que várias CPUs necessitam acesso rápido à memória e por isso utilizam sistemas de cache na própria CPU, o que possui duas complicações. A primeira é que conexão da CPU para a memória se torna um gargalo no sistema. Computadores com memória compartilhada não possuem boa escalabilidade. A segunda diz respeito à coerência da cache, a condição de que ela esteja atualizada com as informações corretas, mais atuais. Mudanças na cache de um processador devem ser replicadas nos outros processadores.

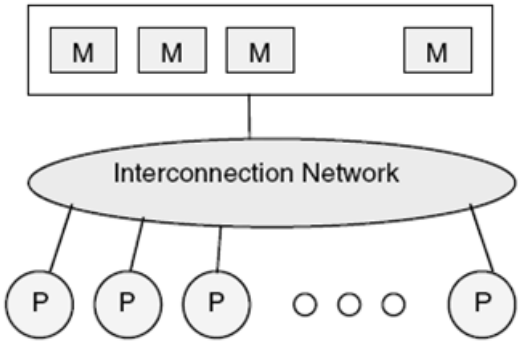

Figura 4: Esquema de Memória Compartilhada.

A programação paralela com memória compartilhada faz o uso dessa arquitetura de memória para obter desempenho em máquinas SMP.

Pontos Positivos:

- Localidade memória/processador;
- Compartilhamento de recursos;
- Semelhante a uma arquitetura seqüencial;
- Programação mais simples.

Pontos Negativos:

- Gargalo no barramento de memória;
- Baixa escalabilidade.

Nas implementações paralelas desenvolvidas neste trabalho, foram usadas as bibliotecas de programação paralela com memória compartilhada *Pthreads* e *OpenMP*.

3.2.1 Pthreads

POSIX Threads, ou *Pthreads*, é o padrão *POSIX* para se trabalhar com *threads*. O padrão define uma API para criação e manipulação de *threads*. Implementações da API estão disponíveis em muitos sistemas Unix como FreeBSD, NetBSD, GNU/Linux, Mac OS X e Solaris, mas implementações para o Microsoft Windows também existem. Por exemplo, o pthreads-w32 está disponível e dá suporte a um subset da API do *Pthreads* para a plataforma Windows 32-bits.

Construções

Pthreads define um conjunto de tipos, funções e constantes para a linguagem de programação C. É implementada com o cabeçalho *pthread.h* e a biblioteca de *threads*. Existem aproximadamente 100 procedimentos na biblioteca *Pthreads*, todos começando com *"pthread_"*, e eles podem ser organizados em quatro grupos:

- Gerenciamento de *Threads*;
- *Mutexes*;
- Variáveis de Condição;

- Sincronização.

Criação e destruição de *threads*

A criação de *threads* é feita com a função *pthread_create* que é chamada da seguinte forma:

```
int  pthread_create(pthread\_t  *  thread, pthread\_attr_t * attr, void *
       (*start\_routine)(void *), void * arg);
```

A função *pthread_create* cria uma nova *thread* que executa concorrentemente com a *thread* que a chamou.

A nova *thread* executa a função *start_routine* passando *arg* como argumento. Ela termina explicitamente, chamando a função *pthread_exit()*, ou implicitamente, retornando da função *start_routine*. O último caso é equivalente a chamar *pthread_exit()* com o resultado retornado para *start_routine* como código de saída.

O argumento *attr* especifica os atributos de *thread* que serão aplicados à nova *thread*. O *attr* pode ainda ser NULL; neste caso os atributos padrões são usados: a *thread* é criada como *joinable* e tem a política de escalonamento padrão.

A destruição de *threads* é feita usando a função *pthread_exit*.

```
void pthread_exit(void *retval);
```

a função *pthread_exit* finaliza a execução da *thread* que a chama. O argumento *retval* é o valor de retorno da *thread*. Pode ser consultado por outra *thread* usando *pthread_join()*.

Junção de *threads*

Uma junção é feito quando deseja-se esperar pelo fim da execução de uma *thread*. A *thread* que chama a rotina pode disparar múltiplas *threads*, e então esperá-las terminarem de executar e pegar os resultados.

```
int pthread_join(pthread_t th, void **thread_return);
```

A função *pthread_join* suspende a execução da *thread* que a chamou até que a *thread* identificada por *th* termine, seja chamando *pthread_exit()* ou sendo cancelada.

Se *thread_return* não é nulo, o valor de retorno de th é armazenado no local apontado por *thread_return.* O valor de retorno da *thread th* é ou o argumento passado para a chamada *pthread_exit*, ou *PTHREAD_CANCELED*, se a *thread th* for cancelada.

Apenas uma *thread* pode esperar pelo término de uma determinada thread. Realizar a chamada de *pthread_join* em uma *thread th* cujo término da execução já esteja sendo aguardado por uma outra *thread* causa um erro.

O código a seguir ilustra a utilização das funções de criação, destruição e junção de *threads* .

```
void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Cria duas threads independentes que irão executar a função */

     iret1 = pthread_create( &thread1, NULL, print_message_function,
(void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function,
(void*) message2);

     /* Aguarda até a execução das threads termine */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
```

```
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Neste exemplo, são criadas duas *threads*, e as duas executam a mesma função. A função simplesmente imprime uma mensagem, que é passa à *thread* como parâmetro. A saída deste código pode ser

```
Thread 1
Thread 2
```

ou

```
Thread 2
Thread 1
```

Dependendo de qual thread termina sua execução primeiro.

Sincronização

Mutexes

Mutex é um mecanismo usado para prevenir inconsistência de dados resultante de operações feitas na mesma área de memória por múltiplas *threads* ao mesmo tempo, ou para evitar condições de corrida onde uma ordem nas operações de memória é esperada. Uma contenção ou condição de corrida normalmente ocorre quando duas ou mais *threads* precisam realizar operações na mesma área de memória, mas os resultados da computação dependem na ordem com que estas operações são realizadas. *mutexes* são usados para serializar recursos compartilhados como a memória. A qualquer momento que um recurso global é acessado por mais de uma *thread*, a fonte deve ter um *mutex* associado a ela. O uso do *mutex* pode ser aplicado para proteger segmentos de memória (regiões críticas) de outras *threads*. *Mutexes* podem ser aplicados apenas em *threads* de um mesmo processo e não funcionam entre diferentes processos como semáforos

O seguinte exemplo ilustra o uso de *mutex* :

