COPPE UFRJ
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

PARALELIZAÇÃO DA RECONSTRUÇÃO DE GEOMETRIAS MOLECULARES

Sandro Pereira Vilela

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Rio de Janeiro
Setembro de 2019

PARALELIZAÇÃO DA RECONSTRUÇÃO DE GEOMETRIAS MOLECULARES

Sandro Pereira Vilela

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Nelson Maculan Filho, D. Habil.

Prof. Luidi Gelabert Simonetti, D.Sc.

Prof. Fabio Protti, D.Sc.

Prof. Alexandre Solon Nery, D.Sc.

Prof. Josefino Cabral Melo Lima, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2019

Vilela, Sandro Pereira

Paralelização da Reconstrução de Geometrias Moleculares/Sandro Pereira Vilela. – Rio de Janeiro: UFRJ/COPPE, 2019.

XIII, 86 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2019.

Referências Bibliográficas: p. 76 – 80.

1. *Dataflow*. 2. Programação Paralela. 3. Geometria de Distâncias. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

"..a arte de programar consiste
na arte de organizar e dominar a
complexidade."
Edsger Dijkstra

Agradecimentos

Aos membros da banca examinadora, por disponibilizarem seu tempo para avaliar este trabalho.

Agradeço aos meus orientadores Felipe França e Leandro Marzulo pela motivação e o acompanhamento deste trabalho e por se disponibilizarem em me atender para tirar dúvidas a qualquer momento. Obrigado pelo apoio e a amizade.

Ao Prof. Nelson Maculan, por toda a ajuda e paciência.

Ao Prof. Carlile Lavor, por me apresentar novos horizontes de pesquisa.

Ao Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ) pela oportunidade de aprender.

Aos meus pais, por sempre estarem ao meu lado me dando forças para seguir em frente e por terem me ensinado o grande valor do estudo, educação e respeito.

À Deus, por me conceder oportunidades e me guiar para o caminho certo. Obrigado por tudo.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

PARALELIZAÇÃO DA RECONSTRUÇÃO DE GEOMETRIAS MOLECULARES

Sandro Pereira Vilela

Setembro/2019

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Programa: Engenharia de Sistemas e Computação

Nesta tese apresentamos uma abordagem para paralelizar o Problema Discreto de Geometria de Distâncias Moleculares, DMDGP, por *Dataflow*, através do particionamento da molécula segundo uma característica intrínseca dela, os vértices de simetria. A ideia consiste em dividir a molécula em partes segundo a distribuição de alguns vértices bem específicos, resolver cada uma das partes em paralelo e depois unir as soluções parciais encontradas através do emprego de matrizes de rotação. Trataremos, também, da paralelização do Problema Discreto de Geometria de Distâncias em Moléculas com Distâncias Intervalares, o *iDMDGP*, por *Dataflow*. Porém, ao invés de dividirmos a molécula, particionaremos o espaço de busca. Apresentamos alguns experimentos computacionais realizados e analisamos o comportamento das abordagens propostas em função do número de *cores* empregados. Os resultados obtidos mostraram-se animadores, foram obtidos ganhos (*speedup*) na grande maioria dos testes realizados, em alguns destes foram alcançados *speedups* acima de 12, o que demonstra a efetividade das abordagens empregadas.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

PARALLELIZATION OF MOLECULAR GEOMETRY RECONSTRUCTION

Sandro Pereira Vilela

September/2019

Advisors: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Department: Systems Engineering and Computer Science

In this thesis we present an methodology to parallelize the Discrete Molecular Distance Geometry Problem (DMDGP) using Dataflow. The method consists in partitioning the molecule according to its intrinsic characteristic, the symmetry vertices. The idea is to break the molecule into parts according to the distribution of some very specific vertices, to solve each part in parallel and then to join the partial solutions found by using rotation matrices. We will also deal with the parallelization of Interval Discretizable Molecular Distance Geometry Problem (iDMDGP) using Dataflow. However, instead of breaking the molecule, we will partition the search space. We present some computational experiments performed and analyze the behavior of the proposed approaches as a function of the number of cores employed. The results were encouraging, gains were obtained (speedups) in the vast majority of the tests performed, in some of these reached speedups above 12, which demonstrates the effectiveness of the approaches employed.

Sumário

Lista de Figuras **x**

Lista de Tabelas **xiii**

1 Introdução **1**

1.1 Motivação 1

1.2 Trabalhos relacionados 3

1.3 Contribuições 3

1.4 Organização da Tese 4

2 Problema de Geometria de Distâncias Moleculares **5**

2.1 *Distance Geometry Problem* 5

2.2 *Molecular Distance Geometry Problem* 6

2.3 *Discretizable Molecular Distance Geometry Problem* 8

2.3.1 Discretização do MDGP 8

2.3.2 Propriedade do posicionamento único 14

2.3.3 Soluções Simétricas 14

2.4 *Branch-and-Prune* 15

3 Paralelismo *DataFlow* **22**

3.1 Introdução 22

3.2 O Modelo *Dataflow* 23

3.3 A biblioteca Sucuri 24

4 Paralelizando o DMDGP **29**

4.1 Simetrias do DMDGP 29

4.1.1 Representação das Soluções 30

4.1.2 Versão adaptada do BP para o uso de simetrias, o *SymBP* . . 31

4.2 Divisão e Conquista 35

4.2.1 Divisão e Conquista para o DMDGP 35

4.2.2 Dividindo com Simetrias 36

4.2.3 Conquistando com o *SymBP* 39

4.2.4 Combinando com Rotações 39
4.3 Paralelizando a resolução do DMDGP por *Dataflow* 44
4.4 Resultados Computacionais Experimentais 46
4.4.1 Experimentos 46
4.4.2 Geração das Instâncias *DMDGPs* 47
4.4.3 Resultados 48
4.4.4 Análise 56

5 Problema de Geometria de Distâncias Moleculares Intervalares 58
5.1 Introdução 58
5.2 Uma abordagem combinatória 58
5.3 O algoritmo *iBP* 60
5.4 Computando as coordenadas das posições candidatas 62
5.5 Paralelizando o *iDMDGP* por *Dataflow* 64
5.6 Resultados Computacionais Experimentais 65
5.6.1 Experimentos 65
5.6.2 Geração das Instâncias *iDMDGPs* 66
5.6.3 Resultados 67
5.6.4 Análise 72

6 Conclusão 74
6.1 Discussão 74
6.1.1 Trabalhos Futuros 75

Referências Bibliográficas 76

A Códigos Fonte 81
A.1 Programa Sucuri *Dataflow*-DMDGP 81
A.2 Programa Sucuri *Dataflow*-iDMDGP 84

Lista de Figuras

2.1 Ilustração em 2D, de uma esfera com raio de 5Å, 1Å=$10^{-10}m$, centralizada no décimo quinto átomo, simulando um experimento de RMN aplicado a uma proteína hipotética de 19 átomos. Fonte: LIBERTI [1]. 7

2.2 Mínimos globais e locais. Fonte: LAVOR [2]. 7

2.3 Comprimento de ligações, ângulos de ligações e ângulos de torção. Fonte: GRAMACHO [3]. 9

2.4 O i-ésimo átomo pode se encontrar em apenas duas posições (i ou i') de modo que seja coerente para ser viável com a distância $d_{i-3,i}$. Fonte: GRAMACHO [3] 10

2.5 Interseção de três superfícies esféricas em $\mathbb{R}^3$, contendo dois pontos. Fonte: FIDALGO [4]. 11

2.6 Simetria das Soluções do DMDGP. Fonte: GRAMACHO [3]. 15

2.7 Inicialização do algoritmo. Os três primeiros nós da árvore. Fonte: FIDALGO [4]. 17

2.8 A ramificação em possíveis soluções, o *Branching*. Fonte: FIDALGO [4] 17

2.9 O primeiro caminho é factível e o segundo é podado. Fonte: FIDALGO [4]. 19

2.10 O primeiro caminho é podado e o segundo é factível. Fonte: FIDALGO [4]. 19

2.11 Os dois caminhos não são factíveis e, portanto, são podados. Fonte: FIDALGO [4]. 20

3.1 Exemplo de um programa em *dataflow*. Código (A): trecho de um código em alto nível. Grafo *Dataflow*(B): o grafo *dataflow* associado. Fonte: GOLDSTEIN [5]. 23

3.2 Diagrama representando a arquitetura da biblioteca *Sucuri*. Fonte: GOLDSTEIN [5]. 25

3.3 Exemplo da criação de um código *Sucuri*. (a) Código Sequencial. (b) Grafo *Dataflow*. (c) Código *Sucuri*. Fonte: ALVES *et al* [6]. 27

4.1 Árvore de Busca referente a uma instância de seis átomos, gerada pela aplicação do BP, em vermelho os ramos podados, logo podemos observar 4 soluções encontradas [4]. 30
4.2 Árvore de Busca representando as soluções encontradas para G pelo BP clássico. Fonte: FIDALGO [7]. 34
4.3 Árvore de Busca representando as soluções encontradas para G pelo *SymBP*. Fonte: FIDALGO [7]. 35
4.4 A primeira transformação aplicada à estrutura deslizante, uma translação. Fonte: FIDALGO [4]. 42
4.5 Segunda transformação aplicada à realização deslizante. Fonte: FIDALGO [4]. 43
4.6 Terceira transformação aplicada à realização deslizante. Fonte: FIDALGO [4]. 44
4.7 Grafo empregado na paralelização por *Dataflow* do DMDGP. 45
4.8 Instância empregada, *lavor100a*, 100 átomos. 49
4.9 Tempo de Execução[s] *versus* Quantidade de *cores*, instância *lavor100a*, 100 átomos. 50
4.10 *Speedup versus* Quantidade de *cores*, instância *lavor100a*. 50
4.11 Instância *lavor200a*, 200 átomos. 51
4.12 Tempo de Execução[s] *versus* Quantidade de *cores*, instância *lavor200a*, 200 átomos. 52
4.13 *Speedup versus* Quantidade de *cores*, instância *lavor200a*. 52
4.14 Instância empregada, *lavor300a*, 300 átomos. 53
4.15 Tempo de Execução[s] *versus* Quantidade de *cores*, instância *lavor300a*, 300 átomos. 54
4.16 *Speedup versus* Quantidade de *cores*, instância *lavor300a*. 54
4.17 Instância *lavor500a*, 500 átomos. 55
4.18 Tempo de Execução[s] *versus* Quantidade de *cores*, instância *lavor500a*, 500 átomos. 56
4.19 *Speedup versus* Quantidade de *cores*, instância *lavor500a*. 56

5.1 (A) Com três distâncias exatas, a interseção é dada por dois pontos. (B) Com duas distâncias exatas e uma intervalar, a interseção é dada por dois arcos. Fonte: LAVOR *et al* [8]. 59
5.2 A interseção de duas esferas com uma concha esférica. Fonte: LAVOR *et al* [9]. 60
5.3 Os vértices de referência *i', i"* e *i"'* induzem um sistema de coordenadas. Fonte: GONÇALVES [10]. 62

5.4 Os dois arcos obtidos pela interseção de duas esferas e uma concha esférica. Fonte: GONÇALVES [10]. 63
5.5 Particionamento da árvore de busca do *iDMDGP*. 64
5.6 Particionamento da árvore de busca do *iDMDGP*. 65
5.7 Tempo de Execução[s] *versus* Número de *Cores*, instância *ilavor200*. . 68
5.8 *Speedup versus* Quantidade de *cores*, instância *ilavor200*. 68
5.9 Tempo de Execução[s] *versus* Número de *cores*, instância *ilavor300*. . 69
5.10 *Speedup versus* Quantidade de *cores*, instância *ilavor300*. 69
5.11 Tempo de Execução[s] *versus* Quantidade de *cores*, instância *ilavor500b*. 70
5.12 *Speedup versus* Quantidade de *cores*, instância *ilavor500b*. 70
5.13 Tempo de Execução *versus* Número de *Cores*, instância *ilavor300b*. 71
5.14 *Speedup versus* Quantidade de *cores*, instância *ilavor300b*. 71
5.15 Tempo de Execução *versus* Número de *Cores*, instância *ilavor500c*. 72
5.16 *Speedup versus* Quantidade de *cores*, instância *ilavor500c*. 72

Lista de Tabelas

4.1 Tempo médio, em segundos, de execução gastos na determinação das soluções de cada uma das partições da instância *lavor100a* empregando 16 *cores*. 49
4.2 Dados referentes à instância *lavor100a*. 49
4.3 Tempo médio, em segundos, de execução gastos na determinação das soluções de cada uma das partições da instância *lavor200a* empregando 16 *cores*. 51
4.4 Dados referentes à instância *lavor200a*. 51
4.5 Tempo médio, em segundos, gastos na determinação das soluções de cada uma das partições da instância *lavor300a* empregando 16 *cores*. 53
4.6 Dados referentes à instância *lavor300a*. 53
4.7 Tempo médio, em segundos, de execução gastos na determinação das soluções de cada uma das partições da instância *lavor500a* empregando 16 *cores*. 55
4.8 Dados referentes à instância *lavor500a*. 55

5.1 Dados referentes a primeira série de experimentos com instâncias *iDMDGP*. 67
5.2 Dados referentes a segunda série de experimentos com instâncias *iDMDGP*. 71

Capítulo 1

Introdução

1.1 Motivação

A área de pesquisa conhecida como Geometria de Distâncias possui como objeto fundamental de estudo o Problema de Geometria de Distâncias (DGP, isto é, *Distance Geometry Problem*) que consiste em determinar um conjunto de pontos em um dado espaço geométrico a partir de algumas distâncias entre eles conhecidas [1]. Este problema tem sido alvo de intensa pesquisa nos últimos anos, aplicações interessantes são encontradas na localização em redes de sensores, reconhecimento de imagens, tomografia da internet, visualização da informação, reconstrução de mapas, reconhecimento de face, segmentação de imagens e em cálculos de estrutura molecular.

Uma das aplicações do DGP, que gerou um grande campo fértil de pesquisa, é a determinação de conformações moleculares: a partir das informações de algumas distâncias entre os pares de átomos que formam uma molécula (provenientes de experimentos de Ressonância Magnética Nuclear, RMN) é possível obter a conformação tridimensional desta, ou seja, as coordenadas de todos os átomos da molécula em estudo, através da resolução de um DGP.

O DGP aplicado às moléculas é conhecido como Problema de Geometria de Distâncias Moleculares (*Molecular Distance Geometry Problema*, MDGP).

Seria possível aplicar o DGP às Proteínas? As Proteínas constituem uma classe de moléculas extremamente importantes. Quase todos os processos biológicos envolvem as funções especializadas de uma ou mais Proteínas. Estas controlam todos os aspectos do metabolismo celular, regulam o movimento de várias espécies moleculares e iônicas através de membranas, convertem e armazenam energia celular e realizam muitas outras atividades. Essencialmente, todas as informações necessárias para iniciar, conduzir e regular cada uma dessas funções devem estar contidas na estrutura da própria proteína. Em geral, a capacidade de uma determinada proteína

desempenhar a sua função na natureza é determinada por sua forma tridimensional, ou conformação [11].

As proteínas são macromoléculas biológicas constituídas por uma ou mais cadeias de aminoácidos. Se considerarmos que existem 20 aminoácidos naturais diferentes, o número total possível de proteínas distintas entre si, do tamanho médio encontrado em nossos corpos, será astronômico, muito maior do que o número de átomos no universo. Em geral, a determinação da forma tridimensional de uma proteína é de grande importância e consiste em um problema desafiador.

Em 2006, LAVOR, LIBERTI e MACULAN [12] propuseram a primeira formulação baseada na estrutura de conformações de proteínas, onde observou-se que é possível formular o MDGP, aplicado à cadeia principal de uma proteína, como um problema de busca em um espaço discreto. Essa formulação foi chamada de Problema Discreto de Geometria de Distâncias Moleculares (DMDGP, Discretizable Molecular Distance Geometry Problem). Em 2011, MUCHERINO, LAVOR e LIBERTI [13] propuseram uma outra formulação discreta para o DGP que se baseia em hipóteses mais fracas, não relacionadas às conformações moleculares. Essa formulação foi chamada de Problema Discreto de Geometria de Distâncias (DDGP, *Discretizable Distance Geometry Problem*). Quanto à complexidade, tanto o DMDGP quanto o DDGP são NP-difíceis, como demonstrado, respectivamente, em [12] e [13].

Uma variação do DMDGP, o Problema Discreto de Geometria de Distâncias em Moléculas com Distâncias Intervalares, (*iDMDGP, Interval Discretizable Molecular Distance Geometry Problem*) foi proposto em [14]. Nesta versão, consideram-se não apenas distâncias exatas, mas também intervalares. Logo, ela possui grande potencial de aplicação em situações práticas (envolvem incertezas). Porém, o espaço de busca a ser explorado é muito maior do que o do DMDGP.

Após extensa pesquisa, FIDALGO e LAVOR [4], em 2015, descobriram a existência de relações de simetria nos vértices de estruturas do DMDGP e a partir destas relações formularam uma interessante abordagem, unindo várias frentes de pesquisas, para se paralelizar instâncias do DMDGP. Porém, não chegaram propriamente a paralelizar uma instância.

Nesta tese iremos empregar os resultados encontrados por FIDALGO e LAVOR [4] e apresentaremos uma abordagem para paralelizar o DMDGP, para isto empregaremos os bons resultados de recentes pesquisas na área de programação paralela envolvendo o modelo de execução por *Dataflow*, notadamente, MARZULO [15], MARZULO et al. [16] e ALVES et al. [6]. O referido modelo tem se mostrado uma opção atraente para programação paralela, além de ser mais transparente para os usuários, ele consegue também prover o desempenho desejado. Pretendemos, também, paralelizar o *iDMDGP* por *Dataflow*.

1.2 Trabalhos relacionados

Existem, na literatura, várias abordagens para o DMDGP. Por exemplo, o método de escalonamento multidimensional de TROSSET [17], a estratégia de redução de grafo de Hendrickson [18, 19], o algoritmo *EMDED* de CRIPPEN e HAVEL [20], o algoritmo DGSOL de MORÉ e WU [21–23], o método de perturbação estocástica de ZOU [24], o algoritmo *Variable Neighborhood Search (VNS)* de LAVOR [25], o algoritmo *Geometric Build-Up* de WU [26], a extensão do algoritmo *Geometric Build-UP* feita por CARVALHO [27], entre outros. Existem também outros levantamentos sobre métodos para a resolução deste problema, que podem ser encontrados em [20, 28–30].

Um apanhado geral sobre a teoria e aplicações no tratamento do DGP, de maneira geral, pode ser encontrado em [31]. Algumas abordagens, muito instrutivas, para o iDMDGP podem ser encontradas em [8–10] e [32, 33]. Certas estratégias para se paralelizar o DMDGP e o iDMDGP foram propostas nos últimos anos, [34] e [3], porém sem se valerem da abordagem *Dataflow*.

1.3 Contribuições

A principal contribuição desta tese consiste no desenvolvimento de duas metodologias que permitem paralelizar o DMDGP e o iDMDGP, respectivamente, empregando-se o paradigma de programação paralela *Dataflow*, em ambos os casos. Na abordagem empregada para se tratar o iDMDGP propomos um novo enfoque, onde particionamos a árvore de busca associada ao problema, ao contrário da abordagem tradicional de se particionar a molécula ao se paralelizar.

Como contribuições secundárias temos a aplicação dos resultados encontrados por FIDALGO e LAVOR [4] na paralelização do DMDGP:

- O emprego dos "vértices de simetria" no particionamento da proteína.
- O uso de uma abordagem de Divisão e Conquista para o DMDGP. Especialmente, a combinação das soluções parciais através do uso de uma translação e duas rotações.

Finalmente, como resultado de nossos estudos publicamos os seguintes trabalhos:

1. *Explorando Paralelismo Dataflow em Geometria de Distâncias Moleculares*, na *IV Escola Regional de Alto Desempenho do Rio de Janeiro*, realizada em maio de 2018. Este artigo [35] apresenta a abordagem que empregamos para se paralelizar por *Dataflow* o MDGP.

2. *Explorando Paralelismo Dataflow em Geometria de Distâncias Moleculares Intervalares*, no periódico *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*, em 2019. Neste trabalho [36] apresentamos a maneira particular que desenvolvemos para se paralelizar por *Dataflow* o iDMDGP.

1.4 Organização da Tese

O trabalho está organizado em seis capítulos. O conteúdo de cada capítulo é apresentado a seguir:

- Capitulo 2: Apresenta-se o Problema de Geometria de Distâncias Moleculares e o principal algoritmo para resolvê-lo, o *Branch-and-Prune*. Neste capítulo o leitor encontrará os conceitos básicos que o ajudarão a entender o tratamento aplicado ao DGP quando este trata de proteínas e distâncias exatas (entre os átomos).

- Capítulo 3: Aborda-se o Paralelismo por *Dataflow*, apresenta-se o modelo *Dataflow* e o uso da biblioteca *Sucuri*, escrita em *Python*. Apresentamos, de maneira resumida, os principais componentes da biblioteca.

- Capítulo 4: Neste capítulo, expomos o tratamento teórico aplicado ao DMDGP empregando-se os resultados encontrados por FIDALGO [4] na paralelização deste. Apresentamos o tratamento por *Dataflow* aplicado e os experimentos realizados.

- Capítulo 5: Apresenta-se o *iDMDGP*, a estratégia por *Dataflow* empregada para paralelizá-lo, o particionamento da árvore de busca associada e os experimentos realizados.

- Capítulo 6: Neste capítulo, avaliamos os resultados obtidos, apresentamos as conclusões e propomos alguns trabalhos futuros.

Capítulo 2

Problema de Geometria de Distâncias Moleculares

2.1 *Distance Geometry Problem*

Inicialmente, apresentaremos uma introdução formal ao chamado *Distance Geometry Problem (DGP)* - ou, em uma tradução à Língua Portuguesa, Problema de Geometria de Distâncias - este consiste (em sua roupagem mais geral) de um problema estritamente geométrico, baseado nos valores das distâncias entre pontos, o qual tem sido ampla e progressivamente estudado nos últimos anos [31].

O *DGP* pode ser caracterizado através da Teoria de Grafos, uma moderna caracterização é apresentada por LIBERTI et al. [31]. Lançaremos mão desta caracterização ao longo de todo este trabalho. Antes de definirmos o DGP, é necessário apresentarmos algumas definições, as quais iremos nos valer no tratamento do DMDGP, oriundas da Teoria de Grafos:

Definição 2.1.1(FIDALGO [4]). Seja $G = (V, E)$ um grafo, dado um vértice $v \in V$, o conjunto $N(v) = \{u \in V : \{u, v\} \in E\}$ é chamado de *vizinhança de* v ou *conjunto de vértices adjacentes a* v.

A definição para imersões em grafos em espaços Euclidianos deve-se a Hendrickson [38].

Definição 2.1.2(FIDALGO [4]). Dado um grafo $G = (V, E)$, uma realização de G em $\mathbb{R}^3$ compreende em uma imersão isométrica $x : G \to \mathbb{R}^3$.

Definição 2.1.3(FIDALGO [4]). Dados uma ordem total $<$ no conjunto de vértices V e um elemento $v \in V$, o conjunto $\gamma(v) = \{u \in V : u < v\}$ é chamado de *Conjunto de Sucessores de* v com relação a $<$.

Definição 2.1.4(FIDALGO [4]). Dado um vértice $v \in V$, seu posto é definido por $\rho(v) = \|\gamma(v)\| + 1$ e representa a quantidade de vértices até v, inclusive seguindo a ordem $<$.

Assim, a definição formal para um DGP será dada por:
Definição 2.1.5(FIDALGO [4]) Dado $K \in \mathbb{Z}_+$ e um grafo simples, não direcionado, $G = (V, E)$ cujas arestas são ponderadas pelos valores da função não-negativa $d : E \to \mathbb{R}_+$, encontre imersões $x : V \to \mathbb{R}^K$ tais que

$$\forall \{u, v\} \in E, \quad \|x(u) - x(v)\| = d(\{u, v\}). \tag{2.1}$$

onde $\|.\|$ representa a norma euclidiana.

A função x é chamada de *realização de* G, ou seja, uma representação de seus vértices em algum espaço euclidiano $\mathbb{R}^k$. Resolver um DGP consiste em associar cada vértice de G a um ponto em $\mathbb{R}^K$ de modo que se satisfaçam, simultaneamente, as equações (2.1). Ou ainda, deseja-se encontrar imersões de G no espaço em questão de modo a preservar as suas características métricas.

Um caso particular importante do DGP, corresponde à sua versão molecular o principal objeto de estudo deste trabalho, o *Problema de Geometria de Distâncias Moleculares*. Iremos descrevê-lo, a seguir.

2.2 *Molecular Distance Geometry Problem*

O **Problema de Geometria de Distâncias Moleculares** (MDGP, *Molecular Distance Geometry Problem*) consiste da determinação da estrutura tridimensional de uma molécula [12]. Ele pode ser definido como o problema de encontrar as coordenadas cartesianas $x_1, ..., x_N \in \mathbb{R}^3$ de átomos de uma molécula tais que

$$|x_i - x_j| = d_{i,j} \quad ([i, j] \in \mathbb{S}), \tag{2.2}$$

onde $\mathbb{S}$ é o conjunto de pares de átomos [i,j] cujas distâncias Euclideanas $d_{i,j}$ são conhecidas. Se todas as distâncias são dadas, o problema pode ser resolvido em tempo linear [37]. Caso contrário, o problema será *NP-hard*, [38].

As distâncias $d_{i,j}$, em 2.2, podem ser obtidas, por exemplo, através dos dados provenientes do processo físico conhecido por **Ressonância Magnética Nuclear (RMN)** juntamente com o conhecimento dos comprimentos das ligações químicas e ângulos das moléculas envolvidas. Usualmente, os dados RMN apenas fornecem as distâncias entre certos átomos de hidrogênios, [39], figura 2.1.

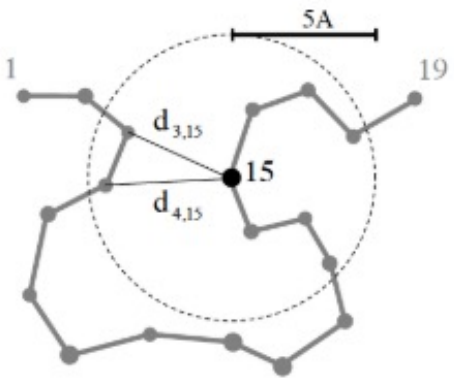


Figura 2.1: Ilustração em 2D, de uma esfera com raio de 5Å, 1Å=$10^{-10}m$, centralizada no décimo quinto átomo, simulando um experimento de RMN aplicado a uma proteína hipotética de 19 átomos. Fonte: LIBERTI [1].

A figura 2.1, apresenta uma esfera de raio 5Å centrada em um dado átomo de uma molécula, ilustra a obtenção de dados via experimentos de RMN, dependendo do dobramento da molécula é possível obter as distâncias entre alguns átomos não contíguos. Neste caso, em específico, têm se as distâncias entre o décimo quinto átomo e todos os outros átomos que se encontram dentro da esfera dada.

