JÉFER BENEDETT DÖRR

ESCALONAMENTO DE TAREFAS NO FECHAMENTO DE LACUNAS EM PLATAFORMAS DE SEQUENCIAMENTO GENÉTICO DE NOVA GERAÇÃO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Luis C. E. De Bona

CURITIBA

2013

JÉFER BENEDETT DÖRR

ESCALONAMENTO DE TAREFAS NO FECHAMENTO DE LACUNAS EM PLATAFORMAS DE SEQUENCIAMENTO GENÉTICO DE NOVA GERAÇÃO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Luis C. E. De Bona

CURITIBA

2013

D716e

DÖRR. Jéfer, Benedett.

Escalonamento de tarefas no fechamento de lacunas em plataformas de sequenciamento genético de nova geração / Jéfer Benedett Dörr - Curitiba, 2013.

68f. : il. [algumas color.], tab.

Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.

Orientador: Luis C. E. De Bona.

Bibliografia: p. 42-48.

1. Bioinformática. 2. Genomas. I. Bona, Luis Carlos Erpen de. II. Universidade Federal do Paraná. III. Título.

CDD: 572.80285

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Jéfer Benedett Dorr, avaliamos o trabalho intitulado, *"Escalonamento de tarefas de fechamento de gaps em plataformas de sequenciamento genético de nova geração"*, cuja defesa foi realizada no dia 05 de setembro de 2013, às 13:30 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela **aprovação** do candidato.

Curitiba, 05 de setembro de 2013.

Prof. Dr. Luis Carlos Erpen de Bona
DINF/UFPR – Orientador

Prof. Dr. Clodis Boscarioli
UNIOESTE – Membro Externo

Prof. Dr. Marcos Alexandre Castilho
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

Dedico esta dissertação a minha esposa e filha que me acompanharam neste projeto.

Agradeço aos meus pais e meu irmão, sem os quais este Mestrado não seria realizável.

Agradeço ao meu orientador Bona por ter me selecionado e acreditado.

Aos professores, pelos conhecimentos adquiridos.

Aos colegas de curso pela convivência e trocas de experiências. Em especial ao Galante por ter auxiliado diretamente no meu trabalho, ao Wesley e ao Luiz Antônio pelo apoio no início do mestrado e aos colegas de mestrado companheiros diretos de laboratório, Rubens, Russo, Ricardo e Aílton.

"A cada dia sabemos mais e entendemos menos".

(Albert Einstein)

"Quanto mais aumenta nosso conhecimento, mais evidente fica nossa ignorância".

(John Fitzgerald Kennedy)

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

A - Adenina

ASiD - *Assembly assistant for SOLiD* - Assistente de montagem para *SOLiD*

C - Citosina

ccNUMA - *Cache coherence non-uniform memory access* - Cache coerente acesso não uniforme a memória

cDNA - *Complementary deoxyribonucleic acid* - Ácido desoxirribonucléico complementar

CPU - *Central processing unit* - Unidade central de processamento

DBG - *De bruijn graphs* - Grafo de bruijn

DNA - *Deoxyribonucleic acid* - Ácido desoxirribonucleico

EDF - *Earliest Deadline First*

E.coli - *Escherichia coli*

FCFS - *First Come First Serve* - Primeiro a chegar, primeiro a ser servido

G - Guanina

HPC - *High performance computing* - Computação de alto desempenho

HTS - *High throughput sequencing* - Sequenciamento de alta vazão

Kpb - Kilo pares de bases ou mil pares de bases

LJF - *Longest Job First* - Maior tarefa primeiro

Mpb - Mega pares de bases ou um milhão de pares de bases

NGS - *Next generation sequencer* - Sequenciador de próxima geração

NCBI - *National center for biotechnology information*

OpenMP - *Open Multi-Processing* - Multi-processamento aberto

pb - Pares de bases

PCR - *Polymerase chain reaction* - Reação em cadeia da polimerase

RNA - *Ribonucleic acid* - Ácido ribonucleico

SAET - *SOLiD accuracy enhancement tool* - Ferramenta de aprimoramento de precisão para *SOLiD*

SJF - *Shortest Job First* - Menor tarefa primeiro

SGI - *Silicon Graphics International Corporation* - Corporação *Silicon Graphics* Internacional

SNP - *Single-nucleotide polymorphism* - Polimorfismos de um único nucleotídeo

SOLiD - *Sequencing by oligonucleotide ligation and detection* - Sequenciamento por ligação de oligonucleotídeos e posterior detecção

SSD - *Solid-state disk* - Disco de estado sólido

T - Tinina

UV - *Ultraviolet* - Ultravioleta

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho propõe um escalonador de tarefas para controlar a demanda de envio de lacunas de genoma para o processamento obtidas pelo processo, considerando os recursos computacionais disponíveis. O objetivo do escalonador é evitar que sejam solicitados mais recursos computacionais do que os que podem ser fornecidos, pois nesse caso, o sistema sofre degradação de desempenho e causa atraso no tempo de processamento da tarefa. A motivação para este trabalho é a melhoria na eficiência da execução do fechamento de lacunas no sequenciamento de genomas. Para a avaliação da proposta, foi implementado um escalonador de lacunas com políticas de escalonamento baseadas no monitoramento dos recursos computacionais. Desta forma, utilizando o escalonador, a melhoria de desempenho na execução das lacunas foi de 56% no tempo de processamento com a implementação do escalonador e depois de 70% diferenciando o uso ou nao do paralelismo, comparando com o uso da solução original, resultado do uso mais eficiente dos recursos disponíveis.

Palavras-chave: Bioinformática, Escalonamento de Tarefas, Sequenciamento Genético.

ABSTRACT

The next-generation genetic sequencers extract a large amount of genetic data into small fragments, disordered and repetitive. Process this large volume of fragmented genetic data to assemble a long genome is a difficult and a large scale computational problem. This processing is performed with the aid of the tool denovo2, a pipeline of programs which performs genetic sequencing with fragments obtained by genetic sequencer. During the execution of denovo2, after assembly of sequence, phase improvement of the result performs the closing of the gaps found in the genome assembled. The program responsible for this closure is the ASID, that calls the assembler velvet with each gap four different times, varying the parameters to search for parts that may complete the gap. This activity generates many concurrent tasks that consume a large amount of computing resources, such as memory. This master thesis proposes a task scheduler to control the demand for sending these gaps for processing, considering the computational resources available. The purpose of the scheduler is to prevent more computational resources are requested than what can be supplied, because in that case, the system suffers performance degradation and causes a delay in the processing time of the task. The motivation for this work is to improve the efficiency of the implementation of the closure of gaps in genome sequencing. For the evaluation of the proposed, we implemented a scheduler gaps with scheduling policies based on monitoring of computational resources. In this way, using the scheduler, the performance improvement in the execution of the gaps was 56 % in processing time with the implementation of the scheduler and after 70 % distinguishing the use or not of parallelism, compared to using the original solution , a result of more efficient use of available resources.

Keywords: bioinformatics, Genetic sequencing, Scheduling Tasks.

CAPÍTULO 1

INTRODUÇÃO

As informações contidas nos genomas podem resultar em novos métodos de diagnósticos, na formulação de novos medicamentos, vacinas, na prevenção e tratamentos mais eficazes contra doenças ou pragas [6, 23]. O genoma é um roteiro orgânico molecular que traz em si todas as informações genéticas da evolução dos seres vivos, é a anatomia molecular de uma espécie armazenada pelo ácido desoxirribonucleico, do inglês *deoxyribonucleic acid - DNA* [51, 62]. O *DNA*, é formado por *monômeros* chamados nucleotídeos. Cada um destes nucleotídeos possui uma molécula denominada *base*, que pode ser *Adenina, Citosina, Timina* ou *Guanina* representadas pelas letras *A, C, T* e *G*, respectivamente.

O sequenciamento do genoma é a técnica utilizada para determinar em que ordem se encontram as bases contidas nos genes do *DNA*. Com o sequenciamento dos genomas, são obtidas informações sobre a linha evolutiva dos organismos possibilitando conhecer e comparar diferentes *DNAs*.

Os sequenciadores genéticos extraem do organismo alvo uma grande quantidade de pequenos fragmentos de *DNA* desordenados. A ordem dos fragmentos de *DNA* obtidas dos sequenciadores genéticos pode ser inferida por meio da informação de sobreposição entre o extremo da direita de um fragmento e o extremo da esquerda de outro [26]. As informações genéticas que compõem o código genético do organismo são obtidas pelo sequenciador em pequenos fragmentos, desordenados, repetitivos e em grande quantidade. Considerando o grande volume de fragmentos de *DNA* obtidos pelo sequenciador, a determinação da sequência de genomas só é possível com o uso de métodos de sequenciamento automáticos.

Estes métodos automáticos têm o objetivo de receber estes conjuntos de fragmentos e organizá-los de forma a montar uma sequência contígua *consensus*, que é o resultado do

alinhamento de múltiplas sequências de bases, onde cada base ocorre com mais frequência em uma certa posição [29], que representa a sequência de *DNA* completa do organismo. Um dos métodos automáticos de sequenciamento empregado atualmente é o *denovo2* [9], um *pipeline* de programas que, para montar a sequência *consensus* com os dados obtidos dos sequenciadores, utiliza três fases: pré-processamento de montagem, processamento de montagem e pós-processamento de montagem.

O pré-processamento de montagem tem como objetivo selecionar uma amostragem dos dados, realizar correção de erros e converter os dados para o formato utilizado pelo montador. O processamento da montagem é responsável por realizar a montagem por sobreposição de uma sequência com a amostragem obtidas da fase anterior. O pós-processamento de montagem realiza a melhoria dos resultados realizando o fechamento das lacunas (*gaps*) encontradas no sequenciamento; ao término da fase é gerado um relatório sobre a qualidade dos resultados.

Dependendo do conjunto de dados a ser processado, a execução do *pipeline denovo2* pode apresentar um consumo bastante elevado de memória, o que ocasiona um problema de degradação de desempenho da aplicação. Este problema pode ser verficado, por exemplo no processamento do conjunto denominado *Conjunto Treinamento1*[1] em uma máquina com 512 *GB* de *RAM*, onde toda memória disponível foi consumida, causando lentidão no sistema computacional como um todo e fazendo com que a execução da tarefa perdurasse por tempo indeterminado e longo.

Para descobrir a origem do problema realizou-se uma análise do comportamento do *pipeline denovo2* utilizando o *Conjunto Treinamento1*, na qual monitorou-se os recursos exigidos ao longo da execução. Foi constatado que o problema do uso de memória é causado na fase de pós-processamento da montagem, mais especificamente na etapa de fechamento de lacunas. Verificou-se que nesta etapa são executadas muitas tarefas ao mesmo tempo que acabam exigindo mais memória do que a disponível, causando o problema já relatado.

[1]Mais detalhes não foram divulgados por questões de sigilo de pesquisa

Uma possível solução para este problema é a execução de uma tarefa de cada vez (execução serial). Embora resolva o problema do uso de memória, esta abordagem não é eficiente pois a execução de uma única tarefa acaba deixando muitos recursos ociosos.

A solução que este trabalho propõe é o uso de um escalonador para coordenar a submissão das tarefas de acordo com a disponibilidade dos recursos, buscando um uso eficiente dos recursos disponíveis e uma possível redução no tempo de execução. Um escalonador é um gerenciador de recursos que utiliza políticas para controlar o acesso e o uso dos recursos pelos consumidores, determinando a ordem de execução das tarefas [13, 32]. Com o uso de um escalonador é possível conseguir um melhor desempenho global por tornar o uso dos recursos mais eficiente, de acordo com [5].

No escalonador proposto, ao invés de processar todas as tarefas ao mesmo tempo e sem nenhum critério como originalmente era feito, as tarefas são disparadas observando a disponibilidade dos recursos necessários para a sua execução. Dessa forma, diversas tarefas podem ser executas em paralelo consumindo os recursos de forma controlada, não causando o problema de degradação de desempenho.

Com o uso do escalonador foi possível controlar as tarefas para que utilizassem os recursos de forma eficiente e fosse possível executar diversas tarefas em paralelo. Com esta abordagem, o tempo de execução do *Conjunto Treinamento1* foi reduzido em 56% em relação ao tempo obtido na implementação original.

O restante do texto está organizado da seguinte maneira. No Capítulo 2 é feita uma apresentação sobre sequenciamento genético, o sequenciamento de próxima geração (*NGS*). O Capítulo 3 introduz o *pipeline denovo2* utilizado para gerar a sequência *consensus* de um genoma e identifica o problema do *pipeline denovo2*. O Capítulo 4 apresenta as estratégias utilizadas para o melhoramento de desempenho, tendo os resultados apresentados no Capítulo 5. Por fim, o Capítulo 6 apresenta as conclusões, e contribuições e trabalhos futuros, o Apêndice A traz com mais detalhes o funcionamento do *pipeline denovo2*.

CAPÍTULO 2

SEQUENCIAMENTO GENÉTICO

Este capítulo apresenta o sequenciamento *NGS* e o *pipeline denovo2* [9], que trabalha os dados obtidos por sequenciadores genéticos *NGS*. Neste caso, especificamente os dados gerados pelo sequenciador, para montar uma sequência genética *consensus* formada pelos caracteres que aparecem com mais frequência em um alinhamento que representa um genoma.

Na Seção 2.1 são apresentados conceitos referentes ao *DNA*, ao genoma e ao processo de montagem de genomas. A Seção 2.2 apresenta o sequenciamento *NGS* que utiliza os dados genômicos obtidos de sequenciadores de *NGS*, fragmentos de leituras curtas (*short reads*) com tamanho de até 45 *bp*, para realizar o sequenciamento por fragmentação randômica de *DNA* (*shotgun*). São também apresentados os conceitos envolvidos e as etapas realizadas no sequenciador que utiliza o método de sequenciamento por ligação de oligonucleotídeos e posterior detecção, do inglês *sequencing by oligonucleotide ligation and detection - SOLiD4*, que obtem os dados genômicos utilizados no *pipeline* de sequenciamento genético.

2.1 Montagem e sequenciamento do *DNA*

Cada uma das bases que compõem o *DNA* possui um complemento, a base A é complementada pela base complementar T, base C é complementada pela base complementar G, base T é complementada pela base complementar A e base G é complementada pela base complementar C.

O *DNA* é uma molécula orgânica disposta no forma de hélice de fita dupla, que armazena os genes e fatores hereditários e transferem as características através das gerações.

As duas fitas que compõe o *DNA* são reversamente complementares, a cada base tem-se a base complementar na outra fita, sendo que a orientação da fita complementar é inversa. A orientação das fitas é dada pela molécula carbono presente nas extremidades, iniciando na extremidade do carbono 5' e terminando na extremidade do carbono 3'. Quanto mais próximo da extremidade do carbono 3', menor é o índice de qualidade conseguido pelo sequenciador [51]. Pode se obter a fita complementar fazendo o complemento de cada base e invertendo a orientação, como demonstrado na Figura 2.1.

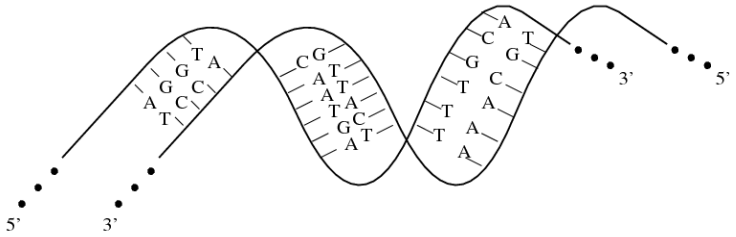

Figura 2.1: Hélice do *DNA*

O genoma resume todos os dados transmitidos de uma geração de seres vivos para outra, armazenados em um organismo através do código genético. O *DNA* é o código genético contido nas estruturas celulares de um corpo. O *DNA* organizado compõe o genoma, que é a totalidade dos genes presentes em um ser vivo. O gene é a parte funcional do *DNA*, são sequências especiais de centenas ou até milhares de pares (do tipo A-T ou C-G) que oferecem as informações básicas para a produção de todas as proteínas que o ser precise produzir. Para se referir ao tamanho de um genoma é utilizada a quantidade de pares de bases, do inglês *base pairs* - *pb*, medida geralmente em milhões de pares de bases - *Mpb*.

O objetivo do sequenciador é determinar a sequência de bases de um fragmento de *DNA* decodificando para os nucleotídeos A, G, T e C, que são distribuídos de forma não igual. O sequenciamento [24] de genomas é relativamente recente na Biologia Molecular, sendo que o primeiro genoma de um organismo não viral a ser sequenciado foi o da bactéria *Haemophilus influenzae*, em 1995 [1], com o tamanho de 1,8 *Mpb*. A *Escherichia coli*, um dos patógenos humanos mais importantes, foi sequenciada em 1997 [4] e encontra-se disponível como genoma completo no banco de dados público *GenBank* [63] do *National*

Center of Biotechnology Information (NCBI) sob o nome *Escherichia coli str. K-12 substr. MG1655*, de tamanho 4,6 *Mbp*, e pode ser buscada pelo *GenBank* accession code *U00096.2*.

O primeiro sequenciador automático utilizava a metodologia de Sanger [46], que liderou a pesquisa genômica por 30 anos. O método *Sanger* foi criado pelo bioquímico inglês *Fred Sanger*, vencedor de dois prêmios Nobel. Em 2005 surgiu o método de pirossequenciamento [21], utilizado no sequenciador *454* [49] da *Roche Applied Science* [48]. Outros métodos foram desenvolvidos a partir deste período, sendo o mais expressivo o método *Polony* [62] que é utilizado no sequenciador *SOLiD4* da *Applied Biosystems* [59], empresa pertencente ao grupo *Life Technologies* [60]. O uso deste método reduziu os custos do sequenciamento genético para aproximadamente 1/10 do custo do sequenciamento utilizando o método de pirossequenciamento e reduziu a taxa de erro dos métodos anteriores [62].

Com a evolução dos métodos de sequenciamento, também evoluiram os equipamentos de sequenciamento genético. Com isto se desenvolveram novas metodologias de alto rendimento e possibilitaram o surgimento dos Sequenciadores de Próxima Geração, do inglês *Next Generation Sequencers - NGS*, que reduziram o custo de operação e disseminaram a tecnologia de sequenciamento genético [23]. Um exemplo é a plataforma de sequenciamento genético da empresa *Applied Biosystems*, com a tecnologia denominada sistema *Sequencing by Oligonucleotide Ligation and Detection - SOLiD* [7], empregada no sequenciador *SOLiD4* [59].