```
/* Função C */
void functionC()
```

```
{
   pthread_mutex_lock( &mutex1 );
   counter++
   pthread_mutex_unlock( &mutex1 );
}
```

Quando a operação *pthread_mutex_lock* é realizada em um *mutex* que está preso por outra *thread*, a *thread* é bloqueada até que o *mutex* seja liberado. Quando a *thread* termina, o *mutex* continua bloqueado, a não ser que seja explicitamente liberado com *pthread_mutex_unlock*.

Variáveis de condição

Uma variável de condição é uma variável do tipo *pthread_cond_t* e é usada com as funções apropriadas para que uma *thread* bloqueie sua execução até que uma determinada condição se torne verdadeira, bem como para avisar outras *threads* bloqueadas que uma condição tornou-se verdadeira. Assim, o mecanismo de variáveis de condição permite a *threads* suspenderem sua execução e liberar o processador até que uma certa condição seja verdadeira. Uma variável de condição deve sempre ser associada a um *mutex* para evitar condições de corrida. Uma condição de corrida poderia ocorrer, por exemplo, quando uma *thread* que está se preparando para esperar, enquanto uma segunda *thread* sinaliza a condição aguardada pela primeira antes que esta efetivamente tenha entrado em espera. A primeira *thread* ficaria esperando eternamente por um sinal que nunca seria enviado. Qualquer *mutex* poderia, a princípio, ser utilizado, não existindo qualquer restrição por parte da biblioteca quanto a uma ligação explícita entre o *mutex* e a variável de condição, apesar de ser uma boa prática de programação manter esta ligação, de forma a evitar erros de programação que poderiam levar a deadlocks.

As seguintes funções são usadas em conjunto com variáveis de condição:

- Criação/Destruição:
  - pthread_cond_init
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  - pthread_cond_destroy
- Esperar uma condição:
  - pthread_cond_wait

  - pthread_cond_timedwait - estipula um limite tempo para o bloqueio.

- Acordar uma thread baseado em uma condição:
  - pthread_cond_signal
  - pthread_cond_broadcast - acorda todas as threads bloqueadas pela variável de condição especificada.

O seguinte código ilustra o uso de *mutex* e variáveis de condição:

```
pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

main()
{
   pthread_t thread1, thread2;

   pthread_create( &thread1, NULL, &functionCount1, NULL);
   pthread_create( &thread2, NULL, &functionCount2, NULL);
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionCount1()
{
   for(;;)
   {
```

```
      pthread_mutex_lock( &condition_mutex );
      while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
      {
         pthread_cond_wait( &condition_cond, &condition_mutex );
      }
      pthread_mutex_unlock( &condition_mutex );

      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

      if(count >= COUNT_DONE) return(NULL);
    }
}

void *functionCount2()
{
    for(;;)
    {
       pthread_mutex_lock( &condition_mutex );
       if( count < COUNT_HALT1 || count > COUNT_HALT2 )
       {
          pthread_cond_signal( &condition_cond );
       }
       pthread_mutex_unlock( &condition_mutex );

       pthread_mutex_lock( &count_mutex );
       count++;
       printf("Counter value functionCount2: %d\n",count);
       pthread_mutex_unlock( &count_mutex );

       if(count >= COUNT_DONE) return(NULL);
    }
```

```
}
```

Neste código, são criadas duas *threads* que tem o objetivo de incrementar a variável *count*, que é uma variável compartilhada por ambas as *threads*. Deste modo, antes de realizar qualquer ação com a variável compartilhada, as threads devem adquirir o *mutex*, realizando um *pthread_mutex_lock(condition_mutex)*. Após finalizar a ação com a variável *count* as *threads* liberam o mutex com a função *pthread_mutex_unlock(condition_mutex)*. Além disso, a variável de condição *condition_cond* é usada para que as threads alternem a tarefa de incrementar a variável *count* quando ela vale *COUNT_HALT1* e *COUNT_HALT2*.

3.2.2 OpenMP

Partes desta subseção foram baseadas em alguns capítulos de (CHANDRA et al., 2001)

O OpenMP (*Open Multi-Processing*) é uma API multi-plataforma para processamento paralelo baseado em memória compartilhada para as linguagens C/C++ e Fortran. Ela consiste de um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente que influenciam na execução do programa. O OpenMP foi especificado por um grupo dos grandes fabricantes de *hardware/software* visando o desenvolvimento de um ambiente que fosse portável e escalável, com uma interface de utilização bem simples e que pudesse ser utilizado tanto para aplicações de grande porte, quanto para aplicações simples de *desktop*. Ele foi desenvolvido inicialmente para Fortran em 1997 (pode ser considerado como algo relativamente novo). No ano seguinte, foi lançada a primeira versão para C/C++. Em 2000 foi lançada a versão 2.0 para Fortran e em 2002 foi lançada a versão para C/C++. A versão atual é a 2.5 e saiu em 2005. Nessa versão, finalmente foram combinados os padrões para Fortran e C/C++.

O OpenMP usa um modelo *fork/join* (semelhante a um modelo mestre/escravo). Há um fluxo de execução principal (a *master thread*) e quando necessário (por exemplo, em uma seção paralela), novas *threads* são disparadas para dividir o trabalho. Por fim, ao fim de uma seção paralela, é feito uma junção. Um recurso interessante do OpenMP é que a sincronização entre as *threads* quase sempre ocorre de maneira implícita, de maneira automática. Isso faz com que sua utilização seja algo muito simples. Inclusive, é possível fazer com que os programas compilem com ou sem OpenMP. Caso a biblioteca seja utilizada, eles serão paralelos, caso contrário, seriais. É importante citar algumas características que diferenciam o OpenMP de outras alternativas. No OpenMP não é possível ver como cada *thread* é criada e inicializada. Também não é visível uma função

separada contendo o código que cada *thread* executa. A divisão de trabalho realizado sobre um arranjo também não é visível explicitamente. Enfim, quase tudo acontece ”por trás das cortinas”, de modo que a utilização seja o mais transparente possível.

Construções para divisão de trabalho

Uma das construções chave é o programa *omp parallel*, que declara uma seção paralela. Quando uma seção paralela é encontrada, *threads* são disparadas conforme necessário, e todas elas começam a executar o código dentro daquela seção paralela. As construções para divisão de trabalho citadas a seguir só funcionam dentro de uma seção paralela. Por exemplo:

```
#pragma omp parallel
{
  printf("Hello World!");
}
```

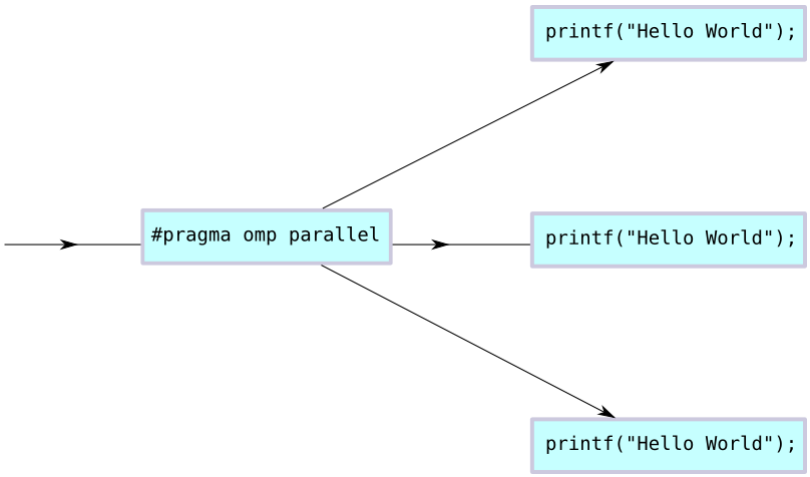

Figura 5: Hello World no OpenMP.

Na definição da seção paralela deve ser passadas várias informações ao compilador, como quais variáveis serão privativas de cada thread e quais serão compartilhadas - isso é feito através das palavras *shared* e *private*. Pode-se também determinar explicitamente quantas *threads* são desejadas, através da palavra *num threads*.

Omp parallel for

A primeira construção importante a ser citada é o *omp_for*. Nesta construção, o intervalo de iteração do *loop* é divido entre as *threads* e essa divisão acontece de forma automática. Por exemplo, dado o seguinte código:

```
#pragma omp parallel for
for(int i = 0; i < 400; ++i)
{
    z[i] = f(x[i], g(y[i]));
}
```

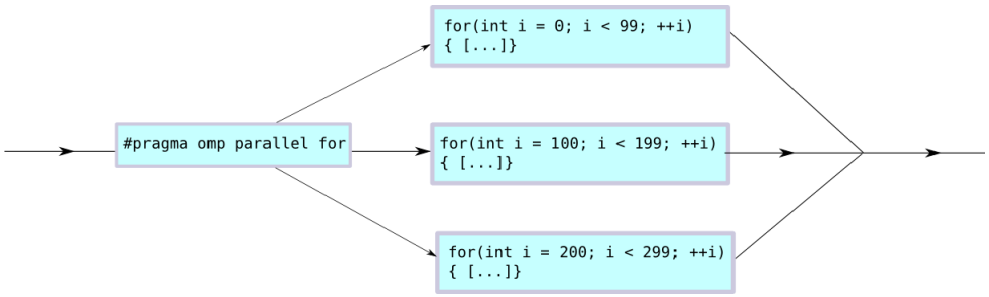

Figura 6: OpenMP *parallel for*.

o corpo do *loop* seria executado uma vez para cada valor de i entre zero e 399, porém não de forma sequencial: os quatrocentos valores nesse intervalo seriam divididos entre as *threads*. Uma divisão possível seria que a *thread* 0 ficasse responsável pelos índices 0 a 99, a *thread* 1 pelos índices 100 a 199, e assim por diante. Essa divisão não é, porém, fixa. É possível alterar (até mesmo em tempo de execução) o número de *threads*, e a divisão de índices se adequará automaticamente.

Omp sections

Outra construção de divisão de trabalho é o *omp sections*. Ela define blocos independentes, que podem ser distribuídos entre as *threads*. Cada *section* vai ser executada apenas por uma *thread*. Trata-se de uma forma simples de paralelizar tarefas distintas que não tem (ou tem pouca) dependência de dados entre si. Um exemplo simples de uso seria a inicialização de dois arranjos com conteúdos distintos e que podem ser inicializado de maneira independente um do outro, como no exemplo a seguir:

```
#pragma omp parallel sections
{
        #pragma omp section
        {
                a();
        }
```

