MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
SECRETARIA DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

FELIPE FERNANDES ALBRECHT

# ALGORITMO OTIMIZADO PARA COMPARAÇÃO E BUSCA DE BASE DE DADOS

Rio de Janeiro
2009

**INSTITUTO MILITAR DE ENGENHARIA**

**FELIPE FERNANDES ALBRECHT**

# ALGORITMO OTIMIZADO PARA COMPARAÇÃO E BUSCA DE BASE DE DADOS

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientador: Prof. Raquel Coelho Gomes Pinto - D.C.
Co-orientador: Prof. Claudia Marcela Justel - D. C.

Rio de Janeiro
2009

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80-Praia Vermelha
Rio de Janeiro-RJ CEP 22290-270



XXXXXAlbrecht, F. F.
Algoritmo otimizado para comparação e busca de base de dados/ Felipe Fernandes Albrecht.
– Rio de Janeiro: Instituto Militar de Engenharia, 2009.
xxx p.: il., tab.

Dissertação (mestrado) – Instituto Militar de Engenharia – Rio de Janeiro, 2009.

1. Bioinformática. 2. Computação paralela e de alto desempenho. 3. Recuperação de Informação. I. Título. II. Instituto Militar de Engenharia.

CDD 629.892

**INSTITUTO MILITAR DE ENGENHARIA**
**FELIPE FERNANDES ALBRECHT**
**ALGORITMO OTIMIZADO PARA COMPARAÇÃO E BUSCA DE BASE DE DADOS**

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientador: Prof. Raquel Coelho Gomes Pinto - D.C.

Co-orientador: Prof. Claudia Marcela Justel - D. C.

Aprovada em 26 de Outubro de 2009 pela seguinte Banca Examinadora:

---

Prof. Raquel Coelho Gomes Pinto - D.C. do IME - Presidente

---

Prof. Claudia Marcela Justel - D. C. do IME

---

Prof. Maria Cláudia Reis Cavalcanti - D. C. do IME

---

Prof. Alberto Martín Rivera Dávila - D. C. da FIOCRUZ

---

Prof. Lauro Luis Armond Whately - D. C. da UFRJ

Rio de Janeiro
2009

Ao meu pai que me ensinou a usar os números e a minha mãe que me ensinou a usar as palavras.

## AGRADECIMENTOS

O maior agradecimento vai para o meu pai, que em toda a minha, insentivou-me nos estudos, trabalho e mostrou como deve ser alguém com carater. Junto a ele, agradeço a minha mãe Rogéria e irmã Lara, que sempre estiveram juntas de mim, nos bons e maus momentos e sempre incentivando meus trabalhos e estudos. Agredeço a família do meu pai e a minha Vó Janildes, por terem insentivado a conclusão do mestrado e especialmente por terem dado a mim, minha mae e irmã, muito apoio durante este ano. Agradeço a minha namorada Débora pelo apoio e carinho dado para a finalização deste trabalho.

Agradeço ao professor Alberto e a professora Yoko, pelo apoio dado no início do mestrado. Ao professor Major Salles, pelo apoio e ajuda. Agradeço a professora Cláudia Justel, pela ajuda na confecção deste trabalho. Agradecimentos a professora Raquel, pela ajuda e apoio enquanto estive distante, agredeço pelo discussões interessantes, idéias e ajuda na confecção deste trabalho. Muitos agradecimentos ao professor Nelson, que sempre me ajudou no Rio de Janeiro, inclusive abrindo as portas da casa dela, pelos trabalhos que fizemos juntos, pelos apoios dados nos trabalhos feitos para outras disciplinas e artigos, pelos almoços e jantas em família e pelo grande apoio dado neste último ano.

Acho difícil que leiam, mas gostaria de agradecer aos colegas de trabalho no Google, especialmente ao Swaminatan Mahadevan, pela experiência e por tudo que aprendi e vivi lá e pelo colega e amigo Yang Ho pelas conversas e trocas de idéias sobre bioinformática.

*Felipe Fernandes Albrecht*

Aum Namah Shivaya

## SUMÁRIO

## LISTA DE ILUSTRAÇÕES

## LISTA DE TABELAS

## RESUMO

A busca por sequências genéticas similares em base de dados é uma das funções básicas na bioinformática. Porém as base de dados de sequências genéticas tem sofrido um crescimento exponencial tornando o baixo desempenho desta busca um problema. O aumento do poder computacional dos processadores tem sido alcançado através da utilização de vários núcleos de processamento em um único chip. As técnicas de busca de sequências genéticas que utilizam estrutura de dados mais otimizadas, como, índices invertidos, não utilizam estes núcleos de processamento extras. Desta forma, este trabalho visa utilizar indexação de dados da base de dados com a paralelização do processo de busca de sequências similares. Durante este trabalho foi desenvolvido um protótipo que utiliza técnicas de paralelização e índices invertidos para a verificação da viabilidade de utilizar estas duas técnicas simultaneamente. Foram executados experimentos para analisar o ganho de desempenho quando utilizados índices invertidos e paralelismo e a qualidade dos resultados quando comparados com outras ferramentas de busca. Os resultados foram promissores, pois o ganho com paralelismo chega a ultrapassar o *speedup* linear, a execução com paralelismo é em média 20 vezes mais rápida que a ferramenta *NCBI BLAST* quando este também usa paralelismo e encontrando mais de 70% dos mesmo alinhamentos reportados pelo *BLAST* para *e-values* iguais ou inferiores a $10e^{-15}$, mostrando assim sua eficácia para encontrar sequências genéticas similares.

## ABSTRACT

The search for similar genetics sequences at data bases is one of the basic functions at the bioinformatics. However the genetics sequences data bases are growing exponentially making the problem the poor performance of this search. The processors computational power growth have been achieved by the utilization of many processing cores in only one chip. The genetics sequences searching techniques that use more optimized data structures, like, inverted index, do not use these extras processing cores. This way, this work aims to use data base data indexing with the parallelization of the similar sequences searching process. During the work a prototype was developed which uses the parallelization techniques and inverted index to verify the viability of to use these two techniques simultaneously. Experiments was executed to analyze the performance gain when inverted index and parallelism are used. The results quality was compared with others search tools. The results was promising, because the prototype is average 20 times faster that the NCBI-BLAST when it also uses parallelism. The prototype found more than 70% of the same alignments found by BLAST for e-values equals or less than $10e^{-15}$, showing its efficiency to find similar genetics sequences.

# 1 INTRODUÇÃO

Uma das mais importantes atividades na bioinformática é a busca por sequências similares em base de dados. Este tipo de busca possui como entrada sequências genéticas, como o Ácido Desoxirribonucléico (DNA), Ácido Ribonucléico (RNA) ou sequências peptídicas, que formam as proteínas. A busca é feita a partir de um conjunto de sequências armazenadas numa base de dados. Procura-se pelas sequências que possuem um alto grau de similaridade com a sequência de entrada. Acerca do procedimento de comparação, algumas informações são importantes. Primeiramente, não são pesquisadas sequências *iguais* à sequência de entrada, porém *similares* ou de forma mais coloquial, "parecidas". Este objetivo, de busca por similaridade e não por igualdade, é resultado de um conceito da genética molecular: homologia. Homologia é o termo utilizado para descrever sequências genéticas que possuam um parentesco, ou seja, que possuam a mesma origem evolutiva.

O problema da busca por sequências genéticas similares está relacionado com o crescimento exponencial das bases de dados de sequências genéticas. Apesar da utilização de heurísticas o tempo do processo de busca de sequências similares está se tornando muito dispendioso. Sendo assim, este trabalho procura utilizar a técnica de índices invertidos para otimizar o processo de busca. Outra questão a ser abordada neste trabalho é o uso de processadores com mais de um núcleo de processamento. Grande parte das ferramentas atuais não fazem uso dos núcleos extras de processamento. Através de pesquisas na literatura, verificou-se que atualmente não existe nenhuma ferramenta de busca de sequências genéticas similares que indexe a base de dados e que também faça uso de processadores com mais de uma unidade de processamento. Desta forma, o objetivo deste trabalho é desenvolver um aplicativo para busca de sequências similares com indexação da base de dados de sequências genéticas, utilizando-se a capacidade computacional dos processadores com mais de um núcleo para minimizar o tempo de busca de sequências genéticas similares.

O protótipo desenvolvido utiliza técnicas de indexação com índices invertidos, paralelização no processo de busca no índice, divisão da base de dados e otimizações no algoritmo de alinhamento das sequências. Ele foi desenvolvido utilizando-se o ambiente

*Java* e foram executados experimentos para verificar o tempo de busca em comparação a outras ferramentas, o ganho de desempenho obtido pelo uso da computação paralela e a qualidade dos resultados. O protótipo possui uma interface *web* e uma interface em modo texto e pode ser utilizado em ambiente de produção.

Para melhor contextualizar o problema e o foco deste trabalho são apresenados no Capítulo 2 uma introdução a biologia molecular, bioinforática e os conceitos de programação paralela. No Capítulo 3 são apresentadas as principais técnicas utilizadas na comparação de sequências e na busca em bases de dados. O Capítulo 4 apresenta a proposta e a metodologia a ser utilizada no desenvolvimento deste trabalho. Neste mesmo capítulo é apresentado o protótipo que implementa as técnicas de indexação estudadas. Na Capítulo 5 são apresentadas os resultados, no Capítulo 6 as conclusões do trabalho.

## 1.1 OBJETIVOS

O objetivo deste trabalho é o desenvolvimento de um protótipo que utilize técnicas de paralelismo combinados com a indexação das sequências genéticas. Além disso é analisada a viabilidade em termos de tempo de processamento, memória consumida e qualidade dos resultados. Sendo assim o principal objetivo deste trabalho é responder a questão se é possível utilizar técnicas de paralelismo na busca de sequências genéticas similares em conjunto com a indexação destes dados obtendo-se resultados satisfatórios em termos de tempo de busca e em qualidade nos resultados.

## 1.2 CONTRIBUIÇÕES

A principal contribuição deste trabalho é a utilização inédita de duas técnicas diferentes para otimizar o processo de busca de sequências genéticas similares, obtendo resultados de boa qualidade. Outras contribuições podem ser listadas como a utilização da plataforma *JAVA* em ambiente de *High Performace Computing* (HPC) e disponibilização de um *software* funcional para busca de sequências genéticas similares utilizando as técnicas descritas.

# 2 CONCEITOS BÁSICOS

Neste capítulo são apresentados os conceitos básicos para compreensão do trabalho. Primeiramente são discutidos os temas de biologia molecular e bioinformática. Em seguida é apresentada uma introdução à computação paralela e por fim as técnicas de comparação e busca de sequências genéticas em base de dados.

## 2.1 BIOLOGIA MOLECULAR E BIOINFORMÁTICA

Nesta seção são apresentados conceitos básicos de biologia, genética molecular e bioinformática necessários para o entendimento deste trabalho. A biologia molecular estuda as reações químicas e as moléculas presentes nos seres vivos.

Todos os seres vivos e vírus possuem suas informações genealógicas transmitidas através de sequências genéticas e estas informações genéticas estão contidas nos seus genomas, que é codificado na forma de RNA ou DNA. Trechos do genoma, os genes, são transcritos para sequências de RNA que servirão de moldes para as proteínas, que são os blocos que constituem os seres vivos.

Estima-se que existam dez milhões - talvez cem milhões - de espécies que atualmente habitam a Terra (ALBERTS, 2004, p. 2). Cada uma destas espécies possui características próprias e meios de reprodução para que a espécie, e consequentemente suas características, sejam perpetuadas. As características de cada espécie são hereditárias e estão armazenadas no seu genoma.

Segundo Alberts et al. (ALBERTS, 2004, p. 2), o **genoma** é a informação genética total carregada por uma célula ou um organismo. No genoma estão todas as caraterísticas genéticas, ou seja, ele contém as informações sobre um organismo. Alberts et al. (ALBERTS, 2004, p. 199) dizem que um **gene** é, normalmente, definido como um segmento de DNA que contém as instruções para produzir uma determinada proteína (ou uma série de proteínas relacionadas) e complementa afirmando que existe uma relação entre a complexidade de um organismo e o número total de genes em seu genoma. Por exemplo, o número total de genes varia entre menos de 500 em bactérias simples a cerca de trinta mil nos humanos. Os genes dos seres vivos estão contidos em moléculas chamadas cromossomos. Os mesmos autores definem o **cromossomo** como uma estrutura composta

por uma molécula de DNA muito longa e proteínas associadas que contém parte, ou toda, informação genética de um organismo. Os seres humanos possuem 23 pares de cromossomos em cada célula do organismo, com exceção das células sexuais, que possuem apenas uma cópia dos 23 cromossomos.

Os genes são segmentos de DNA. O DNA é uma molécula composta por uma sequência de moléculas idênticas, chamadas nucleotídeos. Cada um dos nucleotídeos que formam o DNA é constituído por três partes: um açúcar de cinco carbonos, a desoxirribose, um fosfato e uma base nitrogenada. As bases do DNA podem ser classificadas de acordo com suas propriedades químicas como purinas (a adenina e a guanina) ou como pirimidinas (a citosina e a timina). É importante saber que o DNA é formado por uma fita dupla complementar. Desta forma, cada uma das bases dos nucleotídeos do DNA possui uma base complementar a ela. A base complementar da adenina é a citosina e da guanina é a timina e vice-versa.

Para representar as sequências de DNA, utiliza-se a primeira letra de cada base: $A$ para adenina, $C$ para citosina, $G$ para guanina e $T$ para a timina. Um exemplo de sequência de *DNA* é: $ACTCGGTAC$ e sua sequência complementar é: $CAGATTGCA$.

Strachan e Read (STRACHAN, 2002, p. 1) afirmam que em todas as células, a informação genética está armazenada nas moléculas de DNA. Ele complementa dizendo que regiões específicas das moléculas de DNA servem como moldes para a síntese de moléculas de RNA. O RNA é molecularmente muito similar ao DNA, as diferenças entre eles são duas: o RNA é uma fita simples e a base timina no DNA é substituída pela uracila, $U$, no RNA. As moléculas de RNA são utilizadas direta ou indiretamente para a expressão gênica. A expressão gênica é o processo de leitura dos genes armazenados nos cromossomos, seguido da criação de sequências de RNA baseadas nessas sequências e, por fim, a síntese de proteínas utilizando como moldes estas sequências de RNA criadas a partir da leitura do DNA.

A Figura 2.1 exibe os principais passos da expressão gênica. Observa-se na Figura 2.1 os dois passos para a expressão gênica. O primeiro passo, chamado de transcrição, ocorre no núcleo da célula. Ele consiste na leitura do gene na forma de sequência DNA e na criação de uma sequência de RNA complementar à sequência de DNA. O segundo passo, chamado de tradução, ocorre nos ribossomos da célula e é a síntese de proteínas utilizando como molde a sequência de RNA criada no primeiro passo. O processo de leitura do DNA até a síntese da proteína é conhecido como *dogma central da biologia molecular* e ele segue

o fluxo: DNA → RNA → proteínas (STRACHAN, 2002, p. 9).

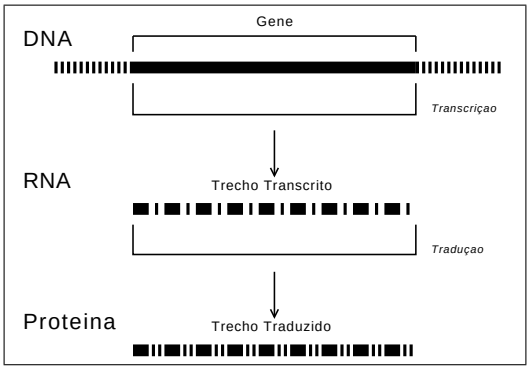


FIG. 2.1: Dogma central da biologia molecular

### 2.1.1 GENÉTICA MOLECULAR

A genética molecular trata de como as informações hereditárias são transmitidas dos seres vivos aos seus descendentes. Como dito na Seção 2.1, as características próprias de cada espécie são hereditárias e estão armazenadas no seu genoma. A importância do genoma está no fato de que, ao mesmo tempo, ele assegura que as informações genéticas serão transmitidas de geração em geração, como também fornece um meio para que ocorram mutações e com isto surgimento de novas características.

Existem diversos tipos de mutações, algumas apenas removem, modificam ou adicionam um nucleotídeo, outras podem remover ou duplicar parte inteira de uma sequência e outras modificar parte da sequência de posição ou invertê-la. Os resultados das mutações podem ser os mais diversos, desde mutações que não afetam o aminoácido que forma a proteína até a duplicação ou remoção do gene que é responsável por determinada proteína.

Estas mudanças que ocorrem no genoma dos organismos são as responsáveis pela sua evolução. Desta forma, remete-se aos conceitos de evolução das espécies publicados pela primeira vez em 1859 no livro "A Origem das Espécies" de Charles Darwin. A Teoria da Evolução, proposta por Charles Darwin, diz que os organismos sofrem mutações entre diferentes gerações e as modificações vantajosas são perpetuadas, enquanto as desvantajosas são eliminadas pela seleção natural.

Alberts et al. (ALBERTS, 2004) afirmam que todas as células executam as operações básicas da mesma forma: síntese de proteínas, hereditariedade, produção de energia. Somando-se isto com a análise dos genes que executam tais operações, pode-se afirmar que todos os seres vivos originam-se de um único ancestral comum. A informação para

estabelecer quais são os parentes mais próximos de cada espécie e sustentar quem poderia ser o último ancestral comum de cada grupo de espécies está contida no seu genoma.

Todos os seres vivos, e consequentemente suas células, desenvolveram-se a partir de um único ancestral. Desta forma, as espécies compartilham semelhanças no genoma. Essas semelhanças traduzem-se em genes e proteínas homólogas (BERNARDES, 2004).

Phillips (PHILLIPS, 2005) explica que o conceito de homologia surgiu primariamente sem relação com o conceito da evolução das espécies e da seleção natural. Phillips informa que Richard Owen introduziu o termo em 1843 para expressar similaridades encontradas em estruturas básicas entre órgãos de animais que ele considerou fundamentalmente mais similar do que outros. Sendo que o termo é derivado da palavra grega *homologia*, que significa "acordo".

A homologia está relacionada com o surgimento de novas características nas espécies, supondo que esta é causada pelas mutações. Conforme detalhado na Seção 2.1, os genes contém as informações para a síntese de proteínas. Durante a existência dos seres vivos, os genes sofrem processos evolutivos, como a duplicação gênica e as mutações. A duplicação gênica é o processo em que a sequência de um gene é copiada uma ou mais vezes no genoma. A Figura 2.2 mostra a ocorrência de dois eventos. Primeiramente um gene $X$ sofre um evento de duplicação, tendo portanto dois genes, $X$ e $X'$. Estes dois genes seguem caminhos evolutivos independentes e o gene $X'$ sofre um evento de mutação, dando origem a um novo gene, o gene $M'$. Os genes $X'$ e $M'$ são genes homólogos, pois possuem um ancestral em comum.

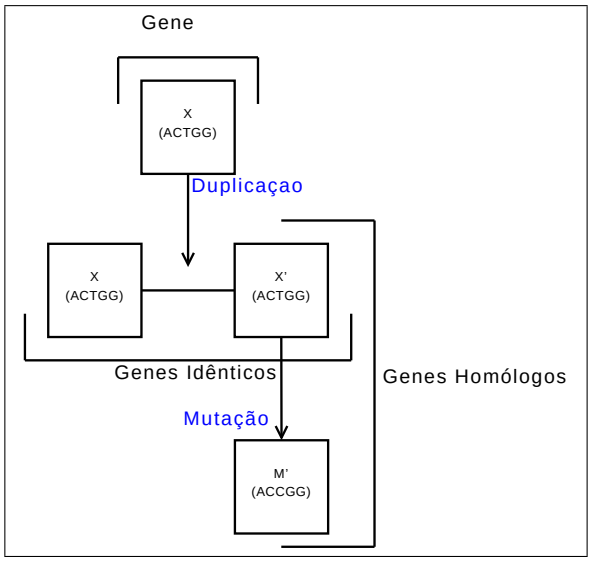


FIG. 2.2: Genes homólogos

O estudo de sequências homólogas é de importância para a identificação e anotação de sequências genéticas e protéicas, para o estudo das relações evolucionárias entre seres vivos e seus componentes, identificação de doenças hereditárias e para o desenvolvimento de fármacos. A homologia entre sequências é refletida em similaridade entre elas, ou seja, as sequências são "parecidas". Desta forma, a busca por sequências similares pode ser utilizada na determinação de sequências homólogas nas bases de dados e para isto, é necessário a comparação e a análise de sequências genéticas.

### 2.1.2 BIOINFORMÁTICA

Com os projetos de sequenciamento de diversos genomas, entre eles o humano, a quantidade de informações nas bases de dados de sequência cresce exponencialmente. Como exemplo do crescimento da quantidade de dados, pode-se verificar na Figura 2.3 o crescimento das sequências de DNA disponíveis no *GenBank* (WHEELER, 2004). O *GenBank* é uma base de dados de sequências nucléicas e protéicas, e informações a respeito delas. Todas as sequências e informações armazenadas nesta base são disponibilizadas livremente.

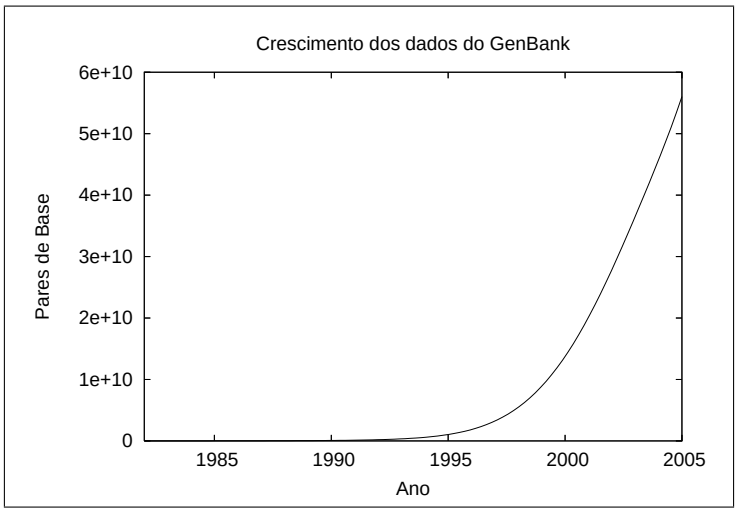


FIG. 2.3: Crescimento da quantidade de dados de sequências no *GenBank*

Os métodos tradicionais de laboratório não são capazes de acompanhar a taxa de crescimento de novas informações. Como consequência, biólogos moleculares passaram a utilizar métodos estatísticos e computacionais capazes de analisar estas grandes quantidades de dados de forma mais automatizada, dando origem a bioinformática. Além de considerar o grande volume de dados depositados nestas bases, deve-se considerar sua

taxa de crescimento

Entre as principais tarefas da bioinformática, pode-se citar: o armazenamento de sequências tanto genéticas quanto protéicas, a busca ou comparação destas sequências, o alinhamento múltiplo de sequências, a predição de genes e a inferência filogenética.

A busca por sequências similares é feita através da comparação entre duas sequências para verificar quantitativamente a similaridade entre elas. Existem diversas formas de comparar sequências genéticas, como a utilização de programação dinâmica, o uso de heurísticas como o BLAST(ALTSCHUL, 1995) e a utilização de modelos probabilísticos, como os modelos ocultos de Markov (ALBRECHT, 2005, p. 145).

Um dos objetivos da busca por sequências similares é poder agrupar sequências homólogas e classificá-las numa família. Esta tarefa da bioinformática é importante porque os testes efetuados em laboratório para classificar uma proteína numa família são caros e demorados (BERNARDES, 2004).

As bases de dados de sequências genéticas e protéicas são depósitos de milhares e até milhões de sequências. Elas se dividem em bases de dados primárias e secundárias. Nas bases primárias os dados são depositados sem grande preocupação em relação a tratamentos específicos como a classificação, anotação e outras formas de curagem. O principal exemplo de base de dados primária é o *GenBank*. Nas bases de dados de sequências genéticas secundárias os dados são curados, classificados e as redundâncias são eliminadas. Como exemplo de bases de dados secundárias pode-se mencionar o *Swiss-Prot*.

Com o aprimoramento das técnicas de sequenciamento, o volume de dados nas bases cresce de forma exponencial, sendo que no ano de 2005, foi anunciado que o número de bases de nucleotídeos de DNA e RNA depositadas em bases de dados públicas superou o valor de cem bilhões (OFHEALTH, 2005). Além de considerar o grande volume de dados depositados nestas bases, deve-se considerar sua taxa de crescimento. Além de considerar um grande volume de dados depositados nestas bases, deve-se considerar também sua taxa de crescimento. No caso particular do *GenBank*, base de dados pública que disponibiliza sequências de nucleotídeos de mais de duzentos mil organismos, seu tamanho dobrou nos útlimos 18 meses. Esta base de dados contém mais de sessenta e cinco bilhões de bases de nucleotídeos em mais de sessenta e uma mil sequências, e quinze milhões de novas sequências foram adicionadas no último ano (BENSON, 2007). Desta forma, são necessárias técnicas que otimizem o tempo de processamento e a quantidade de memória utilizada na busca por sequências similares em bases de dados com tão elevado número

de dados.

## 2.2 COMPUTAÇÃO PARALELA

O uso da computação paralela visa diminuir o tempo total de processamento (*Wall Clock Time*) dividindo o problema em sub-problemas menores e alocando-os em outras unidades de processamento. Nesta seção são apresentados os principais conceitos e informações para quantificar a qualidade da paralelização.

### 2.2.1 MÉTRICAS DE DESEMPENHO

Os ambientes para computação paralela podem ser formados por um processador com vários núcleos, um computador com mais de um processador, um *cluster* computacional ou uma grade computacional. Uma métrica de desempenho muito utilizado na computação paralela é o *speedup*. No contexo de computação paralela,o *speedup* é o ganho de desempenho obtido pela paralelização do processamento entre multiplas unidades de processamento.

Na computação paralela espera-se obter um ganho de desempenho proporcional a quantidade de recursos disponíveis para o processamento. Por exemplo, se com um processador o tempo total de processamento é $t$ segundos, adicionando-se outro processador, espera-se que este tempo caia para $\frac{t}{2}$ segundos. Porém, não é tão simples obter este ganho de desempenho. Uma maneira de ilustrar este problema é considerar que cinco amigos decidiram pintar uma casa com cinco cômodos. Se todos os cômodos possuírem o mesmo tamanho, e se todos pintarem na mesma velocidade, haverá um ganho de cinco vezes. A tarefa torna-se mais complicada se os cômodos possuírem tamanhos diferentes. Caso um cômodos possua o dobro do tamanho, então os cinco pintores não alcançarão o ganho de cinco vezes, porque o tempo total será dominado pelo quarto que necessita de mais tempo para pintar (HERLIHY, 2008).

Analisar o ganho de desempenho deste caso é muito importante para a computação paralela. A fórmula utilizada para calcular o *speedup* é chamada de lei de *Amdahl*. Ela captura a noção de que o ganho de desempenho é limitado pela parte sequencial, não obtendo-se ganho na paralelização. Define-se o *speedup* $S$ de uma tarefa como a razão entre o tempo que um processador demora para completar a tarefa em relação ao tempo de execução em $n$ processadores. A lei de *Amdahl* caracteriza o *speedup* máximo $S$ que pode

ser alcançado por $n$ processadores trabalhando em conjunto numa tarefa: Considerando que $p$ é a fração da tarefa que pode ser executada em paralelo e que a tarefa demora 1 unidade de tempo para ser completada em único processador, com $n$ processadores, a parte paralela demora $p/n$ e a parte sequencial demora $1-p$. Desta forma, a computação paralelizada demora:

$$t = 1 - p + \frac{p}{n} \tag{2.1}$$

A lei de *Amdahl* diz que o *speedup* é a proporção entre o tempo sequencial e o tempo paralelo, então o *speedup* pode ser definido como:

$$S = \frac{1}{1 - p + \frac{p}{n}} \tag{2.2}$$

Para ilustrar as implicações da lei de *Amdahl*, considerando-se o exemplo da pintura dos quartos e assumindo que cada quarto pequeno é uma unidade e o quarto maior são duas unidades. Determinando um pintor, ou processador, por quarto, significa que 5 das 6 unidades podem ser pintadas em paralelo, implicando que $p = 5/6$ e $1 - p = 1/6$. A lei de *Amdahl* determina que o *speedup* resultante é:

$$S = \frac{1}{1 - p + \frac{p}{n}} = \frac{1}{1/6 + 1/6} = 3$$

Interessante que mesmo com cinco pintores, o ganho de desempenho é apenas 3 e isto foi ocasionado por apenas um quarto com o dobro de tamanho. Porém o resultado poderia ser pior. Imaginando-se uma casa com dez quartos e dez pintores, onde cada pintor é responsável por um quarto, porém um dos quartos possui o dobro do tamanho. O resultado do *speedup* seria:

$$S = \frac{1}{1/11 + 1/11} = 5,5$$

Novamente, um pequeno desequilíbrio no tamanho das tarefas e o ganho do desempenho é aproximadamente metade de que alguma análise ingênua esperaria. A solução seria que quando um pintor terminasse de pintar o seu quarto, ele auxiliaria o quarto que está faltando, porém para isto é necessário coordenação entre os pintores. Isto demonstra o que a lei de Amdahl diz: se uma pequena parte parte do problema não for paralelizado, o ganho de desempenho será muito inferior. Utilizando-se como exemplo um computador com dez processadores, caso se paralelize 90% de um problema, haverá um ganho de 5 vezes e não 10 vezes como esperado. Em outras palavras, os 10% restantes diminuem

a utilização do computador pela metade. Desta forma, deve-se investir esforços para paralelizar os 10% restantes. Infelizmente esta paralelização será mais difícil, pois deverá envolver diversas questões de comunicação e coordenação.

