SÉRGIO YOSHIMITSU FUJII

UTILIZAÇÃO DE TÉCNICAS DE OTIMIZAÇÃO DE DESEMPENHO EM BIOINFORMÁTICA. ESTUDO DE CASO: SRNASCANNER

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Bioinformática, Setor de Educação Profissional e Tecnológica, Universidade Federal do Paraná.
Orientador: Prof. Dr. Lucas Ferrari de Oliveira
Orientadora: Prof^a. Dr^a. Maria Berenice Reynaud Steffens

CURITIBA

2012

AGRADECIMENTOS

Após estes dois anos de trabalho intenso, conheci pessoas que mostraram a sua importância e me ajudaram a superar este desafio. Gostaria expressar a minha sincera gratidão às seguintes pessoas:

Aos meus pais, pelo incentivo constante aos meus estudos.

Ao meu orientador Lucas Ferrari de Oliveira que, além de ótimo orientador, é também um grande amigo. O seu conhecimento na área computacional é tão grande quanto a sua motivação em ensinar e ajudar os seus alunos. Este trabalho não seria possível sem a sua ajuda! Gostaria de deixar registrado o meu agradecimento à sua dedicação, tanto neste trabalho quanto nas disciplinas lecionadas. Pessoas como você são as que mantêm a esperança do país em alcançar um alto nível de educação e pesquisa.

À minha co-orientadora Maria Berenice Steffens, que muito me ajudou em questões de Biologia Molecular oferecendo vários artigos para leitura e sempre me recebeu com muito carinho e disponibilidade em todos os nossas reuniões, contribuindo imensamente para o trabalho.

Aos demais professores do Programa de Pós-Graduaçao em Bioinformática.

Às secretárias Léa e Suzana pela amizade e também à disponibilidade diária às nossas necessidades burocráticas.

Aos meus colegas Alysson, Bárbara, Danhylo, Gustavo, Jesse, Juliana, Katia, Leviston, Lucas, Paula, Rafael, Rafaela, Vanely e William pelos ótimos momentos que passamos juntos no laboratório de Bioinformática.

Aos amigos que estão distantes, mas que estão sempre presentes na minha vida: Otávio, Gustavo, Beatriz, Eduardo, Renato e Érico.

À Ana Cláudia de Abreu Ferrari de Oliveira, por apoiar o meu trabalho juntamente com o meu orientador e me receber diversas vezes em sua residência durante o mestrado, seja a trabalho ou a lazer.

Aos órgãos de fomento CAPES e CNPq pelo auxílio financeiro com a bolsa de pós-graduação e verbas de projeto.

SUMÁRIO

Lista de Abreviaturas e Siglas vi

LISTA DE FIGURAS ix

LISTA DE TABELAS x

RESUMO xi

ABSTRACT xiii

1 INTRODUÇÃO 1

1.1 Objetivo 2

1.2 Justificativa 3

2 FUNDAMENTOS TEÓRICOS EM BIOLOGIA MOLECULAR 4

2.1 RNAs Curtos 4

2.1.1 Promotores 5

2.1.1.1 Identificação Computacional de Promotores 6

2.1.2 Terminadores de Transcrição 6

2.1.2.1 Identificação Computacional de Terminadores 7

2.1.3 Estrutura Secundária 8

3 FUNDAMENTOS EM COMPUTAÇÃO DE ALTO DESEMPENHO 10

3.1 Hierarquia de Memória 10

3.2 Computação de Múltiplos Núcleos 11

3.3 Análise de Desempenho 12

3.4 Profilers 13

3.4.1 Gprof 14

3.4.2 Open|SpeedShop 15

3.5 Computação de Alto Desempenho 15
3.6 Bibliotecas 16
3.6.1 STL - *Standard Template Lybrary* 17
3.6.2 POSIX Threads (Pthreads) 19
3.6.3 Bibliotecas de Alto Desempenho 19
3.7 Técnicas de Otimização Manual 20
3.7.1 Eliminação de sub-rotinas 20
3.7.2 Macro 21
3.7.3 *Inlining* 22
3.7.4 Otimização de laços de repetição 23
3.7.4.1 Desvio condicional em laços de repetição 23
3.7.4.2 Desenrolar de laços 24
3.7.5 Conversão de tipos 24
3.7.6 Otimizações aritméticas 25
3.7.6.1 Redução de esforço 26
3.7.6.2 Eliminação de sub-expressões 27

4 SRNASCANNER 28

4.1 Matriz de Peso Posicional (MPP) 28
4.2 PWM_create 31
4.3 Funcionamento 32
4.4 Parâmetros de entrada 33
4.5 Arquivos de saída 36
4.6 Discussão 37

5 MATERIAIS E MÉTODOS 38

5.1 Estação de Trabalho 38
5.2 Ferramentas 38
5.2.1 Profilers 38
5.3 Bactéria 39

6 RESULTADOS **40**

6.1 sRNAScanner Original 40

6.2 sRNAScanner Improved 1a. versão 44

6.3 sRNAScanner Improved 2.a versão 46

6.4 sRNAScanner Improved 3a. versão 50

6.4.1 Otimizações de Conversão e Acesso a Disco 50

6.4.2 Otimizações de Desvios Condicionais 58

7 DISCUSSÃO **63**

8 CONCLUSÃO **65**

8.1 Trabalhos futuros 66

LISTA DE ABREVIATURAS E SIGLAS

A	Adenina
BLAST	*Basic Local Alignment Search Tool*
C	Citosina
CAD	Computação de Alto Desempenho
CPU	*Central Processing Unit*
CSS	*Cumulative Sum of Score*
CUDA	*Compute Unified Device Architecture*
DDR	*Double Data Rate*
DNA	*Desoxiribonucleic Acid* (Ácido Desoxirribonucléico)
HPC	*High Performance Computing*
G	Guanina
GB	Gigabyte
GNU	GNU is Not Unix
Gprof	GNU profiler
GPU	*Graphics Processing Unit*
MB	Megabyte
MPI	*Message Parsing Interface*
MPP	Matriz de Peso Posicional
mRNA	*messenger Ribonucleic Acid* (Ácido Ribonucleico mensageiro)
NNPP	*Neural Network Promoter Prediction*
POSIX	*Portable Operating System Interface*
PWM	*Position Weight Matrix*
RAM	*Random Access Memory*
RNA	*Ribonucleic Acid* (Ácido Ribonucleico)
SAP	Soma Acumulativa de Pontuação

sRNA	*small Ribonucleic Acid* (Ácido Ribonucleico curto)
STL	*Standard Template Library*
T	Timina
U	Uracila
UT	Unidade Transcricional

LISTA DE FIGURAS

2.1 Esquema de uma região promotora de um gene em procariotos cuja transcrição é dependente de sigma 70. As sequências -10 e -35 estão destacadas em azul escuro, enquanto o início da transcrição em verde. Fonte: [Campbell et al., 2001] 5
2.2 Exemplo de uma estrutura formadora de grampo. Fonte: [Ponty, 2012] . 7
2.3 Estruturas secundárias de cinco RNAs curtos diferentes encontrados na archaea *Methanocaldococcus jannaschii*. Fonte: [Thébault et al., 2006] 9

3.1 Relatório de saída do programa original utilizando o profiler gprof. . . . 14
3.2 Exemplo de relatório de saída do Open|SpeedShop utilizando a interface gráfica. 16

4.1 Exemplo de uma matriz de alinhamento. A matriz contém o número de ocorrências $n_{i,j}$ para cada letra i que ocorre na posição j do alinhamento. Abaixo, a sequência consenso é extraída a partir do número de ocorrência de nucleotídeos das quatro sequências alinhadas. N indica que não importa o nucleotídeo, pois nessa posição não há letra dominante. Fonte: [Hertz and Stormo, 1999] 29
4.2 Matriz de peso posicional derivada da matriz de alinhamento mostrada na Figura 4.1. Fonte: [Hertz and Stormo, 1999] 30
4.3 MPPs das regiões promotoras -35 e -10. Fonte: [Sridhar et al., 2010] . 32
4.4 sRNAs da *Escherichia coli* K-12 utilizados para gerar as MPPs das regiões promotoras -35 e -10. a) Sequência da região promotora -35. b) Sequência da região promotora -10. c) sRNAs correspondentes às regiões promotoras. Fonte: [Sridhar et al., 2010] 33
4.5 Terminadores de transcrição da *Escherichia coli* K-12 utilizados para gerar a MPP da região terminadora. Fonte: [Sridhar et al., 2010] 34

4.6 MPPs construídas utilizando sinais transcricionais anteriormente definidas para dez sRNAs da *Escherichia coli* K-12. a) MPP construída utilizando a região promotora -35. b) MPP construída utilizando a região promotora -10. c) MPP construída utilizando região do terminador rho-independente. Fonte: [Sridhar et al., 2010] 35

6.1 Dados do arquivo de saída `RNA_position`. Contém as coordenadas de início e fim dos sRNAs encontrados pelo sRNAscanner. 40

6.2 Dados do arquivo de saída `sRNA.txt`. Contém as coordenadas de início e fim dos sRNAs encontrados pelo sRNAscanner. Abaixo de cada par de coordenadas, os nucleotídeos correspondentes a cada região. . 41

6.3 Relatório de saída do programa original utilizando o profiler gprof. . . . 41

6.4 Relatório de saída do programa original através do Open|SpeedShop. . 42

6.5 Relatório de saída do Open|SpeedShop utilizando a primeira versão otimizada do programa. 50

6.6 Gráfico comparativo entre os tempos médios de execução da versão original e das três versões otimizadas do sRNAscanner. 61

6.7 Gráfico comparativo entre o speedup das três versões otimizadas do sRNAscanner. 62

LISTA DE TABELAS

6.1 Comparação entre as versões do sRNAscanner em relação ao tempo médio de execução. 60
6.2 Comparação entre as versões do sRNAscanner em relação ao *speedup*. 61

RESUMO

A área de bioinformática passou por um crescimento na quantidade de dados biológicos em formato digital devido ao desenvolvimento de novas tecnologias de sequenciamento de DNA e análise de expressão gênica. A base de dados vem crescendo em altas taxas anualmente e, atualmente, o desafio é a transformação dos dados biológicos digitais em conhecimento. No entanto, a dificuldade para processar a enorme quantidade de dados é apenas um dos vários problemas relacionados a este crescimento. A área de Ciência da Computação pode auxiliar na melhoria da bioinformática através da investigação focada em métodos eficientes para a montagem de genomas, expressão gênica, alinhamento de sequências, mineração de dados, predição de sRNA, entre outros. Além da grande quantidade de memória requerida, ferramentas de bioinformática também exigem alta capacidade de processamento. A computação de alto desempenho (CAD), incorporando vários núcleos de processador em uma placa com memória compartilhada e oferecendo técnicas de otimização de código, vem mudando paradigmas na área de computação. Ferramentas de bioinformática modernas precisam tirar proveito da computação paralela, o que sempre foi uma tarefa desafiadora. Porém, a conversão de código sequencial em paralelo é uma tarefa difícil e deve ser precedida por otimização. Essa otimização envolve tornar o programa o mais eficiente possível. Técnicas de otimização manual, por exemplo, otimização aritmética, eliminação de conversão de dados e otimização de loop, ajudam a melhorar o tempo de execução da aplicação. Este trabalho apresenta otimizações realizadas no software sRNAScanner, cuja motivação foram as respostas biológicas do software e o seu tempo de execução elevado. Para otimizar o programa, ferramentas de perfilação foram utilizadas para analisar e avaliar o seu desempenho. O software sofreu alterações em suas funções e em suas estruturas de dados. Utilizando o genoma da bactéria *Salmonella enterica serovar Typhimurium* e técnicas de otimização manual e programação paralela, o tempo médio diminuiu de 23 minutos para 16,283 segun-

dos, apresentando um aumento de desempenho (*speedup*) de 85 vezes. Os arquivos finais e temporário tiveram o conteúdo inalterado em comparação com os mesmos arquivos gravados pelo programa sRNAScanner original. Os resultados mostraram que a aplicação de técnicas de otimização utilizadas em computação de alto desempenho em ferramentas de bioinformática apresentou, neste caso, um ganho de desempenho expressivo.

ABSTRACT

The Bioinformatics area has experienced a rapid growth in the amount of digital biological data due to the development of new high-throughput technologies for DNA sequencing and gene expression analysis in Genomics. The databases growth on high rates annually, and a current challenge is to transform digital biological data into knowledge. However, the difficulty to process the huge amount of data is just one of the several problems regarding to biological database growth. The Computer Science area can help the improvement of bioinformatics through focused research on efficient methods for genome assembly, gene expression, sequence alignment, data mining, sRNA prediction, among others. In addition to a large amount of required memory, bioinformatic tools also demand high processing capacity. Nowadays, modern multicore and manycore architectures have been revolutionizing high-performance computing (HPC), embodying multiple processor cores into a single hardware motherboard with shared memory. Modern bioinformatic tools need to take advantage of parallel computing, which has always been a challenging task. Though, to convert a sequential code into a parallel one is a difficult task and should be preceded by code optimization. This optimization involves making the program as efficient as possible. Many hand tuning techniques might help to improve the application execution time, e.g. arithmetic optimization, data type conversions and loop optimization. In this work, it is presented a hand tuning optimization on software sRNAScanner. However, its biological appeal and elevated runtime has motivated us to analyze and improve its code. To optimize the program, profiling tools were used to evaluate its performance. Thereby, these tools reported the less effective points, which guided us to the focus of optimization. The software's functions and data structures were refactored. Using the genome of the bacterium *Salmonella enterica serovar Typhimurium* and applying the hand tuning and parallel programming techniques, the average runtime decreased from 23 minutes to 16,283 seconds, featuring a speedup of 85. The final and temporary files genera-

ted had their content unchanged when compared with the same files recorded by the original sRNAScanner program. The result showed that HPC optimization techniques applied to bioinformatic tools presents an expressive performance boost.

CAPÍTULO 1

INTRODUÇÃO

O entendimento humano na área de bioinformática é impulsionado pela enorme quantidade de dados que podem ser coletados com equipamentos automatizados que possuem grande capacidade computacional para extrair, analisar, modelar e visualizar os resultados [Farber, 2011].

As técnicas de sequenciamento de DNA em alta performance e análise de expressão gênica estão fornecendo um crescimento exponencial na quantidade de dados biológicos em formato digital. Os exemplos mais claros são o crescimento da informação de sequências de DNA na base de dados GenBank do NCBI (National Center for Biotechnology Information) [1] e o crescimento da base de dados de sequências de proteínas UniProtKB/TrEMBL [2]. Além disso, a 2^a geração de sequenciadores tem quebrado muitas barreiras experimentais na escala de sequenciamento genômico, facilitando a extração de grandes quantidades de sequências que irão proporcionar o futuro crescimento dos bancos de dados biológicos [Schmidt, 2010].

Sob este mesmo aspecto, a detecção de RNAs curtos (*small RNA* - sRNA) não codificantes em bactérias está sendo objeto de estudos em vários artigos de pesquisa, pois eles contribuem em muitos processos celulares de sobrevivência, patogenia e adaptações [Storz et al., 2005]. Em procariotos, os sRNAs exercem uma grande quantidade de diferentes funções como a regulação de esporulação, homeostase de ferro, metabolismo de glicose, reparação de danos, sobrevivência sob estresse oxidativo, manutenção da superfície celular e regulação de patogenicidade [Gottesman and Storz, 2010]. Embora os sRNAs não sejam codificados a peptídeos eles executam funções por meio de pareamento de base sRNA-mRNA ou por antagonizar proteínas alvo através de interações RNA-proteínas [Sridhar et al., 2010].

[1] http://www.ncbi.nlm.nih.gov/genbank/
[2] http://www.uniprot.org/

Várias ferramentas já foram desenvolvidas para a detecção de sRNA. Alguns exemplos podem ser citados, tais como: sRNAScanner [Sridhar et al., 2010], RNAz [Gruber et al., 2010], sRNAPredict [Livny et al., 2005], *Intergenic Sequence Inspector* [Pichon and Felden, 2003] e QRNA [Rivas and Eddy, 2001]. O custo computacional das soluções apresentadas varia de algumas horas para alguns segundos. Porém, é impossível compará-los devido às peculiaridades de cada implementação. No começo da pesquisa, o programa mais recente era o sRNAScanner, que gasta em torno de 48 minutos para executar a busca completa em um genoma de bactéria Escherichia coli K12 MG1655 [Sridhar et al., 2010].

Pesquisadores das áreas de ciência da computação e biólogos enfrentam o desafio de transformar dados genômicos em entendimento biológico. Por isso, as ferramentas de bioinformática devem ser escaláveis, ou seja, elas precisam lidar com um crescimento constante da quantidade de informações. A quantidade de dados biológicos públicos cresce mais rápido do que a capacidade de processamento de um único núcleo do processador. Por isso, as ferramentas modernas de bioinformática precisam tirar vantagem da computação paralela, principalmente das arquiteturas multicore e many-core que estão revolucionando a computação de alto desempenho (CAD) [Schmidt, 2010].

Porém, antes de paralelizar um programa, é necessário melhorar a sua eficácia através de técnicas de otimização. Paralelizar um programa ineficaz apresenta resultados igualmente ineficazes [Cunha et al., 2001].

1.1 Objetivo

O objetivo deste trabalho consiste em estudar técnicas de otimização de algoritmos já consolidadas e aplicá-las a um problema de bioinformática, mostrando em quais aspectos elas podem ser utilizadas. As técnicas foram aplicadas em uma ferramenta de busca de sRNA que foi utilizada como estudo de caso.