O Problema de geometria de distâncias moleculares, MDGP, pode ser tratado como um problema de otimização contínua, onde a função objetivo será dada por

$$f(x_1, ..., x_n) = \sum_{(i,j)\in\mathbb{S}} (|x_i - x_j|^2 - d_{i,j}^2)^2 \tag{2.3}$$

Porém, uma grande barreira que se encontra no uso desta abordagem, 2.3, consiste no fato de que o número de mínimos locais cresce exponencialmente em relação ao número de átomos da molécula dificultando a determinação dos mínimos globais que pretende-se encontrar [2], figura 2.2.

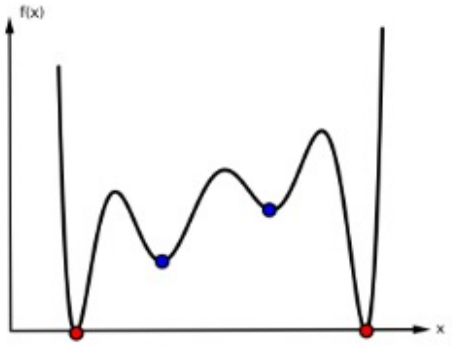


Figura 2.2: Mínimos globais e locais. Fonte: LAVOR [2].

2.3 *Discretizable Molecular Distance Geometry Problem*

Como visto anteriormente, o MDGP pode ser tratado como um problema de otimização contínua. Porém, sob algumas hipóteses adicionais é possível considerar um sub-espaço de busca do MDGP que seja discreto, contornando-se a maioria das dificuldades da abordagem contínua.

Esta possibilidade se concretizou nos trabalhos de L. Liberti, N. Maculan, C. Lavor e A. Mucherino,[12], introduzindo assim uma subclasse do MDGP, chamada de Problema Discreto de Geometria de Distâncias Moleculares (DMDGP, *Discretizable Molecular Distance Geometry Problem*), em [12] foi mostrado que o DMDGP é *NP-hard*.

Assim, o MDGP foi modelado empregando-se a Teoria de Grafos, da mesma maneira que o DGP para $K = 3$, podendo ser apresentado como um problema de decisão:

Molecular Distance Geometry Problem (MDGP) (Teoria de Grafos). Dada uma molécula M com n átomos, seja $G = (V, S, d)$ um grafo simples, não-orientado e ponderado, representando M, da seguinte maneira:

(i) V é o conjunto de vértices que representam os átomos de M, indexados por $1 \leq i \leq n$;

(ii) S é o conjunto de arestas de G, denotadas como um par de vértices $\{i, j\}$, para as quais está disponível o valor de da distância $d_{i,j}$ e

(iii) $d : S \to \mathbb{R}_+$ é um mapa que define os pesos nas arestas de G, representam as distâncias exatas entre os átomos.

Logo, existe uma realização $x : G \to \mathbb{R}^3$ de modo que esta seja uma imersão isométrica de G em $\mathbb{R}^3$? De tal maneira que valha a relação

$$\|x(i) - x(j)\| = d_{i,j} \tag{2.4}$$

para $i, j \in E$. Em caso afirmativo, encontre as conformações para G ques são factíveis em relação às equações 2.4.

2.3.1 Discretização do MDGP

A discretização do MDGP tem a sua motivação baseada na investigação das estruturas de proteínas.

Considere uma molécula como sendo uma sequência de n átomos, onde os comprimentos de ligações covalentes, são denominados por $d_{i,j}$, para $i = 2, ..., n$, os ângulos de ligações covalentes são representados por $\theta_{i-2,i}$, para $i = 3, ..., n$ e os ângulos de torção, $\omega_{i-3,i}$, são definidos pelos vetores normais dos planos definidos pelos átomos $i-3, i-2, i-1$ e $i-2, i-1, i$, respectivamente, figura 2.3.

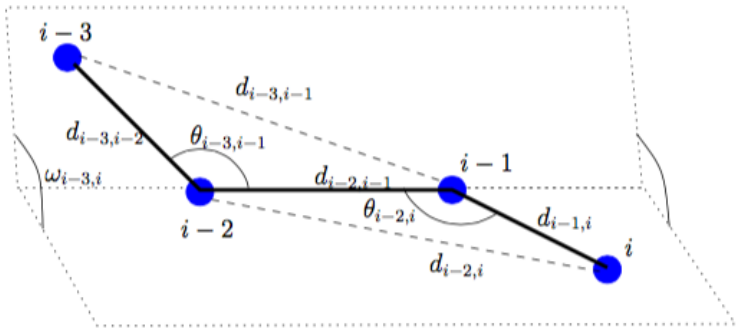


Figura 2.3: Comprimento de ligações, ângulos de ligações e ângulos de torção. Fonte: GRAMACHO [3].

Para a formulação discreta do MDGP, são consideradas as seguintes hipóteses:

Hipótese A1: os comprimentos e os ângulos de ligações, bem como as distâncias entre os átomos separados por 3 ligações consecutivas são conhecidos. Segundo a abordagem de grafos, deve-se existir uma clique entre quaisquer 4 átomos (vértices) consecutivos, onde as arestas estão relacionadas ao fato de que as distâncias envolvidas são conhecidas.

Hipótese A2: Os ângulos de ligações não podem ser múltiplos de π. Ou seja, $d_{i-3,i-1} < d_{i-3,i-2} + d_{i-2,i-1}$

A hipótese A1 é aplicável à maioria das proteínas, pois os comprimentos de ligações são conhecidos a priori. Além disso, a RMN é capaz de obter distâncias entre átomos que estão próximos entre si, e grupos de quatro átomos consecutivos da cadeia principal de uma proteína são frequentemente próximos, com valores menores do que 6Å, o que corresponde à *precisão* da RMN, [39] e [40].

A hipótese A2 é igualmente aplicável às proteínas, dado que não se conhece proteínas com ângulos de ligações covalentes com valor exato de π.

De maneira mais específica ainda, as hipóteses A1 e A2 podem ser desmembradas nas seguintes condições a serem satisfeitas, caso consideremos que V representa os átomos da molécula e o conjunto de arestas E representa as distâncias entre os átomos:

(1) O conjunto de átomos V consiste em uma sequência de n átomos de modo a

existir uma ligação covalente entre cada par de átomos consecutivos;

(2) Os comprimentos das ligações covalentes e dos ângulos entre essas ligações são conhecidos, ou seja, sabe-se os valores de $d_{i-1,i}$, para $i = 2, ..., n$ e $\theta_{i-2,i}$, para $i = 3, ...n$;

(3) As distâncias entre átomos separados por três ligações covalentes são conhecidas, isto é, tem-se os valores $d_{i-3,i}$, para cada $i = 4, ..., n$;

(4) Os átomos do conjunto $\{i-2, i-1, i\}$ são não-colineares, isto é, os ângulos de ligações $\theta_{i-2,i}$ são diferentes de $k\pi$, para $k \in \mathbb{Z}^+$;

Assumindo a validade desse conjunto de condições, é possível mostrar que cada átomo possui um número finito de posições possíveis no espaço tridimensional, respeitando as restrições de distâncias em relação aos outros átomos da cadeia. Será possível considerar, dessa forma, que o espaço de busca por imersões isométricas viáveis dessas estruturas tornar-se-á discreto.

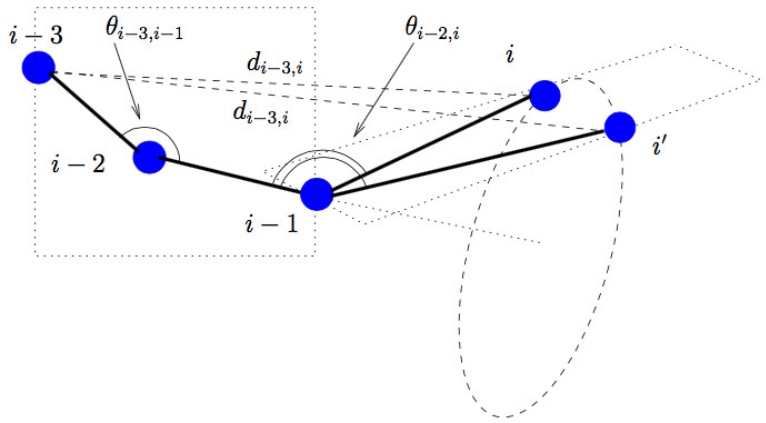


Figura 2.4: O i-ésimo átomo pode se encontrar em apenas duas posições (i ou i') de modo que seja coerente para ser viável com a distância $d_{i-3,i}$. Fonte: GRAMACHO [3] .

Pela formulação discreta, o i-ésimo átomo residirá na intersecção de três esferas centradas nos átomos $i-3, i-2, i-1$ de raios $d_{i-3,i}, d_{i-2,i}, d_{i-1,i}$, respectivamente. Pela Hipótese A2 e pelo fato de dois átomos não poderem nunca assumirem a mesma posição no espaço, a intersecção das três esferas definirá, no máximo, dois pontos (i e i', fig. 2.4). Isto nos permitirá expressar a posição do i-ésimo átomo em termos dos últimos três, dando-nos 2^{n-3} possíveis moléculas.

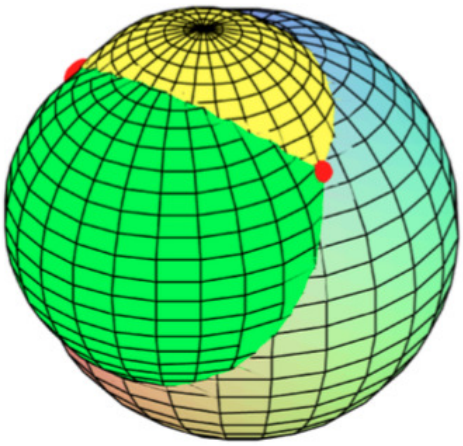


Figura 2.5: Interseção de três superfícies esféricas em $\mathbb{R}^3$, contendo dois pontos. Fonte: FIDALGO [4].

Dados todos os comprimentos de ligações $d_{1,2}, ..., d_{n-1,n}$, ângulos de ligações $\theta_{1,3}, ..., \theta_{n-2,n}$ e ângulos de torção $\omega_{1,4}, ..., \omega_{n-3,n}$ de uma molécula com n átomos, as coordenadas cartesianas $x_i = (x_{i_1}, x_{i_2}, x_{i_3})$, para cada átomo i na molécula, podem ser obtidas utilizando se a seguinte fórmula [41]:

$$\begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \\ 1 \end{bmatrix} = B_1 B_2 ... B_i \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \qquad i \in \{4, ..., n\} \tag{2.5}$$

onde

$$B_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad B_2 = \begin{bmatrix} -1 & 0 & 0 & -d_{1,2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.6}$$

$$B_3 = \begin{bmatrix} -\cos\theta_{1,3} & -\sin\theta_{1,3} & 0 & -d_{2,3}\cos\theta_{1,3} \\ \sin\theta_{1,3} & -\cos\theta_{1,3} & 0 & d_{2,3}\sin\theta_{1,3} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

e

$$B_i = \begin{bmatrix} -\cos\theta_{i-2,i} & -\sin\theta_{i-2,i} & 0 & -d_{i-1,i}\cos\theta_{i-2,i} \\ \sin\theta_{i-2,i}\cos\omega_{i-3,i} & -\cos\theta_{i-2,i}\cos\omega_{i-3,i} & -\sin\omega_{i-3,i} & d_{i-1,i}\sin\theta_{i-2,i}\cos\omega_{i-3,i} \\ \sin\theta_{i-2,i}\sin\omega_{i-3,i} & -\cos\theta_{i-2,i}\sin\omega_{i-3,i} & \cos\omega_{i-3,i} & d_{i-1,i}\sin\theta_{i-2,i}\sin\omega_{i-3,i} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

Para cada quatro átomos consecutivos $x_{i-3}, x_{i-2}, x_{i-1}, x_i$ o cosseno do ângulo de torção $\omega_{i-3,i}$ para $i = 4, ..., n$ pode ser determinado por:

$$\cos\omega_{i-3,i} = \frac{d^2_{i-3,i-2} + d^2_{i-2,i} - 2d_{i-3,i-2}\cos\theta_{i-2,i}\cos\theta_{i-1,i+1} - d^2_{i-3,i}}{2d_{i-3,i-2}d_{i-2,i}\sin\theta_{i-2,i}\sin\theta_{i-1,i+1}} \tag{2.8}$$

o que corresponde a um rearranjo da lei dos cossenos para os ângulos de torção, [12].

Empregando os comprimentos de ligações $d_{1,2}, d_{2,3}$ e o ângulo de ligação $\theta_{1,3}$, podemos calcular as matrizes B_2 e B_3, definidas em (2.4), e assim teremos:

$$x_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},$$

$$x_2 = \begin{pmatrix} -d_{1,2} \\ 0 \\ 0 \end{pmatrix},$$

$$x_3 = \begin{pmatrix} -d_{1,2} + d_{2,3}\cos\theta_{1,3} \\ d_{2,3}\sin\theta_{1,3} \\ 0 \end{pmatrix},$$

fazendo com que os três primeiros átomos da moléculas sejam fixados, Hipótese A1.

Lembrando-se que a distância $d_{1,4}$ é conhecida, através da Hipótese A1, o valor de $\cos\omega_{1,4}$ poderá ser obtido. Assim, teremos que o seno do ângulo de torção $\omega_{1,4}$ poderá ter apenas dois valores possíveis: $\sin\omega_{1,4} = \pm\sqrt{1-\cos^2(\omega_{1,4})}$. Logo, por (2.5) teremos apenas duas posições possíveis (x_4, x'_4) para o quarto átomo da molécula:

$$x_4 = \begin{bmatrix} -d_{1,2} + d_{2,3}\cos\theta_{1,3} - d_{3,4}\cos\theta_{1,3}\cos\theta_{2,4} + d_{3,4}\sin\theta_{1,3}\sin\theta_{2,4}\cos\omega_{1,4} \\ d_{2,3}\cos\theta_{1,3} - d_{3,4}\sin\theta_{1,3}\cos\theta_{2,4} - d_{3,4}\cos\theta_{1,3}\sin\theta_{2,4}\cos\omega_{1,4} \\ d_{3,4}\sin\theta_{2,4}(\sqrt{1-\cos^2\omega_{1,4}}) \end{bmatrix},$$

$$x_4^{'} = \begin{bmatrix} -d_{1,2} + d_{2,3}\cos\theta_{1,3} - d_{3,4}\cos\theta_{1,3}\cos\theta_{2,4} + d_{3,4}\sin\theta_{1,3}\sin\theta_{2,4}\cos\omega_{1,4} \\ d_{2,3}\cos\theta_{1,3} - d_{3,4}\sin\theta_{1,3}\cos\theta_{2,4} - d_{3,4}\cos\theta_{1,3}\sin\theta_{2,4}\cos\omega_{1,4} \\ d_{3,4}\sin\theta_{2,4}(-\sqrt{1-\cos^2\omega_{1,4}}) \end{bmatrix}.$$

No caso do quinto átomo, iremos obter quatro possíveis posições, uma para cada combinação de $\pm\sqrt{1-\cos^2(\omega_{1,4})}$ e $\pm\sqrt{1-\cos^2(\omega_{2,5})}$. Por indução, poderemos observar que o para o i-ésimo átomo, existirá 2^{n-3} posições possíveis. Logo, para representarmos uma molécula como uma sequência linear de n átomos, teremos 2^{n-3} possíveis sequências de ângulos de torção $\omega_{1,4}, ..., \omega_{n-3,n}$, cada uma definindo uma diferente estrutura tridimensional. Utilizando as matrizes B_i (2.5), essa sequência de ângulos de torção poderá ser convertida em uma outra sequência de coordenadas cartesianas $x = (x_i, ..., x_n) \in \mathbb{R}^3$. Portanto, ao final de todo o processo, após percorrermos toda a estrutura da molécula, teremos 2^{n-3} possíveis estruturas (conformações) no $\mathbb{R}^3$, observando-se, assim, a discretização do problema, o que nos permitirá, neste momento, apresentar a formalização do DMDGP.

O DMDGP foi apresentado como um problema de decisão por LIBERTI et al. [42], aplicado a imersões de grafos no espaço Euclideano tridimensional:

Discretizable Molecular Distance Geometry Problem (DMDGP). Dado um grafo ponderado e não-direcionado $G = (V, E, d)$, onde $d : E \to \mathbb{R}_+$, o subconjunto de vértices $U_0 = \{v_1, v_2, v_3\}$ e uma relação de ordem total em V que satisfaz a seguinte relação de axiomas:

(1) $G[U_0]$ é uma clique em três vértices (iniciando a configuração);

(2) para todo vértice v_i com posto $i = \rho(v_i) > 3, G[U_i]$, é a clique com quatro vértices e

(3) para cada vértice v_i, com posto $i = \rho(v_i) > 3$, juntamente com $\{v_{i-3}, v_{i-2}, v_{i-1}\}$, vale a desigualdade

$$d_{i-3,i-1} < d_{i-3,i-2} + d_{i-2,i-1}, \quad (Desigualdade \quad Triangular) \tag{2.9}$$

encontre uma imersão $x : V \to \mathbb{R}^3$ tal que valha $\|x(v_i) - x(v_j)\| = d_{i,j}, \forall\{v_i, v_j\} \in E$.

Assim, este problema pode ser tratado como uma busca no $\mathbb{R}^3$ por imersões isométricas de estruturas de grafos que sigam uma determinada regra para seus vértices e arestas (ordem). Esta ordem se chama DMDGP, fornecendo o nome para a classe de instâncias que satisfazem tais propriedades. Esta relação de ordem $<$, definida como dados do DMDGP, pode ser empregada a fim de construir uma árvore binária parcial e direcionada T de profunidade $n = \|V\|$ de modo que cada nó em um nível $i \leq n$ está associado a uma posição possível para o átomo i e, portanto cada caminho de comprimento n em T representa uma imersão de G em $\mathbb{R}^3$ [42]. Ou seja, o espaço de busca por soluções é, de fato, discreto para as instâncias do DMDGP.

2.3.2 Propriedade do posicionamento único

Como ilustrado na figura 2.4, uma vez que os átomos $i-3, i-2$ e $i-1$ estão fixos, existem sempre duas possíveis posições para o átomo i. Contudo, foi observado que existem alguns casos particulares, onde há somente uma posição possível para se colocar este átomo.

Valendo-se como exemplo as duas posições possíveis (x_4 e x'_4) para o quarto átomo, apresentadas anteriormente, pode-se verificar que a única diferença entre elas se encontra na coordenada z. Entretanto, se a igualdade $\sqrt{1-\cos^2(\omega_{1,4})} = -\sqrt{1-\cos^2(\omega_{1,4})}$ for satisfeita, as posições (x_4 e x'_4) são idênticas, ou seja existe somente uma posição para o quarto átomo. Esta igualdade é verdadeira quando $\cos^2(\omega_{1,4}) = 1$, e isso ocorre quando $\omega_{1,4}$ é múltiplo de π.

2.3.3 Soluções Simétricas

Em [12], foi demonstrado que para qualquer solução S do problema, existirá uma solução S' simétrica a S. Esta simetria ocorrerá em relação ao plano definido pelos três primeiros átomos que são fixados, sendo que qualquer solução de um "lado" deste plano dará origem a uma solução simétrica do outro "lado". A demonstração matemática deste teorema pode ser vista em [12].

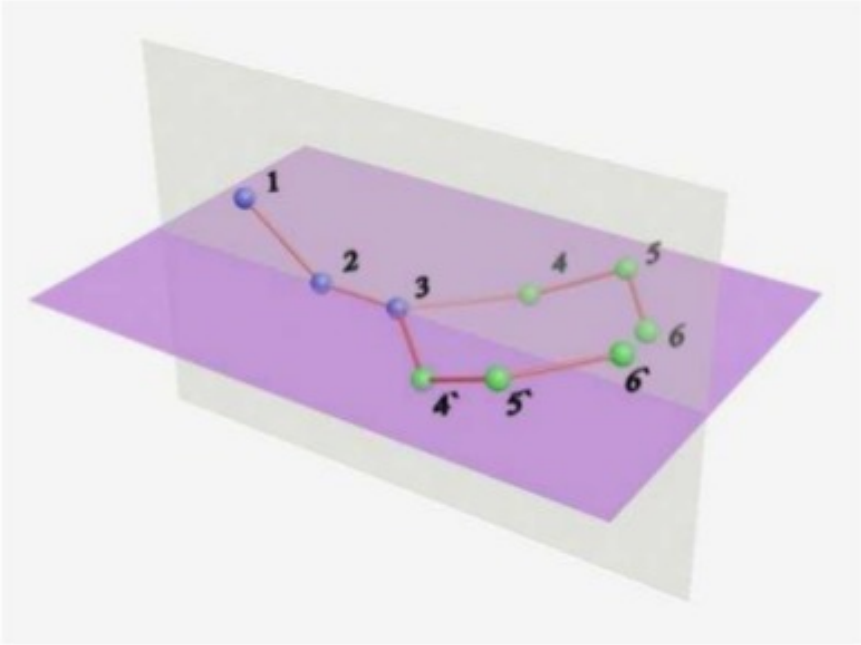


Figura 2.6: Simetria das Soluções do DMDGP. Fonte: GRAMACHO [3].

Na figura 2.4, são mostradas duas soluções simétricas para o DMDGP. Nas duas soluções, os três primeiros átomos estão nas posições 1,2 e 3, respectivamente, já o quarto, quinto e sexto átomos se encontram em posições distintas em cada uma das soluções. Para uma delas, estes átomos estão nas posições 4, 5 e 6, para a outra, eles estão em 4', 5' e 6' respectivamente. Em ambas as soluções, a distância entre quaisquer par de átomos é a mesma. Desta forma, se uma delas é válida, a outra também será. Com isto, ao se encontrar uma solução para o problema, pode-se gerar uma solução simétrica a esta.

2.4 *Branch-and-Prune*

Apresentaremos, a seguir, uma visão geral do funcionamento do principal método para resolver este problema com base na idéia da discretização, valendo-se da interseção de três e quatro esferas.

Usando a linguagem de grafos, consideraremos $G = (V, E, d)$ uma instância DMDGP com n vértices. Para cada vértice $v_i \in \mathbb{V}$, as arestas $\{v_{i-3}, v_i\}, \{v_{i-2}, v_i\}, \{v_{i-1}, v_i\} \in \mathbb{E}$ estarão sempre disponíveis, devido à definição do DMDGP. Assim, sabe-se a *priori* os valores das distâncias $d_{i-3,i}, d_{i-2,i}, d_{i-1,i}$ e posições factíveis para os três vértices.

Define-se três esferas a serem intersectadas:

- a esfera S_1^i centrada no ponto x_{i-3} e com raio $d_{i-3,i}$,
- a esfera S_2^i centrada no ponto x_{i-2} e com raio $d_{i-2,i}$,

- a esfera S_3^i centrada no ponto x_{i-1} e com raio $d_{i-1,i}$.

Da definição do DMDGP teremos que as posições para o vértice v_i devem estar concomitantemente nas três esferas, figura 2.3. Esta é a idéia principal do método empregado na solução do DMDGP. Discorrerremos sucintamente sobre os principais passos do algoritmo na seção a seguir. Antes, analisaremos as distâncias entre os vértices.

Devido ao segundo item da definição do DMDGP, cada vértice cujo posto é estritamente maior do que três possui três vértices antecessores que são adjacentes a ele. Para cada vértice $v_i \in V$, as arestas $\{v_{i-3}, v_i\}, \{v_{i-2}, v_i\}$ e $\{v_{i-1}, v_i\}$ estão disponíveis no conjunto de arestas E. Portanto, consideramos que as distâncias $d_{i-3,i}, d_{i-2,i}$ e $d_{i-1,i}$, são sempre dadas *a priori* para cada $v_i \in V$ com posto maior ou igual a quatro. Através do estudo das conformações das proteínas (os comprimentos das ligações entre os átomos e ângulos), é possível determinar os valores destas três distâncias em todos os casos.

No entanto, vale ressaltar, é possível de se obter os valores das outras distâncias $d_{i,j}$, com $\|i-j\| \geq 4$, de maneira esparsa já que geralmente elas dependem de dados de RMN. Portanto, o conjunto das arestas associadas a esses valores é esparso.

Assim, particionamos de forma disjunta o conjunto de arestas E em dois conjuntos, que são agrupados através das características descritas, em $E = E_d \cup E_p$, onde

$$E_d = \{\{v_i, v_{i+1}\} : i = 1 \leq n-1\} \cup \{\{v_i, v_{i+2}\} : i = 1 \leq n-2\} \cup \{\{v_i, v_{i+3}\} : i = 1 \leq n-3\} \tag{2.10}$$

e

$$E_p = \{\{v_i, v_j\} : \|j-i\| \geq 4\} \tag{2.11}$$

Estes subconjuntos são conhecidos como:
As arestas de E_d são chamadas de **Arestas de Discretização** (ou, do original, *discretization edges*). Já as arestas em E_p são denominadas **Arestas de Poda**, *pruning edges* [1].

Neste trabalho analisaremos o algoritmo de forma sucinta. A estrutura algorítmica geral do BP, *Branch-and-Prune*, utiliza a interseção de três esferas mencionadas anteriormente de forma recursiva, a qual pode ser compreendida através dos três passos que apresentamos a seguir, cujo produto deverá ser uma árvore binária T.

Inicialização

A primeira tarefa consiste na inicialização da estrutura. Nela, os primeiros três vértices $v_1, v_2, v_3 \in \mathbb{V}$ podem ser fixados unicamente nas posições $x_1, x_2, x_3 \in \mathbb{R}^3$ por

meio das equações apresentadas anteriormente. As posições desses vértices estão associadas aos três primeiros nós da árvore T, como ilustrado pela figura 2.5.

Figura 2.7: Inicialização do algoritmo. Os três primeiros nós da árvore. Fonte: FIDALGO [4].

Branching

Neste passo, trataremos da ramificação da árvore binária de possíveis soluções. Seguindo a ordem DMDGP de $\mathbb{V}$, uma vez que tenhamos descoberto as posições para os vértices $v_1, ..., v_{i-1}$, o próximo vértice a ser determinado na sequência será $v_i \in \mathbb{V}$.

Como as arestas de discretização $E_d^i = \{\{v_{i-3}, v_i\}, \{v_{i-2}, v_i\}, \{v_{i-1}, v_i\}\} \subset \mathbb{E}_d$ estão disponíveis, a intersecção das três esferas, representada pela figura 2.3, sempre existe e é não-vazia. O BP utiliza a clique de antecessores $\mathbb{V}_d^i = \{v_{i-3}, v_{i-2}, v_{i-1}\} \subset \mathbb{V}$ para encontrar duas possíveis posições x_i^1 e x_i^2, para que o vértice v_i pertença à estrutura DMDGP para a qual se deseja encontrar as conformações. Tal procedimento é chamado de ***Branching*** ("ramificação"), em analogia ao fato de que essas duas posições estão associadas a um par de nós no nível i da árvore T de soluções (dois novos ramos). Esta ramificação elenca todos os possíveis caminhos até as soluções do DMDGP. Assim, garante-se que, no máximo, existem 2^{n-3} conformações possíveis para G, o que corrobora com a enumerabilidade e finitude do número de soluções.

