UNIVERSIDADE FEDERAL DA BAHIA

DISSERTAÇÃO DE MESTRADO

Adaptação e avaliação de triagem virtual em arquiteturas paralelas híbridas

Everton Mendonça de Jesus

Programa de Pós-Graduação em Ciência da Computação

Salvador
22 de novembro de 2016

EVERTON MENDONÇA DE JESUS

ADAPTAÇÃO E AVALIAÇÃO DE TRIAGEM VIRTUAL EM ARQUITETURAS PARALELAS HÍBRIDAS

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Marcos Ennes Barreto

Salvador
22 de novembro de 2016

Sistema de Bibliotecas - UFBA

Mendonça, Everton.

Adaptação e avaliação de triagem virtual em arquiteturas paralelas híbridas / Everton Mendonça de Jesus – Salvador, 2016.

42p.: il.

Orientador: Prof. Dr. Marcos Ennes Barreto.

Dissertação (Mestrado) – Universidade Federal da Bahia, Instituto de Matemática, 2016.

1. Computação de Alto Desempenho, 2. Bioinformática, 3. Triagem Virtual, 4. Autodock, 5. CUDA. I. Barreto, Marcos Ennes. II. Universidade Federal da Bahia. Instituto de Matemática. III Título.

CDD –

CDU –

TERMO DE APROVAÇÃO

EVERTON MENDONÇA DE JESUS

ADAPTAÇÃO E AVALIAÇÃO DE TRIAGEM VIRTUAL EM ARQUITETURAS PARALELAS HÍBRIDAS

Esta Dissertação de Mestrado foi julgada adequada à obtenção do título de Mestre em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia.

Salvador, 22 de Novembro de 2016

Prof. Dr. Maycon Leone Maciel Peixoto
Universidade Federal da Bahia

Prof. Dr. Murilo do Carmo Boratto
Universidade do Estado da Bahia

Prof. Dr. Samuel Silva da Rocha Pita
Universidade Federal da Bahia

Prof. Dr. Artur Trancoso Lopo de Queiroz
Fiocruz / BA

Dedico este trabalho à minha mãe, Bárbara Mendonça e à minha companheira, Deise Luz.

AGRADECIMENTOS

Agradeço à minha família e, em especial, a minha mãe Bárbara Mendonça por todo o apoio, incentivo e suporte que sempre proporcionou. À Deise Luz, minha companheira agradeço pela paciência e grande incentivo durante a execução deste trabalho. Ao meu orientador, Prof. Dr. Marcos Ennes Barreto pelo conhecimento passado, incentivo e apoio para a conclusão deste trabalho. Ao Prof. Samuel Pita e todo o pessoal do LaBiMM pelo fundamental apoio para a execução deste trabalho. Ao Prof. Murilo Boratto pelas importantes dicas e direcionamentos. Agradeço também à UFBA e ao PGCOMP pela oportunidade e por tornar viável a execução deste mestrado.

"The true measure of a man is not his intelligence or how high he rises in this freak establishment. No, the true measure of a man is this: how quickly can he respond to the needs of others and how much of himself he can give."

—PHILIP K. DICK

RESUMO

A Triagem Virtual é uma metodologia computacional de busca de novos fármacos que verifica a interação entre moléculas (ligantes) e alvos macromoleculares. Este trabalho objetivou a adaptação de uma ferramenta de Triagem Virtual para arquiteturas paralelas com GPUs e *multicore* e avaliação dos seus resultados, buscando com isso aumentar o desempenho da triagem, reduzindo seu tempo de execução e, consequentemente, permitindo a escalabilidade do número de moléculas envolvidas no processo. A ferramenta escolhida para este propósito foi o Autodock devido a sua ampla adoção dentre os pesquisadores de novos fármacos que utilizam a Triagem Virtual. Três implementações foram criadas abordando diferentes técnicas de paralelismo. A primeira foi uma versão *multicore* onde foi utilizado OpenMP, a segunda foi uma implementação em GPUs utilizando CUDA e por fim, foi criada uma implementação híbrida utilizando a versão *multicore* e a versão para GPUs em conjunto. Em todas as abordagens foram alcançados bons resultados em relação ao tempo de execução total, porém a versão híbrida foi a que obteve os melhores resultados. A versão *multicore* alcançou *speedups*, ou ganhos de desempenho, da ordem de 10 vezes. A versão para GPUs alcançou *speedups* da ordem de 28 vezes e a híbrida de 85 vezes. Com estes resultados foi possível determinar que o uso de plataformas de execução paralelas podem, efetivamente, melhorar o desempenho Triagem Virtual.

Palavras-chave: Computação de Alto Desempenho, Bioinformática, Triagem Virtual, Autodock, CUDA

ABSTRACT

Virtual Screening is a computational method used in research of new drugs thar verifies the interaction between small molecules and target proteins. This work aimed to adapt a Virtual Screening tool to parallel execution platforms using GPUs and multicore architectures and evaluate its results searching with this, increase screening performance, reducing execution time and, consequently allowing the scalability of the molecules involved in the process. The tool chosen to this propose was Autodock due to large adoption between pharmaceutical researchers that use Virtual Screening. Three implementations were created approaching different parallel techniques. The first one was a multicore version using OpenMP; the seconde one was an implementation using GPUs with CUDA and, by the end, it was created a hybrid version using multicore and GPU implementations together. In all these approaches good results were achieved related to total execution time, but the hybrid version achieved the best results. The multicore version reached speedups of the order of 10 times. GPU version reached speedups of the order o 28 times while the hybrid implementation achieved a 85 times speedup. With these results was possible to determine that the use of parallel execution platforms can effectivelly increase Virtual Screening performance.

Keywords: High-performance Computing, Virtual Screening, Bioinformatics, Autodock, CUDA

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

INTRODUÇÃO

Um dos maiores desafios atuais da indústria farmacêutica é a descoberta de novos medicamentos. Neste contexto, a Triagem Virtual é uma metodologia computacional muito útil no processo de descoberta de novos fármacos. O seu objetivo é analisar extensas bases de dados de moléculas, buscando identificar quais delas têm maior potencial de interação com alvos macromoleculares, tornando-as candidatas à avaliação bioquímica em laboratório (FERREIRA; GLAUCIUS; ANDRICOPULO, 2011). Estes alvos macromoleculares utilizados no processo de triagem podem ser componentes conhecidos de doenças diversas.

Conforme dito, através da Triagem Virtual busca-se encontrar uma quantidade de moléculas que possam ser comparadas, dentro de um tempo hábil, visto que as bases de dados podem conter um número expressivo de moléculas (SOLT; TOMIN; NIESZ, 2013). Uma boa interação entre uma molécula e um alvo macromolecular pode significar que essa molécula é uma boa candidata para um futuro medicamento, sendo ideal para testes aprofundados em laboratório. Antes do advento da Triagem Virtual, toda a etapa de testes de interação entre moléculas era feito em laboratório, fazendo com que este processo fosse lento, custoso e exigisse um bom planejamento prévio (FERREIRA; GLAUCIUS; ANDRICOPULO, 2011).

Devido ao grande espaço molecular que pode ser utilizado durante o processo de Triagem Virtual, torna-se necessária a criteriosa seleção das moléculas que serão utilizadas na pesquisa. Ainda assim, como a Triagem Virtual é um processo computacionalmente custoso, mesmo um número reduzido de moléculas pode resultar em um tempo substancial até a conclusão da tarefa.

Tradicionalmente, as aplicações de Triagem Virtual são concebidas com o apoio de softwares específicos, como por exemplo o Autodock, mas além deste, existem diversas outras ferramentas utilizadas nesta área.

Entre essas, o Autodock é um dos softwares mais conhecidos para a Triagem Virtual, sendo extensamente utilizado por pesquisadores em busca de novos fármacos e um dos mais citados por trabalhos desenvolvidos na área de Triagem Virtual conforme visto

em (SOUSA; FERNANDES; RAMOS, 2006). Além disso, o Autodock provê resultados de boa qualidade, encontrando boas interações entre ligantes e alvos macromoleculares. Outro ponto importante é que o Autodock é um software livre, sendo distribuído gratuitamente sob a GNU (General Public License), estando portanto disponível para uso sem restrições. Por conta dos motivos expostos acima, o Autodock foi escolhido para ser adaptado para ambientes de execução paralelas neste trabalho.

Quando se fala em arquiteturas paralelas, diversas técnicas, bibliotecas, modelos e arquiteturas podem ser citadas. Tecnologias como MPI (FORUM, 1994), OpenMP (OPENMP, 2016), Intel Xeon Phi (JEFFERS; REINDERS, 2013) e CUDA (NICKOLLS et al., 2008), são algumas das mais utilizadas e comentadas atualmente.

O advento das arquiteturas híbridas compostas por processadores *multicore* e unidades de processamento gráfico propiciou um melhor suporte computacional para aplicações compostas por diversas tarefas (RAICU; FOSTER; ZHAO, 2008). Entretanto, o desenvolvimento de aplicações em arquiteturas paralelas híbridas demanda um grande esforço de programação acerca do hardware adjacente, dos modelos de execução, da localização dos dados e das ferramentas de programação usadas.

Como comentado anteriormente, a Triagem Virtual é um processo computacionalmente lento e custoso por conta das características de execução das ferramentas disponíveis e também das bases de dados utilizadas, que podem ser muito grandes. Por conta deste fato, adaptar o Autodock para estas plataformas de execução paralela mostra-se uma tarefa importante e que pode gerar um impacto positivo nas buscas de novos fármacos. Com adequações deste tipo, podem ser alcançadas reduções do tempo total de execução da ferramenta, chegando por fim, a um melhor desempenho geral do processo de Triagem Virtual.

Até o momento, a maior parte das abordagens existentes para melhoria de desempenho do Autodock basearam-se na utilização de bibliotecas como OpenMP e MPI, atuando em arquiteturas computacionais restritas e ambientes controlados (NORGAN et al., 2011), tornando sua adoção em ambientes heterogêneos uma tarefa complexa. Em resumo, os pesquisadores trabalhando no desenvolvimento de novos fármacos acabam tendo dificuldade em adotar essas ferramentas otimizadas no seu uso cotidiano.

A partir disso, o trabalho proposto objetivou implementar a Triagem Virtual em ambientes paralelos utilizando para isso uma ferramenta já conhecida e utilizada na área. No caso deste trabalho, foi utilizado o Autodock, pois conforme citado anteriormente, ele é bastante utilizado pelos pesquisadores além de ser um software livre e de código aberto, permitindo que sua adaptação seja feita sem maiores percalços. Os ambientes de execução paralela citados correspondem atualmente às arquiteturas mais avançadas capazes de prover suporte a aplicações intensivas de dados, como é o caso da Triagem Virtual, almejando com isso uma melhora substancial de desempenho, buscando uma redução do tempo de execução da tarefa.

Para tanto o Autodock teve seu código-fonte adaptado para a arquitetura *multicore* através da especificação OpenMP e para GPUs através da biblioteca CUDA. Além destas também foi desenvolvida uma implementação híbrida, que utiliza a solução *multicore* e GPU em conjunto. O objetivo destas implementações foi melhorar o desempenho de execução da ferramenta, promovendo através disso uma maior escalabilidade de execução

da ferramenta.

Com estas implementações se buscou melhorar o desempenho da Triagem Virtual através do uso de plataformas de execução paralela, como arquiteturas *multicore* através do OpenMP e GPUs, utilizando CUDA. Este trabalho está estruturado da seguinte forma: No Capítulo 2 são discutidos os conceitos relativos a Triagem Virtual, sua aplicabilidade na busca de novos fármacos, métodos empregados no planejamento de fármacos, bases de dados disponíveis para uso e ferramentas de Triagem Virtual.

No Capítulo 3, são discutidos os conceitos relativos ao Autodock, demonstrando como é a sua utilização para a execução de uma Triagem Virtual e os algoritmos utilizados por ele, com ênfase no algoritmo genético.

O Capítulo 4 discute as características das plataformas de execução paralela, tratando, mais especificamente, de arquiteturas *multicore* e OpenMP e o uso de GPUs com CUDA.

No Capítulo 5 são mostrados os trabalhos relacionados que efetuaram anteriormente a paralelização do Autodock com diversas técnicas e abordagens. São listados trabalhos que utilizam tecnologias como MPI, OpenMP, Intel Xeon Phi, GPUs, FPGAs entre outras.

Já no Capítulo 6 é apresentado em detalhes o trabalho proposto especificando o objetivo e a metodologia utilizada. Além disso, também é apresentada a base de dados utilizada e por fim, são discutidas as implementações criadas para se alcançar o objetivo proposto.

No Capítulo 7 são demonstrados os resultados alcançados através das implentações desenvolvidas e discutidas no Capítulo 6. Aqui é mostrado o ambiente de execução dos testes, a metodologia dos testes efetuados e gráficos de *speedup* e comparativo de tempo.

Por fim, o Capítulo 8 apresenta a conclusão deste trabalho e discute algumas possibilidades de trabalhos futuros.

Capítulo 2

TRIAGEM VIRTUAL

Neste capítulo são discutidos os conceitos relativos à Triagem Virtual, métodos empregados no planejamento de novos fármacos, bases de dados utilizadas e algumas ferramentas de software utilizadas para este fim.