1.2 Justificativa

O surgimento das arquiteturas multi e many-cores, bem como outras tecnologias de aceleração, tais como “Field-Programmable Gate Array” (FPGA) e o Cell/BE (da Sony), permitem a redução do tempo de execução de muitos algoritmos biológicos disponíveis normalmente em hardwares de baixo custo com mais poder para alta performance computacional [Schmidt, 2010]. Essas novas arquiteturas paralelas computacionais criam os seguintes desafios no campo da bioinformática:

- Os novos algoritmos em bioinformática e as aplicações precisam tirar vantagem das novas arquiteturas;
- Ferramentas já existentes precisam ser portadas de forma eficiente para as arquiteturas paralelas que surgem.

Na área de alinhamento genômico, diversos algoritmos e aplicações já foram desenvolvidos. A ferramenta BLAST (*Basic Local Alignment Search Tool*) foi paralelizada para utilização do MPI (*Message Passing Interface*) e também da GPU (*Graphics Processing Unit*), com uma versão chamada GPU-Blast [Vouzis and Sahinidis, 2011]. O algoritmo de alinhamento de sequência Smith-Waterman também foi paralelizado e recebeu várias implementações utilizando CUDA ([Liu et al., 2011], [Liu et al., 2010], [Manavski and Valle, 2008] e [Schatz et al., 2007]).

CAPÍTULO 2

FUNDAMENTOS TEÓRICOS EM BIOLOGIA MOLECULAR

2.1 RNAs Curtos

Os RNAs curtos, denominados sRNA (*small* RNA), são pequenas moléculas com número variável de bases, sendo mais comumente encontradas entre 50 e 250 bases, porém há estudos que afirmam que o número de nucleotídeos pode chegar a 600 [Majdalani et al., 2005]. São transcritas a partir de regiões intergênicas do genoma bacteriano [Altuvia, 2007], e, portanto, não sofrem o processo de tradução. Os RNAs apresentam dois principais alvos: os mRNAs (RNA mensageiro) e as proteínas regulatórias. Nos polipeptídeos estas moléculas alteram a sua estabilidade, dificultando ou facilitando a sua afinidade com o sítio de ligação para transcrição de determinado gene [Vogel and Wagner, 2007]. Entretanto, a maioria dos RNAs curtos de bactérias é caracterizada por agir como regulador pós-transcricional, modulando a estabilidade do RNA mensageiro e/ou alterando o acesso do RNA mensageiro ao maquinário de tradução. Realizam pareamento perfeito de aproximadamente sete ou oito bases consecutivas com o mRNA, dificultado a catálise ribossomal e polimerização de aminoácidos [Busch et al., 2008]. Geralmente essas interações são estabilizadas pela proteína Hfq, uma chaperona conservada em inúmeras espécies de bactérias [Valverde et al., 2008]. Essas reações são respostas às variações no ambiente e estão envolvidas na regulação de diversos processos biológicos [Zhang et al., 2004]. Por muitos anos, os RNAs curtos passaram despercebidos devido à falta de abordagens metodológicas adequadas e difícil detecção e, ainda hoje, a maioria de suas funções e propriedades necessita ser avaliada. Apesar dos esforços e das técnicas citadas, ainda não se pode definir com certeza a quantidade deles nos genomas bacterianos. Porém, estima-se que existam entre 200 e 300 em um genoma médio de 5 milhões de bases, como é o caso da *Escherichia coli* [Riley et al., 2006].

Grande parte dos RNAs curtos atualmente é descoberto por meio de predições computacionais utilizando diversas técnicas, como conservação das sequências, análise dos sinais de transcrição e predição de suas estruturas secundárias.

2.1.1 Promotores

O promotor é uma subseqüência constituinte do gene, cuja função é controlar e regular o processo de transcrição, e consequentemente, a expressão gênica de determinado produto. Está localizado a montante do códon de início de transcrição. Essa região é reconhecida pela enzima RNA polimerase, responsável pela síntese do RNA a partir do DNA molde. Uma subunidade da RNA polimerase, o fator sigma, está envolvido apenas no início da transcrição. A função de sigma é reconhecer e ligar a RNA polimerase aos sítios promotores de DNA, como um sinalizador [Snustad and Simmons, 2008].

Em procariotos, o principal promotor e o mais estudado é conhecido como sigma 70, e é composto por duas seqüências curtas de posição -10 e -35 a montante do início da transcrição (Figura 2.1).

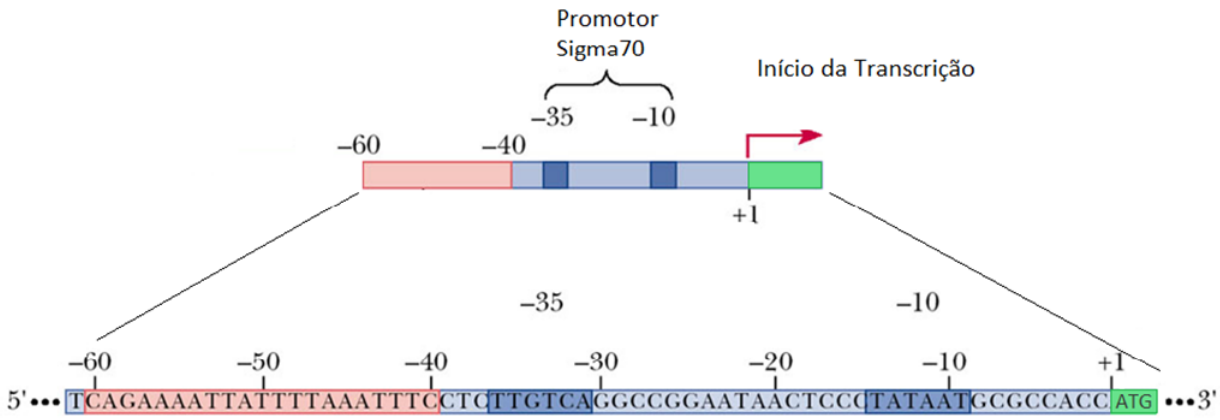

Figura 2.1: Esquema de uma região promotora de um gene em procariotos cuja transcrição é dependente de sigma 70. As sequências -10 e -35 estão destacadas em azul escuro, enquanto o início da transcrição em verde. Fonte: [Campbell et al., 2001]

A sequência -10 é chamada de Pribnow Box, normalmente consistindo de seis nucleotídeos com uma sequência consenso[1] TATAAT. O Pribnow Box é absolutamente

[1]Sequência consenso é uma sequência teórica dada por uma sucessão de bases altamente conservadas, determinada de acordo com aquela que ocorre com maior freqüência na natureza.

essencial para o início da transcrição. A outra sequência, que é a -35, normalmente também possui seis nucleotídeos, e a sua sequência consenso é TTGTCA.

2.1.1.1 Identificação Computacional de Promotores

Para predizer promotores sigma 70, um dos principais softwares existentes é o *Neural Network Promoter Prediction* (NNPP) [Reese, 2001]. A base do NNPP é uma rede neuronal baseada em RNAs curtos comprovados de *Escherichia Coli*. A rede consiste principalmente em duas camadas, uma para reconhecer o Pribnow Box e o outra para reconhecer o sítio de início de transcrição. Ambas as camadas são combinadas em uma unidade de saída, que retorna um escore entre 0 e 1. A precisão é de aproximadamente 80% com um cutoff de 0,8. O software também analisa a fita complementar de forma automática, não sendo necessário efetuar a função complementar reversa e realizar novo processamento. Além do NNPP, existe o BPROM, software com uma precisão similar ao NNPP e que combina características descrevendo motivos funcionais e composição oligonucleotídica.

2.1.2 Terminadores de Transcrição

Um dos fatores pós-transcricionais que podem ser utilizados para a detecção dos RNAs curtos é o Terminador. Terminadores são porções genéticas que normalmente ocorrem no final de um gene ou operon e obrigam a transcrição a parar. Este sinal de término faz com que a enzima polimerase se dissocie, causando a liberação da molécula re RNA recém transcrita. Em procariotos, os terminadores usualmente se dividem em duas categorias: Terminadores Rho-independente e terminadores Rho-dependentes [Valverde et al., 2008]. Terminadores Rho-dependente necessitam de um fator auxiliar para finalizar o processo de transcrição, a proteína Rho. Já o mecanismo de dos terminadores Rho-independentes está relacionado com a sequência de bases no local. São geralmente compostos por uma sequência complementar palíndroma que forma uma região rica em G-C. Quando esta sequência é transcrita,

a região do RNA unifilamentar pode fazer pareamento de bases e gerar uma estrutura conhecida como hairpin ou grampo (Figura 2.2). Na maioria dos terminadores, o grampo é seguido por quatro ou mais uracilas [Snyder and Champness, 2007]. O modelo convencional de terminação transcricional é que o grampo causa a parada da RNA Polimerase como consequência da sua modificação conformacional, o que causa uma torção na fita e dificulta o funcionamento normal da enzima.

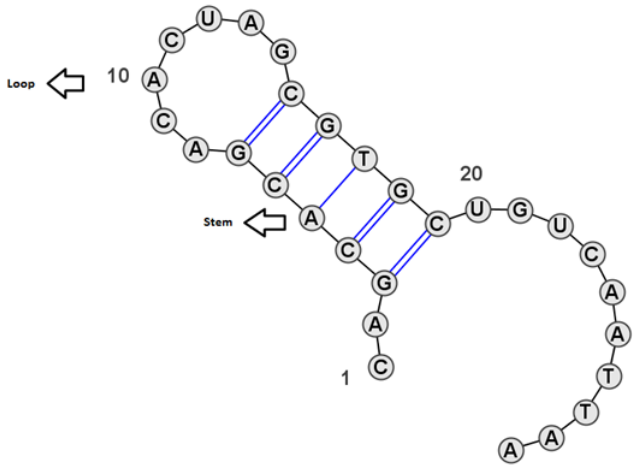

Figura 2.2: Exemplo de uma estrutura formadora de grampo. Fonte: [Ponty, 2012]

2.1.2.1 Identificação Computacional de Terminadores

Para realizar a predição de terminadores, existem dois softwares bastante populares: RNAMotif [Macke et al., 2001] e Transterm [Jacobs et al., 2008]. O primeiro deles utiliza poderosos "motivos" que permitem ao pesquisador especificar em detalhes o tamanho mínimo e máximo, tanto do tronco quanto do loop, que juntos formam o grampo. Além disso, é possível especificar a quantidade mínima de determinada base e a interação entre elas. Em suma, o software cria um banco de dados com sequências que "casam" com o "motivo". São necessários dois parâmetros para a execução: um descritor e um arquivo de regiões intergênicas no formato fasta, no qual as sequências serão procuradas. Ambos retornam um arquivo contendo os respectivos índices e sequências dos possíveis terminadores, juntamente com um escore. Quando os dois softwares são utilizados em conjunto, duas estratégias são possíveis: a primeira

delas é selecionar apenas os terminadores no qual ambos os softwares obtiveram êxito, aumentando consideravelmente a probabilidade de serem terminadores verdadeiros, mas, por outro lado, perdendo muitos terminadores que por alguma razão um dos softwares não conseguiu registrar. A segunda estratégia é utilizar todos os registros dos dois softwares, aumentando bastante o número de terminadores, porém, diminuindo sobremaneira a confiabilidade das predições.

2.1.3 Estrutura Secundária

Uma importante característica dos RNAs curtos é a presença de uma estrutura secundária bem conservada entre espécies próximas. Por consistir em uma fita simples, o RNA tende a formar estruturas secundárias específicas a fim de melhorar a sua estabilidade no ambiente intracelular e se proteger da ação catalítica enzimática. A estrutura primária do RNA é constituída pela sequência de nucleotídeos, enquanto a secundária é definida pela sua conformação. O RNA não possui uma estrutura secundária regular e simples que sirva como ponto de referência, como é a dupla de DNA. Sua conformação é mantida principalmente por interações fracas entre as bases, de acordo com a sua complementaridade. Quebras na hélice causadas por bases mal ou não pareadas em uma ou em ambas as fitas são comuns e resulta em saliências ou alças internas; alças em grampo formam-se entre sequências quase autocomplementares (Figura 2.3). O potencial para estruturas helicoidais pareadas por bases em muitos RNAs é muito grande e os grampos resultantes podem ser considerados o tipo mais comum de estrutura secundária do RNA [Nelson and Lehninger, 2008]. Diversos softwares realizam a análise da estrutura secundária de sequências. Pode-se citar como exemplo o RNAz, aplicativo para plataforma Linux, que utilizando alinhamentos múltiplos, consegue detectar estruturas de RNAs funcionais, como as encontradas em RNAs não-codificantes. Os alinhamentos podem ser obtidos por meio da ferramenta Clustal [Thompson et al., 1994].

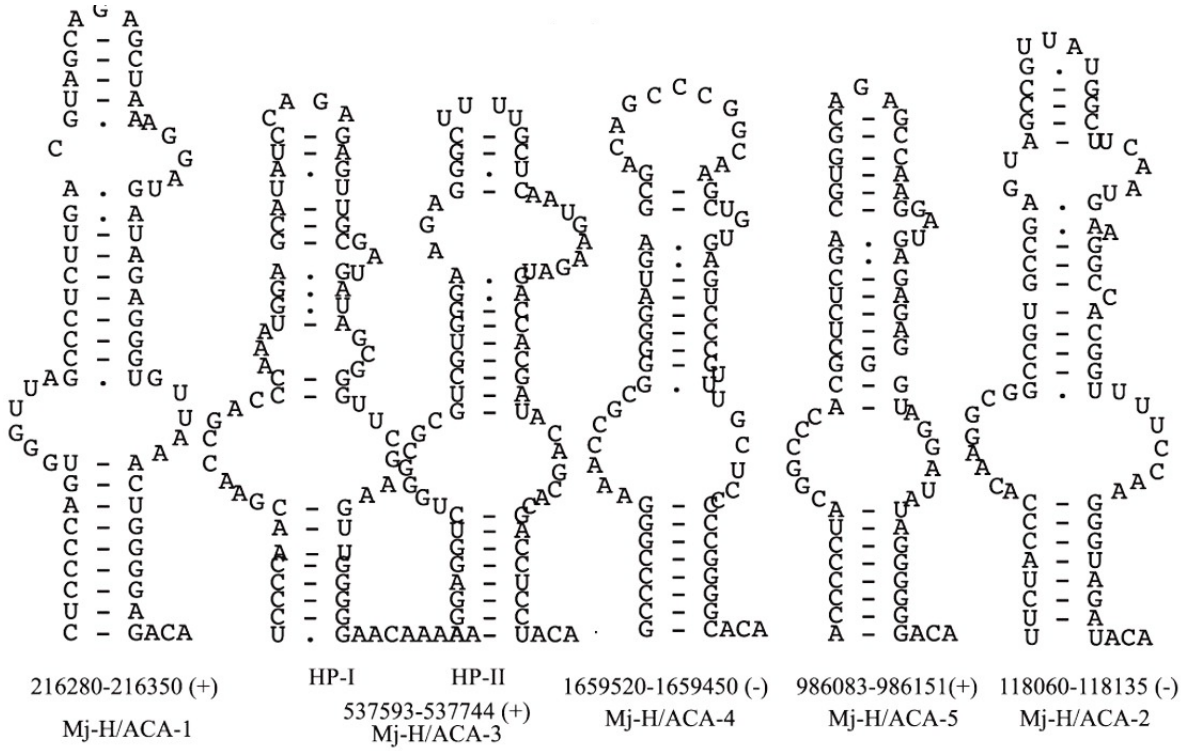

Figura 2.3: Estruturas secundárias de cinco RNAs curtos diferentes encontrados na archaea *Methanocaldococcus jannaschii*. Fonte: [Thébault et al., 2006]

CAPÍTULO 3

FUNDAMENTOS EM COMPUTAÇÃO DE ALTO DESEMPENHO

3.1 Hierarquia de Memória

Para realizar determinada operação, o computador utiliza o processador para buscar os dados na memória e então processá-las. O custo para essa operação é dada pela soma dos ciclos de clock do processador mais o tempo de acesso aos registros na memória. Em um cenário ideal, processador e memória trocariam dados na mesma velocidade. No entanto, a largura de banda do barramento do processador é muito maior do que a do barramento da memória principal. Utilizando a seguinte fórmula, podemos calcular a largura de banda:

$$Largura\_de\_banda = \frac{clock * bits\_transferidos\_por\_pulso\_de\_clock}{8}$$

Por exemplo, com um processador de 3GHz e 64 bits, a largura de banda é de 24GB/s. Com uma memória RAM DDR3 de 800MHz e 64 bits, a largura de banda é de 10,6GB/s. Essa situação é tipicamente chamada de gargalo, ou *bottleneck*. Para amenizar este problema, são utilizadas memórias intermediárias entre a memória principal e o processador, chamada memória cache. Essa memória, que geralmente trabalha na mesma frequência do processador, é utilizada para diminuir o tempo total gasto na comunicação entre o processador e a memória principal. Os dados são copiados da memória principal para a cache em segmentos chamados cache line. Então, esses dados são lidos pelos registradores do processador e executados. Se a instrução seguinte requisitar um dado que estiver no mesmo segmento, o processador pode executar sem precisar buscar o dado na memória principal. Essa situação resulta em um cache hit. No entanto, se o dado não estiver na memória cache, será preciso buscá-lo na memória principal, resultando em um cache miss. O aproveitamento da

memória cache é baseado no princípio da localidade de referência, em que o processador apresenta a tendência de referenciar instruções e dados que estão localizados em endereços próximos na memória principal [Patterson and Hennessy, 2009].

3.2 Computação de Múltiplos Núcleos

Por trinta anos o mais importante método para aumentar a performance dos dispositivos computacionais era aumentar a velocidade de operação de clock do processador. Os primeiros computadores pessoais (PC) da década de 80 tinham uma unidade central de processamento (UCP, em inglês CPU) que operava com um clock interno de 1MHz. Trinta anos mais tarde a maioria dos computadores pessoais tem velocidade de clock entre 1GHz e 4GHz, aproximadamente 1.000 vezes mais rápido que o clock dos PCs originais. Atualmente os fabricantes foram forçados a procurar alternativas para esta tradicional fonte de aumento da força computacional. O limite físico na fabricação de circuitos integrados foi atingida, não sendo mais viável o aumento da velocidade do clock como forma de aumentar a força computacional das arquiteturas existentes [Sanders and Kandrot, 2010].