No sistema *SOLiD* [30], o *DNA* é quebrado em fragmentos de *tags* duplas, com cobertura de 300 a 1000 vezes, o método *Sanger* trabalhava com cobertura entre 4 e 8 vezes. As bibliotecas resultantes contêm milhões de moléculas únicas representando a sequência alvo inteira. Este método provê uma grande economia de tempo e custo. A redução no custo do sequenciamento [20] através da introdução destas novas tecnologias, possibilitou que laboratórios menores tivessem os próprios projetos de sequenciamento. Com esta nova plataforma, o custo de sequenciamento genômico reduziu da casa dos milhões de dólares para os milhares e o tempo de sequenciamento de um organismo grande teve o

tempo reduzido de anos para meses [44].

A *mate pair library* é uma biblioteca de leituras de *tags* duplas que apresentam ambos os sentidos. O sentido normal e o reverso, que são representados pelas letras F e R, respectivamente. As leituras no formato *mate pair*, além da informação das bases, possuem informação sobre a estrutura do *DNA* e contém informação do posicionamento, o que permite alinhamento com maior precisão [4]. Para possibilitar uma boa montagem utilizando as milhares de leituras curtas, é preciso uma boa cobertura e uma grande sobreposição [50]. A Figura 2.2 demonstra exemplos de coberturas, a linha superior (mais grossa) representa o genoma completo e as linhas inferiores representam os fragmentos obtidos das leituras.

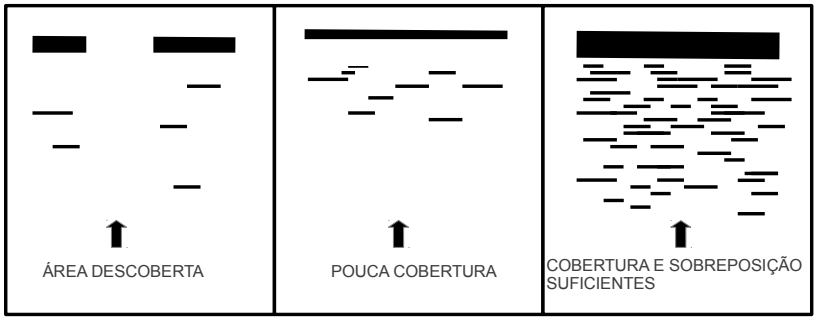

Figura 2.2: Sobreposição e cobertura para as montagens

O primeiro quadrante da Figura 2.2 demonstra um caso onde as leituras obtidas não fazem a cobertura total do genoma, possuí uma área sem cobertura que impossibilita a montagem. O segundo quadrante demonstra um conjunto de entrada onde a cobertura não é suficiente para garantir uma boa montagem; e por último, no terceiro um exemplo de um conjunto de entrada que possui leituras suficientes para uma boa cobertura do genoma alvo, possibilitando assim uma boa montagem. Seja qual for a estratégia de montagem, é importante conseguir uma cobertura adequada, isto é, não deixar nenhuma parte do *DNA* original sem representação no conjunto de fragmentos, e assegurar que haja sobreposição suficiente entre os fragmentos, para permitir uma montagem segura [50].

Estes fragmentos sobrepostos inserem novos conceitos necessários para compreensão dos resultados. A Figura 2.3 demonstra os conceitos de *scaffolds contigs* e lacunas ou

gaps. Os *scaffolds*, em ***1***, são trechos do genoma reconstruído com os fragmentos obtidos do método *shotgun*, compostos por *contigs* e lacunas (*gaps*). Um *contig*, em ***2***, é um trecho contínuo de sequência genômica formado pelos fragmentos representados em ***3***, onde a ordem das bases é conhecida e com o nível de confiança elevada. As lacunas, em ***4***, são lacunas com as bases desconhecidas mas a distância entre as extremidades conhecidas. Logo que os comprimentos dos fragmentos são conhecidos, por se trabalhar com *tags* duplas (*mate pair*) o número de bases entre os *contigs* pode ser estimado.

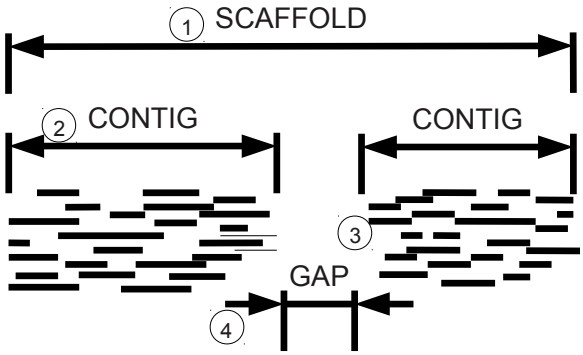

Figura 2.3: Conceitos de *scaffold, contig* e lacuna (*gap*)

2.2 Sequenciadores *next generation sequencer - NGS*

O sucesso das tecnologias de sequenciamento *NGS* [36, 47], não depende apenas da rapidez do processo de produção das leituras pelos modernos sequenciadores genéticos. Depende, e muito, da infraestrutura computacional [12, 19, 33] para oferecer os recursos para resolver o problema da montagem dos fragmentos e de um eficiente conjunto de algoritmos especializados para chegar a uma sequencia *consensus* de um organismo.

A alta demanda por baixo custo de sequenciamento tem impulsionado o desenvolvimento tecnologico de sequenciamentos de alto rendimento, *HTS*, que paralelizam o processo de sequenciamento, produzindo milhares ou milhões de sequências de uma só vez. A classe *NGS* revolucionou as pesquisas biológicas devido a geração de milhões de leituras de pequenas sequências *HTS* de 35 a 400 bases em um espaço curto de tempo e a um custo mais baixo [34].

Os principais desafios, no processamento de dados de sequências genéticas, são a taxa de erro relativamente elevada, entre 0,1% e 1%, por base, e do volume de dados gerado pelos sequenciadores. O sequenciador *SOLiD4* possui acurácia de 99,94%, com fragmentos de tamanho de 50 *bp* [7]. O *SOLiD4* é um sequenciador de alto rendimento, *HTS*, que produz grande quantidade fragmentos genéticos e apresenta alta precisão e velocidade, sendo um representante dos *NGS* [36]. Possui o poder de gerar informação muitas vezes maior que o sequenciamento anterior, o *Sanger*.

A complexidade em analisar as repetições, dificulta a montagem de genomas longos utilizando leituras curtas, de tamanho entre 25 e 50 bases. Nesse âmbito há um problema computacional difícil [11], por ser mais complexo trabalhar com a montagem de leituras curtas, pois há mais possibilidade de ocorrer sobreposição uma vez que estas regiões se repetem, o que diminui a probabilidade de unir corretamente estas sobreposições. A complexidade das sequências aumenta em um fator de quatro (*A, C, T* ou *G*) a cada base extra adicionada [55]. É intuitivo pensar que seria melhor trabalhar com leituras curtas, porém isto resulta em um aumento da probabilidade de se encontrar leituras redundantes [25].

Para obter os dados da plataforma *SOLiD* que servem de entrada para o *pipeline denovo2*, é necessário realizar antes algumas etapas [41]. Primeiramente preparar as bibliotecas, esta tarefa determina de qual forma o *DNA* será manipulado, podendo ser de dois tipos o *fragment library* ou *mate-paired library.* Em seguida, recebe emulsões de reação em cadeia da polimerase, do inglês *Polymerase Chain Reaction - PCR*, necessárias para ocorrer as ligações químicas e que amplifica uma ou algumas cópias da amostra obtida no passo anterior e se encontram anexadas a um componente metálico denominado *bead*; posteriormente a etapa de purificação, quando a emulsão é quebrada para liberação das *beads* de onde se removem amostras contaminadas. Feita a purificação, as *beads* amplificadas e purificadas são ligadas covalentemente a lâminas de vidro por meio dos adaptadores anexados nas etapas anteriores. Em seguida, as lâminas obtidas são submetidas ao sequenciador que faz a leitura da transição entre bases nucleotídicas da sequência

por meio da detecção dos fluorocromos azul *(FAM)*, verde *(Cy3)*, laranja *(TXR)* ou vermelho *(Cy5)*. Consequentemente os sinais de fluorescência podem ser convertidos em sinal colorimétrico [20]. Por fim os sinais de fluorescência [35] são especificados pela única combinação possível de cores que inclui a base conhecida.

Estas leituras estão codificadas em *color space*. Esse sistema de leitura é eficiente na detecção de polimorfismos *SNPs* [45], os quais são facilmente confundidos com erros de sequenciamento em outras plataformas. Estas ligações geram pontos coloridos nas lâminas dentro do sequenciador, que são fotografadas e analisadas. Para isto são utilizadas técnicas de análise computacional de imagens, trabalho realizado pelo próprio sequenciador em um *cluster* interno, das imagens internas são obtidos os dados utilizados para realizar o processamento das montagens. Cada sinal de fluorescência especifica um dinucleotídeo[1] e não uma única base (nucleotídeo), a decodificação dos sinais de leitura é feita combinando-se os dados.

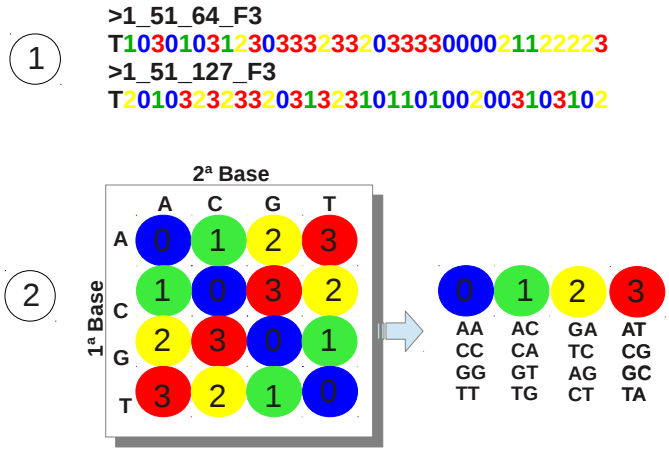

Figura 2.4: Conversão de formatos *color space* para *double encoded*

O *color space* [8] é representado pelos números *0, 1, 2,* ou *3* que identificam as cores lidas pelo sequenciador [39]. Na Figura 2.4, em ***(1)***, um exemplo de um arquivo no formato *color space - csfasta*. No formato *color space*, os valores representam as transições entre os nucleotídeos 0, 1, 2, 3 (também referido como cores azul=0, verde=1, amarelo=2 e vermelho=3), como mostrado em ***(2)*** na Figura 2.4.

[1]Cada cor identificada nas imagens, representa a transição de um nucleotídeo para outro. Então, cada cor representa dois nucleotídeos, ou um dinucleotídeo

CAPÍTULO 3

O *PIPELINE DENOVO2*

Este capítulo apresenta o *pipeline denovo2* que trabalha os dados produzidos pelo sequenciador genético para montar uma sequencia *consensus*. Os dados passam por três diferentes etapas no *denovo2*, pré-processamento de montagem, processamento da montagem e pós-processamento da montagem, apresentadas respectivamente nas Seções 3.1, 3.2 e 3.3. Para encontrar a origem do problema de degradação de desempenho que ocorre durante a execução do *pipeline* de sequenciamento genético *denovo2* e propor uma solução, a Seção 3.4 traz uma análise do *pipeline denovo2*.

Após obtidos os dados pelo sequenciador, é utilizado um *pipeline* de execução chamado *denovo2*. O *pipeline denovo2* é distribuído pela *Life Technologies* [9] e consiste em uma sequência de programas executados para realizar a montagem de uma sequência *consensus* dos dados de entrada, obtendo um genoma como saída. Além da fase de montagem que é a principal, existem as fases de melhoria do conjunto de dados de entrada e a melhoria do resultado de saída, que em conjunto, melhorara o resultado final. O pré-processamento seleciona uma amostragem dos fragmentos de sequências e prepara os dados no formato que o montador *velvet* trabalha, o processamento realiza a montagem dos fragmentos e o pós-processamento tenta realizar fechamento das lacunas encontradas [29]. O fluxo de trabalho do *pipeline denovo2* esta representado pelo diagrama ilustrado na Figura 3.1 adaptada de [9].

Como demonstra a Figura 3.1, o *pipeline denovo2* é consituído por três fases: ***1)*** pré-processamento, que tem como objetivo selecionar uma amostragem dos dados, realizar correção de erros e converter os dados para o formato utilizado pelo montador *velvet*; ***2)*** montagem: a principal fase do *denovo2*, montador utilizado é o *velvet* [65], responsável por realizar a montagem por sobreposição de uma sequência com os fragmentos obtidos do

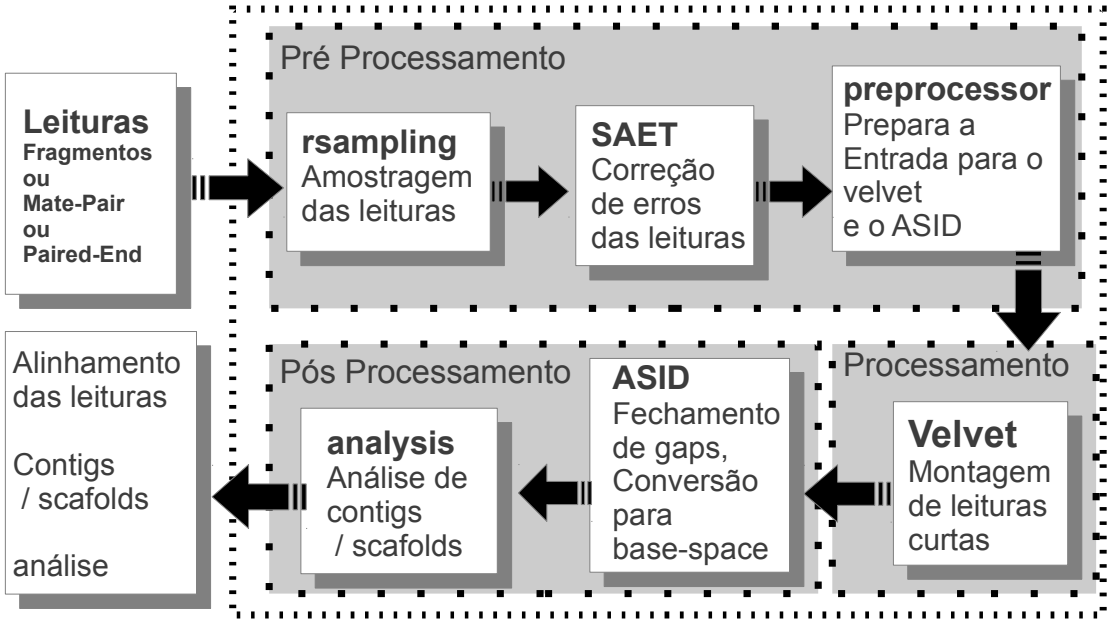

Figura 3.1: Diagrama *pipeline denovo2*

sequenciador genético; e ***3)*** melhoria e análise dos resultados onde é realizado o fechamento dos lacunas ou *gaps* encontrados no sequenciamento e gerado um relatório onde constam dados sobre a qualidade da montagem, índices como tamanho, quantidade e qualidade dos *scafolds* e dos *contigs*.

3.1 Pré-processamento da montagem do genoma

A fase de pré-processamento da montagem é composta por três etapas, onde são realizadas amostragem, melhoria e preparação para a montagem. Destas etapas, apenas a etapa de preparação é de execução obrigatória, pois converte os dados para o formato de dados utilizados no montador *velvet*, uma vez que os *velvet* não foi inicialmente desenvolvido para trabalhar com os dados obtidos do sequenciador *SOLiD4*.

Na primeira etapa da fase de pré-processamento da montagem do *pipeline denovo2* é executado o programa *SOLiD reads sampling - rsampling* [9], que seleciona uma amostragem randômica dos fragmentos para serem utilizadas na montagem. Geralmente as entradas possuem uma taxa de cobertura elevada, entre 600 e 1000 vezes, entretanto, uma cobertura maior do que 300 vezes ocasiona uso de memória elevado e um ganho

no resultado final insignificante [9]. Por este motivo, é realizado o ajuste no arquivo de entrada para uma cobertura máxima de 300 vezes.

A segunda etapa do pré-processamento executa o programa *SOLiD Accuracy Enhancement Tool - SAET* [58], responsável por realizar o melhoramento de precisão, efetuando correção de erros dos arquivos de entrada na fase de pré-montagem. O *SAET* reduz a taxa de erro que é entre 3% e 5% para menos de 1% e aumenta o índice de qualidade *N50* em três vezes [9, 58]. O *N50* é um índice que significa que pelo menos 50% do genoma está representado em *scaffolds* ou *contigs* de tamanho igual ou maior que o indicado por este índice. Como o *SAET* é um programa proprietário e fechado, sem os fontes disponíveis, não é possível saber detalhes da correção realizada.

Na terceira etapa, o *preprocessor* tem como objetivo preparar os dados para serem utilizados como entrada da próxima fase do *denovo2*. Os fragmentos obtidas do sequenciador *SOLiD4* são no formato *color space*. O montador *velvet*, próxima fase do *pipeline*, trabalha com dados no formato *double encoded*, então o *preprocessor* prepara os dados para o formato utilizado pelo *velvet*.

A Figura 3.2 ilustra este processo. O quadrante **(a)** demonstra um exemplo de arquivo *mate pair* em *color space* resultante do sequenciador *SOLiD4*. Os arquivos *F* e *R* do *mate pair* são concatenados de forma intercalada, como mostrado no quadrante **(b)**. Na transição para o quadrante **(c)**, é realizada a conversão para o formato *double encoded* utilizando a seguinte regra, *0=A, 1=C, 2=T, 3=G* e *4=N*, onde *N* indica uma base não identificada. Em seguida, esta conversão elimina a primeira base e realiza a inversão de sentido dos fragmentos referentes ao par *F*.

Apesar de serem opcionais, é indicada a execução das etapas *rsampling* e *SAET* da fase de pré-processamento da montagem. A etapa *preprocessor* é necessária, pois faz a conversão do formato de dados de saída do sequenciador *SOLiD4* para o formato que o *velvet* trabalha para realizar a montagem.