### 2.2.2 PROCESSADORES MULTI-THREADING

Neste trabalho são utilizado os *Chip Multi-Threaded* (CMT), que são processadores que provem suporte a execução de diversas *threads* em *hardware* simultaneamente de diversas formas, incluindo o *Simultaneous Multithreading* (SMT) e o *Chip Multiprocessing* (CMP) (SPRACKLEN, 2005). Entre os CMT pode-se citar o *Niagara* (KONGETIRA, 2005) que possui 8 CMP e cada um com 4 SMT, totalizando até 32 *threads* sendo executadas de forma simultânea. Outros exemplos são os processadores "*dual*" e "*quad*" *core* da *Intel* e *AMD*. Porém grande parte destes processadores são apenas CMP, não sendo SMT, pois eles possuem mais de um núcleo cada um, porém cada núcleo é capaz de executar apenas uma *thread* por vez. A tecnologia *hyper-threading* da *Intel* é uma tecnologia que utiliza SMT, porém sem CMP, desta forma um único núcleo é capaz de executar mais de uma *thread* simultaneamente, simulando para o sistema operacional existir mais de um núcleo de processamento.

A importância de se conhecer e estudar como melhor utilizar estes processadores é que durante décadas os aplicativos obtiveram um ganho de desempenho pelo aumento da velocidade do processadores, ocorrida principalmente pelo aumento do seu *clock*. Porém a energia utilizada por estes processadores era muito alta e a tendência era de necessitar de cada vez mais energia. Neste ponto modificou-se para o uso de mais de um núcleo nos processadores, porque podem oferecer desempenho igual ou superior , mas com menor consumo de energia, já que o *clock* individual de cada processador é menor do que dos antigos processadores com apenas um núcleo (INTEL, 2008).

Na Figura 2.4 é exibido um gráfico mostrando a diferença no desempenho quando são utilizadas técnicas de paralelismo. Até o fim da "era *Ghz*", como foi chamada a época em que os processadores aumentavam seu desempenho com o aumento do *clock*, a não utilização de paralelismo não causava problemas em relação ao desempenho. Com o surgimento dos processadores CMP, ou *multi-core* como exibido na figura, a diferença no desempenho sem a utilização de paralelismo fica evidente e com o passar do tempo esta diferença tende a aumentar.

Para utilizar os núcleos de processamento extras, os *softwares* devem ser adaptados

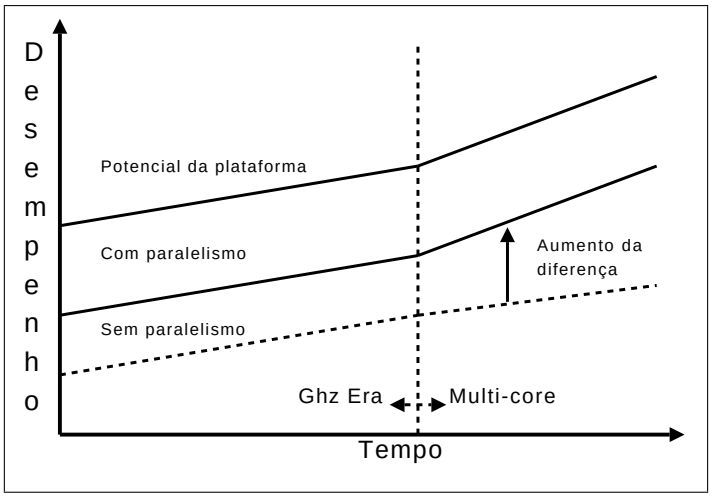


FIG. 2.4: Diferença do desempenho quando utilizado paralelismo. (INTEL, 2008)

para tornar possível paralelizar a sua execução. Esta não é uma tarefa trivial e muitos algoritmos devem ser modificados para a obtenção dos ganhos. Porém como os processadores CMT são uma tendência, deve-se estudar e aplicar as técnicas para melhor utilizar esta capacidade computacional.

# 3 TÉCNICAS PARA COMPARAÇÃO E BUSCA DE SEQUÊNCIAS EM BASES DE DADOS

Neste capítulo são apresentas as principais técnicas utilizadas pelas ferramentas de busca por sequências similares. Primeiramente são introduzidas algumas definições importantes. Em seguida é apresentada a técnica de programação dinâmica, que é a base para o desenvolvimento de outras técnicas. Em seguida são apresentados algoritmos que utilizam heurísticas, como o BLAST e o *FASTA*. Logo após, são apresentadas técnicas para otimizar as heurísticas, como o uso de índices, a distribuição dos dados e a paralelização da busca.

## 3.1 DEFINIÇÕES BÁSICAS

Uma **sequência** é uma sucessão finita de símbolos pertencentes a um certo conjunto $\Sigma$ (o alfabeto). Uma sequência biológica é uma sequência onde $\Sigma = \{A, C, G, T\}$ (DNA), $\Sigma=\{A, C, G, U\}$ (RNA) ou $\Sigma$ é formado por aminoácidos, $\Sigma=\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$ (sequências peptídicas). Uma **sub-sequência** é um sequência que está contida parcialmente ou totalmente em outra sequência. Por exemplo, a sequência de DNA $ACGTCCT$ é uma sub-sequência da sequência $TTACGTCCTA$, ou outro exemplo, a sequência $ACG$ é uma sub-sequência de $ACG$. Segundo Meidanis e Setúbal (MEIDANIS, 1994), um **alinhamento** é a inserção de buracos em pontos arbitrários ao longo das sequências, de modo que elas fiquem com o mesmo comprimento. Além disso, não é permitido que um "buraco" numa das sequências esteja alinhado com outro buraco na outra, onde cada coluna do alinhamento receberá um certo número de pontos e a pontuação total do alinhamento será a soma dos pontos atribuídos às suas colunas. Desta forma, o alinhamento ótimo será aquele que tiver **pontuação máxima** e esta pontuação máxima será chamada de **similaridade** entre as duas sequências. O alinhamento pode ser global, onde busca-se alinhar inteiramente duas sequências, ou local, onde o objetivo é salientar as similaridades pontuais entre duas sequências. Desta forma, as buscas por sequências homólogas são pesquisas por sequências mais similares, ou seja, que possuem as maiores pontuação. Na Figura 3.1 é apresentado um exemplo de alinhamento de duas sequências genéticas pela adição de um buraco.

```
GA-CGGATTAG
GATCGGAATAG
```

FIG. 3.1: Exemplo de alinhamento de sequências

O termo utilizado para descrever a qualidade dos resultados das ferramentas de busca de sequências similares é a sensibilidade. **Sensibilidade** é a capacidade de encontrar similaridade e construir alinhamentos com pares de sub-sequências com menor grau de similaridades entre si. Quanto menor for o grau de similaridade entre um par de sub-sequências que a técnica é capaz de identificar, maior é a sua sensibilidade.

Heurísticas como o *FASTA* (PEARSON, 1988) e o BLAST utilizam o termo *High Score Pair* (HSP) em suas técnicas de busca. Uma **HSP** é um par de sub-sequências chamadas de sementes que, quando alinhadas, resultam numa pontuação acima de um limite estabelecido, determinando uma alta similaridade entre eles. Nas técnicas de busca, as HSP são utilizadas como trechos iniciais de alinhamentos mais longos.

## 3.2 PROGRAMAÇÃO DINÂMICA

Na busca por sequências similares é necessário calcular a similaridade entre a sequência de entrada e as sequências armazenadas na base de dados. O procedimento primário para o alinhamento de sequências é o algoritmo de *Needleman-Wunsch* (NEEDLEMAN, 1970). Este algoritmo calcula a distância mínima entre duas sequências, isto é, o número mínimo de operações de transformação requeridas para converter uma sequência em outra. As duas operações básicas são: a inclusão de um ou mais espaços consecutivos, chamados de buracos e a substituição de elementos individuais. Cada uma destas operações gera um custo previamente definido.

O algoritmo de *Needleman-Wunsch* é da classe dos algoritmos de programação dinâmica (CORMEN, 2001). Nesta classe de algoritmos, o problema é dividido em subproblemas e os resultados da execução dos subproblemas são armazenados e reutilizados. O algoritmo de *Needleman-Wunsch* consta de 3 fases: inicialização da matriz $S$, cálculo das células da matriz $S$ e determinação do alinhamento entre as duas sequências. Na primeira fase do algoritmo é construída e inicializada uma matriz contendo $m + 1$ colunas e $n + 1$ linhas, sendo $m$ o comprimento da sequência de entrada ($M$) e $n$ o comprimento da sequência a ser comparada ($N$). Na primeira linha os valores de cada célula são calcula-

dos como $S(i,0) = i$ e na primeira coluna como $S(0,j) = j$, considerando $i$ a linha atual e $j$ a coluna atual e que a célula $(0,0)$ não é utilizada. A Figura 3.2 exibe a primeira fase do algoritmo *Needleman-Wunsch* utilizando como entrada as sequências $TTTCCAAGGC$ e $TTTCAGCC$.

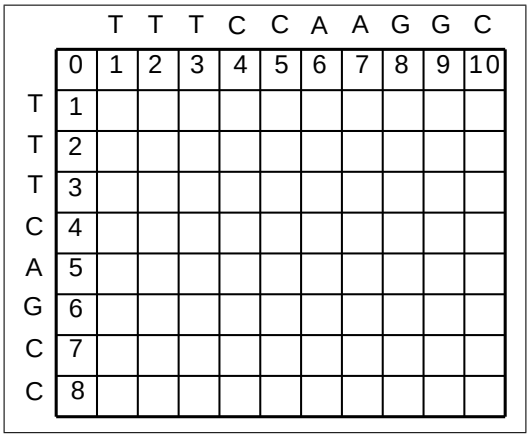


FIG. 3.2: Primeira fase do algoritmo de alinhamento de sequências Needleman-Wunsch

A segunda fase começa uma vez concluída a inicialização da primeira linha e a primeira coluna da matriz com o cálculo das células restantes ordenadamente, por linha. Este cálculo verifica o valor das células $(i-1,j)$, $(i,j-1)$ e $(i-1,j-1)$. A pontuação é cálculada escolhendo a opção mais barata entre a inserção de um buraco ou a pontuação de um encontro ou desencontro, conforme exibido na Figura 3.3. A pontuação para a inserção de um espaço e para um desencontro é normalmente negativa, salvo em alinhamento de sequências protéicas, onde mesmo dois aminoácidos sendo diferentes podem resultar numa pontuação positiva, devido às suas características físico-químicas semelhantes.

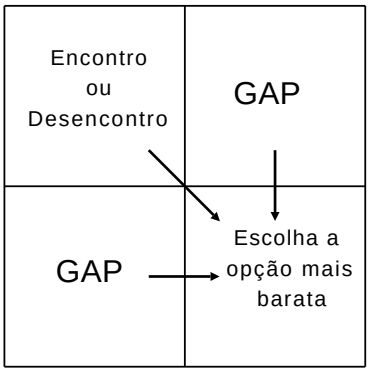


FIG. 3.3: Escolha do valor a ser utilizado na célula

O algoritmo de *Needleman-Wunsch* calcula o valor de cada célula utilizando os dados previamente calculados e armazenados na memória. Desta forma, o algoritmo pode ser

visto como uma recorrência matemática conforme apresentado na Equação (3.1). Sendo que $\delta(M_i, N_j)$ é a função da comparação entre o elemento $M_i$ e $N_j$, normalmente retornando uma pontuação positiva em casos de bases iguais ou aminoácidos similares e uma pontuação negativas para bases diferentes e aminoácidos com pouco similaridade.

$$S_{i,j} = min \begin{cases} S_{i-1,j} - BURACO, \\ S_{i,j-1} - BURACO, \\ S_{i-1,j-1} + \delta(M_i, N_j) \end{cases} \tag{3.1}$$

Na Figura 3.4 é exibida a segunda fase do algoritmo que consiste em calcular todos os valores das células da matriz. Neste exemplo, é utilizado valor de 1 para o $\delta$. As setas indicam o caminho ótimo do alinhamento para a próxima célula e as células com fundo cinza representam o estágio atual do alinhamento e a pontuação de cada célula.

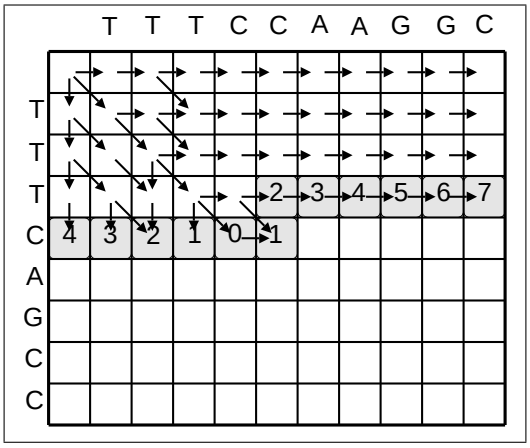


FIG. 3.4: Segunda fase do algoritmo de alinhamento de sequências

Na Figura 3.5 é exibido um trecho da matriz de alinhamento utilizada no exemplo. Nesta figura são exibidos os valores atuais do alinhamento, destancado em cinza as células que foram utilizadas por possuírem o melhor valor.

A terceira e última fase do algoritmo, exibida na Figura 3.6, é conhecida como *trace-back*, que significa percorrer o caminho de volta. Ele é iniciado na célula final $(m+1, n+1)$, que contém a pontuação correspondente ao alinhamento global ótimo e e é verificado em cada célula, a célula que foi utilizada para calcular o seu valor e assim executa-se recursivamente até chegar na célula inicial. Uma outra abordagem é: no momento do cálculo do valor da célula armazenar um ponteiro para a célula ou as células que foram utilizadas para calcular o valor.

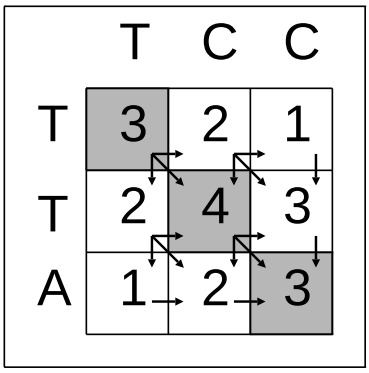


FIG. 3.5: Escolha do valor a ser utilizado na célula

Para apresentar o resultado do alinhamento, segue-se as células que foram utilizadas para o cálculo da última célula do alinhamento. Caso seja um encontro ou desencontro, os dois elementos que representam aquela célula são colocados no alinhamento e caso seja um buraco, um *"-"* que representa este espaço é adicionado na sequência. Assim, o resultado do alinhamento é construído de trás para frente, sendo necessário invertê-lo ao fim.

Como visto na Figura 3.6, podem existir mais de uma solução para o alinhamento de duas sequências. Na Figura 3.7 são apresentadas as quatro possíveis soluções deste alinhamento conforme a pontuação dada de mais um (+1) para desencontros e buracos e zero (0) para encontros. Percebe-se que todas as quatro soluções possuem o mesmo número de acertos, que são representados pelos pontos entre as duas sequências alinhadas.

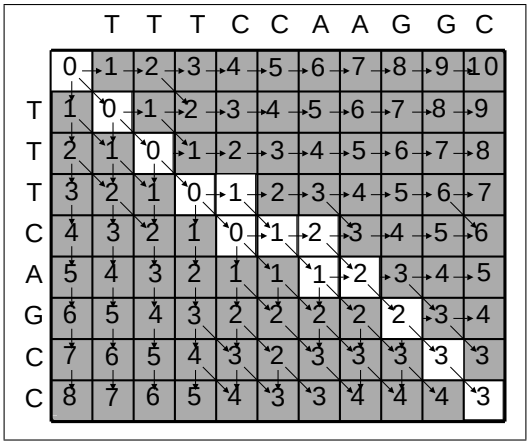


FIG. 3.6: terceira fase do algoritmo de alinhamento de sequências

O algoritmo apresentado anteriormente determina um alinhamento global, ou seja, procura alinhar as duas sequências na sua totalidade. Variações deste algoritmo são

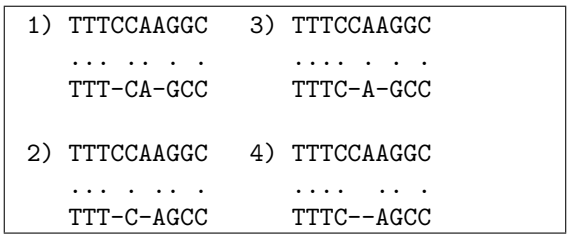


FIG. 3.7: Resultados do alinhamento de sequências

utilizados para alinhar grandes sequências, como sequências cromossômicas e até mesmo genomas. Porém, na busca por sequências similares é interessante ressaltar um outro tipo de alinhamento. O algoritmo de *Smith-Waterman*(SMITH, 1981) é base para os principais algoritmos de alinhamento local. A diferença deste algoritmo com o *Needleman-Wunsch* é que existe um limite para o valor da célula. Por exemplo, utilizando um sistema de pontuação que bonifica em mais uma unidade (+1) os acertos e com penalidade de menos uma unidade (−1) para desencontros e buracos, o limite inferior da pontuação será zero. Desta forma, os trechos que não são similares não prejudicam o alinhamento dos trechos similares e assim consegue-se obter os locais onde há maior similaridade entre as duas sequências alinhadas. O algoritmo pode ser simplificado como a recorrência matemática apresentada na Equação (3.2). Notam-se duas principais diferenças com o algoritmo de *Needleman-Wunsch*: neste algoritmo procura-se maximizar a pontuação, já que a pontuação de acertos é positiva e existe a opção do valor 0 na recorrência, significando que jamais uma célula possuirá valor menor. Após o cálculo dos valores das células pelo algoritmo de *Smith-Waterman*, o processo de *traceback* é idêntico ao utilizado pelo algoritmo de *Needleman-Wunsch*.

$$S_{i,j} = max \begin{cases} S_{i-1,j} - BURACO, \\ S_{i,j-1} - BURACO, \\ S_{i,j} + \delta(M_i, N_j), \\ 0. \end{cases} \tag{3.2}$$

## 3.3 HEURÍSTICAS PARA BUSCA DE SEQUÊNCIAS SIMILARES

Os algoritmos *Smith-Waterman* e *Needleman-Wunsch* são respectivamente os algoritmos mais sensíveis para alinhamento global e local, porém possuem uma alta complexidade de tempo e de espaço, ambas iguais a $O(mn)$. Estes custos tornam seu uso para a busca de sequências similares em bases de dados impraticável. Considerando-se esta alta complexidade computacional, foram desenvolvidos algoritmos que utilizam heurísticas para a busca de sequências similares.

Partindo do problema do alto custo computacional da programação dinâmica, foram desenvolvidos algoritmos que diminuem a quantidade de memória e processamento necessários para comparar sequências. Foi desenvolvido primeiro o algoritmo e o software homônimo *FASTA* (PEARSON, 1988). O *FASTA* trabalha basicamente buscando sub-sequências idênticas às sub-sequências da sequência de entrada e gerando uma HSP, conforme descrito na Seção 3.1, com estes pares. Em seguida é feita a extensão e após o alinhamento.

A **extensão** é um fase onde tenta-se aumentar o comprimento das sub-sequências que formam o HSP em ambas as direções. Para isto, primeiramente tenta-se estender para esquerda. Para cada acerto incrementa-se uma variável que possui a pontuação, para cada desencontro, diminui-se a pontuação. Se a pontuação mais alta menos a pontuação atual for inferior a um limite, a extensão é finalizada e as sequências passam a iniciar na posição com maior pontuação. O mesmo processo é executado na extensão para a direta, encontrando assim um nova posição final para as sequências de entrada e da base de dados da HSP. A Figura 3.8 exibe o processo de extensão. Primeiramente é exibido as duas sequências com suas áreas que foram a HSP destacadas. A seguir é feita a extensão a esquerda, sendo que as novas bases estendidas estão destacadas. Logo após as sequências são estendida a direita e por fim é exibido o resultado final, que será utilizado na fase de alinhamento.

Após a extensão, o *FASTA* utiliza a programação dinâmica para construir o alinhamento unicamente entre as partes similares das sequências. Após o *FASTA*, foi desenvolvido o método e software *BLAST*(ALTSCHUL, 1995), que também constrói HSP para então efetuar o alinhamento usando programação dinâmica.

O BLAST permite que as sementes das sequências comparadas sejam diferentes, porém que possuam um alto valor no alinhamento entre elas. Um diferencial do BLAST é a disponibilização de um valor estatístico, chamado *E-Value*, que descreve a probabilidade

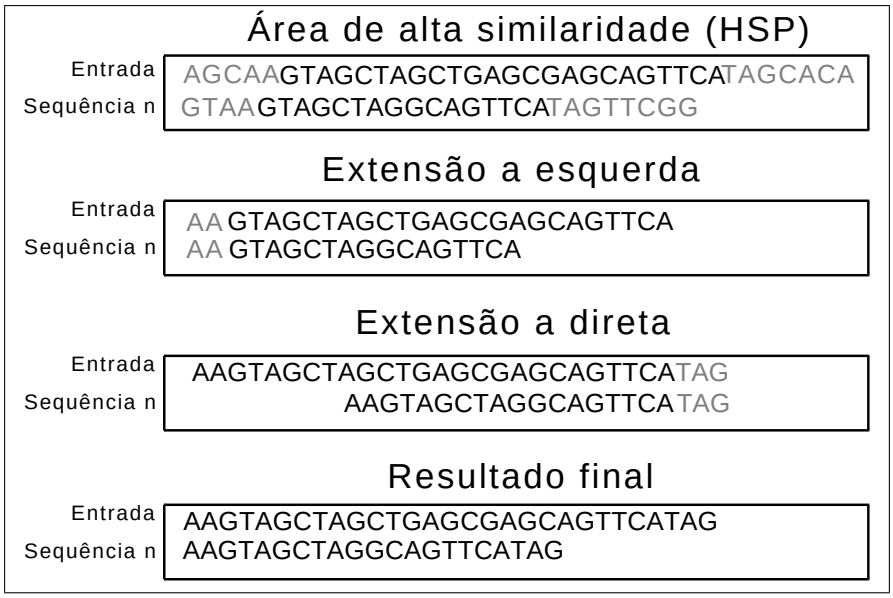


FIG. 3.8: Extensão de uma HSP em ambas as direções.

do alinhamento ter ocorrido ao acaso, ou seja, quanto menor é o *E-Value*, mais provável que haja uma homologia entre as sequências alinhadas.

O *E-Value* do BLAST é calculado pela equação $E = kmn^{-\lambda S}$ chamada de equação de *Karlin-Altschul*. Esta equação informa a quantidade de alinhamentos esperados ($E$) a partir do espaço de de busca ($mn$), onde $m$ é o tamanho efetivo da base de dados e $n$ o tamanho efetivo da sequência de entrada. Tamanho efetivo é o tamanho real menos o comprimento mínimo dos HSPs. A equação também utiliza uma constante $k$ e a pontuação normalizada $\lambda S$ que é baseada no esquema de pontuação para encontros e desencontros nos alinhamentos (KORF, 2003). Desta forma o *E-Value* é muito utilizado na busca de sequências similares pois com ele pode-se saber se o alinhamento encontrado foi ao acaso, *E-Value* alto, ou uma real similariedade foi encontrada, *E-Value* baixo.

Resumidamente, o algoritmo BLAST é uma método heurístico de busca por sementes de tamanho $W$ (sendo o padrão 3 para proteínas e 11 para sequências genéticas) que tenham uma pontuação igual ou superior ao limite $T$ quando alinhadas com a semente da sequências de entrada e pontuada com uma matriz de substituição, que será comentada posteriormente. As sementes na base de dados com pontuação $T$ ou superior são estendidas em ambas as direções na tentativa de encontrar um alinhamento ótimo sem buracos ou um par de alta pontuação (HSP) com uma pontuação de ao menos $S$ e um *E-Value* menor que um limite especificado. As HSPs que satisfazem este critério serão reportadas pelo BLAST, contanto que não ultrapassem a linha de corte da quantidade de

alinhamentos a serem reportados.

Na Figura 3.9 é apresentado um esquema do funcionamento do algoritmo BLAST, onde obtêm-se uma palavra de tamanho $W$ e geram-se todas as possíveis palavras de tamanho $W = 3$. No primeiro passo é verificado nas sequências da base de dados, onde essas palavras ocorrem. Caso encontre, como no exemplo da Figura 3.9(a) onde é encontrada a palavra $PMG$, busca-se continuar o alinhamento em ambas as direções. Após isto, na Figura 3.9(b) desloca-se uma posição na sequência e uma nova busca é executada. Este deslocamento é chamado de **janela deslizante** (*sliding width*) e busca encontrar todas as palavras de tamanho $W$ presentes na sequência.

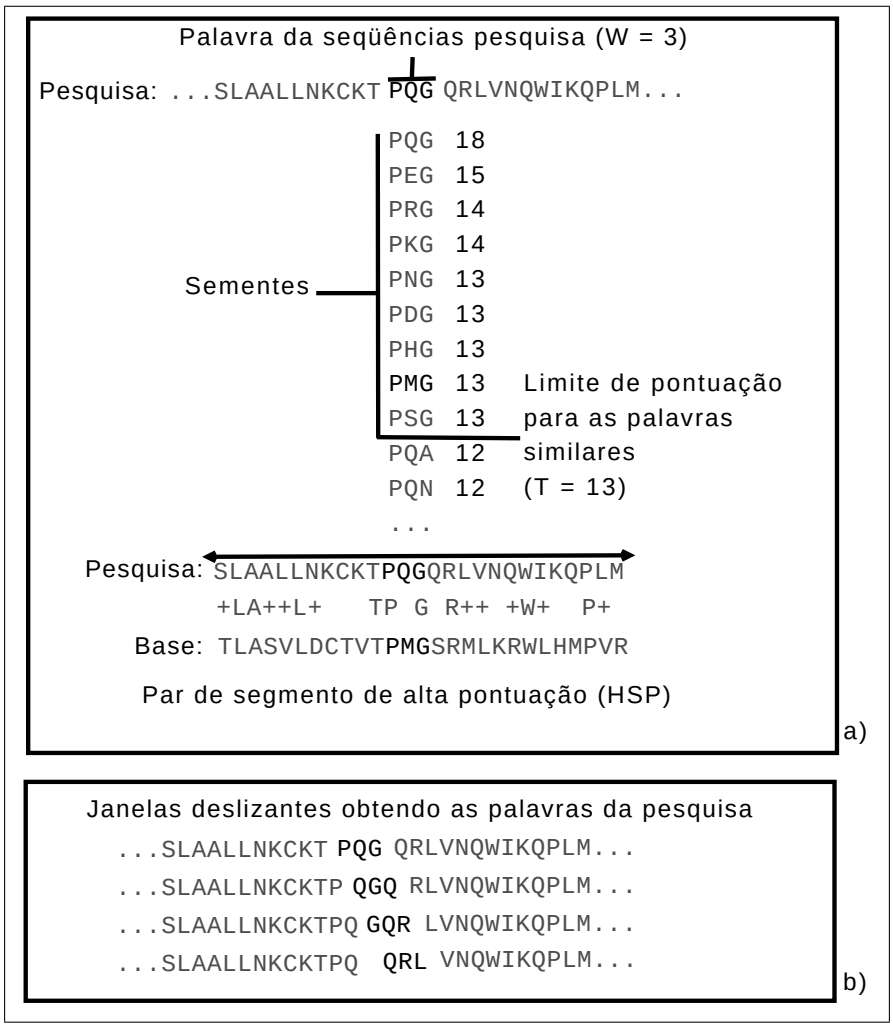


FIG. 3.9: Algoritmo BLAST pesquisando por um HSP

Porém, existe um problema na primeira versão do BLAST, ele não permitia o uso de buracos no alinhamento das sequências. Este problema foi solucionado na sua versão mais recente (ALTSCHUL, 1997), onde os alinhamentos entre as sementes são feitos utilizando

programação dinâmica.

O *FASTA* e o BLAST otimizaram o custo da execução do alinhamento, sendo que o alinhamento só é construído entre trechos que possuam uma similaridade previamente detectada. Porém a complexidade do problema continua sendo de $O(nmq)$, sendo $q$ a quantidade de sequências na base de dados. Isto ocorre, porque para cada sequência na base de dados, devem ser achadas as posições com alta similaridade para, se necessário, efetuar o alinhamento. Uma solução para reduzir o custo computacional é a criação de índices para facilitar o acesso direto aos trechos de alta similaridade.