Desde 2005 as grandes fabricantes de processadores oferecem aos usuários novos processadores com pelo menos dois núcleos (cores). Na atualidade processadores com três, quatro, seis ou oito cores podem ser encontrados para compra. A estratégia de aumento de núcleos seguiu a estratégia de anos atrás dos supercomputadores, porém com um custo muito mais baixo. Sendo que hoje é praticamente impossível comprar um computador com somente um núcleo de processamento. Os fabricantes já anunciaram planos para CPUs com 12 e 16 cores de processamento, confirmando que a computação paralela é uma realidade [Sanders and Kandrot, 2010].

Atualmente, máquinas multi-core são uma realidade. No passado, a velocidade dos processadores dobrava a cada dois anos. Porém, a tendência é agregar um maior número de cores aos processadores. Os cores não serão mais velozes do que os anteriores, porém o aumento da quantidade deles dentro de um único pro-

cessador aumentará as possibilidades de melhorar o tempo de processamento. Para isso, é necessário reescrever o código fonte sequencial transformando-o em paralelo [Bois, 2008], [Milani et al., 2007]. Estas máquinas utilizam memória compartilhada, o que facilita a utilização de threads para a comunicação entre os programas que serão executados. Essas threads podem ser distribuídas através dos cores aumentando o poder de processamento, pois threads são seções de um programa que podem ser processados independente e simultaneamente. Sendo assim, uma tarefa maior será quebrada em tarefas menores diminuindo o tempo total de processamento [Aylward et al., 2007].

3.3 Análise de Desempenho

Um dos principais motivos para a aplicação de técnicas de computação de alto desempenho é a redução do tempo de execução de um programa. Essa métrica apresenta-se como o principal critério na análise de desempenho, embora não seja a única. Entre outros, podem ser citados o aproveitamento da hierarquia de memória (cache, principal e secundária) e o ganho obtido com o uso de vários processadores (multiprocessamento).

O tempo de execução pode ser medido pelo sistema operacional através das rotinas de *system timer* [Cunha et al., 2001]. Essas rotinas são divididas de acordo o tempo gasto pelo programa:

- *User*: tempo gasto ao processar instruções do programa.
- *System*: tempo gasto pelo sistema operacional durante a execução do programa.
- *Elapsed* ou *wall clock* (tempo de parede) ou *real*: tempo total de execução do programa.

No sistema operacional Linux, essas rotinas podem ser observadas através do comando *time*. Exemplo:

```
real    0m24.601s
user    0m23.297s
sys     0m2.552s
```

No entanto, ao executar um programa que utilize mais de uma *thread*, o tempo de usuário é mostrado como se fosse maior do que o tempo total de execução:

```
real    0m16.622s
user    0m25.186s
sys     0m3.524s
```

Isso acontece porque o tempo de usuário é calculado pela soma de um membro da biblioteca *times.h* chamado tms_utime, que retorna o tempo total de um processo. Como o programa é executado em várias threads, o tempo de cada um é somado para gerar o tempo total. Assim, o tempo de usuário acaba sendo maior do que o tempo total de execução.

Outra limitação do comando *time* é permitir somente a medição do tempo total. Caso seja necessário medir o tempo de um determinado trecho do programa (por exemplo, uma rotina), uma alternativa é a utilização de *code timers*. A desvantagem desse método é que o tempo de execução dessas instruções são acrescentadas ao tempo total do programa.

Para realizar uma análise mais profunda de desempenho, são utilizadas ferramentas próprias, chamadas ferramentas de perfilamento (*profilers*).

3.4 Profilers

A característica mais importante dos *profilers* é determinar o trecho do programa que consome mais tempo. Além disso, é possível saber o número de chamadas, a quantidade de ciclos de *clock* e o tempo que cada rotina consome.

Existem várias ferramentas que realizam essa tarefa, dentre elas PAPI, Valgrind, Intel VTune etc. Neste trabalho, foram utilizados o Gprof e o Open|SpeedShop.

3.4.1 Gprof

O GNU gprof é uma ferramenta de perfilamento que fornece informações básicas sobre o desempenho de um programa. Abaixo, um exemplo do relatório de saída do gprof com as informações da ferramenta sRNAscanner:

Time	Cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
94.64	54.25	54.25	132992	407.90		another(int, char)
1.03	56.43	0.59	18473	31.94	31.94	fun(char, float, float)
0.84	56.91	0.48	4314	111.28	111.28	range123(char)
0.31	57.09	0.18	1			slide_m1(void*)
0.21	57.21	0.12	1			slide_m2(void*)
0.12	57.28	0.07	1			p_seq()
0.02	57.29	0.01	1			linear_genome(char*)
0.02	57.30	0.01	1			unique(char, char*, float)
0.02	57.31	0.01	1			some123(char)
0.02	57.32	0.01	1			masking1(char)
0.02	57.33	0.01	1			diff_c_cc(char, int, int)
0.00	57.33	0.00	1	0.00	0.00	_GLOBAL__sub_I_m1
0.00	57.33	0.00	1	0.00	0.00	log_score_m1(char*)
0.00	57.33	0.00	1	0.00	0.00	log_score_m2(char*)
0.00	57.33	0.00	1	0.00	0.00	log_score_m3(char*)

Figura 3.1: Relatório de saída do programa original utilizando o profiler gprof.

Cada coluna apresenta as seguintes informações:

- `% time`: porcentagem do tempo de execução total da rotina em relação ao tempo total do programa.
- `cumulative seconds`: soma do tempo de execução total das rotinas anteriores com o tempo da própria rotina.
- `self seconds`: tempo de execução total de cada rotina.
- `calls`: número de chamadas a uma determinada rotina.
- `self ms/call`: tempo gasto a cada chamada da rotina.
- `total ms/call`: tempo gasto a cada chamada da rotina incluindo as rotinas descendentes.
- `name`: nome das funções.

3.4.2 Open|SpeedShop

O Open|SpeedShop [Galarowicz, 2011] é uma ferramenta de análise de desempenho distribuída pela Krell Institute[1] que utiliza a biblioteca PAPI (*Performance Application Programming Interface*) [London et al., 2001].

Entre suas características, podem ser ser citadas:

- Informações sobre tempo inclusivo (tempo gasto pela função e também pelas funções chamadas dentro dela) e exclusivo (somente o tempo gasto pela função);
- Suporte a interface gráfica;
- Exibição dos caminhos de chamada (*call paths*) de função;
- Captura dos eventos de entrada e saída em Pthreads.

A Figura 3.2 mostra um exemplo da interface gráfica do Open|SpeedShop.

3.5 Computação de Alto Desempenho

Um dos principais motivos para a aplicação de técnicas de computação de alto desempenho é a redução do tempo de execução de um programa. Essa métrica apresenta-se como o principal critério na análise de desempenho, embora não seja a única. Entre outros, podem ser citados o aproveitamento da hierarquia de memória (cache, principal e secundária) e o ganho obtido com o uso de vários processadores (multiprocessamento).

Existem várias maneiras de otimizar um programa. No entanto, as abordagens utilizadas em computação de alto desempenho são as seguintes:

- Utilização de bibliotecas
- Aplicação de técnicas de otimização manual
- Implementação de multiprocessamento

[1] http://www.krellinst.org/

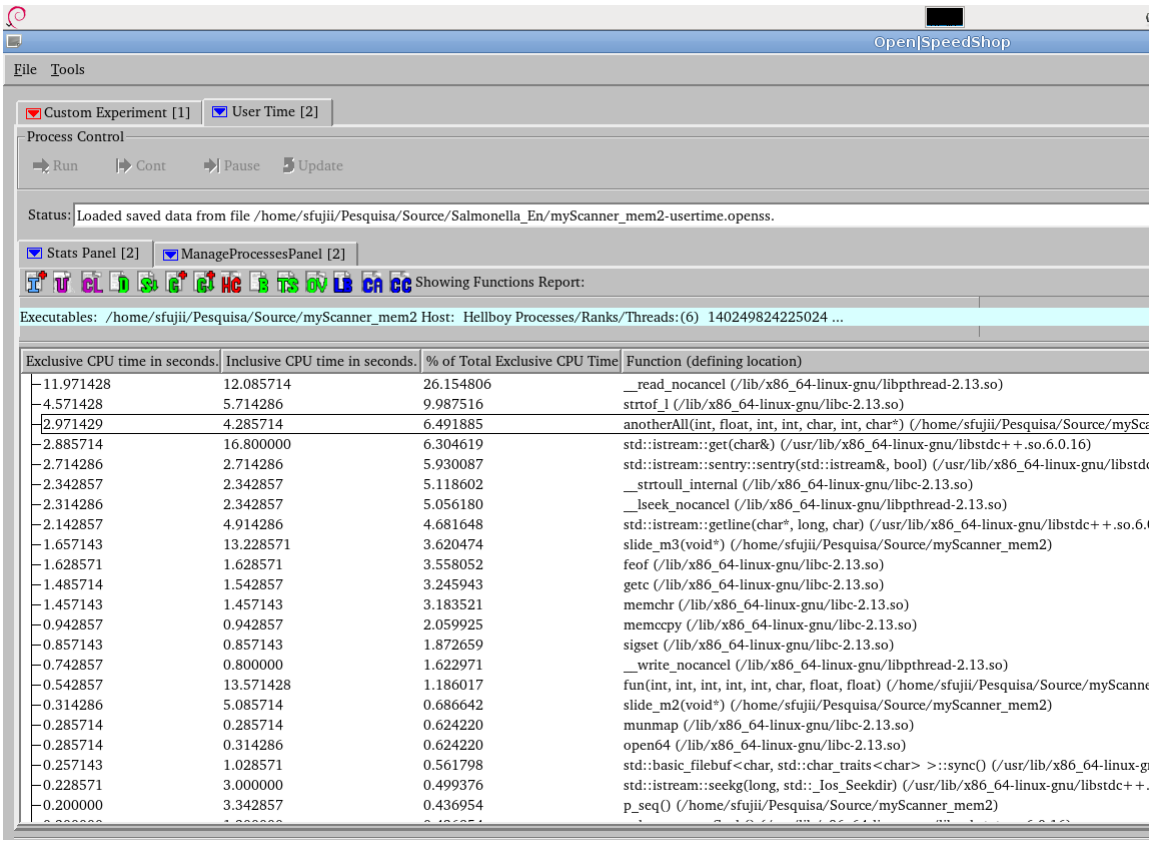

Figura 3.2: Exemplo de relatório de saída do Open|SpeedShop utilizando a interface gráfica.

3.6 Bibliotecas

O objetivo da utilização de bibliotecas é o reaproveitamento de códigos previamente implementados, fazendo uso de conhecimentos com resultados comprovados em aplicações anteriores. Geralmente, cada linguagem apresenta bibliotecas padrão pré-embutidas em sua distribuição. Na linguagem C, tais bibliotecas são utilizadas na maioria dos programas. Alguns exemplos são:

- *stdio.h*: contém definições de entrada e saída padrão, como funções de abertura e fechamento de arquivos e a função *printf* (impressão na tela), assim como suas derivações (*sprintf*, *fprintf*).

- *stdlib.h*: contém definições de funções da biblioteca padrão, como *exit* (término do programa), *atoi* (conversão de texto para o tipo inteiro), *malloc*, *calloc* e *free* (manipulação de memória), *rand* (geração de números aleatórios), *qsort* (algo-

ritmo de ordenação) e *sleep* (suspensão do programa por um período de tempo).

- *string.h*: contém definições de manipulação de *strings* (vetor de caracteres), como *strcmp* (comparação de *strings*), *strlen* (extensão de uma *string*), *strcpy* (cópia de uma *string*), *strcat* (concatenação de *string*), *strstr* (*substring* em uma *string*).

- *math.h*: contém definições de operações matemáticas de trigonometria (seno, cosseno, tangente, arco-seno, arco-cosseno, arco-tangente, seno hiperbólico, cosseno hiperbólico, tangente hiperbólica etc), arredondamentos (*ceil* (arredondamento para cima), *floor* (arredondamento para baixo)), logaritmos, raízes e constantes.

3.6.1 STL - *Standard Template Lybrary*

A *Standard Template Library* (STL) é uma biblioteca de classes da linguagem C++ que descreve quatro componentes: algoritmos, *containers*, funções-objeto e iteradores.

- *Container*: Os *containers* (repositórios) são objetos que armazenam coleções de outros objetos. Existem dois tipos de containers: sequenciais e associativos.

  Os containers sequenciais implementam estruturas de dados comumente utilizados, como vetores, filas, pilhas e listas encadeadas.

  Os containers associativos são tipos abstratos de dados compostos por pares de coleçõs do tipo (chave, valor). Ou seja, para cada chave há um valor associado.

  Um dos containers mais utilizados é o *vector*. Ele funciona da mesma forma que um vetor (matriz unidimensional), porém possui capacidade de armazenamento dinâmica. Em um estudo de caso de [DEITEL and DEITEL, 2006], foi observado o comportamento da alocação de memória. Ao inserir elementos em um *vector*, foram utilizadas as funções *size* (retorna o número de elementos) e *capacity* (retorna a capacidade). Ao adicionar 1 elemento, size e capacity retornaram o valor 1. Com 2 elementos, ambos retornaram o valor 2. Já com 3 elementos, *size*

retornou 3 e *capacity* 4. Portanto, torna-se possível alocar mais um elemento sem que haja a necessidade de alocação de memória. Ao alocar 5 elementos, a capacidade de *vector* dobrou para 8. Assim, observou-se que, ao alocar toda a memória reservada a essa estrutura, o container dobra sua capacidade. Isso evita o *overhead* de alocação de memória a cada elemento adicionado. Por outro lado, ao adicionar um número considerável de elementos, por exemplo 1000000, vector irá alocar 2000000 elementos ao preencher a capacidade anterior. Da mesma forma que o vetor tradicional, *vector* aloca memória de forma contígua. Assim, caso não haja espaço contíguo disponível na memória, os elementos são copiados para a próxima extensão de memória livre. Um programa que utilize muita memória terá um grande *overhead* de cópia e alocação. Em casos extremos, ao não encontrar espaço livre contíguo, o programa pode sofrer falha de segmentação e abortar sua execução. Essa falha ocorre quando um programa tenta ler ou gravar um endereço na memória que está reservado a outro programa. Uma solução é a utilização de containers de alocação dinâmica de memória, como por exemplo *list* (lista encadeada da STL). Com *list*, cada elemento inserido não necessita de espaço contíguo na memória, pois eles são interligados através de seus endereços. No entanto, a complexidade computacional de acesso é na ordem de O(n), enquanto o acesso aleatório permite que *vector* apresente complexidade O(1).

- Iterador: É um objeto que permite navegar pelos objetos de um container. STL apresenta cinco tipos de iteradores: entrada (leitura de sequência de valores), saída (escrita de sequência de dados), avanço (leitura, escrita e avanço), bidirecional (leitura, escrita, avanço e retrocesso) e aleatório (acesso livre)

- Função-objeto: Apresenta classes que sobrecarregam o operador de função *operator*, sendo utilizado para manter e recuperar informações passadas por outras funções.

- Algoritmos: Apresenta algoritmos para acessar uma série de dados em um ite-

rador ou container, o que inclui operações de pesquisa e ordenação.

3.6.2 POSIX Threads (Pthreads)

Em arquiteturas que utilizam vários núcleos de processamento com memória compartilhada, o paralelismo pode ser implementado através de threads. Thread é definido como um fluxo contínuo de instruções que podem ser executadas ao mesmo tempo [Nichols et al., 1996].

Ao perceberem a vantagem do processamento multi-thread, fabricantes de hardware desenvolveram suas próprias versões de tratamento de threads. Essas versões se diferenciavam entre elas, causando dificuldade no desenvolvimento de programas portáveis. Para aproveitar o seu potencial, foi criada uma interface padronizada de programação de threads chamada POSIX threads ou Pthreads.

Pthreads é um conjunto de tipo e chamadas de procedimentos na linguagem C para a criação de threads em ambiente de memória compartilhada [Nichols et al., 1996].

Para a criação de threads, utiliza-se a função `pthread_create`. O protótipo da função é o seguinte:

```
int pthread_create(pthread_t *thread, const pthread_attr_t
*attr, void *(*start_routine)(void*), void *arg);
```

O primeiro argumento (`pthread_t *thread`) é o identificador de cada thread. O segundo (`const pthread_attr_t *attr`) é utilizado para aplicar um atributo específico de `Pthreads`. `void *(*start_routine)(void*)` é a função que invoca uma thread. Por fim, `void *arg` passa um argumento para a função a ser paralelizada.

3.6.3 Bibliotecas de Alto Desempenho

Bibliotecas de alto desempenho visam otimizar operações matemáticas complexas, como sistemas de equações lineares e transformada rápida de Fourier em aplicações de engenharia. Uma biblioteca largamente utilizada que apresenta subrotinas para

soluções de problemas de álgebra linear é a Linear Algebra Package (LAPACK). Para a solução de problemas de álgebra linear em sistemas de memória distribuída, é utilizada a Scalable Linear Algebra Package (ScaLAPACK). Alternativas comuns a essas bibliotecas são a Basic Linear Algebra Subprograms (BLAS), a Basic Linear Algebra Communication Subprogram (BLACS) e a Intel Math Kernel Library (MKL).