2.1 CONCEITOS E APLICABILIDADE

Conforme mencionado, a Triagem Virtual pode ser definida como uma metodologia utilizada na pesquisa de fármacos a qual se baseia em técnicas computacionais, através de softwares específicos, para a exploração de grandes bases de moléculas, como por exemplo a PDB (RCSB, 2016) objetivando identificar aquelas que melhor acoplam a um determinado alvo, geralmente uma enzima ou proteína receptora (BOHACEK; MCMARTIN; GUIDA, 1996). A Triagem Virtual busca reduzir o conjunto de moléculas a uma quantidade que possa ser manipulada, comparada e testada em laboratório, uma vez que o espaço exploratório concebível (universo químico) é bastante extenso.

O suporte computacional para a Triagem Virtual pode variar de um simples computador a uma arquitetura paralela dedicada ou um sistema largamente distribuído, dependendo do método de Triagem Virtual utilizado. Em análises baseadas em ligantes, um único processador pode ser utilizado para a comparação de moléculas através de métodos de similaridade ou de padrões farmacofóricos, que são padrões fisicoquímicos que determinam um grupo de moléculas correlatas. Além deste, também existem as análises baseadas na estrutura do receptor, que podem requisitar arquiteturas com maior poder de processamento para a execução de rotinas de acoplamento e funções de escore a fim de selecionar as moléculas mais ativas (RODRIGUES; MANTOANI et al., 2012). Estas funções de escore estimam a afinidade de um dado ligante em uma orientação específica na macromolécula.

A medida que as ferramentas computacionais evoluem, a quantidade de moléculas que podem ser analisadas também aumenta, caracterizando a Triagem Virtual como uma função de complexidade quadrática onde os valores do resultado da execução são proporcionais ao quadrado do valor do argumento. No caso da Triagem Virtual, o argumento

é a quantidade de iterações a serem executadas para cada molécula. Neste caso, por exemplo, se forem definidas um total de 100 execuções para a Triagem Virtual, serão executadas cerca de 10.000 iterações para se finalizar a tarefa.

No caso do Autodock, quanto maior o número de iterações, maior a precisão dos valores encontrados. Tal fato implica diretamente na acurácia dos resultados obtidos, bem como influencia o tempo de execução necessário para as rotinas de comparação e análise de acoplamento.

Na abordagem baseada em triagem prospectiva, os pares candidatos devem ser verificados através de experimentação. Existe um consenso entre pesquisadores de fármacos de que os resultados apresentados pelos testes retrospectivos não são um bom indicador para estimar a acurácia dos testes prospectivos, fazendo com que a análise prospectiva seja largamente explorada atualmente como um método capaz de prover resultados com um alto grau de corretude (CROSS JASON B.; THOMPSON et al., 2009). No caso do Autodock, é utilizada a abordagem prospectiva durante o processo de triagem objetivando com isso encontrar novas interações entre moléculas. A abordagem retrospectiva é utilizada somente nos momentos em que é necessário validar um novo conjunto de moléculas com uma estrutura já conhecida.

2.2 MÉTODOS EMPREGADOS EM PLANEJAMENTO DE FÁRMACOS

Antes do surgimento da Triagem Virtual, uma das técnicas mais utilizadas era a triagem biológica automatizada em larga escala (HTS - *High Throughput Screening*) (ARMSTRONG, 1999). Acreditava-se que a HTS causaria uma revolução na descoberta de novos fármacos, o que fez com que a indústria farmacêutica investisse pesadamente neste processo. Porém, logo suas limitações tornaram-se evidentes. Um grande número de falsos-positivos e o grande número de compostos presentes no espaço químico a ser analisado acabaram tornando essa técnica inviável sem um planejamento prévio sobre quais compostos devem ser avaliados *in vitro* (FERREIRA; GLAUCIUS; ANDRICOPULO, 2011).

Neste contexto, a Triagem Virtual surgiu como uma alternativa a HTS para auxiliar o processo de planejamento de fármacos. Logo ela se tornou uma técnica popular devido ao seu elevado desempenho, versatilidade, baixo custo e facilidade de uso (FERREIRA; GLAUCIUS; ANDRICOPULO, 2011). Outra possibilidade é a utilização das duas técnicas em conjunto, executando inicialmente as rotinas de Triagem Virtual e, com os resultados obtidos, utilizá-los no planejamento dos testes para HTS ou outras técnicas *in vitro*, conforme exposto em (FERREIRA; GLAUCIUS; ANDRICOPULO, 2011).

A Tabela 2.1 demonstra algumas vantagens e desvantagens em relação as técnicas de Triagem Virtual e HTS.

Conforme comentado anteriormente, a Triagem Virtual pode ser executada através de dois métodos, a Triagem Virtual Baseada na Estrutura do Ligante (LBVS - *Ligand-based Virtual Screening*) e a Triagem Virtual Baseada na Estrutura do Receptor (SBVS - *Structure-based Virtual Screening*).

Na utilização do método da Triagem Virtual baseada na estrutura do ligante (LBVS), compostos bioativos conhecidos são utilizados como referência para comparações através

Tabela 2.1 Vantagens e Desvantagens dos Métodos de Triagem Virtual e HTS (FERREIRA; GLAUCIUS; ANDRICOPULO, 2011)

Método	Vantagens	Desvantagens
HTS	- Informação experimental sobre toda a base de dados - Possibilidade de descoberta de inibidores que atuem em diversos sítios do receptor - Independente de determinação prévia de estrutura tridimensional do alvo	- Alta porcentagem de falso-positivos - Grande exigência de infraestrutura
Triagem Virtual	- Número de compostos a serem testados - Menor exigência de infraestrutura - Maior porcentagem de ligantes específicos entre compostos avaliados experimentalmente	- Necessidade de modelo tridimensional do alvo - Ligantes restritos a um determinado sítio de ligação - Falso-negativos

de similaridade química com outros compostos. Dentro do método de LBVS, podem ser utilizadas diferentes técnicas, sendo elas, (FERREIRA; GLAUCIUS; ANDRICOPULO, 2011) o estudo das relações quantitativas entre a estrutura e atividade (QSAR - *quantitative structure-activity relationships*), as impressões digitais moleculares (*molecular fingerprints*), o emprego de farmacóforos (*pharmacophores*) e o aprendizado de máquina (*machine learning*), utilizando *support vector machines* (SVM).

A Triagem Virtual baseada na estrutura do receptor (SBVS) simula interações moleculares entre ligantes e receptores biológicos (principalmente proteínas), permitindo a descoberta de novos ligantes com maior diversidade estrutural quando comparados aos encontrados através da LBVS. Na primeira fase é realizado o acoplamento do ligante no sítio ativo da molécula, processo denominado de docagem. Na segunda etapa, algoritmos de pontuação são executados com o objetivo de predizer a atividade biológica dos ligantes. Para utilização da técnica de SBVS, é necessário que já seja conhecida a estrutura tridimensional da proteína pesquisada, o que acaba por limitar a sua utilização (FERREIRA; GLAUCIUS; ANDRICOPULO, 2011).

A escolha do método adequado depende de diversos fatores, como o tipo de aplicação a ser executada e a qualidade dos dados disponíveis. Apesar disso, os dados gerados a partir da utilização das duas metodologias podem ser amplamente comparados (CHEESERIGHT; SCOFFIN, 2013).

O Autodock utiliza a análise baseada na estrutura do receptor para o processo de Triagem Virtual. Neste caso, é preciso que seja conhecida a estrutura tridimensional das moléculas envolvidas para que seja possível testar o acoplamento entre as mesmas.

O trabalho de Krüger (KRüGER; EVERS, 2010) faz uma comparação entre os métodos de Triagem Virtual baseada no ligante e baseada na estrutura do receptor. Eles comparam quatro diferente proteínas-alvo (ACE, COX-2, HIV e trombina) nas duas metodologias, utilizando diferentes técnicas dentro de cada uma.

Ao final, os autores concluem que a eficiência do método depende muito do alvo molecular pesquisado. Mesmo dentro de cada método, a depender da técnica ou ferramenta empregada, os resultados encontrados podem variar. Isso acontece porque cada ferramenta pode selecionar diferentes ativos para a mesma atividade biológica. Além disso, o mesmo método pode trabalhar melhor em determinadas atividades que outros, já que os receptores são diversos.

Por fim, eles também concluem que os resultados obtidos pelos diferentes métodos também são complementares, sugerindo que, a depender do planejamento, as duas técnicas podem ser utilizadas em conjunto, provendo resultados com uma acurácia maior.

2.3 BASE DE DADOS DE MACROMOLÉCULAS E MICROMOLÉCULAS

Existem algumas bases de dados disponíveis para que os pesquisadores de novos fármacos utilizem durante as tarefas de Triagem Virtual.

Uma base de dados de macromoléculas bem conhecida é a RCSB PDB (RCSB, 2016). A RCSB possui mais de 120 mil estruturas macromoleculares disponíveis para utilização em Triagem Virtual, disponibilizando estruturas tridimensionais das moléculas, entre outras informações.

Em relação as bases de dados de micromoléculas, uma das mais utilizadas pelos pesquisadores da área é a ZINC (ZINC, 2015). A última versão da ZINC, ZINC15 possui mais de 100 milhões de compostos disponíveis para utilização na Triagem Virtual, todas estas com estruturas tridimensionais disponíveis.

Também pode-se citar a base de dados mantido pelo Núcleo de Bioensaios Biosíntese e Ecofisiologia de Produtos Naturais (NUBBE, 2016), onde são disponibilizados os dados químicos, biológicos e tridimensionais de diversos compostos extraídos do bioma brasileiro. Neste trabalho, foram utilizadas no processo de desenvolvimento moléculas disponibilizadas por esta base de dados.

Tabela 2.2 Principais Bases de Dados de Macromoléculas e Micromoléculas

Base	Tipo
RCSB PDB	Macromoléculas
ZINC	Micromoléculas
NUBBE	Micromoléculas

A escolha das moléculas da base de dados NuBBE e do alvo molecular 1BZL do PDB é discutido com mais detalhes na seção 6.2.

2.4 FERRAMENTAS DE TRIAGEM VIRTUAL

Atualmente, grande parte do processo de Triagem Virtual executado por pesquisadores é feito via software, tais como: o Autodock (AUTODOCK, 2015), o DOCK (KUNTZ, 2015), BLAZE (CRESSET, 2015), DOCKTHOR (GMMSB/LNCC, 2015) e o GSA (LSMC, 2015). Entre estes, alguns utilizam abordagens paralelas e distribuídas, tais como OpenMP, MPI, CUDA entre outros.

Existe um número considerável de ferramentas de Triagem Virtual, com as mais variadas abordagens, enfoques e métodos. Este trabalho irá priorizar o estudo, adaptação e utilização do software livre e de código aberto Autodock. Esta escolha baseia-se principalmente no fato do Autodock ser uma ferramenta amplamente utilizada pelos pesquisadores da área de fármacos, sendo utilizado em diversos estudos de variadas naturezas, conforme explicitado em (SOUSA; FERNANDES; RAMOS, 2006). Além disso, o Autodock é um

software livre e tem seu código fonte disponível para adaptação, sendo esta uma característica muito importante para a execução deste trabalho. Por ser um software livre, diversos trabalhos (AUTODOCK, 2015) o tem utilizado para pesquisas em Triagem Virtual, adaptando-o para utilização em ambientes distribuídos e paralelos nas mais diversas arquiteturas e ferramentas, como MPI e OpenMP.

O DOCK (KUNTZ, 2015) também é um software utilizado no processo de Triagem Virtual, sendo direcionado a verificação do acoplamento entre moléculas de um sistema.

O software BLAZE (CRESSET, 2015) é uma ferramenta proprietária para Triagem Virtual. Ela está disponível para uso em clusters de CPU e também de GPU, podendo utilizar a infraestrutura disponibilizada da Cresset, empresa desenvolvedora do BLAZE ou infraestrutura própria, sob a consultoria da empresa.

O DOCKTHOR (GMMSB/LNCC, 2015) é um software desenvolvido pelo Grupo de Modelagem Molecular de Sistemas Biológicos do Laboratório Nacional de Computação Científica (GMMSB/LNCC). Ele foi criado para auxiliar a comunidade acadêmica na análise e utilização dos métodos de acoplamento de moléculas. O DOCKTHOR disponibiliza em seu próprio portal a utilização da ferramenta através da infraestrutura de alto desempenho do SINAPAD (Sistema Nacional de Alto Desempenho).

O GSA (LSMC, 2015) é uma ferramenta desenvolvida na Universidade de Brasília (UNB) e tem sido implementado com sucesso diversas aplicações, como *folding*, *docking* entre outros.

A ferramenta PyRx (PYRX, 2015) também é uma ferramenta de Triagem Virtual utilizada na busca de potenciais novos fármacos. Ela ajuda seus usuários em todas as etapas do processo de Triagem Virtual, desde a preparação dos dados a até a análise dos resultados ao final do processo, além de fornecer uma interface de uso mais simples. O PyRx utiliza diversas ferramentas durante esse processo, entre elas o Autodock para o processo de acoplamento e o Autodock Tools para geração dos arquivos de entrada.