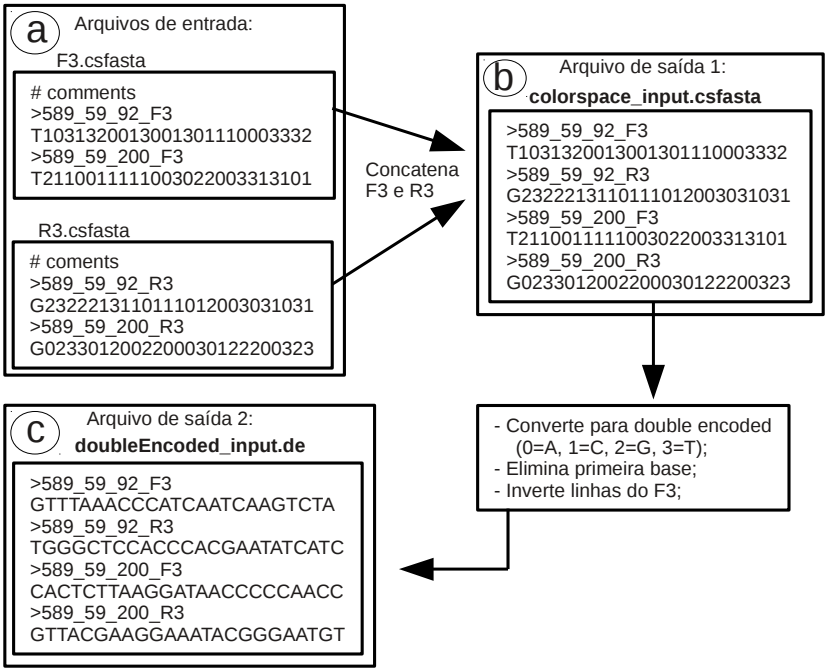

Figura 3.2: Conversão realizada pelo *preprocessor*

3.2 Processamento da montagem do genoma

A fase de processamento da montagem é a principal função deste *pipeline* e é responsável pelo encaixe dos fragmentos em busca de uma sequencia *consensus* única, resultando na montagem do genoma alvo.

O montador *velvet* [65] é um conjunto de vários algoritmos com a função de realizar montagens de genoma com fragmentos curtas [38, 15, 16]. O *velvet* foi inicialmente desenvolvido para o *Illumina* [27] e posteriormente adaptado para a plataforma *SOLiD*, apesar de ser muito utilizado e de produzir bons resultados, é conhecido por consumir grandes quantidades de memória *RAM* [14, 40].

O *velvet* é dividido em dois módulos, o *velveth* e o *velvetg*. O *velveth* gera a partir dos fragmentos de entrada, pequenos pedaços ímpares chamados *k-mers*. Os *k-mers* são gravados apenas uma vez, a quantidade de vezes que eles aparecem e os respectivos caminhos para outros *k-mers* são apenas referenciados no grafo e não explícitos [65]. Com isto, são gerados os arquivos *Roadmap* e *Sequences* que são a saída do *velveth* e a entrada para o *velvetg*. O *Sequences* contém as leituras ou fragmentos convertidos para um formato

interno de *velvet* de onde serão criados os *k-mers* e o *Roadmap* é um roteiro para uso interno do *velvet* que é utilizado pelo *velvetg* para montar o genoma. O *velvetg* utiliza os arquivos *Roadmap* e *Sequences* para montar os grafos que são trabalhados para realizar a montagem.

A principal característica deste montador de leituras curtas, que o distingue de montadores tradicionais é o uso do grafo de *Bruijn - DBG* [57] para o processo de comparação e montagem das leituras. O *DBG* suporta bem a repetição de dados, comum nos genômas, como demonstrado na Figura 3.3 adaptada de [57], que exemplifica o armazenamento destas repetições passo a passo.

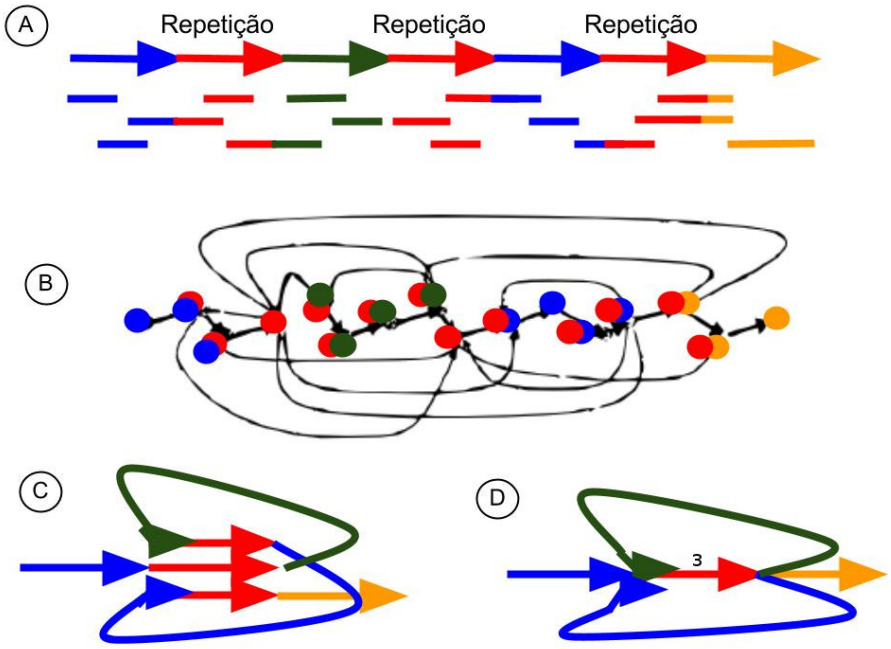

Figura 3.3: Repetições no grafo de *bruijn*

Inicialmente em ***(A)*** um representação do genoma com três repetições; em ***(B)*** uma representação onde cada leitura corresponde a um vértice no gráfico de sobreposição, e dois vértices estão ligados por uma aresta se as leituras correspondentes sobrepõem. Para realizar a montagem seria preciso visitar cada vértice exatamente uma vez, um Problema do Caminho Hamiltoniano. O Problema do Caminho Hamiltoniano é NP-completo [43] e por isto de difícil solução. Em ***(C)*** os passos para montar o *DBG* proposto por [57] que reduz o problema ao Problema do Caminho Euleriano [11] em ***(D)***, de solução mais

simples, onde estão armazenados os mesmos dados que em ***(B)*** mas de forma mais simples e eficiente.

Antes de montar o *DBG*, são criados pré-grafos como estratégia mais eficiente em relação ao uso de mémoria [16]. O *velvet* possui mecanismos para remoção de erros ou repetições que prejudiquem o percurso no grafo [38], o *DBG* é construído com os dados da sequência sem correção, e posteriormente os erros são corrigidos [15, 16, 65]. A complexidade do grafo para grandes organismos e com cobertura elevada é problemática. O módulo *velvetg* cria e manipula o *DBG* [38], para isto percorre o gráfo de diferentes maneiras, de acordo com os parâmetros informados pelo usuário [65]. A edição do grafo foi proposta inicialmente por *Pevzner* [57] e tem como objetivos a remoção dos erros, a edição de repetições de cópias idênticas em uma única aresta e, remoção de pontas e arestas com baixa cobertura, chamadas de ligações espúrias. Apesar de utilizar mais memória [14] que algoritmos similares, o *velvet* produz *contigs* maiores que os montadores *SSAKE, Warren2007*, *VCAKE* [37] e que o *SHARCGS* [10].

3.3 Pós-processamento da montagem do genoma

Realizada a montagem do genoma, é dado início à fase de melhoramento e análise dos resultados, a fase de pós-processamento da montagem. O melhoramento é realizado pelo *ASiD*, e em seguida um *script* faz a análise dos resultados informando índices sobre o resultado final.

O *ASiD* tem como o objetivo identificar e tentar solucionar as lacunas presentes na motagem do genoma. Identificados os trechos que contenham lacunas entre *contigs* e *scafolds*, o *ASiD* invoca uma instância do *velvet* para cada uma das lacunas, o objetivo é tentar realizar o fechamento. Para entradas *mate pair*, o *ASiD* aumenta o valor do *N50* em outro fator de três vezes, além do aumentado pelo *SAET*, e amplia o comprimento dos *contigs* [9, 11, 42].

A etapa final, a *anlisys* consiste em gerar estatísticas do genoma obtido, tarefa é

realizada por um *script* que fornece valores como o *N50* e *max*, usados para verificar a qualidade do resultado final. O *N50* é um índice de qualidade onde o maior N tal que 50% do total de pares de base do genoma esteja contida em *contigs* ou *scaffolds* maior ou igual N *bp*. Isto é, organizando os *contigs* por ordem de tamanho, do maior para o menor, somando o tamanho do maior *contig* até somar 50% do tamanho total do genoma. O último *contig* adicionado representa o valor *N50*, como demonstrado na Figura 3.4 elaborada pelo autor.

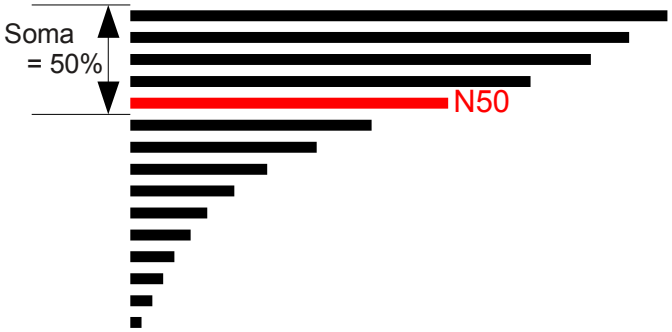

Figura 3.4: Índice de qualidade *n50*

Por exemplo, uma coleção de *contigs* com tamanhos 7, 4, 3, 2, 2, 1, e 1 *kb*, onde o tamanho total é indicado pela soma dos *contigs*, totalizando 20 *kbp*. O índice *N50* seria representado pelo *contig* de tamanho 4, pois ordenado do maior para o menor, e somando os tamanhos dos *contigs*, quando somar o valor 4 será atingido mais que 50% do tamanho total do *contig*, este último *contig* então representa o *N50*. Podem ser cobertos 10 *kbp* com *contigs* maiores que 4 *kbp*. Estes valores, de *N50* e *max*, são utilizados para verificar a qualidade da montagem final. Para exemplificar, na Figura 3.5 o *n50.stats* do *nt_contigs* da bactéria.

```
perc A                : 24
perc C                : 25
perc G                : 25
perc T                : 24
perc N                :  0
Sum contig length     : 4506076
Num contigs           : 810
Mean contig length    : 5563
Median contig length  : 2892
N50 value             : 11357
Max                   : 52043
```

Figura 3.5: Exemplo do índice *N50* da *E.coli*

Os valores que indicam a porcentagem de cada uma das quatro bases (A,C,T e G), vistos na Figura 3.5, o índice $G+C$ da *E. coli* de 50%, o que demonstra um balanceamento entre as bases $A+T$ e $G+C$ [4, 16].

O Apêndice A complementa este capítulo apresentando de forma mais detalhada a forma de uso do *pipeline denovo2*, partindo desde o seu *download*, instalação, execução e maiores detalhes sobre o uso e função de cada uma das fases.

3.4 Identificando o problema de exaustão de recursos no *pipeline denovo2*

Dependendo do conjunto de dados a ser processado, a execução do *pipeline denovo2* pode consumir toda memória disponível, causando lentidão no sistema computacional como um todo e fazendo com que a execução da tarefa seja prolongada indefinidamente.

Para identificar a origem do problema, foi necessário realizar o acompanhamento do trabalho realizado por cada fase do *pipeline denovo2*. Para isto foi utilizada uma máquina de grande porte chamada *Altix-UV 100* que disponibiliza, na sua atual configuração, um total de 256 *GB* de memória *RAM* e 64 núcleos de processamento. Foram executados dois experimentos nos quais utilizou-se o conjunto de dados da bactéria *Escherichia coli*, obtido do banco de dados genéticos público *GenBank* [63], e o conjunto de dados denominado *Conjunto Treinamento1*, fornecido pelo Departamento de Bioquímica da *UFPR*. Para o segundo conjunto de dados não foram disponibilizadas mais informações devido a questões de sigilo de pesquisa.

O primeiro conjunto de dados de entrada utilizado para avaliar execução do *pipeline* foi o da bactéria *Escherichia coli*, com o tamanho de 57,2 *Mpb*. A execução do *pipeline* com este conjunto de dados consumiu apenas 1 *gigabyte* de memória *RAM* e executou normalmente até o término das 4664 tarefas da fase de fechamento de lacunas, que ocupam apenas 2 *MB* de espaço em disco. Portanto, os testes com o conjunto de dados da *E.coli* não permitiram confirmar o motivo do atraso na execução. Toda execução do *denovo2*

utilizando como entrada os dados da *E.coli* está documentada no Apêndice A.

O segundo conjunto de dados, *Conjunto Treinamento1*, possui o tamanho de 111,6 *Mpb*, contém 4756 lacunas que ocupam 8 *GB* de espaço em disco, e resulta em uma saída de 300 *GB* de dados. Com a utilização deste conjunto de dados a execução ocorreu normalmente nas etapas de pré-processamento de montagem e de montagem, no entanto, na fase de pós-processamento de montagem foi possível verificar o consumo da totalidade da memória disponível, o que causou degradação de desempenho na máquina. Desta forma, foi possível identificar que o problema é causado na última fase do *pipeline*.

Dentro da fase de pós-processamento, constatou-se que a etapa de fechamento de lacunas é a responsável pelo problema. A etapa do fechamento de lacunas é realizada pelo programa *ASiD*, representado na Figura 3.6, que para realizar esta tarefa executa o montador *velvet*. Para cada uma das lacunas encontradas o *ASiD* executa um *velvet*, para tentar realizar a montagem dos trechos que ficaram sem representação de bases. Este procedimento, repetido 4 vezes com a variação destes parâmetros, é o responsável por disparar uma grande quantidade de tarefas para serem executadas simultaneamente em *background*.

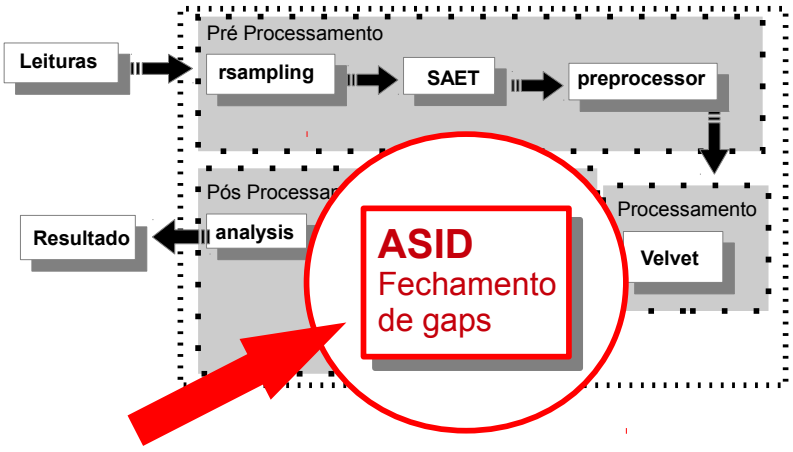

Figura 3.6: Etapa *ASiD* da fase de pós-processamento da montagem, onde foi identificado o problema de exaustão dos recursos

O *ASiD* é dividido em duas partes, os algoritmos *asid_ assemble.sh* e o *asid_ one.sh*. O Algoritmo 3.1 demonstra como o *asid_ assemble.sh* trabalha, repassando todas as lacunas

uma a uma para o *asid_ one.sh*. O Algoritmo 3.2 apresenta o *asid_ one.sh* onde cada lacuna é executada em uma instância. Cada instância do Algoritmo 3.2 executa um *velvet*, sendo executado primeiro o *velveth* e em seguida o *velvetg*, para cada uma das lacunas.

Algoritmo 3.1 Algoritmo *asid_ assemble.sh*

```
GAPSDE ← lacunas;
para todos lacunas em GAPSDE faça
  Execute em background asid_assembly_one.sh para lacuna[i]
fim para
```

Algoritmo 3.2 Algoritmo *asid_ one.sh*

```
Recebe lacuna de asid_assemble.sh
Execute velveth(lacuna) depois
Execute velvetg(lacuna)
```

Para cada uma das 4 chamadas ao *velvet*, sendo 64 processadores e 4756 lacunas para serem processadas é possível estimar, para a forma *multithread*, aproximadamente 300.000 *threads* simultâneas no caso do *Conjunto Treinamento1*. Esta grande quantidade de tarefas simultâneas exige alocação de memória para manter as lacunas em execução. Cada lacuna pode ocupar de poucos *megabytes* até alguns *gigabytes* de memória. Nesta etapa os 512 *GB* de memória *RAM* disponíveis não são suficientes e são utilizados os 8 *GB* de *swap*. De acordo com [56], ao utilizar trocas na memória virtual, *swapping*, a velocidade de acesso aos dados cai drasticamente. O consumo excessivo de memória *RAM* pela grande quantidade de tarefas em execução e a necessidade de escrever e ler muitos dados do disco rígido causada pelo mecanismo de paginação, fazem com que os processos fiquem esperando muito tempo pelo uso da *CPU*. Este problema é conhecido como *thrashing*, que geralmente provoca sérios problemas de desempenho tornando o sistema inutilizável.

Trashing [17] é o fenômeno que ocorre quando é realizada paginação excessiva reduzindo a utilização de *CPU* e a vazão. Como fator agravante do *trashing* o sistema operacional detecta que a *CPU* está ociosa (durante o *ASiD* o uso de *CPU* fica abaixo de 1%) e admite mais processos aumentando o nível de multiprogramação, e consequentemente, a taxa de falha de páginas, piorando o desempenho. O gráfico da Figura 3.7

adaptado de [18] demonstra como o *trashing* afeta o desempenho, o eixo X mostra o aumento do nível de multiprogramação, enquanto o eixo Y demonstra o rendimento da execução. Enquanto o esperado é um aumento de desempenho com o aumento do nível de multiprogramação, o rendimento cai subitamente após uma carga crítica [18].