```
        # pragma omp section
        {
                b();
        }

        # pragma omp section
        {
                c();
        }
}

d();
```

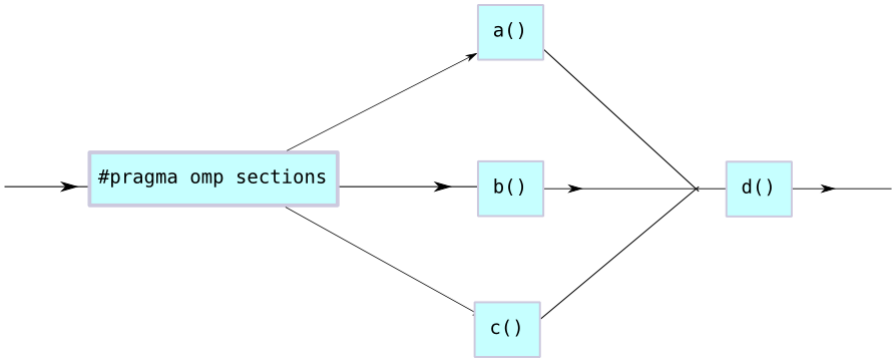

Figura 7: OpenMP *parallel sections*.

Nesse exemplo, as funções a, b e c seriam executadas em paralelo, enquanto a função d só seria chamada após todas as seções paralelas terem sido finalizadas.

Construções para sincronização

Apesar de o OpenMP tentar ser o mais transparente possível para o programador, ainda faz-se necessária a intervenção do programador no momento de definir as operações de sincronização, de modo a garantir consistência entre os estados dos diversos fluxos de execução. As principais construções de sincronização são: *barrier, critical, atomic, single, master, ordered*. *omp barrier* é uma barreira de memória, quando uma thread atinge uma barreira, ela só prossegue após todas as demais atingirem aquela mesma barreira. Esta é a construção mais básica de todas. Algumas construções OpenMP já incluem,

implicitamente, uma barreira no final e, caso o programador não deseje essa barreira, deve incluir uma cláusula *nowait* em seu código.

Funções da biblioteca

Além das diretivas implícitas da biblioteca OpenMP, existe também uma biblioteca com funções que podem ser chamadas de dentro dos programas. É possível determinar com o auxílio da biblioteca, por exemplo, se a seção atual está sendo executada em paralelo ou não (através da função *omp in parallel*). É possível também ligar ou desligar o ajustamento automático do número de *threads*, através das funções *omp_set_dynamic*.

Uma outra forma de especificar parâmetros de execução da aplicação é através do uso de variáveis de ambiente que são lidas pelo sistema de *runtime* do OpenMP. As variáveis disponíveis são: *OMP_SCHEDULE, OMP_NUM_THREADS, OMP_DYNAMIC, OMP_NESTED*. Também é importante comentar sobre as *Locking Functions* e *Timing Routines*. Existem dois tipos de *locks*: *simple* e *nestable*. As simples são *locks* simples, ordinárias. As *nestable* podem ser obtidas diversas vezes pela mesma *thread* que já adquiriu o mesmo *lock* anteriormente. É importante destacar que, neste caso, a *thread* deve liberar o *lock* um número igual de vezes para que outras *threads* consigam obtê-lo. As funções *Locking Functions* e *Timing Routines* são listadas a seguir:

Lock Functions:

- omp_init_lock, omp_init_nest_lock;
- omp_destroy_lock, omp_destroy_nest_lock;
- omp_set_lock, omp_set_nest_lock;
- omp_unset_lock, omp_unset_nest_lock;
- omp_test_lock, omp_test_nest_lock.

Timing Routines:

- omp_get_wtime Function, omp_get_wtick Function.

Vantagens e desvantagens de OpenMP

As principais vantagens de OpenMP são:

- Simplicidade, pois o OpenMP cuida da maioria das coisas para o usuário e a distribuição de tarefas é feita automaticamente pela implementação;

- Paralelizar código já existente é simples, requer poucas modificações, e o paralelismo pode ser implementado de forma incremental, já que não requer grandes modificações estruturais no código. Por exemplo, no caso de um *for*, o OpenMP cuida de quais *threads* irão fazer o que automaticamente e, em geral, o programa continua sendo válido como um programa serial, caso o OpenMP não seja utilizado;
- Compacto, poderoso e simples;

As desvantagens são:

- Requer suporte do compilador;
- Escalabilidade é limitada pela arquitetura da memória;
- Tratamento de erros ainda é um problema;

3.3 Programação com Memória Distribuída

Sua organização é formada por vários computadores, denominados "nós". Estes apresentam suas características particulares como processador, memória e unidade de controle. O processador pode executar uma tarefa independente de acordo com seus próprios dados contidos em sua memória, podendo também executar em paralelo com troca de informações via rede. Logo a estrutura da rede é importantíssima no desempenho da computação paralela.

Os principais pontos positivos do uso de memória distribuída são:

- Compartilhamento de uso (vários usuários simultaneamente);
- Possibilidade de emular outras arquiteturas;
- Escalabilidade.

Os principais pontos negativos do uso de memória distribuída são:

- Comunicação;
- Sincronização;
- Uso ineficiente de memória;

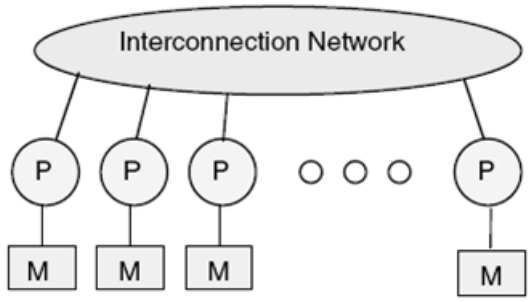

Figura 8: Esquema de Memória Distribuída.

Uma das pricipais tecnologias usadas hoje para se programar com memória distribuída é o MPI (Message Passing Interface), que é uma interface para troca de mensagens entre processos.

4 O Modelo Matemático

Nesta seção reproduzimos parte das Equações Diferenciais Ordinárias (EDOs) que usamos para construir nosso modelo computacional, uma extensão de um modelo anterior (CARNEIRO et al., 1996). Neste modelo, as componentes são: a) linfócitos B periféricos, b) as moléculas Ig que eles produzem, c) os lifócitos T-helper periféricos, e d) os antígenos presentes no corpo. As variáveis que representam estes componentes são: a) B, o tamanho dos clones de linfócitos B; b) F, a concentração de moléculas Ig que eles produzem; c) T, o tamanho dos clones de lifócitos T; e d) A, a concentração de antígenos disponíveis.

Como visto na Equação 4.1, o tamanho de um dado clone de linfócito T T_l decresce exponencialmente com uma taxa de morte constante k_{DT}, e aumenta proporcionalmente com uma taxa de proliferação k_{PT}. O número de células ativadas é dado por α_T, que é determinado por um efeito combinado dos sinais de estímulo (π_l) e inibição (η_l). O sinal de estímulo é unitário ($\pi = 1$, estímulo constante e máximo). O termo $\xi(l,t)$ corresponde a taxa de produção de células do clone l no Timo, órgão onde as células T amadurecem. É uma função que descreve o resultado da seleção tímica, que pode variar com o tempo (t).

$$\frac{dT_l}{dt} = -k_{DT} \cdot T_l + k_{PT} \cdot \alpha_T(\pi_l, \eta_l, T_l) + \xi(l,t) \tag{4.1}$$

O número de células B de um certo clone de linfócitos B (B_i) é dado pela Equação 4.2. É uma função que decresce exponencialmente com uma taxa de morte constante (k_{BD}), e aumenta proporcionalmente a uma taxa de proliferação (k_{PB}) e ao número de células B ativadas no clone (α_B). Esta função é determinada pela amplitude dos sinais de indução (σ_i) e pelo número de células T ativadas especificas disponíveis para cooperação (τ_i). A contribuição da medula óssea é representada pela função $\zeta(i,t)$.