O iDrug (IDRUG, 2015) é um portal que disponibiliza uma interface Web para pesquisa e modelagem de novas drogas. A interface Web fornece ferramentas em tempo real para construção molecular, conversões, visualização e análise de dados.

Glide (GLIDE, 2015) é uma ferramenta para Triagem Virtual que promete oferecer um espectro completo de velocidade e acurácia de resultados, sendo fornecido pela Schrodinger.

Outro exemplo de software para Triagem Virtual é o GEMDOCK (GEMDOCK, 2015), desenvolvido por Jinn-Moon Yang, professor do Instituto de Bioinformática da Universidade Nacional de Chiao Tung. É um software livre para uso em pesquisas tendo versões disponíveis para diversos sistemas operacionais.

Na Tabela 2.3 é apresentado um resumo das ferramentas discutidas acima. Tais programas, apesar de eficientes e eficazes em seus resultados, não são capazes de suportar, de forma isolada, o grande volume de dados disponível nas bases científicas atuais, restringindo o escopo ou mesmo a acurácia dos resultados produzidos. Apesar de existir esforço considerável no intuito de melhorar esse desempenho, bancos de dados como o ZINC, por exemplo, possuem milhões de moléculas disponíveis em sua base. Caso os pesquisadores desejem utilizar um conjunto grande dos dados disponibilizados pela ZINC em suas pesquisas, poderão ter problemas de desempenho para concluir as tarefas de

Tabela 2.3 Algumas Ferramentas Utilizadas na Triagem Virtual

Ferramenta	Principais Características	Licença	Paralelização	Referências
Autodock	Ferramenta para teste de acoplamento entre moléculas. Uso difundido no meio de pesquisa e facilidade de uso	Gratuito	Soluções em MPI, OpenMP, CUDA, Intel Xeon Phi	(MORRIS; HUEY, 2009)
DOCK	Ferramenta para acoplamento de moléculas.	Gratuito	MPI	(KUNTZ, 2015)
BLAZE	Também é uma ferramenta de acoplamento, possuindo também uma versão em cloud.	Pago	Clusters de CPU e GPUs	(CRESSET, 2015)
DOCKTHOR	Sistema de acoplamento de moléculas que utiliza a infraestrutura do LNCC (Laboratório Nacional de Computação Científica)	Gratuito	Não foi possível determinar	(GMMSB/LNCC, 2015)
GSA	Ferramenta gratuita que trabalha efetuando acoplamento e cruzamento de moléculas.	Gratuito	Não foram encontradas referências, mas o código é livre para modificações	(LSMC, 2015)
PyRx	Ferramenta completa para gerenciar todo o processo de Triagem Virtual. Utiliza o Autodock dentro do pipeline para a fase de acoplamento.	Versões antigas tem uso gratuito. Versões mais novas tem custo de uso	Como utiliza o Autodock para a fase de acoplamento, pode se beneficar de soluções paralelizadas deste.	(PYRX, 2015)
iDrug	A ferramenta disponibiliza uma interface WEB para execução de testes de ligação entre proteínas, busca por farmacóforos e testes de acoplamento.	Gratuito	Não é possível determinar.	(IDRUG, 2015)
Glide	Provê um conjunto de soluções para Triagem Virtual com diversos níveis de precisão.	Pago	Sim, possui um esquema de distribuição da carga semelhante ao MPI.	(GLIDE, 2015)
GEMDOCK	Também é uma ferramenta de Triagem Virtual para testar o acoplamento entre moléculas	Gratuito	Não foi possível determinar.	(GEMDOCK, 2015)

Triagem Virtual.

Isto demonstra que é muito importante a busca por novas formas de processar toda essa informação de uma maneira eficiente, pois tradicionalmente a execução destas tarefas em bibliotecas como MPI e OpenMP podem consumir tempo e recursos computacionais consideráveis, tornando a sua utilização lenta e custosa.

Capítulo 3

O que é o Autodock e Como Utilizá-lo

AUTODOCK

Neste capítulo é discutido em mais detalhes o uso do Autodock, qual sua forma de funcionamento, os métodos e algoritmos que ele emprega e como utilizá-lo.

3.1 CONCEITOS

O Autodock é um conjunto de ferramentas de acoplamento para o processo de Triagem Virtual. Ele foi desenvolvido com o objetivo de verificar a interação entre moléculas buscando, a partir disso, descobrir quais desses tem potencial para serem utilizadas como ligantes de determinados alvos macromoleculares (AUTODOCK, 2015). Ele foi desenvolvido no *Molecular Graphics Lab do Scripps Research Institute* como um software livre estando disponível para uso. Além disso, seu código fonte também é disponibilizado para aqueles que desejem fazer alguma modificação no programa.

O Autodock tem aplicação em diversas áreas, como por exemplo cristalografia, desenvolvimento de fármacos, *lead optmization*, Triagem Virtual, entre outros.

Para permitir a observação de todo o espaço configuracional de um ligante em relação a uma proteína, o Autodock utiliza uma técnica de grade para permitir uma análise mais rápida das energias de ligação e testes de estrutura. Durante esta execução, a proteína alvo é inserida na grade. A partir dai, um átomo de teste é colocado em todos os pontos da grade e a interação energética entre este átomo e a proteína alvo é computada e tem seu valor armazendo na grade, disponibilizando-a para busca durante o processo de simulação da acoplamento (MORRIS; HUEY, 2009).

O Autodock utiliza um campo de energia livre semiempírico para prever a interação energética entre os ligantes e a proteína alvo. Este campo de energia livre depende de um modelo termodinâmico que permite a absorção da energia intramolecular entre o ligante e a proteína alvo. Isto pode ser feito através da avaliação das energias de interação nos estados acoplado e desacoplado (HUEY et al., 2007).

Para o processo de acoplamento, o Autodock utiliza diferentes algoritmos para simular as interações entre as moléculas. Ele tem suporte a três deles, o *simulated annealing*,

o algoritmo genético e o algoritmo genético lamarckiano. Para a execução do projeto, foi escolhido e configurado no Autodock o algoritmo genético lamarckiano (LGA - Lamarckian Genetic Algorithm). Esta escolha foi feita pois estudos (MORRIS et al., 1998) demonstram que o LGA é um algoritmo mais eficiente e confiável, obtendo resultados mais robustos e de melhor qualidade.

O Autodock permite que o usuário defina a quantidade de iterações que cada molécula irá fazer na tentativa de fazer a acoplamento. É importante frisar que o algoritmo LGA é estocástico e não-determinístico. Portanto, quanto maior a quantidade de iterações, maior a probabilidade de se encontrar um resultado ótimo, mas isso não é sempre garantido. O resultado ótimo pode ser alcançado com menos iterações, porém a probabilidade disso acontecer é menor.

3.2 ALGORITMO GENÉTICO

Algoritmos Genéticos (AG) são métodos de otimização e busca inspirados nos mecanismos de evolução de populações de seres vivos (LACERDA; CARVALHO, 1999). Estes algoritmos baseiam-se nos mesmos princípios de evolução descritos por Charles Darwin no seu livro A Origem das Espécies.

Em resumo, a ideia principal é buscar os indíviduos que melhor se adaptam ao ambiente durante o processo de evolução. Os mais adaptados vão sendo mantidos durante o processo enquanto que os que não se adaptam vão sendo descartados durante o processo.

O processo de otimização basicamente consiste na busca de uma solução ideal para um problema. Para se alcançar este resultado, várias soluções são tentadas e o resultados destas soluções retro-alimentam o processo até que o resultado final seja encontrado. Para que seja possível fazer esse processo, as técnicas de otimização e busca geralmente apresentam um espaço de busca onde todas as soluções possíveis estão presentes e uma função para avaliar as soluções encontradas durante o processo de otimização.

Este é o processo executado pelo Algoritmo 1 mostrado em (LACERDA; CARVALHO, 1999):

Algoritmo 1 Algoritmo Genético.

Seja $S(t)$ a população de cromossomos na geração t

1: $t \leftarrow 0$
2: inicializar $S(t)$
3: avaliar $S(t)$
4: **enquanto** o critério de parada não for satisfeito **faça**
5: $t \leftarrow t + 1$
6: selecionar $S(t)$ a partir de $S(t-1)$
7: aplicar *crossover* sobre $S(t)$
8: aplicar *mutacao* sobre $S(t)$
9: avaliar $S(t)$
10: **fim enquanto**

Quando o Algoritmo Genético é executado, o primeiro passo é gerar de forma aleatória um conjunto inicial de cromossomos que representam possíveis soluções para o problema. Durante o processo subsequente do AG, esta população é analisada, tendo seus indivíduos devidamente avaliados. A partir deste ponto, os indivíduos considerados mais aptos

são selecionados e os menos aptos descartados. A partir dai, estes indivíduos sofrem modificações através dos processos de *crossover* e mutação, repetindo novamente o passo de avaliação, seleção e descarte, até que uma solução adequada seja alcançada.

Existe uma variação do Algoritmo Genético tradicional definido como Algoritmo Genético Lamarckiano (LGA - *Lamarckian Genetic Algorithm*). Conforme citado, para este trabalho o LGA foi escolhido para execução do Autodock. A principal diferença entre o Algoritmo Genético puro e o LGA, é que o primeiro faz somente buscas globais no espaço de busca definido, enquanto o LGA combina esta estratégia com buscas locais na população próxima ao indivíduo em questão. Esta estratégia proporciona uma maior possibilidade de encontrar bons resultados ao final do processo, pois conforme citado anteriormente, a utilização do LGA provê melhores resultados que o uso do Algoritmo Genético tradicional. (MORRIS et al., 1998).

3.3 COMO UTILIZAR

O processo de Triagem Virtual consiste de várias etapas. A primeira delas é a definição do problema e seleção das moléculas que irão compor o espaço de busca. Existem diversas bases de moléculas públicas e privadas disponíveis na internet, como por exemplo a PDB (RCSB, 2016) e a ZINC (ZINC, 2015). Após essa seleção é necessário fazer o ajuste das moléculas, adicionando hidrogênios e outras partes necessárias. Por fim, deve-se calcular a carga das moléculas e assinalá-las para cada átomo.

A Figura 3.1 apresenta o fluxo do Autodock durante a execução de uma Triagem Virtual.

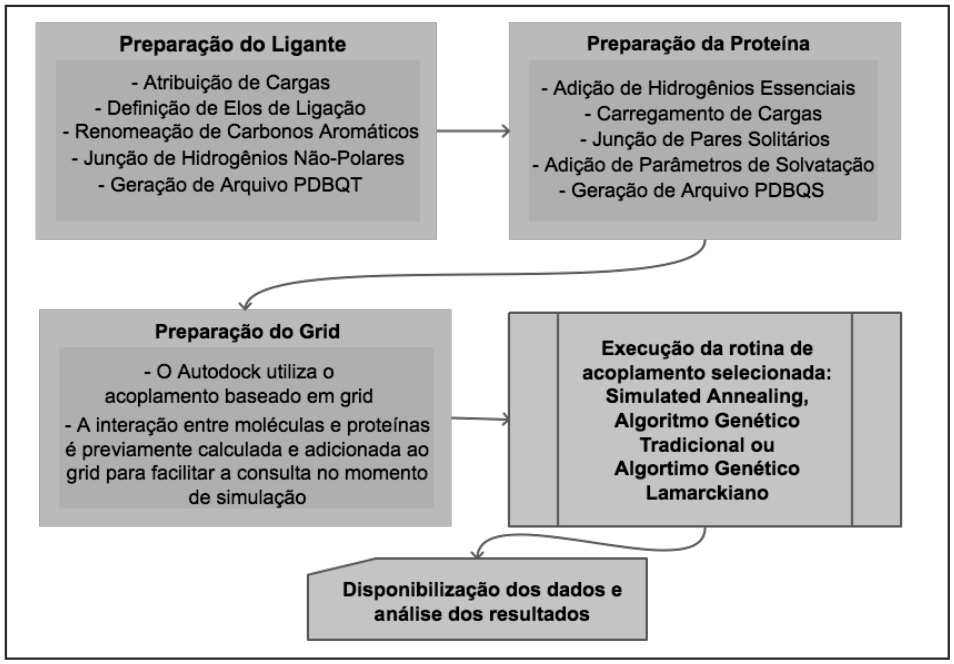

Figura 3.1 Fluxo da Triagem Virtual no Autodock

Existe um software chamado Racoon (AUTODOCK, 2015) que auxilia nas etapas de preparação dos arquivos, automatizando várias tarefas e gerando um script de execução do Autodock ao final do processo. Ele está disponível para download no site do Autodock. Uma etapa importante é a validação das parametrizações, avaliando cargas, grids entre outros parâmetros. Para isso, é utilizada uma molécula de atividade já conhecida, utilizando-a na parametrização construída. Ao final verifica-se o desvio (RMSD) em relação aos dados padrão da molécula. O ideal é que o desvio seja o menor possível.

Após a preparação dos dados estar concluída é que se inicia a acoplamento no Autodock. O software pode ser baixado no próprio site do desenvolvedor (AUTODOCK, 2015), estando disponíveis versões binárias e também o código fonte do programa. Com o autodock instalado, o programa é rodado, efetuando o acoplamento das moléculas e, ao final, é feito o seu ranqueamento. Esta é a etapa de maior custo computacional e onde este trabalho visa atuar para reduzir o tempo de execução do processo de Triagem Virtual.