3.7 Técnicas de Otimização Manual

Ao realizar o perfilamento de um programa, obtém-se informações sobre o tempo gasto por suas rotinas e, consequentemente, seus pontos críticos (*hot spots*). Com essa informação, o primeiro passo é investigar esses pontos críticos de modo a diminuir seu custo computacional. Segundo [Dowd and Severance, 1998], esse custo é causado por *clutter*, ou desordem, que é qualquer atributo que contribui para aumentar o tempo de execução sem contribuir com o resultado. Para eliminar o *clutter*, são aplicadas as técnicas de otimização manual. Vários procedimentos contribuem para o aumento do tempo de execução, por exemplo: chamadas de sub-rotina, desvios condicionais em laços de repetição, conversões de tipo, chamadas ao sistema e variáveis não utilizadas.

3.7.1 Eliminação de sub-rotinas

A inclusão de sub-rotinas é essencial para a modularização do código e para a execução de conjuntos de tarefas comumente realizadas. No entanto, o seu uso deve ser feito de maneira que o desempenho do programa não seja afetado, pois uma chamada de sub-rotina inclui o *overhead* de guardar endereços nos registradores, além da leitura e da manipulação de argumentos [Dowd and Severance, 1998]. Assim, uma sub-rotina que seja chamada várias vezes para a realização de uma simples tarefa será prejudicial para o desempenho.

O código abaixo mostra o exemplo de uma função que auxilia na soma de matrizes:

```
1 int soma(int M1[LINHA][COLUNA], int M2[LINHA][COLUNA], int i, int j) {
2   int resultado;
3   resultado = M1[i][j] + M2[i][j];
4   return resultado;
5 }
6 ...
7 for(int i=0; i<LINHA; i++)
8   for(int j=0; j<COLUNA; j++)
9     matriz[i][j] = soma(M1,M2,i,j);
```

Para realizar o cálculo, a função `soma` é chamada para fazer a soma dos elementos correspondentes aos índices `i` e `j`. O resultado da função é atribuído na função `matriz` com os índices correspondentes. No entanto, o número de chamadas é correspondente ao número de elementos da matriz. Caso a matriz seja muito grande, o tempo de execução irá aumentar devido ao *overhead* de chamadas de sub-rotina, discutido anteriormente.

Neste caso, a solução é realizar o cálculo diretamente, sem o uso da função:

```
1 for(int i=0; i<LINHA; i++)
2   for(int j=0; j<COLUNA; j++)
3     matriz[i][j] = M1[i][j] + M2[i][j];
```

3.7.2 Macro

Uma alternativa para a utilização de pequenas funções é o uso de macros, que são pequenos procedimentos substituídos em tempo de compilação. Ao contrário de sub-rotinas, que são incluídas durante a etapa de ligação, as macros são processadas na primeira passagem do compilador no programa [Dowd and Severance, 1998]. Quando o compilador encontra um padrão que coincide com a macro definida, a porção de código é substituída.

Na linguagem C, as macros são definidas através do comando `#define`, conforme o exemplo a seguir:

```
1   #define calcula_media(x,y) (x+y)/2
```

```
2  main()
3  {
4    float a=10, b=20;
5    float resultado;
6    media = calcula_media(a,b);
7    printf(``%fz\n'',media);
8  }
```

A macro `calcula_media` é definida na linha 1, consistindo de um simples cálculo de média aritmética que recebe dois argumentos. No momento em que é invocada (linha 6), o compilador subtitui o padrão da macro pela sua definição, resultando no seguinte comando:

```
media = (a+b)/2;
```

3.7.3 *Inlining*

Inlining consiste em substituir a chamada de uma função pelo código da função em si. Geralmente, é utilizada quando o tamanho do código é um pouco grande para ser definido em uma macro [Wadleigh and Crawford, 2000]. Essa técnica pode melhorar o desempenho de várias formas:

- Eliminando a chamada à subrotina, o *overhead* da chamada de função (armazenamento e restauração de registradores) é eliminado.
- Trechos de código apresentam maior probabilidade de estarem sujeitas a otimizações do compilador.

No entanto, essa técnica aumenta o tamanho do código-fonte do programa, o que pode torná-lo menos compreensível e com o tempo de compilação maior. Além disso, nem todos os tipos de subrotinas são sujeitas a *inlining*, como por exemplo:

- Subrotinas recursivas.
- Subrotinas que utilizam variáveis locais estáticas (static em C e C++).
- Subrotinas com número de argumentos indefinido.

3.7.4 Otimização de laços de repetição

A maior parte do processamento de um programa é consumida em laços de repetição. Portanto, é importante que somente as operações necessárias estejam presentes nelas. Segundo Dowd e Severance (1998), as otimizações podem ser feitas nas seguintes ocasiões:

- Desvio condicional em laços de repetição
- Desenrolar de laços

3.7.4.1 Desvio condicional em laços de repetição

Uma operação que deve ser evitada em um laço de repetição é o desvio condicional (comando `if`) devido às instruções extras que o comando consome. O desvio condicional pode aparecer nas seguintes ocasiões:

- *Desvio condicional invariante*: o resultado do teste é sempre o mesmo, portanto, o desvio condicional pode ser descartado;
- *Desvio condicional dependente do índice*: o resultado do teste de desvio é conhecida de antemão, variando de acordo com o índice utilizado no laço de repetição. Assim, pode-se fazer um laço de repetição para cada caso, evitando o teste;
- *Desvio condicional independente do laço de repetição*: o teste não depende nem do valor testado e nem do índice do iterador. A otimização pode ser feita com desenrolar de laço ou multiprocessamento.
- *Desvio condicional dependente do laço de repetição*: o teste depende de um valor que muda a cada iteração do laço. Neste caso, não há como otimizar.

3.7.4.2 Desenrolar de laços

A técnica de desenrolar de laços consiste em realizar as iterações de um laço manualmente, economizando o processo de incremento do índice e a alteração do endereço do apontador de pilha a cada iteração. Um exemplo de soma de duas matrizes 4x4 é mostrado a seguir:

```
1 for (i=0; i<4; i++)
2   for(j=0; j<4; j++)
3     m(i,j) = m1(i,j) + m2(i,j)
```

No código mostrado, é feita a soma da matriz através de dois laços de repetição, um para as linhas e outro para as colunas. No código a seguir, é aplicado o processo de desenrolar de laços:

```
1 for (i=0; i<4; i++) {
2   m(i,0) = m1(i,0) + m2(i,0)
3   m(i,1) = m1(i,1) + m2(i,1)
4   m(i,2) = m1(i,2) + m2(i,2)
5   m(i,3) = m1(i,3) + m2(i,3)
6 }
```

Percebe-se que o laço de repetição referente às colunas foi retirado, realizando o comando de cada iteração manualmente. Desse modo, o custo de um laço de repetição é eliminado. O laço referente às linhas também pode ser desenrolado, resultando em 16 comandos de atribuição.

Sendo esta uma das técnicas mais básicas de otimização, ela está presente na maioria dos compiladores atuais [Dowd and Severance, 1998].

3.7.5 Conversão de tipos

A escolha do tipo correto para um dado é essencial para obter alto desempenho. Caso seja escolhido um tipo inadequado, será preciso converter o elemento para outro tipo. A cada conversão, o programa sofre uma penalidade de desempenho. Se essa operação estiver presente em um local de grande atividade (por exemplo, um laço de

repetição), a penalidade será significativa [Dowd and Severance, 1998].

Na linguagem C, a conversão de tipos é feita implicitamente em uma operação binária. A conversão é feita do tipo mais simples para o mais complexo, seguindo a seguinte regra:

`char < int < long < float < double`

O sinal < indica que o tipo da esquerda possui procedência menor do que o tipo da direita.

No entanto, quando deseja-se realizar uma conversão fora desse padrão, é necessário fazer uma conversão explícita, chamada *cast*. A conversão é feita colocando o tipo entre parênteses antes da variável a ser convertida. Um exemplo é mostrado a seguir:

```
1 int a = 3, b = 2;
2 printf(''%f\n´´, (float)a/b);
```

Na linha 2, o resultado da divisão das variáveis `a` e `b` do tipo `int` são convertidos para o tipo `float`. Assim, o resultado a ser mostrado na tela é `1.5`. Caso o *cast* fosse omitido, o valor seria truncado para `1`.

Para conversões do tipo `char` para valor numérico, são utilizadas as seguintes funções especiais:

- `atoi`: converte tipo `char` em `int` (inteiro);
- `atof`: converte tipo `char` em `float` (real).

Essa funções são geralmente utilizadas ao manipular dados de arquivo de texto, pois os valores são lidos no formato `char`. Para evitar esse tipo de conversão, opta-se pela utilização de arquivos no formato binário, em que os dados podem ser lidos no mesmo formato em que foram gravados.

3.7.6 Otimizações aritméticas

As otimizações aritméticas visam diminuir a complexidade de operações de modo que o resultado final seja o mesmo [Dowd and Severance, 1998]. Os exemplos mais co-

nhecidos são: redução de esforço, e eliminação de sub-expressões.

3.7.6.1 Redução de esforço

Esta técnica baseia-se na troca de uma operação de alto custo computacional por outra de menor custo, apresentando o mesmo resultado. Alguns exemplos são vistos a seguir:

- *Multiplicação/divisão por uma constante com deslocamento de bits*: o deslocamento de bits possui baixo custo, pois a operação é feita apenas com o deslocamento de bits da variável, sem invocar as operações matemáticas do processador. Exemplo:

  $$\texttt{a/2} \Rightarrow \texttt{a»1}$$

  Ao invés de realizar a divisão por 2, foi feito um deslocamento à direita, que apresenta o mesmo resultado.

- *Exponenciação por multiplicação*: a exponenciação não é uma operação nativa do processador, sendo necessário invocar uma função para calculá-la. A solução é realizar a exponenciação manualmente, ou seja, multiplicando os valores N vezes o valor da potência, como feito no exemplo a seguir:

  $$10^3 \Rightarrow \texttt{10*10*10}$$

- *Divisão de valor real por multiplicação inversa*: A operação de multiplicação utiliza menos recursos do processador do que a divisão. Assim, ao realizar operações com valor discreto, é recomendável substituir a operação de divisão pela multiplicação, mantendo a relação do valor discreto com o resultado do cálculo. Exemplo:

  $$\texttt{a/2} \Rightarrow \texttt{a*0,5}$$

  No exemplo, a divisão por 2 foi substituída pela multiplicação por $0,5$, que resulta no mesmo valor.

3.7.6.2 Eliminação de sub-expressões

Uma das maneiras de otimizar expressões aritméticas é evitar o cálculo desnecessário isolando expressões invariantes em sub-expressões. Sub-expressões são partes de uma expressão. Por exemplo, $A + B$ é uma sub-expressão de $C * (A + B)$ [Cunha et al., 2001]. A seguir, um exemplo de laço de repetição com expressão aritmética:

```
1 for{int i=0; i<N, i++}
2   m[i] = m[i]/sqrt(x+y);
```

No exemplo citado, percebe-se que o cálculo de raiz quadrada com a função `sqrt` é feita a cada iteração do laço. Porém, é possível identificar que a sub-expressão `1/sqrt(x+y)` é invariante, ou seja, será a mesma em todas as iterações. Assim, ela pode ser isolada e colocada para fora do laço de repetição, resultando no seguinte código:

```
1 temp = 1/sqrt(x+y);
2 for{int i=0; i<N, i++}
3   m[i] = m[i]*temp;
```

A multiplicação da variável `temp` (linha 3), que contém o resultado da sub-expressão `1/sqrt(x+y)`, é mais rápida do que o cálculo da raiz quadrada a cada iteração.

CAPÍTULO 4

SRNASCANNER

O sRNAscanner [Sridhar et al., 2010] é uma ferramenta que propõe um método computacional para identificar unidades transcricionais (UT) de sRNA intergênico em genomas bacterianos completamente sequenciados. Para isso, utiliza sequência de genomas bacterianos completos e tabelas de codificação de proteínas no formato fasta. O sRNAscanner consiste de algoritmos que realizam as seguintes funções:

- Contrução de matrizes de peso posicional (MPP) de sinais transcricionais específicos de sRNA;
- Busca em genomas completamente sequenciados utilizando MPPs para identificar promotores intergênicos “órfãos” e a localização dos terminadores;
- Integração baseada em coordenadas de sinais de promotores e terminadores para definir UTs intergênicas candidatas;
- Seleção de UTs candidatas com base nos valores da soma acumulativa de pontuação (SAP) que estiverem acima de um valor de corte.

Para isso, baseia-se em matriz de peso posicional (MPP), ou *positional weight matrices* (PWM), que contêm sequências de DNA treinadas para identificar regiões de sRNA intergênicas. As matrizes são construídas a partir de um conjunto de sequências baseado no método de Hertz e Storno (1999) através da ferramenta *PWM_create*, que acompanha o programa principal.

4.1 Matriz de Peso Posicional (MPP)

O processo de alinhamento é utilizado para encontrar um padrão de sequência compartilhado por um conjunto de sequências relacionadas. A parte mais importante

nesse processo é o método utilizado para modelar o alinhamento, de modo que as coleções de sequências possam ser descritas concisamente, ao invés de simplesmente listar todas as ocorrências. O método mais simples para descrever uma sequência é a sequência consenso, ou sequência conservada. Essa sequência contém as letras (por exemplo, bases para DNA ou aminoácidos para proteína) altamente conservadas em cada posição do alinhamento. No entanto, a maioria dos alinhamentos não possui uma sequência limitada a uma única base por letra em uma determinada posição. Em algumas posições do alinhamento, qualquer letra pode ser considerada aceitável, sendo que algumas letras podem ocorrer com muito mais frequência do que outras. Assim, uma possível abordagem é a matriz de alinhamento, em que o número de ocorrências para cada letra é listado para cada posição, como mostra a Figura 4.1.

	A	**A**	**T**	**T**	**G**	**A**
	A	**G**	**G**	**T**	**C**	**C**
	A	**G**	**G**	**A**	**T**	**G**
	A	**G**	**G**	**C**	**G**	**T**
	1	**2**	**3**	**4**	**5**	**6**
A	4	1	0	1	0	1
C	0	0	0	1	1	1
G	0	3	3	0	2	1
T	0	0	1	2	1	1
	A	**G**	**G**	**T**	**G**	**N**

Figura 4.1: Exemplo de uma matriz de alinhamento. A matriz contém o número de ocorrências $n_{i,j}$ para cada letra i que ocorre na posição j do alinhamento. Abaixo, a sequência consenso é extraída a partir do número de ocorrência de nucleotídeos das quatro sequências alinhadas. N indica que não importa o nucleotídeo, pois nessa posição não há letra dominante. Fonte: [Hertz and Stormo, 1999]

A partir da matriz de alinhamento, é possível extrair a matriz de peso posicional (MPP). Nessa abordagem, os elementos da matriz representam os pesos utilizados para calcular o quanto uma determinada sequência se aproxima do modelo descrito por essa matriz. A MPP é derivada da matriz de alinhamento utilizando a seguinte

fórmula:

$$\ln \frac{(n_{ij} + pi)/(N+1)}{p_i} \approx \ln \frac{f_{ij}}{p_i}$$

Nessa fórmula, N é o valor total de sequências de entrada e p_i é a probabilidade da letra i ocorrer na posição j de uma sequência de entrada. Assim, para um sistema com quatro componentes (A, C, G e T) a frequência esperada é de 0.25 para cada um dos nucleotídeos. $f_{i,j} = n_{i,j}/N$ é a frequência com que a letra i aparece na posição j.

A MPP gerada a partir da matriz de alinhamento e que apresenta os pesos de cada letra em determinada posição é mostrada na Figura 4.2.

	1	**2**	**3**	**4**	**5**	**6**
A	1.2	0	-1.6	0	-1.6	0
C	-1.6	-1.6	-1.6	0	0	0
G	-1.6	.96	.96	-1.6	.59	0
T	-1.6	-1.6	0	.59	0	0
	A	**G**	**G**	**T**	**G**	**C**

Figura 4.2: Matriz de peso posicional derivada da matriz de alinhamento mostrada na Figura 4.1. Fonte: [Hertz and Stormo, 1999]

Os pesos para cada ocorrência são somados para fornecer a pontuação total da sequência de entrada. Portanto, quanto maior a pontuação, maior é a probabilidade da sequência de entrada apresentar relação com a sequência consenso.

No sRNAscanner, as MPP são construídas com a ferramenta *PWM_create*. Com ele, é possível gerar até três matrizes a cada execução. As informações de entrada são lidas através de um arquivo no formato *fasta* contendo as sequências de DNA selecionadas.

4.2 PWM_create

A primeira etapa do processo é a criação das matrizes de peso posicional (MPP) com o programa PWM_create. Essas matrizes descrevem conjuntos selecionados de sequências de DNA com os seus respectivos pesos para facilitar a eficiente identificação de sRNA intergênico. As MPPs são definidas por sinais transcricionais específicos, que são as regiões promotoras -35 e -10 e os sinais de terminador rho-independente.

O programa PWM_create constrói as três MPPs (regiões promotoras -35 e -10 e os sinais de terminador rho-independente) utilizando a metodologia definida por Hertz e Storno. As sequências de DNA escolhidas são colocadas em um arquivo no formato fasta e lidas pelo programa. O formato do arquivo é apresentado no exemplo abaixo:

```
>EMBOSS_001
CGTCAC
>EMBOSS_002
CACCTG
>EMBOSS_003
TTTGAC
>EMBOSS_004
ACTAAA
>EMBOSS_005
GTGGGC
>EMBOSS_006
GGACAA
>EMBOSS_007
CTCTCA
>EMBOSS_008
TTCGAC
>EMBOSS_009
TTTCCA
>EMBOSS_010
TTGAGT
```

A matriz MMP é mostrada abaixo:

```
A| -0.788457| -0.788457| -0.788457| -0.200671| 0.646627| 0.435318|
C| 0.167054| -0.788457| 0.167054| 0.435318| -0.200671| 0.435318|
G| -0.200671| -0.200671| -0.200671| 0.167054| -0.200671| -0.788457|
T| 0.435318| 0.820981| 0.435318| -0.788457| -0.788457| -0.788457|
```

4.3 Funcionamento

O programa sRNAscanner realiza a busca em genomas completos através de deslocamento genômico para identificar as coordenadas de início e fim de unidades transcricionais intergênicas candidatas. Esse processo é feito através da comparação dessas predições com a tabela de codificação de proteínas correspondente.