Tomemos como exemplo uma instância DMDGP com seis vértices, a árvoe com todas as $2^{6-3} = 2^3 = 8$ possíveis soluções é representada pela figura, a seguir.

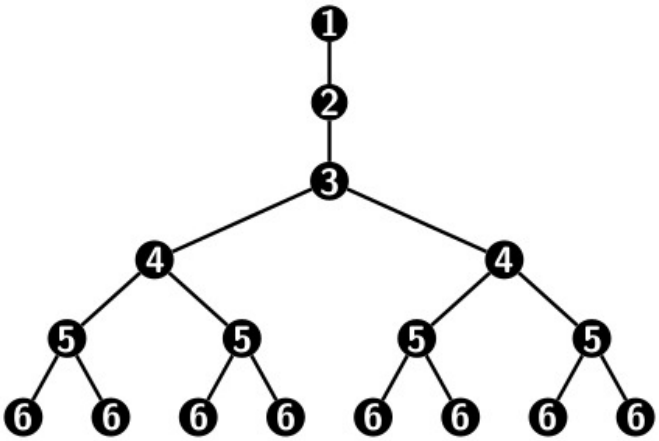


Figura 2.8: A ramificação em possíveis soluções, o *Branching*. Fonte: FIDALGO [4]

No entanto, este passo não nos fornece informação alguma a respeito da factibilidade dessas posições em relação às restrições das distâncias, inicialmente fornecidas como entrada para o algoritmo.

Pruning

Neste último passo escolheremos os "caminhos"corretos a se percorrer na árvore T, das possíveis soluções encontradas durante o *Branching*, desprezando os caminhos infactíveis. Assim, apesar do DMDGP ser um problema $\mathbb{NP}$- difícil, o BP encontrará rapidamente essas soluções. Este fato é que torna o BP um mecanismo eficiente de resolução. Não é necessário "varrer"todo o espaço de busca.

Os caminhos infactíveis serão determinados (descartados) através do processo de "poda". O processo de poda irá requerer um critério numérico adequado a fim de se viabilizar a esolha de uma posição possível, para o átomo em questão, ou o seu descarte. A isso, chamaremos de teste de Poda. Dentre os testes descritos na literatura, adota-se o DDF (*Direct Distance Feasibility*), [43].

Definição:(FIDALGO[4]) Dada uma distância d_{ij}, uma aresta de poda $\{j, i\} \in \mathbb{E}_p$, com $j < i$, e uma tolerância $\epsilon > 0$, o **Teste de Factibilidade DDF** consiste em checar se a desigualdade abaixo é válida.

$$(|x_i - x_j|^2 - d_{ij}^2)^2 < \epsilon$$

Esta tolerância é definida de acordo com a natureza do problema ou por conveniência numérica.

Portanto, após encontrar as duas posições x_i^1 e x_i^2 para o vértice, átomo, v_i, dadas pela discretização, há que se checar a factibilidade de cada uma delas através do Teste DDF para cada aresta em E_p^i.

Três resultados são possíveis para o teste DDF.

(1) **<u>Somente x_i^1 é factível</u>**

Neste caso, a posição x_i^1 é factível em relação ao teste DDF, pois todas as (d_{ij}) arestas $\{v_i, v_j\} \in \mathbb{E}_p^i$ satisfazem a desigualdade

$$(|x_i^1 - x_j|^2 - d_{i,j}^2)^2 < \epsilon$$

Por outro lado, existe uma aresta $\{v_k, v_i \in \mathbb{E}_p^i\}$ que não satisfaz a desigualdade DDF para x_i^2, pois

$$(|x_i^2 - x_k|^2 - d_{i,k}^2)^2 \geq \epsilon$$

Assim, a primeira posição dá origem a um caminho factível, mas a segunda é podada e não gera caminho algum.

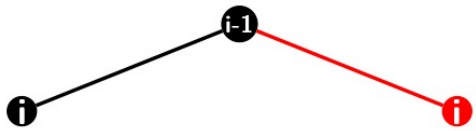


Figura 2.9: O primeiro caminho é factível e o segundo é podado. Fonte: FIDALGO [4].

(2) **Somente x_i^2 é factível**

Uma outra possibilidade consiste na existência de uma aresta $\{v_k, v_i\} \in \mathbb{E}_p^i$ que não satisfaz a desigualdade DDF para x_i^1, pois

$$(|x_i^1 - x_k|^2 - d_{i,k}^2)^2 \geq \epsilon$$

Os caminhos a partir dessa posição serão descartados. Portanto, esse nó será podado e a busca retrocederá ao nó-pai, $i-1$. Ocorrerá, então, um ***backtracking***, ou seja, um "*retorno*"ao nó pai.

Após esse retorno, a busca avança para o teste da segunda posição x_i^2. Esta, por sua vez, é factível, pois todas as arestas $\{v_j, v_i\} \in \mathbb{E}_p^i$ satisfazem a desigualdade

$$(|x_i^2 - x_j|^2 - d_{i,j}^2)^2 < \epsilon$$

Todos os caminhos factíveis deverão, então, passar por esta posição.

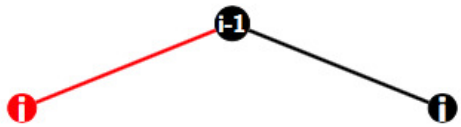


Figura 2.10: O primeiro caminho é podado e o segundo é factível. Fonte: FIDALGO [4].

(2) **x_i^1 e x_i^2 não são viáveis**

A última possibilidade consiste em existir uma aresta $\{v_k, v_i\} \in \mathbb{E}_p^i$ que não satisfaz a desigualdade DDF para x_i^1, pois

$$(|x_i^1 - x_k|^2 - d_{i,k}^2)^2 \geq \epsilon$$

Neste caso, o caminho será podado e o algoritmo retornará a busca para o nó-pai, avançando para a segunda possibilidade. Mas, existe também uma outra aresta $\{v_t, v_i\} \in \mathbb{E}_p^i$ que não satisfaz a desigualdade DDF para esta posição, pois

$$(|x_i^1 - x_t|^2 - d_{i,t}^2)^2 \geq \epsilon$$

Assim, nenhuma das duas posições são factíveis. A busca retrocederá para o nó pai anterior que ainda não teve as duas posições avaliadas pelo teste DDF.

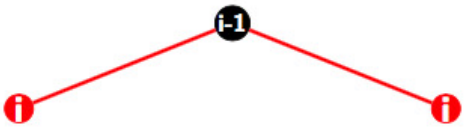


Figura 2.11: Os dois caminhos não são factíveis e, portanto, são podados. Fonte: FIDALGO [4].

Resumindo, quanto ao modo como o BP explora o espaço de busca por soluções, pode-se afirmar que as arestas de discretização de E_d atuam para moldar este espaço em forma de uma árvore binária e que as arestas de poda de E_p indicam o caminho que deve ser percorrido para encontrar as soluções factíveis.

Após definirmos os três passos do BP (inicialização, *branching* e *pruning*), apresentaremos a sua estrutura algorítmica. As informações de entrada são dadas pela instância $G = (V, S, d)$ do DMDGP. No final do processo, o método deve fornecer como saída um grafo em forma de uma árvore binária T para o qual cada nó está associado a uma posição factível em $\mathbb{R}^3$ para o vértice associado ao seu nível. Como já vimos, esta árvore T representa o espaço de busca por soluções factíveis, que é discreto e finito. Com o objetivo de implementar a recursão, define-se o **Nível de um Vértice**, v, como o seu posto, $\rho(v)$. Assim, o índice i do vértice v_i está associado ao nível i de T.

T é inicializada através dos três primeiros vértices, $1 \rightarrow 2 \rightarrow 3$, univocamente nas posições x_1, x_2 *e* $x_3 \in \mathbb{R}^3$. Até este momento não haverá ramificações ou podas. Em seguida, com o índice dos vértices i sendo incrementados de 4 a $n = |V|$, serão armazenados:

- a posição $x_i \in \mathbb{R}^3$ referente ao i-ésimo átomo e a matriz de torção acumulada $C_i = B_1 B_2 ... B_i$ e
- três apontadores em função de i: (i) $P(i)$ que aponta para o nó pai, (ii) $L(i)$ que aponta para os nós à esquerda e (iii) $R(i)$ que aponta para os nós à direita

Quando um determinado nó é infactível, o apontador em questão será inicializado com um valor fictício PRUNED para indicar onde houve uma poda, a fim de que este caminho seja inviabilizado.

A estrutura recursiva do BP é dada pelo pseudocódigo apresentado no Algoritmo 1, extraído de LAVOR *et al.* [43].

Algorithm 1 O algoritmo BP

```
BranchAndPrune(T, v, i)
if (i ≤ n − 1) then
    calcule as matrizes de torção B_i e B'_i;
    recupere a matriz de torção acumulada C_{i−1} referente ao nó-pai P(v);
    calcule as próximas matrizes de torção acumuladas C_i = C_{i−1}B_i e C'_i =
    C_{i−1}B'_i;
    empregue-as no cálculo das posições x_i = C_i y e x'_i = C'_i y;
    if x_i é factível then
        crie um nó z, armazenando C_i e x_i, faça P(z) = v e L(v) = z;
        faça T ← T ∪ {z};
        BranchAndPrune(T, z, i + 1)
    else
        faça L(v) = PRUNED;
    end if
    if x'_i é factível then
        crie um nó z', armazenando C'_i e x'_i, e fazendo P(z') = v e R(v) = z;
        faça T ← T ∪ {z'};
        BranchAndPrune(T, z', i + 1)
    else
        faça R(v) = PRUNED;
    end if
else
    solução armazenada nos nós-pais de n a 1, em busca retrocedida.
end if
```

Capítulo 3

Paralelismo *DataFlow*

3.1 Introdução

Uma das previsões tecnológicas mais duráveis já feitas consiste na Lei de Moore, uma afirmação realizada por Gordon E. Moore, em 1965: *"O número de transistores dos chips terá um aumento de 100%, pelo mesmo custo, a cada período de 18 meses."* [44]. Esta projeção mostrou-se real e padronizou os avanços exponenciais das tecnologias de computação por muito tempo, mais de 50 anos. Porém a validade desta lei está chegando ao final, a extrema miniaturização dos transistores, o que permite que em um *chip* tenhamos atualmente uma quantidade da ordem de centenas de bilhões de transistores, está se aproximando de um limite. Estudos apontam que a validade da Lei de Moore se encerra ao se alcançar o comprimento de onda *Compton* do elétron ($\lambda_C = 1\%$ da distância entre os átomos de silício), o que está previsto para ocorrer aproximadamente em 2036 [44]. Levando-se em conta que, além do crescimento da quantidade de transistores, ocorreu também o aumento da frequência do relógio (*clock* dos processadores) a cada nova geração de processadores, o que contribuiu para dificultar mais ainda o tratamento da dissipação de calor, chegando a valores proibitivos a temperatura [45].

Assim, devido a estes e outros inconvenientes (alto consumo de energia, por exemplo) a indústria tem buscado soluções engenhosas para o contínuo crescimento da capacidade de processamento, até que a ciência crie uma nova tecnologia de fabricação de transistores ou um novo paradigma de processadores. Uma destas soluções (abordagens) é a computação paralela, esta se tornou o paradigma dominante nas arquiteturas de computadores sob a forma de processadores multinúcleos [46]. Porém, escrever programas para aplicações em paralelo não é uma tarefa trivial: *"Concorrência é a próxima e maior revolução em como escrever programas (após a Programação Orientada a Objetos)"* [47].

Uma opção interessante para a escrita de programas em paralelo é a abordagem

por *Dataflow*, ela permite que o programador não se preocupe com os detalhes complexos da paralelização, ao contrário da maioria das outras abordagens existentes (seção 3.3). Recentes pesquisas na área mostram bons resultados, notadamente [15], [16] e [6].

3.2 O Modelo *Dataflow*

Em programação de computadores, *Dataflow Programming* corresponde a um paradigma de programação que modela um programa como um grafo dirigido (*DAG*) o qual representa os dados fluindo entre as operações, implementando assim os princípios e a arquitetura de fluxo de dados. A programação *Dataflow* teve início através das pesquisas de Jack Dennis e seus alunos de pós-graduação no MIT na década de 60 [14].

Programas no modelo *Dataflow* podem ser descritos como um grafo de fluxo de dados onde os nós representam as instruções e uma aresta direcionada $A \rightarrow B$ indica que a instrução A produz um operando que será a entrada da instrução B.

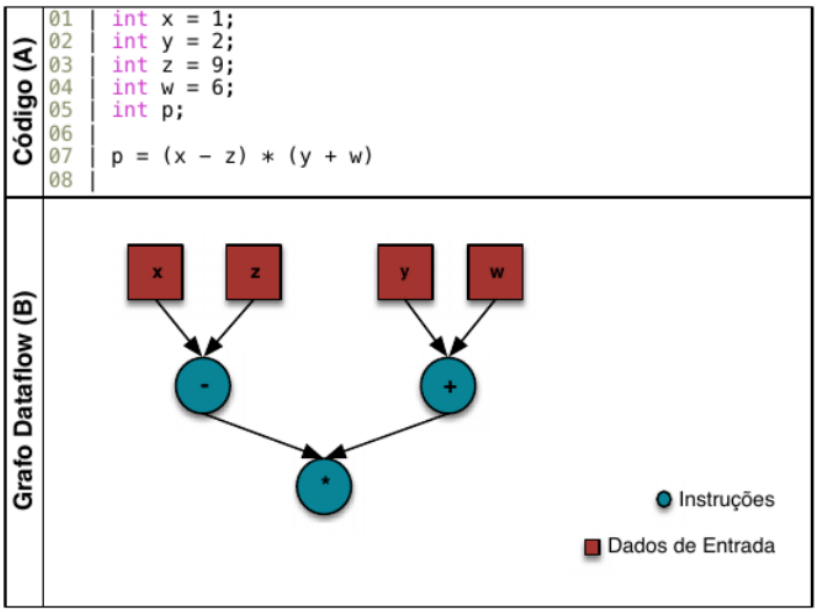


Figura 3.1: Exemplo de um programa em *dataflow*. Código (A): trecho de um código em alto nível. Grafo *Dataflow*(B): o grafo *dataflow* associado. Fonte: GOLDSTEIN [5].

A figura 3.1 apresenta um exemplo de um programa simples em *dataflow*. No campo superior(A) apresenta-se um trecho de um código em alto nível e no quadro inferior (B) o grafo de fluxo de dados associado. Os círculos em azul representam as instruções e as arestas direcionadas entre eles representam as dependências entre as instruções, são o fluxo de dados da aplicação. Os quadrados representam nós especiais, "nós de entrada", os quais recebem os parâmetros de entrada, as instâncias

de entrada do programa. Note que as instruções de soma e multiplicação são independentes e poderiam ser executadas em qualquer ordem e principalmente em paralelo.

O modelo *dataflow* permite que as instruções sejam executadas tão logo seus operandos de entrada estejam disponíveis. Este modelo expõe de uma maneira natural o paralelismo potencial das aplicações, liberando as partes do programa que não dependem entre si para que sejam executadas simultaneamente, limitadas agora apenas à quantidade de recursos disponíveis.

3.3 A biblioteca Sucuri

Podemos afirmar que, tipicamente, a programação por *dataflow* é realizada instanciando blocos de códigos e conectando-os em um gráfico, de acordo com as suas dependências. Isso livra o programador da maior parte do trabalho complexo que é a paralelização de uma aplicação, pois a sincronização é tratada pelo ambiente de execução *dataflow* ou através de uma biblioteca. Porém, ainda existe uma lacuna entre os atuais projetos *dataflow* e as linguagens de alto nível, muito adotadas pela comunidade científica, como *R, Python*, etc. Embora se possa argumentar que tais linguagens não foram projetadas para a computação de alto desempenho e que elas não são inerentemente ideais para a programação paralela, a imensa popularidade delas faz com que valha a pena investigar formas de acelerar a sua execuçao.

Neste contexto, ALVES et al.[6], desenvolveram a Sucuri, uma biblioteca minimalista para linguagem *Python* que fornece programação e execução em *dataflow*. O principal objetivo consiste em que o programador possa implementar as funções do seu programa e, em seguida, atribuí-las aos nós, que são por sua vez, conectados de acordo com as suas dependências. Depois que esses nós e suas respectivas dependências são instanciados o programa poderá ser executado em uma única máquina *multicore* ou em um grupo deles, as funções serão acionadas de acordo com a regra de *dataflow*. Isto significa que se duas funções não tiverem dependências entre elas, estas serão potencialmente executadas em paralelo, dependendo da disponibilidade de núcleos ociosos nas máquinas alocadas.

Resumidamente, podemos afirmar que os principais componentes da *Sucuri* são **Nó**, **Grafo**, **Tarefa**, **Trabalhador** e **Escalonador**:

- **Nó**: São objetos associados a funções e conectados por arestas pelo programador, através do método *add_edge*. As arestas descrevem as dependências entre os dados e os nós representam o processamento a ser realizado quando as dependências forem satisfeitas. Cada nó possui: a função a ser executada ao receber todos os operandos de entrada, a lista de nós destinatários que depen-

dem dos operandos resultantes e a lista de operandos recebidos aguardando casamento;

- **Grafo**: É um objeto empregado como contêiner, representa toda a aplicação *dataflow*. Corresponde a uma estrutura composta por um conjunto de nós e arestas.

- **Tarefa**: Será criada pelo escalonador sempre que todos os operandos de entrada de um determinado nó tornarem-se disponíveis. Cada tarefa conterá a lista de operandos de entrada e o *id* dos nós relativos a ela.

- **Trabalhador**: São processos instanciados pela *Sucuri* para executar tarefas, são responsáveis por processarem as tarefas enviadas pelos escalonadores. Quando um *Trabalhador* encontra-se ocioso ele solicitará uma tarefa ao seu escalonador local. Assim que recebe uma tarefa, o *Trabalhador* consulta o nó correspondente àquela tarefa, no grafo, e executa a função associada. A quantidade de *Trabalhadores* por máquina é recebido como parâmetro pela *Sucuri*.

- **Escalonador**: É responsável pela comunicação entre os nós do Grafo, bem como pelo envio de tarefas aos *Trabalhadores*. É composto por uma *Unidade de Casamento - Matching Unit* e uma *Fila de Prontos*. Possui como função casar os operandos de entrada e gerar tarefas, de acordo com a regra de disparo *dataflow*: cada operando enviado por um *Trabalhador* é armazenado em uma *Unidade de Casamento* até que todos os operandos de entrada de um nó estejam disponíveis, resultando na instanciação de uma tarefa que será inserida na *Fila de Prontos - Ready Queue*, de onde o escalonador irá retirá-la quando um *Trabalhador* ocioso fizer uma requisição.

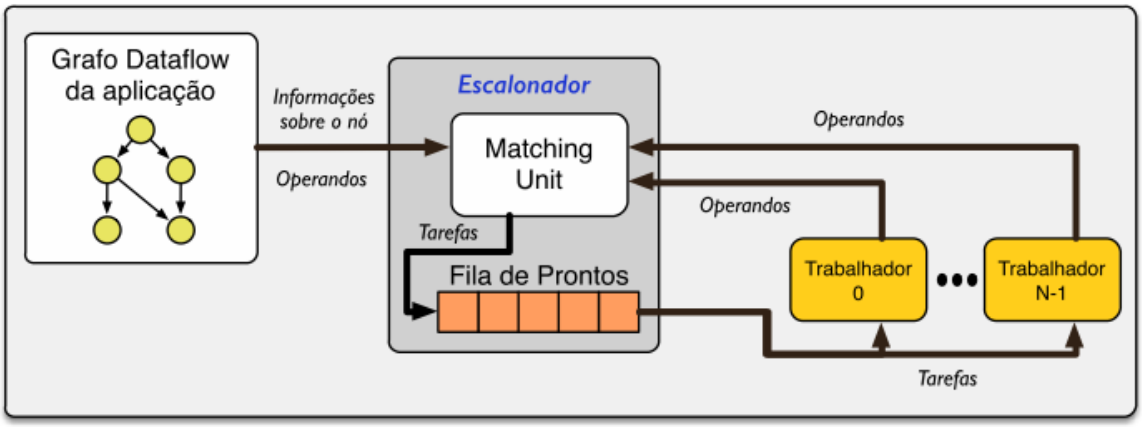


Figura 3.2: Diagrama representando a arquitetura da biblioteca *Sucuri*. Fonte: GOLDSTEIN [5].

Através da fig. 3.2 podemos observar a relação entre os principais componentes da biblioteca *Sucuri*. Em destaque o *Escalonador*, a figura evidencia o seu comportamento, o qual é responsável por alimentar a lista de operandos presente em cada nó do $Grafo$, caso ocorra casamento (todas as dependências para processar aquele nó tenham sido satisfeitas) uma tarefa será criada e adicionada à *Fila de Prontos*. Quando o *Escalonador* receber um operando contendo a mensagem $REQUEST_TASK$ de um *Trabalhador*, uma tarefa da *Fila de Prontos* será removida e enviada ao *Trabalhador* para processá-la.

Uma das vantagens do emprego da *Sucuri* é a possibilidade de estender a execução de uma aplicação *multicore* para *clusters*, para isto basta que se altere uma *flag*. Esta *flag* faz com que a *Sucuri* empregue o padrão *mpi* para a comunicação entre as máquinas.

Para se criar uma aplicação *dataflow* valendo-se da *Sucuri* é necessário os seguintes passos:

(i) instanciar o objeto *DFGraph*, o Grafo da aplicação;

(ii) criar os Nós (objetos) associados às funções definidas pelo programador;

(iii) conectá-los, segundo as suas dependências, empregando o método *add_edge*;

(iv) adicionar os nós ao objeto *DFGraph*;

(v) instanciar o objeto *Scheduler*, o *Escalonador*, o qual receberá o Grafo;

(vi) *e*xecutar o *Escalonador*, através do método *start*.

A fig.3.3 apresenta o exemplo de uma aplicação hipotética que emprega a *Sucuri*. Em (a) tem-se a representação de um código sequencial, em (b) o grafo *dataflow* da aplicação e em (c) é apresentado o código com o emprego da biblioteca *Sucuri*. Considera-se que as funções *f, g, h, w* e *t* são as funções especializadas da aplicação do usuário.

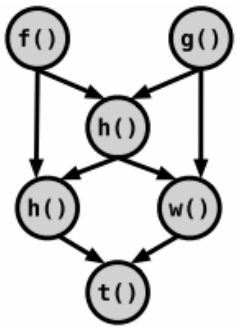
De maneira detalhada, temos em fig.3.3(c), a cada linha:

- linha 1: importa-se a biblioteca *Sucuri*.
- linha 2: instancia-se o grafo *dataflow* da aplicação, um grafo vazio.
- linha 3: instancia-se o *Escalonador*, o qual recebe o grafo instanciado e o número de *Trabalhadores* a ser utilizado pela *Sucuri*.
- linha 4 a 9: criam os nós do grafo, associando cada um deles a uma função. O segundo parâmetro na criação do nó corresponde ao número de operandos de entrada necessários à função associada.

- linha 10 a 15: adiciona-se os nós anteriormente criados ao grafo.
- linha 16 a 23: conecta-se os nós, empregando-se o método *add_edge*, onde o primeiro parâmetro corresponde ao nó destino e o segundo parâmetro é a porta de entrada no nó destino.
- linha 24: o *Escalonador* é inicializado, executa-se a aplicação.

```
01 | a = f()
02 | b = g()
03 | c = h(a,b)
04 | d = h(a,c)
05 | e = w(c,b)
06 | x = t(d,e)
```

(a)



(b)

```
01 | from Sucuri import *
02 | graph = DFGraph()
03 | sched = Scheduler(graph, nWorkers)
04 | a = Node(f, 0)
05 | b = Node(g, 0)
06 | c = Node(h, 2)
07 | d = Node(h, 2)
08 | e = Node(w, 2)
09 | x = Node(t, 2)
10 | graph.add(a)
11 | graph.add(b)
12 | graph.add(c)
13 | graph.add(d)
14 | graph.add(e)
15 | graph.add(x)
16 | a.add_edge(c,0)
17 | b.add_edge(c,1)
18 | a.add_edge(d,0)
19 | c.add_edge(d,1)
20 | c.add_edge(e,0)
21 | b.add_edge(e,1)
22 | d.add_edge(x,0)
23 | e.add_edge(x,1)
24 | sched.start()
```

(c)

Figura 3.3: Exemplo da criação de um código *Sucuri*. (a) Código Sequencial. (b) Grafo *Dataflow*. (c) Código *Sucuri*. Fonte: ALVES *et al* [6].

Este código pode ser executado em uma máquina, ou em um *cluster* de máquinas, para isto basta que o usuário passe o parâmetro *mpi_enabled* como *TRUE* para o *Escalonador*. A *Sucuri* emprega memória compartilhada para a comunicação local.

O *Grafo* da aplicação pode ser composto por vários subgrafos. Cada subgrafo é considerado um nó no grafo principal, tornando-se assim possível a criação de nós

especializados. Tais nós são capazes de representar modelos paralelos ou até mesmo, executar determinadas funções sem a necessidade de qualquer implementação. Logo, naturalmente teve-se a criação de *templates* [6], o que facilita mais ainda a programação no caso de alguns algoritmos que são padrões muitos utilizados em programação paralela, tais como $forkjoin$, *pipeline e wavefront.*

Devido ao desenvolvimento e contínuo aprimoramento de todos estes recursos a *Sucuri* tem obtido excelentes resultados em uma grande gama de aplicações paralelas [6]. Assim, utilizamos em nosso trabalho a biblioteca *Sucuri* com o intuito de facilitar o processo de paralelização de nossas aplicações e empregá-la em uma frente de pesquisa desafiadora como o da determinação da forma espacial das proteínas.

Capítulo 4

Paralelizando o DMDGP

4.1 Simetrias do DMDGP

Este capítulo baseia-se principalmente nos trabalhos realizados por FIDALGO [4] e LAVOR [34]. Através de suas pesquisas descobriu-se a existência de algumas simetrias inerentes à própria natureza da estrutura DMDGP e estas poderiam ser muito úteis.

Inicialmente, demonstrou-se a existência de uma simetria básica no quarto nível da árvore de realizações (soluções), a qual provou ser "universal"no sentido de estar presente em qualquer instância do problema. Depois de longa pesquisa no tema, descobriu-se uma relação de simetria mais geral, que pode ocorrer em vários níveis de uma estrutura DMDGP, [48].

Descobriu-se que tais simetrias são poderosas aliadas na diminuição do tempo de computação do BP. Ao encontrar uma única solução, todas as remanescentes são determinadas através de reflexões parciais desta solução, [48]. Portanto, as simetrias nos permitem conhecer quais os caminhos que o BP deve evitar, afim de que a busca por soluções seja eficiente, até encontrar a primeira solução e, a partir dela, construir todas as outras.