Ao fim de todo o processo, é gerado o ranking das moléculas com melhor acoplamento, sendo estas potenciais candidatas para testes em laboratório.

O Algoritmo 2 exibe as etapas de acoplamento executadas pelo Autodock para a geração dos resultados. Este pseudoalgoritmo foi utilizado como base para as implementações posteriormente feitas e discutidas no Capítulo 6.

Algoritmo 2 Pseudo-Código do Autodock.

```
1: Lê o arquivo de configurações
2: Efetua a preparação dos dados antes de iniciar as iterações (chamadas para snorm())
3: enquanto o número de iterações não for alcançado faça
4:     Inicia a execução da simulação de interação com o Algoritmo Genético Lamarckiano
5:     enquanto todos os indivíduos não tiverem sido analisados faça

6:         Inicia a Busca Global
7:             Efetua a seleção
8:             Efetua o cruzamento
9:             Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
   operator())
10:            Efetua a mutação
11:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
   operator())
12:            Seleciona os melhores indivíduos

13:        Inicia a Busca Local (chamada para RealVector::clone() const e genPh())
14:            Efetua a seleção
15:            Efetua o cruzamento
16:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
   operator())
17:            Efetua a mutação
18:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
   operator())
19:            Seleciona os melhores indivíduos

20:        Analisa os resultados do acoplamento (chamadas para torsion() e qtransform())
21:    fim enquanto
22: fim enquanto
23: Finaliza a execução e disponibiliza os resultados.
```

Inicialmente, o programa lê os arquivos de configuração e faz a preparação dos dados.

Com base no número de iterações definido no arquivo de configuracção, são executados os laços principais do programa.

Na linha 4, é iniciada a simulação e na linha 5 é iniciado o laço de avaliação dos indivíduos do algoritmo genético. Na linha 6 é iniciada a busca global onde são executadas as etapas de seleção, cruzamento e cálculos de energia e interpolação. Por fim, os indivíduos com melhores resultados são selecionados.

Com esta etapa concluída, é iniciada a busca local na linha 13 onde são executados os mesmos passos da busca global, selecionando os melhores indivíduos ao final do processo.

Ao fim das duas etapas, os resultados do acoplamento são avaliados e relatórios finais disponibilizados para os usuários.

Capítulo

4

PLATAFORMAS DE EXECUÇÃO PARALELAS

Neste capítulo são discutidas em mais detalhes as plataformas de execução utilizadas no desenevolvimento deste trabalho. Plataformas de execução paralela permitem que determinados problemas computacionais sejam executados em paralelo, dividindo-os em partes menores e fazendo com que sua conclusão seja mais rápida. Por exemplo, considerando um programa que rode em núcleo de processamento em um tempo **n**, caso a execução deste mesmo programa seja distribuída entre quatro núcleos de processamento diferentes e executado em paralelo, o mesmo problema será executado em um tempo **n/4**.

Apesar dos benefícios, a utilização de plataformas de execução paralela apresenta novos desafios. Pode-se citar entre esses, o gerenciamento da comunicação e sincronização entre diferentes tarefas e problemas como as condições de corrida, onde duas subtarefas diferentes tentam fazer uso simultâneo de um recurso compartilhado.

Entre as plataformas disponíveis, pode-se destacar as arquiteturas *multicore*, onde uma mesma máquina possui mais de um núcleo de processamento de propósito geral (CPU) e as GPUs, hardwares especializados que inicialmente eram usados para aplicações gráficas e que, posteriormente, passaram a também a ser usadas para aplicações científicas.

4.1 ARQUITETURAS MULTICORE E BIBLIOTECA OPENMP

Arquiteturas *multicore* já são uma realidade há alguns anos. Grande parte das arquiteturas *multicore* foram desenvolvidas entre 1975 e o ano 2000. Em 2001 foi disponibilizado o primeiro processador de propósito geral com múltiplos núcleos de processamento, o IBM POWER4 (VAJDA, 2011). A partir deste momento, diversas outras empresas lançaram produtos semelhantes e este tipo de tecnologia tornou-se comum.

A definição de uma arquitetura *multicore* reside em três princípios básicos, sendo estes: existem múltiplos núcleos de processamento; existe uma forma de comunicação entre estes núcleos de processamento; e deve existir uma maneira destes núcleos se comunicarem com periféricos externos (VAJDA, 2011).

Apesar de todos os benefícios proporcionados pelas arquiteturas *multicore*, alguns problemas podem acontecer, entre estes o acesso concorrente à memória global. Geralmente as arquiteturas *multicore* são homogêneas, onde todos os núcleos de processamento possuem o mesmo desempenho e conjunto de instruções. Este tipo de arquitetura é mais simples do ponto de vista programático, permitindo que as tarefas sejam programaticamente distribuídas entre os núcleos de processamento disponíveis. Porém, arquiteturas mais modernas já possuem características heterogêneas, onde os núcleos podem ter conjunto de instruções e desempenho diferenciado entre eles.

Para tomar proveito das vantagens oferecidas por arquiteturas *multicore*, o OpenMP (OPENMP, 2016) é uma ferramenta muito útil. Ela é uma especificação de um conjunto de diretivas para compiladores, bibliotecas e variáveis de ambiente que podem ser implementadas para utilizar paralelismo em programas desenvolvidos utilizando as linguagens FORTRAN e C/C++.

O OpenMP permite que o desenvolvedor implemente paralelismo em seu sistema de forma relativamente simples, utilizando anotações específicas no código para definir as seções que devem ser paralelizadas. Geralmente, um programa não pode ser completamente paralelizado, por isso, o desenvolvedor da aplicação que deseje utilizar os recursos providos pelo OpenMP deve especificar as seções do programa onde o paralelismo será executado.

Por exemplo, geralmente os desenvolvedores optam por paralelizar laços dentro do código, onde, a depender da quantidade de núcleos disponíveis, o programa pode executar diversas iterações de forma simultânea na máquina. Porém, isto pode ocasionar problemas quando existe dependência entre os dados manipulados dentro das seções paralelizadas do código.

Para auxiliar a resolução deste tipo de problema, a especificação do OpenMP define alguns instrumentos de controle e sincronização que permitem ao desenvolvedor controlar a execução das seções paralelas do programa, reduzindo dessa forma a possibilidade de eventuais erros de execução e inconsistências na aplicação.

O OpenMP disponibiliza algumas instruções que permitem ao desenvolvedor controlar a execução do programa. Por exemplo, a instrução `#pragma omp parallel for` define que o for logo abaixo da instrução deverá ser executado de forma paralela nos núcleos paralelos disponíveis na máquina.

Também é possível definir a quantidade de *threads* que serão executadas na seção paralela do código através da instrução `num_threads(n)`, onde n é o total de *threads* que se deseja criar. A instrução `schedule(K)` define o tipo agendamento que as *threads* executarão, onde K pode ter os valores *static* ou *dynamic*. A principal diferença entre estes dois tipos de agendamento é que no tipo *static* a cada *thread* é dada uma porção do código para ser executado. No agendamento com *dynamic*, o OpenMP define uma ou mais iterações para cada *thread* e, quando a *thread* finaliza o trabalho, recebe um novo conjunto de iterações para executar, até sua finalização.

Outra possibilidade importante é a definição do escopo das variáveis envolvidas nos trechos paralelos através das instruções *private* e *shared*, entre outras. A cláusula *private* define que a variável tem uma cópia individual dentro de cada *thread*, enquanto que a instrução *shared* define que esta variável será compartilhada por todas as *threads* em

execução.

O algoritmo 3 ilustra um exemplo de uso da biblioteca OpenMP escrito na linguagem de programação C.

Algoritmo 3 Exemplo de Código OpenMP

```
1: const int size ← 256;
2: double sinTable[size];
3: int n;
4: int k ← 1;
5: #pragma omp parallel for num_threads(4) shared(n) private(k)
6: for (n = 0; n < size; n++) {
7:       sinTable[n] ← sin(2 * M_PI * n / size);
8: }
```

Na linha 5 do código apresentado no Algoritmo 3 é mostrada a instrução do OpenMP que indica que o trecho da instrução *for* deverá executar de forma paralela em quatro núcleos de processamento. Além disso, as *threads* compartilharão a variável `n` e cada *thread* individual terá uma cópia própria da variável `k`.

Caso a instrução da linha 5 seja suprimida, o código irá executar sequencialmente. Para que a execução paralela seja feita no código, é necessário que o programa seja compilado com a diretiva `-fopenmp`, caso contrário, a instrução OpenMP será ignorada e o código será executado sequencialmente.

4.2 UNIDADES DE PROCESSAMENTO GRÁFICO (GPUS) E CUDA

Unidade de Processamento Gráfico - *Graphic Processing Unit* (GPU) é um tipo de microprocessador dedicado ao processamento gráfico de aplicações dos mais variados tipos, como jogos eletrônicos e tratamento de vídeos.

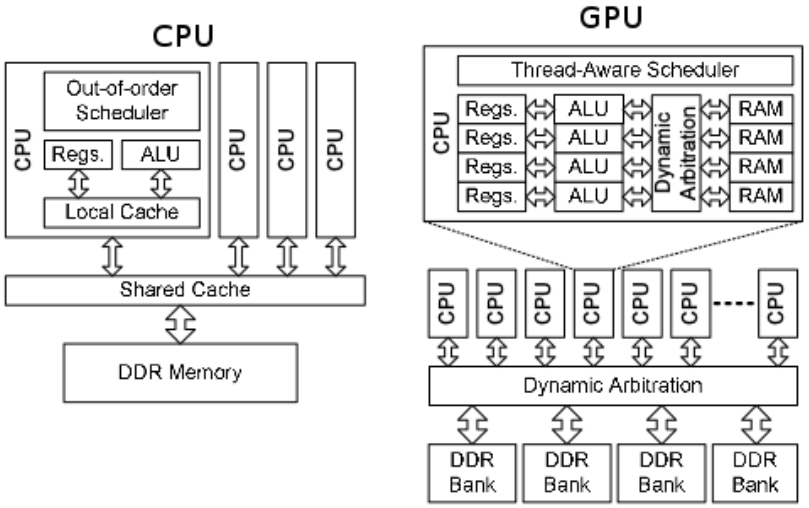

Figura 4.1 Diferença entre Arquiteturas CPU e GPU (THOMAS; HOWES; LUK, 2009)

Em comparação as CPUs mais modernas, como por exemplo os processadores da linha Intel Core i7 (INTEL, 2016), que usualmente possuem entre 4 e 8 núcleos de processamento de propósito geral, uma GPU pode ter até milhares de núcleos individuais especializados, propiciando um poder paralelo muito grande. A Figura 4.1 ilustra bem esta diferença.

Em geral as arquiteturas *multicore* operam da mesma forma que arquiteturas com um único processador, utilizando memória compartilhada para comunicação. Cada *core* executa uma *thread* por vez e possui diversos registradores, uma ALU (*Arithmetic Logical Units*) além de outros componentes.

A principal ideia da GPU é dedicar a maior quantidade de espaço possível para as ALUs, retirando para isso mecanismos existentes nas CPUs, como mecanismos de agendamento de processos e sistemas de cache. Com isso, uma GPU consegue executar diversas *threads* simultâneamente.

Devido à sua grande capacidade de processamento paralelo, as GPUs vem sendo extensamente utilizadas para pesquisas científicas que buscam aprimorar o desempenho das suas aplicações.

Ainda assim, programar tarefas específicas para uma GPU é uma tarefa complicada e dispendiosa. Por conta disso, a NVIDIA, grande fabricante de placas gráficas criou a CUDA, uma plataforma de computação paralela e modelo de programação que permite que os interessados desenvolvam programas que possam utilizar todo o poder de processamento de uma GPU de forma mais simples.

Diversas aplicações já utilizam o poder de processamento das GPUs aliados ao CUDA em sua execução, indo desde aplicações de análise de tráfego aéreo até a busca por tratamentos de doenças (NVIDIA, 2015).

CUDA está disponível para uso nas linguages C/C++ e FORTRAN. No CUDA, a CPU funciona como um *host*, controlando o fluxo da aplicação, enviando tarefas para a GPU, copiando dados de e para a GPU entre outras atribuições. A porção de código que executa dentro da GPU é chamada de *kernel*.

Antes de tudo, é necessário definir a terminologia usualmente utilizada em CUDA. O *host* refere-se a CPU e memória principal presente na máquina, enquanto o *device* refere-se a GPU e sua memória.

O exemplo de código presente no Algoritmo 4 apresenta um código básico que utiliza CUDA para ser executado na GPU.

Entre as linhas 1 e 3, está uma função definida como *kernel*. Esta é a porção de código que irá executar na GPU. Dentro da função main (linhas 5 a 14), é definido o programa principal que irá executar na CPU e coordenar a execução do código na GPU. Na linha 8 é feita alocação de memória e a linha 9 chama a função que executará na GPU. A linha 10, que contém a *flag* `cudaMemcpyDeviceToHost`, indica que será feita a cópia de memória do *device* (GPU) para o *host* (CPU). A linha 11 exibe o resultado da execução do código na GPU e, por fim, a linha 12 libera a memória da GPU. Apesar de simples, o trecho demonstra o uso básico de CUDA para executar código em GPUs.