As unidades transcricionais (UT) candidatas são previstas utilizando a soma acumulativa de pontuação (SAP), selecionando valores que estiverem acima de um determinado valor de corte. A SAP é determinada através da soma de três matrizes específicas de pontuação acumulativa para cada UT.

As matrizes são utilizadas individualmente para examinar completamente o genoma através do deslocamento genômico. Esse deslocamento funciona da seguinte forma: é criada uma janela de nucleotídeos da mesma largura da MPP correspondente. Para as MPPs das regiões promotoras -10 e -35 a largura é de 6 nucleotídeos (Figura 4.3). Para a MPP do terminador, a largura é de 32 nucleotídeos.

	-35						-10					
	T	T	G	A	C	A	T	A	T	A	A	T
T	70	70	20	30	20	20	90	10	60	30	10	100
G	10	0	60	0	10	10	10	0	10	10	20	0
C	10	10	0	0	60	30	0	0	20	0	20	0
A	10	20	20	70	10	40	0	90	10	60	50	0

Figura 4.3: MPPs das regiões promotoras -35 e -10. Fonte: [Sridhar et al., 2010]

Essa janela percorre o genoma, avançando um nucleotídeo por vez. A cada iteração, é atribuída a SAP para cada sequência de nucleotídeos. Esses valores são

armazenados em um arquivo para posterior comparação com o valor de corte atribuído a cada MPP. Os valores de corte são declarados no arquivo de parâmetros *Input.data*. A lista de sRNAs utilizados para gerar as MPPs das regiões promotoras é mostrada na Figura 4.4.

-35 box[a]	-10 box[b]	sRNA[c]
TTGTAA	TTCTAT	*csrB*
TTGTCA	TAAGGT	*dsrA*
CTGATC	TATATT	*gadY*
TTGAGC	TATAGT	*gcvB*
TATACT	TATTCT	*micC*
AAAACA	TAGAAT	*micF*
TTATCC	GATAAT	*oxyS*
GTGACA	TATACT	*rnpB*
TCGACG	TACTAT	*rprA*
TTTATT	TATAAT	*rydC*

Figura 4.4: sRNAs da *Escherichia coli* K-12 utilizados para gerar as MPPs das regiões promotoras -35 e -10. a) Sequência da região promotora -35. b) Sequência da região promotora -10. c) sRNAs correspondentes às regiões promotoras. Fonte: [Sridhar et al., 2010]

A Figura 4.5 mostra as sequências de terminadores Rhô-independentes da *Escherichia coli K-12* utilizadas para gerar a MPP da região terminadora.

As MPPs geradas a partir da lista de SRNAs da Figura 4.4 são mostradas na Figura 4.6.

4.4 Parâmetros de entrada

A entrada do programa é formada por um arquivo de parâmetro (*Input.data*) que contém dezesseis campos essenciais para o seu funcionamento. Os parâmetros são descritos a seguir:

1. *no_of_matrices*: Número de matrizes a serem analisadas. Valor padrão: 3.
2. *Name of the Matrix 1*: Nome da MPP da região promotora -10.
3. *Name of the Matrix 2*: Nome da MPP da região promotora -35.
4. *Name of the Matrix 3*: Nome da MPP da região do terminador.

```
>NC_000913.2_1550372_1550402_eco_c0362
TGTTGCGTTTGTCATCAGTTCTAAATGGCGC
>NC_000913.2_3119518_3119548_eco_c0719
GGCCCGCTAGTAATGCGCTACGGGTATTTAA
>NC_000913.2_2940898_2940928_eco_gcvB
ACCGCCTAATTGCGGTGCTTTTTTTTACCTT
>NC_000913.2_2651715_2651745_eco_IS128
AAATCAGGCGGTTTTTTGTTGCTGGTCCGGT
>NC_000913.2_1435229_1435259_eco_micC
CCGAACAGTCGTCCGGGCTTTTTTTTTAGAA
>NC_000913.2_2311171_2311201_eco_micF
ACCGGATGCCTCGCATTCGGTTTTTTTTACC
>NC_000913.2_1768476_1768506_eco_rprA
CCATCTCCCACGATGGGCTTTTTTTTAACAT
>NC_000913.2_1921335_1921365_eco_ryeA
GTGAAAGAACTGAGGCGGTTTTTTTATTGGA
>NC_000913.2_2151348_2151378_eco_ryeC
AGGTTTCCCCCTCCCCCTGGTGTTCTTAGTA
>NC_000913.2_3054988_3055018_eco_rygC
GCCCTCGCTTCGGTGAGGGCTTTACCGTTAC
>NC_000913.2_2689598_2689628_eco_tke1
AAGCTCCCGGAGTTGGGAGCTTATGATAGTG
>2069510-2069540_eco_isrC_IS102
CCACGTTTGTGGGTACCGGCTTTTTTATTCA
>3236105-3236135_eco_sraF
TGTGCTTCGTGCCGCGTGGCAGCCAAATGCA
>3662574-3662604_eco_gadY
ATTTTTTGCTGTTCATGATAAATATCGAATG
>NC_000913.2_2023239_2023269_eco_dsrA
CCCCTCAGGGTCGGGATTTTTTTATTGTGCA
>NC_000913.2_1762690_1762720_eco_rydB
GTCGCACAGACTTAAGGGTTTTCTTATTTCT
>NC_000913.2_1489461_1489491_eco_rydC
TCGGTTTTAGTACAGGCGTTTTCTTTAAATA
>NC_000913.2_2651687_2651717_eco_ryfA
TTTTGGCGGTTTTTTGTTTTTTAAAGGAAAG
>gi|49175990:c1403706-1403676_eco_isrA
GGATGTCGACAGACTCTATTTTTTTATGCAG
>gi|49175990:c3192759-3192729_eco_ryD
TTCGGGAGGGGCTTTCCCGTTTCAGCCCCCT
>c3578579-3578549_sraI
CGCGCGCTGGAAGCCGGGAAAAATGTGCTGG
```

Figura 4.5: Terminadores de transcrição da *Escherichia coli* K-12 utilizados para gerar a MPP da região terminadora. Fonte: [Sridhar et al., 2010]

5. *Which strand needs to be searched for sRNA signals (p=1 or n=2)*: Sentido da fita (positivo ou negativo) a serem pesquisados os sinais de sRNA. Valor padrão: positivo.

6. *Enter the genome file name (*.fna)*: Nome do arquivo do genoma em formato *fasta* (*.fna).

(A) PWM1-training

```
A| -0.788457| -0.200671| -0.200671|  0.969401| -0.788457|  0.435318|
C| -0.788457| -0.788457|   -2.3979|   -2.3979|  0.820981|  0.167054|
G| -0.788457|   -2.3979|  0.820981|   -2.3979| -0.788457| -0.788457|
T|  0.969401|  0.969401| -0.200671|  0.167054| -0.200671| -0.200671|
```

(B) PWM2-training

```
A|   -2.3979|   1.21302| -0.788457|  0.820981|  0.646627|   -2.3979|
C|   -2.3979|   -2.3979| -0.200671|   -2.3979| -0.200671|   -2.3979|
G| -0.788457|   -2.3979| -0.788457| -0.788457| -0.200671|   -2.3979|
T|   1.21302| -0.788457|  0.820981|  0.167054| -0.788457|   1.31568|
```

(C) PWM3-training

```
A|0.127833|-0.893818|-0.257829|-0.893818|-0.893818|-0.257829|-0.526093|
 -0.526093| -0.04652|-0.893818|-0.526093|-0.526093|-0.257829|-0.526093|
 -0.257829|-0.893818|-  1.4816|-0.526093|  -1.4816|-0.526093|-0.526093|
 -0.893818|-0.257829| 0.127833|-0.526093| 0.127833| -0.04652| 0.276253|
 -0.257829| 0.127833| 0.405465|

C| -0.04652| 0.405465| 0.405465| -0.04652| 0.276253| 0.127833| 0.405465|
   -0.04652| -0.04652| -0.04652|-0.257829|-0.257829| 0.127833| 0.127833|
  -0.526093| 0.276253|-0.257829| 0.127833| -3.09104|-0.257829|-0.893818|
    -1.4816|-0.257829|-  1.4816|-0.893818|-0.526093|-0.893818|-0.257829|
   -0.04652| 0.127833|-0.526093|
G|- 0.04652| 0.127833| -0.04652| 0.276253|0.127833|-0.257829|-0.257829|
   0.405465| 0.405465| 0.276253| 0.276253|0.276253| -0.04652|-0.257829|
   0.519875| 0.405465|  0.62253| 0.519875|0.276253|-0.893818|  -1.4816|
  -0.526093|  -1.4816|-0.893818|-0.257829| -1.4816| 0.127833|-0.893818|
   0.276253|-0.257829| -0.04652|
T|-0.04652| -0.04652|-0.257829| 0.276253|0.127833|0.276253|0.127833|
  -0.04652|-0.526093| 0.276253| 0.276253|0.276253|0.127833|0.405465|
  -0.04652|-0.257829| 0.127833|-0.526093|0.879249|0.800778| 1.01983|
   1.01983| 0.800778| 0.800778| 0.800778| 0.71562|0.405465|0.405465|
  -0.04652| -0.04652| -0.04652|
```

Figura 4.6: MPPs construídas utilizando sinais transcricionais anteriormente definidas para dez sRNAs da *Escherichia coli* K-12. a) MPP construída utilizando a região promotora -35. b) MPP construída utilizando a região promotora -10. c) MPP construída utilizando região do terminador rho-independente. Fonte: [Sridhar et al., 2010]

7. *Search the whole genome (g=1) or intergenic (i=2) region*: Região de pesquisa (genoma completo ou região intergênica). Valor padrão: região intergênica.

8. *Enter the cut-off value 1*: Valor de corte 1. Valor padrão: 2.

9. *Enter the cut-off value 2*: Valor de corte 2. Valor padrão: 2.

10. *Enter the cut-off value 3*: Valor de corte 3. Valor padrão: 3.

11. *Enter the ptt file name*: Nome do arquivo de genes (*.ptt).

12. *Enter the spacer range 1 (between [-35] & [-10] promoter boxes)*: Distância entre

os promotores -35 e -10. Valor padrão: 12-18.

13. *Enter the spacer range 2 (sRNA length)*: o promotor -10 e o sinal do terminador. Valor padrão: 40-350.

14. *Do you want individual intergenic hits for further analysis, if yes (1) no (2)* Informações de acerto individual na região intergênica (sim ou não).

15. *Enter the Unique hit value*: Valor de hit único. Valor padrão: 200.

16. *Enter the minimum Cumulative Sum of Score (CSS)*: Valor mínimo de soma de pontuação cumulativa (SPC). Valor padrão: 14.

O valor de hit único identifica possíveis UTs de um conjunto de hits sobrepostos baseados na presença de coordenadas de início próximas mapeadas dentro de uma janela de tamanho definida, em que o valor padrão é fixado em 200pb. O programa seleciona o UT com o valor máximo de SAP para cada conjunto de sobreposição como o único hit representativo para o conjunto.

4.5 Arquivos de saída

Ao executar o programa, a ferramenta gera os seguintes arquivos de saída:

- *intergenic_TUs.out*: candidatos de unidades transcricionais intergênicas identificadas.
- *RNA_position*: coordenadas de começo e fim dos genes de sRNA intergênico previstos.
- *sRNA.txt*: sequência de sRNA extraída em formato *fasta*.
- *coding*: possíveis *small* ORFs.

Além desses arquivos, são gerados outros 27, que são utilizados como armazenamento de dados temporários e também para fins de conferência de dados intermediários ao final da execução.

4.6 Discussão

Em termos biológicos, o sRNAscanner apresenta resultados satisfatórios, conforme o trabalho publicado por [Sridhar et al., 2010]. Já em termos computacionais, a ferramenta demanda um tempo consideravelmente alto. Segundo Sridhar et al. (2010), em uma estação de trabalho com a configuração 2x Quad-Core Intel Xeon 2,5GHz 8GB RAM DDR2, o sRNAscanner necessita de 76m53s180 para procurar sRNAs na bactéria *Salmonella enterica serovar Typhimurium*, que apresenta em torno de 4,9MB de dados. Após breve investigação ao código-fonte do programa, foi encontrado um ambiente propício para a aplicação de técnicas de computação de alto desempenho. Nos próximos capítulos, serão apresentados os métodos utilizados na otimização da ferramenta de estudo de caso e os resultados obtidos.

CAPÍTULO 5

MATERIAIS E MÉTODOS

5.1 Estação de Trabalho

A estação de trabalho utilizada possui a seguinte configuração:

- Processador AMD Phenom II x6 3,2GHz 1090T Black Edition;
- 16GB de memória RAM DDR3
- Sistema operacional Debian 6.0 GNU/Linux codinome *Wheezy*

5.2 Ferramentas

O compilador utilizado foi o GCC (*GNU Compiler Collection*) versão 4.6.2 fornecida pela distribuição GNU/Debian.

A ferramenta utilizada no estudo de caso foi o sRNAscanner[1], seguindo as instruções sugeridas no manual[2], porém com a compilação utilizando a opção "-O3" para otimizar todos os executáveis gerados.

Os tempos médios de execução apresentados nos resultados foram obtidos através de dez execuções do programa para cada situação.

5.2.1 Profilers

Os profilers utilizados foram gprof e Open|SpeedShop.

O GNU gprof *GNU Binutils* versão 2.22 também fornecido pela distribuição Debian.

O Open|SpeedShop foi utilizado com a versão 2.0.1 [3], sendo compilado e

[1] disponível em http://cluster.physics.iisc.ernet.in/sRNAscanner
[2] disponível em http://cluster.physics.iisc.ernet.in/sRNAscanner/Readme.pdf
[3] disponível em http://http://www.openspeedshop.org/wp/

instalado na estação de trabalho.

5.3 Bactéria

A bactéria utilizada foi a *Salmonella enterica serovar Typhimurium* [McClelland et al., 2001]. A escolha dessa bactéria se deve ao fato dela acompanhar a distribuição do software sRNAscanner sendo, assim, uma fonte de comparação confiável em relação aos sRNAs encontrados pela ferramenta.

CAPÍTULO 6

RESULTADOS

6.1 sRNAScanner Original

Os arquivos gerados pelo sRNAscanner são mostrados nas Figuras 6.1 e 6.2. A Figura 6.1 mostra o arquivo de saída `RNA_position`, que contém as coordenadas de início e fim dos sRNAs encontrados pelo sRNAscanner. Outro arquivo de saída (Figura 6.2), contém as coordenadas de início e fim dos sRNAs encontrados pelo sRNAscanner, além dos nucleotídeos correspondentes a cada região. Esses arquivos foram comparados ao final de cada versão e apresentaram exatamente o mesmo resultado.

```
139669      139730
230161      230370
3092798     3092891
3135311     3135497
3392062     3392232
4080957     4081136
```

Figura 6.1: Dados do arquivo de saída `RNA_position`. Contém as coordenadas de início e fim dos sRNAs encontrados pelo sRNAscanner.

Para a análise do desempenho do programa, foi utilizado inicialmente o profiler gprof, que fornece informações sobre o tempo de execução (em segundos) e a porcentagem de cada função. As funções são ordenadas de cima para baixo, de acordo com o tempo de execução. O relatório de saída do sRNAScanner original é mostrado na Figura 6.3.

No relatório da Figura 6.3, observa-se que a função `another` utiliza grande parte (94,64%) do tempo de processamento. Caso houvesse uma redução de 80% no tempo de execução dessa função, a redução no tempo total seria de 75,5%. Com base nessa informação, conclui-se que a função `another` apresenta alto potencial para a aplicação de técnicas de otimização.

```
>sRNA|139669|139730|
GCCGCTCGCGTCGCAAACTGACACTTTATATTTTTCTCGGATTATATTGAGTCCGTTTTAAA
>sRNA|230161|230370|
ATTATTTTTTTATCGAAGTAATAAAAAACAACTCTATTAAAATAAATTAATTGCCACGAGTAAACAAAAAAAA
TGTAATTTAGCATTTTTTGTTTTTTTTAATGAAATTTATACCAAAAATAAATGTTACGCGTCATTGCTATATA
TTCAGTACTGTAATTCTGATCTTTTTTTATTCATTTTGACATGTTTATTGTTTTTTTGGCTCTC
>sRNA|3092798|3092891|
TGATTGATAATTATTAAATTTTCTGGAAATTCTTAAATTAATTGCCCGTCACAAGGCGGAGAATGGTTTTAGT
TAATGATAAAAAATGCGCTTT
>sRNA|3135311|3135497|
GGCGGCACTTCCTGAGCCGGAACGAAAAGTTTTATCGGAATGCGTGTTCTGATGGGCTTTTGGCTTACGGTTG
TGATGTTGTGTTGTTGTGTTTGCAATTGGTCTGCGATTCAGACCACGGTAGCGAGACTACCCTTTTTCACTTC
CTGTACATTTACCCTGTCTGTCCATAGTGATTAATGTAGCA
>sRNA|3392062|3392232|
GTTACGCGCAAAGGGGAGTAACTTCATTGCCGGTCGATCGTCATTACGATGTGTGCAAAACCACATCCGGTCA
CCGGGCAGCCATAAAGGAATGCGTCAGCGTATTCCTTTATTGTTGTAAGTGAGACCTTGCCGAATGGCAAGGT
CTATGCATAAAAGCAACGGCTAATG
>sRNA|4080957|4081136|
AAAAATTTAACAAGCAAAAGTGATTTTTCGATTCGATTTTTAGCAGACTGATATTTACAAAGTTGATTACTCT
CTGTTTTAACAAAAGAAGCGACGTTGTCATGTTGAAACGAAACCGTAAATCCAGAGTTAAATGTAACAACGCT
GAAATGCGTTATACCTGTCGATTTTTAACACAAG
```

Figura 6.2: Dados do arquivo de saída `sRNA.txt`. Contém as coordenadas de início e fim dos sRNAs encontrados pelo sRNAscanner. Abaixo de cada par de coordenadas, os nucleotídeos correspondentes a cada região.