Em (CAMERON, 2007) é apresentada uma técnica de busca e alinhamento de sequências similares de modo análogo ao BLAST, porém ela utiliza as sequências compactadas em todos os estágios da busca. Esta técnica foi implementada no *FSA-BLAST*, *software* similar ao BLAST, porém, segundo os autores, graças às otimizações utilizadas, chega a ser duas vezes mais veloz que o BLAST original. A compactação é feita utilizando 2 *bits* para cada base da sequência, sendo 00 para a $A$, 01 para $C$, 10 para $G$ e 11 para $T$. Desta forma, quatro bases ficam armazenadas num único *byte* como um vetor de *bits*, diminuindo o uso de memória em aproximadamente 4 vezes. Na Figura 3.10 é apresentado um exemplo de compactação de sequências feita pelo BLAST e *FSA-BLAST*. Neste exemplo é compactada a sequência *ATGACNTGCG*. O $N$ presente nesta sequência é um caractere coringa, onde qualquer outro caractere pode estar nesta posição. Ele é trocado por um caractere já existente para manter o alfabeto em 4, utilizando-se 2 *bits*, e caractere $A$ foi escolhido por motivo de simplicidade. Isto porque segundo (CAMERON, 2007), mais de 99% das bases armazenadas no *GenBank* são uma das quatro bases normais e 98% das bases coringas são o caracter especial $N$, que representa todas as bases. Ou seja, a diminuição da sensibilidade pelo não armazenamento destes caracteres coringa não é perceptível. Outra vantagem da compactação é a comparação das sequências, pois com a codificação delas é possível comparar 4 bases de cada vez.

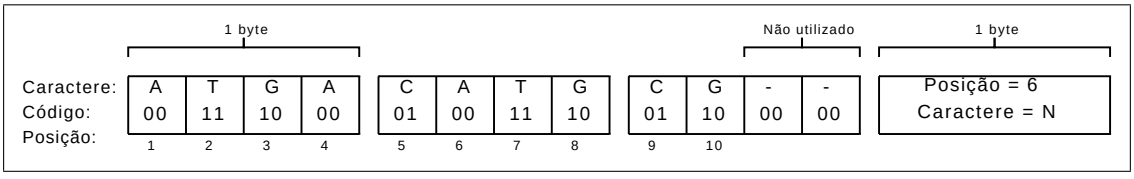


FIG. 3.10: Codificação das sequências feita pelo BLAST e *FSA-BLAST*.

## 3.4 ÍNDICES NA BUSCA DE SEQUÊNCIAS SIMILARES

Índices são estruturas de dados que permitem obter informações sobre a localização dos dados indexados de forma mais eficiente do que por uma busca linear, otimizando a busca pelos dados indexados. Eles são utilizados na área de Recuperação de Informações (*information retrieval*), principalmente em ferramentas de busca *web* como o *Google* e *Yahoo* e bibliotecas de ferramenta de busca de textos como o *Lucene* (FOUNDATION, 2009a). Os índices utilizados nos métodos de busca de sequências similares são índices invertidos. Eles armazenam as localizações de sub-sequências na base de dados e assim eliminam a necessidade de localizá-las através de uma pesquisa linear.

Duas estruturas de dados são normalmente utilizadas para a criação de índices para a busca de sequências similares: as árvores de sufixo e vetores. Em Gusfield (GUSFIELD, 1997) são utilizadas árvores de sufixo, uma estrutura de dados que permite acessar a posição de uma sub-sequência em tempo linear. Segundo Gusfield (GUSFIELD, 1997), as árvores de sufixo podem ser usadas para resolver problemas de casamento de sequências exatas em tempo linear, porém, o autor diz, que sua real virtude é a realização de tarefas mais complexas em tempo linear, como obtenção de todas as sequências repetitivas e obtenção do ancestral comum mais longo. Gusfield define que uma árvore de sufixos $T$ para uma cadeia de $m$-caracteres $S$ é uma árvore direcionada com raiz e com exatamente $m$ folhas numeradas de 1 a $m$. Cada nó interno, com exceção da raiz, tem ao menos dois filhos e cada aresta é rotulada com uma sub-cadeia não vazia de $S$. Duas arestas fora de um nó não podem ter o mesmo rótulo começando com o mesmo caracter. A característica principal de uma árvore de sufixo é que para qualquer folha $i$, a concatenação dos rótulos dos vértices no caminho da raiz até a folha $i$ descreve exatamente o sufixo de $S$ que inicia na posição $i$, isto é, que se escreve $S[i..m]$. Normalmente é utilizado o símbolo \$ para informar o fim da cadeia, então, a cadeia pode ser representada como $S\$$.

Um exemplo de aplicação das árvores de sufixo é apresentado por Giladi et al. (GILADI, 2001), onde um algoritmo para a construção de árvores para busca de sequências quase-exatas é utilizado. Porém, o ganho no tempo de acesso a sub-sequências tem um alto custo em relação ao consumo de memória. Ning et al. (NING, 2001) justifica isto dizendo que no trabalho de Delcher et al. (DELCHER, 1999) há um gasto de 37 *bytes* por base na sua implementação utilizando árvores de sufixo. Como comparação, para o genoma humano, com aproximadamente três bilhões de pares de bases, são necessários

103 *gigabytes* de memória. Delcher et al. (DELCHER, 2002) apresentam uma otimização em relação ao seu trabalho anterior, onde o algoritmo para alinhamento de genomas utilizando árvores de sufixo, é três vezes mais veloz que a versão anterior e utiliza um terço de memória. Mesmo com essa redução, a quantidade de memória necessária para armazenar a árvore de sufixos do genoma humano é em torno de 34 *gigabytes*.

Um trabalho chamado *ESAsearch*, sobre o uso de índices para a localização de subsequências é descrito em (BECKSTETTE, 2006). Neste trabalho, a estrutura de dados utilizada para índices, são os *arrays* de sufixo que auxiliam a localização de trechos similares e depois o alinhamento é construído com o uso de *Position Specific Scoring Matrices* (PSSMs) (GRIBSKOV M, 1987). O trabalho de Beckstette et al. (BECKSTETTE, 2006) apresenta um algoritmo com boa sensibilidade e desempenho, porém poucas informações são dadas sobre o consumo de memória.

Uma alternativa para a criação de índices é a utilização de vetores, também chamados de índices invertidos no contexto da área de Recuperação de Informações. Estes vetores são bidimensionais, onde cada linha indica onde determinada palavra é encontrada no texto. A vantagem da utilização de vetores é o acesso a todas as localizações de determinada palavra em tempo $O(1)$.

Entre as técnicas para busca de sequências similares com a utilização de índices invertidos podemos citar: o *Sequence Search and Alignment by Hashing Algorithm* (SSAHA) (NING, 2001), o *BLAST-Like Alignment Tool* (BLAT) (KENT, 2002), *PatternHunter* (MA, 2002) o *miBLAST* (KIM, 2005), *Megablast* (TAN, 2005) e *MegaBlast* (MORGULIS, 2008).

O SSAHA (NING, 2001) é baseado na organização da base de dados de DNA numa tabela *hash* e no fato de que o computador possui memória *RAM* suficiente para armazenar toda esta tabela. O SSAHA utiliza uma **janela deslizante não sobreposta** (este conceito é apresentado no Capítulo 4) com um comprimento $n$ e percorre todas as sequências da base de dados obtendo as suas sub-sequências não sobrepostas. Então é calculado o valor *hash* da sub-sequência e armazenado nesta posição da tabela a posição na sequência e em qual sequência a sub-sequência se encontra. A tabela *hash* do SSAHA funciona como um índice invertido, permitindo saber em tempo $O(1)$ as localizações das sub-sequência de tamanho $n$. Porém, o SSAHA, não possui uma boa sensibilidade, ou seja, não encontra similaridade entre pares de sequências com baixa similaridade. Isto ocorre porque ele inicialmente pesquisa sub-sequências idênticas e não similares e a indexação de **sequências não sobrepostas** (este conceito é apresentado no Capítulo 4), ao invés de sequências so-

brepostas, diminui a sensibilidade deste método. Segundo os autores, esta característica ocorre porque o SSAHA foi desenvolvido inicialmente para a montagem de sequências pós-sequenciamento e para a detecção de polimorfismo em único nucleotídeo.

O BLAT (KENT, 2002) é muito similar ao *BLAST*, ambos pesquisam por sementes iguais na sequência de entrada e nas sequências da base de dados e então estendem estes encontros em HSPs. Porém o BLAT difere do BLAST porque ele constrói um índice invertido percorrendo todas as sequências de forma não sobreposta da base de dados e armazena este índice invertido na memória e o *BLAST* retorna cada área homóloga entre duas sequências como alinhamentos separados enquanto o BLAT retorna como uma única área mais longa. Uma melhoria do BLAT em relação ao SSAHA é que ele necessita de 5 bytes para cada entrada no índice, enquanto o SSAHA necessita de 8 bytes. Segundo (JIANG, 2007) o BLAT lida bem com sequências muito longas na base de dados, porém não é recomendável sequências de entrada com mais 200.000 bases de comprimento. (JIANG, 2007) informa que o BLAT é 500 vezes mais veloz que o BLAST para entradas de sequências de Ácido Ribonucléico mensageiro (mRNA) e DNA e 50 vezes mais veloz para proteínas e o espaço requerido para armazenar os índices é de aproximadamente 30% do tamanho da base de dados.

Na mesma linha de funcionamento do SSAHA e do BLAT tem-se o *PatternHunter* (MA, 2002). Nele também é construído um índice invertido e então são buscadas as posições das sub-sequências, as sementes são estendidas e então é construído um alinhamento entre a área da sequência de entrada e a área da sequência da base de dados que possui a alta similaridade. O *PatternHunter* possui dois diferenciais em relação ao SSAHA e ao BLAT: ele é implementado em Java, enquanto os outros dois são implementados em C++ e o mais importante, o uso de máscaras nas sub-sequências para otimizar a sensibilidade no processo de armazenamento e busca no índice. As **máscaras** utilizadas pelo *PatterHunter* são sequências de zeros e uns, onde o zero significa que a base naquela posição deve ser ignorada. Por exemplo: aplicando a máscara “01101101” na sequência “AGTCCAGT” terá a sequência resultante “GTCAT” e esta nova sequência será armazenada no índice. O ganho na sensibilidade se dá porque caso na sequência de entrada haja a sub-sequência “CGTACACAT” ela será encontrada no índice porque aplicando a máscara nela, ela ficará “GTCAT”. Desta forma, pode-se indexar sub-sequências mais longas, que normalmente diminuem a sensibilidade do algoritmo de busca e mesmo assim encontrar similaridades entre elas, ou seja, utilizando máscaras não se busca no índice sequências exatas, mas

sequências similares à sequência de entrada. Em (MA, 2002) é afirmado que o *PatternHunter* possui melhor qualidade nos resultados, é vinte vezes mais rápido e necessita um décimo da memória em relação ao BLAST e também faz buscas com sequências mais longas, com 19 milhões de bases, algo que o BLAST não é capaz.

O *miBLAST* (KIM, 2005) armazena no índice apenas em quais sequências as subsequências estão, não armazenando suas posições. Por esta característica ele é melhor aplicado na busca de sequências curtas de 20 a 128 bases de comprimento, acima disto o ganho de tempo em relação ao BLAST é muito baixo. Um problema com o *miBLAST* é o espaço necessário para o índice. Para a base de dados *Human UniGene* contendo 3,19 gigabytes de dados, usando o valor de 11 bases para o comprimento das sub-sequências indexadas, são necessários 11,94 *gigabytes* de espaço para armazenar o índice.

O *MegaBLAST* (ZHANG, 2000) é uma versão aprimorada do BLAST que acelera as buscas e melhora o rendimento total do *software* graças a um algoritmo guloso e processamento em lotes, obtendo ganhos de desempenho de mais de cem vezes em relação ao BLAST. O *MegaBLAST* requer uma área contínua de no mínimo 30 bases para ser considerada uma HSP, enquanto o BLAST requer 12 bases, com isto diminuindo consideravelmente a sensibilidade do *MegaBLAST*. Segundo (MA, 2002) o *MegaBLAST* possui problemas com sequências de entrada muito grandes, e segundo (TAN, 2005) o *MegaBLAST* consome uma grande quantidade de memória, que é proporcional ao produto do comprimento da sequência de entrada e do tamanho da base de dados. Em (TAN, 2005) é apresentado uma versão otimizada para o *MegaBLAST*, que sobrepõe as operações de escrita dos resultados em disco com o processamento. Ao invés de indexar a sequência de entrada, são indexadas as sequências da base de dados e ele também possui uma versão paralelizada utilizando *Message Passing Interface* (MPI), onde se distribui a base entre os nós de um *cluster*. O consumo de memória também é muito alto, sendo que para uma base de dados de 10 megabytes e um conjunto de sequências de entrada de 100 kilobytes são necessários 112 megabytes de memória, contra 277 megabytes da versão original.

O trabalho de (MORGULIS, 2008) é uma versão do *MegaBLAST* onde a base de dados é indexada antes da busca, como ocorre com BLAT, SSAHA e *miBLAST*. Esta versão indexada do *MegaBLAST* apresenta tempos de busca melhores que a versão não indexada, mas a qualidade nos resultados é um pouco inferior a versão original. O maior problema é o consumo de memória. O arquivo de informações sobre o *software*[1] que

[1] ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/indexed_megablast/README.usage

implementa este método diz que os índices são aproximadamente quatro vezes o tamanho da base de dados e se todos os dados do índice não couberem na memória principal, não há vantagens em utilizar este método.

Entre trabalhos que utilizam índices pode-se ainda apresentar o trabalho de (KALAFUS, 2004), que é utilizado para alinhar genomas e para isto utiliza tabelas *hash* e um algoritmo para a obtenção de sequências repetitivas e de sequências curtas apresentado em (RENEKER, 2005). Métodos baseadas em transformação não são discutidos por serem considerados fora do escopo deste trabalho, porém Jiang (JIANG, 2007) apresenta os principais métodos que utilizam esta técnica.

A criação de índices para acesso a sub-sequências tem a vantagem de otimizar o tempo de execução do algoritmo, por outro lado, é necessário mais memória disponível para a sua execução. Uma solução é distribuir a base de dados e seus índices entre os nós de um *cluster* e desta forma diminuir os requisitos de memória para cada nó e também paralelizar a execução do algoritmo.

## 3.5 DISTRIBUIÇÃO E PARALELIZAÇÃO DA BUSCA DE SEQUÊNCIAS SIMILARES

O paralelismo pode ser provido através do uso de computadores com múltiplos núcleos de processamento com acesso uniforme a memória (*Symmetric Multiprocessing* (SMP)), de computadores com múltiplos núcleos, porém sem acesso uniforme a memória (*Non-Uniform Memory Access* (NUMA)) ou de *clusters* ou agrupamentos computacionais. Os *clusters* são grupos de computadores conectados entre si, normalmente através de uma rede *ethernet*, que trabalham de forma sincronizada para resolver um problema em comum.

Existem duas abordagens principais para a paralelização do *BLAST*: (1) Tem-se como entrada um conjunto de sequências, também chamado de lote, que são distribuídas pelos nós do *cluster* e cada nó possui uma cópia da base de dados inteira ou parte dela, e (2) a paralelização do algoritmo de busca por sequências similares associado a fragmentação da base de dados. Um exemplo de execução por lote é apresentado na Figura 3.11. Nesta figura, é apresentado um conjunto de sequências como entrada para um *cluster* formado por quatro nós, sendo um o agendador e três os trabalhadores. A base de dados contendo as sequências é idêntica nos nós trabalhadores. Caso houvesse mais sequências do que nós disponíveis, seria utilizada uma fila de espera para que quando um nó terminasse uma busca, ele obteria nesta fila, a próxima sequência. Ao final é retornado um relatório

independente contendo os resultados da busca de cada uma das sequências de entrada. Um exemplo é o *SS-Wrapper* (WANG, 2004), que funciona como uma camada superior ao BLAST dividindo as sequências do lote a ser pesquisado entre os nós do *cluster*.

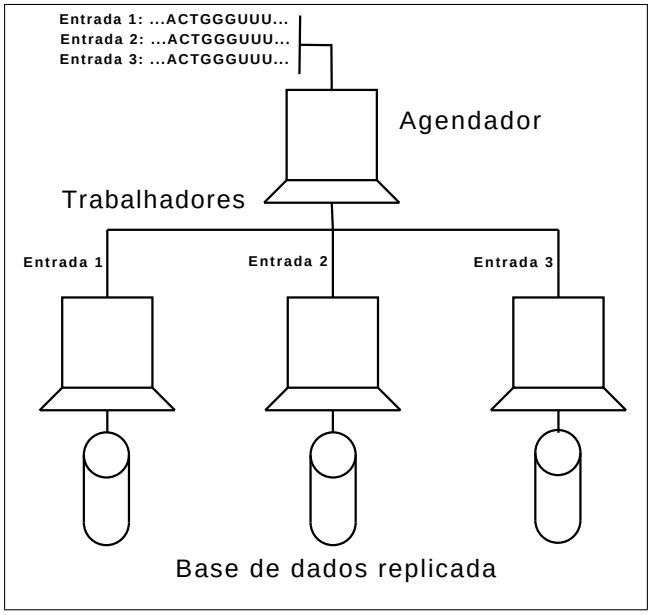


FIG. 3.11: Um lote de sequências para busca por similaridade.

A técnica de distribuição de sequência de entrada é interessante somente quando há um grande volume de sequências a serem pesquisadas, mas ela não reduz o tempo de busca individual de cada sequência do lote de entrada. A fim de minimizar o tempo de busca individual de cada sequência, é utilizada a distribuição da base de dados e a paralelização do algoritmo de busca. Um exemplo clássico desta abordagem é o *mpiBLAST* (DARLING, 2003). No *mpiBLAST* a base de dados é dividida em $n$ fragmentos de tamanhos iguais e no momento da execução, cada nó do *cluster* efetua a busca num sub-conjunto destes fragmentos. Na Figura 3.12 é mostrado um exemplo de distribuição, onde a base é dividida e cada nó é responsável por executar a busca numa parte da base. Cada sequência de entrada é direcionada para todos os nós trabalhadores do *cluster*, onde cada um faz a busca por similaridade na sua parte da base de dados e retorna o resultado para o agendador, que une os resultados das busca feitas pelos nós e a retorna ao requerente.

O ganho obtido pelo *mpiBLAST* é dado primeiramente pela paralelização da execução, pois a base é dividida e os fragmentos são pesquisados simultaneamente pelos nós envolvidos. Outro ganho é gerado devido a fragmentação da base de dados, permitindo a carga integral do fragmento na memória. O *mpiBLAST* também possui otimizações (THORSEN,

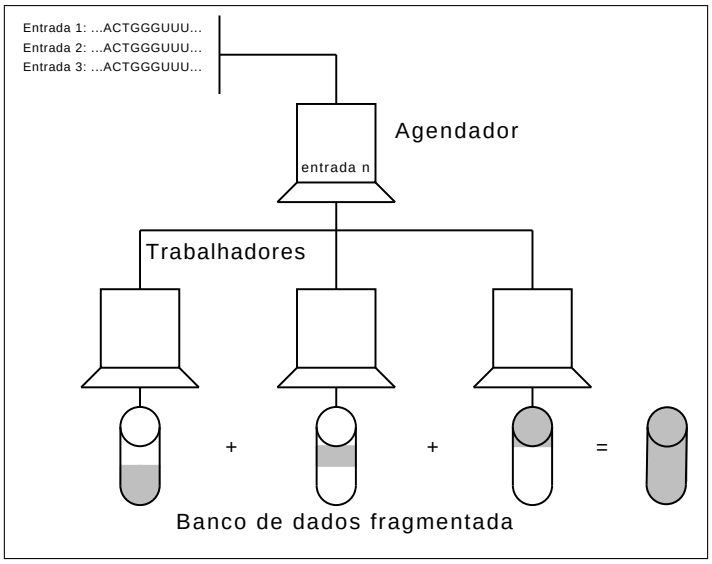


FIG. 3.12: Distribuição e paralelização da execução do *mpiBLAST*

2007) no seu sistema de entrada e saída que permitem alta escalabilidade em supercomputadores, como o *IBM Blue Gene/L (BG/L)*.

O BLAST do *NCBI* oferece a capacidade de execução do processo de busca utilizando-se paralelismo similar ao *mpiBLAST*, porém, ao invés de utilizar diversos computadores de um *cluster*, ele utiliza os processadores disponíveis em um único computador. Nesta paralelização, provida através do parâmetro "-*a* $N$", sendo $N$ a quantidade de processadores que serão utilizados, o *BLAST* utiliza $N$ *threads* para o processo de busca, onde cada processador é responsável por executar a busca numa $\frac{1}{N}$ parte da base de dados.

# 4 GENOOGLE

No Capítulo 3 foram apresentadas as principais técnicas e otimizações usadas na busca por sequências similares. As técnicas que tem maior sensibilidade são as que utilizam programação dinâmica, porém, elas necessitam de maior poder computacional e de memória, pois alinham as sequências de forma completa. Para melhorar o desempenho dos algoritmos, foram desenvolvidas heurísticas como BLAST, que buscam pontos iniciais de alta similaridade, chamados HSP, para depois fazer o alinhamento. É necessário ainda uma busca linear entre todas as sequências da base de dados para encontrar as sementes iniciais que constituem os HSP. Abordagens como o SSAHA e BLAT criam um índice invertido de sub-sequências, onde é possível verificar onde estão localizadas as sementes, sem ter que pesquisar linearmente em todas as sequências da base de dados. Porém o uso de índices implica em maior consumo de memória. Uma outra abordagem para otimizar as buscas, é a distribuição das bases em nós de um *cluster* e paralelização da busca entre estes nós. Porém ainda é necessário buscar as sementes linearmente na partição local da base de dados do nó.

A partir da pesquisa bibliográfica realizada, percebe-se que nenhuma das ferramentas estudadas faz uso de distribuição e paralelização da busca combinadas com o uso de índices. Desta forma, existe a possibilidade de explorar estas técnicas em conjunto para obter melhores resultados em relação ao tempo de execução dos algoritmos de busca por sequências similares em bases de dados. A proposta deste trabalho é implementar um servidor para busca de sequências genéticas que oferece baixo tempo de resposta, através da combinação de técnicas de indexação e paralelização em diferentes níveis.

Nas próximas seções é apresentado o protótipo *Genoogle* que utiliza as técnicas de indexação e paralelização estudas. Primeiramente é dada uma introdução à arquitetura do *Genoogle*, em seguida são apresentadas as técnicas de indexação que ele utiliza e finalmente é descrito o processo de busca utilizado.

## 4.1 ARQUITETURA DO GENOOGLE

O *Genoogle* é um sistema que tem como objetivo fazer a busca por sequências similares de forma rápida. Um índice invertido é utilizado para encontrar de forma ágil sub-sequências

similares na base de dados. Além disso o processo de busca é paralelizado para fazer uso de multiprocessadores.

O *Genoogle* opera de modo similar ao *BLAST* (ALTSCHUL, 1995, 1997) e ao *FSA-BLAST* (CAMERON, 2007): dado uma sequência de entrada, um conjunto de parâmetros e uma base de dados onde a busca será realizada, retornam-se todas as sequências similares a partir dos parâmetros especificados. O *Genoogle* se distingue das demais ferramentas pela utilização de índices, possuindo técnicas para otimizar a sensibilidade da busca, e utilização de sistemas com mais de um núcleo de processamento.

O *Genoogle* é dividido em quatro principais componentes: *1)* codificação das sequências, *2)* indexação das sequências, *3)* armazenamento das bases de dados e *4)* busca de sequências similares. A codificação é responsável pela conversão das sequências de formato texto para um formato binário e vice-versa. A indexação é responsável por armazenar as posições das ocorrências de sub-sequências na base de dados e auxilia na recuperação destas informações. A base de dados armazena as sequências e suas informações. O componente de busca é responsável por buscar as sequências similares à sequência de entrada na base de dados e por retornar os resultados ao requerente da busca.

A codificação e a indexação das sequências da base de dados são feitas uma única vez no momento que o software é iniciado. Esta etapa é denominada de pré-processamento. Para tal formatação é necessário informar quais são os arquivos contendo as sequências que serão formatados, o comprimento das sub-sequências e em quantas sub-bases de dados a base de dados será dividida.

Após o pré-processamento, dois conjuntos de dados estão disponíveis: sequências da base de dados codificadas e fragmentadas, e o índice invertido para as sub-sequências que compõem estas sequências. Estes dados são utilizados para a busca das sequências similares. A busca por sequências similares é feita com a sequência de entrada dada pelo usuário e com a utilização dos dados gerados no pré-processamento. Nas Seção 4.2 e Seção 4.3 são descritas com mais detalhes estas duas fases. A Figura 4.1 apresenta de forma geral o processo de pré-processamento e busca e os dados necessários para cada fase.

## 4.2 PRÉ-PROCESSAMENTO

A fase de pré-processamento é realizada uma única vez e não depende da sequência de entrada. Logo o tempo de busca de uma sequência não inclui o tempo de pré-processamento.

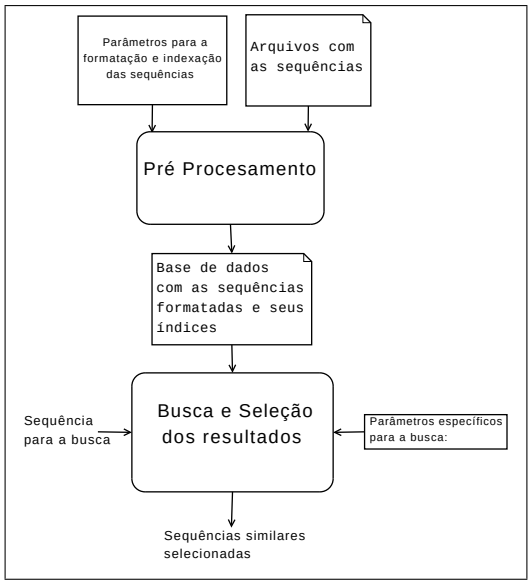


FIG. 4.1: Visão geral da execução.

No pré-processamento são construídos todos os dados necessários para a busca.

O pré-processamento é feito sobre um conjunto de dados de sequências que é armazenado em um arquivo no formato *FASTA*[2]. Este é um formato de texto que contém uma ou mais sequências genéticas, onde cada sequência possui algumas informações sobre ela e seu conteúdo. Os arquivos contendo as sequências são lidos e cada sequência é codificada e armazenada em outro arquivo. Esta codificação tem como objetivo diminuir o espaço necessário para armazenamento, facilitar a indexação e melhorar o desempenho das buscas e alinhamentos. Também é necessário definir o comprimento das sub-sequências, a quantidade de fragmentos gerados durante a divisão da base de dados e a máscara que será aplicada nas sub-sequências (o uso de máscara será explicado na Seção 4.2.1).

Cada base de uma sequência é codificada em 2 bits, ou seja, cada sub-sequência de 16 bases é armazenada num inteiro de 32 bits. Esta codificação junto com as demais informações da sequência (nome, código da sequência, identificador e descrição) são escritas em um arquivo no disco, compondo uma base de dados de sequências. O valor de 16 bases como comprimento das sub-sequências é um parâmetro definido no momento da codificação das sequências. Este valor pode ser modificado e variar entre 1 e 16 bases por sub-sequência.

Em seguida é feita a indexação das sequências codificadas. O índice invertido é construído neste momento indexando cada sub-sequência das sequências codificadas. Cada

[2]Em http://www.ncbi.nlm.nih.gov/blast/fasta.shtml há uma descrição do formato do arquivo.

sequência na base de dados é percorrida através de janelas não sobrepostas de acordo com o comprimento definido para as sub-sequências. Para cada sub-sequência é adicionada no índice invertido uma tupla contendo o identificador da sequência onde ela ocorre e a posição nesta sequência. Na Figura 4.2 é exibida a estrutura do índice invertido para o alfabeto de DNA e com sub-sequências de 8 bases de comprimento.

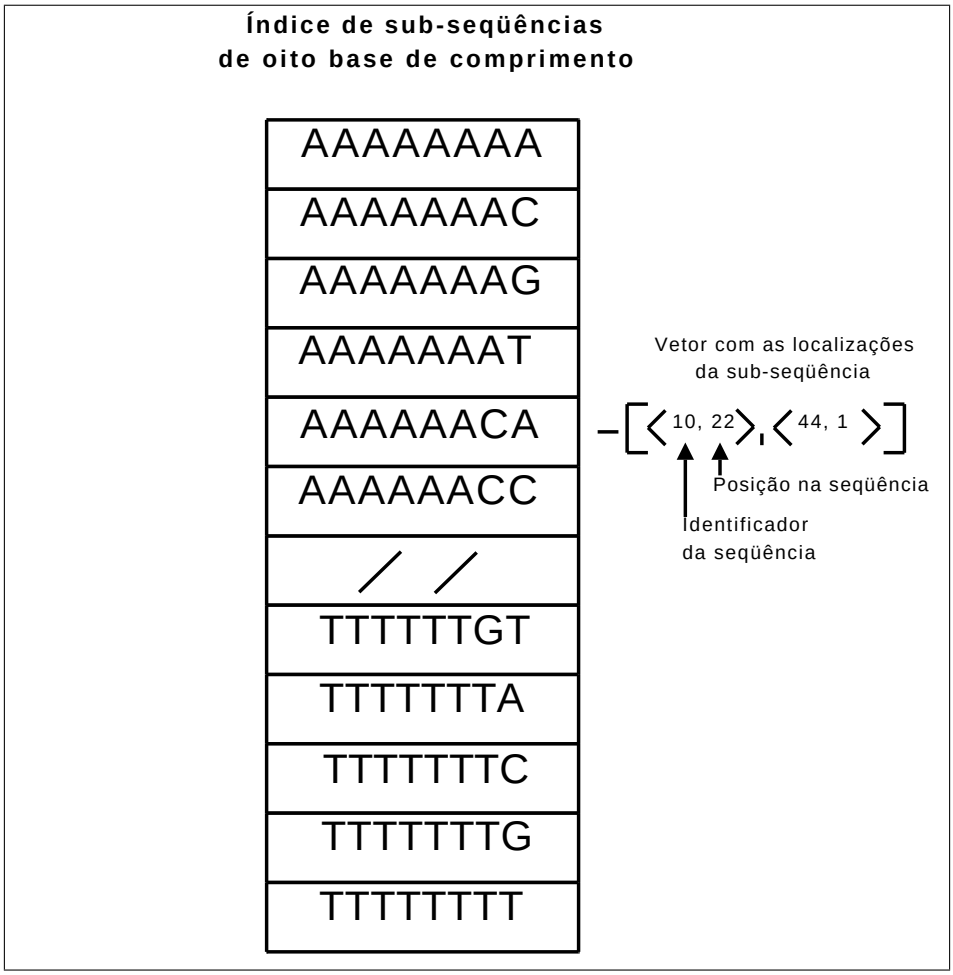


FIG. 4.2: Estrutura do índice invertido de sub-sequências

É importante salientar que quanto menor o comprimento das sub-sequências, maior será o tamanho da base de dados, da tabela de índices e também a quantidade de sub-sequências encontradas no índice, aumentando assim a sensibilidade da busca e aumentando o tempo da busca. Por outro lado, quanto maior for o comprimento, menor é o índice, porém menor é a sensibilidade. A variação do tamanho da estrutura do índice invertido ocorre porque as sequências são indexadas de forma não sobrepostas, então, quanto mais longas forem as sub-sequências, menor será a quantidade delas. Por exemplo, tendo uma base de dados contendo um milhão de sequências, e o comprimento das

sub-sequências de 10 bases, no total existirão aproximadamente 100.000 sub-sequências. Caso o comprimento das sub-sequências seja de 12 bases, serão aproximadamente 83.000 sub-sequências, obtendo uma redução de aproximadamente 20%. Porém a sensibilidade é prejudicada, porque ao invés de serem necessárias 10 bases idênticas para que a sub-sequência seja considerada idêntica, serão necessárias 12 bases.