Com o intuito de organizar os dados de simetria de uma estrutura $G = (V, E, d)$ do DMDGP, Mucherino *et al.*, [49], definiram o subconjunto de vértices, S_G. Através deste é possível inferir o número de soluções de um DMDGP.

Teorema 3.1(FIDALGO [4]): Os vértices do conjunto S_G consistem em todos os vértices de simetria de G.

$$S_G = \{v_i \in \mathbb{V} : \nexists \{v_j, v_k\} \in E \textit{ tal que } j + 3 < i \geq k\}$$

Corolário 3.1(FIDALGO [4]): O número de soluções de um DMDGP associado a uma instância $G = (V, E, d)$ é dado por

$$|S| = 2^{|S_G|}$$

Assim, uma vez que conhecemos o número de vértices de simetria sabemos de antemão, sem a necessidade de executar o BP, a quantidade de soluções de uma dada instância, o que é muito útil. A demonstração do Teorema 3.1 pode ser encontrada em [4].

4.1.1 Representação das Soluções

FIDALGO [4] apresenta uma maneira computacionalmente mais eficiente para representar as soluções de um DMDGP, descobertas através do BP, corresponde a uma estrutura binária que permite identificar biunivocamente cada uma das soluções como uma sequência de zeros e uns. A seguir, apresentamos as etapas necessárias para a geração de tais representações.

- As três primeiras posições são representadas por uma sequência de três zeros;

A partir da quarta posição em diante:

- a entrada será 0, caso represente a posição à esquerda;
- a entrada será 1, caso represente a posição à direita.

Consideremos o exemplo a seguir retirado de [4], apresentamos uma árvore de busca (gerada pelo BP) referente a uma instância de seis átomos, onde ao final foram encontradas 4 soluções:

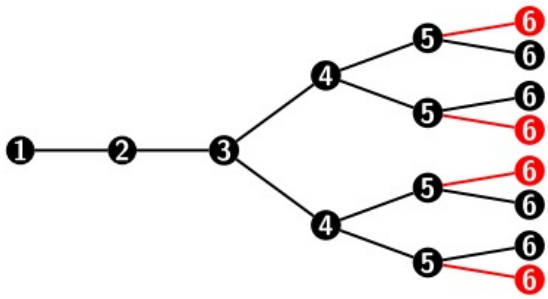


Figura 4.1: Árvore de Busca referente a uma instância de seis átomos, gerada pela aplicação do BP, em vermelho os ramos podados, logo podemos observar 4 soluções encontradas [4].

As quatro soluções encontradas são representadas, respectivamente, por:

- $x^1 = (0, 0, 0, 0, 0, 1)$
- $x^2 = (0, 0, 0, 0, 1, 0)$
- $x^3 = (0, 0, 0, 1, 0, 1)$

- $x^4 = (0, 0, 0, 1, 1, 0)$

Esta representação será de grande valia para representar as soluções do DMDGP. No próximo capítulo a empregaremos na descrição do algoritmo que utilizaremos no tratamento do DMDGP.

4.1.2 Versão adaptada do BP para o uso de simetrias, o *SymBP*

As informações sobre quais vértices da estrutura DMDGP possuem a carcterística de definir simetrias possibilitam não somente a contagem parcial ou total do número de soluções como também a definição de todas as possíveis soluções existentes através de apenas uma delas.

Assim, desenvolveu-se um algoritmo, *SymBP*, que determina todas as soluções de um DMDGP usando esta estrutura de simetrias. Ele emprega o BP no modo tradicional para encontrar a primeira solução. Em seguida, determina-se todas as outras soluções usando reflexões parciais da primeira solução e das que são encontradas subsequentemente, [4, 49].

O *SymBP* recebe:

- v é o vértice atual a ser posicionado no espaço.
- n corresponde ao número de vértices, átomos, da instância $G = (V, E, d)$, $n = |V|$
- d é o conjunto de distâncias disponíveis para a instância,
- S é o conjunto de simetrias existentes para a instância G, neste trabalho S_G.
- $nsols$, número de soluções encontradas,
- $prev$, vetor de variáveis binárias (0/1), vide seção 4.1.1, o qual armazena em si a última solução encontrada e traduzida para o formato binário de representação, [4].

A seguir, o pseudocódigo do algoritmo, como apresentado em [4, 49], o $SymBP$.

Algorithm 2 - O algoritmo *SymBP*

1: $symBP(v, n, d, S, nsols, prev)$
2: **if** $(v > n)$ **then**
3: $nsols \leftarrow nsols + 1$; ▷ Uma solução é encontrada
4: **remova** todos os galhos tais que: ▷ Descarte os galhos infactíveis

- sua raíz é $u \in S$;
- suas folhas são $v_f < n$.

5: **return** $nsols$;
6: **end if**
7: ▷Ramifique em nós possíveis
8: **if** $v \in S$ **then**
9: **calcule** x_v^0;
10: **faça** $prev(v) = 0$;
11: $symBP(v + 1, n, d, S, nsols, prev)$;
12: **calcule** x_v^1;
13: **faça** $prev(v) = 1$;
14: $symBP(v + 1, n, d, S, nsols, prev)$;
15: **end if**
16: ▷Sem podas, não há soluções ainda
17: ▷Calcule as posições viáveis: faça as podas
18: **if** $((v \notin S)$ **and** $(nsols = 0))$ **then**
19: **calcule** x_v^0;
20: **if** (x_v^0 é factível) **then**
21: **faça** $prev(v) = 0$;
22: $symBP(v + 1, n, d, S, nsols, prev)$;
23: **end if**
24: **if** $(nsols = 0)$ **then**
25: **calcule** x_v^1;
26: **if** (x_v^1 é factível) **then**
27: **faça** $prev(v) = 1$;
28: $symBP(v + 1, n, d, S, nsols, prev)$;
29: **end if**
30: **end if**
31: **end if**
32: ▷ Se a primeira solução já foi calculada:
33: **if** $((v \notin S)$**and** $(nsols > 0))$ **then**
34: **calcule** $x_v^{\neg prev(v)}$;
35: **faça** $prev(v) = \neg prev(v)$; ▷ as outras serão geradas por simetria
36: $symBP(v + 1, n, d, S, nsols, prev)$;
37: **end if**
38: **return** $nsols$;

O algoritmo *SymBP*, assim como o BP, também emprega a recursividade. Ele verifica se o último vértice, na ordem DMDGP, foi posicionado na última chamada realizada deste método. Neste caso, o parâmetro de contagem de soluções $nsols$ é

atualizado, pois uma solução foi encontrada

$$nsols \leftarrow nsols + 1$$

Caso uma das soluções já tenha sido determinada, então as escolhas de 0/1 estarão definidas para cada vértice, $v_i \notin S_G$. Com o intuito de *"refinar"* a atual árvore de soluções os caminhos que não definem soluções são removidos. Os únicos galhos parciais que são mantidos na árvore são os que possuem algum $v_i \in S_G$ como raiz. Portanto, considera-se, apenas, pares de galhos simétricos factíveis, aqueles que possuem suas raízes nos níveis $i-1$, para cada $v_i \in S_G$. Após esta checagem, a existência ou não de solução, o algoritmo atende ao seu passo básico. A cada chamada do $SymBP$, a pertinência do vértice atual v_i ao conjunto de simetrias S_G é testada.

Em caso afirmativo, o método calculará as soluções x_i^0 e x_i^1 e avançará para o nível seguinte, $v_i \leftarrow v_{i+1}$ e recursivamente será acionado novamente.

Caso contrário, se $v_i \notin S_G$, há que se distinguir dois casos:

(1) Nenhuma solução foi determinada ainda:

- acaso nenhuma soluçao tenha sido encontrada ainda, então ($nsols = 0$), as informações sobre simetrias não podem ser exploradas ainda e a mecânica do BP convencional é aplicada;
- o vetor binário $prev$ deve ser mantido atualizado, a fim de que ele contenha a primeira solução, assim que for encontrada;
- sem perda de generalidade, caso uma solução seja encontrada para o vértice v_i na posição x_i^0, então a posição x_i^1 deve ser automaticamente descartada;

(2) A primeira solução já foi encontrada:

- acaso a primeira solução já tenha sido encontrada, o $SymBP$, construirá todas as outras soluções a partir das informações de S_G aplicando as reflexões parciais;
- Não há mais ramificações a se fazer na árvore (*branchings*), pois $x_i^{\neg prev(v_i)}$ só pode ser factível se $x_i^{prev(v_i)}$ for factível para a solução anterior;
- Também não há mais podas, pois todas as posições a serem geradas são factíveis por simetria, o que implica em uma drástica redução no custo computacional.

Com o intuito de ilustrar a execução do *SymBP* apresentaremos, a seguir, um exemplo retirado de [7].

Exemplo 4.1.2. Seja a instância $G = (V, E, d)$ do DMDGP com onze vértices no total e arestas de podas iguais a

$$E_p = \{\{1, 7\}, \{5, 11\}, \{7, 11\}\} \tag{4.1}$$

considere que o conjunto de soluções associado a esta instância foi obtido através do BP. Em seguida, determine o conjunto de simetrias desta instância (empregue o teorema 3.1),

$$S_G = \{v_4, v_8\}. \tag{4.2}$$

Logo,

$$|S| = 2^{|S_G|} = 2^2 = 4, \tag{4.3}$$

Assim, pode-se afirmar que existem quatro soluções associadas a esta instância.

Na fig. 4.2 apresentamos uma árvore que representa a execução do BP clássico para G, os caminhos em vermelho representam as posições que foram descartadas devido à infactibilidade dos nós (folhas) e, consequentemente, mapeam as regiões onde ocorrem *bactrackings* nas buscas em profundidade pelas soluções.

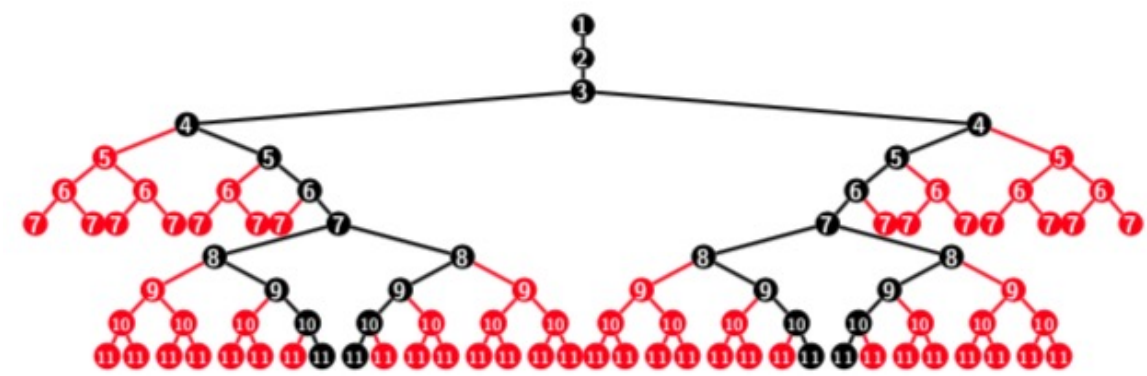


Figura 4.2: Árvore de Busca representando as soluções encontradas para G pelo BP clássico. Fonte: FIDALGO [7].

Aplicaremos, agora, o *SymBP* à mesma instância. A árvore de busca gerada é representada pela fig. 4.3.

Da análise desta árvore concluimos que:

- o intervalo de tempo de execução do *SymBP* é menor do que o do BP clássico, pois este é proporcional à quantidade de *backtrackings* realizados, de fato é o que realmente consome tempo no BP. A quantidade de retornos no *SymBP* foi menor do que a metade em relação ao BP.

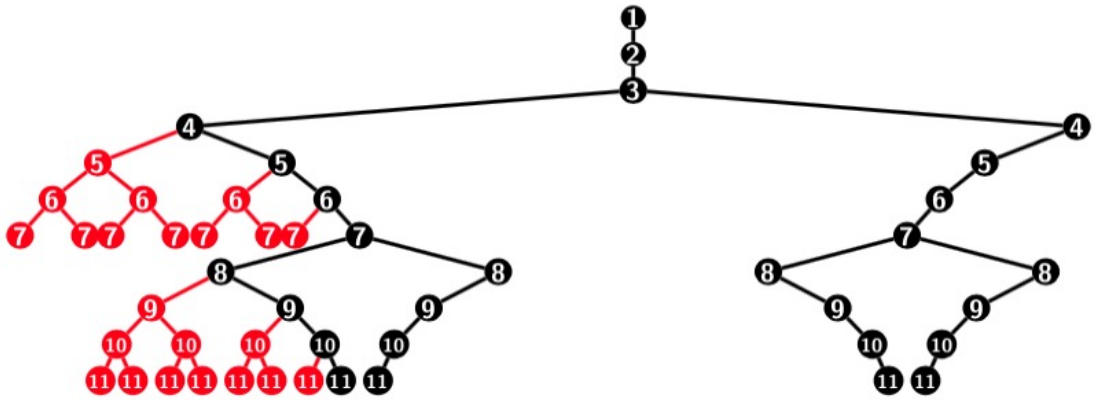


Figura 4.3: Árvore de Busca representando as soluções encontradas para G pelo *SymBP*. Fonte: FIDALGO [7].

- tais retornos acontecem, até que se encontre a primeira solução x_1, dada por

$$x_1 = (0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1); \tag{4.4}$$

- a segunda solução x_2 corresponde ao reflexo da primeira solução, pela simetria de v_8, ou seja,

$$x_1 = (0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0); \tag{4.5}$$

- a terceira x_3 e a quarta x_4 soluções correspondem aos reflexos da segunda e primeira soluções, respectivamente, pela simetria de v_4, isto é,

$$x_3 = (0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1) \quad e \quad x_4 = (0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0). \tag{4.6}$$

4.2 Divisão e Conquista

4.2.1 Divisão e Conquista para o DMDGP

Nesta seção trataremos de uma estratégia, apresentada em, [4], para paralelizar uma instância DMDGP valendo-se principalmente das propriedades de simetria inerentes destas estruturas.

A idéia de paralelizar o DMDGP não é inédita, MUCHERINO *et al.* [34], em trabalho de 2010, introduziram uma forma de dividir uma instância DMDGP em sub-instâncias, resolvendo cada uma por meio de processadores independentes (computação paralela). O objetivo consistiu em obter soluções parciais e uni-las a fim de formarem soluções completas.

Motivados por este modelo, em paralelo, NUCCI *et al.*, [50], propuseram o chamado método de múltiplas árvores de realizações, com o intuito de computar as

realizações de instâncias DMDGP. Neste método, dado uma instância já dividida, utiliza-se o BP para calcular as árvores parciais de realizações (as que armazenam as soluções parciais) e, então, realiza-se uma translação e duas rotações para unir tais árvores, duas-a-duas.

Em 2013, Gramacho, [3], também propôs uma versão paralela do algoritmo BP para o DMDGP, valendo-se de uma nova "ordem" para os átomos da cadeia principal de uma molécula de proteína.

Ambos os trabalhos, citados acima, lidam com uma estratégia DC para resolver o DMDGP, mas nenhum deles referem-se aos seus três estágios:

- Dividir: o problema é dividido em um determinando número de subproblemas que devem ser estruturalmente análogos ao problema original, mas em tamanho menor;
- Conquistar: tais subproblemas são solucionados através da estratégia recursiva de que se dispões;
- Combinar: as soluções dos subproblemas são combinadas a fim de obter uma solução do problema original.

Em [4], Fidalgo, fornece uma abordagem Dividir-e-Conquistar de modo completo tendo por resultado uma eficiência característica da própria estrutura de cada instância.

4.2.2 Dividindo com Simetrias

Valendo-se da propriedade de que alguns vértices geram simetrias, Fidalgo, [4], propõe particionar o grafo G em uma sequência $G_1, G_2, ..., G_k$ de subgrafos $G_i = (V_i, E_i, d)$, com a mesma ordem DMDGP em cada V_i, de modo que

$$G = G_1 \cup ... \cup G_k$$

Assim, nao poderá existir informações de poda envolvendo os vértices de uma determinada parte para excluir galhos infactíveis em outra parte na aplicação do BP. Para tanto o conjunto S_G, o qual corresponde à coleção dos vértices de simetria, passará por um pré-processamento dando origem ao conjunto P_G, gerador do particionamento que desejarmos, através dos seguintes passos:

- Descarte do quarto vértice v_4;
- Caso o último vértice v_n pertença a S_G, descarte-o, também;
- Tome o próximo vértice v_{s_1} e o inclua como v_{p_1} em P_G.

- Avalie quais dos próximos vértices formam uma sequência de vértices consecutivos. Caso isso ocorra, descarte todos estes vértices e tome o próximo dos vértices de simetria como v_{p_2} e o inclua em P_G.Logo, existe a possibilidade de alguns vértices de simetria não serem geradores de nenhuma das partes, mas serem apenas elementos. Esses vértices descartados são apenas elementos de uma partição, mas não são geradores.

- Caso contrário, tome v_{s_2} como v_{p_2} e o inclua em P_G.

- Procedendo assim, esgote todas as possibilidades de obter os geradores de partes.

- Acaso, o último elemento de S_G ainda não tenha sido descartado, então o inclua como o último elemento v_{p_s} de P_G.

Desta maneira, teremos o subconjunto de S_G ao qual chamaremos de *Conjunto Gerador da Partição por Simetrias*

$$P_G = \{v_{p_1}, ..., v_{p_s}\} \subseteq S_G \subset V$$

De posse desse conjunto, vamos propor uma partição de G em uma sequência de subgrafos $G_1, G_2, ..., G_k,$, com $k = s + 1$, tal que

$$G = G_1 \cup ... \cup G_k \qquad e \qquad |G_i \cap G_{i+1}| = 3$$

de forma que cada vértice gerador (de simetria) em P_G corresponderá ao quarto vértice de um dos processos, exceto do primeiro cujo quarto vértice será v_4. Todos estes passos são implementados através dos algoritmos [4], a seguir.

Algorithm 3 - O algoritmo $FindSym$

1: $FyndSym(d)$
2: Defina uma matriz D triangular superior a partir de d, fazendo $D(i,j) \leftarrow d_{i,j}$, para $j > i$;
3: Defina um vetor de zeros sym de comprimento n;
4: Como o quarto nível sempre define simetria, faça $sym(4) = 1$;
5: **para** (cada coluna $j > 4$ de D) **faça**
6: **se** (a submatriz $D(i : j-4, j : n)$ é nula) **então**
7: $sym(j) \leftarrow 1$;
8: **fim se**
9: **fim para**
10: Defina um vetor vazio de índices de simetria S;
11: $ind \leftarrow 0$;
12: **para** (j de 1 a n)**faça**
13: **se**($sym(j) = 1$)**então**
14: $S(ind) \leftarrow j$;
15: $ind \leftarrow ind + 1$;
16: **fim se**
17: **fim para**
18: **return** D e S

Algorithm 4 - O algoritmo $SymSplit$

1: $SymSplit(d)$
2: $(D, S) = FindSym(d)$;
3: $n \leftarrow c(D)$, onde $c(D)$ é o número de colunas de D.
4: Exclua o primeiro elemento de S.
5: **se**($S(|S|) = n$)**então**
6: Exclua o último elemento de S
7: **fim se**
8: Defina uma matriz $split$ com duas colunas e $|S| + 1$ linhas identicamente nula
9: $split(1, 1 : 2) \leftarrow \begin{bmatrix} 1 & S(1) \end{bmatrix}$
10: **para** (i de 2 a $|S|$)**faça**
11: $split(1, 1 : 2) \leftarrow \begin{bmatrix} split(i-1, 2) - 3 & S(1) - 1 \end{bmatrix}$
12: **fim para**
13: $split(|S| + 1, 1 : 2) \leftarrow \begin{bmatrix} split(|S|, 2) - 3 & n \end{bmatrix}$
14: **return**$split$

O primeiro passo a se dividir a instância G dada é descobrir quais vértices de V possuem a característica de definirem simetrias ao longo da estrutura. O Algoritmo 3, *FindSym*, realiza esta tarefa.

Uma vez encontrados os vértices de simetria, é necessário definir uma rotina que de fato particione completamente o grafo G em relação a estas simetrias. O

Algoritmo 4, *SymSPlit*, realiza esta tarefa utilizando os dados de distâncias e os dados de simetria provenientes da saída do Algoritmo 3.

4.2.3 Conquistando com o *SymBP*

A determinação de uma estrutura DMDGP com o algoritmo BP não é temporalmente homogênea em cada vértice, isto é, ele não determina cada vértice com o mesmo tempo. Pelo contrário, existem setores que são muito "lentos"e setores em que ele funciona muito rápido e eficientemente.

Assim, Fidalgo,[4], estudou o funcionamento do *Branch & Prune* com o intuito de determinar em quais regiões da instância, de fato, o BP demora mais tempo para conseguir realizações factíveis. Através da análise cuidadosa dos piores casos de instâncias de diversos tamanhos e estruturas ele identificou que os setores "lentos"estão intimamentes relacionados à ausência de informações de distância de poda em subconjuntos de vértices sucessivos.

Em resumo:

- para os vértices v_j que possuem alguma aresta de poda $\{i, j\} \in \mathbb{E}_p$ associada a eles, há apenas uma posição factível para cada trio de posições associadas aos vértices v_i, v_{i+1}, v_{i+2} (intersecção de quatro esferas) e, portanto, o BP determina as realizações rapidamente;

- para os vértices v_j para os quais não existe incidência de arestas de poda, o BP é bem mais lento em encontrar o caminho factível.

Como o *SymBP* baseia-se no BP, ele terá o mesmo comportamento, será lento nos mesmos setores em que o BP é lento.

Enfim, como estratégia para conquistar os subproblemas gerados pelo particionamento da molécula iremos empregar o *SymBP* em cada partição obtendo-se assim as soluções de cada parte.

4.2.4 Combinando com Rotações

Até agora, mencionamos métodos que dividem uma instância G do DMDGP em partições (sub-instâncias), tirando proveito da estrutura de simetrias inerente a ela, com o intuito de submeter cada uma destas partições ao *SymBP*.

Dessa maneira, será produzido uma sequência de sub-árvores de soluções que armazenam "pedaços"de soluções factíveis, inclusive para G, garantia dada pelo corte em vértices de simetria.

Assim, o que nos resta para completar a abordagem Dividir-e-Conquistar é realizarmos uma combinação eficiente desses *pedaços* de soluções. Para isto

aplicaremos a abordagem apresentada em [50], a qual se vale de uma translação e duas rotações para unir cada partição. A seguir, apresentamos esta estratégia. As definições, nomenclaturas e notações empregadas foram amplamente extraídas de [50].

Considere uma molécula M com n átomos, representada pelo grafo $G = (V, E, d)$ que possui uma ordem DMDGP $<$, em seus n vértices.

Assim, poderemos tratar esta molécula (instância) como uma sequência de números $1, 2, ..., n$, os índices dos vértices de V.

Definição 4.2.4.1(NUCCI [50]): Um intervalo $I = [a, b]$ de M consiste na subsequência ordenada dos átomos

$$I = \{a, a+1, ..., b-1, b\}, \qquad com \quad 1 \leq a \leq b \leq n \tag{4.7}$$

O comprimento desse intervalo corresponde a $|I| = b - a$.

Até este momento, tratamos das realizações de instâncias completas. Porém, a partir da concepção de subestruturas geradas por intervalos (partições) poderemos tratar também das instâncias ditas incompletas.

Definição (NUCCI [50]): Uma realização do intervalo $I = [a, b]$ de M é uma aplicação

$$\begin{aligned} R_{a,b} : [a, b] &\to \mathbb{R}^3 \\ u &\to R_{a,b}(u) \end{aligned} \tag{4.8}$$

Esta realização será factível se satisfizer as restrições

$$|R_{a,b}(i) - R_{a,b}(j)| = d_{i,j} \qquad \forall \{i, j\} \subset E \cap I \tag{4.9}$$

Caso contrário, é dita ser infactível.

Definição (NUCCI [50]): Uma realização $R_{a,b}$ é completa se $[a, b] = [1, n]$. Caso contrário, $R_{a,b}$ será parcial.

As realizações parciais possuem grande importância em nosso trabalho, através delas empreenderemos a união das soluções encontradas para cada partição da molécula. Segundo NUCCI [50], a idéia de trabalhar com realizações parciais foi motivada por algumas heurísticas exploradas pelo próprio autor e pela iniciativa de se paralelizar o BP realizada por MUCHERINO [34].

Assim, com o intuito de formalizarmos mais alguns conceitos, a seguir.

Dado dois intervalos $[a, x]$ e $[b, y]$ que correspondam a duas partições de uma molécula M tais que

$$a < b < x < y \tag{4.10}$$

Suponha que os dois intervalos possuam, ao menos três vértices em comum [50], o que implicará em

$$x - b \geq 2 \tag{4.11}$$

Exemplo: Suponha uma molécula composta de 12 átomos e particionada em 2 intervalos, com 3 vértices em comum. Uma configuração possível é

$$M = [1, 7] \cup [5, 12]$$

Logo, $x - b = 7 - 5 = 2$. Pois $x = 7$ e $b = 5$. Os vértices 5, 6 e 7 encontram-se em ambos os intervalos.

Para que a união das realizações parciais $R_{a,x}$ e $R'_{b,y}$ gere $R_{a,y}$, cobrindo todo o conjunto de vértices $[a, y]$, NUCCI [50] desenvolveu um esquema que em primeiro lugar determina três vértices que pertençam ao conjunto $[a, x] \cap [b, y]$ e em seguida lança mão de três transformações Euclidianas: uma translação e duas rotações.

Este esquema fixa uma das realizações e move a outra (através das transformações) de tal maneira que os vértices (pontos) em comum sejam mantidos em suas posições originais enquanto que os outros vértices da realização móvel sejam transformados de tal maneira que todos os vértices, ao final, pertençam ao mesmo sistema de coordenadas de referência.

De uma maneira mais simples: $R_{a,x}$ é mantida fixa enquanto transforma-se (move-se) a realização $R'_{b,y}$.

Definição. (FIDALGO [4]) A estrutura fixa será chamada de *Realização Base* e a estrutura a ser transformada será denominada *Realização Deslizante.*

A escolha de três pontos, i, j, k, na interseção dos dois intervalos, consistirá dos três últimos (vértices) pontos da realização base. Segue as três transformações Euclidianas, como apresentadas em [4]:

(i) **Translação:** Através desta transformação será executado a translação da realização deslizante, R'_{by}, em direção à realização base, R_{ax}, de modo que o primeiro vértice da realização deslizante, $R'_{b,y}$, coincida com o antepenúltimo da realização

base, $R_{a,x}(k)$. O vetor translação será dado por:

$$\overrightarrow{V} = R_{a,x}(i) - R'_{b,y}(i) \tag{4.12}$$

O vetor translação deverá ser aplicado a toda estrutura auxliar, a realização deslizante, fig. 4.4.