Conforme visto no exemplo de código presente no Algoritmo 4, o *kernel* é a porção de código que irá executar na GPU, sendo chamado a partir do *host* da aplicação. Para que o compilador possa identificar qual código irá ser executado na GPU, é necessária

Algoritmo 4 Exemplo de Código CUDA executado em GPU

```
 1: __global__ void add( int a, int b, int *c ) {
 2:     *c = a + b;
 3: }
 4:
 5: int main( void ) {
 6:     int c;
 7:     int *dev_c;
 8:     cudaMalloc( (void**)&dev_c, sizeof(int) );
 9:     add<<<1,1>>>( 2, 7, dev_c );
10:     cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost );
11:     printf( "2 + 7 = %d", c );
12:     cudaFree( dev_c );
13:     return 0;
14: }
```

a utilização de qualificadores. No caso do exemplo citado, foi utilizado o qualificador `__global__`, definindo que este *kernel* pode ser chamado a partir do *host*. Também existe o qualificador `__device__` para definir *kernels* que só podem ser chamados a partir de outros *kernels* do programa.

Além disso, outro fator importante a ser frisado é que o *host* e o *device* são entidades distintas dentro do sistema e, portanto, possuem recursos próprios como memória, por exemplo. Por conta deste fato, CUDA disponibiliza algumas funções que permitem alocação de memória e gerenciamento desta entre o *host* e o *device*. As funções disponibilizadas são: `cudaMalloc()`, que aloca memória no *device*; `cudaMemCpy()`, que permite que dados sejam transitados entre o *host* e o *device*; e `cudaFree()`, que libera memória previamente alocada no *device*. Caso seja necessário declarar uma variável dentro do *kernel* que deve ser compartilhada entre todas as *threads*, esta deve ser declarada com o qualificador `__shared__`.

Outro conceito importante quando se faz uso CUDA é a utilização de *blocks* e *threads*. No exemplo presente no Algoritmo 4, na linha 9 é feita a chamada ao *kernel* presente no *device*. Os valores 1 presentes entre os sinais de maior e menor na chamada a função são, respectivamente, a quantidade de *blocks* e *threads* chamados para execução do *kernel* citado. A multiplicação do número de *blocks* pelo número de *threads* representa, efetivamente, o número de *threads* paralelas que irão executar aquele *kernel* no *device*. A junção de *blocks* e *threads* é definida como um *grid*.

Durante o processamento no *kernel*, para que seja possível acessar os dados desejados, CUDA disponibiliza algumas variáveis de acesso aos *blocks*, *threads*, sendo estes, respectivamente, `blockIdx` e `threadIdx`. Além destas, também são disponibilizadas as variáveis `blockDim`, que contém a quantidade de *threads* em um *block* e `gridDim`, que contém o número de *blocks* em um deterinado *grid*.

Conforme visto, as plataformas de execução paralela dispõem de recursos que permitem aos desenvolvedores tomar proveito das vantagens promovidas pela paralelização das aplicações. O uso de arquiteturas *multicore* através da especificação OpenMP e GPUs através do CUDA podem ser úteis e efetivamente atuar no aumento de desempenho da execução de tarefas computacionais diversas.

Capítulo 5

TRABALHOS RELACIONADOS

Neste capítulo são discutidos alguns trabalhos relacionados com o tema proposto por esta dissertação. Usualmente, para se medir o ganho de tempo proporcionado entre dois sistemas que buscam resolver o mesmo problema, utiliza-se a métrica do *speedup*. Nos trabalhos citados abaixo, os autores utilizam esta métrica com o objetivo de comparar as soluções demonstradas com abordagens anteriores.

5.1 MPI, OPENMP E INTEL XEON PHI

Em (PRAKHOV; CHERNORUDSKIY; GAINULLIN, 2010) é proposta uma solução de paralelização do Autodock para clusters no ambiente Microsoft Windows utilizando MPI chamada de VSDocker, sendo esta a primeira implementação de uma ferramenta de Triagem Virtual para o sistema operacional Microsoft Windows. Para fazer os testes eles utilizaram os mesmos parâmetros e base de dados utilizados no DOVIS, citado em (JIANG et al., 2008), atingindo um desempenho de 420 ligantes/CPU/dia.

No trabalho de (NORGAN et al., 2011) o Autodock é paralelizado para OpenMP e MPI, utilizando as duas soluções em conjunto buscando uma maior desempenho. Os testes neste trabalho também foram feitos com a base de dados ZINC. As arquiteturas utilizadas foram de grande porte, mais especificamente, IBM POWER7 e IBM BlueGene/P. Os autores afirmam chegar a um *speedup* máximo de 22x.

Por fim, em (CHENG et al., 2015) os autores fazem a paralelização do Autodock utilizando OpenMP e Intel Xeon Phi, comparado os seus resultados. Inicialmente uma versão em OpenMP foi desenvolvida e após isso, o código foi portado para a Intel Xeon Phi. O *speedup* médio das duas implementações gira em torno de 12x enquanto que, a medida que o número de iterações aumenta, *speedups* da ordem de 18x são alcançados.

5.2 GPUS E FPGA

Em (KANNAN; GANJI, 2010) os autores fazem uma migração do Autodock para GPUs NVIDIA, descrevendo a estratégia utilizada por eles para isso. Neste trabalho os autores

alcançaram um *speedup* de cerca de 50x na função de avaliação e entre 10x e 47x no algoritmo principal do Autodock. Inicialmente, duas propostas de imeplementação foram discutidas. Na primeira, definida como "*Per Thread Parallel Breakdown Approach*", uma *thread* GPU é dedicada exclusivamente para o cálculo de energia de um indivíduo. Na segunda, definida como "*Per Block Parallel Breakdown Approach*", uma *thread block* na GPU é dedicada ao cálculo da energia de um indivíduo. Por fim, a estratégia escolhida foi uma terceira, definida como "*PerBlockCached*", onde todas as coordenadas atômicas das moléculas são mantidas numa memória compartilhada, eliminando o *overhead* de leitura necessário a cada execução.

Em (PECHAN; FEHER; BERCES, 2010) é proposta a adaptação do Autodock para a arquitetura FPGA (XILINX, 2016). Neste, eles atingem um *speedup* médio de 24x em relação a execução sequencial em uma CPU de 3.2 GHZ. Os testes foram feitos com 60 receptores-ligantes e 100 iterações nas duas arquiteturas. O tempo médio de execução de cada execução foi de 79.4 segundos no Autodock sem modificação e 3.1 segundos na versão rodando na placa FPGA.

Em (PECHAN; FEHER, 2011) eles expandem o trabalho feito em 2010 (PECHAN; FEHER; BERCES, 2010) propondo uma comparação do Autodock adaptado para GPU's NVIDIA e FPGA. Eles também testaram as duas abordagens contra duas proteínas diferentes do Protein Database Bank (PDB), com códigos de busca **1HVR** e **2CPP**. Neste trabalho eles alcaçam um *speedup* de 35x para a proteína **1HVR** e 12x para a **2CPP** na FPGA, utilizando os mesmos parâmetros do trabalho anterior. Nas implementações em GPUs, duas diferentes placas foram usadas. Uma NVIDIA GT220 e uma NVIDIA GTX260. Na primeira os *speedups* foram de 12x para a proteína **1HVR** e 17x para a **2CPP**. Na segunda placa, os *speedups* foram de 57x e 75x para **1HVR** e **2CPP**, respectivamente.

Em (OUYANG; KWOH, 2012) o Autodock é paralelizado para GPUs utilizando nos testes a base de dados fornecida pelo próprio Autodock. Eles utilizaram duas abordagens de implementação. Na primeira, denominada de *Offload Parallel Genetic Algorithm*, o host CPU é utilizado para controlar todo o processo evolucionário do algoritmo enquanto que as tarefas mais dispendiosas são executadas na GPU, alcançando ao final dos testes *speedups* na ordem de 100x. Na segunda abordagem, denominada de *In-kernel Multideme Parallel Genetic Algorithm with Asynchronous Migration* a população do Algoritmo Genético é subdivida em subpopulações menores (demes) e todo o processo evolucionário do algoritmo é executado na GPU. Nesta segunda abordagem, são alcançados *speedups* médios da ordem de 90x. Para todos os testes, 100 iterações são utilizadas.

5.3 OUTRAS ABORDAGENS

No trabalho de (JIANG et al., 2007) é criado o DOVIS, um *pipeline* paralelizado em clusters para Triagem Virtual que utiliza, entre outras ferramentas, o Autodock. Neste trabalho, após a fase de implementações, os autores testaram o pipeline na base de dados ZINC (ZINC, 2015). A base ZINC possuia, na época do estudo, cerca de 2,3 milhões de compostos (versão 6). O objetivo principal deste teste era analisar quantos ligantes poderiam ser selecionados por dia. Utilizando 128 CPUs, a implementação alcançou a

marca de 700 ligantes/CPU/dia. Após esses testes toda a base de dados do ZINC foi utilizada na busca de ligantes que interagissem com a cadeia de proteínas da Ricina. Utilizando 256 processadores, todo o processo levou cerca de 77.000 horas de CPU para ser completado.

Em (KHODADE et al., 2007) foi empregado o Autodock na versão 3.0.5 paralelizado para uso com MPI. Neste estudo os autores conseguiram reduzir o tempo de acoplamento de uma proteína do HIV de 81 minutos em um único processador para cerca de 1 minuto em um cluster de 96 processadores. Neste trabalho foi utilizado o algoritmo LGA com uma população de 50 indivíduos e um número de iterações de 100 e 200. Os autores alcançaram um *speedup* máximo de 85x.

Em (JIANG et al., 2008) os autores criam o DOVIS 2.0, atualizando o trabalho feito em (JIANG et al., 2007) alterando-o para a versão 4.0 do Autodock. Além disso, eles também refizeram o processo de paralelização desenvolvido no trabalho anterior, alcançando uma redução no número de operações com arquivos em mais de 95%. Neste trabalho os autores alcançam o processamento de 670 ligantes/CPU/dia.

No trabalho de (ATILGAN; HU, 2010) os autores alteram o código-fonte do Autodock, substituindo o algoritmo padrão por um novo, tendo como objetivo buscar um melhor desempenho. Segundo eles, os algoritmos padrão do Autodock possuem limitações e, devido a natureza estocástica dos mesmos, necessitam de diversas execuções para se alcançar bons resultados. Por conta disso, eles propõem a criação de um novo algoritmo buscando reduzir os tempos de execução e melhorar os resultados finais encontrados.

A Tabela 5.1 resume os Trabalhos apresentados neste capítulo focando-se nos resultados de desempenho encontrados.

Tabela 5.1 Comparativo de Desempenho entre Trabalhos Relacionados

Trabalho	Speedup/Desempenho	Referências
VSDocker: A tool for parallel high-throughput virtual screening using AutoDock on Windows-based computer clusters	Processamento de 420 ligantes por CPU por dia	(PRAKHOV; CHERNORUDSKIY; GAINULLIN, 2010)
Multilevel parallelization of autodock 4.2 (NORGAN et al., 2011)	Speedup de 22x	(NORGAN et al., 2011)
mD3DOCKxb: A Deep Parallel Optimized Software for Molecular Docking with Intel Xeon Phi Coprocessors (CHENG et al., 2015)	Speedup médio de 12x mas foram alcançados picos de 18x	(CHENG et al., 2015)
Porting Autodock to CUDA	Speedups variaram entre 10x a 47x e a função de avaliação chegou a um speedup de 50x	(KANNAN; GANJI, 2010)
FPGA-based acceleration of the AutoDock molecular docking software	Speedup médio de 24x	(PECHAN; FEHER; BERCES, 2010)
Molecular Docking on FPGA and GPU Platforms	Speedups entre 12x e 35x na FPGA e entre 57x e 75x na GPU	(PECHAN; FEHER, 2011)
GPU Accelerated Molecular Docking with Parallel Genetic Algorithm	Speedup máximo de 90x, mas exige um alto número de iterações para alcançar este resultado	(OUYANG; KWOH, 2012)
DOVIS: A Tool for High-Throughput Virtual Screening	Processamento de 700 ligantes por CPU por dia	(JIANG et al., 2007)
Parallel implementation of AutoDock	Speedup máximo de 85x	(KHODADE et al., 2007)
DOVIS 2.0: an efficient and easy to use parallel virtual screening tool based on AutoDock 4.0	Processamento de 670 ligantes por CPU por dia	(JIANG et al., 2008)
Efficient protein-ligand docking using sustainable evolutionary algorithms	Não fica claro o desempenho alcançado pois o trabalho é focado na comparação entre valores finais de energia.	(ATILGAN; HU, 2010)

Conforme exposto, existem algumas abordagens que exploram a paralelização do Autodock utilizando diferentes plataformas de execução paralela como OpenMP, MPI, GPUs, FPGA e placas Intel Xeon Phi. Reduzindo-se o escopo ao OpenMP e GPUs, percebe-se que não existem comparações diretas entre os dois modelos e também não foram encontradas soluções que utilizam as duas em conjunto. Além disso, nos trabalhos pesquisados, não ficam claros os ambientes utilizados para desenvolver e validar os trabalhos, dificultando sua reprodução e posterior disponibilização para os usuários finais,

neste caso, os pesquisadores da área de fármacos.