O uso de janelas não sobrepostas tem como objetivo diminuir a quantidade de memória necessária para o armazenamento do índice invertido. Utilizando como exemplo a sequência "AGTCACGGTACGTAGG", utilizando janelas sobrepostas, são geradas as seguintes sub-sequências: *AGTCACGG,TCACGGT, TCACGGTA, CACGGTAC, ACGGTACG, CGGTACGT, GGTACGTA, GTACGTAG, TACGTAGG*, totalizando nove sub-sequências. Para janelas sobrepostas, a quantidade de sub-sequências pode ser calculada como $n - (s - 1)$, sendo $n$ o comprimento da sequência e $s$ o comprimento definido para cada sub-sequência. Por exemplo, considerando aproximadamente dez mil sequências e cada uma com trezentas bases, resultará em: $100.000(300 - (8 - 1)) = 29.300.000$ entradas no índice. Considerando que cada entrada no índice requer quatro bytes, são necessários aproximadamente 120 megabytes na memória para armazenar estas entradas. Dado que o *Genbank* (NCBI, 2006) possui mais de um milhão e meio de sequências, para indexá-las seria necessário aproximadamente 3 gigabytes. Segundo sua mais recente nota de lançamento[3] o *Genbank* está dobrando de tamanho a cada 18 meses.

Com a utilização de janelas não sobrepostas, a sequência "AGTCACGGTACGTAGG" terá as seguintes sub-sequências: *AGTCACG, TACGTAGG*. O cálculo da quantidade de sub-sequências utilizando janelas não sobrepostas é $\frac{n}{s}$. Com a utilização de janelas não sobrepostas, há um ganho de espaço de aproximadamente oito vezes o comprimento de cada sequência indexada. Porém este ganho é acompanhado com uma perda de sensibilidade. A perda ocorre porque nem todas as sub-sequências possíveis são indexadas.

Para compreender melhor o problema, a Figura 4.3 apresenta uma indexação e a busca neste índice com duas sequências hipotéticas. Na geração do índice foram usadas janelas não sobrepostas de comprimento 4. No processo de busca, a sequência de entrada "*ACASAEAZUL*" é divida em sub-sequências sobrepostas que são é pesquisadas no índice. Durante a busca, nota-se que não foi encontrada nenhuma sub-sequência no índice, mesmo existindo duas sub-sequências idênticas:*CASA* e *EAZUL*.

O problema de não encontrar sub-sequências no índice ocorre quando a sequência

[3]Nota de lançamento disponível em ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt

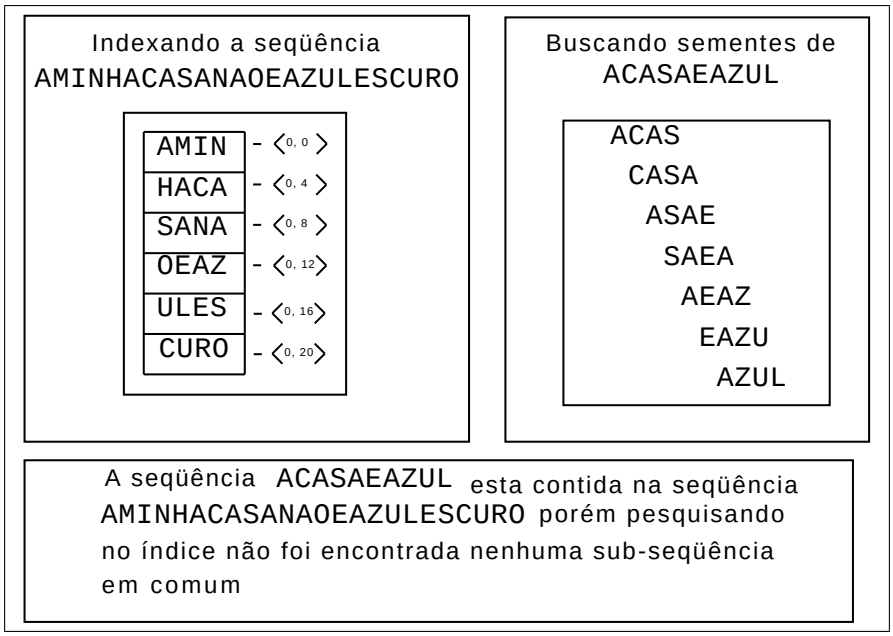


FIG. 4.3: Indexação de janelas não sobrepostas não retornando todas as sub-sequências.

de entrada possui sub-sequências similares às sub-sequências da base de dados com o comprimento igual ou um pouco maior das sub-sequências que são indexadas. A razão disto é que conforme dito anteriormente, as sequências indexadas são divididas em sub-sequências não sobrepostas, podendo ocorrer, como mostrado na Figura 4.3, casos em que sub-sequências não são encontradas, mesmo estando contidas nas sequências da base de dados. Através de estudos e análise do algoritmo e de sua implementação, verificou-se que este problema nos casos do *SSAHA*(NING, 2001) e *BLAT*(KENT, 2002), diminuindo consideravelmente a sensibilidade dos seus processos de busca.

Analisando a Figura 4.3 percebe-se que ao deslocarmos duas letras para a direita da sub-sequência *CASA*, ela terá 50% de similaridade com a entrada do índice *HACA* ou se o deslocamento for para a esquerda, cinquenta 50% com *SANA*. Neste caso, a utilização de deslocamentos das sub-sequências em tempo de execução resolve o problema, solução utilizada pelo *SSAHA*.

Utilizando outro exemplo, a seguir é analisada a relação entre as sub-sequências de entrada *AEAZ* e *EAZU* e a sub-sequência indexada *OEAZ*. Alinhando as sub-sequências *AEAZ* com *OEAZ*, é notado que apenas a primeira letra é diferente, possuindo uma similaridade de 75%. Mesmo com esta alta similaridade, a sub-sequência não é encontrada no índice e nem é possível utilizar deslocamentos. A sub-sequência de entrada seguinte, *EAZU*, pode ser deslocada e então alinhada com a sub-sequência *AEAZ* e então adicionada

na lista de sub-sequências encontradas no índice. O objetivo deste exemplo é demonstrar a perda de sensibilidade. Enquanto há um trecho de cinco letras *EAZUL* idêntico entre a sequência de entrada e a sequência indexada, utilizando apenas a busca no índice, não é encontrado nenhuma sub-sequência similar. Utilizando deslocamento, é encontrada uma sub-sequência, na qual três bases são idênticas. Na Figura 4.4 é apresentado a busca no índice utilizando sequências com três bases idênticas não importando a posição onde ocorra, porém devem ser consecutivas. Como exibido na figura, três sub-sequências de entrada são encontradas no índice e elas serão utilizadas numa fase seguinte como sementes para a construção dos HSP e então para a construção do alinhamento completo.

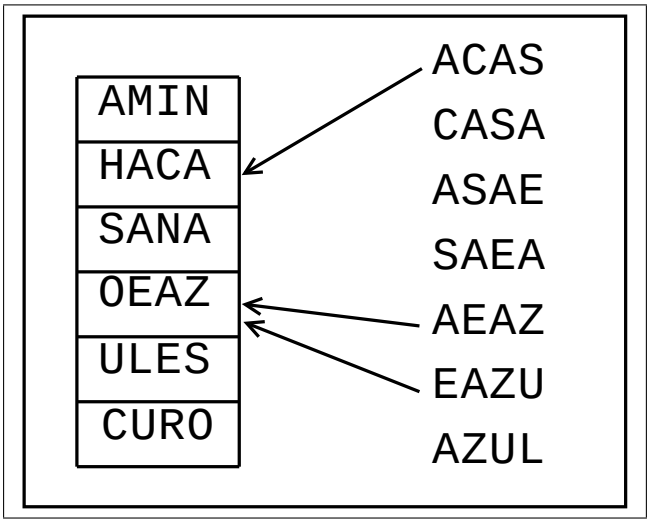


FIG. 4.4: Indexação de janelas não sobrepostas não retornando todas as sequências

### 4.2.1 MÁSCARAS PARA AS SUB-SEQUÊNCIAS

Uma técnica para melhorar a sensibilidade é a utilização de máscaras para as sub-sequências. As máscaras, baseadas no trabalho do *PatternHunter*(MA, 2002), informam quais trechos das sub-sequências devem ser mantidos ou removidos. Desta forma há dois ganhos: a sensibilidade, pois permite a busca no índice com sub-sequências não exatas, e ganho no espaço do índice, porque sub-sequências mais longas serão transformadas em sub-sequências mais curtas, havendo menos entradas no índice e também um menor número de sub-sequências.

As máscaras servem para permitir diferenças entre sub-sequências indexadas e sub-sequências da sequência de entrada, aumentando a probabilidade de encontrar sub-sequências similares no índice. As diferenças permitidas são especificadas na máscara, onde o carac-

tere *1* indica que aquela posição deve ser preservada, enquanto o *0* indica que ela deve ser removida. Desta forma, aplicar a máscara numa sub-sequência pode ser definido como uma função que recebe dois parâmetros: a sub-sequência e a máscara, e retorna uma sub-sequência obtida a partir da sub-sequência original eliminando algumas bases.

Utilizando como exemplo o índice da Figura 4.2, onde não é encontrada nenhuma entrada no índice, na Figura 4.5 é aplicada a máscara *0111* nas sub-sequências. Com a utilização da máscara, a sub-sequência *EAZ* é encontrada no índice.

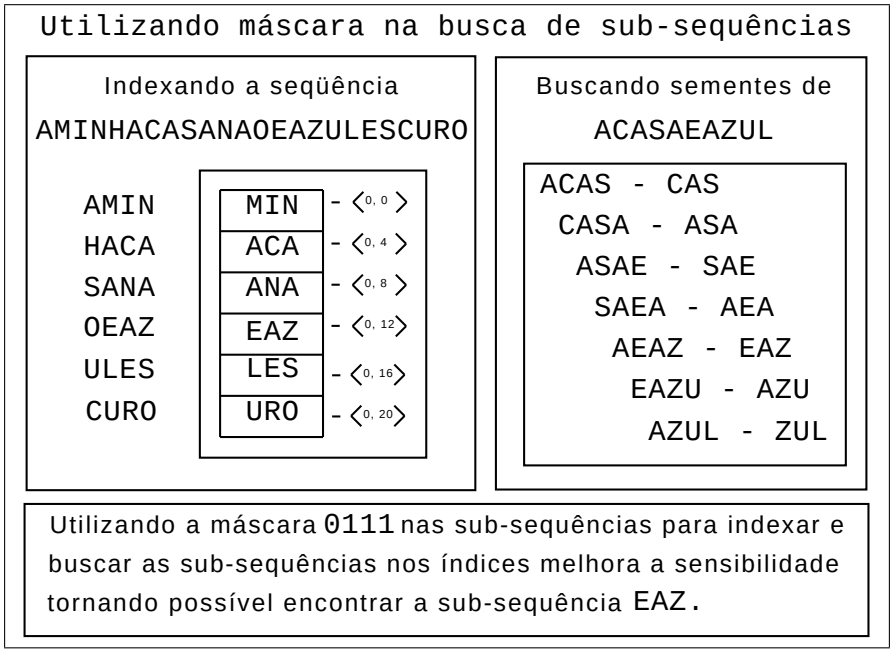


FIG. 4.5: Indexação de janelas não sobrepostas utilizando máscara.

Em (MA, 2002) é afirmado que a máscara *111010010100110111* é aquela que apresenta melhores resultados em questão de sensibilidade e esta máscara será utilizada ao longo deste trabalho, porém, sendo um parâmetro de configuração, pode-se modifica-la para cada execução do *Genoogle*.

### 4.2.2 ESTRUTURAS DE DADOS PARA INDICES INVERTIDOS

Neste trabalho, os índices invertidos são construídos utilizando um vetor de vetor de inteiros. O tamanho do vetor principal é a quantidade de sub-sequências possíveis, ou seja, para um índice invertido de sequências de DNA, considerando que o alfabeto possui 4 letras e o comprimento das sub-sequências é igual a $s$, obtemos $4^s$ sub-sequências possíveis. O tamanho dos vetores de cada sub-sequência varia de acordo com a quanti-

dade de ocorrências de cada sub-sequência na base de dados. Para cada ocorrência de uma sub-sequência no índice, utilizam-se dois inteiros de 4 bytes. Um inteiro armazena o identificador da sequência que contém a sub-sequência e o outro a posição que esta sub-sequência ocupa na sequência. É possível utilizar inteiros de 2 bytes, porém neste caso o índice ficará limitado a indexar no máximo 65536 ($2^{16}$) sequências e cada sequência indexada poderá ter no máximo este comprimento. Utilizando 4 bytes para o identificador da sequência e mais 4 para a posição, é possível indexar aproximadamente $4,2$ bilhões ($2^{32}$) de sequências e cada uma contendo mais de $4,2$ bilhões de bases, sendo o limite imposto apenas pela quantidade de memória disponível para armazenar as informações das sub-sequências.

São necessários $4^{11}$ entradas para armazenar apenas o vetor principal utilizando sub-sequências de 11 bases de comprimento, sendo que cada entrada ocupa 16 bytes (8 bytes para o ponteiro e 8 bytes para a estrutura do vetor que armazena as entradas da sub-sequência) totalizando, apenas para o vetor principal e a estrutura dos vetores de cada entrada, sem dados, 64 megabytes ($4^{11} * 16$) de espaço na memória para armazená-lo. No caso de sub-sequências com 10 bases de comprimento, serão necessários 16 megabytes, e no caso de sub-sequências de 12 bases de comprimento são necessários 256 megabytes de espaço na memória. Na Tabela 4.1 é exibido o custo de memória para armazenar a estrutura do índice invertido de acordo com o comprimento das sub-sequências. Percebe-se que há um crescimento exponencial de espaço requerido em função do comprimento das sub-sequências. Isto ocorre porque a quantidade de entradas neste vetor é dada pela expressão $4^s$, desta forma, deve ser analisado o espaço requerido para armazenar a estrutura do índice invertido juntamente com o espaço requerido para armazenar os vetores contendo as informações das sub-sequências.

| Comprimento das sub-sequências | Quantidade de entradas | Espaço requerido (MB) |
|---|---|---|
| 9 | 262.144 | 4 |
| 10 | 1.048.576 | 16 |
| 11 | 4.194.304 | 64 |
| 12 | 16.777.216 | 256 |
| 13 | 67.108.864 | 512 |

TAB. 4.1: Espaço necessário para armazenar a estrutura do índice invertido de acordo com o comprimento das sub-sequências.

Cada entrada que representa uma sub-sequência na base de dados necessita de 8 bytes.

Desta forma, o tamanho das sub-sequências influencia diretamente o tamanho do índice, pois como as sub-sequências são indexadas de forma não sobrepostas, quanto mais longa as sub-sequências, menos sub-sequências são indexadas e menos espaço é necessário para armazenar estas informações. Por exemplo, uma base de dados com 1 bilhão de bases e com sub-sequências de 10 bases de comprimento, serão geradas 100 milhões de entradas no índice, necessitando de 800 megabytes de espaço em memória para armazenar apenas as entradas. Caso o comprimento das sub-sequências seja 11, são necessários $90,9$ milhões de entradas no índice e 727 megabytes de espaço e caso o comprimento das sub-sequências seja 12, são necessários $83,3$ milhões de entradas, necessitando de $666,6$ megabytes de espaço na memória para armazenar as entradas das sub-sequências. Sub-sequências mais longas tem a vantagem de que o índice invertido ocupa menos espaço em memória, porém a sensibilidade é prejudicada, porque as chances de haver encontros diminui.

Na Tabela 4.2 é analisado o espaço requerido para armazenar o índice em memória. Nesta tabela são utilizados como exemplo três bases de dados contendo 500 milhões, 1 bilhão e 4 bilhões de bases ao todo e dividindo as sequências em sub-sequências de 11 a 13 de bases de comprimento. Nesta tabela é apresentado o espaço requerido para armazenar as informações das sub-sequências, o espaço requerido para armazenar a estrutura do índice invertido e a proporção do custo de memória da estrutura sobre todo o índice. Percebe-se que o custo da estrutura do índice é amortizada com o tamanho da base de dados. Para bases de dados pequenas, isto é, até 500 milhões de bases, o custo da estrutura do índice invertido para sub-sequências de 12 bases de comprimento representa 45% do espaço total requerido para armazenar todo o índice invertido, porém, utilizando sub-sequências com 11 bases, o custo da estrutura cai para aproximadamente 15%. Por outro lado, para uma base de dados com 4 bilhões de bases, o custo da estrutura do índice invertido utilizando sub-sequências com 11 bases de comprimento é menos de 9% do custo total e caso se usar sub-sequências com 10 bases, o custo de espaço requerido para armazenar apenas a estrutura do índice será aproximadamente 2% em relação ao espaço total necessário. Desta forma, pode-se constatar que o uso de sub-sequências mais longas não são interessantes por causa do custo de memória para armazenar a estrutura do índice invertido, porque mesmo em base de dados maiores, com 4 bilhões de bases o custo da estrutura do índice invertido para sub-sequências com 12 bases de comprimento é quase 10% do custo total, e no caso de sub-sequências de 13 bases de comprimento a estrutura do índice custa $17,2$% da memória total necessária.

| Base de dados | Entradas | | | Estrutura | Espaço | |
|---|---|---|---|---|---|---|
| Tamanho | Compr. | Quantidade | Espaço (mb) | Espaço (mb) | Total (mb) | Relação |
| 500 milhões | 9 | 55 milhões | 444 | 4 | 448 | 0,89% |
| | 10 | 50 milhões | 381 | 16 | 397 | 4,03% |
| | 11 | 45 milhões | 342 | 64 | 406 | 15,7% |
| | 12 | 41 milhões | 312 | 256 | 568 | 45,0% |
| | 13 | 38 milhões | 307 | 512 | 819 | 62,5% |
| 1 bilhão | 9 | 110 milhões | 888 | 4 | 892 | 0,44% |
| | 10 | 100 milhões | 762 | 16 | 778 | 2,05% |
| | 11 | 91 milhões | 649 | 64 | 712 | 8,98% |
| | 12 | 83 milhões | 633 | 256 | 889 | 28,8% |
| | 13 | 77 milhões | 616 | 512 | 1.128 | 45,3% |
| 4 bilhões | 9 | 440 milhões | 3520 | 4 | 3.524 | 0,11% |
| | 10 | 400 milhões | 3052 | 16 | 3.068 | 0,52% |
| | 11 | 363 milhões | 2769 | 64 | 2.833 | 2,25% |
| | 12 | 333 milhões | 2540 | 256 | 2.796 | 9,15% |
| | 13 | 308 milhões | 2464 | 512 | 2.976 | 17,2% |

TAB. 4.2: Espaço necessário para armazenar os índices invertidos e a relação de espaço necessário para as entradas e para a estrutura do índice

.

Junto com esta questão do custo da estrutura do índice, sub-sequências mais longas têm menos chance de serem encontradas no índice, porque são necessários encontros exatos mais longos e desta forma, diminui-se a sensibilidade do processo de busca. Analisando a Tabela 4.2 verifica-se que para bases de dados de 1 e 4 gigabytes, sub-sequências de 9 e 10 bases de comprimento necessitam de mais espaço no índice, mesmo tendo menor custo da estrutura. E a utilização de sub-sequências com 12 ou 13 bases tem o problema do custo da estrutura em relação ao tamanho total do índice. Assim, utilizam-se neste trabalho sub-sequências de 10 a 12 bases de comprimento, para comparar a questão de desempenho e sensibilidade. Porém outra questão é importante, o uso de máscaras influencia diretamente a questão da sensibilidade e do espaço em memória utilizado pelo índice.

Desta forma, utilizam-se as máscaras descritas na Seção 4.2.1, pois elas permitem trabalhar com sub-sequências mais longas sem prejudicar a sensibilidade. Na construção do índice, neste caso, as sequências da base de dados são lidas e divididas em sub-sequências com o comprimento da máscara que neste trabalho é 18 bases de comprimento, resultando em sub-sequências de 11 bases. Assim, para cada sub-sequência não sobreposta de 18 bases é aplicada a máscara, resultando numa sub-sequência de 11 bases que será

indexada. Desta forma, utiliza-se uma estrutura para indexar sub-sequências de 11 bases, porém a divisão das sequências é feita em sub-sequências de 18 bases, diminuindo o custo de memória. Para uma base de dados com 4 bilhões de bases e sub-sequências com 11 bases, haverá 363 milhões de sub-sequências, totalizando 2.833 megabytes de espaço necessário para armazenar o índice invertido. Utilizando-se máscaras com 18 bases de comprimento, haverá 222 milhões de sub-sequências e o índice invertido necessitará de 1.759 megabytes, resultando num ganho de 38% no total de memória necessária.

A fase de construção do índice invertido necessita de mais memória do que o índice invertido final. Isto ocorre porque na construção do índice invertido não é possível saber quantas entradas cada sub-sequência possuirá no índice, então é utilizado uma estrutura de dados que é um vetor que aumenta de tamanho sob demanda, porém esta estrutura de dados dinâmica utiliza mais memória que o vetor final, que é utilizado após todas as sequências serem indexadas. De forma geral, observou-se que para a construção do índice invertido, necessita-se de uma quantidade de memória 25% superior ao tamanho da base de dados. Importante lembrar que após a construção do índice invertido esta memória é liberada.

## 4.3 PROCESSO DE BUSCA DE SEQUÊNCIAS SIMILARES

Após a execução do pré-processamento, o *Genoogle* está pronto para executar buscas de sequências similares. O processo da busca é dividido em:

- processamento da sequência de entrada;
- busca no índice invertido e geração das HSPs;
- extensão e junção das HSPs sobrepostas e junção das HSPs sobrepostas;
- seleção das melhores HSPs e alinhamento local entre as HSPs;
- seleção dos melhores alinhamentos.

Para o processo de busca o protótipo utiliza alguns parâmetros que podem ser configurados individualmente para cada busca executada. Estes parâmetros são:

- quantidade de *threads* que são utilizadas na busca no índice invertido;
- em quantas vezes a sequência de entrada será fragmentada e comprimento mínimo de cada fragmento;

- distancia máxima entre as sub-sequências retornada pela indexação e pertencentes a uma mesma sequência para serem consideradas de uma mesma HSP;
- *E-Value* mínimo que cada HSP deve possuir para ir para a fase de extensão, considerando-se seu comprimento total;
- diferença máxima entre a pontuação mais alta encontrada e a pontuação atual no processo de extensão. Se a diferença ultrapassar este parâmetro, este processo de extensão é finalizado.
- quantidade máxima de HSPs que serão estendidas, alinhadas e retornadas para o usuário.

Todos estes parâmetros podem ser definidos no momento da busca e influenciam diretamente na sensibilidade dos resultado e no tempo total. Por exemplo, a distância máxima influencia na maior sensibilidade, porque flexibiliza a distância das entradas encontradas. Porém distâncias máximas muito altas aumentam o tempo da busca, porque aumentam a quantidade de HSPs armazenadas durante a busca e que são necessárias comparar com novas HSPs. O *E-Value* mínimo é um filtro para que apenas HSPs com comprimentos relevantes sejam estendidos e alinhados. Outro parâmetro fundamental para o desempenho é a quantidade máxima de HSPs que são estendidos e alinhados. Dependendo da sequência de entrada e dos parâmetros utilizados, podem haver até milhares de HSPs, porém apenas uma fração destes são interessantes. Desta forma é possível limitar e utilizar nas fases posteriores apenas as HSPs mais longas. Como exemplo, o BLAST estende e alinha apenas as 50 melhores HSP. Nas próximas seções estes parâmetros são descritos com mais detalhes.

### 4.3.1 PROCESSAMENTO DA SEQUÊNCIA DE ENTRADA

No processamento da sequência de entrada percorre-se esta sequência de forma sobreposta, gerando sequências cujos tamanhos são iguais ao comprimento da máscara a ser aplicada. Em seguida a máscara é aplicada em cada sub-sequência e a sub-sequência resultante é codificada para a codificação utilizada pelo *genooogle*. Contrariamente às sequências indexadas, utiliza-se janelas sobrepostas para gerar as sub-sequências da sequência de entrada. A Figura 4.6 mostra as fases de processamento da sequência de entrada.

Como dito na Seção 2.1, o DNA é uma fita dupla, ou seja, no momento do sequenciamento, apenas uma das fitas é sequenciada. Desta forma, no momento da busca, deve-se utilizar a sequência de entrada e seu complemento reverso. Então, em fato, no *Genoogle* e outras ferramentas de busca, são realizadas duas buscas, uma da sequência de entrada e outra do seu complemento reverso.

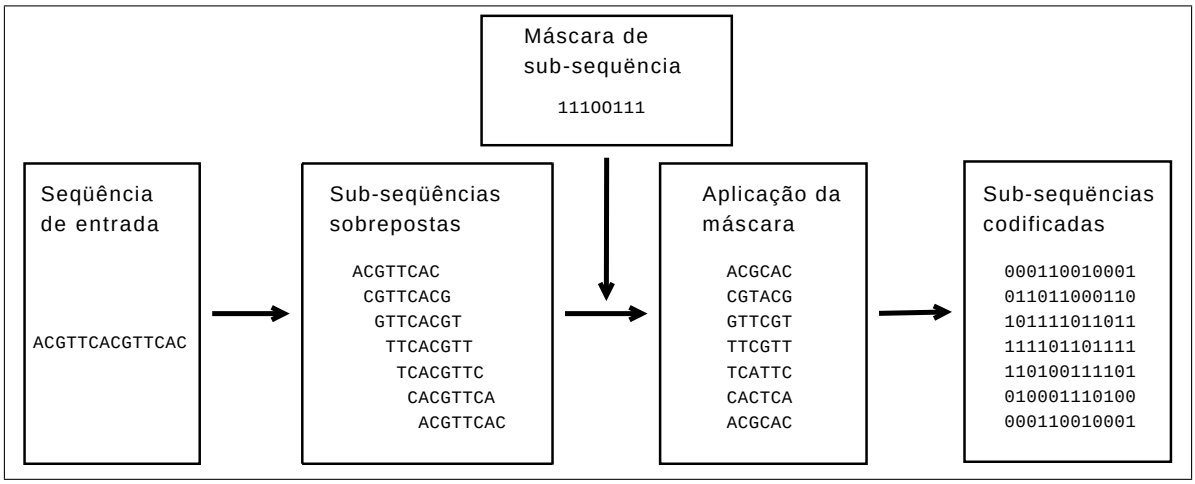


FIG. 4.6: Processamento da sequência de entrada

### 4.3.2 BUSCA NO ÍNDICE E GERAÇÃO DAS HSP

Na Figura 4.7 é exibido o processo de busca no índice invertido. A sequência de entrada é dividida em sub-sequências sobrepostas, é aplicado a máscara sobre elas e por fim elas são codificadas. Como as sequências são codificadas de forma binária dentro de um inteiro, é possível obter o valor da sub-sequência diretamente, tornando o processo de busca no índice simples e direto, pois a posição de determinada sequência no índice, é o seu próprio valor.

Com o valor da sub-sequência codificada são obtidas no índice invertido as posições onde esta sub-sequência ocorre na base de dados. As posições obtidas são armazenadas num vetor. Cada entrada deste vetor está relacionado com uma sequência da base de dados e nela são armazenadas informações sobre as áreas encontradas na sequência de entrada e que possuem similaridade com uma ou mais áreas na sequência da base de dados. Estas áreas armazenadas no vetor são chamadas de HSPs, e elas possuem cinco informações: as posições iniciais e finais na sequência de entrada e na sequência da base de dados e o comprimento desta área. A Figura 4.8 mostra uma área onde foram encontradas três entradas no índice e foi criada uma HSP contendo-as.