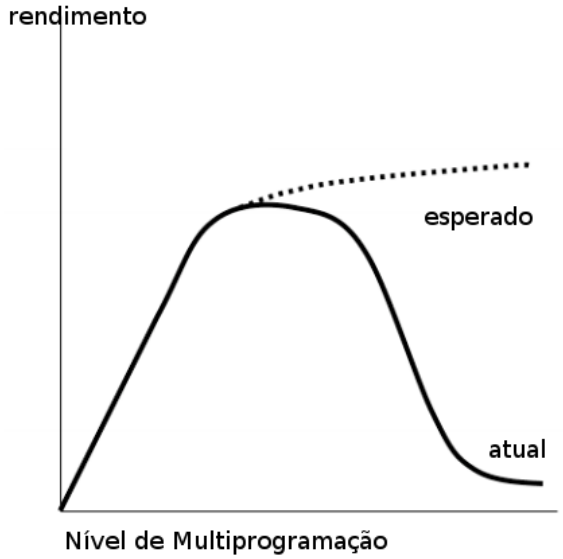

Figura 3.7: Comportamento do *trashing*.

Com os experimentos realizados foi possível constatar que o problema é a grande quantidade de tarefas sendo executas em paralelo. Uma maneira de resolver este problema é organizar a instanciação destas tarefas levando em conta a disponibilidade de recursos computacionais. Na Capítulo 4 apresenta-se um escalonador de tarefas para tal fim.

CAPÍTULO 4

IMPLEMENTAÇÃO DE UM ESCALONADOR DE TAREFAS PARA FECHAMENTO DE LACUNAS NO SEQUENCIAMENTO GENÉTICO

Neste capítulo são apresentadas as soluções para os problemas identificados no Capítulo 3 e uma implementação que atende aos requisitos identificados para a solução do problema.

O objetivo deste trabalho é tornar possível uma execução completa do *pipeline denovo2* sem sobrecarregar o computador ou ocasionar erros. A primeira tetantiva para resolver este problema foi reduzir o número de *threads* utilizada pelo *velvet*. Foi avaliado um *velvet* alternativo na etapa do *ASiD* que utiliza apenas uma *thread* por instância, reduzindo o número total de tarefas sendo executadas simultaneamente. Com esta abordagem a execução pode ser finalizada, no entanto o problema de *trashing* não foi totalmente resolvido.

Outra abordagem utilizada para este problema foi a execução de uma tarefa de cada vez. Ao invés de serem submetidas todas as tarefas simultaneamente, esta abordagem os submete de forma sequencial, uma após o término da anterior. Desta forma, cada tarefa tem à sua disposição todos os recursos da máquina, no entanto, os recursos computacionais ficam subutilizados. Embora resolva o problema do uso de memória, esta abordagem não é eficiente, uma vez que a execução de uma única tarefa acaba deixando muitos recursos ociosos.

Com a técnica de escalonar as tarefas de acordo com recursos computacionais disponíveis, foi atingido o objetivo de tornar possível uma execução completa, abordou-se a possibilidade de minimizar o tempo de execução. Uma forma de possibilitar diminuição

de tempo de execução é aproveitar de forma mais eficiente os recursos computacionais disponíveis. Uma possível estratégia para melhorar o aproveitamento de recursos é permitir que diversas tarefas possam ser executadas simultaneamente, verificando a quantidade de recursos disponíveis antes de lançar novas tarefas. O objetivo aqui é fazer com que as tarefas que estão sendo executadas não solicitem mais recursos do que os disponíveis, obtendo assim um melhor uso da memória disponível e evitando o *trashing*. O uso de um escalonador de tarefas poderia atender este objetivo, e foi a abordagem adotada.

4.1 Escalonadores de tarefas

É considerado um escalonador de tarefas um componente que faz a gerência de recursos. É de extrema importância para sistemas paralelos e distribuídos, podendo ser considerado um dos problemas mais desafiantes desta área. Escalonar consiste em determinar em que ordem as tarefas serão executados.

Um escalonador é a ferramenta que permite realizar o controle das tarefas de acordo com as políticas, restrições apresentadas [13, 32]. O problema básico do escalonador é satisfazer objetivos como obter tempo de resposta rápido, aumentar a vazão, evitar a espera indefinida e conciliar tarefas de alta e baixa prioridade [56]. Alguns critérios de escalonamento podem ser maximizar a utilização do processador, maximizar a produção do sistema (vazão), minimizar o tempo de execução (*turnaround*)[31, 54, 64].

O escalonamento das tarefas envolve três principais componentes: consumidores, recursos e política. As tarefas esperando para serem executados são os consumidores. Os recursos são os recursos computacionais disponíveis, como a memória, o disco e o processador. Por fim, a política de escalonamento é o conjunto de regras utilizado para determinar quando e qual tarefa deverá ser executado. A política definida pelo escalonador impacta diretamente no desempenho da aplicação que se esta controlando. As políticas utilizadas no escalonador proposto são a previsão de uso de memória e a taxa de acesso ao disco. O escalonador dispara uma tarefa quando a memória disponível for maior que a quantidade

de memória prevista para a execução da tarefa e o acesso ao disco for menor que um determinado limiar. Como descreve [2], as características de cada tarefa impactam na performance do escalonamento das tarefas. A principal característica das tarefas a serem processados por este trabalho é a de serem tarefas quem consomem bastante memória e realizam um grande volume de entrada e saída, sendo caracterizadas principalmente como *memory-bound* mas também como *I/O-bound*.

4.2 Algoritmos de escalonamento de tarefas

Os algoritmos de escalonamento utilizam critérios para criar uma fila das tarefas a serem processadas, os principais algoritmos são o *First Come First Serve - FCFS* onde o primeiro a chegar é o primeiro a ser processado, é o mais simples e mais fundamental algoritmo de escalonamento. O *Earliest Deadline First - EDF* é uma política que programar todos os trabalhos recebidos, de acordo coma data de vencimento especificada ou prazo. Os trabalhos recebidos serão processados ??ou colocados na fila de acordo com a cronologia do prazo para o término. O trabalho com o menor prazo será colocado em primeiro lugar na fila de processamento. No *Shortest Job First - SJF* que vem primeiro é a tarefa menor, com menor tempo de execução. Os trabalhos estão na fila com o tempo de execução menor colocado em primeiro lugar e o trabalho com a mais longo tempo de execução colocado passada e dada a prioridade mais baixa. Em casos extremos, quando os trabalhos com tempo de execução mais curtos não para de chegar, os trabalhos com tempo de execução mais longo pode nunca ter a chance de executar, esperarando indefinidamente. No *Longest Job First - LJF* o comportamento é o inverso do *SJF*. Embora reduza o tempo de resposta, o processamento das tarefas maiores primeiro minimiza o tempo de *makespan* [3].

4.3 Previsão de uso de memória para fechamento de lacunas

Identificar a quantidade de memória que uma lacuna irá requisitar durante o seu processamento não é uma tarefa trivial. Uma estratégia é relacionar o espaço ocupado em disco com o consumo de memória de cada lacuna, mas é possível que conjuntos de dados diferentes ocupem o mesmo espaço em disco que outros conjuntos. Então duas lacunas com o mesmo espaço ocupado em disco podem ter uma grande diferença no tempo de execução e na memória consumida.

Para tentar prever o uso de memória durante a execução das lacunas, foram selecionadas 4 amostras de lacunas para 4 diferentes intervalos de tamanhos. As amostras foram classificadas de acordo com o tamanho ocupado em disco em 4 diferentes grupos: pequeno (de 0 a 17 *MB*), médio (de 17,1 a 37 *MB*), grande (de 37,1 a 77 *MB*) e muito grande (acima de 77,1 *MB*).

Para cada uma destas classes foi calculada a média de uso de memória *RAM* e somado o desvio padrão, de modo que estes dados projetem a expectativa máxima do uso de memória, como demonstra a Tabela 4.1. Com estes valores foi criada uma base de dados de valores estimados que cada classe de lacuna utiliza durante o processamento. Estes valores são utilizados na política de decisão do escalonador para avaliar se submete novas lacunas ou, se aguarda até liberar a quantidade necessária de memória.

Tabela 4.1: Expectativa de uso de memória - *ASiD*

	Pequeno	Médio	Grande	Muito Grande
Tamanho em disco (*MB*)	17	37	77	112
Média de Uso de Memória (*GB*)	3,5	10	32	80
Desvio Padrão	0,5	3	3	5
Previsão de Uso de Memória *RAM* (*GB*)	4	13	35	85

4.4 Atraso no acesso ao disco durante o fechamento de lacunas

Como a quantidade de leituras e escritas no disco é bastante elevada durante todo o tempo de execução do fechamento das lacunas, o disco atinge taxas de utilização elevadas.

Quando o acesso ao disco está saturado, o dispositivo se torna um gargalo, reduzindo o desempenho do sistema. Logo, este é mais um ponto a ser monitorado e utilizado como política para o escalonador. O índice de saturação do disco pode ser consultado pelo escalonador de forma dinâmica sempre antes do envio de uma nova tarefa.

4.5 Proposta de escalonamento de lacunas

O escalonador proposto utiliza as informações apontadas ao longo do Capítulo 3 como políticas de escalonamento. Ao receber uma nova lacuna para ser processada, o escalonador já conhece a expectativa de uso máximo de memória *RAM* para aquela lacuna. A memória livre da máquina naquele momento é então consultada para verificar se existe a possibilidade de executar a lacuna sem extrapolar os limites de memória disponível. Caso a memória possa alocar aquela lacuna, realiza-se uma consulta da vazão de acesso ao disco. o Algoritmo 4.1 demonstra a implementação deste procedimento.

Algoritmo 4.1 Implementação de algoritmo para escalonamento das lacunas do *ASiD*.

```
limite_io ← 30%
para lacuna[n] em vet_lacunas faça
  lacuna[n].mem_prevista ← tamanho(lacuna.classe)
  fila_lacunas.insere(lacunas[n], ASiDx)
fim para
enquanto fila_lacunas ≠ ∅ faça
  acesso_disco ← sistema.taxa_disco
  memoria_disponivel ← sistema.memoria_livre
  se ((acesso_disco) > (limite_io)) então
    Esperando I_O
  senão
    se (memoria_disponivel > lacuna[n].mem_prevista) então
      Executa(ASiDx, lacunas.[n])
    fim se
  fim se
fim enquanto
```

Realizadas estas duas verificações, de memória e de disco, caso tenham suas condições satisfeitas, o escalonador pode submeter as lacunas para ser executado. Em seguida, o escalonador recebe outra lacuna e realiza este mesmo processo de verificação antes de submetê-lo à execução. Algumas tarefas sofrem atraso de submissão, mas é garantida

toda execução dentro dos limites da capacidade da máquina, garantindo uma execução mais efetiva resultando em ganho de desempenho.

4.6 Implementação de um escalonador de tarefas para fechamento de lacunas no sequenciamento genético

Com base nos dados até aqui apontados e nos testes realizados, foi possível elencar pontos principais para desenvolver uma solução. Baseado nestes pontos e testes, foi implementada uma solução mais eficiente do que a utilizada no *denovo2* na fase do *ASiD*. A implementação da solução é um escalonador e passa a ser denominado *asid_dispatcherUFPR.py*. Um *script* escrito em *Python* que utiliza *threads* para instanciar novas lacunas a partir de uma fila de prioridades. Novas tarefas são lançadas de acordo com critérios das políticas de escalonamento.

Ao invés de submeter todas as lacunas sem nenhum critério, como faz o *asid_assembly.sh*, a ideia é coordenar a submissão das tarefas. O escalonador *asid_dispatcherUFPR.py* analisa os pontos apresentados no decorrer deste capítulo para decidir se instancia mais uma lacuna para ser processado ou se aguarda para não sobrecarregar o sistema, atrasando o início de algumas tarefas, mas evitando todos os problemas de depreciação de desempenho relatados.

Um vetor inicialmente recebe todos os arquivos que representam as lacunas encontradas, estas lacunas são colocados em um fila de prioridades de acordo com o tamanho em disco que ocupam. Ao serem enfileirados de acordo com a classe de tamanho a que pertecem, carregam consigo a estimativa de uso de memória para o seu processamento, permitindo um controle virtual de uso de memória. O controle virtual de memória é realizado com uma varíavel de controle que recebe a quantidade de memória disponível do sistema e, ao lançar um nova lacuna decrementa o valor estimado do uso de memória daquela lacuna, e ao término incrementa o valor. O objetivo é nunca exceder a memória disponível do sistema, para não entrar no uso de *swap* e concorrência excessiva, evitando

trashing e a degradação de desempenho.

Este capítulo abordou os pontos necessários para otimizar o resultado da fase de fechamento de lacunas realizado pelo *ASiD*. Foi apresentado um escalonador que implementa estes pontos e assume a tarefa antes realizada pelo *asid_assembly.sh*.

CAPÍTULO 5

AVALIAÇÃO EXPERIMENTAL

Este capítulo apresenta os resultados alcançados com este trabalho. São comparados os resultados de desempenho originais com os resultados obtidos com a proposta, considerando as melhorias sugeridas.

Foram analisadas as execuções do *script* original do *ASiD*, denominado *asid_assembly.sh*, e comparadas com as execuções da implementação do escalonador de tarefas proposto, denominado *asid_dispatcherUFPR.py*. Ambos utilizaram como conjunto de testes as lacunas resultantes da execução do *Conjunto Treinamento1* no *pipeline* de sequenciamento genético *denovo2*. Ao executar o *denovo2*, as lacunas encontradas durante a execução ficam armazenados na pasta *.postprocessor gap_reads*.

Como durante a execução do *pipeline* o *script assemble.pl*, que de fato é o *denovo2*, executa o fechamento das lacunas quatro vezes consecutivas para cada lacuna, alterando apenas os parâmetros passados para o *ASiD* foi utilizado um pequeno programa em *shell script* para agrupar as execuções consecutivas. O escalonador de tarefas *asid_dispatcherUFPR.py* executa para cada lacuna as quatro variações de parâmetros ao mesmo tempo em diferentes *threads*, tirando algum proveito da localidade dos dados.

A Seção 5.1 apresenta o ambiente computacional onde foram realizados os experimentos, a Seção 5.2 começa a apresentar os resultados obtidos que são detalhados nas Subseções 5.2.1, 5.2.2, 5.2.3 e 5.2.4.

5.1 Ambiente de experimentação

O ambiente computacional onde os experimentos foram realizados é constituído de uma máquina de grande porte *Altix-UV 100* [52] da *Silicon Graphics Internacional - SGI*. A *Altix-UV 100* tem como diferencial a moderna arquitetura de memória compartilhada, *cache coherence Non-Uniform Memory Access - ccNUMA* [22] e possibilita grande escalabilidade. Sua configuração atual é de 512 *GB* de memória *RAM* e 64 núcleos de processadores providos por 8 pastilhas de processadores *Intel Xeon Processor E7-8837* de 8 núcleos com 24 *Megabytes - MB* de memória *cache* e 2.66 *Gigahertz - GHz* de frequência cada. É possível escalar esta máquina para até 12 *Terabytes - TB* de memória *RAM* e 96 pastilhas de processadores, o que disponibilizaria 768 núcleos.

O sistema operacional utilizado é o *GNU/Linux* com a distribuição *SUSE* na versão *Enterprise Server 11 SP1* da arquitetura *x86_64* e o sistema de arquivos é o *third extended file system - ext3*. As ferramentas utilizadas para monitorar a memória e o disco foram o *free* e o *iostat*, ambas disponíveis por padrão no sistema operacional *GNU/Linux*.

5.2 Avaliação dos resultados obtidos

Para avaliar os resultados obtidos foi realizada uma análise comparativa do comportamento dos recursos computacionais durante a execução do *asid_assembly.sh* original com a execução do *asid_dispatcherUFPR.py*, que implementa nas políticas de escalonamento os pontos levantados durante este trabalho que poderiam resultar em melhoria de desempenho.

As ferramentas utilizadas no monitoramento dos recursos são todas ferramentas do próprio sistema operacional *GNU/Linux*. Os resultados foram obtidos observando os *logs* gerados durante a execução de cada um das duas alternativas para o fechamento das lacunas pelo *ASiD*.

5.2.1 Comparação de uso de memória *RAM*

O gráfico da Figura 5.1 demonstra em vermelho o uso de memória durante a execução do *script* original *asid_assembly.sh* e em azul a implementação alternativa *asid_dispatcherUFPR.py*. No eixo *Y* está representada em *GB* a quantidade de memória *RAM* utilizada nas execuções ao longo do tempo em horas, que é representado no eixo *X*. A linha em verde indica o limite em *GB* da memória disponível, valor que nunca deveria ser extrapolado.

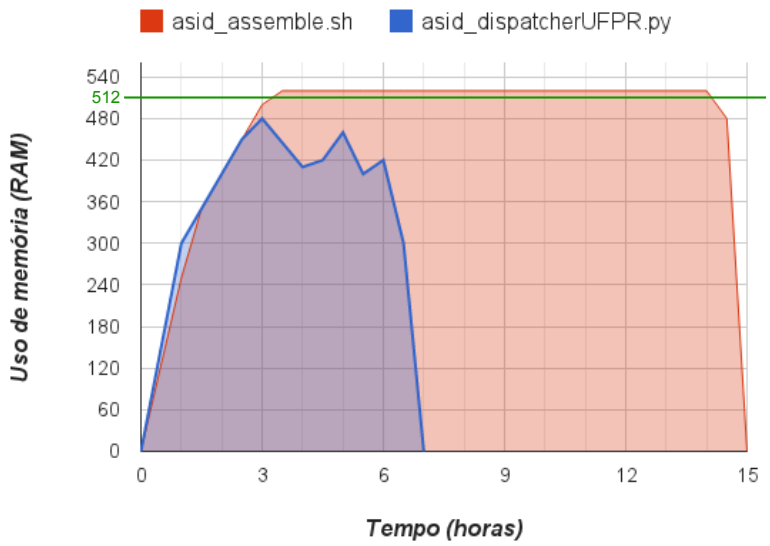

Figura 5.1: Comparativo do uso de memória entre o *ASiD* original e a solução implementada

Como pode ser visto no gráfico da Figura 5.1, o *asid_assembly.sh*, em vermelho, solicitou a totalidade da memória *RAM* disponível e consumiu todo o *swap*, mantendo, durante toda a execução, o seu uso máximo. O uso da multiprogramação com o consumo de toda memória e todo o *swap* ocasionam o fenômeno de *trashing*, estendendo o tempo necessário para terminar o processamento e obter o resultado. O *asid_dispatcherUFPR.py*, em azul, em nenhum momento extrapola a memória disponível e mantém uma margem de segurança para deixar memória para uso do sistema. Quando o uso de memória *RAM* chega perto do limite definido como mínimo necessário para manter o sistema em pleno funcionamento, o escalonador *asid_dispatcherUFPR.py* aguarda liberar memória para

instanciar novas lacunas. Por este motivo, podem ser observadas no gráfico curvas com o comportamento parecidos com dentes de serra. Quando é liberada memória suficiente, outra lacuna é lançada para a execução. A memória vai sendo liberada e o escalonador vai efetuando dinamicamente a verificação até que a quantidade livre seja suficiente para todo o processamento da próxima lacuna, de acordo com a previsão de uso de memória da política de escalonamento.