Capítulo 6

TRABALHO PROPOSTO

Neste capítulo é apresentado o o objetivo do trabalho proposto, a metodologia empregada durante sua execução, a base de dados utilizada e as implementações desenvolvidas no decorrer deste trabalho.

6.1 OBJETIVO E METODOLOGIA

O principal objetivo deste trabalho foi adaptar e avaliar rotinas de Triagem Virtual em plataformas de computação paralela. Para alcançar este propósito, este trabalho foi iniciado com a revisão bibliográfica da área de Triagem Virtual incluindo nesta o estudo dos métodos de planejamento de fármacos, bases de dados de moléculas disponíveis para uso e algumas ferramentas. Neste momento foi definido que a ferramenta de Triagem Virtual a ser paralelizada seria o Autodock.

Além disso, também foram estudadas plataformas de computação paralela. Dentro deste tópico, foi dada maior ênfase às arquiteturas *multicore* e GPUs, que são o foco deste trabalho. Também foram iniciados os estudos de conceitos e programação *multicore* com OpenMP. Também foram feitos estudos dos conceitos e programação em GPUs utilizando CUDA.

Posteriormente foram iniciados os testes com o Autodock para entendimento do seu fluxo de execução e, finalizada esta etapa, foram feitos testes com o *dataset* escolhido. Este *dataset* é discutido em mais detalhes na subseção 6.2.

Com as fases anteriores concluídas, foram iniciadas as implementações das modificações no Autodock objetivando tomar proveito das vantagens proporcionadas pelas plataformas de execução paralelas.

Para as execuções *multicore* utilizando OpenMP foram utilizadas 24 *threads* simultâneas em todas as implementações. Todas as implementações para GPUs com CUDA foram executadas utilizando-se somente uma GPU.

Para este trabalho, três diferentes versões foram concebidas e implementadas. A primeira é uma versão que utiliza arquiteturas *multicore* através do OpenMP. A segunda implementação utiliza GPUs com CUDA para paralelizar a execução do Autodock. Por

fim, foi desenvolvida uma versão do Autodock que tomasse proveito das vantagens proporcionadas pelas arquituras *multicore* e pelas GPUs. Tomando como base as duas implementações previamente desenvolvidas, esta nova versão híbrida foi desenvolvida tendo como objetivo aumentar ainda mais o desempenho do Autodock.

Por fim, as implementações foram executadas e tiveram seus resultados validados e testados entre si e com a execução sequencial do Autodock.

6.2 BASE DE DADOS UTILIZADA

Neste trabalho foram utilizado os dados presentes no NuBBE (NUBBE, 2016), que é uma base de produtos naturais mantido pelo Núcleo de Bioensaios, Biosíntese e Ecofisiologia de Produtos Naturais. Desta base de dados foi selecionado um conjunto de 83 moléculas para serem testadas contra a proteína alvo.

Como proteína alvo, foi escolhida a **1BZL** (PDB, 2016) presente na base de dados Protein Data Bank (PDB). O arquivo disponibilizado pela base de dados representa a estrutura cristalográfica da enzima tripanotiona redutase do *Trypanosoma cruzi*. O *Trypanosoma cruzi* é o agente transmissor da doença de Chagas. A proteína e o conjunto de moléculas da NuBBE foram escolhidos pois estudos com estes dados estavam sendo conduzidos no Laboratório de Bioinformática e Modelagem Molecular da Faculdade de Farmácia da Universidade Federal da Bahia.

De acordo com depoimentos dos responsáveis pelos estudos, a triagem deste total de moléculas levava em torno de 24 horas para ser concluída, tornando-as boas candidatas para servirem como experimento para o processo de paralelização da Triagem Virtual.

6.3 IMPLEMENTAÇÕES

Até o momento, grande parte das abordagens anteriores que buscam melhorar o desempenho do Autodock são baseadas na utilização de bibliotecas como OpenMP e MPI. No Capítulo 5 são listados os trabalhos relacionados que abordam estas soluções. Nestes trabalhos são apresentadas diversas ferramentas e abordagens diferentes de paralelização do Autodock.

Em geral, algumas destas implementações são feitas em ambientes computacionais restritos e controlados. Por exemplo, no trabalho de (NORGAN et al., 2011), os autores fazem a adaptação do Autodock para OpenMP e MPI nas arquiteturas IBM BlueGene/P (IBM, 2016a) e IBM POWER7 Server (IBM, 2016b). Este fator dificulta a adoção destas ferramentas otimizadas no desenvolvimento rotineiro de novos fármacos, pois muitos pesquisadores não tem acesso a este tipo de infraestrutura.

Neste trabalho, o principal objetivo foi estudar as técnicas de paralelização de Triagem Virtual e como elas pode ser melhoradas. Pelas suas vantagens, expostas no Capítulo 3, o Autodock foi escolhido para ser paralelizado para arquiteturas *multicore* e GPUs. Além disso, também buscou-se abstrair dos usuários finais algumas tarefas como particionamento e distribuição dos dados necessários para executar o Autodock.

Para isso, o Autodock teve seu código adaptado para CUDA e OpenMP. Além destas, o Autodock também foi portado para uma arquitetura híbrida que utiliza conjuntamente

OpenMP e CUDA. Esta implementação foi definida como CUDAMP e teve como objetivo aumentar ainda mais o desempenho geral da aplicação.

6.3.1 Testes Iniciais

Antes de iniciar a adaptação do Autodock, algumas execuções foram feitas com a base de dados de testes com o objetivo de verificar a desempenho de tempo da aplicação. Para isso, o número de iterações foi sendo duplicado a cada execução partindo de uma iteração até 256 iterações, tendo seus respectivos tempos registrados. Estes tempos foram medidos utilizando a função da linguagem C `times()`. Esta mesma metodologia foi utilizadas para verificar as outras implementações.

Após isso, o software GNU gprof (GRAHAM; KESSLER; MCKUSICK, 1982) foi utilizado para determinar quais partes da aplicação consomem mais recursos da máquina. Estas execuções serviram como ponto de partida para guiar a adaptação do Autodock para as arquiteturas paralelas. O gprof determinou que as funções mais custosas foram a `eintcal()`, que calcula a energia interna de uma molécula específica e a `trilinterp()`, que calcula a interpolação trilinear. Cada uma delas consome, em média, 40% do tempo total de execução.

A Tabela 6.1 apresenta as dez primeiras funções que mais consomem recursos durante a execução do Autodock de acordo com o gprof.

Tabela 6.1 Funções que Mais Consomem Recursos Durante a Execução do Autodock

% Tempo	Tempo de execução cumulativo (segundos)	Tempo de execução da função (segundos)	Total de chamadas	Nome da função/instrução
38.61	57.51	57.51	25011902	eintcal()
31.47	104.39	46.88	25011902	trilinterp()
4.12	110.52	6.13	25011905	torsion()
3.32	115.47	4.95	152341404	snorm()
2.96	119.88	4.41	388981680	RealVector::clone() const
2.28	123.28	3.40	22815183	genPh()
1.96	126.20	2.92	25011885	Eval::operator() (Representation const* const*)
1.59	128.57	2.37	25011905	qtransform()
1.59	130.94	2.37	40334660	Phenotype::operator=(Phenotype const&)

É importante frisar que, devido a natureza estocástica do algoritmo genético, estes valores podem ter pequenas variações. Além disto, outro fator que pode influenciar na aferição destes dados são as características intrínsecas dos dados utilizados. De toda forma, mesmo após três execuções e análise visual dos dados gerados pelo gprof, as informações tendem a seguir a orientação apresentada na Tabela 6.1.

A implementação sequencial, apresentada no Algoritmo 2 foi tomada como base para se criar as implementações paralelas.

6.3.2 Implementação em OpenMP

Depois dos testes iniciais, a aplicação foi adaptada para utilizar OpenMP e, seguindo a mesma metodologia aplicada nos testes iniciais, teve seus tempos devidamente medidos e registrados. Estas execuções com OpenMP mostraram uma melhoria de desempenho em relação as execuções sequenciais.

De acordo com (NORGAN et al., 2011) e posterior investigação no código-fonte, o Autodock utiliza um gerador de números aleatórios, *Randon Number Generator* (RNG) que utiliza o algoritmo determinístico IGNLGI, que faz parte da biblioteca RANLIB presente na linguagem C. Com isso, é possível gerar números aleatórios baseados no *timestamp* do sistema. As *threads* em OpenMP são criadas simultaneamente e, sem a utilização de um RNG, todas as *threads* possuirão o mesmo número de identificação. Para resolver este problema, o identificador da *thread* é incluído no momento da geração do RNG, permitindo que um número único seja garantido.

Após isso, o programa foi paralelizado no laço onde as iterações são executadas. O número de *threads* definido para a execução foi de 24. Além disso, foi necessário adicionar alguns pontos críticos no código com a notação específica do OpenMP com o objetivo de evitar interferência entre as *threads*.

O Algoritmo 5 apresenta o pseudo-código da implementação feita em OpenMP para adaptar o Autodock para a arquitetura *multicore*.

Algoritmo 5 Pseudo-Código do Autodock Portado para Multicore com OpenMP

```
1: Lê o arquivo de configurações
2: Efetua a preparação dos dados antes de iniciar as iterações (chamadas para snorm())
3: #pragma omp parallel for schedule(static) num_threads(n)
4: enquanto o número de iterações não for alcançado faça
5:     Inicia a execução da simulação de ligação com o Algoritmo Genético Lamarckiano
6:     enquanto todos os indivíduos não tiverem sido analisados faça

7:         Inicia a Busca Global
8:             Efetua a seleção
9:             Efetua o cruzamento
10:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
11:            Efetua a mutação
12:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
13:            Seleciona os melhores indivíduos

14:        Inicia a Busca Local (chamada para RealVector::clone() const e genPh())
15:            Efetua a seleção
16:            Efetua o cruzamento
17:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
18:            Efetua a mutação
19:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
20:            Seleciona os melhores indivíduos

21:        Analisa os resultados do acoplamento (chamadas para torsion() e qtransform())
22:    fim enquanto
23: fim enquanto
24: Finaliza a execução e disponibiliza os resultados.
```

6.3.3 Implementação em CUDA

Na implementação em CUDA, as funções mais custosas do programa, `eintcal()` e `trilinterp()`, foram implementadas como *kernels* CUDA. A função `eintcal()` efetua o cálculo de energia de um ligante. A função `trilinterp()` efetua cálculos de interpolação do ligante. Estas duas funções são as que consomem mais tempo e rercursos de processamento de acordo com o relatório do gprof apresentado na Tabela 6.1. Neste trabalho, uma GPU NVIDIA foi utilizada para executar os experimentos.

Antes da execução do processo de acoplamento, memória é alocada na GPU para uso. Após isso, o programa continua a execução, e, durante as fases de busca global e busca local os *kernels* são executados. O tamanho de cada bloco de *threads* foi fixado em 128. O tamanho da grid varia de acordo ao tamanho da população determinada para a execução do LGA. Este número é calculado dividindo o tamanho do bloco (128) pelo tamanho da população, arredondando o resultado final.

O pseudo-código apresentado no Algoritmo 6 apresenta a implementação feita em CUDA para adaptar o Autodock para GPUs.

Algoritmo 6 Pseudo-Código do Autodock Portado para GPUs com CUDA

```
1: Lê o arquivo de configurações
2: Efetua a preparação dos dados antes de iniciar as iterações (chamadas para snorm())
3: Efetua a alocação de memória na GPU
4: enquanto o número de iterações não for alcançado faça
5:     Inicia a execução da simulação de ligação com o Algoritmo Genético Lamarckiano
6:     enquanto todos os indivíduos não tiverem sido analisados faça
7:         Inicia a Busca Global
8:             Efetua a seleção
9:             Efetua o cruzamento
10:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
11:            Efetua a mutação
12:            //cálculo de energia e interpolação da molécula implementados como kernels CUDA
13:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
14:            Seleciona os melhores indivíduos

15:        Inicia a Busca Local (chamada para RealVector::clone() const e genPh())
16:            Efetua a seleção
17:            Efetua o cruzamento
18:            //cálculo de energia e interpolação da molécula implementados como kernels CUDA
19:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
20:            Efetua a mutação
21:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
22:            Seleciona os melhores indivíduos

23:        Analisa os resultados do acoplamento (chamadas para torsion() e qtransform())
24:    fim enquanto
25: fim enquanto
26: Finaliza a execução e disponibiliza os resultados
```

6.3.4 Implementação Híbrida - multicore + GPU

Por fim, uma versão híbrida do Autodock foi feita. Esta implementação utiliza conjuntamente OpenMP e CUDA objetivando uma melhoria de desempenho ao se executar as duas abordagens anteriores conjuntamente.

Em relação a implementação, o código previamente desenvolvido foi utilizado como base para esta nova versão.

A versão CUDA desenvolvida anteriormente foi utilizada como base e, a partir dela, a versão em OpenMP foi replicada nesta nova versão. A partir dela, foram adicionadas as instruções OpenMP onde existe o laço de iterações. A partir deste ponto, seguindo a implementação CUDA feita previamente, as chamadas para as funções `eintcal()` e `trilinterp` foram substituídas por *kernels*.