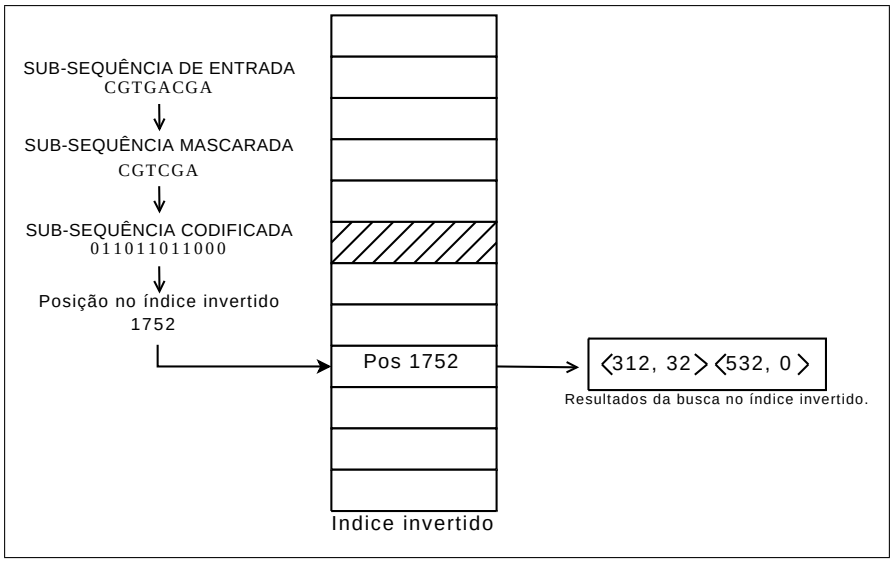


FIG. 4.7: Obtenção dos dados do índice invertido

Durante a leitura da sequência de entrada e busca no índice das sub-sequência, três operações são possíveis sobre as informações obtidas: inserir nova HSP, juntar com outra HSP e remover HSP. São comparadas as posições finais da HSP já existente com as posições iniciais da nova HSP. Caso as distâncias estejam abaixo de um limite estabelecido, estas duas HSPs são agrupadas numa única HSP, onde as posições iniciais são da HSP já existente e as finais da nova HSP. Se a distância entre o início da sub-sequencia obtida do índice e o fim da HSP existente for acima do limite pré-estabelecido, estas HSPs não serão agrupadas. Caso esta posição esteja mais distante, verifica-se se o comprimento da HSP existente é maior ou igual ao comprimento mínimo,se for, a HSP é armazenada para a próxima fase do algoritmo, caso não seja, ela é descartada. Se a entrada no índice não for agrupada com nenhuma HSP existente, uma nova HSP é criada e adicionada na coleção de HSPs da sequência da base de dados que ela corresponde.

Após percorrer todas as sub-sequências da sequência de entrada, é feita a última varredura nas HSPs ainda não armazenadas para verificar quais possuem o comprimento mínimo e então armazená-las. Por fim, são retornadas as HSPs de cada sequência em relação a sequência de entrada.

A Figura 4.9 exibe um exemplo de um vetor com as informações obtidas do índice. No vetor são armazenadas as HSPs entre a sequência de entrada e a sequência da base de dados representada pela posição no vetor. Cada posição no vetor representa uma sequência na base de dados e contém uma lista de HSPs. Cada uma representada por

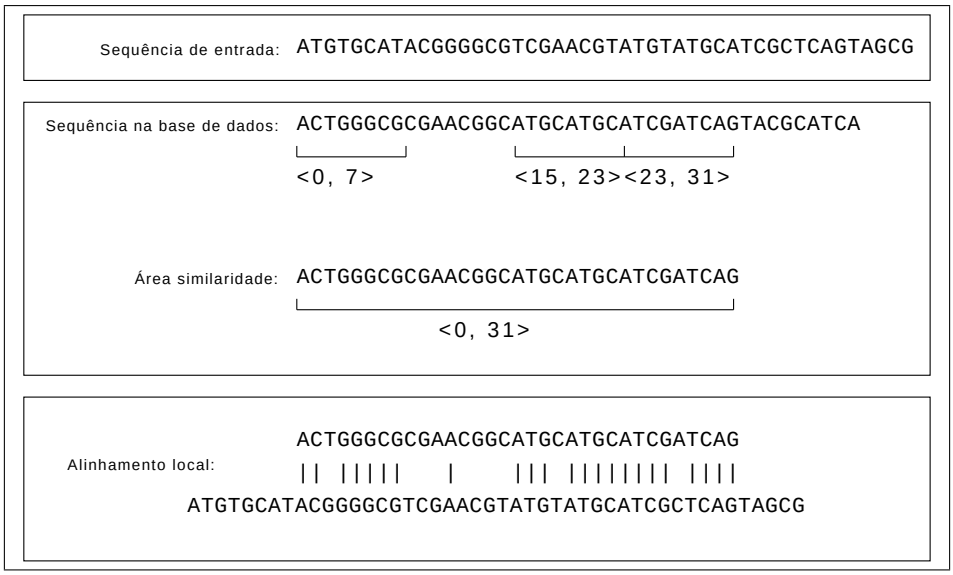


FIG. 4.8: Encontro das HSPs e alinhamento local.

uma tupla na figura contém a posição inicial e final na sequência da base de dados, a posição inicial e final na sequência de entrada e o comprimento que é o mínimo entre o comprimento da área de similaridade em relação a sequência da base de dados e em relação a sequência de entrada.

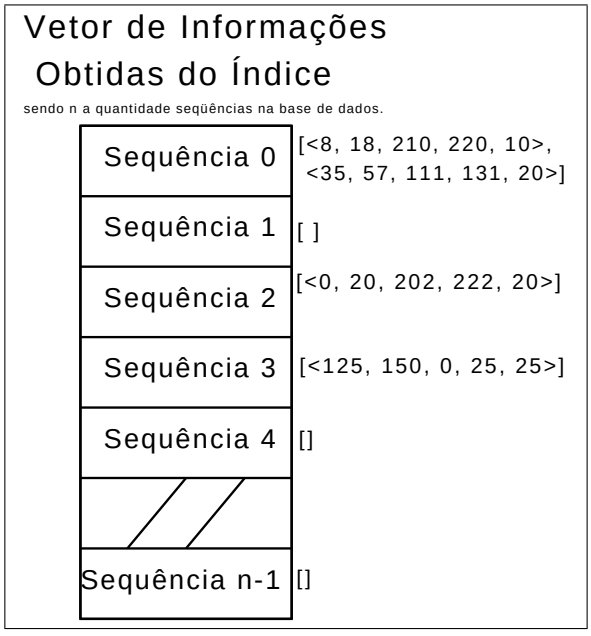


FIG. 4.9: Vetor com informações obtidas do índice

### 4.3.3 EXTENSÃO E JUNÇÃO DAS HSPS

Após a criação das HSPs, estendem-se elas em ambas as direções conforme descrito na Seção 3.3 para tentar ampliar o comprimento do alinhamento.

Durante a fase de extensão, pode ocorrer a sobreposição de duas ou mais HSPs que estavam próximas. Para não exibir resultados duplicados, é verificado se as HSPs possuem uma região sobreposta e neste caso, as HSPs são agrupadas numa nova HSP.

### 4.3.4 SELEÇÃO DAS HSPS, ALINHAMENTO LOCAL E ORDENAÇÃO DOS ALINHAMENTOS

Após a extensão e junção das HSPs, elas são ordenadas de acordo com seus comprimentos de modo decrescente. O *Genoogle* possui um parâmetro onde é especificado a quantidade máxima de alinhamentos que devem ser retornados, definido pela constante $r$. Desta forma, são selecionadas as $r$ HSPs mais longas para se realizar os alinhamentos. O objetivo deste parâmetro é diminuir o número de alinhamentos feitos, diminuindo o custo computacional, e retornar apenas os alinhamentos mais significativos. De qualquer forma, é possível definir que o *Genoogle* retorne todos as sub-sequências similares encontradas.

Para cada HSP é realizado um alinhamento local utilizando uma versão modificada do algoritmo de *Smith-Waterman* entre a sub-sequência da sequência de entrada e a sub-sequência da sequência da base de dados que formam a *HSP*. O algoritmo é modificado de forma que ao invés de usar a recorrência para construir um alinhamento entre as duas sub-sequências, limita-se às bases que serão utilizadas no alinhamento. Esta limitação é feita porque as duas sub-sequências alinhadas possuem uma alta similaridade, ou seja, não há necessidade de percorrer horizontalmente toda a matriz de alinhamento, pode-se restringir a algumas células próximas da diagonal.

Na Figura 4.10 é possível ver a matriz para o alinhamento de sequências utilizando programação dinâmica. Nesta matriz percorre-se apenas a área em branco, que é definida pela constante $k$. Como sabe-se que as sequências alinhadas possuem uma alta similaridade, o valor de $k$ pode ser reduzido. Desta forma, reduz-se o custo computacional de $O(nm)$ para $(km)$, sendo $k$ uma constante, o alinhamento passará a ter um custo linear. Por exemplo, para alinhar duas sequências com 350 bases cada uma, para calcular toda a matriz, são necessárias 122.500 comparações no total. Se limitar a distância da diagonal em 5 bases para cima e 5 para baixo, serão necessárias 3.500 comparações e como espera-

se que as sequências de entrada sejam similares, a perda na qualidade do alinhamento é baixa ou até mesmo nula.

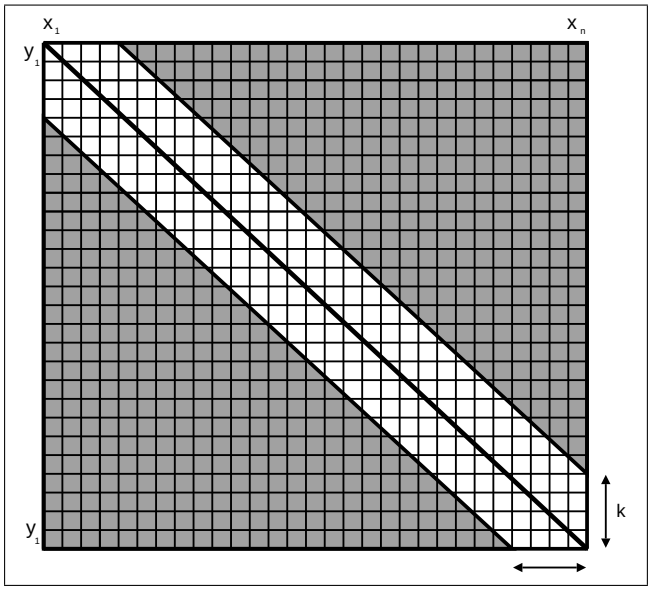


FIG. 4.10: Matriz do alinhamento, limitando-se da diagonal a distância $k$.

Ao limitar a distância da diagonal optimiza-se o processamento do alinhamento, porém a memória necessária para a construção da matriz para o alinhamento continua a mesma: $mn$. Para alinhamentos curtos, este custo de memória não é um problema grave, porém para alinhamentos de sequências mais longas, pode-se ocasionar na utilização total da memória disponível. Por exemplo, alinhar duas sequências com 10.000 bases cada necessita-se de aproximadamente $381mb$ de memória apenas para a matriz de alinhamento. Para diminuir o uso de memória, dividiu-se a as sequências em trechos e alinhou-se este trechos. Com um par de sequências de 10.000 bases cada, pode-se dividir em 4 trechos de 2.500 bases e construir 4 alinhamentos cada e no fim unir os resultados para formar o resultado final. A Figura 4.11 exibe um exemplo onde o alinhamento é dividido em duas partes e limita-se o alinhamento a uma distância $k$ da diagonal. Na figura, são construídos duas matrizes e o alinhamento estende-se a partir da diagonal destas matrizes. Com a utilização desta técnica, há uma economia de significativa. No exemplo das sequências com 10.000 bases divididas em 4 trechos, são necessários 93 megabytes de memória contra os 381 megabytes sem a utilização desta técnica.

Este alinhamento e sua pontuação são armazenados para que sejam ordenados para a exibição ao usuário. A Figura 4.12 exibe as fases do algoritmo que são responsáveis pela construção do alinhamento. Primeiramente são obtidas as HSPs através do índice inver-

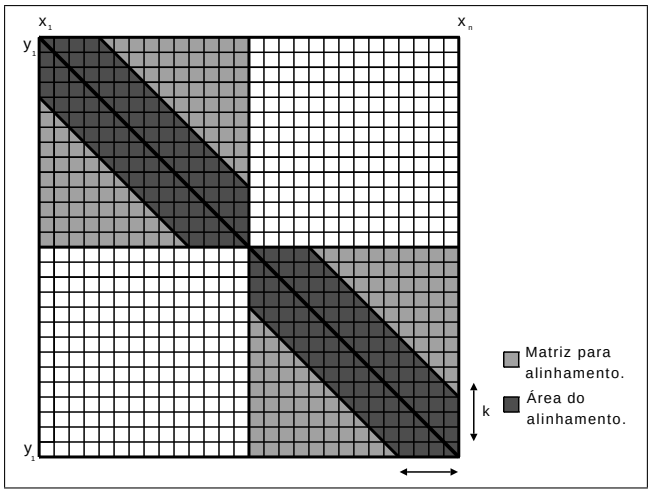


FIG. 4.11: Matriz do alinhamento, dividindo-se o processo de alinhamento e limitando-se da diagonal a distância $k$.

tido, então elas são estendidas em ambas as direções e então é construído um alinhamento com o resultado da extensão. Por fim ordenam-se todos os alinhamentos utilizando as suas pontuações de forma decrescente, retornando-as para o requisitante da busca.

## 4.4 DESENVOLVIMENTO DOS PROCESSOS DE PARALELIZAÇÃO

Na Seção 4.3 foi apresentado o algoritmo de busca desenvolvido. Para otimizar o desempenho da busca, o algoritmo foi paralelizado com o intuito de utilizar computadores multi-processados. São paralelizados três pontos no algoritmo: o acesso ao índice e aos dados, processamento da sequência de entrada e os alinhamentos.

### 4.4.1 FRAGMENTAÇÃO DA BASE DE DADOS

A paralelização do acesso ao índice e aos dados é feita dividindo a base de dados em fragmentos. Esta paralelização é similar a feita pelo *mpiBLAST*, exibido na Figura 3.12, sendo que ao invés de usar múltiplos computadores, a divisão é feita no mesmo computador e são executados múltiplas linhas de processamento (*threads*). A Figura 4.13 exibe como é feito a paralelização utilizando múltiplas *threads* com a divisão da base de dados. Cada sequência de entrada é enviada a todas as *threads*, e cada uma é responsável por executar a busca numa fração da base de dados. Então, todo o processo de busca descrito no Capítulo 4 é feito individualmente por cada *thread* e no final os resultados são agrupados.

Analisando o processo de busca com a fragmentação da base de dados, verificou-se que

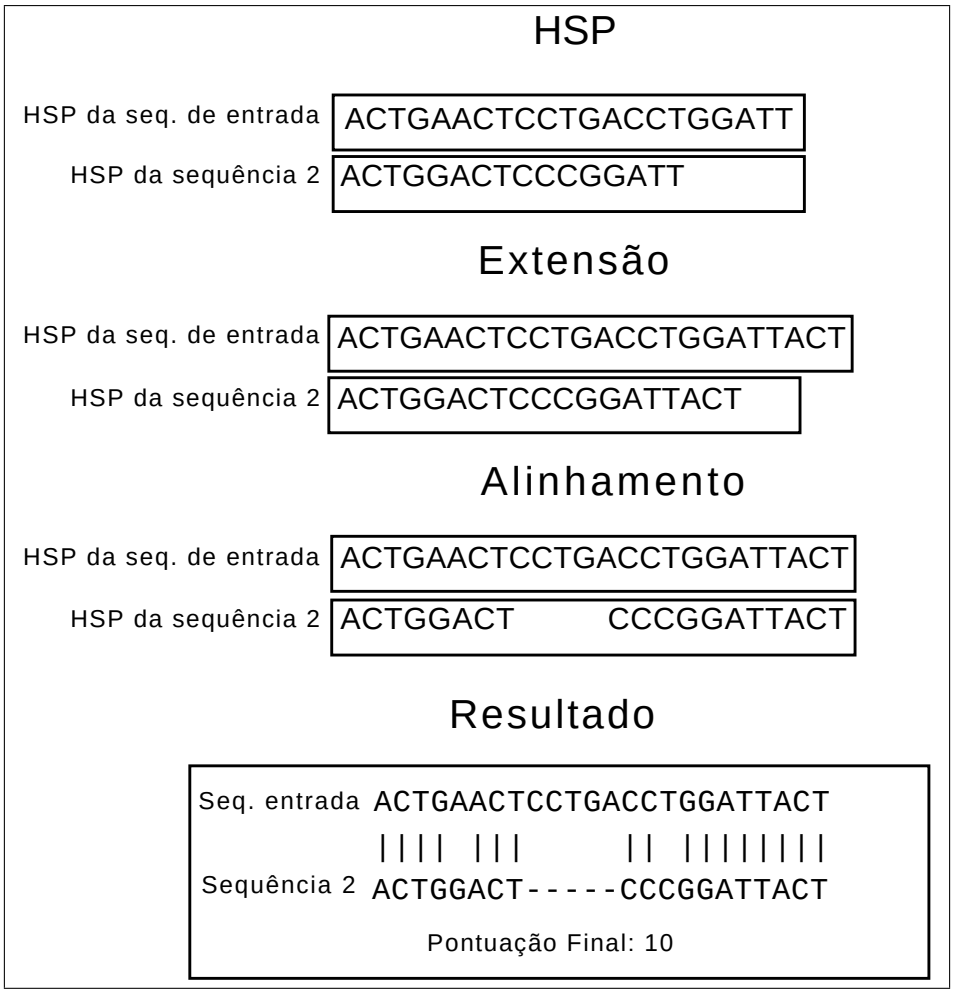


FIG. 4.12: As fases do algoritmo do Genoogle.

nem todas as *threads* dispendem o mesmo tempo de execução. Isto ocorre porque alguns fragmentos contêm mais sequências similares em relação a sequência de entrada do que outros. Uma maneira de minimizar este desbalanceamento de carga é dividir a base em mais fragmentos, porém conforme mostrado na Tabela 4.2 há um custo de memória para cada fragmento da base de dados e esta questão não resolveria o problema. Analisando o desempenho da busca, verificou-se que o tempo da busca no índice representa uma fração pequena do tempo total e que com a utilização da base de dados fragmentada, o principal problema é o tempo despendido no alinhamento das sequências. Desta forma, decidiu-se paralelizar os alinhamentos das sequências utilizando todas as *threads* possíveis.

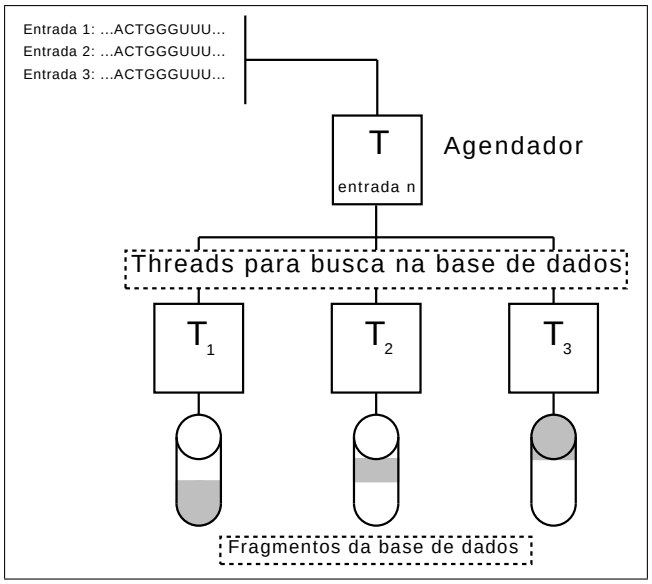


FIG. 4.13: Distribuição e paralelização da execução utilizando fragmentação da base de dados.

.

### 4.4.2 PARALELIZAÇÃO DO ALINHAMENTO

Com a fragmentação da base de dados, cada *thread* faz todo o processo de busca de sequências similares, retornando apenas os alinhamentos e o agendador é o responsável por selecionar os melhores alinhamentos. Esta técnica é interessante para bases de dados muito grandes, porém não paraleliza todo o processo de busca: sendo assim, uma nova fase foi desenvolvida para estender e alinhar as sequências. A Figura 4.14 exibe o funcionamento desta técnica, onde depois de fazer as buscas nos índices da base de dados fragmentada, as *threads* enviam as HSPs encontradas para outro componente que paraleliza o restante da busca e retorna os resultados a serem filtrados pelo agendador.

Na Figura 4.15 é exibido o funcionamento do componente que estende e alinha as sequências de forma paralela. As *threads* buscam as sub-sequências da sequência de entrada nos índices dos fragmentos das bases de dados e no final da busca adicionam as HSPs numa coleção compartilhada por todas as *threads* de busca nos índices. Esta coleção é ordenada pelo comprimento das HSPs e então são selecionadas as mais longas, de acordo com o parâmetro que especifica quantos alinhamentos devem ser retornados. Esta coleção contendo as HSPs selecionadas e ordenadas é passada para o componente de extensão e alinhamento, que para cada uma instancia uma classe que executará a extensão e alinhamento. Esta instância é adicionada a uma fila de *threads* a serem executadas. A

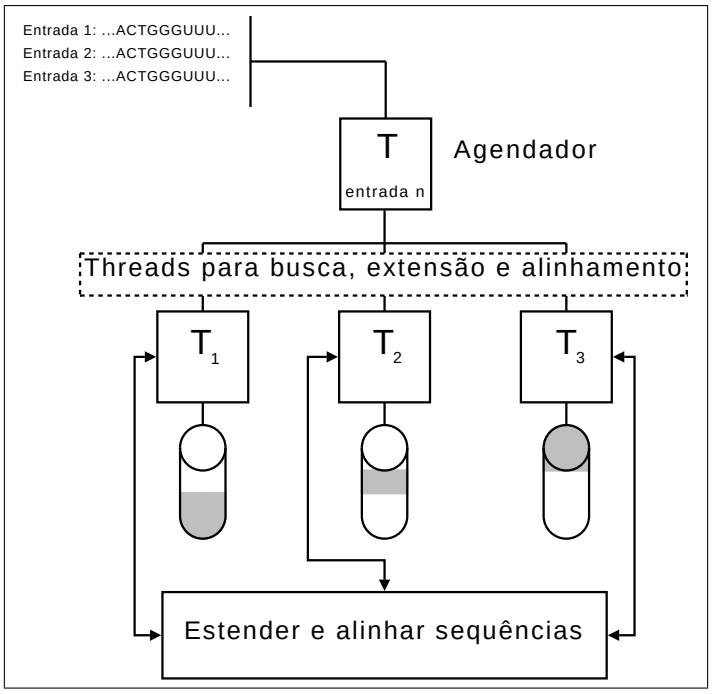


FIG. 4.14: Distribuição e paralelização da execução utilizando fragmentação da base de dados e *threads* para extensão e alinhamentos.

.

classe que executa a extensão e alinhamento tem uma referência para a coleção onde serão armazenados os resultados finais.

A classe denominada executor de *threads* gerencia a fila de *threads* que farão as extensões e alinhamentos. O executor de *threads* é configurado para executar um certo número de *threads* simultaneamente e quando uma *thread* é finalizada, os resultados são armazenados e o executor obtém uma nova *thread* para executar, até que todas as *threads* tenham sido executadas.

É apresentado na Figura 4.16 o processo de busca utilizando o paralelismo. Nesta figura é possível observar as quatro principais fases do processo de busca e as duas fases onde utiliza-se o paralelismo. Mesmo sendo apenas duas fases em quatro que são paralelizadas, o tempo de execução delas para uma sequência de entrada de 1000 bases representa 98% do tempo total do processo de busca. Nesta figura a base de dados é dividida em três fragmentos e consequentemente três *threads* são responsáveis por fazer as buscas no índice e gerar as HSPs. Para construir os alinhamentos, neste exemplo, são utilizadas quatro *threads* e no final o resultado é selecionado. O mais importante desta figura é a representação do processo de busca em 2 fases de busca e construção de dados e 2 de ordenação e filtragem deles.

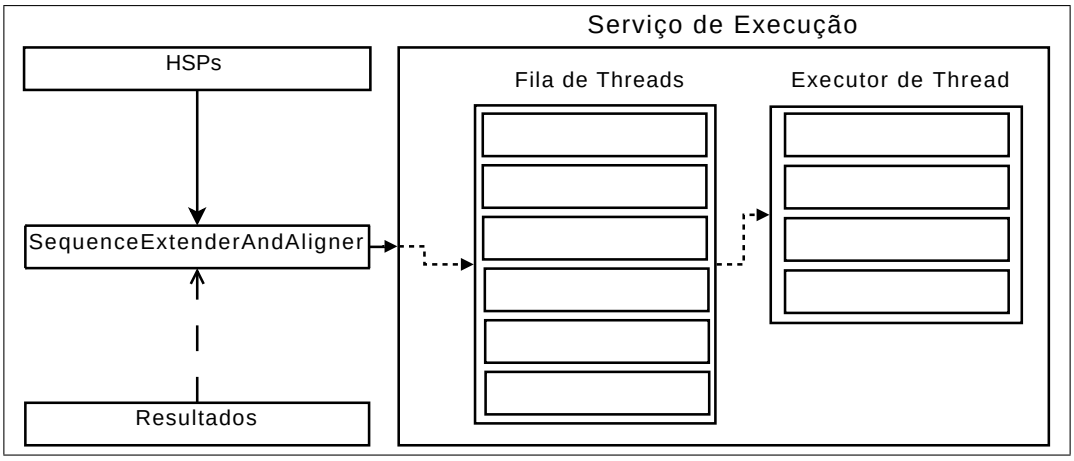


FIG. 4.15: Fila de execução das *threads* para extensão e alinhamento.

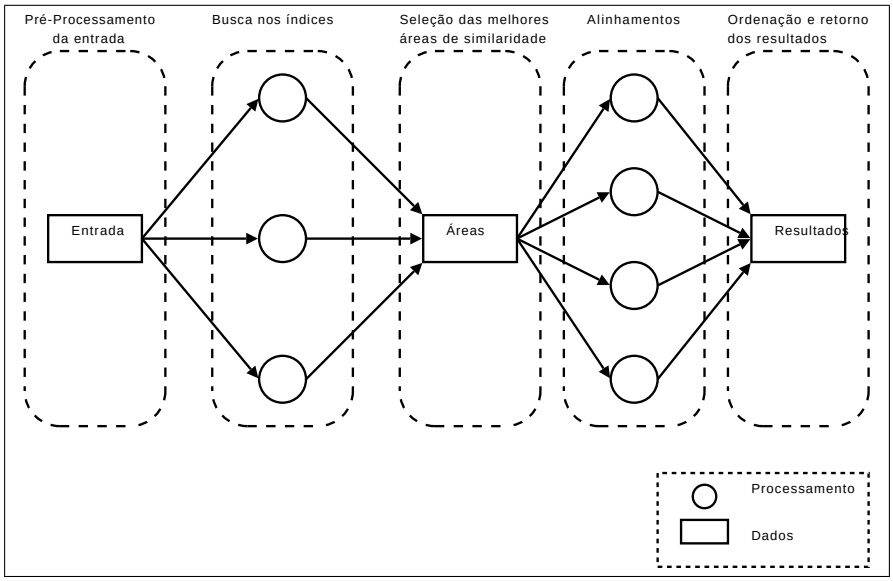


FIG. 4.16: Processo de busca e utilização de *threads*.

### 4.4.3 PARALELIZAÇÃO DO PROCESSAMENTO DA SEQUÊNCIA DE ENTRADA

Para uma sequência de entrada de 1000 bases de comprimento o tempo para o processamento da sequência de entrada e a seleção das HSPs representa aproximadamente 2% do tempo total do processo de busca, porém, quanto maior o comprimento da sequência de entrada, maior é o tempo necessário para o processamento desta sequência. Desta forma, verificou-se que para sequências de entradas longas, acima de um milhão de bases, o tempo necessário para o processamento destas sequências é muito longo, representando mais de 60% do tempo total do processo de busca. O principal motivo pela demora do processamento de sequências de entrada longas, é a aplicação das máscaras nas sub-sequências.

Mesmo sendo uma rotina simples, ela é repetida $n$ vezes, sendo $n$ o comprimento da sequência de entrada. Apesar da otimização da rotina de aplicação de máscaras, o tempo total consumido por ela continua sendo uma parte muito grande do tempo total de busca. Desta forma, a terceira paralelização é a paralelização do processamento da sequência de entrada, dividindo-a em segmentos.

A divisão da sequência de entrada em segmentos e a busca paralelizada tem outra vantagem: para utilizar o paralelismo da divisão da base de dados é necessário construir índices invertidos para cada fragmento, e com isto o espaço em memória necessário para armazenar a estrutura do índice invertido aumenta. Dividindo a sequência de entrada e não a base de dados, é necessário apenas um índice invertido. Conforme apresentado na Tabela 4.2, o custo da estrutura do índice é alto, sendo que para uma base de dados com sub-sequências de 11 bases de comprimento, são necessários 64 megabytes. Considerando que será utilizado um computador com 8 núcleos de processamento ao todo, é interessante dividir a base de dados em 8 partes, sendo necessários 512 megabytes apenas para armazenar a estrutura dos índices dos fragmentos das bases de dados. Desta forma, dividindo a sequência de entrada ao invés da base de dados, é possível paralelizar o processo de busca sem a sobrecarga de mais estruturas de índices.