Utilizando esta abordagem de controle na submissão das tarefas pelo monitoramento da memória *RAM*, foi possível que durante toda a execução do *asid_ dispatcherUFPR.py* fossem respeitados os limites do sistema, resultando em uma redução de 55% no tempo de execução com o uso do escalonador, sendo avaliado para o mesmo conjunto de dados *Conjunto Treinamento1* utilizados na execução da implementação original do *ASiD*.

5.2.2 Comparação de concorrência no acesso ao disco

O uso do disco foi monitorado pelo resultado do comando *iostat*, que reporta estatísticas de entrada e saída do sistema. No modo de exibição extendida, com o uso do parâmetro *-x*, é possível visualizar o resultado do campo *%util*. Este campo indica a porcentagem da utilização e vazão do acesso ao disco. Quando o campo *%util* indica 100% de saturação no acesso e ocasiona atraso. Nos testes sem controlar o acesso ao disco, este valor estrapolava os 100%, chegando a picos de até 5000%. Neste caso a espera pelo acesso ao disco se tornou um gargalo que atrasou todo o sistema. Com o controle resultante do uso do escalonador, este índice se manteve baixo, contribuindo para a redução do tempo de execução.

O gráfico da Figura 5.2 compara os resultados do comando *iostat* do sistema operacional *GNU/Linux*, demonstrando o pico da porcentagem de utilização do disco durante a execução do *asid_ assembly.sh* e do *asid_ dispatcherUFPR.py*. Como pode ser observado o valor da variável *%util* do *asid_ assembly.sh*, em vermelho, atingiu valores de pico de até 5000%. Esta variável indica o quanto está ocupada a capacidade do disco para atender a

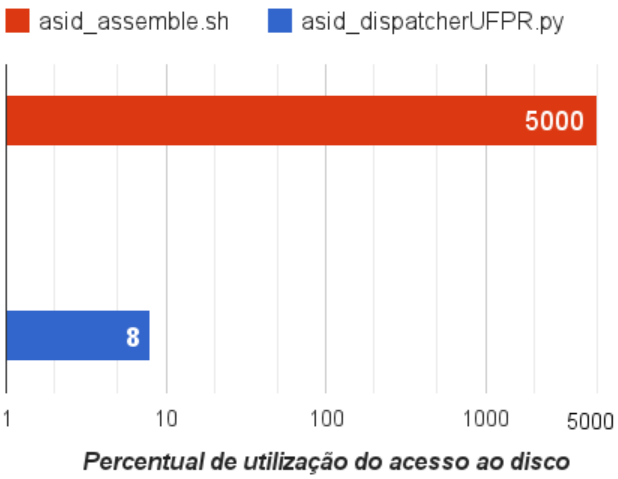

Figura 5.2: Comparativo entre o pico do *iostat* do *ASiD* original e da solução proposta

novas requisições, a taxa de ocupação da vazão do disco. Valores acima de 100% indicam que o acesso esta congestionado, o que resulta em espera no acesso ao disco e consequentemente uma maior demora para o término da atividade. Então as operações de leitura e escrita, sendo um gargalo e atrasando a evolução da execução do fechamento de lacunas. No caso do *asid_ dispatcherUFPR.py*, em azul, devido à implementação da verificação dos recursos computacionais antes de lançar novas lacunas, este valor se manteve abaixo de 10% durante todo o tempo da execução, não ocasionando nenhum atraso.

Foi observado que gerenciando o uso de memória, o problema de atraso por causa do gargalo no acesso ao disco foi eliminado e, portanto este critério de escalonamento não se faz mais necessário.

5.2.3 Comparação de uso de *cpu*

O gráfico da Figura 5.3 demonstra o uso de *CPU* durante a execução do *asid_ assembly.sh* e o Gráfico 5.4 demonstra do *asid_ dispatcherUFPR.py* para serem comparadas as duas abordagens. Em vermelho está representado o uso de *CPU* durante a execução do *asid_ assembly.sh*, que devido ao problema de espera por operações de entrada e de saída ocasionado pelo *trashing*, o uso do *CPU* permance baixo, abaixo de 1%. Em azul pode-se observar um melhor aproveitamento do tempo de *CPU* pelo

asid_ dispatcherUFPR.py. Por monitorar os recursos e não sobrecarregar o sistema, lançando novas tarefas apenas quando o recurso necessário para execução estiver disponível, foi possível evitar o problema do *thrashing*. Não sofrendo da degradação causada pelo *trashing* pode-se aproveitar melhor a capacidade de processamento e com isto, o tempo de execução diminui.

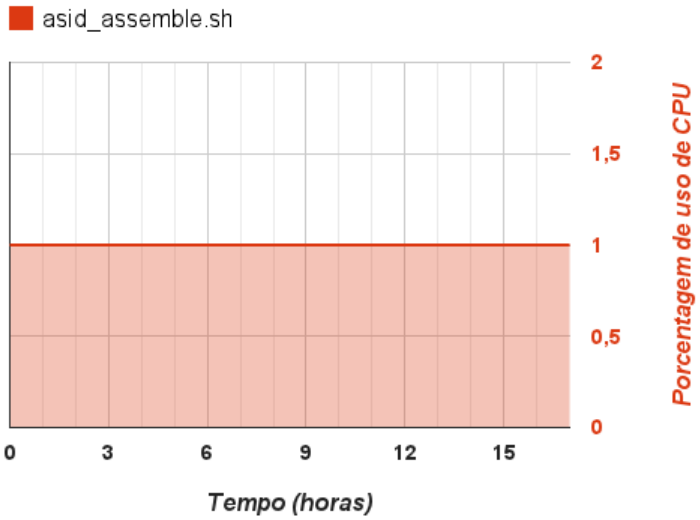

Figura 5.3: Demonstrativo do uso de *cpu* durante execução do *asid_ assembly.sh*

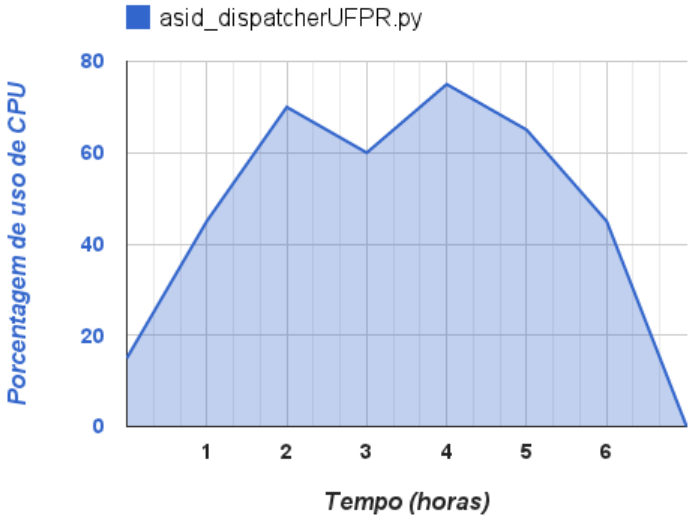

Figura 5.4: Demonstrativo do uso de *cpu* durante execução do *asid_ dispatcherUFPR.py*

5.2.4 Comparação dos tempos de execução

Como consequência do monitoramento dos recursos e gerenciamento das tarefas, o gráfico da Figura 5.5 ilustra a diferença dos tempos obtidos na execução das duas abordagens utilizando o mesmo conjunto de lacunas, resultantes do *Conjunto Treinamento1*. No eixo X é apresentado o tempo em horas e no eixo Y estão representados em azul o *asid_ assembly.sh* e em vermelho o *asid_ dispatcherUFPR.py*.

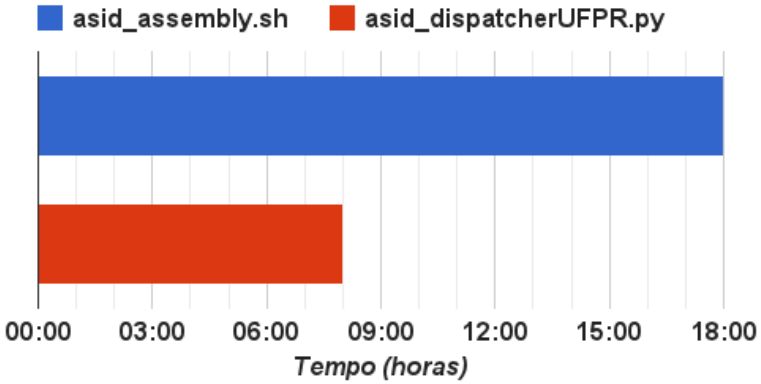

Figura 5.5: Comparativo entre o tempo de execução do *ASiD* original e da solução proposta

Comparando os resultados do *asid_ assembly.sh* original e do *asid_ dispatcherUFPR.py* foi obtida uma redução de 56% no tempo da execução do *ASiD*. O *asid_ dispatcherUFPR.py* conseguiu obter melhoria no desempenho, executando todas as lacunas em menos tempo que o *asid_ assembly.sh* para realizar o mesmo trabalho.

Após comparados os desempenhos pela análise dos gráficos, a Tabela 5.1 compara os problemas e soluções encontrados e enfrentados por cada uma das duas abordagens apresentadas. Os fatores de concorrência e espera no acesso ao disco foi eliminado com o uso do escalonamento das tarefas, enquanto no *asid_ assembly.sh* é um problema que ocasiona atraso e exaustão da memória *RAM* disponível que com o uso do escalonador não houve exaustão de memória, o que evitou a degradação de desempenho; o que não é verdade para a implementação padrão do *ASiD* que consome toda memória e sofre com a degradação do desempenho; a solução com o uso do escalonamento das tarefas possui

integridade do resultado, uma vez que executa exatamente o mesmo comando com os mesmos parâmetros para cada uma das lacunas encontradas.

Tabela 5.1: Comparativo da implementação original e da implementação com a solução proposta.

Problemas/Solução	*asid_ assembly.sh*	*asid_ dispatcherUFPR.py*
Concorrência no Acesso ao Disco	Sim	Não
Exaustão da Memória *RAM* Disponível	Sim	Não
Degradação de Desempenho	Sim	Não
Integridade do Resultado	Sim	Sim

Como mostra a Tabela 5.1, o *asid_ dispatcherUFPR.py* administra os problemas de concorrência no acesso a disco e da exaustão da memória, com isto não faz uso de *swap* e não entra na zona crítica de degradação de desempenho nem no problema de *trashing*. As entradas, os comandos e os parâmetros executados são exatamente os mesmos. O diferencial é o fato de observar o status do ambiente computacional utilizado antes de enviar mais lacunas para serem processadas.

Durante este trabalho foi identificado que a versão *multithread*, pelo fato de a quantidade de *threads* ser bastante superior ao número de núcleos de processamento. Em nenhum momento ou teste esta versão obteve resultados melhores do que a versão sem este paralelismo de processamento. No entanto, de forma individual para as lacunas grandes, a versão paralela sim é mais rápida do que a versão sem o paralelismo. O que não é verdade para as tarefas pequenas, que se processadas com o uso de paralelismo demoram mais do que a versão sequencial. Como a quantidade de tarefas a serem processadas são bastante elevadas, o tempo ganho com as poucas tarefas grandes era perdido com as muitas e bem mais comuns tarefas bastante pequenas. Em nenhum momento foi vantajoso o uso do *velvet* na sua versão *multithread* na fase do *ASiD*. O teste foi repetido na tentativa de utilizar o *velvet multithread* no *asid_ dispatcherUFPR.py*, mas nem assim se mostrou vantajoso, demorando 50% a mais do que o tempo de execução da versão sequencial em todos os testes.

Porém em testes individuais de cada tarefa, existia um ganho de tempo em tarefas grandes com o uso de paralelismo. Logo, diferenciar as versões paralela e sequencial de

acordo com o tamanho da tarefa a ser processada parecia uma solução interessante. Então foi avaliado o tamanho da tarefa, sendo que acima de 10 *megabytes* o *velvet* utilizado seria o paralelo, caso menor o *velvet* sequencial.

Desta forma foi possível conseguir mais um ganho de tempo na execução total do fechamento de lacunas. Os testes foram realizados com uma amostragem do *Conjunto Treinamento1*, denominada a partir de agora como *Conjunto Amostragem1*. O *Conjunto Amostragem1* é composto de 300 lacunas, compostas de representantes de todos os tamanhos de lacunas encontrados no *Conjunto Treinamento1*. A Figura 5.6 demonstra no eixo X o tempo em horas, a primeira barra em azul demonstra o tempo decorrido para a execução do *Conjunto Amostragem1* com a implementação original. A segunda barra, em vermelho, demonstra o tempo necessário para a execução utilizando o escalonador proposto até este momento. E seguiu o comportamento do *Conjunto Treinamento1* reduzindo o tempo de execução em 56%. Já terceira barra, em amarelo, demonstra o algoritmo proposto do escalonador juntamente com a diferenciação de uso do *velvet* sequencial e paralelo, de acordo com o tamanho das lacunas a serem processadas. Esta última abordagem conseguiu reduzir ainda mais o tempo de processamento das lacunas. Comparando com a proposta original, o tempo foi reduzido em 70%. Atingindo o objetivo de reduzir ainda mais o tempo do escalonador inicialmente proposto.

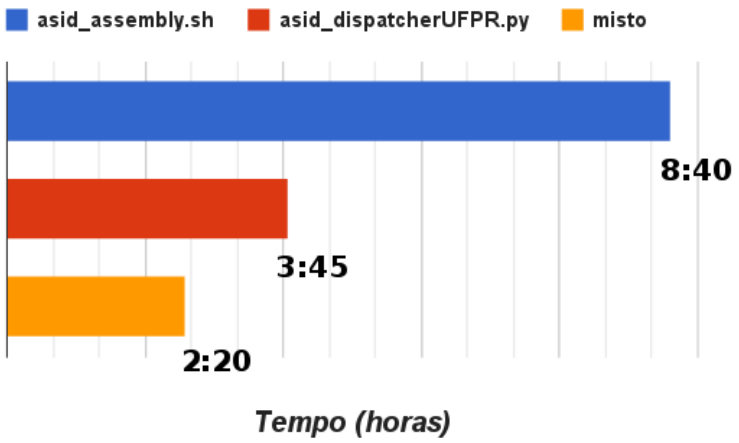

Figura 5.6: Tempos da implementação original X implementação do escalonador X Tempos da versão mista sequencial/paralela

Estes testes apontam que o paralelismo do *velvet* no caso onde muitas das lacunas a

serem processadas são pequenas, o que é o mais comum, não é eficiente. Não apenas pela elevada quantidade de tarefas sendo executadas em paralelo, mas também pela questão de que para processar pequenas tarefas, o paralelismo pode não ser interessante. Dada esta conclusão, a abordagem de diferenciação de uso das formas sequencial e paralela do *velvet* permite uma execução mais eficiente.

CAPÍTULO 6

CONCLUSÃO

Na fase de pós-montagem do *pipeline denovo2*, quando a etapa de fechamento de lacunas é executada pelo programa *ASiD*, são colocadas em execução uma grande quantidade de tarefas. Como estas tarefas são lançadas sequencialmente e sem nenhum critério até que todas sejam executadas, pode ocorrer solicitação de recursos de memória maior do que o ambiente computacional pode disponibilizar.

Manter todas as tarefas em execução faz com que a memória disponível se acabe, e como consequência o sistema entra em uma situação de degradação total do desempenho, causando atraso no processamento das tarefas. Esta situação poderia ser evitada se antes de lançar novas tarefas, fossem verificados os recursos computacionais disponíveis, esta é a função de um escalonador de tarefas. Para tal, foi proposta a implementação de um escalonador de tarefas que utiliza políticas de escalonamento baseadas no monitoramento dos recursos computacionais para decidir quando lançar novas lacunas para serem processadas. Com o escalonador proposto, ao invés de processar todas as tarefas ao mesmo tempo e sem nenhum critério como era feito originalmente, as tarefas são disparadas observando a disponibilidade dos recursos necessários para a sua execução. Dessa forma, diversas tarefas podem ser executas em paralelo consumindo os recursos de forma controlada, não causando o problema de degradação de desempenho e resultando na redução do tempo necessário para o processamento das tarefas.

Com o uso do escalonador foi possível controlar as tarefas para que utilizassem os recursos de forma eficiente e fosse possível executar diversas tarefas em paralelo. Com esta abordagem, o tempo de execução do conjunto de dados *Conjunto Treinamento1* utilizado nos experiementos foi reduzido em 56% em relação ao tempo obtido na implementação original. Este tempo foi posteriormente melhorado para uma redução de 70%, que foi

atingida diferenciando o uso das versões sequencial e paralela do *velvet*. Uma vez que o conjunto de dados a ser processado é constituído principalmente de tarefas pequenas, e o uso da versão paralela acaba não sendo eficiente como o esperado para tarefas pequenas pois, o uso de paralelismo pode causar mais demora devido ao gerenciamento de abertura e fechamento das *threads* do que executar na forma sequencial. Por este motivo a versão paralela não se mostrava tão eficiente quanto a sequencial, no entanto para as tarefas maiores, a forma mais eficiente é utilizar o parelelismo, então uma abordagem mista consegue aproveitar a implementação mais eficiente de acordo com a tarefa a ser processada.

Entre as dificuldades encontradas durante este estudo, destacam-se a pouca documentação do *denovo2* e de cada um dos programas que o compõem. O acesso aos dados genônicos para os primeiros testes com o *pipeline* para compreender o problema; o espaço em disco para armazenar os dados de entrada inicialmente e posteriormente, para armazenar os resultados; o acesso a computadores com memória suficiente para executar os programas.