$$\frac{dB_i}{dt} = -k_{BD} \cdot B_i + k_{PB} \cdot \alpha_B(\sigma_i, \tau_i, B_i) + \zeta(i,t) \tag{4.2}$$

A concentração de moléculas Ig solúveis (F_i) é descrita pela Equação 4.3, que decresce com uma taxa proporcional ao número de moléculas livres (k_{DF}) ou ao número de complexos que elas formam com ligantes disponíveis ($k_{DC} \cdot \sigma_i$), e aumenta proporcionalmente (k_{SF}) a mesma função α_B do número de células B ativadas (produtoras de Ig).

$$\frac{dF_i}{dt} = -(k_{DF} + k_{DC} \cdot \sigma_i) \cdot F_i + k_{SF} \cdot \alpha_B(\sigma_i, \tau_i, B_i) \tag{4.3}$$

Maiores detalhes das equações, como as funções α_T, τ, σ, α_B, foram omitidos neste trabalho, visto que o seu foco é a paralelização do modelo.

5 *Implementação*

5.1 Implementação Numérica

Para implementar as EDOs numericamente foi usado o esquema de Diferenças Finitas, e sua implementação numérica foi desenvolvida utilizando a linguagem de programação C. O comportamento das populações em um passo de tempo infinitesimalmente pequeno é descrito pelas EDOs, e o objetivo aqui era executar simulações de grandes intervalos de tempo e representar as quantidades de células destas populações no decorrer da simulação. Para realizar isto, um esquema de diferenças finitas para frente foi usado para aproximar as derivadas. A primeira derivada de uma função de distribuição de população P é, por definição:

$$\frac{dP}{dt} = \lim_{h \to 0} \frac{P(a+h) - P(a)}{h} \tag{5.1}$$

Uma aproximação razoável para a derivada seria assumir

$$\frac{dP}{dt} \approx \frac{P(a+h) - P(a)}{h} \tag{5.2}$$

para valores muito pequenos de h. Então os termos da equação são rearranjados, da forma que

$$P(a+h) = P(a) + \frac{dP}{dt} * h \tag{5.3}$$

Como no modelo sabe-se o valor de $\frac{dP}{dt}$ (a própria EDO) e $P(a)$ (a condição inicial), é fácil calcular $P(a+h)$. Deste modo, assumindo um valor significativamente pequeno para h (0.001), para que a solução possa convergir, o intervalo de tempo T é dividido por h, fornecendo o número de passos de tempo da simulação, e, começando na condição inicial, o valor de $P(a+h)$ é calculado sucessivamente para cada próximo passo de tempo. Ao

final da simulação têm-se todos os valores da função $P(a)$, para todo o intervalo de tempo.

Esta estratégia é usada para calcular os valores das quantidades de células de todas as populações do modelo, a cada passo de tempo. A união destes valores, em todos os passos de tempo, é o resultado da simulação

5.2 Processo de Paralelização

O modelo matemático do SIH implementado neste trabalho é composto por um sistema de três EDOs. Calcular o valor das EDOs (equações 4.1, 4.2, e 4.3) é, na verdade, uma tarefa cara. Elas dependem dos parâmetros τ_i e σ_i, cujos valores precisam ser calculados a cada passo de tempo, para cada população, e estes cálculos são funções complexas(CARNEIRO et al., 1996), da ordem de $O(n^3)$, onde n é o número de populações. Em outras palavras, quando o número de populações cresce, o esforço computacional cresce exponencialmente. Simulações com um grande número de populações de células distintas são interessantes por serem mais realísticas do ponto de vista biológico.

Com o objetivo de reduzir o custo computacional de nosso modelo em máquinas multiprocessadas e/ou com múltiplos núcleos, implementamos três versões paralelas do mesmo, duas utilizando a biblioteca de *threads* do padrão POSIX (*Pthreads*)(BUTENHOF, 1997) e uma utilizando a API de programação paralela de memória compartilhada OpenMP(CHANDRA et al., 2001). Estas versões paralelas dividem o esforço computacional de calcular o valor das EDOs em partes, que são executadas concorrentemente pelos distintos processadores e/ou núcleos.

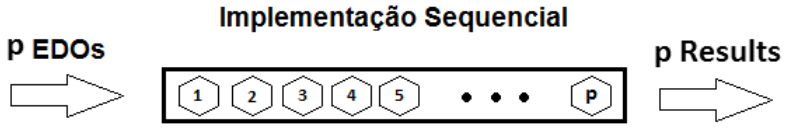

Figura 9: Na implementação sequencial, todo trabalho é feito por apenas um processador/núcleo. Muitas tarefas que poderiam ser feitas ao mesmo tempo ficam paradas, esperando sua chance de executar.

5.2.1 TSP

Na primeira versão utilizando *Pthreads*, a paralelização foi implementada usando um modelo mestre-escravo: a cada passo de tempo, ao invés de executar todo o trabalho

em apenas um processador/núcleo, o fluxo mestre cria um número arbitrário de fluxos escravos e divide entre esses o trabalho de calcular as EDOs de cada população. Assim, com um número p de populações de cada tipo (células T, B e moléculas Ig) e usando n fluxos escravos, cada fluxo será responsável, a cada passo de tempo, pelo cálculo das EDOs de $\frac{p}{n}$ populações de cada tipo. No final de cada passo de tempo, uma operação de sincronização (*join*) é realizada para garantir que os resultados dos cálculos sejam consistentes e que portanto possam ser utilizados pelos demais fluxos no próximo passo de tempo. Esta implementação é chamada de TSP(*threads sem pool*).

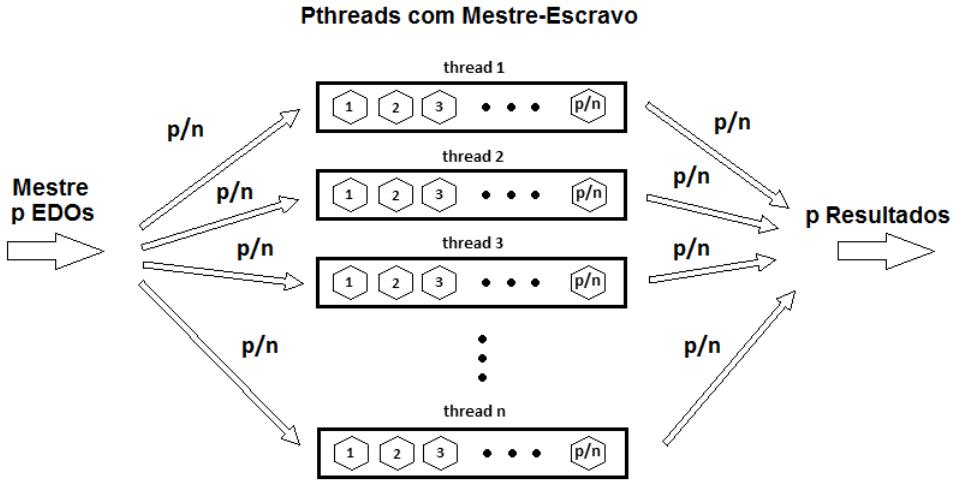

Figura 10: n fluxos dividem a tarefa de calcular p EDOs.

5.2.2 TCP

Na segunda implementação feita com *Pthreads*, novamente utilizamos um modelo mestre-escravo, porém desta vez em conjunto com um *pool* de *threads*. Nesta implementação o fluxo mestre cria um número fixo de fluxos escravos no início do programa, que persistem até o seu término. Então, a cada passo de tempo, ao invés de criar novos fluxos e designar a eles os cálculos das EDOs, o fluxo mestre insere estas tarefas no final de uma fila. Os fluxos escravos removem as tarefas do início da fila, e realizam seu processamento. Quando um fluxo escravo completa a tarefa, simplesmente busca uma nova tarefa a ser executada na fila. A idéia é que o emprego da técnica de *pool* de *threads* possa reduzir o tempo gasto a cada passo de tempo para se criar e destruir fluxos. Chamaremos esta implementação de TCP (*threads com pool*).

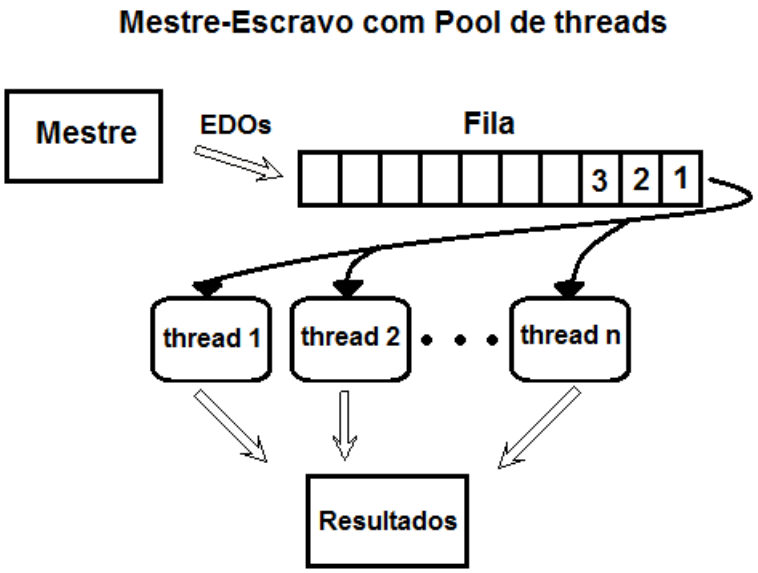

Figura 11: Fluxo mestre insere tarefas no fim de uma fila, enquanto os escravos retiram-as e executam-as.

5.2.3 OpenMP

Para realizar a paralelização do código utilizando OpenMP, adicionamos a diretiva *#pragma omp parallel for* antes do laço que realiza o cálculo das EDOs. Com isso, a cada passo de tempo, um número de fluxos estipulado pelo programador é criado, e divide-se entre eles os cálculos de cada EDO, numa abordagem semelhante a utilizada na versão TSP.

O código usado na implementação OpenMP é listado abaixo:

```
if(PARALELO){
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for
        for (i = 0; i < nPop; i++) {
            tcellPopulationAux[i] = tcellPopulation[i] + h*passoTcell(i);
            bcellPopulationAux[i] = bcellPopulation[i] + h*passoBcell(i);
            igPopulationAux[i] = igPopulation[i] + h*passoIg(i);
```

```
        }

    }
```

6 Avaliação Experimental

6.1 Avaliação Experimental

Os experimentos foram feitos em um processador Intel Core i7-860 de 1.2 GHz, com 8 GB RAM, 8 MB L2 cache. O sistema executa uma versão 64-bits do Linux kernel 2.6.31. O *gcc* versão 4.4.2 foi usado para compilar todas as versões do nosso programa. Apesar do processador i7 ter quatro cores físicos, a tecnologia *Hyper-Threading* (HT) dá a ilusão de que 8 fluxos podem ser executados concorrentemente. Devido as capacidades da tecnologia HT, avaliamos o desempenho das nossas implementações paralelas com até 8 fluxos. O simulador foi executado cinco vezes em todas as versões do código, e o desvio padrão foi menor que 5%. A aceleração (*speedup*) foi obtida dividindo o tempo médio de execução do código sequencial pelos tempos médios de execução das versões paralelas.

Em nossos experimentos, utilizamos os mesmos parâmetros de simulação usados em (CARNEIRO et al., 1996):

k_{DT}	k_{PT}	k_{BD}	k_{PB}	k_{DF}	k_{DC}	k_{SF}
0.15	0.2	0.1	0.3	0.04	0.008	4.0

A quantidade inicial de células utilizada em cada população foi: a) populações de células T = 485 células, b) populações de células B = 250, c) população de moléculas Ig solúveis = 195. Executamos simulações de 500 unidades de tempo, com um h de 0.001, o que portanto equivale a um total de 500.000 passos de tempo em cada simulação.

Para fins de comparação, efetuamos simulações com 8, 16, 32 e 64 populações.

A Figura 12 apresenta os ganhos de desempenho obtidos com a implementação TSP do modelo. O maior speedup foi de 3.66, obtido utilizando-se 8 threads, rodando uma simulações com 64 populações. Nesta configuração, a implementação sequencial demorava 10 horas e 47 minutos, enquanto a implementação TSP apenas 2 horas e 56 minutos. Vemos que para poucas populações o tempo economizado com o uso dos diferentes núcleos

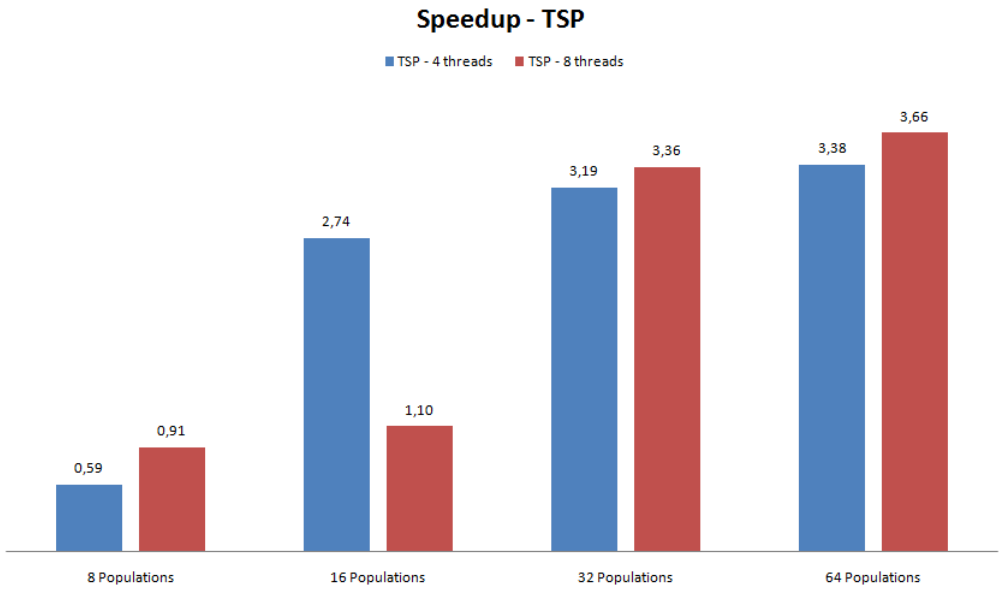

Figura 12: *Speedups* obtidos na implementação TSP.

do processador é menor que o *overhead* de paralelização, o que causa um grande impacto no ganho de desempenho. Este *overhead* está concentrado no processo de criação e junção de *threads*. Isto explica também o menor ganho de desempenho para as execuções com 8 *threads*: o dobro de threads são criadas a cada passo de tempo, causando um *overhead* de paralelização muito maior. Quando o número de populações aumenta, este custo de criação e junção de threads se torna insignificante.

A Figura 13 mostra os ganhos de desempenho obtidos com a implementação TCP. O maior speedup foi de 4.09, obtido utilizando-se 8 threads, e rodando uma simulações com 64 populações. Nesta configuração, o tempo total de execução foi de 2 horas e 36 minutos. Nota-se que, diferentemente da implementação TSP, o *overhead* de paralelização nesta implementação é muito similar nas execuções com 4 e 8 *threads*, pois agora o *overhead* não esta na criação e destruição de *threads*, e sim nas operações de manipulação do *pool de threads*.