Uma sequência de entrada de comprimento $m$ não pode ser dividida em segmentos com exatamente $\frac{m}{3}$ bases de comprimento, porque se uma HSP comecar no final de um fragmento e terminar no início de outro fragmento, ela será considerada como duas HSPs separadas. Além disso, se o tamanho delas for inferior ao comprimento mínimo considerado, elas serão descartadas. Para resolver este problema de sensibilidade, a divisão da sequência de entrada é feita sobrepondo o trecho final de cada segmento. O comprimento do trecho sobreposto é definido por $d - s$, onde $d$ é o comprimento mínimo de uma HSP e $s$ o comprimento das sub-sequências. A Figura 4.17 exemplifica o problema, onde uma sequência de entrada com 60 bases de comprimento é dividida em dois segmentos e a HSP também é dividida. Para solucionar isto, sobrepõe-se parte dos segmentos consecutivos, desta forma a HSP será encontrada no primeiro segmento e se houver continuação dela no segundo segmento, elas seriam unidas numa única HSP numa fase posterior.

Na Figura 4.18 é exibido o processo de busca utilizando segmentação da sequência de entrada. A sequência de entrada é dividida em sub-entradas, buscas ocorrem simultaneamente no mesmo índice seguido da junção das HSPs que pertencem a mesma sequência da base de dados. Na Figura 4.18 a sequência de entrada é dividida em duas sub-entradas

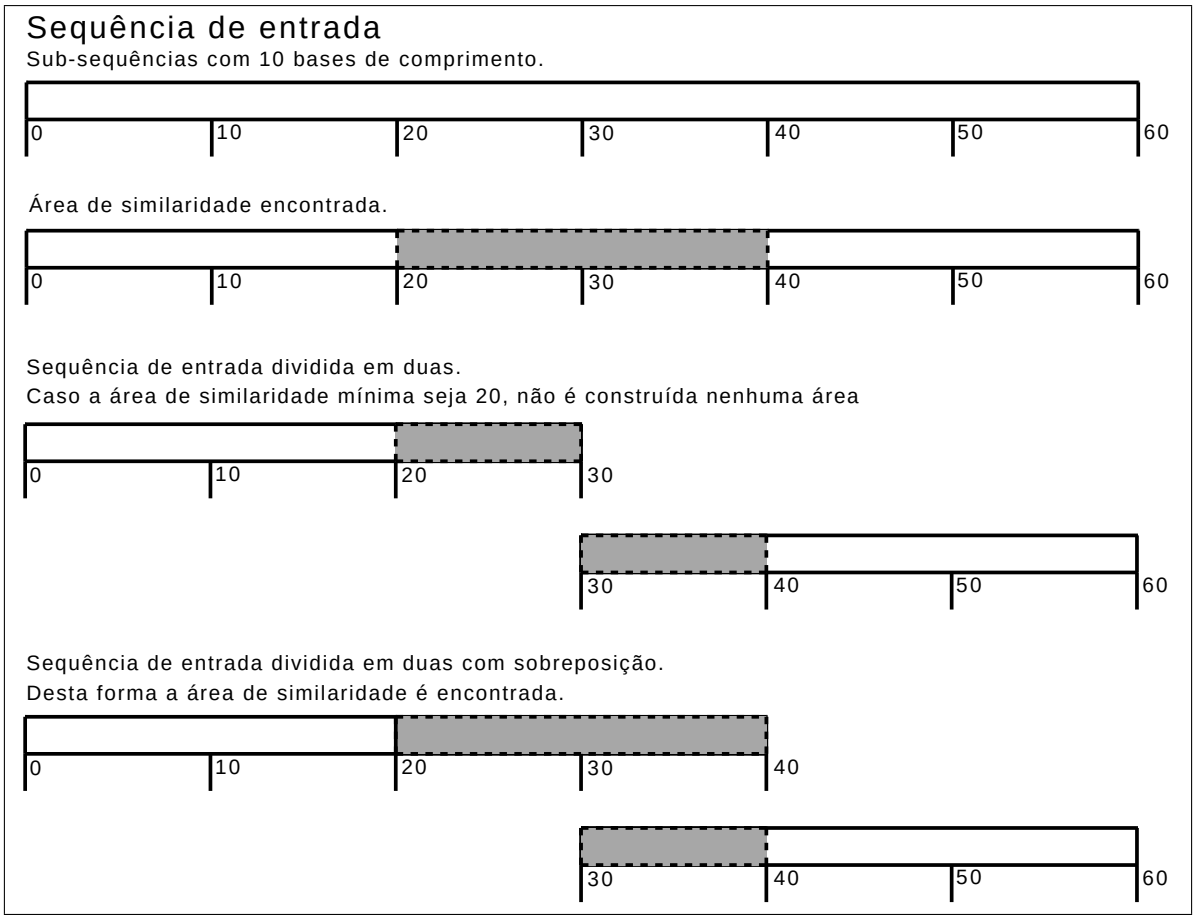


FIG. 4.17: Divisão da sequência de entrada.

e a base de dados em dois fragmentos. Desta forma, são utilizadas quatro *threads* para a busca no índice invertido, tendo a sobrecarga da utilização de apenas dois índices invertidos. O ganho com a criação de sub-entradas é dado pela paralelização do processamento da sequência de entrada e da paralelização da busca no mesmo índice invertido.

A sobrecarga da segmentação sobreposta é muito baixa. Considerando que uma sequência de entrada possua 1000 bases de comprimento e que o comprimento das sub-sequências são 11 bases, o mínimo para a HSP são 22 bases. Caso a sequência de entrada seja dividida em quatro segmentos, serão quatro segmentos de 261 bases, totalizando 1044 bases ao todo, tendo uma sobrecarga de $4,4\%$, se a sequência de entrada for dividida em 8 haverá uma sobrecarga de $8,8\%$. Estas sobrecargas diminuem conforme a sequência de entrada aumenta e desta forma verificou-se que há necessidade de limitar o comprimento mínimo de cada segmento da sequência de entrada. Analisando os tempos de execução, concluiu-se que o comprimento mínimo de cada segmento deve ser 10 vezes o comprimento das sub-sequências. Valores abaixo deste número não apresentam melhorias no

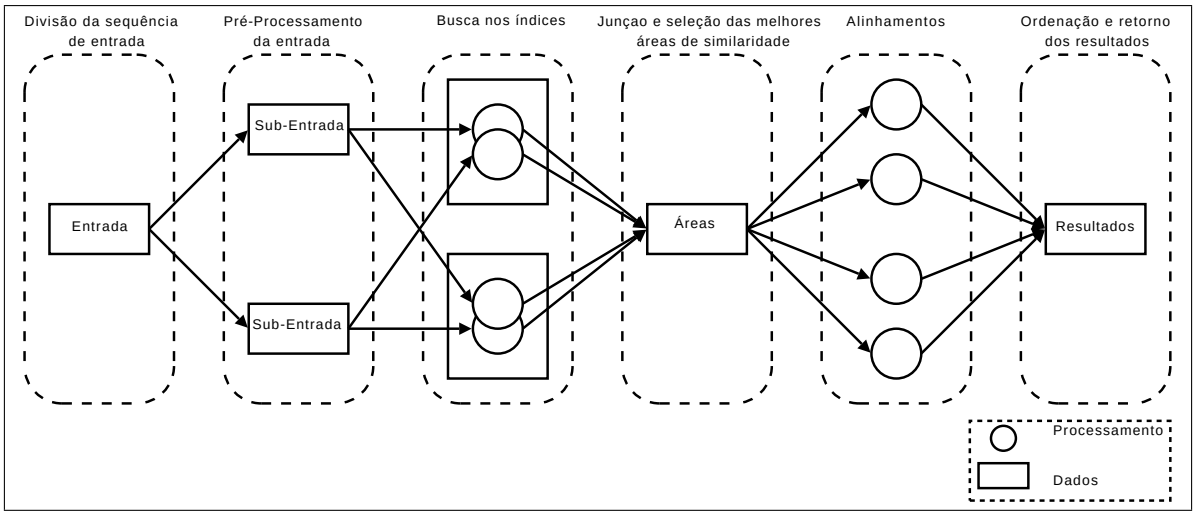


FIG. 4.18: Processo de busca e utilização de *threads*.

desempenho devido a sobrecarga apresentada e do custo de junção das áreas encontradas. Para otimizar o processo de busca no índice, é verificado se o comprimento da sequência dividido pela quantidade de *threads* é maior ou igual ao comprimento mínimo para os segmentos. Caso seja menor, o comprimento dos segmentos é computado considerando-se uma *thread* a menos, até que obtenha-se o comprimento mínimo ou apenas uma *thread* para a fase do processo de busca.

### 4.4.4 SELEÇÃO DA ESTRATÉGIA DE PARALELISMO

Por causa das sobrecargas do paralelismo, tanto na sequência de entrada como na base de dados, é necessário identificar a melhor estratégia para os diferentes tamanhos de entrada. O tamanho da base de dados pode ser classificado da seguinte forma:

- muito pequena: até $10^2$ bases;
- pequena: entre $10^2$ e $10^3$ bases;
- média: entre $10^3$ e $10^6$ bases;
- grande: entre $10^6$ e $10^9$ bases;
- muito grande: acima de $10^9$ bases.

A importância desta classificação é definir a estratégia de paralelização, pois dependendo da quantidade de dados, algumas paralelizações podem piorar o desempenho.

Por exemplo: para base de dados muito pequenas e pequenas, a paralelização da sequência de entrada não é vantajosa, por causa da sobrecarga discutida anteriormente. A mesma questão se aplica a divisão da base de dados, se a base for muito pequena, não é interessante dividi-la. A Tabela 4.3 apresenta um resumo de quais estratégias são interessantes utilizar em relação ao tamanho da base de dados.

| | | Base de dados | | | | |
|---|---|---|---|---|---|---|
| | | Muito Pequena | Pequena | Média | Grande | Muito Grande |
| Entrada | Muito Pequena | A | A, | A | A, B | A, B |
| | Pequena | A, | A, | A, E | A, E, B | A, E, B |
| | Média | A, E | A, E | A, E | A, E, B | A, E, B |
| | Grande | A, E | A, E | A, E | A, E, B | A, E, B |
| | Muito Grande | A, E | A, E | A, E | A, E, B | A, E, B |
| Legenda: A: Alinhamentos, E: Entrada, B: Base de dados. | | | | | | |

TAB. 4.3: Estratégias de paralelização em relação ao tamanho dos dados.

Com as estratégias de paralelização, o algoritmo de busca desenvolvido possui grande flexibilidade, tanto para base de dados pequenas até para base de dados muito grandes. É importante salientar que o paralelismo na segmentação da sequência de entrada e nas extensões pode ser definido no momento da busca, pois não dependem de nenhum processamento anterior, porém, a paralelização utilizando a divisão da base de dados necessita de dados do pré-processamento. A divisão da base de dados é definida de acordo com o tamanho da base, que é imutável durante a execução do protótipo.

## 4.5 DESENVOLVIMENTO DO PROTÓTIPO

Para validar os conceitos de indexação e paralelização apresentados, foi desenvolvido um protótipo chamado de *Genoogle*. Nesta seção é apresentado o processo de desenvolvimento do protótipo, desde a escolha do ambiente de programação, das bibliotecas, as ideias implementadas e as técnicas utilizadas na sua implementação. Primeiramente é discutida a utilização do ambiente Java e das bibliotecas auxiliares. Após é descrito o processo de desenvolvimento do protótipo, desde as suas concepções iniciais até a versão em que foram executadas as validações.

### 4.5.1 AMBIENTE JAVA E BIBLIOTECAS UTILIZADAS

O protótipo foi desenvolvido no ambiente Java versão 1.6. O ambiente é composto basicamente pela *Java Virtual Machine* (JVM) ou máquina virtual Java, a biblioteca da máquina virtual, compilador *javac* que compila o código fonte para o formato binário da maquina virtual e ferramentas auxiliares como depuradores e ferramentas de análise de desempenho.

Os motivos da escolha deste ambiente e sua linguagem são: facilidade no gerenciamento de memória, biblioteca com as principais estruturas de dados, *framework* para o uso de *threads* e sincronizadores de *threads* na própria linguagem. Na linguagem *Java* a alocação de memória é feita através da criação de objetos e dados na pilha e com o uso do operador *new*, usado para a criação de objetos, de forma similar à linguagem *C++*. A diferença está no momento da liberação da memória: enquanto em linguagens como *C*, *Pascal* e *C++* o objeto deve ser destruído ou a memória deve ser desalocada manualmente a JVM possui um *Garbage Collector* (GC) ou coletor de lixo, que verifica quais objetos não são mais utilizados e os remove da memória, liberando o espaço ocupado. A utilização de um GC é importante para minimizar o risco de memória não desalocada e desperdício de memória e o risco de utilização de dados inválidos.

O ambiente Java possui uma completa coleção de classes, variando desde classes para compactação de dados, acesso de redes, componentes gráficos e as classes de estrutura de dados, como listas, pilhas e tabelas *hash* que compõem o *Java Collections Framework*. Este *framework* é importante porque diminui a necessidade de implementar diversas estruturas de dados que são utilizadas no desenvolvimento do protótipo, removendo a necessidade de reprogramação e consequentemente a criação de testes e diminuindo consideravelmente o risco de falhas de programação, ou *bugs*. O ambiente Java também possui um *framework* para gerenciamento da execução de múltiplas *threads* chamado de *Task Scheduling Framework* ou também chamado de *Executor Framework*. Este *framework* possui *Executors* (executores) que são utilizados para chamadas, agendamento, execuções e controle de tarefas assíncronas, de acordo com regras de execução. As implementações dos *Executors* proveem execuções das tarefas dentro de uma única *thread* existente, dentro de uma *thread* executada em segundo plano, dentro de uma nova *thread*, num conjunto de *threads* ou o desenvolvedor pode implementar seu próprio *Executor*. As implementações existentes na biblioteca padrão oferecem políticas de execução como limites da fila de *threads* e limites de *threads* sendo executadas concorrentemente que melhoram a estabil-

idade do aplicativo, prevenindo o consumo de todos os recursos do sistema. O ambiente Java também possui uma gama de sincronizadores, como semáforos, *mutexes* e barreiras, *latchs*(trincos). Mais importante do que possuir estes sincronizadores na biblioteca do ambiente, é a otimização da JVM para a sincronização de *threads*, isto é, a própria JVM possui otimizações para melhorar o desempenho de aplicativos onde são executadas muitas *threads* e que façam o uso do sincronizadores.

No desenvolvimento do protótipo foi utilizada a biblioteca *BioJava* (HOLLAND, 2008). O *BioJava* é um projeto dedicado a prover um *framework* para processamento de dados biológicos, oferecendo meios para leitura e escrita de sequências genéticas e proteicas, acesso a base de dados biológicas, ferramentas e interfaces para análise de sequências, rotinas de estatística e programação dinâmica. No início da implementação do protótipo o *BioJava* foi utilizado para leitura, armazenamento e alinhamento de sequências genéticas. Com o desenvolvimento do protótipo, foi verificado que as rotinas de leitura dos arquivos *FASTA* consumia muita memória. Utilizando a estrutura do *BioJava* foram desenvolvidas novas classes para a leitura do arquivo e armazenamento das sequências. Questão similar ocorreu com o alinhamento de sequências, onde verificou-se que a implementação do algoritmo de alinhamento *Smith-Waterman* consumia mais memória e tempo de processamento que o necessário. Portanto a implementação foi otimizada e também foi implementada a técnica para restringir a distância da diagonal apresentada na Seção 4.3.4. Desta forma, utiliza-se do *BioJava* apenas as interfaces definidas por ele e algumas classes bases, sendo que as implementações das classe finais foram refeitas para melhorar desempenho do algoritmo.

### 4.5.2 HISTÓRICO DO DESENVOLVIMENTO

O protótipo começou com o desenvolvimento de um índice invertido utilizando uma tabela *hash*. Esta primeira versão lia os arquivos *FASTA*, dividia cada sequência em sub-sequências não sobrepostas e armazenava as suas posições numa tabela *hash* como um índice invertido. Esta primeira versão foi importante para perceber que a utilização de tabelas *hash* consome muito tempo na transformação das sub-sequências em chaves da tabela, pois para cada sub-sequência, deve ser aplicada a função *hash* no momento de armazenar suas posições e buscar esta sequência. Com isto, verificou-se que as sub-sequências já devem possuir seus valores *hash* antes de inserção na tabela *hash*. Além disto, verificou-se a necessidade de compactar as sub-sequências para armazená-las na

tabela *hash*.

Para obter o valor *hash* de forma mais eficiente e reduzir o espaço necessáro, foi utilizada uma técnica de compactação de sequências similar a descrita na Seção 3.3, onde as sequências são armazenadas em vetores de *bits*. As duas principais diferenças são a utilização de $16bits$ ou $32bits$ ao invés de $8bits$ no armazenamento das sub-sequências e não armazenar os caracteres especiais, Outra vantagem de compactar sequências dentro de máscaras de *bits* é que é possível obter o valor *hash* das sub-sequências apenas obtendo o valor em decimal do vetor de *bits*. Por exemplo, o resultado da codificação da sub-sequência $CATGCAAG$ é $0100111001000010$ e se converter este valor de binário para decimal o resultado é $20034$.

Após a utilização de compactação de sequências, verificou-se que a utilização de tabelas *hash* não é mais necessária. Isto porque não há necessidade de transformar as sub-sequências para inserção na tabela. Desta forma modificou-se o uso de tabelas *hash* para vetores nos índices invertidos, sendo que cada posição do vetor contém as posições de uma sub-sequência. Uma das vantagens da utilização de vetores é o acesso aos dados da sub-sequência, mesmo que o acesso utilizando tabelas *hash* e vetores possuem complexidade $O(1)$, o código nas tabelas *hash* é mais complexo e computacionalmente mais caro. Outra vantagem é que os vetores possuem o tamanho exato para o número de sub-sequências, enquanto que a quantidade de entradas na tabela *hash* pode variar para mais ou menos, desperdiçando espaço na memória ou, havendo colisões, diminuindo o desempenho no acesso às sub-sequências.

Cada entrada no vetor do índice invertido possui uma lista contendo em quais sequências e em quais posições da sequência a sub-sequência relatada está contida. A informação da sequência e posição estão codificadas dentro de um *long*, inteiro de 64 bits, sendo que os 32 bits inferiores é para a posição e os 32 superiores para o código da sequência. A opção de utilizar um inteiro para armazenar estes dados e não uma variável ou um vetor é dada por causa da economia de memória. O espaço total necessário para armazenar uma entrada, caso utiliza-se uma classe ou vetor, seria de 16 bytes (8 para a classe, 4 para um inteiro e mais 4 para o outro inteiro), com a utilização de um inteiro de 64 *bits*, o uso de memória fica restrito a 8 bytes.

Para minimizar o requerimento de memória do protótipo, foi desenvolvido o uso dos índices invertidos armazenados em disco, ao invés na memória principal. O funcionamento do método e a qualidade dos resultados são os mesmos, porém o desempenho é

prejudicando por causa do acesso ao disco. O uso de índices invertidos armazenados em disco degrada o desempenho em 50% quando o há apenas uma *thread* fazendo a busca no índice invertido, porém a degradação de desempenho aumenta consideravelmente com a utilização de mais *threads* na busca no índice, desta forma, tornando o acesso ao disco um gargalo no desempenho do processo de busca e diminuindo a escalabilidade do sistema. Por este motivo o uso de índices em disco não foi mais utilizado.

O método de busca sofreu alterações ao longo do desenvolvimento. A primeira versão separava as sequências em sub-sequências de 8 bases de comprimento. Durante a fase de busca, armazenava-se as sub-sequências encontradas e em qual posição da sequência da base de dados elas ocorrem. Fases posteriores utilizavam uma tabela para verificar onde a sub-sequência ocorre na sequência de entrada, chamada de tabela *lookup*, e fazia-se a filtragem das entradas encontradas e após a junção destas áreas similares e criação das HSPs. Este método obteve tempo de execução melhor que o BLAST com a mesma qualidade nos resultados, porém ele necessitava de muita memória para armazenar as entradas encontradas para a fase de filtragem. Isto porque haviam muitas sub-sequências encontradas e a fase de filtragem necessitava de todas as entradas encontradas no índice para que a análise pudesse ser feita.

O algoritmo de busca foi modificado para fazer a filtragem das HSPs no momento da obtenção destes dados. Para isto passou-se a armazenar e criar HSPs já no momento da busca dos dados no índice e consequentemente o uso da tabela *lookup* perdeu sua utilidade, invés de armazenar qual sub-sequência e que posição na sequência da base de dados nas entradas do índice, passou-se a criar e armazenar a HSP, informando o seu início e fim na sequência de entrada e início e fim na sequência da base de dados. A vantagem desta modificação é a economia de memória, pois as entradas que não serão utilizadas para a formação de HSPs serão descartadas no momento da busca no índice.

Analisando o processo de busca, verificou-se que as sub-sequências eram muito curtas e consequentemente muitas entradas eram encontradas no processo de busca, interferindo negativamente na filtragem das entradas do índice e na criação das HSPs Analisando ferramentas para encontro de informações (*information retrieval*) como o Apache Lucene (FOUNDATION, 2009a; GOSPODNETIC, 2005) e livros sobre o tema (GROSSMAN, 2004), verificou-se que nesta área os dados de entrada ou *queries* dificilmente ultrapassam a quantidade de 10 palavras. Sistemas como o *Google*, limitam a entrada dos dados a 32 palavras. Para fins de comparação, uma sequências de entrada pequena, com

100 bases, possui 100 palavras. Tendo em vista que dividir as sequências da base de dados de forma não sobreposta piora a sensibilidade da busca, a melhor solução é aumentar o comprimento das sub-sequências e com isto diminuir a quantidade de sequências no índice. Desta forma, outra modificação foi oferecer a possibilidade de ajustar o comprimento das sub-sequências no processamento da base de dados. Desta forma, uma sub-sequência pode possuir até 16 bases de comprimento, diminuindo a quantidade de sub-sequências no índice invertido e diminuindo a chance das sub-sequências serem encontradas, pois antes era necessário um encontro exato de 8 bases de comprimento, agora é necessário um encontro de 16 bases. Quantificando, com 8 bases de comprimento, a chance de encontrar uma sub-sequência exata no índice é de 1 em 65.536, com 16 bases de comprimento as chances já caem para 1 em mais de 4 bilhões. Através deste cálculo das probabilidades, testes com o protótipo e análise do funcionamento de outros métodos como o BLAST, verificou-se que o comprimento das sub-sequências estava muito longo. Como foi implementado a possibilidade de variar o comprimento das sequências durante a fase de processamento da base de dados, analisou-se diferentes comprimentos de sub-sequências variando de 9 a 18 bases. Através de testes e análise dos resultados verificou-se que as melhores opções são entre 10 e 12 bases, pois apresentam uma sensibilidade razoável e um bom tempo de execução, devido ao número mais reduzido de sub-sequências armazenadas no índice invertido e menos encontros obtidos do índice.

Para melhorar a sensibilidade da busca, foi desenvolvido um segundo índice denominado de índice de sub-sequências similares. Este índice contém uma entrada para cada sub-sequência possível e em cada entrada uma lista das sub-sequências similares. Neste caso, uma sub-sequência similar é uma sub-sequência onde há a divergência em uma base. Este índice fica armazenado no disco por causa do seu tamanho total e para cada sub-sequência da sequência de entrada, são obtidas todas as sub-sequências similares e são utilizadas na pesquisa do índice invertido. O uso do índice de sub-sequências similares otimizou a sensibilidade, sendo que em muitos testes a sensibilidade desde protótipo superou do BLAST. O problema deste índice é que por causa do seu grande volume de dados, ele necessita estar armazenado em disco e com isto o acesso a estes dados é mais lento, principalmente quando o acesso a disco é compartilhado entre múltiplas *threads*. Foi testada a possibilidade de gerar as sub-sequências similares durante o processo de busca e com isto verificou-se outro problema: o aumento significativo da quantidade de acessos ao índice invertido. Por exemplo, uma sub-sequência de 11 bases de comprimento

possui 33 sub-sequências similares e com isto são feitos 34 acessos ao índice apenas para uma sub-sequência.

Foi realizada uma pesquisa na literatura sobre técnicas para melhorar a sensibilidade de busca. Nesta pesquisa foi achada a ferramenta *PatternHunter* (MA, 2002) que utiliza máscaras para permitir o encontro não exato de sub-sequências. Na Seção 3.4 foi apresentada uma descrição sobre o uso das máscaras nas sub-sequências. Através de testes, verificou-se que a máscara apresentada no trabalho de (MA, 2002) apresenta uma melhora na sensibilidade sem sobrecarga adicional ao processo de busca. Outra questão notada em relação ao uso de máscaras é a diminuição da quantidade de dados no índice. Isto ocorre porque as sequências mascaradas são mais longas do que o resultado final. Por exemplo: utilizando-se máscaras, haverá aproximadamente $t/m$ sub-sequências no índice invertido, contra $t/s$, sendo $t$ o total de bases armazenadas na base de dados e $m$ o comprimento da máscara e $s$ o comprimento das sub-sequência. Considerando uma máscara onde a sequência de entrada possui 18 bases de comprimento, sub-sequências de 11 bases e uma base de dados contendo 4 bilhões de bases, caso não fossem utilizadas máscaras, haveria aproximadamente 363 milhões de entradas no índice e utilizando-se máscaras, este número cai para aproximadamente 222 milhões de entradas, um ganho de quase 40% no espaço utilizado para armazenar as entradas no índice.

Durante o processo de desenvolvimento do método sequencial, foram implementadas as paralelizações descritas na Seção 4.4. Primeiramente foi implementado o uso de múltiplos fragmentos de base de dados, após a paralelização dos alinhamentos e então a divisão da sequência de entrada. Para otimizar o tempo de busca, a seguir foi implementado o filtro na quantidade de alinhamentos que serão realizados e por fim o limite diagonal no algoritmo de *Smith-Waterman*.

### 4.5.3 INTERFACE

O protótipo foi desenvolvido utilizando uma arquitetura cliente-servidor com uma interface *web* e uma interface modo texto que são utilizadas pelo usuário fazer as buscas. O servidor *web* utiliza o *Apache Tomcat* (FOUNDATION, 2009b) para o servidor *web* e utiliza-se a tecnologia *JavaServer Pages* (JSP) para a construção das páginas *web* e comunicação com o servidor. O JSP é uma tecnologia que permite construir páginas *web* dinâmicas de modo simples utilizando o ambiente Java. Dentro do código fonte JSP há as *tags HyperText Markup Language* (HTML) e os comandos em Java para ler a entrada

do usuário, efetuar a busca e exibir os resultados.

Na Figura 4.19 é exibida a página inicial da interface *web* do *Genoogle*. Nesta página é possível observar que há uma barra de entrada onde a consulta é inserida e são incluídos três *links* de exemplos. Na Figura 4.20 são exibidos os resultados de uma busca. Os resultados são apresentados de forma decrescente em relação a pontuação total dos alinhamentos de cada sequência encontrada. Para cada sequência similar encontrada, exibe-se o identificador e outras informações da sequência, inclusive um *link* (*More info*) para a página do *National Center for Biotechnology Information* (NCBI) onde podem ser obtidas mais informações sobre a sequência. Para cada HSP são exibidos o alinhamento, a pontuação, o *E-Value* e a localização do alinhamento. Esta interface *web* é extremamente simples, utilizada apenas para demonstração e prova de conceito e diversas melhorias ainda podem ser feitas.