O fato de ser um problema de *Big Data* dificultou a gestão dos dados, e a simples ação de realizar a cópia de um conjunto de dados para comparar resultados demorava horas para ser realizada. Por muitas vezes todos os resultados precisavam ser eliminados, pois o volume de dados gerado a cada execução era bastante elevado.

Como sugestão de trabalho futuro, um ponto é melhorar a qualidade dos resultados e reduzir o tempo da montagem, ou seja, uma nova implementação do *rsampling*. O *rsampling*, na fase de pré-montagem, faz um corte aleatório dos dados. Este corte poderia utilizar técnicas de Inteligência Artificial, ou mesmo comparação da qualidade dos dados selecionados, evitando o uso de dados com baixa qualidade, desta forma melhorando o resultado e reduzindo o tempo da montagem por trabalhar com dados melhores.

Como durante a execução do *pipeline denovo2* é grande o volume de dados movimentado entre memória e o disco. Por exemplo, somente na fase do *ASiD*, um conjunto de dados de 4756 lacunas que ocupa aproximadamente 8 *GB* de espaço em disco, movimenta mais de 1 *TB* de dados, volume muitas vezes maior para o *pipeline* inteiro. Movimentação

de leitura e escrita de dados acontecem a cada uma das etapas do *pipeline*. Cada fase lê os dados do disco, realiza o respectivo trabalho nestes dados e escreve o resultado em disco, para em seguida ser lido pela outra fase. Este volume de movimentação de dados entre memória e disco é outro ponto que reduz drasticamente o desempenho da execução do *denovo2*. Uma abordagem de execução em memória ou em discos *Solid-state Disk - (SSD)* poderiam apresentar bons resultados, e são também propostas de trabalhos futuros.

BIBLIOGRAFIA

[1] FLEISHMANN, R. D.; ADAMS, M. D. e WHITE, O. Whole-genome random sequencing and assembly of haemophilus influenzae rd. *Science (New York, N.Y.)*, 269(5223):496–498, julho de 1995.

[2] AIDA, K. Effect of job size characteristics on job scheduling performance. *Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2000.

[3] AZMIL, Z. R. M.; BAKAR, K. A. e ABDULLAH, A. H. Performance comparison of priority rule scheduling algorithms using different inter arrival time jobs in grid environment. *International Journal of Grid and Distributed Computing (IJGDC)*, 2011.

[4] DURFEE, T.; NELSON, R.; BALDWIN, S. e PLUNKETT, G. The complete genome sequence of Escherichia coli DH10B: insights into the biology of a laboratory workhorse. *Journal of bacteriology (JB)*, 1997.

[5] BATAT, A. e DROR, F. Gang scheduling with memory considerations. *14th International Parallel and Distributed Processing Symposium*, páginas 109–114, 1999.

[6] BAYAT, A. Science, medicine, and the future Bioinformatics. *BMJ*, 2002.

[7] Applied Biosystems. See the Difference - Discover the Quality Genome. "Disponível em: `http://www.wzw.tum.de/physio/fileadmin/1.Seite/GemSemTierwissenschaft/Rygus/SOLiD-4.pdf`. Acessado em Abr./2011", 2004.

[8] Applied Biosystems. Color space analysis in the solid system: the theory, advantages and solutions. "Disponível em: `https://www.ncbi.nlm.nih.gov`. Acessado em Mar./2011", 2009.

[9] Applied Biosystems. SOLiD de novo accessory tools 2.0 1. "Disponível em: `http://gsaf.cssb.utexas.edu/wiki/images/7/71/DeNovo_Assembly_Pipeline_2.0.pdf`. Acessado em Fev./2012", 2010.

[10] DOHM, J. C.; LOTTAZ, C.; BORODINA, T. e HIMMELBAUER, H. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Res.*, 2007.

[11] CHAISSON, M. J.; BRINZA, D. e PEVZNER, P. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome research*, 2009.

[12] BROWN C. BIGDATA: Small: DA: DCM: Low-memory Streaming Prefilters for Biological Sequencing Data. *DCM*, 2012.

[13] CASAVANT, T. L. e KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 1988.

[14] ZHANG, W.; CHEN, J. e YANG, Y. A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. *PloS one*, 2011.

[15] ZERBINO D. R. Velvet Manual - version 1.1. "Disponível em: `http://helix.nih.gov/Applications/velvet_manual.pdf`. Acessado em Dez./2011", 2008.

[16] ZERBINO D. R. Genome assembly and comparison using de Bruijn graphs. *Hinxton, Cambridge, United Kingdom*, 2009.

[17] DENNING, P. Thrashing: its causes and prevention. *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), New York, NY, USA, 1968. ACM.

[18] DENNING, P. Thrashing. *Wiley Encyclopedia of Computer Science and Engineering*. Wiley Encyclopedia of Computer Science and Engineering, 2008.

[19] RIEDEL, M.; WOLF, F.; DIETER, K. Research advances by using interoperable e-science infrastructures. *Cluster Computing*, 2009.

[20] MARDIS E. The impact of next-generation sequencing technology on genetics. *Trends in genetics : TIG*, 2008.

[21] MARGULIES, M.; et. al. Genome sequencing in microfabricated high-density picolitre reactors. "Disponível em: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1464427\&tool=pmcentrez\&rendertype=abstract`. Acessado em Mar./2012", 2005.

[22] FLYNN, M. J. e RUDD, K. W. Parallel architectures. *ACM Computing Surveys*, 1996.

[23] GALPERIN, M. e KOONIN, E. . From complete genome sequence to complete understanding? *Trends Biotechnol*, 2011.

[24] ROOSTA S. H. Parallel processing and parallel algorithms: Theory and computation. *Springer-Verlag*, 1999.

[25] WHITEFORD, N.; HASLAM, N. e WEBER, G. An analysis of the feasibility of short read sequencing. "Disponível em: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1278949\&tool=pmcentrez\&rendertype=abstract`. Acessado em Jan./2012", 2005.

[26] IDURY, R. M. e WATERMEN, M. S. A new algorithm for DNA sequence assembly. *Journal of computational biology : a journal of computational molecular cell biology*, 1995.

[27] Illumina. An Introduction to Next-Generation Sequencing Technology. "Disponível em: `http://res.illumina.com/documents/products/illumina_sequencing_introduction.pdf`. Acessado em Mai./2012", 2010.

[28] Illumina. Illumina - sequencing technology. "Disponível em: `http://www.illumina.com/technology/sequencing_technology.ilmn`. Acessado em Jun./2012", 2012.

[29] MEIDANIS J. A simple toolkit for dna fragment assembly. *American Mathematical Society*, 1999.

[30] MCKERNAN, K.; BLANCHARD, A.; KOTLER, L. e COSTA, G. Reagents, methods and libraries for bead-based sequencing. *European Patent Application Publication*, 2010.

[31] KUNZ, T. The influence of different workload descriptions on a heuristic load balancing scheme. *Software Engineering, IEEE Transactions on*, 1991.

[32] EL-REWINI, H.; LEWIS, T. G. e ALI, H. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[33] RIEDEL, M.; STREIT, A.; WOLF, F.; LIPPERT, T. e KRANZLMÜLLER, D. Classification of Different Approaches for e-Science Applications in Next Generation Computing Infrastructures. *IEEE 4th International Conference on eScience*, 2008.

[34] PETTERSSON, E.; LUNDEBERG, J. e AHMADIAN, A. Generations of sequencing technologies. *ncbi*, 2009.

[35] HUDSON M. Sequencing breakthroughs for genomic ecology and evolutionary biology. "Disponível em: `http://www.ncbi.nlm.nih.gov/pubmed/21585713`. Acessado em Mar./2012", 2008.

[36] METZKER M. L. Sequencing technologies, the next generation. *Nature Reviews Genetics*, 2009.

[37] JECK, W. R.; MARDIS, E. R. et al. Extending assembly of short DNA sequences to handle error. *Bioinformatics (Oxford, England)*, 2007.

[38] ZERBINO, D. R.; MCEWEN, G. K.; MARGULIES, E. H. e BIRNEY, E. Pebble and rock band: heuristic resolution of repeats and scaffolding in the velvet short-read de novo assembler. *PloS one*, 2009.

[39] YUTAO, F.; MCKERNAN, K. J. et al. SOLiD Sequencing and 2-Base Encoding Single Base Insertions / Deletions. *applied biosystems*, 2007.

[40] MILLER, J. R, KOREN, S., e SUTTON, G. Assembly algorithms for next-generation sequencing data. *Genomics*, 2010.

[41] PANDEY, V.; NUTTER, R. C. e PREDIGER, E. *Next-generation genome sequencing: Twoards personalized medicine.* Wiley-VCH verlag GmbH Co, 2008.

[42] CHAISSON, M.; PEVZNER, P. e TANG, H. Fragment assembly with short reads. *Bioinformatics (Oxford, England)*, 2004.

[43] COMPEAU, P.; PEVZNER, P. e TESLER, G. How to apply de Bruijn graphs to genome assembly. *Nature biotechnology*, 2011.

[44] SUNDQUIST, A.; RONAGHI, M.; TANG, H.; PEVZNER, P. e BATZOGLOU, S. Whole-genome sequencing and assembly with high-throughput, short-read technologies. *PloS one*, 2007.

[45] ROZAS, J.; SANCHEZ-DELBARRIO, J. C. e MESSEGUER, X. DnaSP, DNA polymorphism analyses by the coalescent and other methods. *Bioinformatics*, 2003.

[46] SANGER, F. e NICKLEN, S. DNA sequencing with chain-terminating. *Sanger*, 1977.

[47] DE BONA, F.; OSSOWSKI, S.; SCHNEEBERGER, K. e RÄTSCH, G. Optimal spliced alignments of short sequence reads. *Bioinformatics (Oxford, England)*, 2008.

[48] Roche Applied Science. Roche applied science. "Disponível em: `https://www.roche-applied-science.com`. Acessado em Jun./2012", 1996-2012.

[49] 454 Life Science Sequencer. 454 life science sequencer. "Disponível em: `http://www.454.com/`. Acessado em Jun./2012", 1996-2012.

[50] MEIDANIS, J. SETUBAL, J. C. Uma introdução a biologia computacional. *IX Escola de Computação - Recife*, 1994.

[51] MEIDANIS, J. SETUBAL, J. C. *Introduction to Computacional Molecular Biology.* PWS Publishing Company, 1997.

[52] SGI. Technical Advances in the SGI UV Architecture . "Disponível em: `http://www.sgi.com/pdfs/4192.pdf`. Acessado em Jul./2011", 2012.

[53] SOLiD. *preprocessor Documentation*, 2008.

[54] SETIA, S.; SQUILLANTE, M. e NAIK, V. The impact of job memory requirements on gang-scheduling performance. *SIGMETRICS Perform. Eval. Rev.*, 1999.

[55] WARREN, R. L.; SUTTON, G. G. e JONES, S. Assembling millions of short DNA sequences using SSAKE. "Disponível em: `http://www.ncbi.nlm.nih.gov/pubmed/17158514`. Acessado em Jan./2012", 2007.

[56] TANENBAUM, A. S. e ZUCCHI, W. L. *Organização estruturada de computadores*. Pearson Prentice Hall, 2009.

[57] PEVZNER, P. ; TANG, H e WATERMAN, M. S. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences of the United States of America*, 2001.

[58] Life Technologies. De Novo Error Correction for SOLiD data SAET v.2.2. "Disponível em: `http://www.biostars.org/static/downloads/solid/solid-denovo-assembly/saet.2.2/SAET.v2.2.pdf`. Acessado em Mar./2011", 2009.

[59] Life Technologies. Applied biosystems. "Disponível em: `http://www.appliedbiosystems.com`. Acessado em Jun./2012", 2011.

[60] Life Technologies. Life technologies. "Disponível em: `http://www.lifetechnologies.com/`. Acessado em Jun./2012", 2012.

[61] Life Technologies. Project denovo. "Disponível em: `http://solidsoftwaretools.com/gf/project/denovo`. Acessado em Jun./2012", 2012.

[62] SHENDURE, J.; WANG, M. D.; et al. Accurate multiplex polony sequencing of an evolved bacterial genome. *Science (New York, N.Y.)*, 2005.

[63] BENSON, D.; KARSCH-MIZRACHI, I.; LIPMAN, D. J.; OSTELL, J.; WHELLER, D. L. GenBank. *Nucleic acids research*, 2008.

[64] WU, M. e SUN, X. Memory conscious task partition and scheduling in grid environments. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, Washington, DC, USA, 2004. IEEE Computer Society.

[65] ZERBINO, D. R. e BIRNEY, E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 2008.

APÊNDICE A

APÊNDICE - FLUXO DE EXECUÇÃO DO *PIPELINE DENOVO2*

O objetivo deste apêndice é complementar o Capítulo 2 auxiliando na compreenção de cada fase do *pipeline denovo2*, iniciando do *download* do *pipeline denovo2* ao processo de instalação, configuração de variáveis de ambiente e a execução.

Durante a evolução do processo de *download*, instalação e uso do *pipeline denovo2*, foram encontradas algumas dificuldades comuns a outros usuários que estão aqui respondidas. Muitas das dúvidas comuns e dificuldades no uso do *pipeline denovo2* são pelo fato de a documentação tanto de uso quando de desenvolvimento do *pipeline denovo2* serem praticamente inexistentes. O que existem são alguns poucos artigos como os do *Zerbino*, idealizador do *velvet* [38, 15, 65] e uma pequena documentação simples [9] desenvolvida pela [59], além de documentos específicos sobre programas que integram o *pipeline*, como o [58] do *SAET*, por exemplo. Fora estes, o que existem são apenas alguns artigos correlacionados, comparando as tecnologias de *NGS* como [11, 14, 27, 36, 40], ou comparando estruturas e técnicas utilizadas pelos sequenciadores, por exemplo, o *DBG* usado pelo *velvet* [43, 16].

Este Apêndice A busca diminuir a escassez de fontes para a compreensão deste trabalho. Para isto, será apresentado um acompanhamento da evolução da execução do *pipeline denovo2*, e a cada etapa do *pipeline* serão apontados os arquivos criados e trabalhos realizadas até a última fase com o arquivo de saída com os *contigs* e *scafolfds* do genoma alvo, bem como a análise do que foi gerado.

A.1 Executando o *pipeline denovo2*

O *pipeline denovo2* era disponibilizado no site da *SOLiD Software Tools*, onde todos os artigos o citam [61]. O site que continha arquivo *denovo2.tgz* não esta mais disponível desde meados de 2012, podendo atualmente ser encontrado no site do autor deste trabalho[1]. Para descompactar o arquivo *denovo2.tgz* baixado, pode ser utilizado o *gzip* com a opção *-d*, executando o comando: *gzip -d denovo2.tgz*. Após descompactado, é criada a pasta *denovo2*, contendo os arquivos que juntos compõem o *pipeline denovo2*. Estes arquivos e suas estruturas são parcialmente mostrados nas Tabelas **??** e A.2, onde os itens em negrito indicam as pastas que são expandidas a direita.

Tabela A.1: Estrutura parcial de arquivos do *pipeline denovo2*

asid.1.0	asid_assembly_one.sh asid_assembly.sh asid_light Makefile src	asid_light.cpp util.cxx util.h util.o zcompress.cxx zcompress.h zutil.cxx zutil.h zutil.o
assemble.pl		
Makefile		
saet.2.2	saet_mp	
utils	assembly_stats.pl cumlength rsampling Makefile reverse_and_concatenate_de_genome.pl solid_denovo_preprocessor_v1.2.pl **src**	cumlength.cpp prefilter.cpp rsampling.cpp

[1] www.inf.ufpr.br/jbdorr

Tabela A.2: Estrutura parcial de arquivos do *pipeline denovo2* - pasta *velvet*

velvet_X.X.XX	Makefile update_velvet.sh velvetg velvetg_de velvet velveth_de **src**	readSet.c allocArray.h graph.c preGraph.c autoOpen.c splay.c recycleBin.c graphReConstruction.c binarySequences.c graphReConstruction.h roadMap.c binarySequences.h graphStats.c concatenatedGraph.c run2.c kmer.c run.c concatenatedPreGraph.c kmerOccurenceTable.c scaffold.c correctedGraph.c kseq.h shortReadPairs.c dfib.c splayTable.h fib.c fibHeap.h fibpriv.h dfibHeap.h dfibpriv.h preGraphConstruction.c splayTable.c tightString.h fibHeap.c preGraph.h tightString.c utility.c passageMarker.h readCoherentGraph.c locallyCorrectedGraph.h dfibHeap.c passageMarker.c

Como pode ser observado nas Tabelas **??** e A.2, o *denovo2* possui diversos algoritmos e bibliotecas. Para o desenvolvimento deste conjunto de algorítmos foram utilizadas diferentes linguagens de programação como *Perl, Shell Script, C, C++ e Java.* O *velvet* sozinho já é uma coleção bastante grande de diferentes algoritmos e com uma grande quantidade de linhas de código, o que o torna complexo.

A versão do *velvet* que acompanha o *pipeline* é antiga, no entanto, dentro desta pasta existe um *script* em *shell*, de nome *$update_ velvet.sh*, que sendo executado faz a atualização para a versão atual utilizando o repositório *git* do projeto.

Um das possíveis formatos de entradas para executar o *denovo2* é o formato de *tags* duplas - *mate-pair* em *color space*, com a terminação *.csfasta*, sendo composto pelos pares *F* e *R* exemplificados respectivamente na Tabela A.3 e na Tabela A.4, e os seus respectivos arquivos de qualidade terminados em *.qual*, este representado e exemplificado na Tabela A.5. Nos arquivos *color space* da Tabela A.4 os pontos indicam ausência de cor.

Tabela A.3: Exemplo do arquivo de entrada *f3.csfasta*

R3.csfasta
>469_29_17_F3
T2033031030123133032323113101332112233313212131032 0
>469_29_1434_F3
T132113.212313112122223110213122111211222 0..2123221

Tabela A.4: Exemplo do arquivo de entrada *r3.csfasta*

R3.csfasta
>469_29_17_R3
G203231123031.21203101312013030001223312331120032 0
>469_29_1434_R3
G100233132323220.00232110010212211121122 20..2123221

Tabela A.5: Exemplo do arquivo de entrada *f3.qual*.