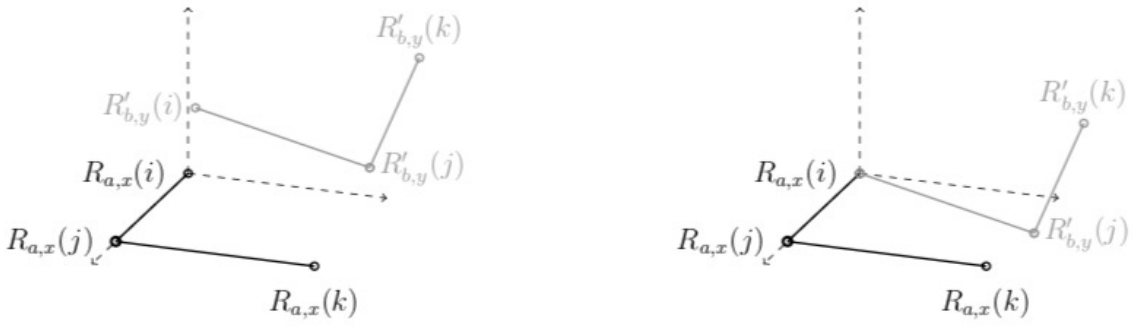


Figura 4.4: A primeira transformação aplicada à estrutura deslizante, uma translação. Fonte: FIDALGO [4].

(ii) **Primeira Rotação**

O objetivo desta transformação consiste em rotacionar os segmentos que ligam, respectivamente, os vértices $R'_{b,y}(j)$ e $R'_{b,y}(i)$ e os vértices $R_{a,x}(j)$ e $R_{a,x}(i)$, no plano que os contém, de modo que $R_{a,x}(j) \leftarrow R'_{b,y}(j)$. De fato, estes quatro vértices pertencerão ao mesmo plano pois se reduzirão a três, já que $R'_{b,y}(i) = R_{a,x}(i)$. Esses pares de pontos no $\mathbb{R}^3$, formam os vetores

$$\begin{aligned} \overrightarrow{L_{ij}} &= R_{a,x}(j) - R_{a,x}(i) \\ \overrightarrow{L'_{ij}} &= R'_{b,y}(j) - R'_{b,y}(i) \end{aligned} \tag{4.13}$$

Estes dois vetores possuem um ponto em comum, $R_{a,x}(i) = R'_{b,y}(i)$, considerando este ponto como o vértice O, haverá um ângulo θ entre eles, determinado pela lei dos cossenos como

$$\theta = \arccos\left(\frac{\left|\overrightarrow{L_{ij}}\right|^2 + \left|\overrightarrow{L'_{ij}}\right|^2 - \left|\overrightarrow{L_{ij}} - \overrightarrow{L'_{ij}}\right|^2}{2\left|\overrightarrow{L_{ij}}\right|\left|\overrightarrow{L'_{ij}}\right|}\right) \tag{4.14}$$

Assim, basta aplicar uma rotação em $R'_{b,y}$, em função de θ e cujo eixo é orientado pelo versor $\widehat{n}$, fig. 4.5,

$$\widehat{n} = \frac{\overrightarrow{L'_{ij}} \times \overrightarrow{L_{ij}}}{\left|\overrightarrow{L'_{ij}} \times \overrightarrow{L_{ij}}\right|} \tag{4.15}$$

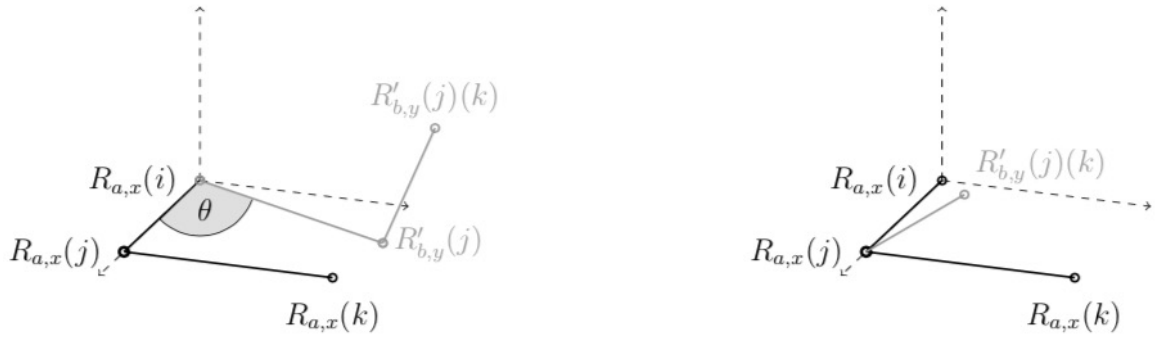


Figura 4.5: Segunda transformação aplicada à realização deslizante. Fonte: FIDALGO [4].

(iii)**Segunda Rotação:** A terceira transformação corresponde a uma segunda rotação sobre a realização deslizante, $R_{a,x} \leftarrow R'_{b,y}(k)$, sem alterar quaisquer das modificações realizadas previamente. Seja os vetores em $\mathbb{R}^3$

$$\begin{aligned} \overrightarrow{L_{jk}} &= R_{a,x}(k) - R_{a,x}(j) \\ \overrightarrow{L'_{jk}} &= R'_{b,y}(k) - R'_{b,y}(j) \end{aligned} \tag{4.16}$$

Considere a reta que passa pelo vetor $\overrightarrow{L_{i,j}}$, como direção para o eixo de rotação gerado pelo vetor unitário $\widehat{m}$, $\widehat{m}//\overrightarrow{L_{ij}}$. É claro que $\|\overrightarrow{L_{jk}}\| = \|\overrightarrow{L'_{jk}}\|$, logo as extremidades $R_{a,x}(k)$ e $R'_{b,y}(k)$ estão em um mesmo círculo contido no plano ortogonal à reta que contém $\overrightarrow{L_{ij}}$, o plano $\mathbb{P}$. Trataremos as projeções de $\overrightarrow{L_{jk}}$ e $\overrightarrow{L'_{jk}}$ sobre o plano $\mathbb{P}$ como $\overrightarrow{P_k}$ e $\overrightarrow{P'_k}$, respectivamente

$$\begin{aligned} \overrightarrow{P_k} &= M\overrightarrow{L_{jk}} \\ \overrightarrow{P'_k} &= M\overrightarrow{L'_{jk}} \end{aligned} \tag{4.17}$$

onde M corresponde à matriz de projeção no subespaço ortogonal a $\overrightarrow{L_{ij}}$,

$$M = I - \overrightarrow{L_{ij}}\overrightarrow{L_{ij}}^T \tag{4.18}$$

A realização $R'_{b,y}$ deverá ser rotacionada em torno de $\widehat{m}$ pelo ângulo φ, fig. 4.6.

$$\varphi = \arccos\left(\frac{\left|\overrightarrow{P_k}\right|^2 + \left|\overrightarrow{P'_k}\right|^2 - \left|\overrightarrow{P_k} - \overrightarrow{P'_k}\right|^2}{2\left|\overrightarrow{P_k}\right|\left|\overrightarrow{P'_k}\right|}\right) \tag{4.19}$$

φ é determinado a partir da Lei dos Cossenos.

As duas rotações são implementadas através de matrizes de rotação.

Figura 4.6: Terceira transformação aplicada à realização deslizante. Fonte: FIDALGO [4].

4.3 Paralelizando a resolução do DMDGP por *Dataflow*

Um dos objetivos de nossa pesquisa consiste em paralelizar o tratamento aplicado a uma instância do DMDGP, valendo-se das idéias apresentadas nas seções e capítulos, anteriores deste trabalho. Em grande parte, estas idéias basicamente são provenientes de uma grande pesquisa realizada por NUCCI[50] , FIDALGO[4] e LAVOR [34].

Pretendemos tomar uma instância DMDGP, que simula dados reais [44] oriundos da RMN, e dividi-la em partições, os cortes serão feitos segundo os pontos de simetria existentes na estrutura. Após isto, submeteremos cada partição ao método *SymBP*, uma vez que todas as soluções de cada partição tenham sido encontradas realizaremos a união das partes, de todas as soluções, empregando o método descrito na seção 4.2.4, uma translação e duas rotações.

Empregaremos neste trabalho a biblioteca *Dataflow Sucuri* como base para implementarmos o paralelismo no DMDGP.

Como apresentado no capítulo 3, a Sucuri é uma biblioteca minimalista para a programação *Dataflow* em *Python*, onde o desenvolvedor implementa as funções do seu programa e as associa aos objetos *Node* (nó) os quais são conectados através de arestas, de acordo com as suas dependências de dados. De maneira geral, se duas funções não tiverem dependências entre elas, estas serão potencialmente executadas em paralelo.

A figura 4.7 apresenta o tratamento (o grafo *Dataflow*) que elaboramos para a paralelização do DMDGP valendo-se da biblioteca *Sucuri* (o código fonte do programa encontra-se na seção A.1).

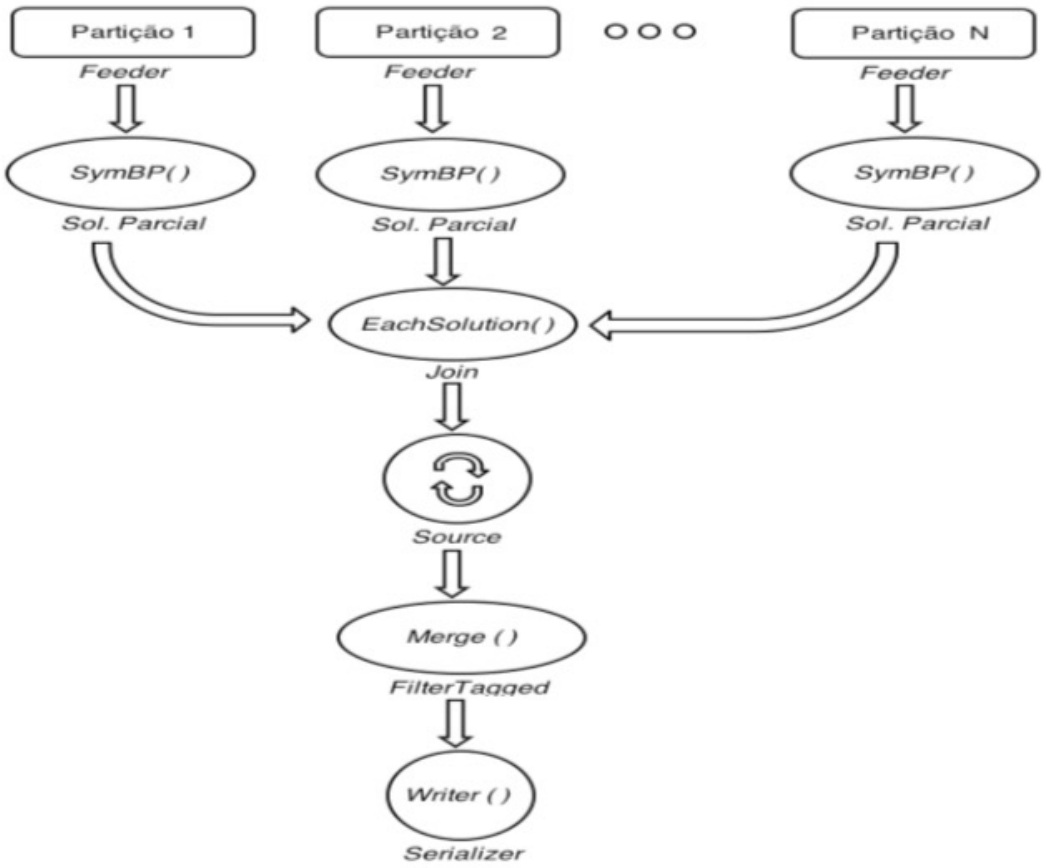


Figura 4.7: Grafo empregado na paralelização por *Dataflow* do DMDGP.

Uma vez que a molécula tenha sido particionada em função da estrutura dos vértices de simetria, as informações referentes a cada partição serão atribuidas aos nós *Feeders* (nós especializados cuja única função é repassar o operando recebido para os próximos nós, no grafo) os quais as repassarão aos nós que executam a rotina *SymBP*, obtendo-se assim a solução de cada uma das partições, as soluções parciais.

Todas as soluções parciais são passadas a um único nó, este seleciona as soluções a serem unidas e as repassam a um nó especializado, *Source*, que atuará sobre um objeto *Python* iterável. A execução do método *run* deste nó durará até que o conteúdo iterável recebido se esgote. Assim, ele repassará ao próximo nó, a cada vez, o conjunto das partes de cada solução. Os dados produzidos pelo nó *Source* serão consumidos por um nó especializado, *FilterTagged*, que realizará a união das partes de cada solução passada. Como os dados podem ser processados fora de ordem, pelo nó *Source*, o último nó deverá reordená-los, o nó especializado *Serializer*. E assim, obtêm-se, no último nó, o conjunto das soluções do DMDGP.

4.4 Resultados Computacionais Experimentais

4.4.1 Experimentos

Nesta seção apresentaremos os experimentos computacionais realizados com o objetivo de testar a abordagem empregada para paralelizar o DMDGP por *Dataflow* proposta nesta tese. As instâncias DMDGP empregadas foram artificialmente geradas, baseadas no trabalho de [51], as chamadas *Lavor Instances*.

Os experimentos computacionais foram realizados em um computador *multicore* com as seguintes especifações:

- Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz
- 16 *cores* (hyper-threading off)

O algoritmo paralelo por *Dataflow* proposto, o chamaremos de *SymBP Paralelo*, foi implementado na linguagem de programação *Python* 2.7.

A qualidade de cada solução encontrada foi determinada através da medida *Largest Distance Error (LDE)*, que consiste na função de penalidade mais empregada na avaliação de soluções para problemas moleculares de Geometria de Distâncias como um problema de otimização [52]. O LDE é definido como:

$$LDE = \frac{1}{|E|} \sum_{(i,j)\in E} \frac{||x_i - x_j| - d_{ij}|}{d_{ij}} \tag{4.20}$$

onde E corresponde ao conjunto de todas as distâncias conhecidas. Quanto menor o valor de LDE melhor será a qualidade da solução encontrada.

Nos experimentos foram empregadas instâncias DMDGP (seção 4.4.2) artificialmente geradas com n=100, 200, 300 e 500 vértices (átomos).

Cada instância foi submetida 10 vezes à rotina *SymBP Paralelo*, com o intuito de obtermos um comportamento médio, para cada conjunto de *cores* empregado.

Os resultados destes testes (seção 4.4.3) serão apresentados por meio de 2 tabelas e 1 gráfico, para cada instância selecionada, como segue detalhado a seguir:

- a primeira tabela possui como objetivo apresentar o tempo de execução consumido pelo *SymBP Paralelo* para encontrar todas as soluções de cada uma das partições da molécula, o número de soluções encontradas e a quantidade de átomos de cada partição.

- a segunda tabela apresenta, na segunda coluna, o tempo consumido pela versão serial do *SymBP* no ataque a toda a molécula, (na terceira coluna) o tempo total consumido pela nossa abordagem em paralelo (empregando 16 *cores*) ao tratar toda a molécula (englobando a fase de união das soluções de todas as partições), na quarta coluna encontra-se o tempo consumido apenas na fase de união das soluções (o *Merge*) e por final, apresentamos o *Speedup Máximo* alcançado e o LDE da *melhor solução* encontrada.

- os gráficos apresentam o tempo de Execução para se tratar toda a molécula, versão paralela, em função da quantidade de *cores* empregados e o *Speedup* em função do número de *cores*. Vale lembrar que ao submeter o *SymBP Paralelo* a execução de um *core* apenas, a rotina reduz-se à sua versão Serial.

4.4.2 Geração das Instâncias *DMDGPs*

Em nossos experimentos a geração de instâncias para o DMDGP envolveu a criação de estruturas artificiais que satisfazem as hipóteses impostas ao MDGP (seção 2.3.1), para isto nos baseamos no trabalho de [51].

As instâncias DMDGP baseiam-se em um modelo inicialmente proposto por PHILIPS [53]. Este modelo considera uma molécula como uma cadeia de N átomos, cujas coordenadas cartesianas são dadas por $x_1, ..., x_N \in \mathbb{R}^3$. Vale lembrar que empregamos esta abordagem para modelar o DMDGP (capítulo 2), onde consideramos que para todo par de átomos consecutivos (i, j) teremos $d_{i,j}$, o comprimento de ligação (corresponde à distância Euclidiana entre eles) e o ângulo de ligação $\theta_{i,j}$.

Nos cálculos de todas as estruturas moleculares assumimos que todos os comprimentos de ligações e ângulos de ligação são fixos, em seus valores de equilíbrio ($d_{i,j} = 1,526$ Å; $\theta_{i,j} = 109,5°$) [51]. Com estas informações e valendo-se da teoria

descrita no capítulo 2 (desta tese), poderemos posicionar os três primeiros átomos da cadeia molecular: o primeiro será fixo em, $x_1 = (0, 0, 0)$, o segundo será posicionado em $x_2 = (-d_{1,2}, 0, 0)$ e o terceiro será em $x_3 = (-d_{1,2} + d_{2,3} \cos\theta_{1,3}, d_{2,3} \sin\theta_{1,3}, 0)$. O quarto átomo será determinado através do ângulo de torção $\omega_{1,4}$; o quinto átomo será definido pelos ângulos de torção $\omega_{1,4}$ e $\omega_{2,5}$; o sexto átomo pelos ângulos de torção $\omega_{1,4}$, $\omega_{2,5}$ e $\omega_{3,6}$; e assim por diante, de maneira iterativa segundo o emprego das equações 2.5, 2.6, 2.7 e 2.8.

Os ângulos de torção que usamos correspondem às perturbações dos ângulos, preferenciais [51], 60°, 180° e 300°. Assim, iremos gerar cada ângulo de torção, de uma molécula, através da seleção de um valor ω a partir do conjunto de ângulos Ω,

$$\Omega = \{60°, 180°, 300°\} \tag{4.21}$$

somando-o a outro a partir do conjunto

$$\{\omega + i : i = -15°, ..., 15°\}. \tag{4.22}$$

Ambas, as escolhas, são aleatórias.

Assim, ao final destes procedimentos, obtemos uma estrutura molecular (que satisfaz o DMDGP) e através desta geraremos, agora, uma instância típica do DMDGP: uma vez de posse da estrutura molecular construiremos um conjunto que contenha todas as distâncias entre os átomos. Em seguida, realizaremos uma poda neste conjunto, valor de corte igual a 4 Ångström (Å) (as distâncias abaixo desse valor serão consideradas como disponíveis no problema e as distâncias acima desse valor foram tratadas como indisponíveis). Depois de aplicado este corte, o novo conjunto de distâncias obtido tornará se o *input* de nosso programa, as chamadas *Lavor Instances*.

Em nossos experimentos cada instância DMDGP empregada foi designada pela palavra *lavor* seguida pela quantidade de átomos. Exemplo: uma instância DMDGP com 50 átomos será intitulada *lavor50*.

4.4.3 Resultados

Instância 1: $n = 100$ átomos

A primeira instância empregada possui $n = 100$ vértices e a chamamos de *lavor100a*, figura 4.8. Esta instância possui 10 vértices de simetria, $S_G = \{4, 42, 43, 44, 45, 49, 50, 51, 88, 89\}$, o que implicará na geração de 4 partições, G_1, G_2, G_3 e G_4, as quais possuirão como índices inferiores e superiores as linhas da matriz *splitlavor100*, a seguir:

$$splitlavor100 = \begin{bmatrix} 1 & 41 \\ 39 & 48 \\ 46 & 87 \\ 85 & 100 \end{bmatrix} \tag{4.23}$$

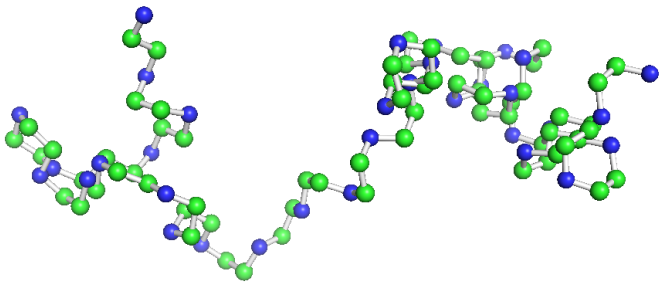


Figura 4.8: Instância empregada, *lavor100a*, 100 átomos.

Na Tabela 4.1 temos os tempos de execução médios encontrados ao submeter a instância *lavor100a* ao *SymBP Paralelo* empregando-se 16 *cores*, em específico: os tempos gastos no encontro das soluções de cada partição da instância.

Partições	Qtd. Átomos	Qtd. Sol.	Tempo Exec.[s]
Part 0	41	2	$0,0233$
Part 1	10	16	$0,0091$
Part 2	42	8	$0,0328$
Part 3	16	4	$0,0093$

Tabela 4.1: Tempo médio, em segundos, de execução gastos na determinação das soluções de cada uma das partições da instância *lavor100a* empregando 16 *cores*.

Na Tabela 4.2 encontramos os resultados referentes aos tempos de execução das versões paralela e serial, o *Speedup Máximo* alcançado e o menor LDE encontrado. Através das figuras 4.9 e 4.10 podemos observar a redução no tempo de execução em função da quantidade de *cores* empregado.

Instância	T. Serial [s]	T. Paralelo[s]	Merge [s]	*Speedup M.*	LDE
lavor100a	$3,5146$	$0,5879$	$0,4283$	$5,97$	$2,68 \times 10^{-14}$

Tabela 4.2: Dados referentes à instância *lavor100a*.

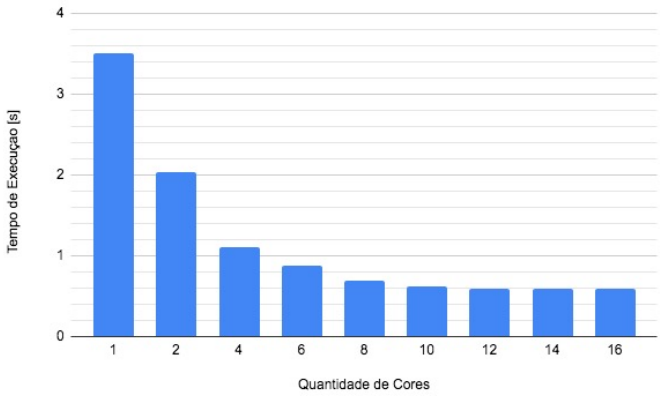


Figura 4.9: Tempo de Execução[s] *versus* Quantidade de *cores*, instância *lavor100a*, 100 átomos.

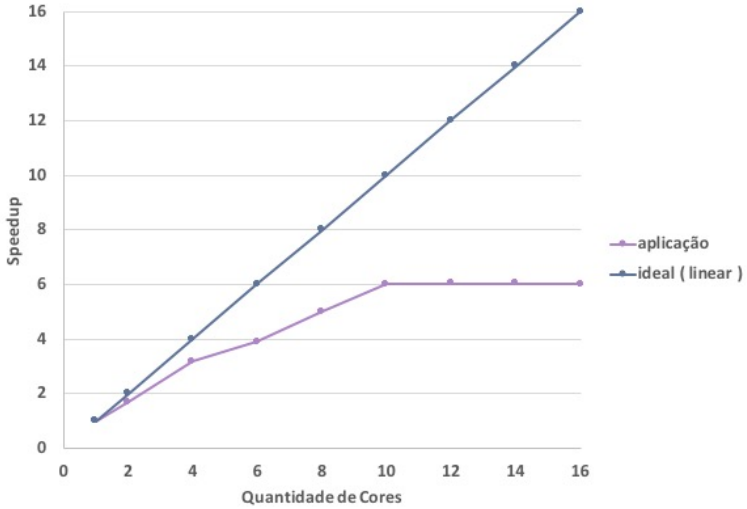


Figura 4.10: *Speedup versus* Quantidade de *cores*, instância *lavor100a*.

Instância 2: $n = 200$ átomos

A segunda instância testada possui $n = 200$ vértices e foi chamada de *lavor200a*, figura 4.10. Ela possui 16 vértices de simetria o que implicará em $2^{16} = 65536$ soluções a serem encontradas, $S_G = \{4, 122, 123, 124, 125, 127, 128, 130, 131, 155, 156, 157, 161, 162, 163, 164\}$, a partir do conjunto de vértices . A molécula foi dividida em 6 partições:

$$splitlavor200 = \begin{bmatrix} 1 & 121 \\ 119 & 126 \\ 124 & 129 \\ 127 & 154 \\ 152 & 160 \\ 158 & 200 \end{bmatrix} \tag{4.24}$$

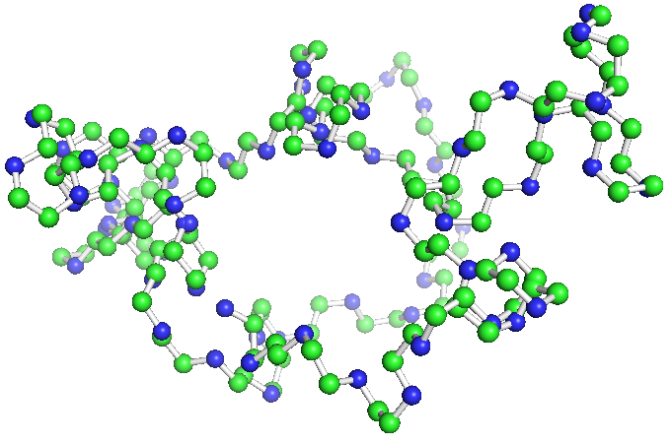


Figura 4.11: Instância *lavor200a*, 200 átomos.

Na Tabela 4.3 apresentamos os valores encontrados para o tempo de execução da rotina *SymBP Paralelo* para resolver para cada partição da molécula, em questão:

Partições	Qtd. Átomos	Qtd. Sol.	Tempo Exec.[s]
Part 0	121	2	$0,9741$
Part 1	8	16	$0,0055$
Part 2	6	4	$0,0007$
Part 3	28	4	$0,0123$
Part 4	9	8	$0,0021$
Part 5	43	16	$0,6868$

Tabela 4.3: Tempo médio, em segundos, de execução gastos na determinação das soluções de cada uma das partições da instância *lavor200a* empregando 16 *cores*.

Na Tabela 4.4 encontramos os resultados referentes aos tempos de execução das versões serial e paralela, o *Speedup Máximo* alcançado e o menor LDE encontrado, nas fig. 4.12 e 4.13 podemos observar a redução no tempo de processamento em função da quantidade de *cores* empregada para a instância *lavor200a*.

Instância	T. Serial [s]	T. Paralelo[s]	Merge [s]	*Speedup M.*	LDE
lavor200a	$308,4066$	$44,6471$	$42,6259$	$6,91$	$1,94 \times 10^{-14}$

Tabela 4.4: Dados referentes à instância *lavor200a*.

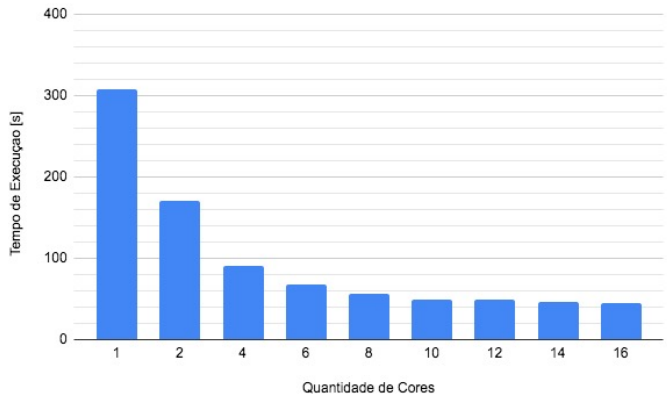


Figura 4.12: Tempo de Execução[s] *versus* Quantidade de *cores*, instância *lavor200a*, 200 átomos.