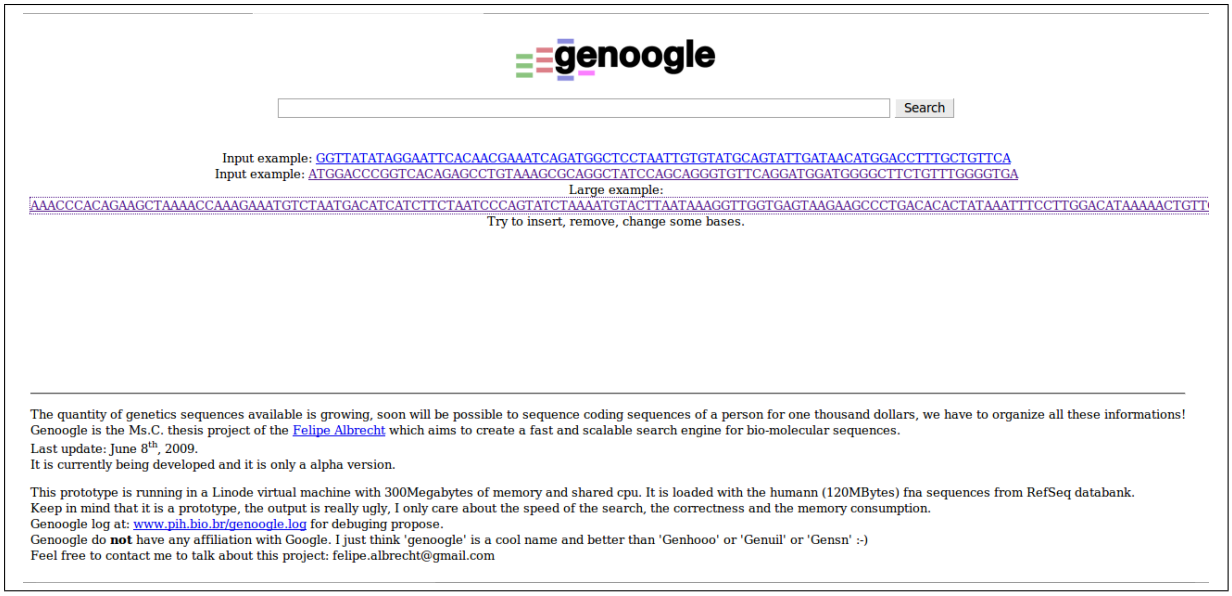


FIG. 4.19: Página inicial da interface web do Genoogle

A principal interface do *Genoogle* para a execução das buscas é uma interface em modo texto onde os comandos para as buscas são realizadas. Os resultados das buscas são armazenados num arquivo *Extensible Markup Language* (XML) definido pelo usuário. A seguir um exemplo de uma busca:

```
search Genomes_RefSeq BA000002 result_file QuerySplitQuantity=2 MaxThreadsIndexSearch=2 MaxHitsResults=20
```

Este comando solicita a execução de uma busca na base de dados *Genomes_ReSeq* utilizando as sequências do arquivo *BA000002* como sequências de entrada e o resultado

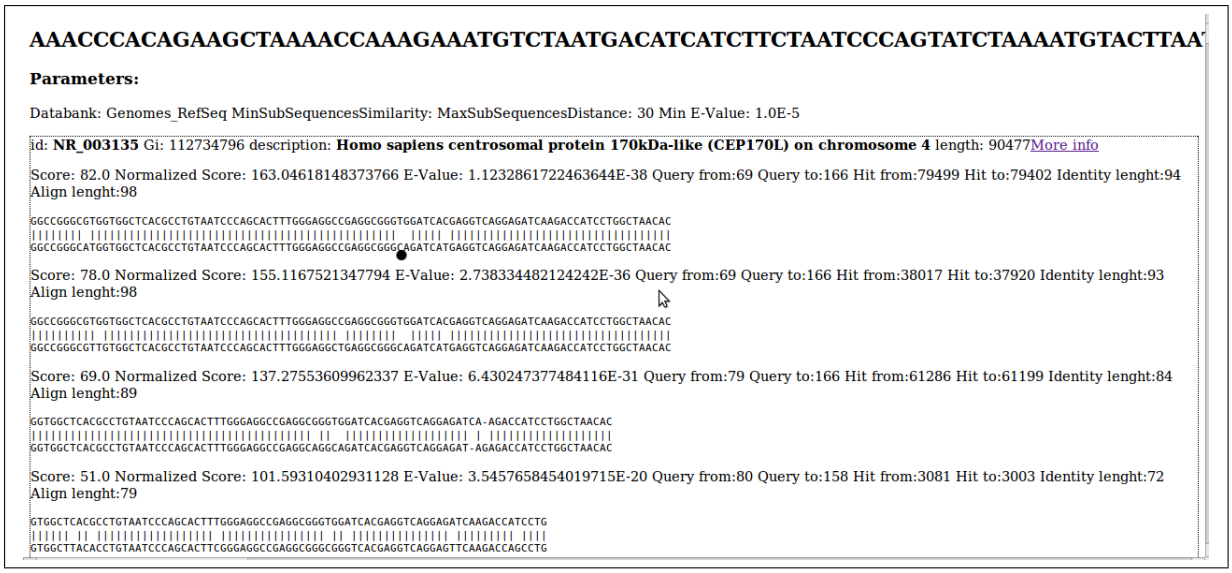


FIG. 4.20: Página de resultados da interface web do Genoogle

será armazenado no arquivo *result_file.xml*. Também são informados parâmetros adicionais para a busca. O parâmetro *QuerySplitQuantity=2* informa que a sequência de entrada deve ser dividida em duas, o parâmetro *MaxThreadsIndexSearch=2* diz que devem ser utilizadas duas *threads* na busca do índice, nas extensões e alinhamentos e o parâmetro *MaxHitsResults=20* informa para que sejam utilizados apenas os 20 melhores HSP nas fases de extensão e alinhamentos. Estes parâmetros adicionais são opcionais. Na Figura 4.21 são exibidas a execução do comando para o exemplo anterior e as saídas geradas pelo *Genoogle*, como a fase do algoritmo que está sendo executada e o tempo gasto em cada fase.

A interface modo texto também possui o comando para listar as bases de dados disponíveis (*list*), obter a lista de parâmetros para buscas (*parameters*), executar o coletor de lixo para liberar memória (*gc*), executar o último comando executado (*prev*) e executar um arquivo com outros comandos (*batch*). O comando *batch* é particularmente interessante, pois com ele é possível escrever um arquivo com todas as buscas que devem ser executadas e mandar executar este lote de comandos sem intervenção do usuário durante as buscas definidas. Por exemplo, para a execução dos experimentos do *Genoogle*, foram criados arquivos de lote com as buscas que deveriam ser executadas para cada sequência de entrada e um arquivo de lote que executava estes arquivos. Desta forma, foi-se necessário apenas iniciar o *Genoogle*, mandar executar o arquivo lote principal e aguardar até que todas as buscas fossem executadas, sem necessidade de intervenção do

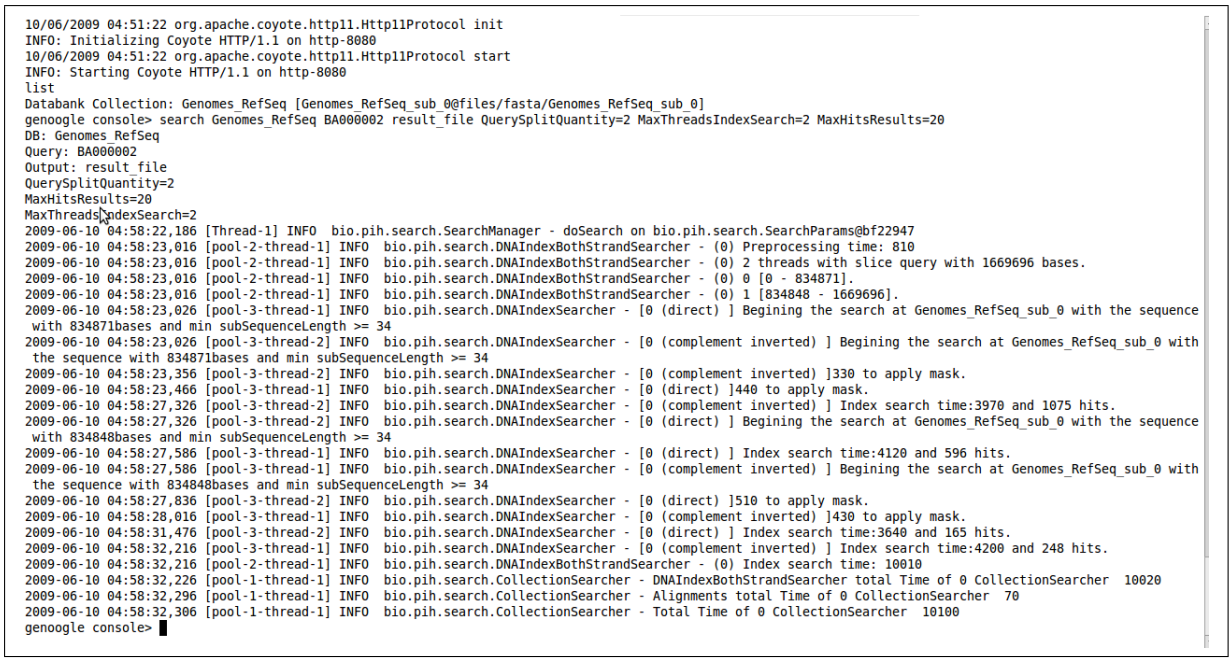


FIG. 4.21: Interface modo texto do Genoogle.

usuário durante a execução.

# 5 RESULTADOS

Para validar o funcionamento do protótipo e consequentemente das técnicas apresentadas ao longo deste trabalho, foram feitos vários experimentos. A análise dos resultados foi dividida em duas partes: análise do desempenho e análise da qualidade dos resultados da busca.

Para os experimentos, foi utilizado uma base de dados com as sequências da fase 3 do projeto do genoma humano (NCBI, 2008a) em conjunto com outras base de dados do projeto *Reference Sequence project* (RefSeq). Segundo (NCBI, 2008b), o RefSeq provê referência das sequências das moléculas que ocorrem naturalmente no dogma central da biologia molecular, dos cromossomos aos mRNA até as proteínas. As bases de dados do RefSeq foram utilizadas porque possuem sequências anotadas, já verificadas, de alta qualidade e que participam ativamente dos trabalhos de biologia molecular.

A base de dados do projeto do genoma humano chama-se *hs_phase3* e possui aproximadamente 3.8 gigabytes. As bases utilizadas do RefSeq são a *cow* com aproximadamente 57 megabytes, *frog* com 20 megabytes, *human* com 112 megabytes, *mouse* com 93 megabytes, *rat* com 73 megabytes e *zebrafish* com 62 megabytes, totalizando aproximadamente 417 megabytes. Nos experimentos foram utilizadas as bases do RefSeq junto com a *hs_phase3*, totalizando 4.25 gigabytes e sendo necessário aproximadamente 2 gigabytes de memória para a execução do protótipo após a fase da construção do índice invertido, porém sendo necessários aproximadamente 5 gigabytes na fase de construção do índice invertido.

Para a execução dos experimentos, foram gerados 11 conjuntos de sequências de entrada. Cada conjunto contém 11 sequências que têm aproximadamente mesmo tamanho, sendo 1 sequência obtida da base de dados e 10 mutações desta sequência. Os conjuntos são com sequências de 80, 200, 500, 1.000, 5.000, 10.000, 50.000, 100.000, 500.000 e 1.000.000 bases.

Nos experimentos foi utilizado um computador que possui 16 gigabytes de memória *RAM*, com 2 processadores *Xeon* $1,6$ *Ghz* com 4 núcleos de processamento cada. O sistema operacional é Linux versão 2.6.18. Foi utilizado o ambiente Java versão 6 com a JVM *JRockit* versão 3.0.3.

## 5.1 ANÁLISE DO DESEMPENHO EM RELAÇÃO ÀS TÉCNICAS DE PARALELIZAÇÃO

Os experimentos inicialmente foram utilizados para verificar o ganho de desempenho com a paralelização do processo de busca. Após a execução das buscas com diferentes configurações de paralelização, os tempos são comparados. Além disso, foram realizados experimentos para comparar o desempenho do protótipo com outras ferramentas.

### 5.1.1 ANÁLISE DAS TÉCNICAS DE PARALELIZAÇÃO

Para a execução dos experimentos foram geradas 3 bases de dados: uma base de dados sem divisão, outra base dividida em duas partes e uma terceira base de dados dividida em quatro partes. Para cada base de dados, foram executadas as mesmas buscas. Nos experimentos variaram a quantidade de *threads* utilizadas na busca, no índice invertido, nas extensões e alinhamentos.

As buscas utilizando-se sequências de entrada com 80 bases de comprimento foram executadas dividindo-se a sequência de entrada em 1 ou 2 partes e utilizando-se até 4 *threads* para a busca no índice, extensões e alinhamentos. Estas buscas foram executadas nas bases de dados divididas em 1, 2 ou 4 fragmentos. O tempo médio das buscas ficou entre 10 e $20ms$, sendo que a execução das 11 buscas, demoraram em média $150ms$. Foi observado que o uso de um ou mais *threads* não apresentou melhorias significativas no tempo de execução, o mesmo aconteceu com a fragmentação da base. O motivo de não haver ganho no tempo de execução da busca é que as sequências de entrada são muito pequenas, consequentemente o tempo total da busca é muito baixo e a paralelização não chega a ser vantajosa neste caso.

Utilizando as sequências de entrada com 200 bases foi possível perceber um ganho de desempenho quanto utilizadas técnicas de paralelismo. Utilizando-se 1 *thread*, sem dividir a sequência de entrada e sem dividir a base de dados, demorou-se $460ms$ para as 11 execuções. Com a mesma base, porém dividindo-se a sequência de entrada em 2 segmentos e utilizando-se 2 *threads*, a mesma busca demorou $340ms$, obtendo um ganho de aproximadamente 27%. Novamente o motivo de não ter alcançado um melhor ganho no desempenho é o pequeno tamanho das sequências. Com as mesmas entradas, porém utilizando-se 4 *threads*, divindo-se a sequência de entrada em 2 segmentos e a base de dados dividida em 4 fragmentos, os resultados foram retornados em $270ms$, dando um ganho de $42,6\%$ em relação a execução sequencial. Na Tabela 5.1 são exibidos os tempos de busca

para as sequências de entrada de 80 bases e 200 bases. Nota-se que para sequências de entrada de 80 bases, dividindo-as em 4 segmentos para a busca, o *speedup* é menor que 1, ou seja, há um decréscimo de desempenho. Isto ocorre por causa do custo da divisão da sequência de entrada, que não compensa o ganho obtido pelo paralelismo.

| Tamanho | Fragmentos | Tempo (ms) | Speedup |
|---|---|---|---|
| 80 | 1 | 140 | – |
| | 2 | 120 | 1,17 |
| | 4 | 160 | 0,875 |
| 200 | 1 | 460 | – |
| | 2 | 240 | 1,35 |
| | 4 | 270 | 1,70 |

TAB. 5.1: Speedup para sequências de entrada de 80 e 200 pares de base.

Os experimentos com sequências de entrada de 500 bases e 1.000 bases tiveram praticamente o mesmo ganho de desempenho que as buscas com entradas de 200 bases. Nas Tabela 5.2 e Tabela 5.3 são apresentados os tempos de execução em relação a divisão da base de dados e da sequência de entrada. Os *speedups* obtidos foram baixos, sendo o melhor, de 1,56, quando não utilizada a divisão da base de dados e dividindo-se a sequência de entrada em 4 segmentos.

| Divisão da base de dados | Divisão da sequência de entrada | Tempo (ms) | Speedup |
|---|---|---|---|
| 1 | 1 | 340 | – |
| | 2 | 280 | 1,21 |
| | 3 | 268 | 1,26 |
| | 4 | 239 | 1,42 |
| | 5 | 250 | 1,36 |
| 2 | 1 | 320 | 1,06 |
| | 2 | 300 | 1,13 |
| | 3 | 280 | 1,21 |
| | 4 | 270 | 1,25 |
| | 5 | 270 | 1,25 |
| 4 | 1 | 230 | 1,48 |
| | 2 | 280 | 1,21 |
| | 3 | 270 | 1,25 |
| | 4 | 287 | 1,18 |
| | 5 | 250 | 1,36 |

TAB. 5.2: Speedup para sequências de entrada de 500 pares de base em relação a divisão da base de dados e da sequência de entrada.

| Divisão da base de dados | Divisão da sequência de entrada | Tempo (ms) | Speedup |
|---|---|---|---|
| 1 | 1 | 571 | – |
| | 2 | 440 | 1,30 |
| | 3 | 415 | 1,38 |
| | 4 | 370 | 1,54 |
| | 5 | 420 | 1,36 |
| | 6 | 420 | 1,36 |
| | 7 | 430 | 1,36 |
| | 8 | 451 | 1,33 |
| 2 | 1 | 540 | 1,06 |
| | 2 | 420 | 1,36 |
| | 3 | 419 | 1,36 |
| | 4 | 409 | 1,40 |
| | 5 | 416 | 1,37 |
| | 6 | 426 | 1,34 |
| | 7 | 425 | 1,34 |
| | 8 | 416 | 1,37 |
| 4 | 1 | 775 | 0,74 |
| | 2 | 446 | 1,28 |
| | 3 | 415 | 1,38 |
| | 4 | 410 | 1,39 |
| | 5 | 400 | 1,42 |
| | 6 | 380 | 1,48 |
| | 7 | 420 | 1,36 |
| | 8 | 431 | 1,32 |

TAB. 5.3: Speedup para sequências de entrada com 1000 pares de base em relação a divisão da base de dados e da sequência de entrada.

Com sequência de entrada de 5.000 bases, o tempo de execução utilizando apenas uma *thread* e sem divisão da base de dados é de $2.400ms$. Dividindo-se a sequência de entrada em 3 partes e utilizando-se 3 *threads* obteve-se o tempo de $1.341ms$. Dividindo-se a base de dados em 4 fragmentos, e usando apenas 1 *thread* por fragmento, totalizando 4 *threads*, para busca no índice, extensão e alinhamentos, o tempo total é $2.100ms$, ou seja, um pequeno ganho comparando-se com o tempo sem divisão da base de dados. Porém, utilizando-se 4 *threads* para busca no índice, alinhamentos e extensão, o tempo é $1.630ms$. A Figura 5.1 apresenta o gráfico do desempenho em relação a divisão da base de dados e o número de *threads* utilizadas. É interessante notar que o ganho de desempenho ocorre até 3 *threads* e depois piora. Isto ocorre novamente devido ao tamanho da sequência de entrada, e também porque quando a base de dados é dividida em dois

ou quatro fragmentos, para cada fragmento são executados suas próprias *threads*: Logo se está designado a utilização de 4 *threads* na busca no índice, ou seja, a sequência de entrada é dividida em 4 segmentos, e nos alinhamentos numa base de dados dividida em 4 fragmentos, na realidade serão 16 *threads*, 4 em cada fragmento da base de dados. Sendo assim, a utilização de um computador com 8 unidades de processamento, e uma base de dados dividida em 4 fragmentos, teoricamente não oferece ganhos o uso de mais de 2 *threads* para busca no índice e alinhamentos.

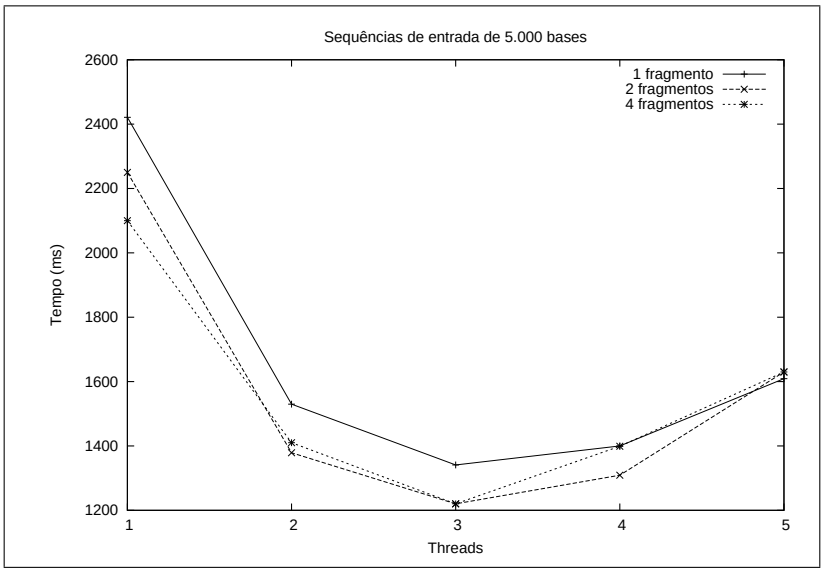


FIG. 5.1: Comparação de tempo em relação ao número de threads para uma sequências de entrada com 5000 bases.

Para melhor analisar a influência da quantidade de *threads* no tempo total da busca, foram feitas mais 16 execuções com as sequências de entrada de 5.000 bases, utilizando-se base de dados dividida em 4 fragmentos. Destas execuções, 8 foram feitas sem dividir as sequências de entrada e 8 dividindo-as em 2 segmentos, variou-se também o número de *threads* utilizadas nas extensões e alinhamentos. Na Figura 5.2 podem-se analisar os resultados destas execuções. Nota-se que o tempo de execução é melhor quando não há divisão da sequência de entrada, porém o melhor tempo é o que utiliza a maior quantidade de *threads* possível. Isto ocorreu porque o gargalo destas buscas são as extensões e alinhamentos, e disponibilizando-se mais *threads*, o tempo de busca diminuiu. Por fim, utilizando-se uma base de dados dividida em 4 fragmentos, sem dividir a sequência de entrada e 6 *threads* para as extensões e alinhamentos, conseguiu-se um tempo de $800ms$, ou seja, um ganho de 62% ou de 3 vezes sobre o tempo original. É um ganho interessante, porém ainda bem inferior a um *speedup* esperado de 8 devido às 8 unidades de

processamento disponíveis.

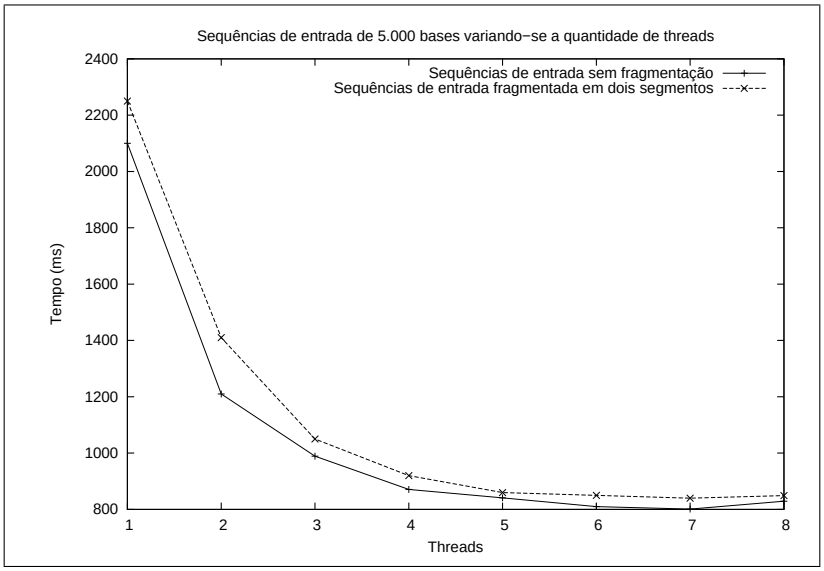


FIG. 5.2: Comparação de tempo com a utilização de mais threads no processo de busca para sequências de entrada com 5.000 bases.

As buscas para as sequências de entrada de 10.000 bases tiveram comportamentos bem similares as de 5.000 bases. O tempo de execução das 11 buscas sem paralelismo foi de $5.318ms$. Com a base de dados dividida em 4 fragmentos, utilizando-se 8 *threads* e dividindo-se a sequência de entrada em 3 partes, o tempo total foi de $1050ms$. Interessante notar que o tempo total para a sequência de entrada dividida de 1 a 4 segmentos foi praticamente o mesmo, variando de 1.050 a $1.070ms$. Ou seja, o ganho provido pelo paralelismo empata com a sobrecarga gerada. Outro ponto importante é que o ganho de desempenho dado pelo paralelismo chega até de 80%, com um *speedup* aproximadamente de 5. Na Figura 5.3 é exibido o gráfico do desempenho das execuções com sequências de entrada com 10.000 bases.

Para sequências de entrada com 50.000 bases, o tempo dsegmentos execução das 11 buscas sem paralelismo é de $31.499ms$. Utilizando-se a base de dados dividida em 4 fragmentos, 8 *threads* e dividindo-se a sequência de entrada em 4 segmentos, o tempo total é de $4.440ms$. Neste caso, o uso da divisão da sequência de entrada é mais vantajoso. Se utilizar a base de dados dividida da mesma forma e também 8 *threads*, mas sem utilizar a divisão da sequência de entrada, o tempo total fica em $5.259ms$. Ou seja, a utilização do paralelismo na divisão da sequência de entrada gera uma redução de 17% do tempo de processamento. Neste caso, utilizando-se todos os paralelismos possíveis, houve um ganho de 76% ou um *speedup* aproximado $7,10$, muito próximo ao *speedup* almejado de

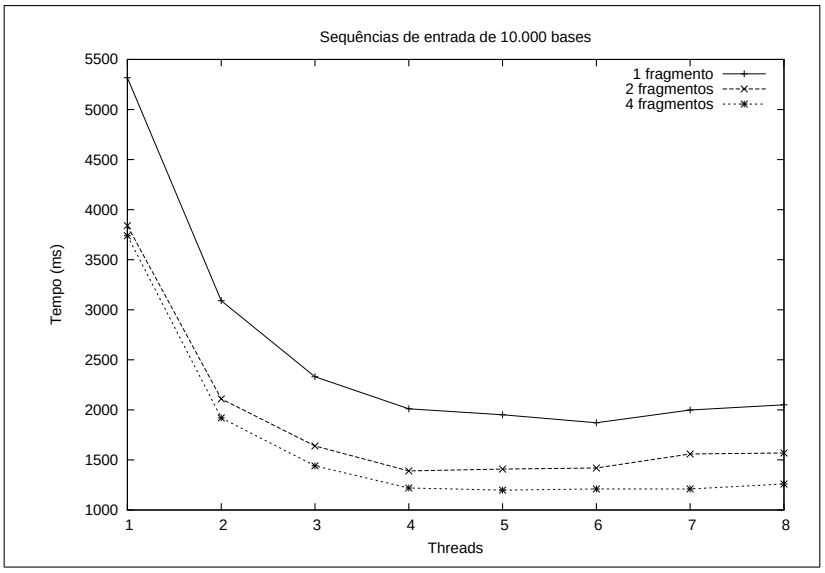


FIG. 5.3: Comparação de tempo em relação ao número de threads para sequências de entrada com 10.000 bases.

8. No gráfico da Figura 5.4 é possível ver a curva de tempo decrescente com o aumento do número de *threads* e com a maior divisão da base de dados. Nota-se que sem dividir a base de dados, só há ganho no tempo de busca com o uso de até 5 *threads*. Dividindo-se a base de dados em 4 partes, o ganho utilizando-se 7 ou 8 *threads* é praticamente o mesmo.

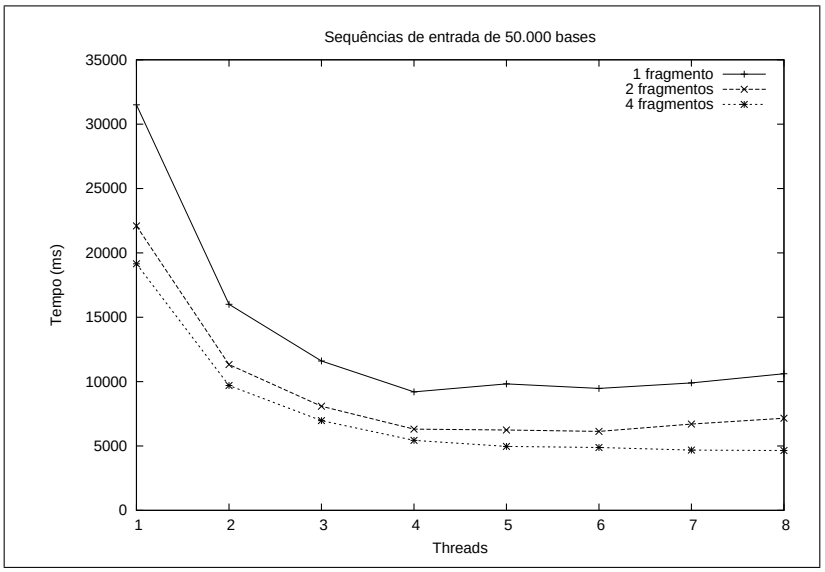


FIG. 5.4: Comparação de tempo em relação ao número de threads para sequências de entrada com 50.000 bases.

Com as sequências de entradas de 100.000 bases conseguiu-se um ganho de $87,6\%$, ou seja, um *speedup* de $8,06$ entre a execução sem paralelização ($75.610ms$) e a execução

dividindo-se a base de dados em 4 fragmentos, a sequência de entrada em 8 segmentos e utilizando-se 8 *threads* na busca no índice e nos alinhamentos ($9.380ms$). No gráfico apresentado na Figura 5.5 é possível observar o ganho de desempenho em relação à divisão da base de dados e em relação à divisão da sequência de entrada e ao número de *threads* utilizados.

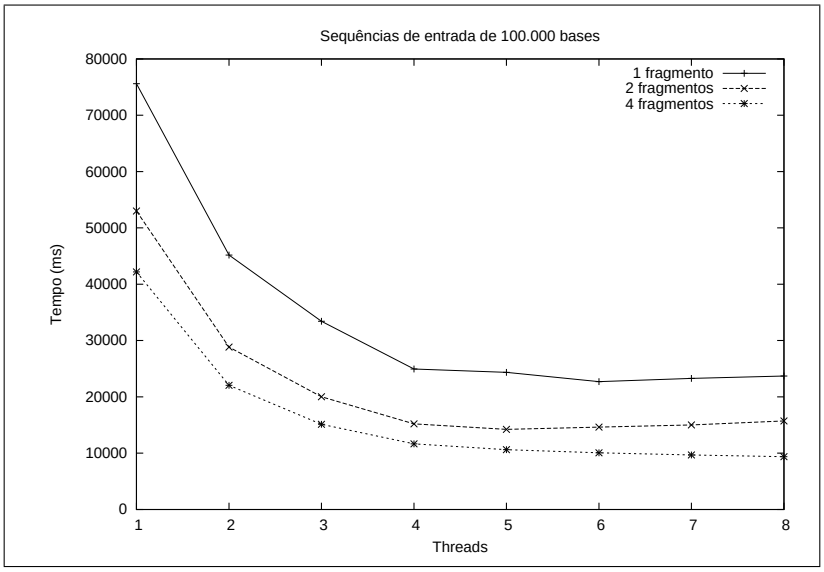


FIG. 5.5: Comparação de tempo em relação ao número de threads para sequências de entrada com 100.000 bases.