F3.qual
>469_29_17_F3
30 31 24 22 25 17 20 21 17 29 22 30 15 2 31 15 21 4 3 28 10 24 26 18 22
17 24 4 8 12 10 14 5 21 15 5 23 12 13 7 6 15 14 17 6 18 21 12 11 13
>469_29_1434_F3
31 24 29 31 30 27 2 31 30 27 31 27 22 30 28 29 32 21 31 31 23 22 31 30 23
31 16 17 22 13 8 21 31 17 7 31 8 29 23 13 8 22 2 1 14 8 27 20 10 17

O *script assemble.pl* é responsável por executar todo o *pipeline*. Por padrão, este *pipeline* já contém parâmetros pré-definidos, mas, caso sejam passados novos, estes irão prevalecer. Juntamente com os arquivos do *denovo2*, existe uma pasta chamada *sample*. Nesta pasta estão os arquivos *F3.csfasta, R3.csfasta, F3.qual* e *R3.qual* que podem ser utilizados para verificar se esta tudo funcionando para executar o *pipeline*.

Antes de cada execução, é necessário exportar uma variável de ambiente de nome *denovo2*. Esta variável deve conter o caminho de onde está localizado o *denovo2* no disco e, pode ser feito executando o comando *$denovo2=‘pwd‘* e em seguida, exportando a variável com o comando *$export denovo2* na linha de comando.

```
$denovo2=‘pwd‘
$export denovo2
```

Para executar este teste simples pode ser executa a linha de comando chamando o *script assemble.pl* e informando os arquivos de entrada e os parâmetros necessários, no seguinte formato: *script assemble.pl*, arquivo *F3.csfasta*, arquivo *F3.qual*, tamanho de referência, diretório de saída definido pelo parâmetro *-outdir*, o parâmetro *-r3* informa que o próximo é o arquivo *R3.csfasta*, o parâmetro *-r3qv* indica que o próximo arquivo é o da qualidade *R3.qual*, o parâmetro *-ins_length* refere-se à distância entre os pares, depende de cada técnica de sequenciamento e é informado pelo sequênciador na saída dos dados, e por fim, o parâmetro *-ins_length_sd* indica a variação do tamanho estimado. Por exemplo:

```
./assemble.pl sample/f3.csfasta sample/f3.qual 20000 -outdir assembly2 -r3
sample/r3.csfasta -r3qv sample/r3.qual -ins_length 3500 -ins_length_sd 700
```

No caso da verificação inicial bem sucedida é então possível realizar outras rodadas.

Em seguida, serão apresentados individualmente a função e o trabalho realizado por cada um dos programas integrantes do *pieline denovo2*. A sequência das fases é demonstrada da Figura A.1 adaptada de [9].

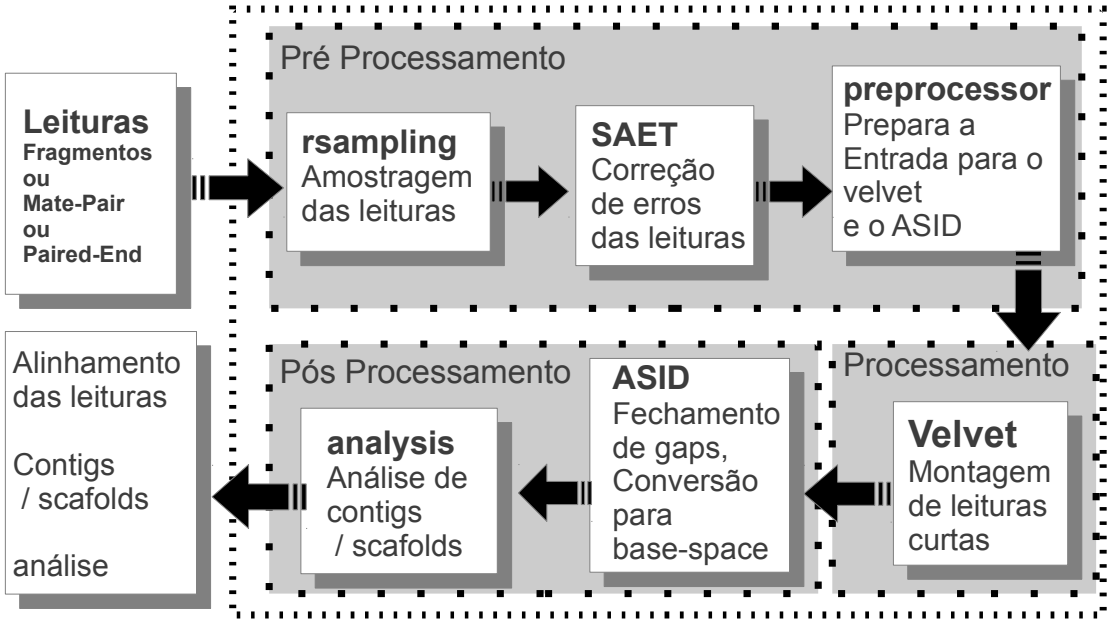

Figura A.1: Diagrama do *pipeline denovo2*

A.2 Execução do pré-processamento da montagem

A fase de pré-montagem realiza a preparação dos dados para a montagem. Na Subseção A.2.1 é apresentado o *rsampling* que seleciona uma amostragem do conjunto de dados que será processado. Na Subseção A.2.2 o *SAET* realiza correção nos dados obtidos do *SOLiD4*. Como última etapa da fase de pré-processamento o *preprocessor* apresentado na Subseção A.2.3 realiza a conversão dos dados para o formato utilizado pelo montador *velvet*.

A.2.1 Pré-processamento da montagem - *rsampling*

O *rsampling* é um programa escrito em *C++* e distribuído sob a licença *copyright*. Foi desenvolvido em 2010 pelo cientista bioinformata *Dumitru Brinza* da *Life Technologies*. A sintaxe de execução do *rsampling* para *mate pair* é a seguinte:

```
./rsampling f3.csfasta f3.qual refLength -r3 r3.csfasta -r3qv r3.qual [-options]
```

Os arquivos utilizados como entrada para este programa, são as leituras *mate pair* no formato *color space - csfasta*. Esta entrada é constituída pelos pares *F3* e *R3* e seus respectivos arquivos de qualidade.

Como saída, o *rsampling* gera os arquivos *subreads.csfasta* e *subreads2.csfasta*. Estes arquivos, são obtidos a partir de uma amostragem selecionada aleatoriamente dos arquivos *csfasta* de entrada. Para acompanhar os arquivos com uma amostra das leituras de entrada, são gerados também, os seus respectivos arquivos de qualidade, *subreads.qual* e *subreads2.qual*.

A.2.2 Pré-processamento da montagem - *SAET*

O *SOLiD Accuracy Enhancement Tool - SAET* [58] é uma ferramenta para correção de erros sob a licença *copyrigth*, de propriedade da *Life Technologics*, e não possui código fonte aberto. Foi desenvolvida em 2010 e atualmente está na versão *2.2*.

O tempo de execução e o fator de correção de erro realizados pelo *SAET*, dependem do tamanho da entrada e do número de voltas locais (*-localrounds*) e globais (*-globalrounds*) executadas pelo programa. O primeiro parâmetro permite corrigir os erros encontrados nas rodadas locais nas leituras usando o espectro pré-computado. O segundo parâmetro recalcula o espectro após cada rodada global e permite corrigir erros na leitura com a rodada local baseando-se nos novos valores. Por padrão são 2 rodadas globais e o tamanho do genoma dividido por 8 rodadas locais. Com o valor padrão de rodadas global/local a vazão de dados esperada é de 1 *Gb* por hora e a quantidade de *RAM* usada não ultrapassa 2 *GB*, segundo a documentação em [58]. Porém, nos testes executados este valor foi bastante ultrapassado por cada uma das *threads* em execução.

O *SAET* trabalha com *multithreads* e realiza muito acesso a disco, a execução é demorada e bastante representativa dentro do tempo total de execução do *pipeline* como um todo. Sua sintaxe de execução é

```
./saet_mp <reads.csfasta> <reads.qual> <refLength> [-options]
```

O arquivo de entrada é no formato *color space*. O título de cada sequência, assim como as duas primeiras bases são ignoradas. Como saída, o *SAET* escreve um diretório denominado *fixed*, e nele, os arquivos *reads.csfasta* e o respectivo arquivo de qualidade *reads.qual*. Alguns dos seus parâmetros são adicionados automaticamente com valores padrão pelo *pipeline denovo2* durante a execução. São eles:

- *-qvhigh qv*: este parâmetro evita correção das bases com qualidade acima de *qv*, onde *qv* tem como valor padrão de 25. Com isto bases com a qualidade maior que 25 não serão corrigidas, diminuindo o tempo de execução do *SAET*;
- *-trimqv tq*: corta o final do arquivo de leituras até encontrar uma posição com qualidade acima de *tq*, onde *tq* tem como valor padrão de 25. Caso o tamanho da leitura restante seja menor do que o valor passado em *seed size* + 2 , a leitura é descartada.
- *-maxtrim mt*: corta o final do arquivo de leituras até encontrar uma base com qualidade acima de *mt*, onde *mt* tem como valor padrão de 25. Caso o tamanho da leitura restante seja menor do que o valor passado em *seed size* + 2 , a leitura é descartada.

A.2.3 Pré-processamento da montagem - *preprocessor*

O *preprocessor* [53], atualmente na versão *1.2*, foi escrito por *Vrunda Sheth* e *Craig Cummings*, recebe como entrada os arquivos *F3_ reads.csfasta* e *R3_ reads.csfasta* (*subreads.csfasta* e *subreads2.csfasta*), e a partir deles faz a conversão dos dados do formato *color space* com a terminologia *.csfasta* representado pelos números *0123*, para o formato de pseudobases *ACTG*, chamado *double encoded* com a terminação *.de*.

A saída do *preprocessor* pode ser encontrada na pasta *./preprocessor*, são dois grandes arquivos, *colorspace_ input.csfasta* que é a concatenação dos arquivos *subreads.csfasta* e *subreads2.csfasta* e, em seguida sofre a conversão de *color space* para *double encoded*,

transformando os números que representavam as cores em letras que representam pseudo-bases, gerando então o arquivo *doubleEncoded_ input.de* que será posteriormente utilizado pelo *velvet*.

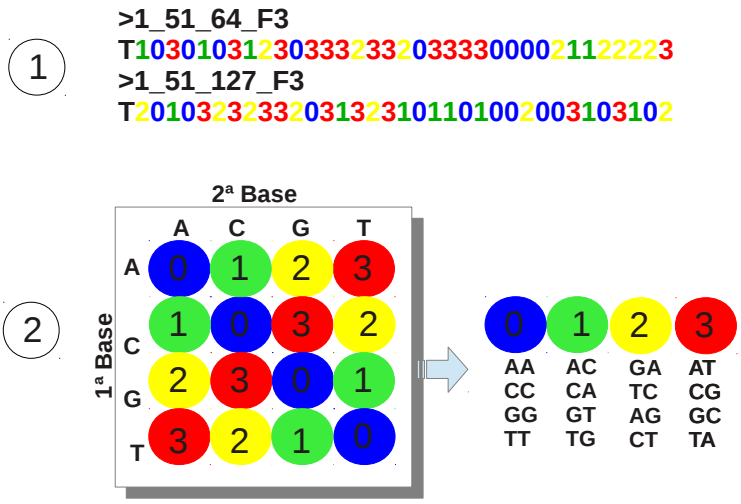

Figura A.2: Conversão do formato *double encoded* para *color space*

Durante a execução, são eliminadas leituras órfãos, sem par, invertidas as leituras do arquivo *F3* para a orientação utilizada pelo *velvet*, convertido do formato *SOLiD color space fasta - csfasta* [8] para o formato *double encoded* e, por fim, removida a primeira base de cada linha [53]. Na Figura A.2, em *(1)*, um exemplo de um arquivo *csfasta*. No arquivo no formato *csfasta*, os valores representam as transições entre os nucleotídeos 0, 1, 2, 3 (também referido como cores azul = 0, verde = 1, amarelo = 2 e vermelho = 3). O método para a descodificação é como o demonstrado em **(2)** na Figura A.2.

Por exemplo, uma sequência de *DNA* com 5 bases, *AACTA* em *double encoded*, é representada pelo valor numérico 0123, em *color space*, representando as respectivas cores. A codificação utilizada foi *AA=0, AC=1, CT=2* e *TA=3*, conforme pode ser verificado na Figura A.2.

Na Figura A.3, é demonstrado o funcionamento do *preprocessor*, em **(a)**, os arquivos *F3* e *R3* do *mate pair* no formato *csfasta*. Em **(b)** os arquivos de entrada foram concatenados de forma intercalada, gerando o arquivo *colorspace_ input.csfasta*. Em **(c)** o arquivo gerado é o *doubleEncoded_ input.de*, originado a partir do arquivo anterior após ser

convertido de *csfasta* para *double encoded* e removendo a primeira base de cada uma das linhas. As linhas referentes a *tag F3* são invertidas para respeitar a orientação utilizada pelo *velvet*.

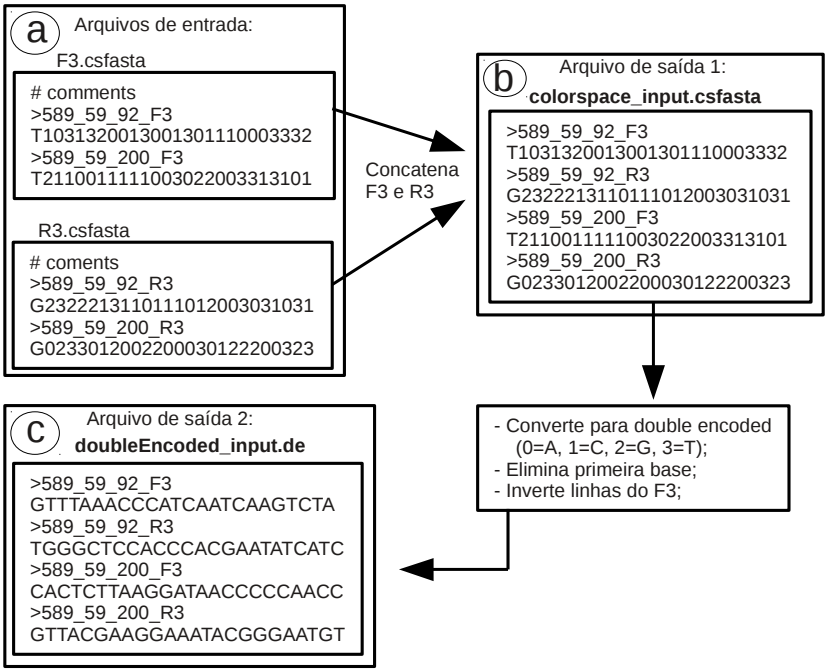

Figura A.3: Conversão realizada pelo *preprocessor*

A.3 Execução do processamento da montagem do genoma

A fase de processamento da montagem é composta apenas pelo *velvet*. Ele recebe uma amostragem dos dados, com correção de erros e convertida para o formato em que trabalha, para então realizar a montagem, principal atividade deste *pipeline*. Posteriormente, os dados montados vão para uma fase de melhoria (fechamento de lacunas) e análise de resultados.

A.3.1 Montagem do genoma - *velvet*

O montador *velvet* [65], é um conjunto de vários algoritmos com a função de realizar montagens de genoma com leituras curtas [38, 15, 16]. Foi desenvolvido por *Daniel Zerbino* e *Ewan Birney* do *European Bioinformatics Institute - EMBL-EBI, Cambridge, Inglaterra*

[65]. O *velvet* foi inicialmente desenvolvido para o *Illumina* [27, 28], e posteriormente foi adaptado para a plataforma *SOLiD*.

O *velvet* é dividido em dois módulos, o *velveth* e o *velvetg*. O *velveth* gera os *k-mers*, pequenos pedaços ímpares, a partir das leituras de entrada do programa. Os *k-mers* são gravados apenas uma vez, a quantidade de vezes que eles aparecem e os respectivos caminhos para outros *k-mers* são apenas referenciados no grafo e não explícitos [65]. Com isto são gerados os arquivos *Roadmap* e *Sequences* para o *velvetg*.

O *velvetg* utiliza estes dados para montar os grafos que são trabalhados para realizar uma montagem. A principal característica deste montador de leituras curtas, que o distingue de montadores tradicionais é o uso do grafo de *Bruijn - DBG* [57] para o processo de comparação e montagem das leituras. O grafo de *DBG* suporta bem a repetição de dados, comum nos genômas, como demonstrado na Figura A.4 adaptada de [57], que exemplifica o armazenamento destas repetições. Antes de montar o *DBG*, são criados pré-grafos como estratégia mais eficiente em relação ao uso de mémoria [16].

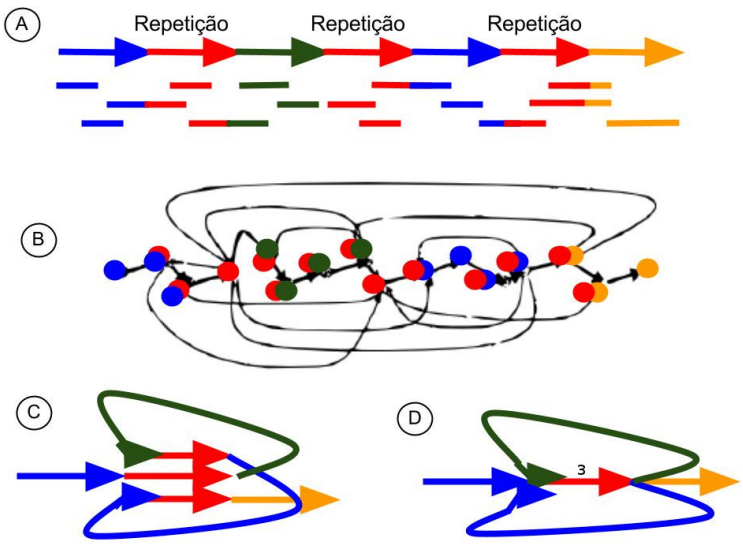

Figura A.4: Repetições no grafo de *bruijn*

O *velvet* possui mecanismos para remoção de erros ou repetições que prejudiquem o percurso no grafo [38], o *DBG* é construído com os dados da sequência não corrigidos, posteriormente, corrige os erros [15, 16, 65]. A complexidade do gráfo para grandes organismos e com cobertura elevada é problemática. O módulo *velvetg* cria e manipula o

DBG [38], para isto percorre o gráfo de diferentes maneiras, de acordo com os parâmetros informados pelo usuário [65]. A edição do grafo foi proposta inicialmente por *Pevzner* [57] e tem como objetivos a remoção dos erros, a edição de repetições de cópias idênticas em uma única aresta e, remoção de pontas e arestas com baixa cobertura, chamadas de ligações espúrias. Apesar de utilizar mais memória que algoritmos similares [14], o *velvet* produz *contigs* maiores que o *SSAKE* [55], que o *VCAKE* [37] e que o *SHARCGS* [10] e por este motivo tem se destacado.