Time	Cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
94.64	54.25	54.25	132992	407.90		another(int, char)
1.03	56.43	0.59	18473	31.94	31.94	fun(char, float, float)
0.84	56.91	0.48	4314	111.28	111.28	range123(char)
0.31	57.09	0.18	1			slide_m1(void*)
0.21	57.21	0.12	1			slide_m2(void*)
0.12	57.28	0.07	1			p_seq()
0.02	57.29	0.01	1			linear_genome(char*)
0.02	57.30	0.01	1			unique(char, char*, float)
0.02	57.31	0.01	1			some123(char)
0.02	57.32	0.01	1			masking1(char)
0.02	57.33	0.01	1			diff_c_cc(char, int, int)
0.00	57.33	0.00	1	0.00	0.00	_GLOBAL__sub_I_m1
0.00	57.33	0.00	1	0.00	0.00	log_score_m1(char*)
0.00	57.33	0.00	1	0.00	0.00	log_score_m2(char*)
0.00	57.33	0.00	1	0.00	0.00	log_score_m3(char*)

Figura 6.3: Relatório de saída do programa original utilizando o profiler gprof.

Em uma segunda etapa de avaliação, o programa foi submetido à análise com o profiler Open|SpeedShop. O relatório gerado é mostrado na Figura 6.4.

Novamente `another` é apontada como a função de maior custo. No entanto, novas informações são fornecidas. A mais relevante é o tempo gasto pela função `strtof_l`, que faz a conversão de dados do tipo `string` para números reais (`float`). O tempo de processamento exclusivo é de 512,17 segundos (42,18% do tempo total), sendo a principal responsável pelo alto tempo de execução.

A função `another` é mostrada abaixo:

Exclusive CPU time in seconds.	Inclusive CPU time in seconds.	% of Total Exclusive CPU Time	Function (defining location)
52.199999	1275.428546	3.848341	another(int, float, int, int, char) (/home/sfujii/Pesquisa/Source/sRNAscanner_original)
572.171417	733.771414	42.182201	strtof_l (/lib/x86_64-linux-gnu/libc-2.13.so)
149.285711	363.514278	11.005793	std::istream::getline(char*, long, char) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.16)
152.999997	152.999997	11.279621	__strtoull_internal (/lib/x86_64-linux-gnu/libc-2.13.so)
120.914283	120.914283	8.914165	memchr (/lib/x86_64-linux-gnu/libc-2.13.so)
104.114284	104.114284	7.675619	sigset (/lib/x86_64-linux-gnu/libc-2.13.so)
65.828570	65.828570	4.853081	memccpy (/lib/x86_64-linux-gnu/libc-2.13.so)
52.142856	52.142856	3.844128	std::istream::sentry::sentry(std::istream&, bool) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6...
34.399999	34.399999	2.536072	strerror_r (/lib/x86_64-linux-gnu/libc-2.13.so)
0.114286	29.057142	0.008425	std::basic_filebuf<char, std::char_traits<char> >::underflow() (/usr/lib/x86_64-linux-gnu/...
0.114286	28.999999	0.008425	std::__basic_file<char>::xsgetn(char*, long) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.1...
28.599999	28.857142	2.108478	__read_nocancel (/lib/x86_64-linux-gnu/libpthread-2.13.so)
2.600000	17.885714	0.191680	std::istream::get(char&) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.16)
1.171429	14.971428	0.086361	fun(int, int, int, int, int, char, float, float) (/home/sfujii/Pesquisa/Source/sRNAscanner_origi...
1.571429	12.228571	0.115850	slide_m3(void*) (/home/sfujii/Pesquisa/Source/sRNAscanner_original)
0.257143	6.514286	0.018957	slide_m2(void*) (/home/sfujii/Pesquisa/Source/sRNAscanner_original)
4.714286	4.714286	0.347551	strtod (/lib/x86_64-linux-gnu/libc-2.13.so)
4.114286	4.114286	0.303318	strtoq (/lib/x86_64-linux-gnu/libc-2.13.so)
0.057143	3.942857	0.004213	slide_m1(void*) (/home/sfujii/Pesquisa/Source/sRNAscanner_original)
0.142857	3.428571	0.010532	p_seq() (/home/sfujii/Pesquisa/Source/sRNAscanner_original)
0.028571	2.485714	0.002106	std::istream::seekg(long, std::_Ios_Seekdir) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.16)

Figura 6.4: Relatório de saída do programa original através do Open|SpeedShop.

```
1  void another(int xx,float y,int a,int b,char flag) {
2        int yy,c;
3        float z;
4        char array[10];
5        ofstream f_final;
6        fstream thresh2;
7        f_final.open("temp/final_c_cc.txt",ios::app);
8        ifstream f_cc;
9        f_cc.open("temp/out_cc.txt",ios::in);
10       thresh2.open("temp/threshold2.txt",ios::in);
11       while ((!f_cc.eof())&&(!thresh2.eof())) {
12         f_cc.getline(array,10);
13         yy=atoi(array);
14         thresh2.getline(array,10);
15         z=atof(array);
16         if (yy==0)
17         break;
18         if(flag==’p’)
19         c=yy-xx;
20         else if(flag==’n’)
21           c=xx-yy;
22         if(c>b)break;
23         if((c>=a)&&(c<=b)) {
24            if(flag==’n’) {
```

```
25                f_final<<(xx+r1)<<"|"<<c<<"|"<<(yy-r2+1)<<"|"<<y<<"|"<<z
                      <<"|c"<<"\n";
26                f_final << flush;
27             }
28             else if (flag=='p') {
29                f_final<<(xx-r1)<<"|"<<c<<"|"<<(yy+r2)<<"|"<<y<<"|"<<z
                      <<"|"<<"\n";
30                f_final<<flush;
31             }
32          }
33       }
34 }
```

No código apresentado, há duas conversões de tipo. A primeira é exibida na linha 13:

```
yy = atoi (array);
```

A segunda é observada na linha 15:

```
z = atof (array);
```

A operação da linha 13 (`atoi`) converte uma variável do tipo literal para inteiro e `atof`, na linha 15, converte uma variável literal para números reais.

Outra função que utiliza grande parte do tempo é a `getline`, que faz a leitura de uma linha em um fluxo (*stream*) de arquivo. Na função `another`, é observado o uso de `getline` nas linhas 12 e 14, ou seja, antes das conversões de tipo são feitas leituras nos arquivos `out_cc.txt` e `threshold2.txt` com os seguintes comandos:

```
f_cc.getline (array,10); thresh2.getline(array,10);
```

No relatório de saída do gprof, foi visto que a função `another` representa 94,64% do tempo total de execução. Outra informação relevante é o número de chamadas, em que a função é invocada 132992 vezes. Ou seja, operações de alto custo computacional são utilizadas em uma sub-rotina de uso intenso. Ao analisar a função, percebe-se que as conversões estão localizadas em um laço de repetição, em que a condição de parada é o final de arquivo `out_cc.txt` e `thresh2.txt`. Considerando a bactéria utilizada (*Salmonella enterica serovar Typhimurium*), os arquivos

contêm 85770 linhas, que neste caso representam o número de iterações. Multiplicando o número de chamadas pelo número de iterações do laço, chega-se ao número de 11 bilhões de vezes em que cada operação é realizada. Considerando as quatro operações (duas de conversão de tipos e duas de leitura de arquivo), o número aumenta para 44 bilhões.

O tempo total de execução do programa original com a bactéria *Salmonella enterica serovar Typhimurium* e com o arquivo de argumentos *Input.data* padrão mostrado na seção 4.4 apresentou a média de 23m04s523, ou seja, 1384s523.

6.2 sRNAScanner Improved 1a. versão

A hierarquia de memória define que a memória mais rápida deve ser utilizada sempre que possível [Cunha et al., 2001]. Isso significa que o uso da memória secundária não é aconselhável como armazenamento temporário de arquivos. Neste caso, os valores de threshold e das coordenadas dos promotores da região -10 foram armazenados nos arquivos *threshold2.txt* e *out_cc.txt*, respectivamente, sendo lidas no ponto de maior acesso do programa. De acordo com Cunha et al. (2001), a utilização da memória secundária não é recomendável nessa situação. Desse modo, foram feitas alterações para que os dados fossem armazenados na memória principal, eliminando o alto custo de acesso ao disco.

Nesta primeira versão, foi necessário alterar a estrutura de dados em vários trechos do programa. Os dados dos arquivos *threshold2.txt* e *out_cc.txt* foram armazenados em uma matriz unidimensional utilizando o componente *vector*, da biblioteca STL, como segue:

```
1 vector<int> v_xcc;
2 vector<float v_thresh2;
3 ...
4
5 void *slide_m2(void *arg)
6 ...
```

```
7     v_xcc.push_back(bnum_local);
8     v_thresh2.push_back(sum);
9 ...
```

Os valores das coordenadas de promotores e threshold da região -10 foram armazenados nos containers *vector* `v_thresh2` e `v_xcc`, respectivamente. Dessa forma, o processo de leitura não necessita do acesso ao disco, eliminando a função `getline` e, consequentemente, o *overhead* de leitura de arquivo e a conversão de tipo.

Abaixo o trecho que contém a leitura dos dados através do container *vector* na função `another`:

```
1 for (int i=0; i < x_cc_numoflines; i++)
2 {
3     int yy;
4     float z;
5     ...
6     yy=v_xcc[k];
7     z=v_thresh2[k];
8     ...
9 }
```

Com as primeiras alterações na função `another` o tempo médio de execução caiu para 44s.374. Para medir o ganho de performance foi utilizado o cálculo:

$$speedup = \frac{tempo\_programa\_original}{tempo\_programa\_otimizado}$$

Aplicando a fórmula com os valores obtidos (em segundos):

$$speedup\_sRNAScanner = \frac{1384,523}{44,374} = 31$$

Assim, o speedup da primeira versão do sRNAScanner Improved foi de 31 vezes.

6.3 sRNAScanner Improved 2.a versão

Aplicando otimizações manuais (eliminação de conversão de tipos e acesso a disco) na função another, obteve-se um ganho significativo de desempenho. No entanto, devido à natureza da função otimizada, que apresenta características favoráveis para a divisão de tarefas, foram aplicadas técnicas de processamento paralelo. Apesar da função `another` consumir a maioria dos recursos, a paralelização dessa função não é a abordagem ideal, pois muitas chamadas são realizadas. Analisando a estrutura, é conveniente paralelizar a função que a chama, que neste caso é a `diff_c_cc`. Desse modo, cada chamada dessa função paralelizada realiza as chamadas subsequentes da função `another`. A função `diff_c_cc` foi alterada de modo que o seu trabalho pudesse ser dividido em vários processos. A biblioteca `Pthreads` foi escolhida para realizar o multiprocessamento.

A função `diff_c_cc` original, mostrada a seguir, é executada serialmente:

```
1 void diff_c_cc(char flag,int a,int b) {
2   cout <<"Checking spacer value one ...\n";
3   int i=0,j=0;;
4   unsigned long int xx;
5   float yy;
6   char arr[10],array[10],ch;
7   fstream f_cx,thresh;
8   f_cx.open("temp/out_c_extra.txt",ios::in);
9   thresh.open("temp/threshold1.txt",ios::in);
10  while((!f_cx.eof())&&(!thresh.eof())) {
11    f_cx.getline(arr,10);
12    xx=atoi(arr);
13    if(xx==0) break;
14
15    thresh.getline(array,10);
16    yy=atof(array);
17    if(yy==0) break;
18
19    another(xx,yy,a,b,flag);
```

```
20    }
21 }
```

Nota-se que os dados de coordenadas de início do promotor -35 (`out_c_extra`) e seus respectivos valores de limiar (`threshold1`) são lidos de forma integral, ou seja, do início ao fim. Nesta situação, determinados valores de `out_c_extra` e `threshold1` devem ser processados em seus respectivos processos. Uma possível abordagem é dividí-las em fragmentos a serem utilizados por diferentes processos.

Abaixo, trecho do código da função `diff_c_cc_threads` para vários processos:

```
1 void diff_c_cc_threads(char flag,int a,int b) {
2    ...
3    list<THRESH_OUTCX>::iterator it_thresh_outcx;
4    list<float>::iterator it_thresh;
5    list<int>::iterator it_out_c_extra;
6    ...
7    for (i=0, it_thresh_outcx = l_thresh_outcx.begin(); i < p->threadNum;
        it_thresh_outcx++, i++);
8    ...
9    for (it_thresh = it_thresh_outcx->l_thresh.begin(), it_out_c_extra =
        it_thresh_outcx->l_out_c_extra.begin(); it_thresh!= it_thresh_outcx
        ->l_thresh.end(); it_thresh++, it_out_c_extra++)
10   {
11     xx = *it_out_c_extra;
12     yy = *it_thresh;
13     anotherAll(xx,yy,p->a,p->b,p->flag,p->threadNum,tids);
14   }
15 }
```

Na solução apresentada, é criada uma estrutura que contém os valores de `out_c_extra` e `threshold1` respectivos a cada processo.

`typedef struct thresh_outcx1 list<float> l_thresh; list<int> l_out_c_extra; THRESH_OUTCX;`

Os valores dessa estrutura são armazenados em uma lista em que cada th-

read é responsável por um elemento:

```
list<THRESH_OUTCX> l_thresh_outcx;
```

Dessa forma, cada thread acessa a sua estrutura correspondente através da navegação pelos elementos da lista, feita a partir de um iterador. O iterador é um componente STL que percorre um container. No caso de listas encadeadas, o seu uso é necessário devido à impossibilidade de acessar os elementos diretamente, como acontece com vetores. Por exemplo, em componentes vector, os elementos podem ser acessados aleatoriamente por meio de índices.

No laço de repetição da linha 7 na função `diff_c_c_threads`, o iterador `it_thresh_outcx` navega na lista da estrutura `l_thresh_outcx` até encontrar o elemento correspondente ao processo.

Por exemplo, se o programa executa 6 threads, a lista `l_thresh_outcx` deve conter 6 elementos, cada um representando uma estrutura. Se o processo em execução possui o identificador 3, `p->threadNum` tem o valor 3 e o iterador percorre a lista até encontrar `p->threadNum` contendo esse valor. O identificador é atribuído na função `diff_threads`, que será visto logo a seguir.

Após referenciar o elemento desejado com o iterador `it_thresh_outcx` (linha 3), utiliza-se os iteradores declarados nas linhas 4 e 5 para navegar nas listas `l_thresh` e `l_out_c_extra` contidas na estrutura, conforme a linha 9. Esses elementos são, então, passados como argumento para a função another otimizada que processa somente os valores de threshold e coordenadas da região promotora -35.

Para criar as threads, foi implementada a função `diff_threads`:

```
1 void diff_threads(char flag, int a, int b) {
2   int err;
3   void *tret;
4   int i;
5   PARAM_T param_threads[NUM_THREADS];
6   pthread_t diff_tid[NUM_THREADS];
7   cout <<"Sliding in Progress (threaded)..\n";
8
```

```
9   for (i=0; i < NUM_THREADS; i++) {
10    param_threads[i].a = a;
11    param_threads[i].b = b;
12    param_threads[i].flag = flag;
13    param_threads[i].threadNum = i;
14    err=pthread_create(&diff_tid[i],NULL,diffAll,(void*)&param_threads[i
          ]);
15  }
16
17  for (i=0; i < NUM_THREADS; i++) {
18    pthread_join(diff_tid[i],&tret);
19  }
20 }
```

Nessa função, os processos são criados utilizando `pthread_create`, na linha 14. A constante `NUM_THREADS` indica o número de threads e é um argumento passado na linha de comando ao executar o programa. Na linha 18, `pthread_join` aguarda o término dos processos para prosseguir com o fluxo da função.