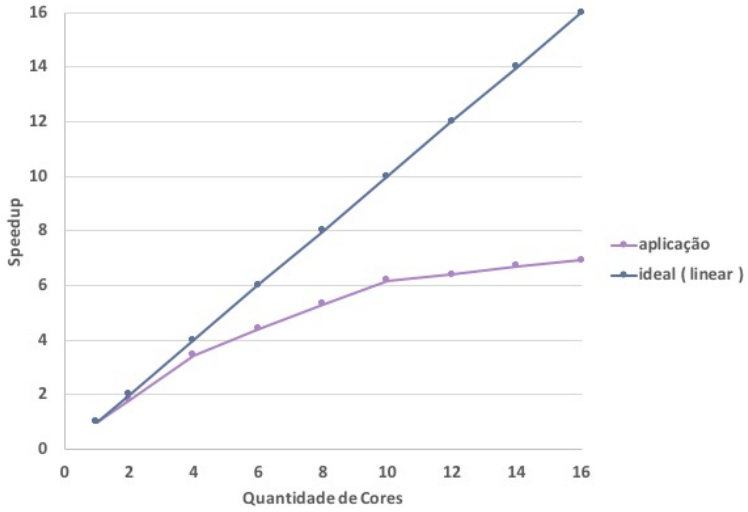


Figura 4.13: *Speedup versus* Quantidade de *cores*, instância *lavor200a*.

Instância 3: $n = 300$ átomos

A terceira instância tratada possui $n = 300$ átomos e a chamamos de *lavor300a*, figura 4.12. Esta instância possui 11 vértices de simetria, logo, pretendemos encontrar 2048 soluções. O conjunto de vértices de simetria, para a instância em questão, encontrado foi: $S_G = \{4, 103, 104, 118, 119, 120, 121, 126, 128, 299, 300\}$, o que implicou na geração de 6 partições, as quais possuem como índices inferiores e superiores as linhas da matriz *splitlavor300*, a seguir:

$$splitlavor100 = \begin{bmatrix} 1 & 102 \\ 100 & 117 \\ 115 & 125 \\ 123 & 127 \\ 125 & 298 \\ 296 & 300 \end{bmatrix} \tag{4.25}$$

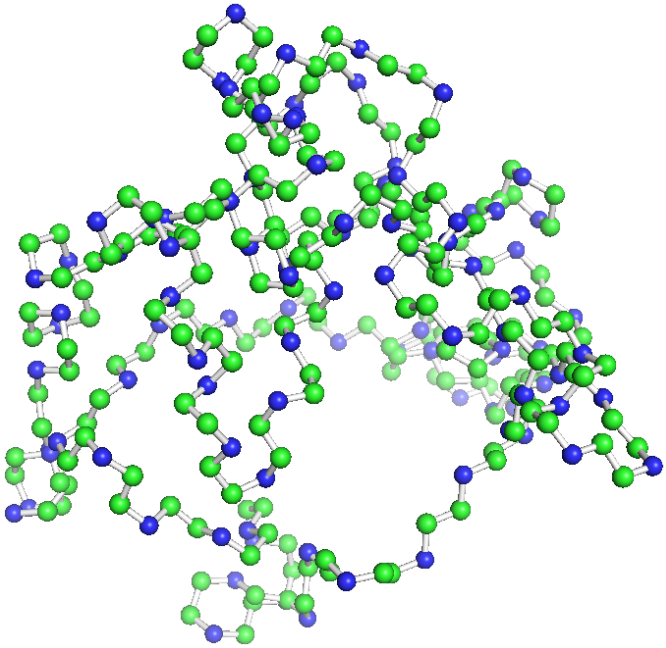


Figura 4.14: Instância empregada, *lavor300a*, 300 átomos.

Na Tabela 4.5 temos os tempos de execução médios encontrados ao submeter a instância *lavor300a* ao *SymBP Paralelo*, empregando-se 16 *cores*, em específico: os tempos gastos no encontro das soluções de cada partição da instância.

Partições	Qtd. Átomos	Qtd. Sol.	Tempo Exec.[s]
Part 0	102	2	$0,1368$
Part 1	18	4	$0,0078$
Part 2	11	16	$0,0093$
Part 3	5	2	$0,0012$
Part 4	174	2	$158,4394$
Part 5	5	4	$0,0013$

Tabela 4.5: Tempo médio, em segundos, gastos na determinação das soluções de cada uma das partições da instância *lavor300a* empregando 16 *cores*.

Instância	T. Serial [s]	T. Paralelo[s]	Merge [s]	*Speedup M.*	LDE
lavor300a	$179,86$	$160,1657$	$1,2263$	$1,13$	$4,97 \times 10^{-15}$

Tabela 4.6: Dados referentes à instância *lavor300a*.

Na fig. 4.15 temos o gráfico Tempo de Exec. *versus* Qtd. de *cores* para a *lavor300a*, na fig. 4.16 o ganho em função do número de *cores* e na tabela 4.6 os dados experimentais da instância *lavor300a*.

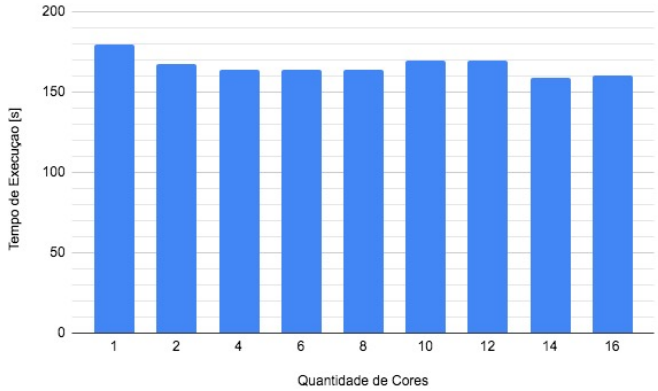


Figura 4.15: Tempo de Execução[s] *versus* Quantidade de *cores*, instância *lavor300a*, 300 átomos.

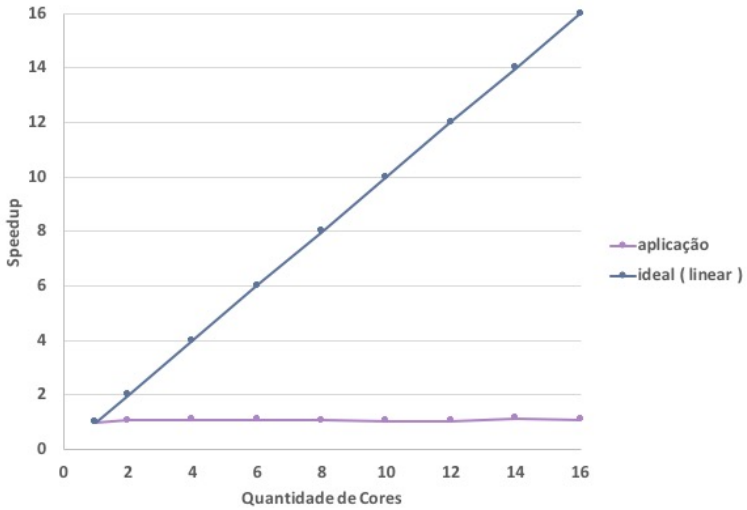


Figura 4.16: *Speedup versus* Quantidade de *cores*, instância *lavor300a*.

Instância 4: $n = 500$ átomos

A quarta instância testada possui $n = 500$ vértices e foi chamada de *lavor500a*, figura 4.14. Ela possui 8 vértices de simetria o que implicará em 256 soluções a serem encontradas. A molécula foi dividida em 6 partições:

$$splitlavor500 = \begin{bmatrix} 1 & 4 \\ 2 & 194 \\ 192 & 196 \\ 194 & 269 \\ 267 & 369 \\ 367 & 500 \end{bmatrix} \tag{4.26}$$

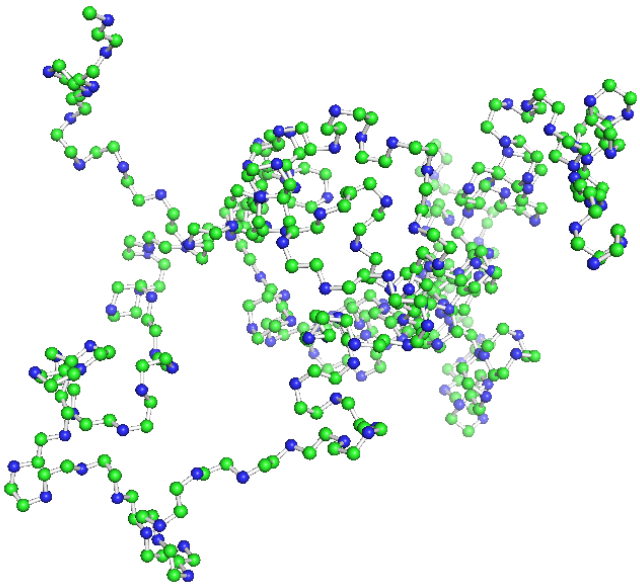


Figura 4.17: Instância *lavor500a*, 500 átomos.

Na Tabela 4.7 apresentamos os valores encontrados para o tempo de processamento de cada partição da molécula, em questão:

Partições	Qtd. Átomos	Qtd. Sol.	Tempo Exec.[s]
Part 0	4	2	$0,0012$
Part 1	193	16	$1,7227$
Part 2	5	2	$0,0019$
Part 3	76	16	$0,1522$
Part 4	103	4	$0,4000$
Part 5	134	16	$1,9138$

Tabela 4.7: Tempo médio, em segundos, de execução gastos na determinação das soluções de cada uma das partições da instância *lavor500a* empregando 16 *cores*.

Na Tabela 4.8 temos os resultados experimentais para a instância *lavor500a*, na fig. 4.18 o seu gráfico T. de Exec. *versus* Qtd. de *cores* e na fig. 4.19 o gráfico *speedup* x *cores*.

Instância	T. Serial [s]	T. Paralelo[s]	Merge [s]	*Speedup M.*	LDE
lavor500a	$2397,3108$	$198,4715$	$196,7715$	$12,08$	$3,74 \times 10^{-14}$

Tabela 4.8: Dados referentes à instância *lavor500a*.

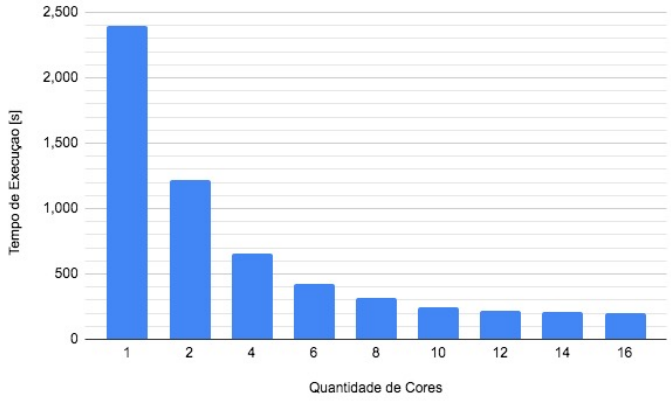


Figura 4.18: Tempo de Execução[s] *versus* Quantidade de *cores*, instância *lavor500a*, 500 átomos.

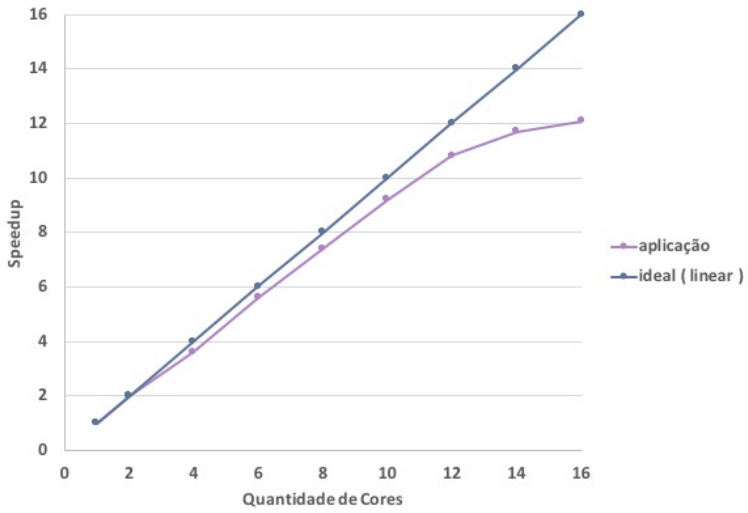


Figura 4.19: *Speedup versus* Quantidade de *cores*, instância *lavor500a*.

4.4.4 Análise

Da análise dos resultados encontrados podemos observar uma redução no tempo de execução da abordagem em paralelo para a maioria dos casos, notadamente a instância *lavor500* (*speedup*= 12.08). Porém, para as outras instâncias podemos afirmar que o nosso código "não escala bem". Isto se deve a alguns fatores:

- as partições são desbalanceadas na maioria das vezes, pois são determinadas em função de características intrínsicas da instância molecular (os vértices de simetria) o que impacta na minimização do tempo em que os *cores* não estão "ocupados", afetando o desempenho global de todo o programa. A instância *lavor300* é um exemplo extremo deste caso, o tempo de processamento de uma das suas partições ocupou todo o tempo de execução da aplicação.

- a fase do *Merge*, união das soluções parciais, pode em alguns casos chegar a consumir mais de 90% do tempo de execução total do programa, vide a instância *lavor200a* com $95,4\%$ do tempo dedicado à fase do *Merge*. Uma vez que ela possui 65536 (soluções) processos (na fase do *Merge*) a serem alocados em alguns *cores*, o gerenciamento desta quantidade de processos demandará um alto custo para a nossa aplicação, sem contar no tempo total necessário para a união das soluções parciais de todas as soluções. Se a quantidade de processos envolvidos na fase do *Merge* não for significativa isto não será um problema, vide a instância *lavor500a*, com (99,1%) do tempo dedicado à fase do *Merge* porém com apenas 256 (soluções) processos.

Assim, podemos concluir que o emprego dos vértices de simetria como pontos de partida para o particionamento de moléculas no tratamento do DMDGP não é recomendado devido gerar partições altamente desbalanceadas. A fase de união das soluções parciais, através do emprego de matrizes de rotação, mostrou-se custosa na maioria dos casos.

Por último, vale ressaltar, que a qualidade das soluções obtidas (baixo valor para o LDE) e os *speedups* alcançados mostram a efetividade da abordagem empregada para a maioria dos casos.

Capítulo 5

Problema de Geometria de Distâncias Moleculares Intervalares

5.1 Introdução

Como visto anteriormente, o DGP consiste na determinação das coordenadas de um determinado conjunto de pontos, a partir de algumas distâncias conhecidas entre os pares desses pontos [31]. Quando as distâncias entre alguns destes são dadas por intervalos, temos a versão intervalar do problema. Para o *DGP* relacionado a proteínas e empregando-se distâncias intervalares têm se o *Interval Discretizable Molecular Distance Geometry Problem (iDMDGP)*, proposto em LAVOR *et al.* [9]. Neste capítulo iremos tratar da aplicação do *iDMDGP* na reconstrução de estruturas tridimensionais de proteínas, através do emprego de dados (intervalares) que simulam medidas experimentais obtidas por Ressonância Magnética Nuclear. Empregaremos, para isto, uma abordagem paralela para o *iDMDGP*, baseada no modelo de execução *Dataflow*, o qual, tem se mostrado uma boa opção para a programação paralela [16].

5.2 Uma abordagem combinatória

Uma adaptação do algoritmo *Branch and Prune*, BP, para distâncias intervalares foi proposto em [9], onde os intervalos são substituidos por um conjunto finito de pontos discretos. Nos referimos a esta adaptação do BP como *interval BP (iBP)*. Quando todas as distâncias disponíveis são exatas, todas as posições candidatas de uma dada conformação molecular podem ser enumeradas. Isto, no entanto não é possível na presença de distâncias intervalares, pois um subconjunto "contínuo"de

posições poderá ser computado para alguns átomos.

Seja $G = (\mathbb{V}, \mathbb{E}, d)$, um grafo não-dirigido e ponderado, representando uma instância *iDMDGP*. Em seguida, considere o conjunto $\mathbb{V}$ como a representação dos átomos de uma dada molécula e $\{u, v\} \in \mathbb{E}$, se a distância entre os átomos u e v estiverem a disposição. O conjunto d relaciona cada aresta, $\{u, v\} \in \mathbb{E}$, a um intervalo $[\underline{d}_{u,v}, \overline{d}_{u,v}]$. Assim, podemos afirmar que o *iDMDGP* possui como objetivo encontrar uma conformação molecular em um espaço tridimensional tal que:

$$\underline{d}_{u,v} \leq |x_u - x_v| \leq \overline{d}_{u,v}, \quad \forall \{u, v\} \in \mathbb{E}. \tag{5.1}$$

Observe que $\underline{d}_{u,v}$ e $\overline{d}_{u,v}$ designam, respectivamente, os limites inferior e superior para a distância $d_{u,v}$, quando $d_{u,v}$ é exato têm se $\underline{d}_{u,v} = \overline{d}_{u,v}$.

Com base na geometria das proteínas, é possível definir uma ordem sobre os átomos onde são conhecidas as distâncias $d_{i-1,i}, i = 2, ..., n$ e $d_{i-2,i}, i = 3, ..., n$, que são precisas, e as distâncias $d_{i-3,i}, i = 4, ..., n$, que podem ser provenientes da RMN, representadas por intervalos $[\underline{d}_{i-3,i}, \overline{d}_{i-3,i}]$. O fato das distâncias $d_{i-3,i}$ não serem mais valores precisos significa que não teremos mais a interseção de três esferas, gerando dois pontos (fig. 5.1 A). Quando uma das esferas tem raio intervalar, obtemos dois arcos na circunferência dada pela interseção das duas esferas de raios precisos, como o apresentado nas figuras fig. 5.1 b e fig. 5.2.

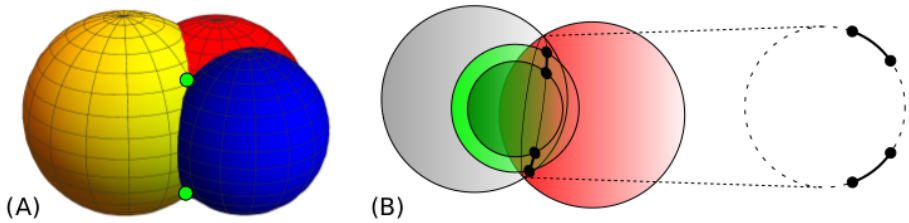


Figura 5.1: (A) Com três distâncias exatas, a interseção é dada por dois pontos. (B) Com duas distâncias exatas e uma intervalar, a interseção é dada por dois arcos. Fonte: LAVOR *et al* [8].

Uma abordagem simples para poder continuar aplicando a ideia do BP, agora chamado de *iBP* por conta das distâncias intervalares, seria a de selecionar algumas amostras do intervalo $[\underline{d}_{i-3,i}, \overline{d}_{i-3,i}]$. Nesse caso, cada distância escolhida geraria dois pontos em $\mathbb{R}^3$. O primeiro problema é que a árvore de busca do *iBP* deixaria de ser binária, aumentando exponencialmente de tamanho. Porém, a conseguência drástica é que o *iBP* poderia varrer toda a árvore e não encontrar uma solução. Ou seja, deixaria de ser um método exato, tornando-se uma heurística.

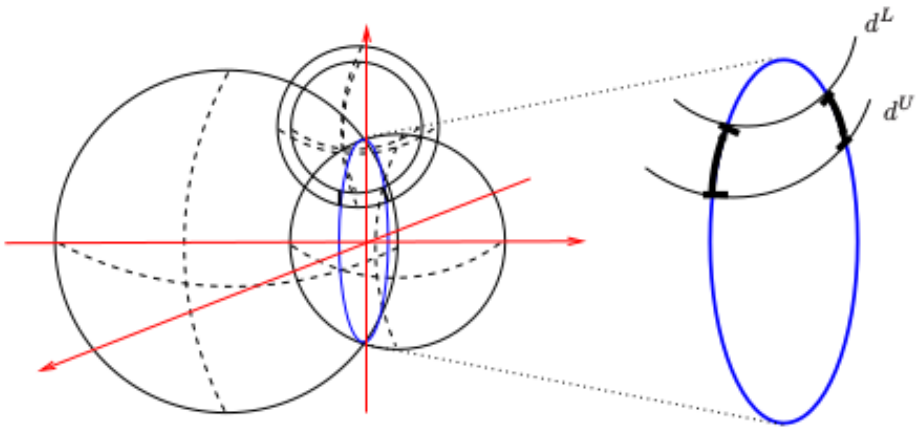


Figura 5.2: A interseção de duas esferas com uma concha esférica. Fonte: LAVOR *et al* [9].

Em geral o MDGP é formulado como um problema de otimização contínuo onde a função objetivo consiste em uma função "penalidade"capaz de medir a violação das restrições. Sob certas considerações, o domínio deste problema de otimização pode ser discretizado podendo-se aplicar uma abordagem combinatória [9], a nossa estratégia consistirá em aplicar isto ao *iDMDGP*.

5.3 O algoritmo *iBP*

Seja $G = (\mathbb{V}, \mathbb{E}, d)$, uma instância *iDMDGP*. A subclasse de instâncias *iDMDGP* que deveremos considerar é definida a seguir, como um caso particular do $iDDGP_3$ *(interval Discretizable DGP in dimension 3)*.

Definição 5.1: (GONÇALVES [10]) O $iDDGP_3$ é caracterizado por:
Dado um grafo não-dirigido ponderado $G = (\mathbb{V}, \mathbb{E}, d)$, onde $\mathbb{E}' \subset \mathbb{E}$ consiste no subconjunto de arestas para as quais os pesos d representam distâncias exatas, diremos que G representa uma instância do $iDDGP_3$ se e somente se existir uma ordem para os vértices de $\mathbb{V}$ que verifique as seguintes condições:

(a) $G_C = (\mathbb{V}_C, \mathbb{E}_C) \equiv \mathbb{G}[\{1, 2, 3\}]$ é um *clique* e $\mathbb{E}_C \subset \mathbb{E}'$;

(b) $\forall i \in \{4, ..., |V|\}$, existirá $\{i', i'', i'''\}$ tais que

1. $i''' < i, i'' < i, i' < i$;
2. $\{(i'', i), (i', i)\} \subset \mathbb{E}'$ e $(i''', i) \in \mathbb{E}$;
3. $d_{i',i'''} < d_{i',i''} + d_{i'',i'''}$.

Ordens que satisfaçam (a) e (b) são chamadas de *Ordens de Discretização*. Os átomos do conjunto $\{i''', i'', i'\}$ são chamados de *átomos de referência* e as $d_{i''',i}, d_{i'',i}, d_{i',i}$, *distâncias de referência*.

A condição (a) permite fixarmos os 3 primeiros átomos de maneira única, evitando que consideremos soluções congruentes as quais podem ser obtidas através de rotações e translações. A suposição (b1) assegura a existência de três átomos de referência para todo $i > 3$, e a suposição (b2) garante que ao menos uma das três distâncias de referência deverá representar um intervalo. Finalmente, a condição (b3) previne (evita) que os átomos de referência sejam colineares. Sob estas condições, o *iDMDGP* poderá ser discretizado, ou seja, a instância pertencerá a classe de instâncias $iDDGP_3$. Neste caso, o domínio de pesquisa (o espaço de busca) será uma árvore, onde os nós conterão as posições atômicas candidatas organizadas, camada por camada.

Algorithm 5 O algoritmo *iBP*

```
iBP(i, n, d, D)
if (i > n) then
   //uma solução foi encontrada
   print  a solução encontrada
else
   //computação das coordenadas
   if (d_{i''',i} é 1 intervalo) then
      Selecione D amostras (distâncias) existentes no intervalo;
      Para cada uma das D amostras compute as 2 posições
      candidatas associadas;
      Adicione as posições candidatas computadas à lista L;
   else
      //temos d_{i''',i}, uma distância exata;
      Compute as 2 posições candidatas associadas a d_{i''',i};
      Adicione as 2 posições candidatas encontradas à lista L;
   end if
   //verificando a viabilidade das posições computadas
   for K ∈ {1, ..., |L|} do
      if (x_i^k é viável) then
         iBP(i + 1, n, d, D)
      end if
   end for
end if
```

Nós empregamos um algoritmo *interval Branch & Prune iBP* [9] para encontrar as soluções associadas às instâncias discretizáveis, o alg.5 consiste em um esboço deste algoritmo. O algoritmo *iBP* realiza uma pesquisa recursiva pela árvore de busca, a qual representa o domínio da pesquisa. A cada chamada recursiva, posições candidatas para o átomo atual são computadas através da exploração das coordenadas dos átomos previamente alocados. Quando todas as distâncias de referência são exatas, então duas posições atômicas são computadas. Quando uma das distâncias de referência consiste em um intervalo computamos 2D posições atômicas, para o

átomo em questão.

A cada chamada do algoritmo, *"i"* será o átomo corrente para o qual as posições candidatas serão pesquisadas, *"n"* corresponderá ao número total de átomos que formam a molécula considerada, *"d"* será a lista de distancias disponíveis, exatas e intervalares. *"D"* corresponderá ao fator de discretização, número de amostras que serão selecionadas do intervalo de distânciais em questão ($d_{i''',i}$), condição (b2). No algoritmo *iBP* (linhas 11 e 15) empregamos uma lista L de posições, de onde as posições candidatas serão extraídas.

Para toda nova posição computada cria-se um novo ramo da árvore, no *iBP* esta fase é chamada de *"branching phase"*. Para toda nova posição encontrada a sua viabilidade é verificada através de um teste onde se confirma se ela satisfaz as condições (restrições) impostas (b2), esta fase é conhecida como *"pruning phase"*.

5.4 Computando as coordenadas das posições candidatas

O método empregado para computar as posições candidatas para cada chamada recursiva do algoritmo *iBP* possui uma importância fundamental. Enquanto pesquisamos as posições atômicas candidatas para o átomo i consideramos que os átomos de referência $\{i''', i'', i'\}$ já se encontram posicionados. Estes 3 átomos definem um sistema de coordenadas local centrado em i' [10].