Os mesmos parâmetros e configurações utilizados nas duas abordagens anteriores foram reutilizados nesta nova versão.

O pseudo-código apresentado no Algoritmo 7 apresenta a implementação híbrida que combina as implementações *multicore* e GPU.

Algoritmo 7 Pseudo-Código do Autodock Portado para Multicore com OpenMP e GPUs com CUDA

```
1: Lê o arquivo de configurações
2: Efetua a preparação dos dados antes de iniciar as iterações (chamadas para snorm())
3: Efetua a alocação de memória na GPU
4: #pragma omp parallel for schedule(static) num_threads(n)
5: enquanto o número de iterações não for alcançado faça
6:     Inicia a execução da simulação de ligação com o Algoritmo Genético Lamarckiano
7:     enquanto todos os indivíduos não tiverem sido analisados faça
8:         Inicia a Busca Global
9:             Efetua a seleção
10:            Efetua o cruzamento
11:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
12:            Efetua a mutação
13:            //cálculo de energia e interpolação da molécula implementados como kernels CUDA
14:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
15:            Seleciona os melhores indivíduos

16:        Inicia a Busca Local (chamada para RealVector::clone() const e genPh())
17:            Efetua a seleção
18:            Efetua o cruzamento
19:            //cálculo de energia e interpolação da molécula implementados como kernels CUDA
20:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
21:            Efetua a mutação
22:            Calcula energia e interpolação da molécula (chamadas para eintcal(), trilinterp() e
    operator())
23:            Seleciona os melhores indivíduos

24:        Analisa os resultados do acoplamento (chamadas para torsion() e qtransform())
25:    fim enquanto
26: fim enquanto
27: Finaliza a execução e disponibiliza os resultados
```

Capítulo 7

RESULTADOS

Este capítulo apresenta os resultados dos experimentos feitos para validar as implementações criadas no capítulo 6.

7.1 AMBIENTE DE EXECUÇÃO DOS TESTES

O ambiente de execução dos testes foi o seguinte:

> Ambiente composto de dois Intel Xeon com 2.26 GHz e 24 GB DDR3 de memória principal. Cada unidade é um processador de vinte e quatro núcleos com 12 MB de memória cache. A máquina contém ainda duas GPUs NVIDIA Tesla C2050 com 14 *stream multiprocessors* (SM) e 32 *stream processors* (SP) cada, totalizando 448 cores.

Para simplificar o entendimento dos gráficos e tabelas que exibem os resultados dos testes, a execução do Autodock sem alterações, utilizando somente uma *thread* é denominada de **Sequencial** enquanto o uso de várias *threads* através do uso do OpenMP, conforme definido no Capítulo 6, é denominado de "**CPU/Cores**". Como já explicitado anteriormente, neste trabalho foram utilizadas 24 *threads*.

O uso de uma GPU da NVIDIA para a implementação em CUDA é denominado de "**1GPU**". Os resultados da implementação híbrida, que utiliza OpenMP e GPU em conjunto, é definida como "**CPU/Cores+1GPU**".

7.2 BASE DE DADOS UTILIZADA

Conforme citado no Capítulo 6, para testar as implementações feitas a base de dados da NuBBE foi utilizada. Da base de dados disponibilizada por eles, foi selecionado um total de 83 moléculas e, dentre estas, uma molécula foi aleatoriamente escolhida para que os testes de desempenho fossem feitos. Como proteína alvo, foi selecionada na base de dados PDB a proteína com código de busca 1BZL, que representa a estrutura cristalográfica da enzima tripanotiona redutase do *Trypanosoma cruzi*.

7.3 METODOLOGIA DOS TESTES

Todos os testes foram executados no mesmo ambiente e com os mesmos parâmetros de execução. Para que se pudesse efetivamente medir a melhoria proporcionada pelas soluções paralelizadas, inicialmente foram feitos testes com o Autodock sem modificações e executando de maneira sequencial.

A partir da molécula selecionada, o número de iterações foi sendo duplicado a cada execução, partindo de uma iteração até o total de 256. A partir disso, todas as outras implementações (CPU/Cores, 1GPU e CPU/Cores+1GPU) foram testadas utilizando a mesma metodologia e tiveram seus tempos de execução registrados. A partir disso foi possível verificar o ganho de desempenho alcançados pelas implementações e verificar qual das soluções teve o melhor desempenho.

7.4 RESULTADOS - SPEEDUP

O ganho de desempenho, ou *speedup*, é uma métrica utilizada para medir o ganho de desempenho entre dois sistemas que buscam resolver o mesmo problema. No caso deste trabalho, o *speedup* refere-se ao ganho proporcionado pelas implementações que utilizam plataformas de execução paralela em relação a execução sequencial do Autodock. Para se calcular o *speedup* alcançado, divide-se o tempo da execução sequencial pelo tempo da execução paralelizada.

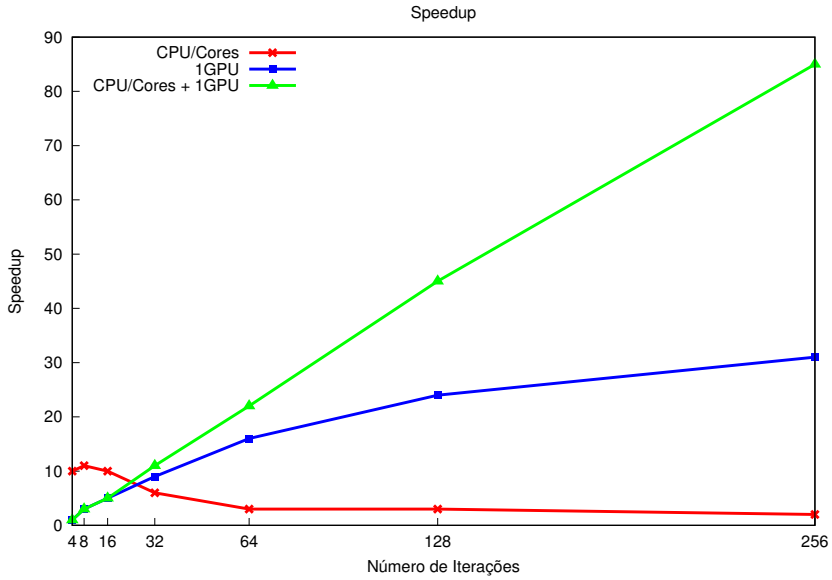

Figura 7.1 Speedup

Na figura 7.1, é possível notar que a implementação CPU/Cores alcançou uma boa redução total de tempo, alcançando um pico de *speedup* de até 10 vezes, porém, após cerca de 64 iterações cai até a faixa de 3 vezes, ficando estável a partir deste ponto. Ainda assim, isso é um ganho considerável, chegando a uma melhoria na ordem de 65%

em relação a implementação sequencial, conforme pode ser atestado na Figura 7.2 e na Tabela 7.1.

Ainda de acordo com a Figura 7.1, a implementação 1GPU foi bem mais rápida que a CPU/Cores, alcançando um *speedup* máximo de 28 vezes e um ganho de tempo da ordem de cerca de 96%.

Por fim, também é possível notar que a implementação híbrida, CPU/Cores+1GPU obteve os melhores resultados e sendo mais rápida que as duas soluções anteriores, alcançando um *speedup* de 85 vezes e diminuição no tempo total de cerca de 98,8%. Isto demonstra que a utilização combinada de arquiteturas *multicore* juntamente a GPUs pode trazer bons ganhos de desempenho quando comparado a utilização destas separadamente.

Apesar deste ganho substancial de velocidade no tempo de execução, foi notada uma redução na precisão dos resultados finais de energia encontrados ao final do processo de docagem nas implementações 1GPU e CPU/Cores+1GPU. De toda forma, devido a natureza estoástica do Algoritmo Genético, não é possível determinar se essa redução deveu-se ao fato da execução da Triagem Virtual ter sido paralelizada.

7.5 RESULTADOS - COMPARATIVO DE TEMPO

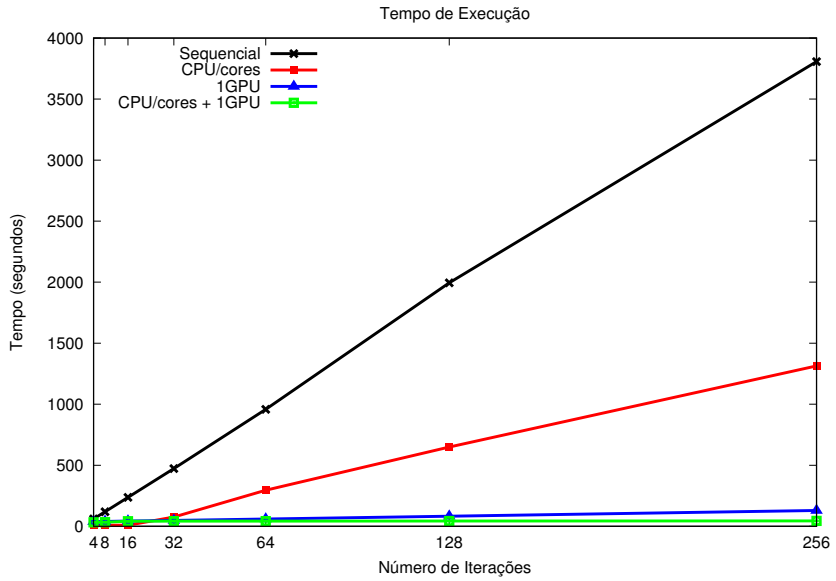

Figura 7.2 Comparativo de Tempo Entre Implementações Sequencial, CPU/Cores, 1GPU and CPU/Cores + 1GPU

A Figura 7.2 mostra o tempo das diferentes implementações. Novamente é possível notar uma boa redução de tempo em todas as implementações paralelas, mas notadamente, a implementação híbrida teve uma redução substancial, conforme pode ser notado no gráfico.

A Tabela 7.1 mostra os tempos de execução do Autodock para todas as implementações feitas para este trabalho. O tempo de cada iteração é mostrado, começando

com uma iteração e duplicando esta quantidade até o total de 256 iterações. Com esta abordagem, também podemos notar que, na execução sequencial, o tempo dobra sempre que o número de iterações é duplicado.

Outro fato notado é que, até 16 iterações, as implementações Sequencial e CPU/Cores são mais rápidas que as implementações 1GPU e CPU/Cores + 1GPU. Isto acontece porque antes da execução do processo de acoplamento efetivamente começar, é necessário alocar memória na GPU e preparar os dados para execução.

A partir de 32 iterações, as implementações que utilizam GPU (1GPU) e *multicore* + GPU (CPU/Cores+1GPU) passam a ter um tempo de execução menor que a solução CPU/Cores. Além disso, a partir de 32 de iterações, a solução híbrida (CPU/-Cores+1GPU) passa a ter um desempenho maior que a solução 1GPU.

Tabela 7.1 Comparação de Tempo de Execução das Implementações para Autodock (em segundos)

Iterações	**Sequencial**	**CPU/Cores**	**1GPU**	**CPU/Cores + 1GPU**
1	15.97	**13.50**	37.26	42.12
2	30.33	**13.20**	37.72	35.26
4	60.25	**10.17**	38.40	33.79
8	119.66	**09.98**	39.70	34.27
16	237.61	**09.48**	43.01	41.94
32	473.04	76.79	**48.16**	**42.30**
64	958.89	295.24	**59.91**	**42.82**
128	1995.260	649.78	**82.85**	**43.42**
256	3807.390	1314.63	**129.04**	**44.56**

Com os testes efetuados, foi possível verificar qual a melhor estratégia para paralelização. No caso das implementações feitas para este trabalho, a solução híbrida foi que apresentou os melhores resultados, sendo esta uma opção adequada para a paralelização de problemas com um alto custo computacional. Este resultado foi alcançado pelo uso da arquitetura *multicore* e GPU em conjunto, através do particionamento dos dados e tarefas entre os recursos disponíveis.

Capítulo 8

CONCLUSÃO

A Triagem Virtual insere-se na área de Planejamento de Fármacos, efetuando simulações com ligantes e proteínas alvo conhecidas na busca de interações que permitam auxiliar na busca de novos medicamentos.

Apesar das suas vantangens, a Triagem Virtual é um método computacionalmente custoso, pois, a depender do conjunto de moléculas e dos parâmetros definidos para a simulação, a operação pode levar um tempo considerável para ser concluída.

Partindo deste problema, a utilização de plataformas de execução paralelas podem ser de grande utilidade, buscando uma melhoria de desempenho e, por consequência, um aumento da escalabilidade do processo de Triagem Virtual.

As arquiteturas *multicore* são largamente difundidas. O OpenMP é uma especificação que permite aos desenvolvedores adicionar paralelismo as suas aplicações buscando com isso uma melhoria de desempenho.

GPUs são comumentes utilizadas para acelerar aplicações gráficas. Nos últimos anos, porém, elas também vem sendo utilizadas para pesquisas científicas nas mais variadas áreas de conhecimento. O ponto em comum nessas aplicações é que todas elas demandam um alto poder de processamento para operações matemáticas e as GPUs, devido a sua arquitetura, são ideiais para este tipo de operação.

O principal objetivo deste trabalho foi adaptar e avaliar o uso de arquiteturas paralelas no processo de Triagem Virtual e buscar uma melhoria no desempenho geral de execução do processo.