Os identificadores são atribuídos na linha 13 da função `diff_threads`. no seguinte trecho da função `diff_threads`:

Após aplicar o método de processamento paralelo utilizando *Pthreads*, o tempo médio de execução diminuiu de 44s674 para 36s528. O speedup em relação à versão otimizada sem multiprocessamento foi de:

$$speedup\_sRNAScanner\_paralelo = \frac{44,674}{36,528} = 1,22$$

O speedup em relação à versão original foi o seguinte:

$$speedup\_sRNAScanner\_paralelo = \frac{1384,523}{36,528} = 37,9$$

6.4 sRNAScanner Improved 3a. versão

Utilizando técnicas de otimização serial manual somente ao trecho crítico do programa, houve um ganho de desempenho de 31 vezes. Com base nesse resultado, outras partes do código foram investigadas para aplicar otimizações. Ao executar o profiler Open|SpeedShop na versão até então aprimorada, foi obtido o seguinte relatório (Figura 6.5):

Exclusive CPU time in seconds.	Inclusive CPU time in seconds.	% of Total Exclusive CPU Time	Function (defining location)
11.971428	12.085714	26.154806	__read_nocancel (/lib/x86_64-linux-gnu/libpthread-2.13.so)
4.571428	5.714286	9.987516	strtof_l (/lib/x86_64-linux-gnu/libc-2.13.so)
2.971429	4.285714	6.491885	anotherAll(int, float, int, int, char, int, char*) (/home/sfujii/Pesquisa/Source/myScanner_me...
2.885714	16.800000	6.304619	std::istream::get(char&) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.16)
2.714286	2.714286	5.930087	std::istream::sentry::sentry(std::istream&, bool) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6...
2.342857	2.342857	5.118602	__strtoull_internal (/lib/x86_64-linux-gnu/libc-2.13.so)
2.314286	2.342857	5.056180	__lseek_nocancel (/lib/x86_64-linux-gnu/libpthread-2.13.so)
2.142857	4.914286	4.681648	std::istream::getline(char*, long, char) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.16)
1.657143	13.228571	3.620474	slide_m3(void*) (/home/sfujii/Pesquisa/Source/myScanner_mem2)
1.628571	1.628571	3.558052	feof (/lib/x86_64-linux-gnu/libc-2.13.so)
1.485714	1.542857	3.245943	getc (/lib/x86_64-linux-gnu/libc-2.13.so)
1.457143	1.457143	3.183521	memchr (/lib/x86_64-linux-gnu/libc-2.13.so)
0.942857	0.942857	2.059925	memccpy (/lib/x86_64-linux-gnu/libc-2.13.so)
0.857143	0.857143	1.872659	sigset (/lib/x86_64-linux-gnu/libc-2.13.so)
0.742857	0.800000	1.622971	__write_nocancel (/lib/x86_64-linux-gnu/libpthread-2.13.so)
0.542857	13.571428	1.186017	fun(int, int, int, int, int, char, float, float) (/home/sfujii/Pesquisa/Source/myScanner_mem2)
0.314286	5.085714	0.686642	slide_m2(void*) (/home/sfujii/Pesquisa/Source/myScanner_mem2)
0.285714	0.285714	0.624220	munmap (/lib/x86_64-linux-gnu/libc-2.13.so)
0.285714	0.314286	0.624220	open64 (/lib/x86_64-linux-gnu/libc-2.13.so)
0.257143	1.028571	0.561798	std::basic_filebuf<char, std::char_traits<char> >::sync() (/usr/lib/x86_64-linux-gnu/libstd...
0.228571	3.000000	0.499376	std::istream::seekg(long, std::_Ios_Seekdir) (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.16)
0.200000	3.342857	0.436954	p_seq() (/home/sfujii/Pesquisa/Source/myScanner_mem2)

Figura 6.5: Relatório de saída do Open|SpeedShop utilizando a primeira versão otimizada do programa.

De acordo com o relatório, o consumo de tempo da função `another` diminuiu de 1275 segundos para 4,28 segundos após as otimizações. Contudo, as funções `strtof_l`, `get` e `getline`, que realizam a conversão de literal para número real, leitura de arquivo e leitura de literal, respectivamente, mostravam grande uso do processador. Como essas operações foram eliminadas na etapa de processamento crítico do programa, conclui-se que outras partes do código estejam utilizando recursos semelhantes.

6.4.1 Otimizações de Conversão e Acesso a Disco

Após a investigação, foram vistas várias funções que armazenavam e liam informações em arquivos de texto na mesma execução, solicitando, consequentemente, diversas conversões de tipo. Em outros trechos, foram encontradas porções de código

potencialmente otimizáveis utilizando as técnicas estudadas.

As seguintes funções foram alteradas para obter aumento de desempenho:

- `void special(char p_or_n,int a,int b);`
- `void fun(int num1,int num2,int num4,int a,int b,char p_or_n,float tt1,float tt2);`
- `void some123(char p_or_n);`
- `void *slide_m1(void *arg);`
- `void *slide_m2(void *arg);`
- `void *slide_m3(void *arg);`

A função `special` é invocada após o trecho otimizado com *Pthreads*, que consiste nas funções `another` e `diff_c_cc`. O código é mostrado abaixo:

```
1 void special(char p_or_n,int a,int b) {
2   fstream f1;
3   ofstream g("temp/final.txt",ios::trunc);
4   int i=0,j=0,k=0,num1,num2,num4,m=0,n=0;
5   char ch,arr[10],brr[10],d[10],t1[10],t2[10];
6   float x,y,tt1,tt2;
7   f1.open("temp/final_c_cc.txt",ios::in);
8
9   while((f1.get(ch))&&(ch!=EOF)) {
10        while(ch!='|') {
11          arr[i++]=ch;
12          f1.get(ch);
13        }arr[i]='\0';
14        num1=atoi(arr);i=0;
15
16        while((f1.get(ch))&&(ch!='|'))
17        d[k++]=ch;d[k]='\0';
18        num4=atoi(d);k=0;
```

```
19
20          while((f1.get(ch))&&(ch!='|'))
21          brr[j++]=ch;brr[j]='\0';
22          num2=atoi(brr);j=0;
23
24          while((f1.get(ch))&&(ch!='|'))
25          t1[m++]=ch;
26          tt1=atof(t1);m=0;
27
28          while((f1.get(ch))&&(ch!='|'))
29          t2[n++]=ch;
30          tt2=atof(t2);n=0;
31
32          while((f1.get(ch))&&(ch!='\n'));
33
34          fun(num1,num2,num4,a,b,p_or_n,tt1,tt2);
35    }
36 }
```

Observou-se que entre as linhas 10 e 32 (linhas pares) era utilizada a mesma abordagem identificada na função `another`, onde havia leitura de dados em arquivo (`final_c_cc.txt`) e conversão de tipo. A função `special` com otimizações manuais resultou no código a seguir:

```
1 void special(char p_or_n,int a,int b) {
2   int i=0,num1,num2,num4;
3   float tt1,tt2;
4   list<FINAL_PARAM>::iterator it;
5
6   for(i=0,it=l_final_c_cc.begin(); i < l_final_c_cc.size(); it++,i++) {
7     num1 = it->xxr1;
8     num4 = it->c;
9     num2 = it->yyr2;
10    tt1 = it->thresh1;
11    tt2 = it->out_c_extra;
12    fun(num1,num2,num4,a,b,p_or_n,tt1,tt2);
```

```
13   }
14 }
```

Os dados que eram lidos do arquivo (`final_c_cc.txt`) foram armazenados em um componente `list` nomeado `l_final_c_cc`, que contém uma lista de estruturas que guardam os mesmos dados presentes no arquivo. Utilizando um iterador, os valores contidos nos campos da estrutura são atribuídos como argumento da função `fun`.

A função `fun` apresentava a mesma forma de leitura de dados em arquivo, como pode ser visto entre as linhas 13 e 16 a seguir:

```
1 void fun(int num1,int num2,int num4,int a,int b,char p_or_n,float tt1,
     float tt2) {
2   int num3,diff,noo;
3   char c[10],arr[10];
4   float qwe;
5   ofstream g("temp/final.txt",ios::app);
6   fstream f3,ttt,fp;
7   f3.open("temp/out_ccc.txt",ios::in);
8   fp.open("temp/out_ccc_extra.txt",ios::in);
9   ttt.open("temp/threshold3.txt",ios::in);
10  while(!f3.eof()) {
11    f3.getline(c,10);
12    num3=atoi(c);
13    fp.getline(c,10);
14    oo=atoi(c);
15    ttt.getline(arr,10);
16    qwe=atof(arr);
17    diff=num3-num2;
18    if(p_or_n=='n')diff=num2-num3;
19    if((diff>=a)&&(diff<=b)) {
20      if(p_or_n=='p') {
21        g << num1 << "|" << num4 << "|" << diff << "|" << noo << "|";
22        g << tt1 << "|" << tt2 << "|" << qwe <<"|" <<  "\n";
23      }
```

```
24        else {
25          g << num1 << "|" << num4 << "|" << diff << "|" << noo << "|";
26          g << tt1 << "|" << tt2 << "|" << qwe << "|c" <<  "\n";
27        }
28        g << flush;
29      }
30    }
31  }
```

Na versão otimizada, os valores foram armazenados anteriormente (função `slide_m3`) nos componentes `vector` `v_xccc`, `v_xcccx` e `v_thresh3` para serem lidos na função `fun`. Esse procedimento descartou a necessidade de leitura dos arquivos `out_ccc.txt`, `out_ccc_extra.txt` e `threshold3.txt`. A função otimizada ficou como segue:

```
1 void fun(int num1,int num2,int num4,int a,int b,char p_or_n,float tt1,
     float tt2) {
2   int num3,diff,noo,i=0;
3   char c[10],arr[10];
4   float qwe;
5   ofstream g("temp/final.txt",ios::app);
6   FINAL final_aux;
7
8   for (i=0; i < v_xccc.size(); i++) {
9     num3 = v_xccc[i];
10    noo = v_xcccx[i];
11    qwe = v_thresh3[i];
12    diff=num3-num2;
13    if(p_or_n=='n')diff=num2-num3;
14    if((diff>=a)&&(diff<=b)) {
15      final_aux.num1 = num1;
16      final_aux.num4 = num4;
17      final_aux.diff = diff;
18      final_aux.noo = noo;
19      final_aux.tt1 = tt1;
```

```
20        final_aux.tt2 = tt2;
21        final_aux.qwe = qwe;
22
23        l_final.push_back(final_aux);
24
25        if(p_or_n=='p') {
26          g << num1 << "|" << num4 << "|" << diff << "|" << noo << "|";
27          g << tt1 << "|" << tt2 << "|" << qwe <<"|" <<  "\n";
28        }
29        else {
30          g << num1 << "|" << num4 << "|" << diff << "|" << noo << "|";
31          g << tt1 << "|" << tt2 << "|" << qwe << "|c" <<  "\n";
32        }
33        g << flush;
34      }
35    }
36 }
```

Os dados apresentados entre as linhas 15 e 21 foram armazenados em uma lista encadeada do tipo abstrato de dado `FINAL` que guarda estruturas contendo campos destinados a cada dado. Essa manipulação foi feita para evitar novo acesso ao disco na função `some123`, que utiliza esses dados.

Na função `some123` a leitura em arquivo de texto e conversão de dados pode ser visto entre as linhas 11 e 61, a seguir:

```
1 void some123(char p_or_n){
2   fstream f1;
3   ofstream f3,f4;
4   f3.open("temp/inter123.txt",ios::trunc);
5   f4.open("temp/overlap123.txt",ios::trunc);
6   int i=0,j=0,k=0,m=0,n=0,p=0,q;
7   char dhaval[100];
8   f1.open("temp/final.txt",ios::in);
9   void range();
10  while(!f1.eof()) {
```

```
11      f1.getline(dhaval,100);
12      char arr[10]={'\0'};
13      while(dhaval[p]!='|') {
14        arr[i++]=dhaval[p];
15        p++;
16      }
17      num1=atoi(arr);i=0;
18      if(num1==0)break;
19      p++;
20      char d[10]={'\0'};
21      while(dhaval[p]!='|') {
22        d[k++]=dhaval[p];;
23        p++;
24      }
25      num4=atoi(d);k=0;
26      p++;
27      while(dhaval[p]==' ')p++;
28      char brr[10]={'\0'};
29      while(dhaval[p]!='|') {
30        brr[j++]=dhaval[p];
31        p++;
32      }
33      num5=atoi(brr);j=0;
34      p++;
35      char dd[10]={'\0'};
36      while(dhaval[p]!='|') {
37        dd[k++]=dhaval[p];;
38        p++;
39      }
40      num2=atoi(dd);k=0;
41      p++;
42      char t1[10]={'\0'};
43      while(dhaval[p]!='|') {
44        t1[m++]=dhaval[p];
45        p++;
```

```
46      }
47      tt1=atof(t1);m=0;
48      p++;
49      char t2[10]={'\0'};
50      while(dhaval[p]!='|') {
51        t2[n++]=dhaval[p];
52        p++;
53      }
54      tt2=atof(t2);n=0;
55      p++;
56      char t3[10]={'\0'};
57      while(dhaval[p]!='|') {
58        t3[m++]=dhaval[p];
59        p++;
60      }
61      tt3=atof(t3);m=0;
62      p=0;
63      range123(p_or_n);
64    }
65 }
```

Na versão otimizada, a leitura de arquivo é evitada por meio do componente *list* `l_final`, utilizado na função `fun` para armazenar os dados que são lidos na função `some123`. As alterações são mostradas abaixo:

```
1  void some123(char p_or_n) {
2    int i=0,j=0,k=0,m=0,n=0,p=0,q;
3    char dhaval[100];
4    list<FINAL>::iterator it_final;
5    for(it_final = l_final.begin(); it_final != l_final.end(); it_final++)
         {
6      num1 = it_final->num1;
7      num4 = it_final->num4;
8      num5 = it_final->diff;
9      num2 = it_final->noo;
10     tt1  = it_final->tt1;
```

```
11      tt2  = it_final->tt2;
12      tt3  = it_final->qwe;
13      range123(p_or_n);
14    }
15 }
```

6.4.2 Otimizações de Desvios Condicionais

De acordo com as técnicas de otimização serial vistas na seção 3.7, o desvio condicional deve ser evitado devido ao seu alto custo computacional. Uma característica observada nas funções `slide_m1`, `slide_m2` e `slide_m3` é o uso do comando `if` no seguinte trecho:

```
1 void *slide_m3(void *arg) {
2   ...
3   while((fp.get(c))&&(c!=EOF)) {
4     ...
5     if (p_or_n=='n')
6       ...
7     else if (p_or_n=='p')
8       ...
9   }
10 }
```

O comando `if` está presente em um laço de repetição que realiza o número de iterações equivalente ao conteúdo do arquivo `linear.txt`. Esse arquivo contém os nucleotídeos do genoma de entrada. No caso da *Salmonella enterica serovar Thyphimurium*, são 4857433 nucleotídeos, o que resulta no número de iterações necessárias e, consequentemente, o número de operações de desvio condicional `if`.

A solução sugerida inicia-se pelo armazenamento do conteúdo do arquivo `linear.txt` em uma lista encadeada `list` [1] com uma pequena modificação na função `linear_genome`, que é responsável pela criação do arquivo `linear.txt`:

[1] Foi utilizada lista encadeada ao invés de vetor devido às características de alocação de memória discutidas na seção 3.6.1

```
1 list<char> linear;
2 ...
3 void linear_genome(char *fname) {
4   fstream fp1,fp2;
5   fp1.open(fname,ios::in);
6   fp2.open("temp/linear",ios::out);
7   while((fp1.get(c))&&(c!=EOF)) {
8     if(c=='>') {
9       while(c!='\n')
10      fp1.get(c);
11    }
12    else if(c=='\n');
13      else if (c=='A'||c=='G'||c=='T'||c=='C') {
14        fp2.put(c);
15        no_of_bases++;
16        fp2.put('\n');
17        linear.push_back(c);
18      }
19  }
20 }
```

Na linha 14, o elemento do tipo `char` correspondente a um nucleotídeo é armazenado no arquivo `linear.txt` e, na linha 17, na lista escadeada `linear`. Como o arquivo `linear.txt` é lido nas funções `slide_m1`, `slide_m2` e `slide_m3`, a lista criada beneficia essas três funções, prevenindo a necessidade de leitura em arquivo.

O próximo passo foi retirar o comando `if` do laço de repetição. A solução encontrada foi replicar o código para cada condição.

```
1 void *slide_m3(void *arg) {
2   list<char>::iterator it;
3   list<char>::iterator aux;
4   aux=linear.begin();
5
6   it = linear.begin();
```

```
7   ll = (linear.size()-r3);
8
9   if (p_or_n=='n') {
10    for(l=0;l<ll;l++)
11      ...
12  }
13  else {
14    for(l=0;l<ll;l++)
15      ...
16  }
17 }
```

Ao aplicar as otimizações descritas, o tempo médio de execução caiu de 36s528 para 16s283. O speedup em relação à versão paralelizada é o seguinte:

$$speedup\_sRNAScanner\_v3 = \frac{36,528}{16,283} = 2,24$$

O speedup em relação à versão original é apresentado abaixo:

$$speedup\_sRNAScanner\_v3 = \frac{1384,523}{16,283} = 85,03$$

A Tabela 6.1 apresenta o tempo de execução e a Tabela 6.2 apresenta o *speedup* das versões do sRNAscanner.

Tabela 6.1: Comparação entre as versões do sRNAscanner em relação ao tempo médio de execução.

Versão	1 thread (segundos)	6 threads (segundos)
Original	1384,523	1112,842
Improved 1	44,374	-
Improved 2	44,382	36,528
Improved 3	24,776	16,283

A Figura 6.6 mostra a relação entre o tempo de execução do programa original e das versões otimizadas, sendo que, quanto menor o valor, melhor o desempenho. Na Figura 6.7 observa-se pelo cálculo do *speedup* a evolução no desenvolvimento das

Tabela 6.2: Comparação entre as versões do sRNAscanner em relação ao *speedup*.

Versão	1 thread	6 threads
Original	1	1,24
Improved 1	31,2	-
Improved 2	31,12	37,9
Improved 3	55,89	85,03

versões, sendo que a última versão obteve o melhor ganho de performance.

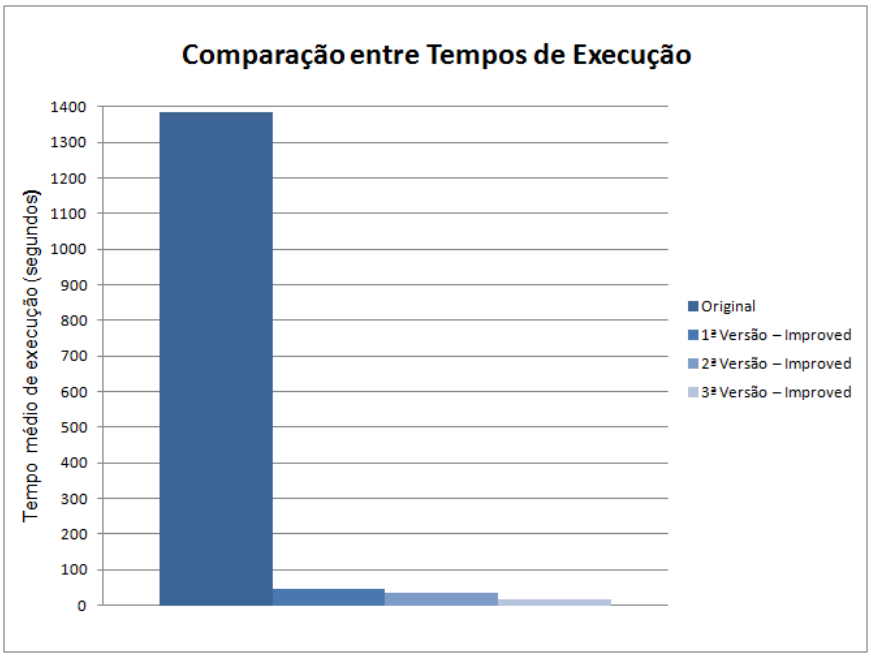

Figura 6.6: Gráfico comparativo entre os tempos médios de execução da versão original e das três versões otimizadas do sRNAscanner.