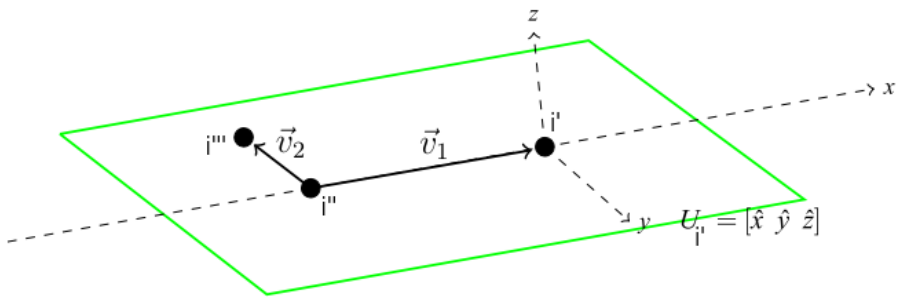


Figura 5.3: Os vértices de referência *i', i"* e *i"'* induzem um sistema de coordenadas. Fonte: GONÇALVES [10].

Seja $\vec{v}_1$ o vetor de i'' a i' e $\vec{v}_2$ o vetor de i'' a i'''. O eixo x para o sistema em i' poderá ser definido por $\vec{v}_1$, o vetor unitário nesta direção será $\widehat{x} = \frac{\overrightarrow{v_1}}{\|\overrightarrow{v_1}\|}$. Além disso, o produto vetorial $\overrightarrow{v_1} \times \overrightarrow{v_2}$ produzirá outro vetor que definirá o eixo z, cujo vetor unitário será $\widehat{z}$. Finalmente, o produto vetorial $\widehat{x} \times \widehat{z}$ gerará o vetor que definirá o eixo y (vetor unitário $\widehat{y}$).

Estes três vetores unitários são as colunas da matriz $U_{i'} = [\widehat{x}\ \widehat{y}\ \widehat{z}]$, cujo papel reside em converter diretamente as posições dos vértices do sistema de coordenadas

definido em i' para o sistema canônico. De uma outra maneira, $U_{i'}$ é uma matriz de rotações (mudança de base) do sistema local em i' para o sistema canônico de coordenadas [10].

Uma vez que a matriz $U_{i'}$ tenha sido computada, as coordenadas canônicas cartesianas para uma posição candidata para o átomo i pode ser obtida por:

$$x_i(\omega_i) = x_{i'} + U_{i'} \begin{bmatrix} -d_{i',i} cos\,(\theta_i) \\ d_{i',i} sin\,(\theta_i)\, cos\,(\omega_i) \\ d_{i',i} sin\,(\theta_i)\, sin\,(\omega_i) \end{bmatrix} \tag{5.2}$$

onde, θ_i e ω_i são os ângulos relacionados às coordenadas esféricas do átomo i.

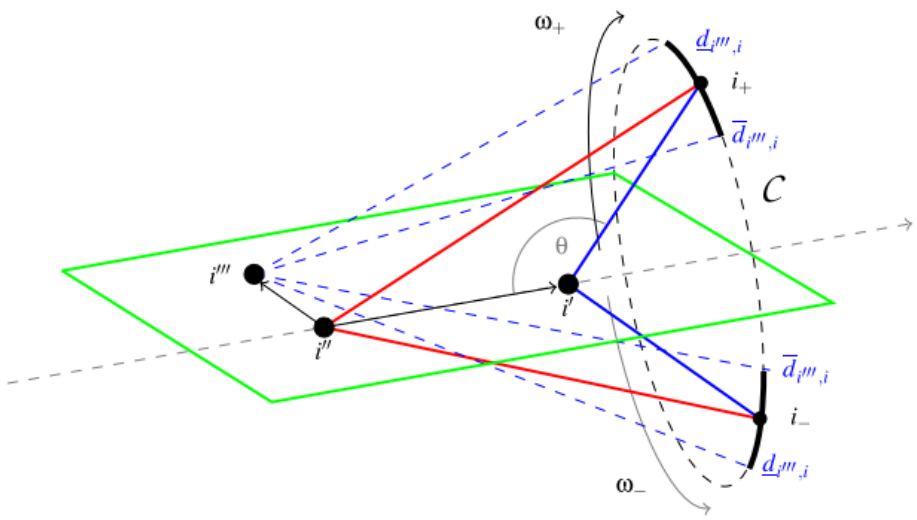


Figura 5.4: Os dois arcos obtidos pela interseção de duas esferas e uma concha esférica. Fonte: GONÇALVES [10].

Como podemos observar, de maneira mais detalhada, pela fig. 5.4 :

- θ_i: ângulo formado pelos segmentos (i, i') e (i', i'');
- ω_i: ângulo formado pelos dois planos definidos pelos tripletos (i', i'', i''') e (i'', i', i).

Os cossenos dos ângulos θ_i e ω_i podem ser computados através das posições dos vértices i', i'' e i''' e das distâncias disponíveis $d_{i'i}$, $d_{i''i}$ e $d_{i'''i}$ [10]. Assim, teremos

$$cos\,(\omega_i) = \frac{cos\,(\theta_{i''',i'',i}) - cos\,(\theta_{i',i'',i}) \cdot cos\,(\theta_{i',i'',i'''})}{sin\,(\theta_{i',i'',i}) \cdot sin\,(\theta_{i',i'',i'''})} \tag{5.3}$$

onde consideraremos os valores positivos dos "senos", e

$$cos\,(\theta_i) = cos\,(\theta_{i'',i',i}) = \frac{d^2_{i'i''} + d^2_{i'i} + d^2_{i''i}}{2 d_{i'i''} d_{i'i}} \tag{5.4}$$

Neste momento, uma vez mais, vale relembrarmos que se as 3 distâncias de referência forem todas exatas teremos a interseção de 3 esferas o que produzirá 2 pontos, com a probabilidade 1. Os 2 pontos x_i^+ e x_i^- correspondem aos valores opostos, ω_i^+ e ω_i^-, para o ângulo ω_i. Quando uma das três distâncias representa um intervalo (ver definição 5.1), a terceira esfera torna-se uma casca esférica, a interseção produz duas curvas (fig. 5.4). Estas duas curvas correspondem aos dois intervalos $[\underline{\omega}_i^+, \overline{\omega}_i^+]$ e $[\underline{\omega}_i^-, \overline{\omega}_i^-]$, para o ângulo ω_i. Com o objetivo de discretizar estes intervalos, um certo número de pontos, D, deverão ser selecionados a partir destas duas curvas.

5.5 Paralelizando o *iDMDGP* por *Dataflow*

Como metodologia para encontrar as soluções (as posições atômicas) da equação 5.1, empregaremos o algoritmo *iBP*, o qual, explorará o espaço de busca (topologia em árvore), de maneira recursiva, verificando a cada novo ramo adicionado, se as soluções (posições atômicas) encontradas satisfazem as condições impostas pelos dados experimentais, *pruning phase*. A cada nível que exploramos testamos várias posições, o *branching*. Ao se descobrir posições inviáveis reduz-se drasticamente o espaço de busca, pois assim que descobrimos uma posição não conforme deixamos de explorar aquele ramo (o ramo da árvore de busca que possui o nó em questão como raiz). No pior cenário, sem a poda, a árvore de busca crescerá exponencialmente.

A estratégia que empregaremos para percorrermos a árvore de busca de maneira eficiente será o paralelismo por *Dataflow*, de tal maneira que a árvore será particionada a partir de um determinado nível. Cada subárvore, ramo, tornar-se-á assim uma partição e será submetida a um *worker* (*core*), o qual encontrará as soluções associadas, fig. 5.5.

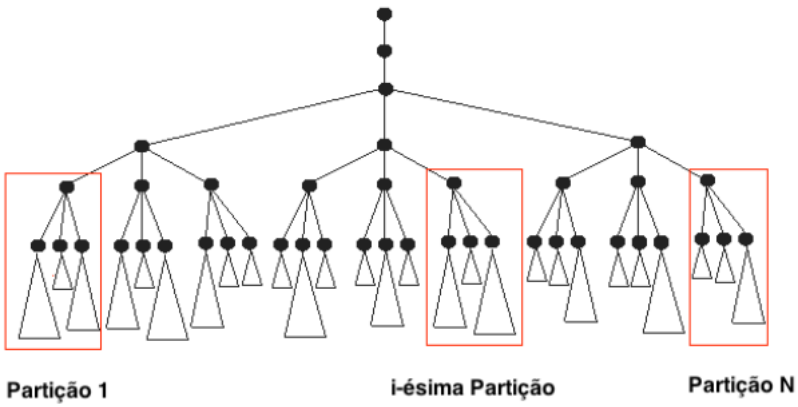


Figura 5.5: Particionamento da árvore de busca do *iDMDGP*.

Uma vez que se tem as soluções encontradas por todos os *workers* seleciona-se a solução que possua o menor valor para o *LDE, Largest Distance Error*, uma medida

da qualidade da solução.

Como apresentado no capítulo 3, empregamos neste trabalho a biblioteca *Dataflow Sucuri* como base para implementarmos o paralelismo no *iDMDGP*.

A fig. 5.6 apresenta o tratamento que desenvolvemos para a paralelização do *iDMDGP* valendo-se da biblioteca *Sucuri*, em específico, o grafo empregado (o código fonte do programa encontra-se na seção A.2).

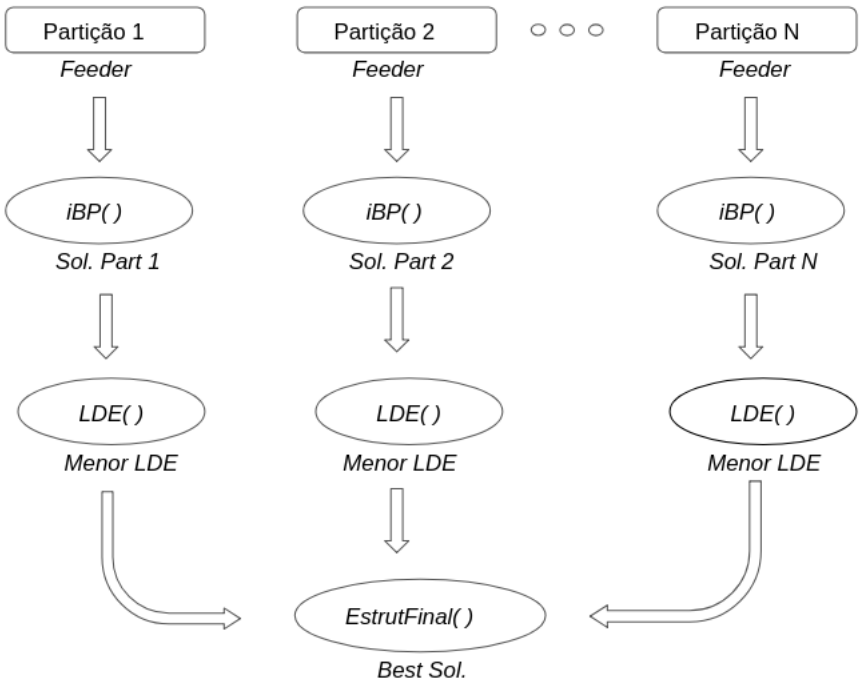


Figura 5.6: Particionamento da árvore de busca do *iDMDGP*.

Inicialmente exploraremos a árvore de busca, empregando o algoritmo *Branch & Prune*, descrito em LAVOR *et al.* [6], até um determinado nível, a partir deste nível particionaremos a árvore de busca em função do número de soluções encontradas até o presente nível. As informações referentes a cada partição serão atribuidas aos nós *Feeders* (nós especializados) os quais as repassarão aos nós que executam a rotina *iBP*, obtendo-se assim as soluções associadas a cada uma das partições. Estas soluções serão direcionadas aos nós que selecionarão as soluções com o menor *LDE* e as repassarão a um único nó, o qual será capaz de selecionar globalmente a solução com o menor *LDE*, a melhor solução.

5.6 Resultados Computacionais Experimentais

5.6.1 Experimentos

Realizamos duas baterias de experimentos, em computadores *multicores*, empregando o algoritmo proposto, este paraleliza por *Dataflow* o *iDMDGP*. O programa

foi implementado na linguagem de programação *Python 2.7*.

A qualidade de cada solução encontrada foi determinada através da medida, adaptada ao *iDMDGP*, *Largest Distance Error (LDE)* [10]:

$$LDE(X) = \frac{1}{|E|} \sum_{(u,v)\in E} \left[\frac{max(\underline{d}(u,v) - |x_u - x_v|, 0)}{\underline{d}(u,v)} + \frac{max(|x_u - x_v| - \overline{d}(u,v), 0)}{\overline{d}(u,v)} \right] \tag{5.5}$$

onde: $X = (x_1, x_2, ..., x_n)$, E corresponde ao conjunto de todas as distâncias conhecidas, $|\cdot|$ a cardinalidade e, por último, $\underline{d}(u,v)$ e $\overline{d}(u,v)$ designam, respectivamente, os limites inferior e superior para a distância $d(u,v)$.

Nos experimentos foram empregadas instâncias *iDMDGP* (seção 5.6.2), artificialmente geradas, as chamadas *Lavor Instances* [51], com $n = 200, 300$ e 500 vértices, para a primeira bateria de experimentos, e $n = 300, 500$ e 1000 vértices para a segunda.

Os experimentos foram realizados 10 vezes, para cada instância (exceto as instâncias contendo 1000 átomos) e conjunto de *cores* empregados, com o intuito de obter um comportamento médio. Foram realizadas duas séries de experimentos (seção 5.6.3).

5.6.2 Geração das Instâncias *iDMDGPs*

Na criação das instâncias para o iDMDGP nos valemos de um procedimento semelhante ao empregado para o caso de estruturas DMDGP (seção 4.4.2), em verdade se baseia nele. As principais diferenças entre estes foram:

- Os valores dos ângulos de ligação (dos átomos da estrutura molecular artificialmente gerada) empregados no novo procedimento foram fixados em $\theta_i = 114,59°$.

- A geração de cada ângulo de torção (da estrutura molecular iDMDGP) foi realizada através da seleção aleatória de um valor ω_i, tal que

$$\omega_i \in [0, 2\pi]. \tag{5.6}$$

- Após o procedimento de poda, descrito na seção 4.4.2, algumas *distâncias de referência* ($d_{i''',i}$), aleatoriamente selecionadas, foram representadas sob a forma de um intervalo.

Assim, ao final deste novo procedimento obtemos um conjunto de distâncias, que serve como *input* ao nosso programa que resolve o iDMDGP. Cada instância iDMDGP empregada foi intitulada pela palavra *ilavor* seguida pela quantidade de átomos. Exemplo: uma instância iDMDGP com 15 átomos será denominada *ilavor15*.

5.6.3 Resultados

Série 1:

A primeira série de experimentos foi realizada em um computador *multicore*, processador *Intel(R) Core(TM)* i9-7900X, velocidade de 3.30 GHz, 10 *cores*. Os resultados são apresentados na Tabela 5.1.

Instância	Qtd. Átomos	Partições	T. Serial[s]	T. Paralelo[s]	*Speedup M.*
ilavor200	200	24	76.378	11.350	6.74
ilavor300	300	62	268.488	38.750	6.93
ilavor500a	500	26	111.420	17.787	6.26
ilavor500b	500	122	2879.756	549.066	5.24

Tabela 5.1: Dados referentes a primeira série de experimentos com instâncias *iDMDGP*.

Os experimentos foram realizados com instâncias de 200, 300 e 500 átomos. As instâncias de 500 átomos, no caso duas, são diferentes entre si. Empregamos um fator de *branching* igual a 5 e particionamos as árvores de busca a partir do sexto nível. Na tabela 1 temos os tempos de execução encontrados nos experimentos, a coluna *T. Serial[s]* corresponde ao tempo gasto por apenas 1 *core*, enquanto a coluna *T. Paralelo[s]* ao tempo consumido empregando os 10 *cores*. A coluna *Partições* corresponde ao número de particionamentos da árvore de busca gerada para cada instância e a coluna *Speedup M.* contém os *Speedups Máximo* alcançados pelo nosso programa nos experimentos.

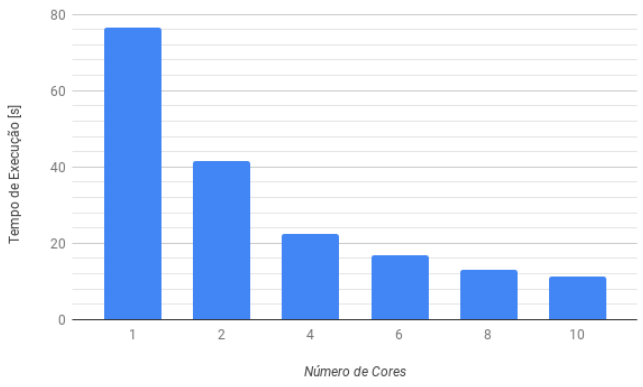


Figura 5.7: Tempo de Execução[s] *versus* Número de *Cores*, instância *ilavor200*.

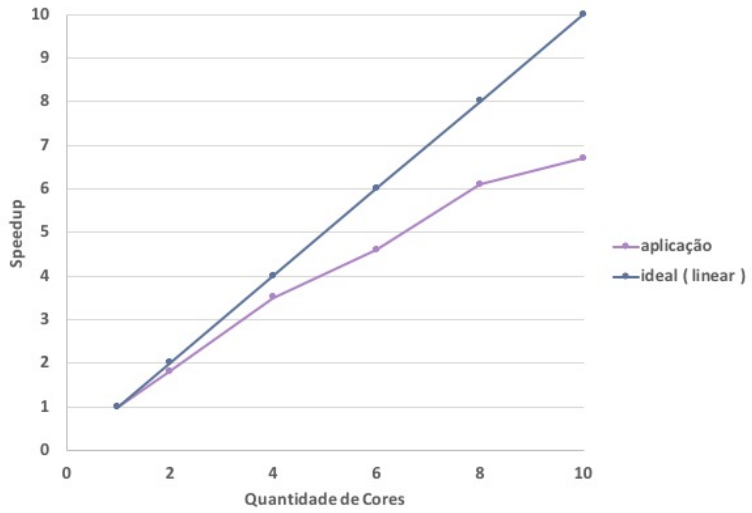


Figura 5.8: *Speedup versus* Quantidade de *cores*, instância *ilavor200*.

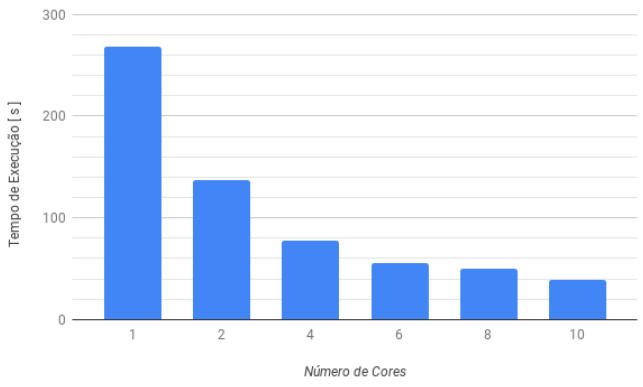


Figura 5.9: Tempo de Execução[s] *versus* Número de *cores*, instância *ilavor300*.

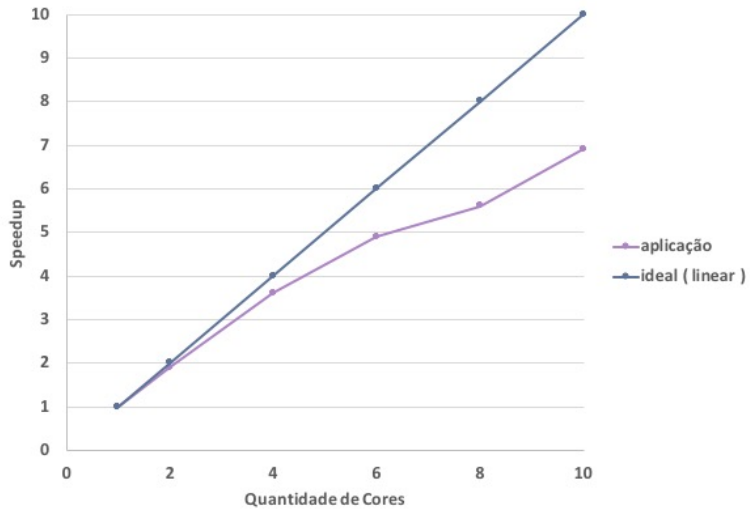


Figura 5.10: *Speedup versus* Quantidade de *cores*, instância *ilavor300*.

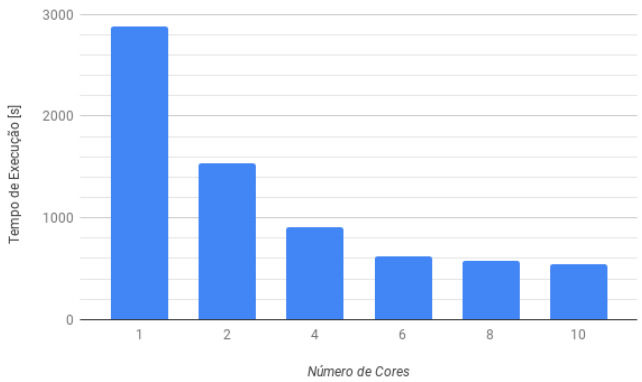


Figura 5.11: Tempo de Execução[s] *versus* Quantidade de *cores*, instância *ilavor500b*.

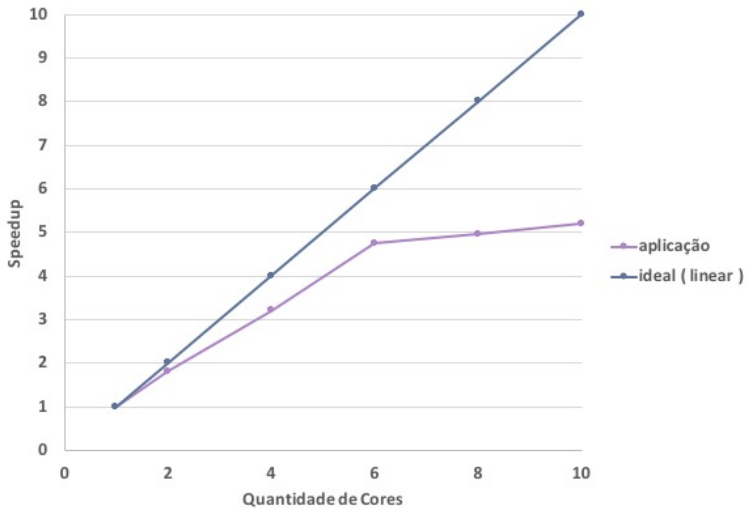


Figura 5.12: *Speedup versus* Quantidade de *cores*, instância *ilavor500b*.

Nas figuras 5.7, 5.9 e 5.11 apresentamos os (gráficos) tempos de execução em função do número de *cores (workers)* para as instâncias de 200, 300 e 500 átomos, enquanto que nas figs. 5.8, 5.10 e 5.12 temos os gráficos *speedup* x *cores*, destas.

Série 2:

A segunda série de experimentos foi realizada em um computador *multicore*, Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz, 16 *cores*. Os resultados são apresentados na Tabela 5.2.

Os experimentos foram realizados com instâncias de 300, 500 e 1000 átomos. Empregamos um fator de *branching* igual a 5 e particionamos as árvores de busca a partir do vigésimo nível. Assim, da mesma maneira que na primeira série de experimentos do *iDMDGP*, na tabela 5.2 apresentamos os tempos de execução encontrados nos experimentos, o *Speedup Máximo* e a quantidade de partições de cada instância.

Instância	Qtd. Átomos	Partições	T. Serial[s]	T. Paralelo[s]	*Speedup M.*
ilavor300b	300	24	671,9181	65,8838	10,20
ilavor500c	500	130	4228,3606	347,0159	12,19
ilavor1000a	1000	382	–	494770,5893	–
ilavor1000b	1000	28	–	–	–

Tabela 5.2: Dados referentes a segunda série de experimentos com instâncias *iDMDGP*.

O processamento da instância *ilavor1000b* foi interrompido após 34 horas de processamento, pois o consumo da memória atingiu valores proibitivos para a máquina, no caso 44 GB, enquanto que o limite da máquina é de 47 GB. Assim, não conseguimos "resolvê-la". Nas outras instâncias a memória consumida, ao longo de todo o processo, não passou de 4GB, no máximo.

A instância *ilavor1000a* obteve um valor alto para o tempo de execução ($\sim$ 6 dias), com *16 cores*, de tal maneira que optamos por não realizar outros testes, com ela, com o objetivo de computar o *Speedup*.

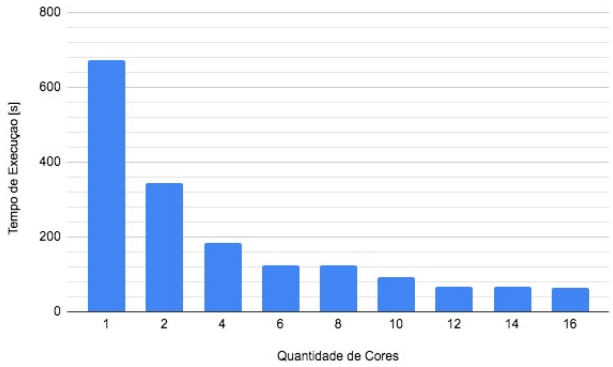


Figura 5.13: Tempo de Execução *versus* Número de *Cores*, instância *ilavor300b*.

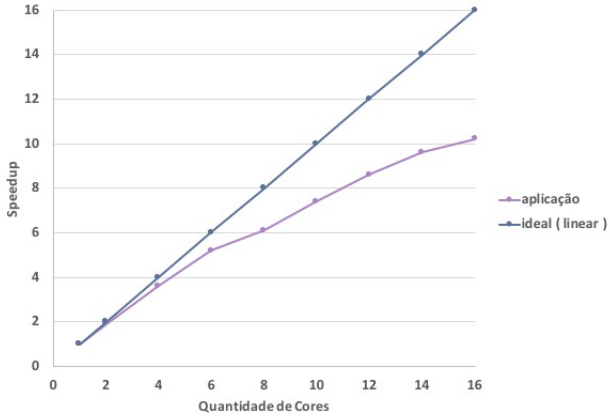


Figura 5.14: *Speedup versus* Quantidade de *cores*, instância *ilavor300b*.

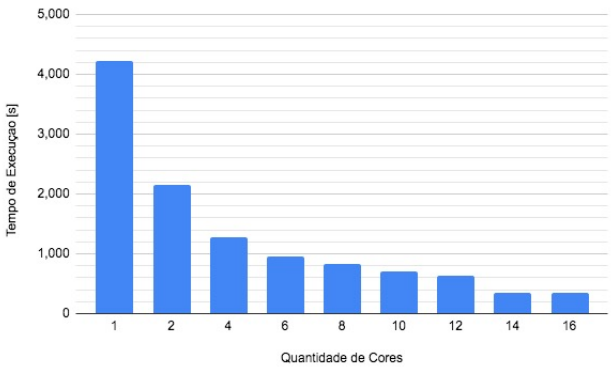


Figura 5.15: Tempo de Execução *versus* Número de *Cores*, instância *ilavor500c*.