Para alcançar este objetivo, foi escolhida uma ferramenta de Triagem Virtual para ter sua execução paralelizada. No caso deste trabalho, a ferramenta Autodock foi escolhida para este fim. A razão desta escolha reside no fato do Autodock ser uma ferramenta difundida entre os pesquisadores da área de desenvolvimento de novos fármacos. Além disso, o Autodock é distribuído sob a licença GPL e tem seu código fonte disponibilizado gratuitamente, permitindo que este seja adaptado para os fins necessários.

Inicialmente foram feitos testes de execução no Autodock tendo como objetivo analisar o fluxo de execução, determinar o desempenho corrente da ferramenta e buscar

quais pontos do programa podem ser paralelizados. A partir disso, foi definido que três diferentes variações de implementação do Autodock seriam feitas.

A primeira consistiu em adaptar o Autodock para arquiteturas *multicore* utilizando para isto OpenMP. Bons resultados foram alcançados com esta implementação, chegando a alcançar *speedups* de 10x no melhor cenário.

A segunda implementação foi feita em CUDA, buscando utilizar o poder de processamento de GPUs para a Triagem Virtual. Devido a sua arquitetura, a solução para GPUs alcançou resultados ainda melhores, chegando a um *speedup* de 28x no melhor cenário.

Por fim, foi criada uma implementação híbrida que utiliza conjuntamente a arquitetura *multicore* e GPUs para aumentar o desempenho da Triagem Virtual. Esta foi a implementação que alcançou os melhores resultados, chegando a um *speedup* de 85x no melhor cenário.

A partir destes resultados, foi possível determinar que o uso de plataformas de execução paralela, pode, efetivamente, melhorar substancialmente o desempenho da Triagem Virtual. Com isto, o tempo requerido para efetuar todo o processo diminui consideravelmente, permitindo que conjuntos de dados maiores possam ser utilizados para se efetuar a Triagem Virtual, aumentando a possibilidade de se encontrar bons candidatos para possíveis medicamentos para doenças conhecidas.

Por fim, estas implementações permitem que os pesquisadores da área de fármacos possam se beneficiar das melhorias propiciadas pelas arquiteturas paralelas.

Além disso, um artigo com os resultados desta pesquisa foi submetido para o IEEE International Symposium on Performance Analysis of Systems and Software, que acontecerá no mês de abril de 2017 em São Francisco, California.

Como trabalhos futuros, podem ser feitos testes das implementações desenvolvidas neste trabalho com diferentes conjuntos de ligantes e alvos macromoleculares para verificar o desempenho alcançado em diferentes conjuntos de dados. Outra possibilidade é a implementação do Autodock para dispositivos Intel Xeon Phi. Também é possível verificar a possibilidade de implementar o Autodock para a utilização de duas GPUs homogêneas com o objetivo de melhorar ainda mais os resultados encontrados e, por fim, verificar a possibilidade de implementação de soluções híbridas entre as arquiteturas.

REFERÊNCIAS BIBLIOGRÁFICAS

ARMSTRONG, J. W. *A review of high-throughput screening approaches for drug discovery*. [S.l.], 1999.

ATILGAN, E.; HU, J. Efficient protein-ligand docking using sustainable evolutionary algorithms. *2010 10th International Conference on Hybrid Intelligent Systems, HIS 2010*, p. 113–118, 2010.

AUTODOCK. 2015. Acessado em 12 de junho de 2015. Disponível em: ⟨http://autodock.scripps.edu/⟩.

BOHACEK, R. S.; MCMARTIN, C.; GUIDA, W. C. The art and practice of structure-based drug design: a molecular modeling perspective. *Med. Res*, 1996.

CHEESERIGHT, T.; SCOFFIN, R. Make way for virtual screening. *June issue of Innovations in Pharmaceutical Technology*, p. 16–19, 2013.

CHENG, Q. et al. mD3DOCKxb: A Deep Parallel Optimized Software for Molecular Docking with Intel Xeon Phi Coprocessors. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, n. Mic, p. 725–728, 2015.

CRESSET. *Blaze: effective ligand-based virtual screening to dramatically increase your wet screening hit rate at a fraction of the cost*. 2015. Acessado em 14 de junho de 2015. Disponível em: ⟨http://www.cresset-group.com/products/blaze/⟩.

CROSS JASON B.; THOMPSON, D. C. et al. Comparison of several molecular docking programs: Pose prediction and virtual screening accuracy. *J. Chem. Inf. Model., 49 (6), pp 14551474*, 2009.

FERREIRA, R.; GLAUCIUS, O.; ANDRICOPULO, A. Integração das técnicas de triagem virtual e triagem biológica automatizada em alta escala: oportunidades e desafios em p&d de fármacos. *Química Nova, São Paulo*, v. 34, n. 10, p. 1770–1778, 2011.

FORUM, M. P. *MPI: A Message-Passing Interface Standard*. Knoxville, TN, USA, 1994.

GEMDOCK. *GEMDOCK : molecular docking tool*. 2015. Acessado em 22 de julho de 2015. Disponível em: ⟨http://gemdock.life.nctu.edu.tw/dock/⟩.

GLIDE. *GLIDE - A complete solution for ligand-receptor docking*. 2015. Acessado em 22 de julho de 2015. Disponível em: ⟨http://www.schrodinger.com/Glide⟩.

GMMSB/LNCC. *DockThor: A receptor-ligand docking program*. 2015. Acessado em 29 de junho de 2015. Disponível em: ⟨http://dockthor.lncc.br/⟩.

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. Gprof: A call graph execution profiler. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 17, n. 6, p. 120–126, jun. 1982. ISSN 0362-1340. Disponível em: ⟨http://doi.acm.org/10.1145/872726.806987⟩.

HUEY, R. et al. A semiempirical free energy force field with charge-based desolvation. *Journal of Computational Chemistry*, Wiley Subscription Services, Inc., A Wiley Company, v. 28, n. 6, p. 1145–1152, 2007. ISSN 1096-987X. Disponível em: ⟨http://dx.doi.org/10.1002/jcc.20634⟩.

IBM. *IBM BlueGene*. 2016. Disponível em: ⟨http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/bluegene/⟩.

IBM. *IBM Power7*. 2016. Acessado em 18 de março de 2016. Disponível em: ⟨http://www-03.ibm.com/systems/power/hardware/775/index.html⟩.

IDRUG. *iDrug - An Online Interactive Drug Discovery and Design Platform*. 2015. Acessado em 21 de julho de 2015. Disponível em: ⟨http://lilab.ecust.edu.cn/idrug/⟩.

INTEL. *7th Generation Intel Core i7 Processors*. 2016. Acessado em 27 de outubro de 2016. Disponível em: ⟨http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html⟩.

JEFFERS, J.; REINDERS, J. *Intel Xeon Phi Coprocessor High Performance Programming*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 9780124104143, 9780124104945.

JIANG, X. et al. DOVIS 2.0: an efficient and easy to use parallel virtual screening tool based on AutoDock 4.0. *Chemistry Central journal*, v. 2, p. 18, 2008. ISSN 1752-153X.

JIANG, X. et al. DOVIS: A Tool for High-Throughput Virtual Screening. *2007 DoD High Performance Computing Modernization Program Users Group Conference*, v. 2007, p. 421–424, 2007.

KANNAN, S.; GANJI, R. Porting Autodock to CUDA. *IEEE Congress on Evolutionary Computation*, n. October, p. 1–8, 2010.

KHODADE, P. et al. Parallel implementation of AutoDock. *Journal of Applied Crystallography*, v. 40, n. 3, p. 598–599, 2007. ISSN 00218898.

KRüGER, D. M.; EVERS, A. Comparison of structure- and ligand-based virtual screening protocols considering hit list complementarity and enrichment factors. *ChemMedChem*, v. 5, n. 1, p. 148–158, 2010.

KUNTZ, I. *The official UCSF DOCK website*. 2015. Acessado em 14 de junho de 2015. Disponível em: ⟨http://dock.compbio.ucsf.edu/⟩.

LACERDA, E. G. de; CARVALHO, A. de. Introdução aos algoritmos genéticos. *Sistemas inteligentes: aplicaçoes a recursos hıdricos e ciências ambientais*, v. 1, p. 99–148, 1999.

LSMC. *GSA: Stochastic Dynamics Through Generalized Simulated Annealing*. 2015. Acessado em 29 de junho de 2015. Disponível em: ⟨http://www.cursosvirtuais.pro.br/gsa/⟩.

MORRIS, G. et al. Automated docking using a lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry*, v. 19, n. 1639-1662, 1998.

MORRIS, G.; HUEY, R. AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility. *Journal of ...*, v. 30, n. 16, p. 2785–2791, 2009. ISSN 1096-987X.

NICKOLLS, J. et al. Scalable parallel programming with cuda. *Queue*, ACM, New York, NY, USA, v. 6, n. 2, p. 40–53, mar. 2008. ISSN 1542-7730. Disponível em: ⟨http://doi.acm.org/10.1145/1365490.1365500⟩.

NORGAN, A. P. et al. Multilevel parallelization of autodock 4.2. *Journal of Cheminformatics*, v. 3, n. 1, p. 1–9, 2011. ISSN 17582946.

NUBBE. *NuBBE - Núcleo de Bioensaios, Biosíntese e Ecofisiologia de Produtos Naturais*. 2016. Acessado em 11 de outubro de 2016. Disponível em: ⟨http://nubbe.iq.unesp.br/portal/index.html⟩.

NVIDIA. *Plataforma de computação paralela CUDA*. 2015. Acessado em 12 de junho de 2015. Disponível em: ⟨http://www.nvidia.com/object/cuda_home_new.html⟩.

OPENMP. *The OpenMP API specification for parallel programming*. 2016. Acessado em 10 de outubro de 2016. Disponível em: ⟨http://openmp.org/wp/⟩.

OUYANG, X.; KWOH, C. GPU Accelerated Molecular Docking with Parallel Genetic Algorithm. *Proceedings of the 2012 IEEE 18th International ...*, p. 694–695, 2012. Disponível em: ⟨http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6413633$\backslash$nhttp://dl.acm.org/citation.cfm?id=2474562⟩.

PDB, R. *1BZL - CRYSTAL STRUCTURE OF TRYPANOSOMA CRUZI*. 2016. Acessado em 11 de outubro de 2016. Disponível em: ⟨http://www.rcsb.org/pdb/explore/explore.do?structureId=1BZL⟩.

PECHAN, I.; FEHER, B. Molecular Docking on FPGA and GPU Platforms. *2011 21st International Conference on Field Programmable Logic and Applications*, p. 474–477, 2011.

PECHAN, I.; FEHER, B.; BERCES, A. FPGA-based acceleration of the AutoDock molecular docking software. *Ph.D. Research in Microelectronics and Electronics (PRIME), 2010 Conference on*, 2010.

PRAKHOV, N. D.; CHERNORUDSKIY, A. L.; GAINULLIN, M. R. VSDocker: A tool for parallel high-throughput virtual screening using AutoDock on Windows-based computer clusters. *Bioinformatics*, v. 26, n. 10, p. 1374–1375, 2010. ISSN 13674803.

PYRX. *PyRx - Virtual Screening Tool*. 2015. Acessado em 21 de julho de 2015. Disponível em: ⟨http://mgltools.scripps.edu/documentation/links/pyrx-virtual-screening-tool⟩.

RAICU, I.; FOSTER, I. T.; ZHAO, Y. Many-task computing for grids and supercomputers. *Proceedings of the IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08). Austin, TX*, 2008.

RCSB. *RCSB PDB Protein Data Bank*. 2016. Acessado em 28 de janeiro de 2016. Disponível em: ⟨http://www.rcsb.org⟩.

RODRIGUES, R.; MANTOANI, S. et al. Estratégias de triagem virtual no planejamento de fármacos. *Revista Virtual de Química*, v. 4, n. 6, 2012.

SOLT, I.; TOMIN, A.; NIESZ, K. *New approaches to virtual screening*. 2013. Acessado em 15 de Junho de 2015. Disponível em: ⟨http://www.dddmag.com/articles/2013/12/new-approaches-virtual-screening⟩.

SOUSA, S. F.; FERNANDES, P. A.; RAMOS, M. J. Protein–ligand docking: Current status and future challenges. *Proteins: Structure, Function, and Bioinformatics*, Wiley Subscription Services, Inc., A Wiley Company, v. 65, n. 1, p. 15–26, 2006. ISSN 1097-0134. Disponível em: ⟨http://dx.doi.org/10.1002/prot.21082⟩.

THOMAS, D. B.; HOWES, L.; LUK, W. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2009. (FPGA '09), p. 63–72. ISBN 978-1-60558-410-2. Disponível em: ⟨http://doi.acm.org/10.1145/1508128.1508139⟩.

VAJDA, A. *Programming many-core chips*. [S.l.]: Springer Science & Business Media, 2011.

XILINX. *FPGA - Field Programmable Gate Array*. 2016. Acessado em 02 de fevereiro de 2016. Disponível em: ⟨http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm⟩.

ZINC. *ZINC - A Free Database of Commercially-available Compounds for Virtual Screening*. 2015. Acessado em 22 de julho de 2015. Disponível em: ⟨http://zinc.docking.org/⟩.