A Figura 14 mostra os ganhos de desempenho obtidos com a implementação OpenMP. O maior speedup foi de 4.77, obtido utilizando-se 8 threads, e rodando uma simulações com 8 populações. Nesta configuração, a implementação sequencial demorava 1 minuto e

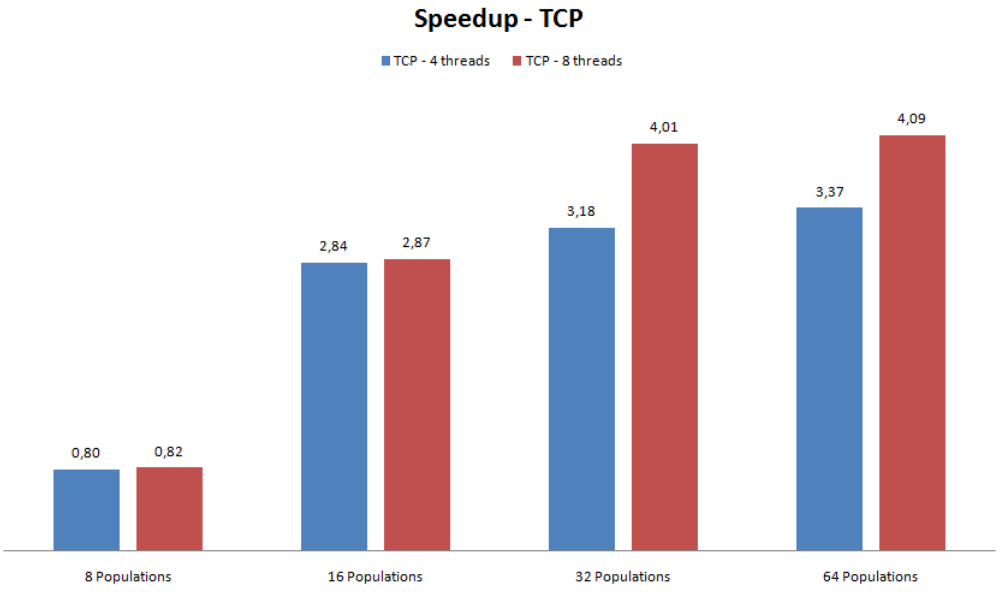

Figura 13: *Speedups* obtidos na implementação TCP.

3 segundos, enquanto a implementação OpenMP com 8 threads apenas 13 segundos. O ganho de speedup permanece estável nas diferentes simulações executadas, se mantendo na casa de 3.2 a 3.6 quando se utiliza 4 threads e entre 4.1 e 4.7 quando se utiliza 8 threads.

A Figura 15 apresenta as curvas de aceleração para as diferentes implementações paralelas do modelo. Nota-se que a melhor aceleração foi obtida pela implementação OpenMP usando 8 threads, obtendo ganhos de 4.1 até 4.7 vezes mais rápidas que a versão sequencial. Pode-se ver também que a implementação do *pool de threads* obteve sucesso em reduzir o overhead de paralelização resultante da criação e join de threads, a cada passo de tempo. As versões Pthreads que utilizam o *pool de threads* são ligeiramente mais rápidas do que as que não utilizam o *pool*.

Como ja foi ressaltado, em ambas implementações Pthreads executando simulações com um número muito pequeno de populações, o *overhead* da paralelização prejudica o ganho de desempenho. Entretanto, à medida que o número de populações aumenta, este custo se torna insignificante. Esta característica não foi investigada pois estamos interessados em melhorar o desempenho do programa para simulações que tem um tempo

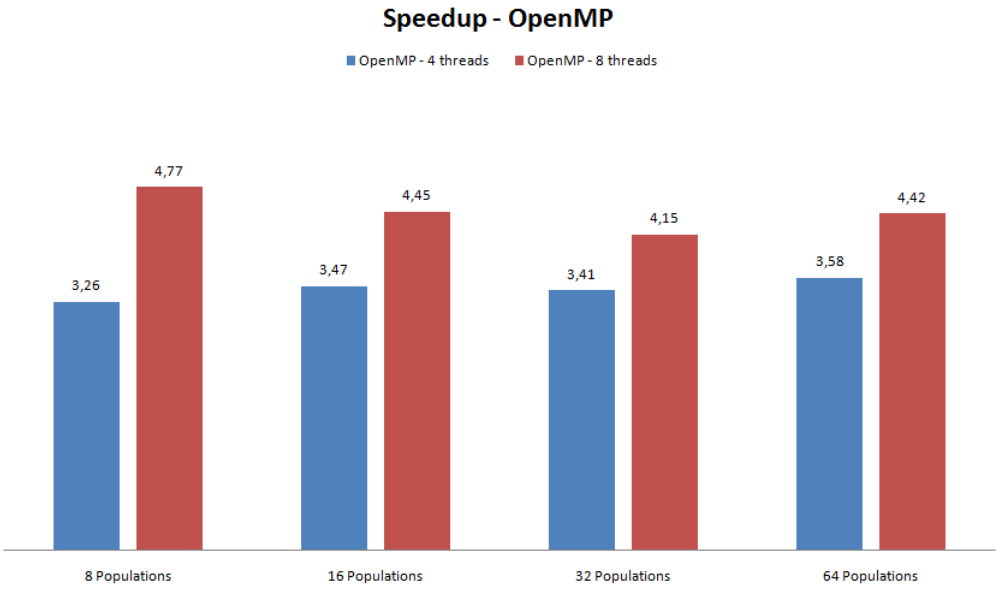

Figura 14: *Speedups* obtidos na implementação OpenMP.

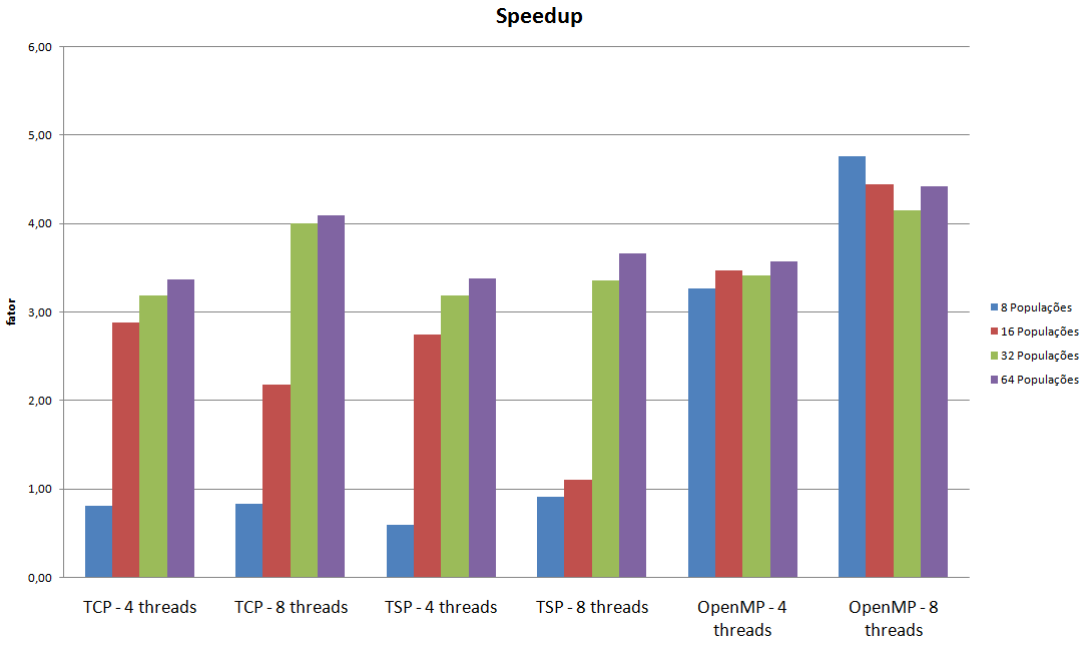

Figura 15: *Speedups* obtidos.

de execução significativo (muitas populações). Observa-se também que este fenômeno não ocorreu nas simulações executadas na implementação OpenMP, ambas com 4 e 8 threads. Uma explicação para isto seria que o OpenMP deve possuir rotinas otimizadas para fazer o tratamento destes *overheads* de paralelização, mas esta é uma hipótese que ainda precisa ser investigada.

7 Conclusão

Neste trabalho apresentamos um modelo matemático e computacional que simula as interações de células B, T, e anticorpos, em diferentes cenários, baseado na teoria da Rede Imunológica de Jerne (JERNE, 1974). Um sistema de EDOs é usado para modelar as interações. Calcular o valor das EDOs é uma tarefa cara: a complexidade deste problema é $O(N^3)$. Com o objetivo de melhorar o desempenho do nosso modelo computacional, apresentamos três implementações paralelas do mesmo. A biblioteca *Pthreads* foi usada em duas das versões, enquanto OpenMP foi usado na terceira versão. Resultados experimentais mostraram que a paralelização foi eficiente para melhorar o desempenho do programa, com ganhos de desempenho de até 4.7 vezes.