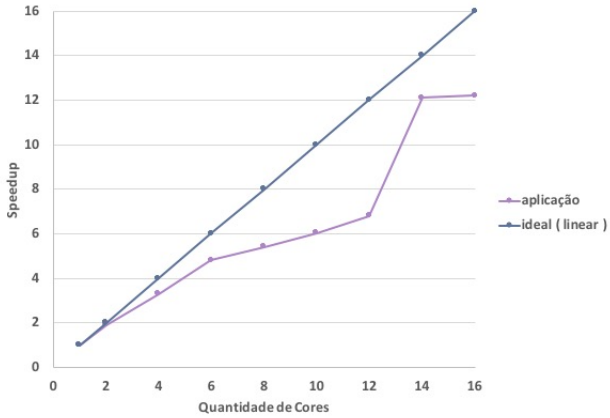


Figura 5.16: *Speedup versus* Quantidade de *cores*, instância *ilavor500c*.

Nas figuras 5.13 e 5.15 apresentamos os (gráficos) tempos de execução em função do número de *cores (workers)* para as instâncias de 300 e 500 átomos, enquanto que nas figs. 5.14 e 5.16 apresentamos os gráficos *speedup* x *cores*, destas.

5.6.4 Análise

Os resultados apresentados nas tabelas 5.1 e 5.2 mostram que a abordagem envolvendo o paralelismo apresentou uma redução, no tempo de execução, considerável quando comparado à versão serial. Dos gráficos, figuras 5.7 a 5.12, notamos o mesmo comportamento, notadamente para a instância *ilavor500c*. Porém, apesar da redução no tempo de execução não obtivemos bons resultados quanto aos valores de *speedup* para alguns casos.

Um dos fatores que consideramos como razão para não termos encontrado melhores valores para os *speedups* (o maior valor encontrado foi de 12.19, para a instância *ilavor500c*, com o emprego de 16 *cores*) consiste no fato de que as partições, na sua grande maioria, não são bem balanceadas. Por exemplo:

- a instância *ilavor500b* (da primeira série de experimentos com o *iDMDGP*),

com 122 partições investigadas, verificamos que dentre estas a grande maioria foi "resolvida", podada, em menos de um milésimo de segundo de processamento. Ao final do processo de exploração da árvore de busca, associada à esta instância, observamos que apenas 6 partições (em cada partição foram encontradas 1514 soluções, total de 9084 soluções) dominaram a maior parte do processamento, as 6 consumiram aproximadamente 338 segundos (mais da metade do tempo total de processamento), enquanto isto 4 *cores* permaneceram inativos.

- o mesmo comportamento foi observado na segunda série de experimentos, para as instâncias que não "escalaram bem ", notadamente a instância *ilavor1000a*, a maioria das partições associadas à árvore de busca desta instância, inicialmente 382, foram podadas em milésimos de segundos (350 partições), 24 partições consumiram um tempo de aproximadamente 200 segundos até serem podadas, 6 tiveram um tempo de execução igual a 256000 segundos ($\sim$ 71 horas) até a poda e 2, ao final, consumiram o tempo de aproximadamente 495000 segundos ($\sim$138 horas). Foram encontradas 8 soluções no total.

Como podemos observar, o problema em alguns casos não gera uma boa distribuição de carga. É necessário uma abordagem que envolva o escalonamento dinâmico de tarefas, escolha mais adequada em casos desta natureza (desbalanceamento de carga). Nesta pesquisa empregamos o escalonamento estático.

É interessante comentarmos o experimento referente à instância *ilavor1000b*, um exemplo do "pior caso"que pode ocorrer. Ela não nos fornece informações suficientes (do seu conjunto de distâncias) para que possamos realizar podas eficientes, de tal maneira que a quantidade de "ramos"da árvore de busca a serem explorados cresça exponencialmente.

Por último, assim como no caso do DMDGP, vale ressaltar que quando as partições são bem balanceadas a abordagem empregada obtém bons resultados, vejamos os *speedups* obtidos no tratamento das instâncias *ilavor300b* e *ilavor500c*, respectivamente, 10.2 e 12.2, este último próximo do caso linear.

Capítulo 6

Conclusão

6.1 Discussão

Neste trabalho, em primeiro lugar, apresentamos uma abordagem por *Dataflow* para paralelizar o DMDGP. Realizamos uma série de experimentos computacionais com instâncias, que simulam dados provenientes de experimentos de RMN, geradas artificialmente. Os experimentos apesar de mostrarem a efetividade do emprego da biblioteca *Dataflow Sucuri* no tratamento do DMDGP, indicam que o código não escala bem em algumas situações. Isto deve-se, em grande parte, a maneira particular proposta para se particionar as instâncias DMDGP, os vértices de simetria. O emprego das propriedades dos vértices de simetria foi motivado pelos interessantes resultados apresentados por FIDALGO [4], os quais apontavam na direção de que se molécula fosse particionada segundo os seus vértices de simetria as soluções encontradas, via união das soluções parciais, não necessitariam ser testadas. Isto, inicialmente, sugeria um possível ganho computacional. Porém, como podemos observar pelos experimentos, a ocorrência de partições desbalanceadas é frequente e acaba por comprometer o desempenho computacional.

Vale enfatizar que em nossa abordagem optamos por encontrar todas as soluções possíveis e não apenas algumas. O que em alguns casos acaba por ser uma tarefa árdua: no DMDGP não é raro encontrar uma instância com 2^{20} soluções, o que implicará no gerenciamento da alocação de uma quantidade enorme de processos.

Uma estratégia que contornaria a maioria dos problemas encontrados em nossa abordagem para o DMDGP seria a de empregarmos o procedimento que aplicamos no *iDMDGP*, ao invés de particionarmos a molécula dividiríamos o espaço de busca. Assim, não teriamos o inconveniente de partições absurdamente desbalanceadas e não precisaríamos nos preocupar com o emprego de matrizes de rotações para realizar as uniões das soluções parciais. Claro, valendo-se do escalonamento dinâmico.

Posteriormente, abordamos o *iDMDGP*, e assim como no caso do DMDGP em

algumas situações o código não escala bem. Uma vez mais, isto ocorre devido as partições serem altamente desbalanceadas. Assim, o tratamento indicado, em ambos os casos, envolve o emprego do escalonamento dinâmico.

O emprego da abordagem por *Dataflow* e por sua vez o uso da biblioteca *Sucuri*, escrita em *Python*, facilitou muito o projeto e a escrita dos programas. No entanto, faltou-nos otimizar o código: descobrir os gargalos de consumo computacional e reescrevê-los na linguagem C, isto potencialmente reduziria o tempo de execução em até três ordens de grandeza, na maioria dos casos é necessário que se reescreva apenas 20% do código (regra de Pareto).

Os bons resultados encontrados, para as instâncias DMDGP e *iDMDGP*, ocorreram quando as partições possuiam boa distribuição de carga, isto mostra que para estes casos as abordagens empregadas são boas.

6.1.1 Trabalhos Futuros

Com o intuito de dar prosseguimento à pesquisa, elencamos algumas frentes de trabalho:

- Quanto ao DMDGP, pretendemos aplicar a mesma abordagem aplicada ao *iDMDGP*, particionaremos a árvore de busca, ao invés da molécula, sem a necessidade da fase de uniões das soluções parciais.

- Com relação ao *iDMDGP*, pretendemos atacá-lo com uma abordagem que empregue o escalonamento dinâmico.

- Desejamos, também, investigar (*iDMDGP*) o pariticionamento a partir de vários níveis (valores diferentes de 6 e 20), níveis mais profundos e o emprego do fator de *branching* com valores maiores do que 5. O *branching* possui uma importância capital no tempo total de processamento pois ele determina a quantidade de posições atômicas a serem exploradas, por exemplo: um fator igual a 5, em uma instância de 500 átomos, implicará numa quantidade de aproximadamente 5^{497} posições atômicas a serem investigadas. Um número maior do que a quantidade de átomos do universo ($\sim 10^{87}$).

Referências Bibliográficas

[1] LIBERTI, L., LAVOR, C., MACULAN, N., et al. "Euclidean distance geometry and applications", *SIAM Review*, v. 56, n. 1, pp. 3–69, 2014.

[2] LAVOR, C., LIBERTI, L. "Um convite à Geometria de Distâncias", *CNMAC - SBMAC*, 2014.

[3] GRAMACHO, W. *Algoritmos sequenciais e paralelos para problemas de geometria molecular*. Tese de D. Sc., COPPE/UFRJ, Rio de Janeiro, Brasil, 2013.

[4] FIDALGO, F. D. C. *Dividindo e Conquistando com Simetrias em Geometria de Distâncias*. Tese de D.Sc., IMECC/UNICAMP, Campinas, SP, Brasil, 2015.

[5] GOLDSTEIN, B. F. "Construção de núcleos paralelos de álgebra linear computacional com execução guiada por fluxo de dados", *Dissertação de Mestrado, UFRJ*, 2015.

[6] ALVES, T. A. O., GOLDSTEIN, B. F., FRANÇA, F. M. G. "A minimalistic dataflow programming library for python", *Computer Architecture and High Performance Computing Workshop(SBAC-PADW), International Symposium.*, 2014.

[7] FIDALGO, F., RODRIGUEZ, J. "Exploiting symmetries in . a divide-and-conquer approach for solving DMDGP". In: *Proceedings of the Many Faces of Distances Workshop*, pp. 1–3, Campinas, 2014.

[8] LAVOR, C., MACULAN, N., SOUZA, M., et al. "Álgebra e Geometria no Cálculo de Estrutura Molecular", *31º Colóquio Brasileiro de Matemática - IMPA*, 2017.

[9] LAVOR, C., LIBERTI, L., MUCHERINO, A. "The interval Branch-and-Prune algortihm for the discretizable molecular distance geometry problem with inexact distances", *Journal of Global Optimization*, pp. 1–17, 2011.

[10] GONÇALVES, D. S., M. A. "Discretization orders and efficient computation of cartesian coordinates for distance geometry", *Optimization Letters*, v. 8, n. 7, pp. 2111–2125, 2014.

[11] GARRET, R. H., GRISHAM, C. M. *Principles of Biochemistry with a human focus.* 4 ed. Boston, USA, Cengage Learning, 2010.

[12] LIBERTI, L., L. C. M. A., MACULAN, N. "The Discretizable Molecular Distance Geometry Problem", *Computational Optimization and Applications*, v. 52, pp. 115–146, 2012.

[13] MUCHERINO, A., LAVOR, C., LIBERTI, L. "The discretizable distance geometry problem", *Optimiztion Letters*, v. jun., pp. 1–16, 2011. ISSN: 1862-4472.

[14] DENNIS, J. B., FOSSEN, J. B. "Introduction to Data Flow Schemas", *Computation Structures Group Memo, MIT*, 1973.

[15] MARZULO, L. A. J. "Explorando linhas de execução paralelas com programação orientada por fluxo de dados", *Tese (Doutorado em Eng. de Sistemas e Computação)- Programa de Engenharia de Sistemas e Computação, COPPE - Universidade Federal do Rio de Janeiro*, 2011.

[16] MARZULO, L. A. J., ALVES, T. A. O., GOLDSTEIN, B. F., et al. "Exploiting Parallelism in Linear Algebra Kernels through Dataflow Execution", *International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, 2015.

[17] TROSSET, M. W. "Applications of Multidimensional Scaling to Molecular Conformation", *Computing Science and Statistics*, v. 29, n. 1, pp. 148–152, 1998.

[18] HENDRICKSON, B. "The molecule problem: Exploiting structure in global optimization", *SIAM Journal on Optimization*, v. 5, n. 4, pp. 835–857, 1995.

[19] HENDRICKSON, B. "Conditions for Unique Graph Realizations", *SIAM Journal on Optimization*, v. 21, n. 1, pp. 65–84, 1992.

[20] CRIPPEN, G., HAVEL, T. "Distance Geometry and Molecular Conformation", *New York, Research Studies Press Ltd.*, 1988.

[21] MORÉ, J. J., WU, Z. "Distance geometry optimization for protein structures", *Journal of Global Optimization*, v. 15, n. 3, pp. 219–234, 1999.

[22] MORÉ, J. J., WU, Z. "Global Continuation for Distance Geometry Problems", *SIAM Journal on Computing*, v. 7, n. 3, pp. 814–836, 1997.

[23] MORÉ, J. J., WU, Z. "Smoothing techniques for macromolecular global optimization", *Nonlinear Optimization and Applications*, pp. 297–312, 1996.

[24] ZOU, Z., BIRD, R. H., SCHNABEL, R. B. "A stochastic/pertubation global optimization algorithm for distance geometry problems", *Journal of Global Optimization*, v. 11, n. 1, pp. 91–105, 1997.

[25] LAVOR, C., LIBERTI, L., MACULAN, N. "Global Optimization: Scientific and Engineering Case Studies", *Computational Experience With The Molecular Distance Geometry Problem*, New York, Springer, 2006.

[26] WU, D., WU, Z. "An updated geometric build-up algorithm for solving the molecular distance geometry problems with sparse distance data", *Journal of Global Optimization*, v. 37, n. 4, pp. 661–673, 2007.

[27] CARVALHO, R., LAVOR, C., PROTTI, F. "Extending the geometric build-up algorithm for the molecular distance geometry problem", *Inf. Process. Lett.*, v. 108, n. 4, pp. 234–237, 2008.

[28] HAVEL, T. F. "An Evaluation of Computational Strategies for Use in the Determination of Protein Structure from Distance Constraints obtained by Nuclear Magnetic Resonance", *Progress in Biphysics and MOlecular Biology, Pergamon Press*, pp. 43–78, , Oxford, England, 1991.

[29] LAVOR, C., LIBERTI, L., MACULAN, N. "An overview of distinct approaches for the molecular distance geometry problem", *Encyclopedia of Optimization*, Berlin, 2nd Edition, 2008.

[30] LAVOR, C., LIBERTI, L., MACULAN, N., et al. "Recent advances on the discretizable molecular distance geometry problem", *European Journal of Operational Research*, 2011.

[31] MACULAN, N., MUCHERINO, A., LAVOR, C., et al. *Distance Geometry. Theory, Methods, and Applications.* Springer New York Heidelberg Dordrecht London, 2013. ISBN: 978-1-4614-5127-3. doi: 10.1007/978-1-4614-5128-0.

[32] GONÇALVES, D. S., MUCHERINO, A., LAVOR, C., et al. "Recent advances on the interval distance geometry", *J. Glob. Optim.*, v. 1, n. 69, pp. 525–545, 2017.

[33] GONÇALVES, D., MUCHERINO, A., LAVOR, C. "An adaptive branching scheme for the Branch & Prune algorithm applied to Distance Geometry", *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, v. 2, pp. 457–463, 2014.

[34] MUCHERINO, A., LAVOR, C., LIBERTI, L., et al. "A parallel version of the Branch-and-Prune algorithm for the molecular distance geometry problem", *IEEE Conference Proceedings, ACS/IEEE International Conference on Computer Systems and Applications (AICCSA10), Hammamet, Tunisia*, pp. 1–6, 2010.

[35] VILELA, S. P., MARZULO, L. A. J., LAVOR, C., et al. "Explorando Paralelismo Dataflow em Geometria de Distâncias Moleculares", *IV Escola Regional de Alto Desempenho do Rio de Janeiro*, v. 1, 2018.

[36] VILELA, S. P., MARZULO, L. A. J., FRANÇA, F. M. G. "Explorando Paralelismo Dataflow em Geometria de Distâncias Moleculares Intervalares", *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*, v. 1, 2019.

[37] DONG, Q., WU, Z. "A linear-time algorithm for solving the molecular distance geometry problem with exact inter-atomic distances", *Journal of Global Optimization*, v. 22, pp. 365–375, 2002.

[38] SAXE, J. B. "Embeddability of weighted graphs in k-space is strongly NP-hard", *Proc. 17th Allerton Conference in Communications, Control, and Computing, Monticello, IL*, v. 22, pp. 480–489, 1979.

[39] CREIGHTON, T. "Proteins: Structures and Molecular Properties", *W. H. Freeman and Company, New York*, 1993.

[40] SCHLICK, T. "Molecular Modeling and Simulation: An Interdisciplinary Guide", *Secaucaus, NJ, USA, Springer-Verlag New York, Inc. ISBN: 038795404X*, 2002.

[41] PHILLIPS, A., R. J. W. V. "Molecular structure determination by convex global underestimation of local energy minima", ., 1996.

[42] LIBERTI, L., LAVOR, C., MUCHERINO, A., et al. "Molecular distance geometry methods: from continuous to discrete", *International Transactions in Operational Research*, v. 18, pp. 33–51, 2010.

[43] LAVOR, C., LIBERTI, L., MACULAN, N., et al. "The discretizable molecular distance geometry problem", *Computational Optimization and Applications 52*, pp. 115–146, 2012.

[44] DENNING, P. J., LEWIS, T. G. "Exponential Laws of Computing Growth", *Communications of the ACM*, v. 60, n. 1, pp. 54–65, 2017.

[45] PARKHURST, J., DARRINGER, J., GRUNDMANN, B. "From single core to multi-core: preparing for a new exponential", *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pp. 67–72, 2006.

[46] ASANOVIC, K. E. A. "The Landscape of Parallel Computing Research:A View from Berkeley", *Technical Report UCB/EECS*, v. 183, 2006.

[47] SUTTER, H. "The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software", *Dr. Dobb's*, v. 30, n. 3, 2005.

[48] LIBERTI, L., MASSON, B., LEE, J., et al. "On the number of realizations of certain Hennenberg graphs arising in protein conformation", *Discrete Applied Mathematics*, 2014.

[49] MUCHERINO, A., LAVOR, C., LIBERTI., L. "Exploiting symmetry properties of the Discretizable Molecular Distance Geometry Problem", *Journal of Bioinformatics and Computational Biology 10*, pp. 1–6, 2012.

[50] NUCCI, P., NOGUEIRA, L., LAVOR, C. "Solving the Discretizable Molecular Distance Geometry Problem by multiple realization trees", *Distance Geometry: Theory, Methods and Applications.*, pp. 161–176, 2013.

[51] LAVOR, C. "On generating instances for the molecular distance geometry problem", *Ed. por L. Liberti e N. Maculan*, v. 84, pp. 405–414, 2006.

[52] MUCHERINO, A., LIBERTI, L., LAVOR, C. "MD-Jeep: an implementation of a Branch-and-Prune Algorithm for Distance Geometry Problems", *Lecture Notes in Computer Science*, v. 6327, pp. 186–197, 2010.

[53] PHILLIPS, A. T., ROSEN, J. B., WALKE, V. H. "Molecular structure determination by convex underestimation of local encergy minima", *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, v. 23, pp. 181–198, 1996.

Apêndice A

Códigos Fonte

A.1 Programa Sucuri *Dataflow*-DMDGP

Código fonte de programa escrito em *Python* empregando-se a biblioteca Sucuri *Dataflow* para paralelizar o DMDGP (seção 4.3), programa principal:

```
#Programa Sucuri DMDGP

import numpy as np
from treelib import Node, Tree
import time
import sys, os
from funcoesAuxiliares import *

sys.path.insert(0, "/home/san/Programas/Sucuri-master")
from pyDF import *

sys.setrecursionlimit(1000000)

class SourceIn(Node): #source class
    def __init__(self):
        self.inport = [[]]
        self.dsts = []
        self.tagcounter = 0
        self.affinity = [0]

    def run(self, args, workerid, operq):
        for e in args[0].val:
            tag = self.tagcounter
            opers = self.create_oper(TaggedValue(e,tag), workerid,
                operq)
            for oper in opers:
                oper.request_task = False
```

```
            self.sendops(opers, operq)
            self.tagcounter += 1
        opers = [Oper(workerid, None, None, None)] #sinalize eof
        self.sendops(opers, operq)

    def f(self, line):
        #default source operation
        return line

def main():

    # 1-) abrindo o arquivo com as informacoes a respeito das distancias

    nmr_file=sys.argv[1]
    nmr_worker=int(sys.argv[2])
    arquivoDistancias = open(nmr_file, 'r')

    # armazenar as distancias entre os atomos em uma matrix: Dist
    # coluna1= atomoI, coluna2=atomoJ, coluna3= distancia entre atomoI e atomoJ, coluna4=tipo atomI, coluna5=tipo atomJ

    lista_de_linhas = arquivoDistancias.readlines()
    numeroLinhas = (len(lista_de_linhas))
    arquivoDistancias.close()
    arquivoDistancias = open(nmr_file, 'r')
    count = 0
    Dist = [[0 for m in range(5)] for i in range(numeroLinhas)]  # cria ListaBidimensional: numeroLinhas X 5

    for i in range(0, numeroLinhas):
        linha = arquivoDistancias.readline()
        valores = linha.split()
        Dist[i][0] = valores[0]
        Dist[i][1] = valores[1]
        Dist[i][2] = valores[2]
        Dist[i][3] = valores[4]
        Dist[i][4] = valores[5]
        count = count + 1
    numeroAtomos = Dist[numeroLinhas - 1][1]

    # criaremos o conjunto com as arestas de poda (matriz):
    cJpoda = matrizArestasDePoda(Dist, numeroLinhas)

    conjuntoDEPODAS = np.array(cJpoda)
    numeroAtomos = int(numeroAtomos)
    conjuntoSimetriaC = conjSimetrias(Dist, numeroAtomos, numeroLinhas)
```

```
    Dist = np.array(Dist)
    numeroAtomos = int (Dist[numeroLinhas - 1][1])

    #vamos descobrir os limites atomicos do particionamento do backbone
        de atomos

    matParticao = SymSplit(Dist,numeroAtomos,numeroLinhas)
    print '4-) Particao:\n', matParticao
    matParticao = np.array(matParticao)
    dim_matParticao = matParticao.shape

    # inicio do paralelismo

    nworkers = nmr_worker
    ntasks = dim_matParticao[0]
    print '\nntasks:', ntasks

    #particionando fora do laco dos Feeders

    DistPart =[]
    for i in xrange(ntasks):
        Distp = distPartition(Dist, matParticao[i][0], matParticao[i
            ][1], numeroLinhas)
        DistPart.append(Distp)

    # A estrutura particionada foi armazenda em DistPart
    #print "Dimensoes de DistPart:", len(DistPart)

    graph  = DFGraph()
    sched = Scheduler(graph, nworkers, False)
    EstrutFinal = Node(eachSolution, ntasks)
    graph.add(EstrutFinal)

    for i in xrange(ntasks):
        print 'i:', i
        idPlusPart=[] #lista que contem o id (primeiro campo) e a
            particao correspondente (segundo campo)
        idPlusPart.append(i)
        DisI = np.array(DistPart[i])
        idPlusPart.append(DisI)
        Estrut = Feeder(idPlusPart)
        graph.add(Estrut)
        BP_partial = Node(bpPartialTime, 1)
        graph.add(BP_partial)
        Estrut.add_edge(BP_partial, 0)
        BP_partial.add_edge(EstrutFinal, i)
```

```
115     Caminhos = Node(retornaCadaCaminho,1)
        graph.add(Caminhos)
117     EstrutFinal.add_edge(Caminhos,0)
        reader = SourceIn()
119     graph.add(reader)
        Caminhos.add_edge(reader,0)
121     MergeNoh = FilterTagged(mergeLista,1)
        graph.add(MergeNoh)
123     reader.add_edge(MergeNoh,0)

125     # salvando toda as solucoes em uma lista
        lista_sol=[]
127     tmpNode = Serializer(dummy,1)
        graph.add(tmpNode)
129     MergeNoh.add_edge(tmpNode,0)
        sched.start()
131 main()
```

Listing A.1: Programa Sucuri-DMDGP (programa principal), escrito em Python

A.2 Programa Sucuri *Dataflow*-iDMDGP

Código fonte do programa escrito em *Python* empregando-se a biblioteca Sucuri *Dataflow* para paralelizar o iDMDGP (seção 5.5), programa principal:

```
2 #Programa que implementa a versao paralela, via Sucuri, do IBP

4 from numpy import *
  from funcoesAuxiliaresII import *
6 import sys
  sys.path.insert(0, "/home/san/Programas/Sucuri-master/")
8 from pyDF import *
  sys.setrecursionlimit(1000000)
10
  def main():
12     # 1-) abrindo o arquivo com as informacoes a respeito das
           distancias

14     arquivoDistancias = open(sys.argv[3], 'r')

16     # armazenar as distancias entre os atomos em uma matrix: Dist
       # coluna1= atomo1, coluna2=atomo2, coluna3= limite inferior do
           intervalo de distancias,
18     # coluna4= limite superior do intervalo de distancias,
```

```
    # coluna5=tipo atom1, coluna6=tipo atom2

    lista_de_linhas = arquivoDistancias.readlines()
    numeroLinhas = (len(lista_de_linhas))
    arquivoDistancias.close()
    arquivoDistancias = open(sys.argv[3],'r')
    count = 0
    Dist = [[0 for m in range(6)] for i in range(numeroLinhas)]  # cria
        ListaBidimensional: numeroLinhas X 5

    for i in range(0, numeroLinhas):
        linha = arquivoDistancias.readline()
        valores = linha.split()
        Dist[i][0] = valores[0]
        Dist[i][1] = valores[1]
        Dist[i][2] = valores[2]
        Dist[i][3] = valores[3]
        Dist[i][4] = valores[4]
        Dist[i][5] = valores[5]
        count = count + 1

    numeroAtomos = Dist[numeroLinhas - 1][1]

    #Fase 1
    #encontrar todas as solucoes dos N primeiros niveis, no caso os 5
        primeiros

    nivelCorte = int(sys.argv[1])#padrao igual a 6
    branchMeio = int(sys.argv[2])#padrao igual a 5
    arvoreMult = NohArvore(None)
    ListSol=[]
    ListSol.append(0)
    matrixDist = igeraVetTempConstIBP(numeroAtomos,Dist,numeroLinhas)
    matrixDist = np.array(matrixDist)
    SolFase1 = GeraNNiveis_IBP(1,numeroAtomos,arvoreMult,matrixDist,
        branchMeio,ListSol,nivelCorte)

    #termino da Fase 1, em SolFase1 teremos "todas as solucoes da
        primeira fase"
    #Em verdade teremos armazendado em SolFase1, os ponteiros para o
        ultimo Noh de cada solucao

    #Fase 2, os Feeders

    nworkers = int(sys.argv[4])
    ntasks = SolFase1[0]
    graph = DFGraph()
```

```
    sched = Scheduler(graph, nworkers,False)
    EstrutFinal = Node(solGalactica,ntasks)
    graph.add(EstrutFinal)

    for i in range(1,len(SolFase1)):
        megaFeeder =[]
        megaFeeder.append(nivelCorte)
        megaFeeder.append(numeroAtomos)
        megaFeeder.append(SolFase1[i])
        megaFeeder.append(matrixDist)
        megaFeeder.append(branchMeio)
        solFirstFase = Feeder(megaFeeder)
        graph.add(solFirstFase)
        IBPpartial = Node(BPintervalarParalelo,1)
        graph.add(IBPpartial)
        LDEpartial = Node(findBestSol,1)
        graph.add(LDEpartial)

        #as arestas
        solFirstFase.add_edge(IBPpartial,0)
        IBPpartial.add_edge(LDEpartial,0)
        LDEpartial.add_edge(EstrutFinal,i-1)

    sched.start()
main()
```

Listing A.2: Programa Sucuri-iDMDGP, apenas o programa principal (sem as funções auxiliares).