Em Abril de 2011 foi lançada versão *multithread* do *velvet* que utiliza para isto o *openmp*. *Velvet* é dividido em duas fases o *velveth* e o *velvetg*, chamados pelo *run.c* e *run2.c*, respectivamente. Não existe uma explicação nem a real necessidade de ele ser dividido em duas fases. Como ele foi adaptado para ser utilizado com os dados do sequenciador *SOLiD*. Para que ele funcione desta forma, deve ser recompilado passando parâmetro *color*. Feita esta compilação, são gerados dois novos executáveis, o *velveth_de* e o *velveg_de*, estes trabalham com a entrada no formato do arquivo *doubleEnconded.de*, saída do *preprocessor*. Ainda, é necessário o parâmetro para que o *velvet* possa se aproveitar do processamento paralelo em *multithread*, fazendo o uso das diretivas *openmp* inseridas no código fonte, para isto ele tem que ser compilado com *OPENMP=1*. Então para o uso do *velvet* neste trabalho, ele foi compilado da seguinte forma:

```
make color 'OPENMP=1'
```

O *velveth* ajuda a construir o conjunto de dados para o programa seguinte, o *velvetg*. Velveth pega uma série de seqüências e produz uma *hashtable*, em seguida, gera dois arquivos em um diretório de saída *Sequences* e *Roadmaps*, que são necessários para *velvetg*.

Para executar diretamente o *velveth* pode ser feito chamando o *script* de execução, informando o diretório de saída e o tamanho do *k-mer*, por fim o arquivo que contém as sequências em *double encoded*. Exemplo:

```
./velveth_de <outvelvet> 31 <doubleEncoded_input.de>
```

O *velvetg* cria um vetor de anotações utilizando os dados do arquivo *Roadmaps* fornecido pelo *velveth*. O *velvetg* simplifica este arquivo retirando as interrupções causadas por términos e inícios de leituras. Com este vetor, ele constrói um pré-grafo, que será um dos arquivos de saída, o *PreGraph*. Depois disto vem o processo de remoção de erros, bolhas, regiões de quimera e repetições, então temos o grafo final, que é escrito no diretório de saída no arquivo *Lastgraph*. Esta fase gera como saída o arquivo *contigs.fa*. Utilizar pré-grafos para montar o grafo de *bruijn* é uma estratégia mais eficiente em relação ao uso de memória [16]. Para executar o *velvetg* diretamente, basta chamar o executável do *velvetg_de* e informar a pasta onde está a saída do *velveth*. Exemplo:

```
./velvetg_de outvelvet -exp_cov auto -cov_cutoff auto
```

Como outros parâmetros para esta fase, tem-se:

- Tamanho de fragmento (*ins_length*): definido com valores entre 80% e 120% da distância aproximada entre as leituras pareadas;
- Profundidade de cobertura esperada (*exp_cov*): definido com valores entre *1/3* e *5/3* da profundidade de cobertura esperada;
- Cobertura mínima (*cov_cutoff*): definido com valores representando *1/4* e *1/5* da profundidade de cobertura esperada;
- Tamanho mínimo de *contig* exportado (*min_contig_lgth*): valores entre 100 e 5.000, a fim de obter mais informações com valores menores e com menos redundância com valores maiores;

Os grafos são construídos no *velvetg* utilizando uma tabela *hash* das leituras indexadas por um *k-mer*. Os *contigs* são produzidos pelo mapeamento das leituras e são formados seguindo as transições do grafo, unindo as que forem ambíguas. Com isto, é possível reduzir a memória necessária para trabalhar com estes grafos, uma vez que sequências de *DNA* possuem muitos dados repetidos, além do mais pela grande cobertura utilizada para possibilitar uma melhor montagem.

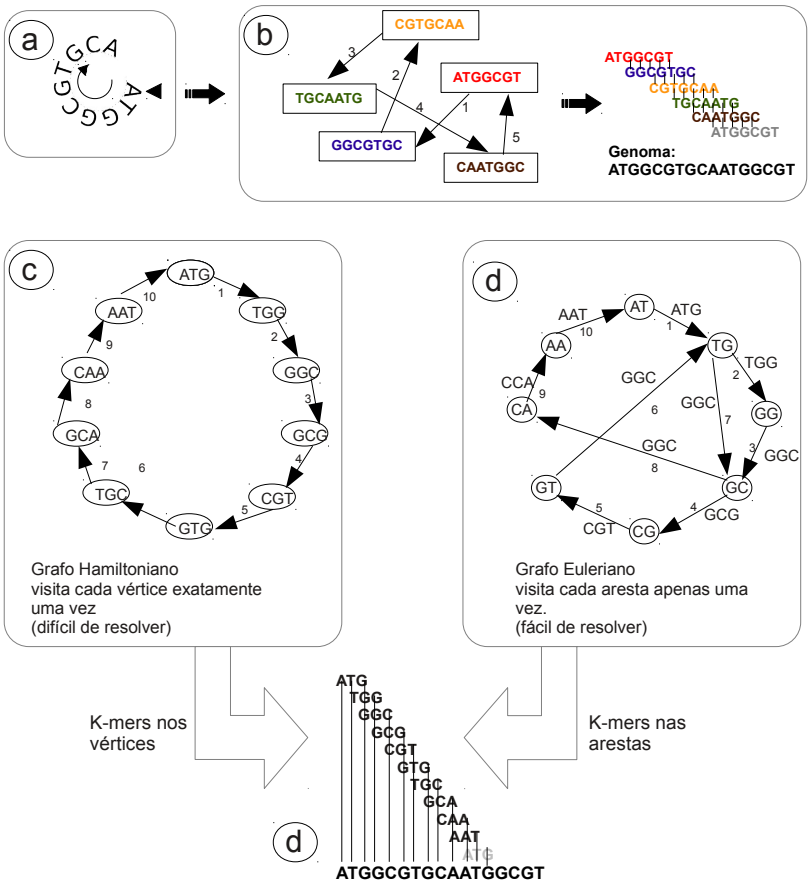

Figura A.5: Exemplos grafo hamiltoniano e de grafo euleriano

A Figura A.5 demonstra um exemplo de duas diferentes estratégias de montagens de genomas, o Caminho Hamiltoniano e o Caminho Euleriano [11]. Em **(a)**, há exemplo de um genoma circular; em **(b)** os algoritmos de montagem *Sanger*, representam as leituras com nós nos grafos, as arestas representam a sequência. Caminhando pelo ciclo hamiltoniano (de solução difícil), seguindo os nós pela sequência das arestas, é possível reconstruir o genoma circular inicial com o alinhamento das leituras. A última leitura, *ATG* demonstrada em **(d)** em cinza por ser a repetição da parte inicial da sequência. Em **(c)**, uma técnica alternativa de montagem, utiliza o *k-mer* com $k = 3$ para *ATGGCGT*, o que compreende *ATG, TGG, GGC, GCG* e *CGT*. Seguindo o caminho hamiltoniano, reconstruindo o genoma com o deslocamento de uma posição. Este procedimento reconstrói o genoma mas não é escalável para grandes gráfos, é um problema de solução difícil. Os algoritmos de montagem de *NGS*, onde o *velvet* está incluso, trabalham com grafo de *bruijn* para representar os *k-mer* são utilizadas as arestas, os nós representam o prefixo e

o sufixo. Por exemplo a aresta *k-mer ATG*, tem o prefixo *AT* e o sufixo *TG*. Seguindo o caminho euleriano, é possível reconstruir o genoma pelo alinhamento sucessivo dos *k-mers* (pelas sucessivas arestas) com deslocamento de uma posição. Esta tarefa monta a sequência do genoma da mesma forma que a tarefa anterior, porém, é resolvido um problema computacional muito mais fácil [43].

A.4 Execução da pós-processamento da montagem

A pós-montagem tem o objetivo de melhorar a qualidade do resultado final. Nela é realizada o fechamento das lacunas mostrado na Subseção A.4.1 e a análise do resultado como apresentado na Subseção A.4.2.

A.4.1 Pós-processamento da montagem - *ASiD*

O Assistente de Montagem para *SOLiD*, *Assembly Assistant for SOLiD - ASiD*, está na versão 1.0, também é da *Life Technologies*, está licenciado sob a *copyrigth* e foi desenvolvido por *Dumitru Brinza* em *C++*.

O fase do *ASiD* é composta por três diferentes programas, sendo o *asid_ assembly.sh* e *asid_ assembly_ one.sh* escritos em *shell script* e o *asid_ light* escrito em *c++*. O objetivo principal do *ASiD* é realizar melhoria do resultado final, para isto ele tenta realizar o fechamento de lacunas, por fim ele realiza uma conversão do formato *double encoded* para o formato *base space*.

O *asid_ assembly.sh* define alguns parâmetros, entre eles a localização do *velvet* a ser utilizado para tentar encontrar os fragmentos para realizar o fechamentos das lacunas. Uma variável recebe todos os arquivos da pasta de lacunas, com isto o *asid_ assembly.sh* realiza chamadas do *asid_ assembly_ one.sh* em *background*, uma para cada um dos arquivos contidos na pasta *gap_ reads*. Por sua vez, o *asid_ assembly_ one.sh* chama o *velvetg* e o *velveth* para cada um destes arquivos de lacunas. Como o objetivo do *ASiD* é realizar

fechamentos de lacunas encontrando trechos faltantes dentros dos fragmentos, de leituras, este processo é repetido quatro vezes variando os parâmetros do *asid_ assembly.sh*. Estas chamadas são realizadas pelo *script assemble.pl*, que é o responsável pelo *pipeline denovo2*.

Ao término desta etapa, o *asid_ light* faz a junção e conversão destes dados para o formato *base-space*. O *asid_ light* é executado utilizando como entradas os arquivos *Lastgraph*, *contigs.fa* e *Sequences* gerados pelo *velvet* além do diretório *gaps_ reads*, que contém as lacunas encontradas. Ele tem como função realizar o fechamento de lacunas entre as sequências (*contigs*) e entre conjuntos de sequências (*scafolds*).

O *ASiD* é o programa que pode ser o responsável pelo maior uso de recursos computacionais do *pipeline* todo, como é a responsabilidade de tentar fechar as lacunas, vai depender de quantos e o tamanho destas lacunas encontradas em cada caso de teste. Para cada uma das lacunas encontradas é gravado um arquivo na pasta *.postprocessor gap_ reads*. O *ASiD* chama um *velvet* para tentar resolver cada um destas lacunas, o que pode gerar de milhares de instâncias do *velvet* rodando com diferentes tamanhos. Caso este que pode consumir muitos recursos computacionais. Para esta tarefa, são executados na sequência os algoritmos representados pelos pseudo-códigos **??** e A.2.

Algoritmo A.1 Apêndice: algoritmo *asid_ assemble.sh*

```
GAPSDE ← lacunas;
para todos lacunas em GAPSDE faça
  Execute em background asid_assembly_one.sh para lacuna[i]
fim para
```

Algoritmo A.2 Apêndice: algoritmo *asid_ one.sh*

```
Recebe lacuna de asid_assemble.sh
Execute velveth(lacuna) depois
Execute velvetg(lacuna)
```

O Processo de tentativa de fechamento destas lacunas ocorre como mostrado na Figura A.6. O *velvet* tenta, dentro do arquivo da própria lacuna, encontrar as sequências necessárias para o fechamento.

Posterior ao trabalho de fechamento de lacunas, o *ASiD* ainda tem outra chamada dentro do *pipeline* que faz o trabalho de pós-processamento, realizando a conversão dos

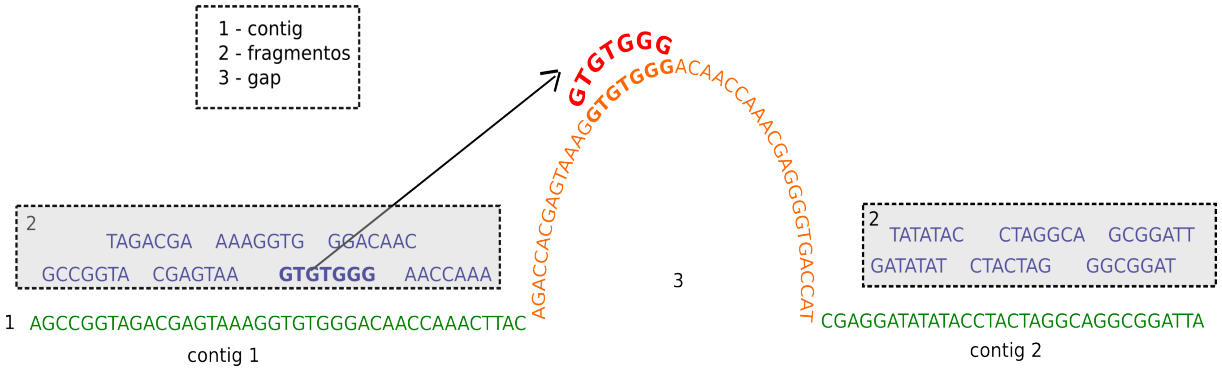

Figura A.6: Esquema do funcionamento do *ASiD*

dados *color space* para o formato *base space*, com a terminologia *.fa*. Para esta conversão, é utilizada a referência *0=A*, *1=C*, *2=T*, *3=G* e *4=N*. Por exemplo, *T01234 -> ACGTN*. Com isto tem-se os arquivos *nt_ scaffolds.fa* e *nt_ contigs.fa* escritos no diretório de saída que são o resultado final.

A.4.2 Pós-processamento da montagem - *analisys*

A fase final consiste na análise e estatísticas referentes a sequência montada. Esta é gerada por um *script* em *perl*, *analyse.pl* sob *copyrigth* da *Life Technologies*. Os dados são escritos no diretório *Analysis* e que tem como subpastas as pastas *contigs*, *nt_ contigs*, *nt_ scaffolds* e *scaffolds*, em cada uma delas os respectivos arquivos *n50.stats.txt* e *cumulative.len.txt*. Como demonstrado a seguir:

Saída do *Analysis*:

- *assembly/analysis/base_ contigs* diretório contendo análise dos *contigsscaffolds* em *base-space*;
- *assembly/analysis/scaffolds* diretório contendo análise dos *scaffolds double encoded*;
- *assembly/analysis/contigs* diretório contendo análise de *contigs double encoded*;

- *assembly/analysis/merge_ scaffolds* diretório contendo análise de *merged scaffolds*;
- *assembly/analysis/merge_ contigs* diretório contendo análise de *merged contigs analysis.*

Cada um deste diretórios contém os seguintes arquivos:

- *n50.stats.txt* arquivo com estatísticas dos *contigs*;
- *cumulative.len.txt* uma lista dos *contigs* ordenada em ordem decrescente e a soma deles.

A.5 *Acompanhamento de uma rodada do denovo2*

Para exemplificar e demonstrar o fluxo de arquivos criados, será apresentado um acompanhamento na execução da bactéria *Escherichia coli str. K-12 substr. MG1655*, de tamanho 4,6 *Mbp*.

Para executar um teste com a bactéria *E.coli*, foi utilizada a seguinte linha de comando:

```
./assemble.pl ecoli_600x_F3.csfasta ecoli_600x_F3.qual 4600000
-r3 ecoli_600x_R3.csfasta -r3qv ecoli_600x_R3.qual -outdir out
 -ins_length 1200 -ins_length_sd 500
```

Tabela A.6: Entrada e saída do *pipeline denovo2*

a	**b**	**c**
Arquivos de Entrada:	F3.csfasta	R3.csfasta F3.qual R3.qual
Arquivos de Saída:	out (20G)	postprocessor/gap_reads () x (1,9G) y (1,9G)

A Tabela A.6 demonstra os arquivos de entrada e seus tamanhos, e compara com o tamanho da saída gerada.

A Tabela A.7 demonstra um exemplo de Fluxo de Execução do *Pipeline denovo2*.

Tabela A.7: Fluxo de execução do *pipeline denovo2*

Fase	**Tempo**	**Pico de Memória**	**Arquivos Criados**
rsampling	5min	1GB	subreads2.csfasta subreads2.qual subreads.csfasta subreads.qual
SAET	60min	8.7GB VS max 2GB	./fixed/saet.log.txt ./fixed/spect.[0-X].swp (temporário) ./fixed/subreads.csfasta.swp (temporário) ./fixed/subreads.csfasta (1,9G)
preprocessor	5min	5GB	.preprocessorcolorspace_input.csfasta (1,9G) .preprocessordoubleEncoded_input.de (1,8G)
velveth	30min	15GB	Sequences (M) Roadmap ()
velvetg	5min	15GB	PreGraph (20M) Graph2 (720 M) LastGraph (530 M) contigs.fa (5 M) stats.txt (500 K)
ASiD	15min	1GB	./postprocessor/gap_reads (20 MB) asid_ntcontigs.tmp (6,6 M) color_reads.ma (1,8 G) colorspace_input.idx (440 M)
analisys	5min	1GB	n50.stats.txt *cumulative.len.txt*

Este Apêndice A detalhou a função e funcionamento de cada cada fase do *pipeline denovo2*. Foram mostrados os arquivos de entrada e de saída de cada uma delas e alguns parâmetros Todo este pipeline e os programas que o compõe, são carentes de documen-

tação. Este trabalho documentou e analisou o funcionamento e comportamento de cada uma das fases do *denovo2*. Usualmente não é comum para este fim executar individualmente cada um destes programas, esta tarefa é realizada em lote pelo *script assemble.pl*, o que constitui então o *pipeline denovo2*. A execução individual foi demonstrada apenas para fins de melhorar a compreensão da função de cada uma das fases.