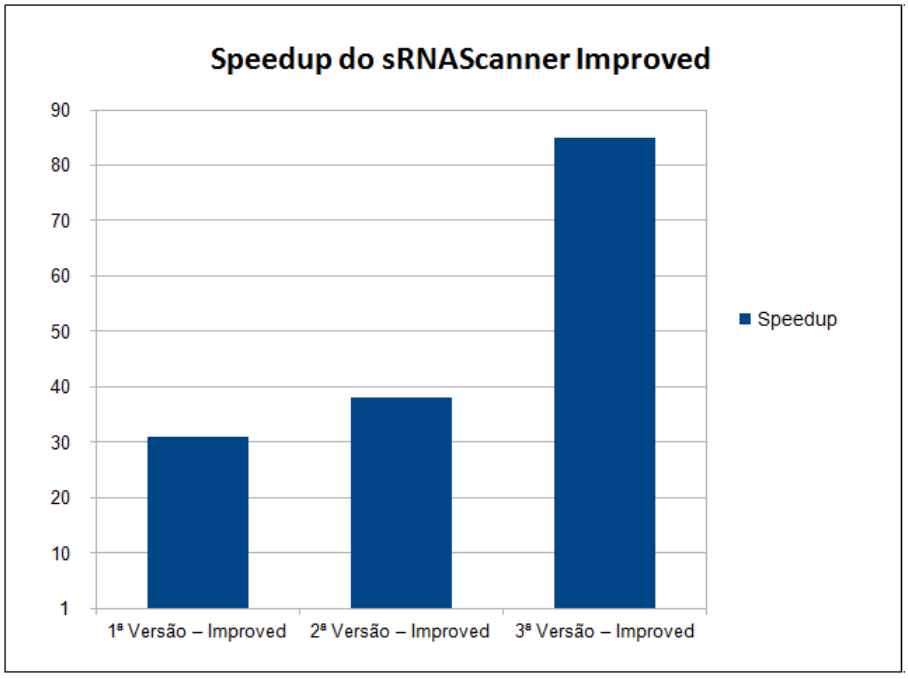

Figura 6.7: Gráfico comparativo entre o speedup das três versões otimizadas do sRNAscanner.

CAPÍTULO 7

DISCUSSÃO

Os resultados dos testes demonstram a importância da utilização de técnicas de otimização serial no desempenho da ferramenta de estudo de caso. Nas três versões de otimização apresentadas, o impacto ficou evidente em relação à relevância das melhorias aplicadas a cada uma delas.

Na primeira otimização foi obtido um *speedup* de 31 vezes, mostrando que a utilização da estratégia de leitura em disco aliada a conversão de tipos comprometia o desempenho do programa. 94,64% do tempo de execução era gasto em uma única rotina que efetuava essas operações centenas de milhares de vezes. Ao retirar essas operações, o tempo de execução do programa caiu de 1384s523 para um tempo médio de 44s374.

A segunda otimização utilizou a paralelização de uma função visando dividir a execução da tarefa. Nos resultados apresentados, o programa foi executado com 6 threads. O ganho em relação à primeira versão foi de 1,22 vezes, ou seja, uma melhora de 22%. Isso mostra que mesmo tarefas paralelizadas podem não ter o alto ganho esperado. Já em relação à versão original, o ganho foi de 37,9 vezes, o que diminuiu o tempo de execução para 36s528.

A última versão estendeu as otimização seriais a outras partes do programa. Neste caso, a versão com Pthreads foi utilizada como base e algumas outras funções foram analisadas, sendo alvos de melhoria. As funções que apresentavam leitura de arquivos e conversões de tipo foram alteradas e tipos de dados estruturados (list e vector) foram utilizados para armazenar e manipular os dados entre as funções. Além disso, desvios condicionais que estavam dentro de laços foram retirados deles, ficando assim na parte externa do laço. Com essas alterações, foi feita uma refatoração do código original e o tempo de execução caiu para 16s283 com um *speedup* de 85,03

em relação a versão original e de 2,24 em relação a 2^a. versão.

O uso de STL como biblioteca auxiliar facilitou a refatoração do código, diminuindo o tempo de implementação e permitindo o teste de diferentes estruturas de dados. Através de seu uso, foi possível perceber que a utilização de matriz com memória alocada estaticamente não é recomendável na manipulação de grande quantidade de dados. Infelizmente, pela facilidade de uso e também pelos vícios adquiridos na academia, esse tipo de estrutura de dados possui a forte tendência de ser a primeira opção na implementação de ferramentas de dados computacionais.

A ferramenta gprof mostrou-se capaz de fornecer informações básicas relevantes sobre o desempenho do programa. No entanto, Open|SpeeedShop forneceu informações pontuais sobre as funções exatas do compilador que estavam consumindo tempo, como as rotinas responsáveis pela leitura de arquivo e conversão de tipo.

CAPÍTULO 8

CONCLUSÃO

A aplicação de técnicas de computação de alto desempenho em ferramentas de bioinformática é de grande utilidade. O uso intenso de processamento com o objetivo de extrair informação relevante através de sequências de símbolos representa o cenário ideal para a inserção de metodologias que visam extrair o máximo de desempenho do equipamento disponível.

O aproveitamento dos recursos de programação paralela devem partir da programação efetiva utilizando um único processo. Por exemplo, o processamento efetuado em uma etapa de alto custo, como uma conversão de tipo em um laço de repetição, não será melhorado plenamente somente com o uso de multiprocessamento. Nesse caso, os conceitos de programação serial otimizada devem ser revisitados para evitar que operações de alto custo sejam utilizadas indiscriminadamente.

A busca desenfreada pela utilização de vários processadores acaba ocultando a real necessidade de analisar o código e descobrir o gargalo existente. Este trabalho mostrou que o aumento de desempenho passa primeiramente pela análise serial do programa. Esta negligência com o desempenho é resultado da pressão existente na busca de novos resultados. Assim, ao atingir esse objetivo, a ferramenta é disponibilizada sem ser submetida à etapa de análise de desempenho e consequente refatoração. Infelizmente, após a conclusão de uma ferramenta, ela é geralmente abandonada para a criação de uma nova, pois a pesquisa exige a renovação da teoria estudada. Em bioinformática, essa prática é comum, pois existe a necessidade do pesquisador conhecer as áreas de biologia e computação, voltando seus conhecimentos para a construção de uma ferramenta que apresente os resultados esperados em tempo de implementação hábil. Felizmente, alguns trabalhos foram publicados com ênfase em desempenho, como Velvet [Zerbino and Birney, 2008] e GPUBlast

[Vouzis and Sahinidis, 2011], indicando a necessidade de melhorar também o meio de obtenção dos resultados.

8.1 Trabalhos futuros

As seguintes etapas foram planejadas para dar continuidade à pesquisa desenvolvida:

- Testes da ferramenta em uma estação de trabalho de alto desempenho (cluster);
- Aplicação de técnicas de computação em alto desempenho em outras ferramentas de bioinformática que apresentem grande tempo de processamento;
- Testes com novos organismos para verificar a influência de dados a serem analisados dentro do programa;
- Implementação de busca de sRNAs em vários organismos ao mesmo tempo, diminuindo a necessidade de executar o programa várias vezes, já que em uma única execução seria obtido o resultado de vários organismos;
- Utilização de placa gráfica utilizando GPGPU (*General-Purpose Computation on Graphics*) com a tecnologia CUDA (*Compute Unified Device Architecture*).

REFERÊNCIAS

[Altuvia, 2007] Altuvia, S. (2007). Identification of bacterial small non-coding rnas: experimental approaches. *Current Opinion in Microbiology*, 10(3):257 – 261.

[Aylward et al., 2007] Aylward, Jomier, Barre, Davis, and Ibanez (2007). Optimizing itk registration methods for multi-processor, shared-memory systems. Technical report.

[Bois, 2008] Bois, A. R. D. (2008). Memórias transacionais e troca de mensagens: Duas alternativas para programação de máquinas multi-core. *8a. Escola Regional de Alto Desempenho (ERAD 2008)*.

[Busch et al., 2008] Busch, A., Richter, A. S., and Backofen, R. (2008). Intarna: efficient prediction of bacterial srna targets incorporating target site accessibility and seed regions. *Bioinformatics*, 24(24):2849–2856.

[Campbell et al., 2001] Campbell, M., Ferreira, H., and Carlini, C. (2001). *Bioquímica*, volume 3. Artmed Editora.

[Cunha et al., 2001] Cunha, M., alvaro Coutinho, and jose Telles (2001). high performance computing applied to boundary elements: Potential problems. *Cilamce*, 27:182–188.

[DEITEL and DEITEL, 2006] DEITEL, H. and DEITEL, P. (2006). *C++ Como Programar - 5a edição*. PRENTICE HALL BRASIL.

[Dowd and Severance, 1998] Dowd, K. and Severance, C. (1998). *High performance computing*. RISC architectures, optimization & benchmarks. O'Reilly.

[Farber, 2011] Farber, R. (2011). *Introduction to GPGPUs and Massively Threaded Programming*. CRC Press.

[Galarowicz, 2011] Galarowicz, J. (2011). An introduction into performance analysis for hpc systems with openspeedshop. In *Super Computing 2011*.

[Gottesman and Storz, 2010] Gottesman, S. and Storz, G. (2010). Bacterial small RNA regulators: Versatile roles and rapidly evolving variations. *Cold Spring Harbor perspectives in biology*.

[Gruber et al., 2010] Gruber, A. R., Findeiß, S., Washietl, S., Hofacker, I. L., and Stadler, P. F. (2010). Rnaz 2.0: Improved noncoding rna detection. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, 15:69–79.

[Hertz and Stormo, 1999] Hertz, G. Z. and Stormo, G. D. (1999). Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*, 15(7):563–577.

[Jacobs et al., 2008] Jacobs, G. H. H., Chen, A., Stevens, S. G. G., Stockwell, P. A. A., Black, M. A. A., Tate, W. P. P., and Brown, C. M. M. (2008). Transterm: a database to aid the analysis of regulatory sequences in mrnas. *Nucleic acids research*, 37(1):D72–D76.

[Liu et al., 2010] Liu, Y., Schmidt, B., and Maskell, D. (2010). CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93+.

[Liu et al., 2011] Liu, Y., Schmidt, B., and Maskell, D. (2011). *Parallel Bioinformatics Algorithms for CUDA-Enabled GPUs*. CRC Press.

[Livny et al., 2005] Livny, J., Fogel, M. A., Davis, B. M., and Waldor, M. K. (2005). sRNAPredict: an integrative computational approach to identify sRNAs in bacterial genomes. *Nucleic Acids Res.*, 33(13):4096–105.

[London et al., 2001] London, K., Moore, S., Mucci, P., Seymour, K., and Luczak, R. (2001). The papi cross-platform interface to hardware performance counters. In *Department of Defense Users Group Conference Proceedings*, pages 18–21.

[Macke et al., 2001] Macke, T. J., Ecker, D. J., Gutell, R. R., Gautheret, D., Case, D. A., and Sampath, R. (2001). RNAMotif, an RNA secondary structure definition and search algorithm. *Nucleic acids research*, 29(22):4724–4735.

[Majdalani et al., 2005] Majdalani, N., Vanderpool, C., and Gottesman, S. (2005). Bacterial small RNA regulators. *Critical Reviews in Biochemistry and Molecular Biology*, 40(2):93–113.

[Manavski and Valle, 2008] Manavski, S. A. and Valle, G. (2008). CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics*, 9 Suppl 2(Suppl 2):S10–9.

[McClelland et al., 2001] McClelland, M., Sanderson, K. E., Spieth, J., Clifton, S. W., Latreille, P., Courtney, L., Porwollik, S., Ali, J., Dante, M., Du, F., Hou, S., Layman, D., Leonard, S., Nguyen, C., Scott, K., Holmes, A., Grewal, N., Mulvaney, E., Ryan, E., Sun, H., Florea, L., Miller, W., Stoneking, T., Nhan, M., Waterston, R., and Wilson, R. K. (2001). Complete genome sequence of Salmonella enterica serovar Typhimurium LT2. *Nature*, 413(6858):852–6.

[Milani et al., 2007] Milani, C. R., Cavalheiro, G. G., and de Oliveira, L. F. (2007). Ferramenta multithread de visualização interativa para auxílio na detecção do foco epileptogênico. *8a. Escola Regional de Alto Desempenho (ERAD 2008)*.

[Nelson and Lehninger, 2008] Nelson, K. and Lehninger, A. (2008). *Princípios de Bioquímica*. SARVIER.

[Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads programming - a POSIX standard for better multiprocessing.* O'Reilly.

[Patterson and Hennessy, 2009] Patterson, D. and Hennessy, J. (2009). *Computer organization and design: the hardware/software interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Morgan Kaufmann.

[Pichon and Felden, 2003] Pichon, C. and Felden, B. (2003). Intergenic sequence inspector: searching and identifying bacterial rnas. *Bioinformatics*, 19(13):1707–1709.

[Ponty, 2012] Ponty, Y. (2012). Varna: Visualization applet for rna. http://varna.lri.fr. Acessado em 10/02/2012.

[Reese, 2001] Reese, M. G. (2001). Application of a time-delay neural network to promoter annotation in the drosophila melanogaster genome. *Computers & Chemistry*, 26(1):51–56.

[Riley et al., 2006] Riley, M., Abe, T., Arnaud, M. B., Berlyn, M. K. B., Blattner, F. R., Chaudhuri, R. R., Glasner, J. D., Horiuchi, T., Keseler, I. M., Kosuge, T., Mori, H., Perna, N. T., Plunkett, G., Rudd, K. E., Serres, M. H., Thomas, G. H., Thomson, N. R., Wishart, D., and Wanner, B. L. (2006). Escherichia coli k-12: a cooperatively developed annotation snapshot–2005. *Nucl. Acids Res.*, 34(1):1–9.

[Rivas and Eddy, 2001] Rivas, E. and Eddy, S. R. (2001). Noncoding RNA gene detection using comparative sequence analysis. *BMC bioinformatics*, 2(1):8+.

[Sanders and Kandrot, 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley.

[Schatz et al., 2007] Schatz, M., Trapnell, C., Delcher, A., and Varshney, A. (2007). High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474+.

[Schmidt, 2010] Schmidt, B. (2010). *Bioinformatics: High Performance Parallel Computer Architectures (Embedded Multi-Core Systems*. CRC Press.

[Snustad and Simmons, 2008] Snustad, D. and Simmons, M. (2008). *Fundamentos de Genética*. GUANABARA.

[Snyder and Champness, 2007] Snyder, L. and Champness, W. (2007). *Molecular genetics of bacteria*. American Society Mic Series. ASM Press.

[Sridhar et al., 2010] Sridhar, J., Narmada, S. R., Sabarinathan, R., Ou, H.-Y., Deng, Z., Sekar, K., Rafi, Z. A., and Rajakumar, K. (2010). sRNAscanner: A Computational Tool for Intergenic Small RNA Detection in Bacterial Genomes. *PLoS ONE*, 5(8):e11970+.

[Storz et al., 2005] Storz, G., Altuvia, S., and Wassarman, K. M. (2005). An abundance of RNA regulators. *Annual review of biochemistry*, 74:199–217.

[Thébault et al., 2006] Thébault, P., De Givry, S., Schiex, T., and Gaspin, C. (2006). Searching rna motifs and their intermolecular contacts with constraint networks. *Bioinformatics*, 22(17):2074–2080.

[Thompson et al., 1994] Thompson, J. D., Higgins, D. G., and Gibson, T. J. (1994). CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680.

[Valverde et al., 2008] Valverde, C., Livny, J., Schluter, J.-P., Reinkensmeier, J., Becker, A., and Parisi, G. (2008). Prediction of Sinorhizobium meliloti sRNA genes and experimental detection in strain 2011. *BMC Genomics*, 9:416+.

[Vogel and Wagner, 2007] Vogel, J. and Wagner, E. G. H. (2007). Target identification of small noncoding rnas in bacteria. *Current Opinion in Microbiology*, 10(3):262–270.

[Vouzis and Sahinidis, 2011] Vouzis, P. D. and Sahinidis, N. V. (2011). Gpu-blast. *Bioinformatics*, 27:182–188.

[Wadleigh and Crawford, 2000] Wadleigh, K. and Crawford, I. (2000). *Software optimization for high-performance computing*. HP Professional Series. Prentice Hall PTR.

[Zerbino and Birney, 2008] Zerbino, D. R. and Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–9.

[Zhang et al., 2004] Zhang, Y., Zhang, Z., Ling, L., Shi, B., and Chen, R. (2004). Conservation analysis of small rna genes in escherichia coli. *Bioinformatics*, 20:599–603.

SÉRGIO YOSHIMITSU FUJII

UTILIZAÇÃO DE TÉCNICAS DE OTIMIZAÇÃO DE DESEMPENHO EM BIOINFORMÁTICA. ESTUDO DE CASO: SRNASCANNER

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Bioinformática, Setor de Educação Profissional e Tecnológica, Universidade Federal do Paraná.
Orientador: Prof. Dr. Lucas Ferrari de Oliveira
Orientadora: Prof^a. Dr^a. Maria Berenice Reynaud Steffens

CURITIBA

2012