Como trabalhos futuros, uma versão mais completa do modelo pode ser implementada, incluindo, por exemplo, funções de afinidade existentes na literatura. Este novo modelo irá precisar de ainda maior poder computacional. Assim, também será necessário desenvolver outra versão paralela do código, fazendo uso de GPGPUs *General-Purpose computation on Graphics Processing Units*)(KIRK; HWU, 2010). As GPUs possuem centenas de núcleos (ABI-CHAHLA; CHARPENTIER, 2008), assim elas poderiam ser usadas de modo que cada núcleo resolva as EDOs de uma população diferente, o que possibilitaria simular cenários com centenas de populações. Acreditamos que usando essa nova versão paralela, o código será capaz de analisar modelos mais complexos e realísticos.

Referências

ABI-CHAHLA, F.; CHARPENTIER, F. Nvidia geforce gtx 260/280 review—specifications: Better! *Tom's Hardware*, 2008.

AL., L. J. K. et. Trends in multi-core dsp plataforms. *IEEE Signal Processing Magazine, Special Issue on Signal Processing on Platforms with Multiple Cores*, 2009.

BECK, G.; HABICHT, G. S. Immunity and the invertebrates. *Scientific American*, p. 60–66, november 1996.

BOER, R. J. D.; HOGEWEG, P. Idiotypic networks incorporating t-b cell co-operation. the conditions for percolation. *Journal of Theoretical Biology*, v. 139, p. 17–38, 1989.

BOER, R. J. D.; PERELSON, A. S.; KEVREKIDIS, I. G. A simple idiotypic network model with complex dynamics. *Chemical Eng. Science*, v. 45, p. 2375–2382, 1990.

BROWNLEE, J. *Antigen-antibody interaction*. [S.l.], April 2007.

BUTENHOF, D. R. *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-63392-2.

CARNEIRO, J. et al. A model of the immune network with b-t cell co-operation. i-prototypical structures and dynamics. *J. Theor. Biol.*, v. 182, p. 513–529, 1996.

CHANDRA, R. et al. *Parallel Programming in OpenMP*. 1. ed. [S.l.]: Morgan Kaufmann Publishers, 2001. ISBN 1-55860-671-8.

COUSSENS, L. M.; WERB, Z. Inflammatory cells and cancer. *Journal of Experimental Medicine*, v. 193, n. 6, p. F23–26, 2001.

COUTINHO, A. A walk with francisco varela from first- to secondgeneration networks: In search of the structure, dynamics and metadynamics of an organism-centered immune system. *Biological Research*, v. 36, n. 1, p. 17–26, 2003.

JANEWAY, C. et al. *Immunobiology*. 5th ed.. ed. [S.l.]: Garland Science, New York and London, 2001.

JERNE, N. K. Towards a network theory of the immune system. *Ann. Immunol. Inst.Pasteur*, v. 125C, p. 373–389, 1974.

KALB, R. M. G. *Massively Parallel, Optical, and Neural Computing in the United States*. [S.l.]: Moxley, 1998.

KIRK, D.; HWU, W. *Massively Parallel Processors: A Hands-on Approach*. [S.l.]: Morgan Kaufmann, 2010.

MAYER, G. *Immunology - Chapter One: Innate (non-specific) Immunity*. [S.l.]: USC School of Medicine, 2007.

PATTERSON, J. L. H. D. A. *Arquitetura De Computadores*. [S.l.]: Campus, 2003.

SULZER, B.; WEISBUCH, G. Idiotypic regulation of b cell differentiation. *Bulletin of Mathematical Biology*, v. 57, n. 6, p. 841–864, 1995.

W, K. D. H. gammadelta t cells link innate and adaptive immune responses. *Chemical Immunology and Allergy*, 2005.