Para sequências de entrada com 500.000 bases, conseguiu-se alcançar um *speedup* de 8.7 entre a versão sem paralelização ($393.450ms$) e a execução dividindo-se a base de dados em 4 fragmentos, a sequência de entrada em 8 segmentos e utilizando-se 8 *threads* na busca no índice e nos alinhamentos ($45.120ms$). No gráfico apresentado na Figura 5.6 é possível observar o ganho de desempenho quando utilizado os paralelismos. Interessante notar que neste caso, o ganho de desempenho ultrapassou a quantidade de núcleos de processamento disponíveis.

O último conjunto de entradas executados é o que contém sequências com 1.000.000 bases. Devido ao tamanho, ao invés de se executar o experimentos com 11 sequências de entradas, os experimentos foram executados com apenas uma sequência. O tempo para a busca sem utilizar nenhum paralelismo foi de $76.909ms$. Utilizando a divisão da base de dados em 4 fragmentos, dividindo a sequência de entrada em 8 segmentos e utilizando 8 *threads* a execução da busca levou $8.780ms$, alcançando um *speedup* de $8,75$.

Esta seção teve o objetivo de demostrar os ganhos de desempenho em relação às técnicas de paralelização utilizadas. Na Figura 5.8 é exibido um gráfico com o *speedup* em

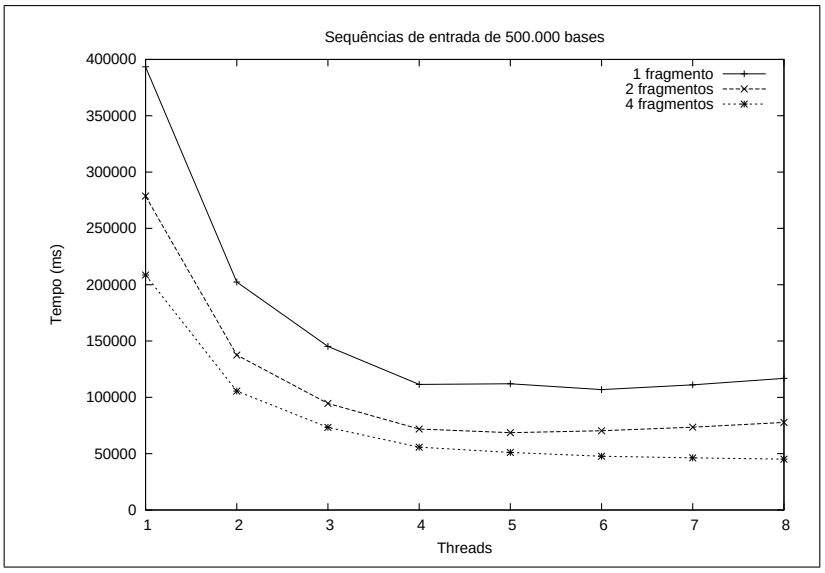


FIG. 5.6: Comparação de tempo em relação ao número de threads para sequências de entrada com 500.000 bases.

relação ao comprimento das sequências de entrada. Para sequências de entradas de até 5.000 bases o ganho da utilização dos paralelismos é bem baixa, gerando um *speedup* de apenas 2 e em casos em que a sequência de entrada é dividida em mais de 4 segmentos, o tempo total aumenta, ao invés de diminuir. Isto ocorre porque o tempo das buscas individuais para estas entradas são muito baixo, no máximo $200ms$, e o tempo para as sincronizações internas do sistema de busca impacta no tempo total destas buscas. Para as sequências de entrada com 10.000 bases já é possível ter um ganho significativo com a utilização das técnicas de paralelização. Com sequências de entradas de 50.000 bases e acima, os *speedups* são muito próximos a 8 e com sequências de entrada de 500.000 e 1.000.000 de bases os *speedup* superam este valor. Desta forma, mostrou-se a eficácia dos métodos de paralelização. Os métodos que apresentaram melhores resultados são os que, além da divisão da base de dados, também utilizam paralelismo tanto para fazer a busca no índice como para fazer os alinhamentos.

Na próxima seção é comparado o desempenho do protótipo desenvolvido e do BLAST.

### 5.1.2 COMPARAÇÃO COM OUTRAS FERRAMENTAS

Inicialmente pretendia-se comparar o desempenho do protótipo com o BLAST do NCBI, o BLAT, *MegaBLAST*. A versão do *MegaBLAST* que utiliza índices (MORGULIS, 2008) e o *Pattern Hunter* (MA, 2002). O uso de ferramentas que utilizam árvores de sufixo

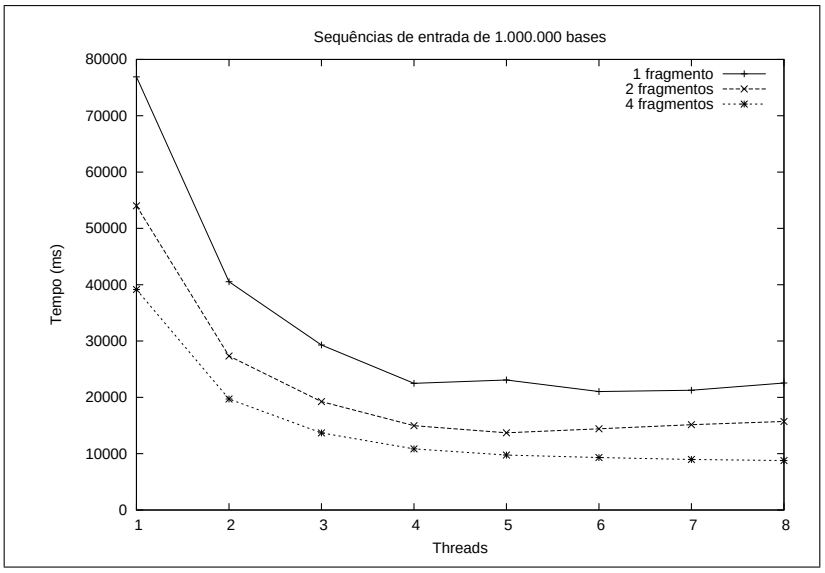


FIG. 5.7: Comparação de tempo em relação ao número de threads para sequências de entrada com 1.000.000 bases.

foi descartado por causa da quantidade de memória requerida por estas ferramentas, conforme foi descrito na Seção 3.4. Através de testes, verificou-se que o BLAT não consegue lidar com bases de dados maiores de 4 gigabytes, a base utilizada nos experimentos possui 4.25 gigabytes. Isto provavelmente ocorre porque ele foi desenvolvido e compilado para uma plataforma de 32 *bits*, onde o limite para a leitura de arquivos é 4 gigabytes ($2^{32}$ bytes). Segundo o trabalho de (MA, 2002), o *MegaBLAST* foi desenvolvido para ser eficiente em termos de tempo de execução, porém a qualidade dos resultados é inferior ao BLAST, isto se deve ao tamanho mínimo das sementes utilizadas, que é 28 bases. Por causa da qualidade inferior dos resultados, decidiu-se não executar experimentos com esta ferramenta. O trabalho de (MORGULIS, 2008) não pôde ser verificado pois a necessidade de memória é 4 vezes o tamanho da base, exigindo mais do que os 16 gigabytes disponíveis. E no caso do *PatternHunter* não foi possível obtê-lo para executar os experimentos. Desta forma, decidiu-se comparar o desempenho apenas com o BLAST. Através do artigo do *PatternHunter*, será estimado o ganho de desempenho em relação a esta ferramenta através do ganho descrito por ela em relação ao BLAST.

O BLAST foi executado com os mesmos conjuntos de sequências utilizados nos experimentos da Seção 5.1. Para cada conjunto de entrada, executou-se o BLAST duas vezes, uma sem utilizar paralelismo e outra utilizando-se a opção "*-a 8*" que informa quantos processadores devem ser utilizados na busca, neste caso serão utilizados 8 processadores. Desta forma foram feitas 20 execuções e o tempo de execução foi medido utilizando o

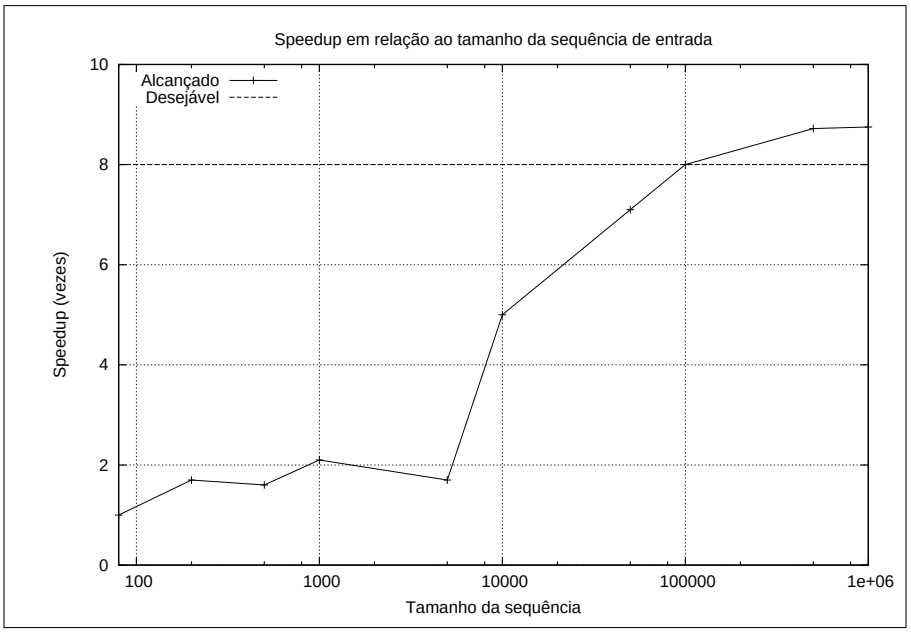


FIG. 5.8: Speedup em relação ao tamanho da sequência de entrada

utilitário *time*.

Primeiramente são comparados os tempos sem utilização de paralelismo tanto no BLAST quanto no protótipo. Os tempos da execução em relação ao tamanho da sequências de entrada são exibidos na Tabela 5.4. Nas execuções sem paralelismo, o *Genoogle* é de 16 a 42 vezes mais rápido que o BLAST. Esta comparação demonstra principalmente o ganho de desempenho no uso dos índices invertidos. Considerando-se que o *PatternHunter* (MA, 2002) é 20 vezes mais rápido que o BLAST, na maioria das buscas, o *Genoogle* empata ou supera este tempo.

| Tamanho das entradas (bases) | BLAST (ms) | Genoogle (ms) | Ganho (vezes) |
|---:|---:|---:|---:|
| 80 | 5.572 | 150 | 37,00 |
| 200 | 8.882 | 460 | 19,30 |
| 500 | 14.488 | 340 | 42,61 |
| 1.000 | 19.087 | 570 | 33,48 |
| 5.000 | 58.902 | 2.400 | 24,54 |
| 10.000 | 98.160 | 5.318 | 18,45 |
| 50.000 | 604.785 | 31.499 | 19,20 |
| 100.000 | 1.973.333 | 75.610 | 26,09 |
| 500.000 | 7.700.571 | 393.450 | 19,57 |
| 1.000.000 | 1.229.988 | 76.909 | 16,00 |

TAB. 5.4: Comparação de tempo entre o BLAST e protótipo desenvolvido sem a utilização de paralelismo.

A seguir são analisados os tempos de execução utilizando-se o paralelismo. Os resultados são apresentados na Tabela 5.5. Interessante notar que para sequências de entrada menores, até 5.000 bases os ganhos de tempo em relação ao BLAST não são tão bons, motivo já discutido na seção anterior. Para sequências maiores a diferença de tempo chega até aproximadamente 29 vezes no conjunto de sequências de entrada com 100.000 bases. Somando-se os tempos de todas as execuções realizadas, o ganho de desempenho do protótipo em relação ao BLAST chega a $26,66$ vezes.

Importante relembrar que outras ferramentas que utilizam índices invertidos, como o *PatternHunter*, não fazem uso de mais de um núcleo de processamento, desta forma, se o BLAST demora $60.000ms$ para executar uma busca sem paralelização, o *PatternHunter* demorará, seguindo a estimativa de 20 vezes mais rápido, $3.000ms$. Considerando-se que o ganho de desempenho do *Genoogle* com as técnicas de paralelização em relação ao BLAST sem paralelização é o tempo total do BLAST sem paralelização ($11.713.768ms$) dividido pelo tempo total do *Genoogle* com paralelização ($67.011ms$), o ganho é de $174,80$ vezes. Como o *PatternHunter* é 20 vezes mais rápido que o BLAST, o *Genoogle*, quando utilizadas as técnicas de paralelismo, é $8,74$ vezes mais rápido que *PatternHunter*, número ligeiramente superior a quantidade de processadores utilizados a mais.

| Tamanho das entradas (bases) | Blast (ms) | Genoogle (ms) | Ganho (vezes) |
|---:|---:|---:|---:|
| 80 | 1.061 | 150 | $7,00$ |
| 200 | 2.145 | 270 | $7,94$ |
| 500 | 3.170 | 210 | $15,09$ |
| 1.000 | 2.853 | 270 | $10,56$ |
| 5.000 | 10.387 | 1.341 | $7,74$ |
| 10.000 | 13.027 | 1.050 | $12,40$ |
| 50.000 | 78.067 | 4.440 | $17,58$ |
| 100.000 | 276.779 | 9.380 | $29,50$ |
| 500.000 | 1.206.212 | 45.120 | $26,73$ |
| 1.000.000 | 193.090 | 8.780 | $22,00$ |

TAB. 5.5: Comparação de tempo entre o BLAST e protótipo desenvolvido utilizando paralelismo.

Nesta seção os desempenhos do *Genoogle* e do BLAST foram comparados e foi feita uma estimativa de desempenho em relação ao *PatternHunter*. Os ganhos de desempenho em relação ao BLAST são bastante significativos, com ganhos superiores a 20 vezes em relação ao tempo de execução, tanto nas execuções sem utilização do paralelismo, como

as com utilização. Os ganhos estimados em relação ao *PatternHunter* também são bons e demonstram o ganho quando são utilizadas técnicas de paralelismo no processo de busca de sequências similares. Na próxima sessão é discutido a qualidade dos resultados obtidos pelo *Genoogle* em relação ao BLAST.

## 5.2 ANÁLISE DA QUALIDADE DOS RESULTADOS

A qualidade dos resultados foi analisada comparando os resultados do *Genoogle* com o do BLAST e verificando quais sequências foram identificadas como similares e qual porcentagem de sequências que não foram identificadas no *Genoogle* mas foram identificadas no BLAST.

Para cada sequência de entrada, criou-se uma coleção com os alinhamentos encontrados pelo BLAST. Foi verificado se estes alinhamentos foram encontrados pelo *Genoogle* e então contabilizados quantos alinhamentos foram encontrados e quantos não foram e por fim, foi calculada a porcentagem para cada faixa de *E-Value* variando de $10e^{-90}$ a $10e^{0}$.

Na Figura 5.9 é exibido o gráfico que mostra a proporção de alinhamentos encontrados pelo *Genoogle* em relação aos encontrados pelo BLAST de acordo com o *E-Value* do alinhamento. Este gráfico foi gerado a partir da verificação de quais alinhamentos encontrados pelo BLAST também foram encontrados pelo *Genoogle*. As sequencias utilizadas foram as sequencias utilizadas pelos testes de desempenho. Neste gráfico é possível observar que até o *E-Value* $10e^{-35}$ mais de 90% dos alinhamentos foram encontrados pelo *Genoogle*. Até o *E-Value* $10e^{-15}$ mais de 60% dos alinhamentos do BLAST foram encontrados e com *E-Value* $10e^{10}$ praticamente 55% dos alinhamentos foram encontrados. Acima deste *E-Value*, a quantidade de alinhamentos encontrados fica abaixo dos 40%. A Tabela 5.6 exibe as porcentagens das *HSP*s encontradas.

Analisando-se o gráfico da Figura 5.9, percebe-se uma boa qualidade do *Genoogle* no processo de busca para alinhamentos com *E-Value* inferior a $10e^{-25}$, isto se dá porque alinhamentos com este *E-Value* são normalmente longos, acima de 100 bases e consequentemente são encontrados mais facilmentes. Percebe-se que a partir do *E-Value* $10e^{-30}$ há uma queda na qualidade dos resultados onde estabiliza-se perto do *E-Value* 1. Alinhamentos com *E-Value* superior a $0,005$ não são alinhamentos em que pode-se inferir uma homologia próxima (ANDERSON, 1998). Desta forma, pode-se observar que o protótipo desenvolvido possui uma excelente qualidade até *E-Value* máximo de $10e^{-20}$, porém há uma queda na qualidade dos resultados até $10e^{-5}$ e acima deste valor é uma área em que

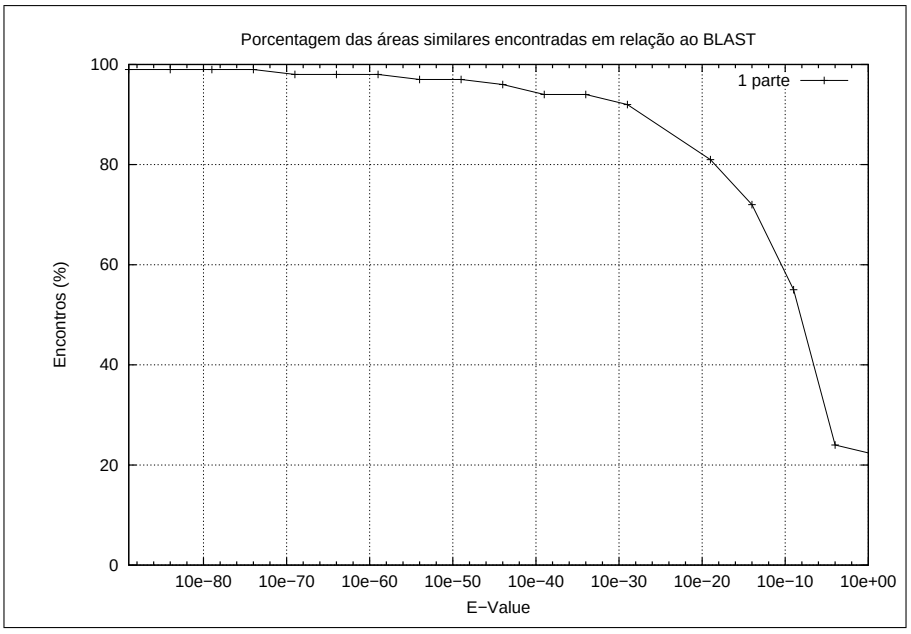


FIG. 5.9: Porcentagem das HSPs encontradas em relação ao BLAST

os alinhamentos não devem ser considerados para inferência de homologias.

Desta forma, mostrou-se que o *Genoogle* apresenta resultados com qualidade comparáveis ao BLAST quando o *E-Value* do alinhamento é representativo, demonstrando uma possível homologia entre as sequências alinhadas. Buscas mais sensíveis podem ser alcançadas modificando-se parâmetros da busca, como a distância máxima entre as informações obtidas do índice e o comprimento mínimo das HSPs. Modificando-se o comprimento mínimo para sequências com o *E-Value* inferior a $1e^{-5}$ a porcentagem de encontros para *E-Value* subiu para aproximadamente 80% e com um acréscimo mínimo de 3% no tempo de busca. Mesclando-se a técnica de indexação com as três técnicas de paralelização e as opções relacionadas a sensibilidade das buscas, o *Genoogle*, mesmo sendo um protótipo, mostrou-se uma ferramenta extremamente eficiente e com resultados de boa qualidade nos processos de busca de sequências similares.

Entre as técnicas de busca de sequências genéticas similares, a que possui maior sensibilidade são as que utilizam unicamente programação dinâmica, como implementações dos algoritmos de *Smith-Waterman* disponíveis no pacote *SSEARCH* (EBI, 2009). Por fazerem uma busca completa na base de dados, estas técnicas apresentam um desempenho muito pior do que as heurísticas apresentadas. Comparando o BLAST que utiliza heurísticas, com as ferramentas que utilizam indexação, estas oferecem uma melhora no desempenho, porém apresentam uma degradação na qualidade dos resultados obti-

| E-Value | Porcentagem encontrada |
|---|---|
| $10e^{-90}$ | 99 |
| $10e^{-85}$ | 99 |
| $10e^{-80}$ | 99 |
| $10e^{-75}$ | 99 |
| $10e^{-70}$ | 98 |
| $10e^{-65}$ | 98 |
| $10e^{-60}$ | 98 |
| $10e^{-55}$ | 97 |
| $10e^{-50}$ | 97 |
| $10e^{-45}$ | 96 |
| $10e^{-40}$ | 94 |
| $10e^{-35}$ | 94 |
| $10e^{-30}$ | 92 |
| $10e^{-20}$ | 81 |
| $10e^{-15}$ | 72 |
| $10e^{-10}$ | 55 |
| $10e^{-5}$ | 24 |
| $10e^{-0}$ | 22 |

TAB. 5.6: Comparação das HSPs encontradas em relação ao BLAST

dos. Trabalhos como o *miBLAST* (KIM, 2005) dizem que pretendem oferecer a mesma qualidade dos resultados que o BLAST, porém não apresentam quaisquer comparação de sensibilidade. As modificações propostas em (MORGULIS, 2008; TAN, 2005) no *MEGABLAST* (ZHANG, 2000) também não apresentam qualquer comparação de sensibilidade, deixando muito vago a sensibilidade destes métodos e ferramentas de busca de sequências similares. Entre as ferramentas que utilizam índices, apenas o *PatternHunter* (MA, 2002) apresenta uma comparação em relação ao BLAST, onde demonstra uma qualidade similar dos resultados.

# 6 CONCLUSÕES

Neste trabalho foram utilizadas e desenvolvidas técnicas e implementada-as em um protótipo denominado Genoogle cujo principal objetivo é otimizar buscas por sequências similares em base de dados de sequências genéticas. O protótipo utiliza a combinação de diferentes técnicas, usadas individualmente por outras ferramentas, com o intuito de diminuir o custo computacional da busca. Além disso, foi proposto o uso de técnicas de paralelização tendo em vista a ampla utilização de multiprocessadores e a necessidade de aproveitar esta capacidade computacional.

A busca por sequências similares foi otimizada inicialmente através da combinação das seguintes técnicas:

- Codificação das sequências;
- Uso de máscaras;
- Indexação da base de dados;
- Modificações no algoritmo de *Smith-Watermann*.

Os experimentos realizados demonstraram que a combinação destas técnicas melhora o desempenho das buscas sem aumentar exageradamente o uso de memória, como acontece nas ferramentas que utilizam indexação. O desempenho do *Genoogle* foi comparado com o desempenho do BLAST gerando um ganho deaté 42 vezes.

Além disso, foi realizada uma análise da qualidade dos resultados gerados comparando-os com os resultados gerados pelo BLAST. Foi demonstrado que o *Genoogle* apresenta resultados com qualidade comparáveis ao BLAST quando o *E-Value* do alinhamento é representativo e buscas mais sensíveis podem ser alcançadas modificando-se os parâmetros da busca.

Um melhor aproveitamento da capacidade computacional de multiprocessadores foi alcançado através da fragmentação da base de dados, da paralelização do alinhamento, e da paralelização do processamento da sequência de entrada. O uso dessas três técnicas combinadas gerou um *speedup* de mais de 8 vezes ao executar experimentos em um com-

putador com 8 processadores. Comparando-se a busca com paralelismo do *Genoogle* e do BLAST, mostrou-se que o *Genoogle* possui um desempenho superior de até 29 vezes.

Os experimentos demonstraram que a paralelização do processamento da sequência de entrada melhora o desempenho quando o comprimento da sequência é igual ou superior a 500 bases. A fragmentação da base de dados também apresenta uma limitação por causa do consumo de memória utilizado para armazenar as estruturas dos índices invertidos, porém o uso de 4 fragmentos para uma base de dados com mais de $4,25$ gigabytes não apresentou um custo de memória alto, necessitando de aproximadamente 2 gigas de memoria para a execução do *Genoogle* após a construção dos índices invertidos.

## 6.1 CONTRIBUIÇÕES

A seguir são listadas as contribuições deste trabalho:

- Uso combinado das diferentes técnicas relacionadas no início deste capítulo na busca por sequências similares;
- A utilização da indexação de forma a não extrapolar o consumo de memória no armazenamento do índice. Esta economia de memória foi alcançada através do uso de máscaras e da divisão das sequências da base de dados em sub-sequências de forma não sobreposta;
- O uso de indexação sem diminuir a sensibilidade dos resultados gerados;
- O aproveitamento da arquitetura de multiprocessadores através da paralelização do algoritmo de busca;
- O uso combinado da indexação com as técnicas de paralelização;
- O desenvolvimento de um software funcional para busca de sequências genéticas similares em base de dados utilizando-se as técnicas descritas.

## 6.2 TRABALHOS FUTUROS

Diversos trabalhos futuros podem ser derivados do que foi apresentado. Inicialmente, pode-se estudar o uso de técnicas para otimizar o consumo de memória necessário na indexação, como o uso de técnicas de compressão de índices invertidos descritas em (WITTEN, 1999).

Uma proposta de trabalho é a melhoria das interfaces de comunicação do protótipo. Entre elas, melhorias nas interfaces com o usuário via página *web* e modo texto. A implementação de uma comunicação via *Web-Services* (W3C, 2007) para automatizar o processo de recebimento e resposta de requisições seria de grande valia.

Outra proposta é um estudo aprofundado da influência dos parâmetros de busca na qualidade e no desempenho do *Genoogle*. A variação dos parâmetros disponíveis pode melhorar o tempo de busca e influenciar positivamente ou negativamente na qualidade dos resultados.

# 7 REFERÊNCIAS BIBLIOGRÁFICAS


ALBERTS, B., JOHNSON, A., LEWIS, J., RAFF, M., ROBERTS, K. e WALTER, P. *Biologia molecular da célula*. Artmed, Porto Alegre, 2 edition, 2004.

ALBRECHT, F. F. Técnicas para comparação e visualização de similaridades entre seqüências genéticas. Em *Anais...*, págs. 139–149, Blumenau, 2005. Seminário de Computação, Universidade Regional de Blumenau.

ALTSCHUL, F., GISH, W., MILLER, W., MYERS, E. e LIPMAN, D. A basic local alignment search tool. *Jornal of Molecular Biology*, 215(3):403–410, out. 1995.

ALTSCHUL, S. F., SCHAFFER, A. A., ZHANG, Z., MILLER, W. e LIPMAN, D. J. Gapped blast and psi-blast: a new generation os protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

ANDERSON, I. e BRASS, A. Searching dna databases for similarities to dna sequences: when is a match significant? *Bioinformatics*, 14(4):349–356, January 1998.

BECKSTETTE, M., HOMANN, R., GIEGERICH, R. e KURTZ, S. Fast index based algorithms and software for matching position specific scoring matrices. *BMC Bioinformatics*, 7:389–25, 2006.

BENSON, D. A., KARSCH-MIZRACHI, I., LIPMAN, D. J., OSTELL, J. e WHEELER, D. L. Genbank. *Nucleic Acids Research*, 35(Database issue):D21–D25, Dec. 2007.

BERNARDES, J. S. Detecção de homologias distantes utilizando hmms e informações estruturais. Mestrando em ciências em engenharia de sistemas e computação, COPPE/Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2004.

CAMERON, M. e WILLIAMS, H. Comparing compressed sequences for faster nucleotide blast searches. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 4(3):349–364, 2007. ISSN 1545-5963.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. e STEIN, C. *Introduction to Algorithms*. MIT Press, Massachusetts, 2 edition, 2001.

DARLING, A., CAREY, L. e FENG, W. The desing, implementation, and evaluation of mpiblast. Em *Proceddings...*, págs. 20–34, San Jose, CA, 2003. International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with the ClusterWorld Conference & Expo, LA-UR.

DELCHER, A., KASIF, S., FLETSCHMANN, R., PETTERSON, J., WHITE, O. e SALZBERG, S. Alignment of whole genomes. *Nucleic Acids Res.*, 27:2369–2376, 1999.

DELCHER, A. L., PHILLIPPY, A., CARLTON, J. e SALZBERG, S. L. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11): 2478–2483, 2002.

EBI. Ssearch tool for similarity and homology searching — ebi, 2009. URL `http://www.ebi.ac.uk/Tools/ssearch/`.

FOUNDATION, T. A. S. Apache lucene, 2009a. URL `http://lucene.apache.org`.

FOUNDATION, T. A. S. Apache tomcat, 2009b. URL `http://apache.apache.org`.

GILADI, E., WALKER, M. G., WANG, J. Z. e VOLKMUTH, W. Sst: an algorithm for finding near-exact sequences matches in time proportional to the logarithm of the database size. *Bioinformatics*, 18(6):873–879, 2001.

GOSPODNETIC, O. e HATCHER, E. *Lucene in Action*. Manning, Greewich, CT, 2005.

GRIBSKOV M, MCLACHLAN AD, E. D. Profile analysis: detection of distantly related proteins. *Proc Natl Acad Sci USA*, 13:4355–8, 1987.

GROSSMAN, D. A. e FRIEDER, O. *Information Retrieval*. Springer, Dordrecht, The Netherlands, 2004.

GUSFIELD, D. *Algorithms on strings, trees, and sequences*. Cambridge University Press, Cambridge, UK, 1997.

HERLIHY, M. e SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Massachusetts, 2008.

HOLLAND, R., DOWN, T., POCOCK, M., PRLI, A., HUEN, D., JAMES, K., FOISY, S., DRÄGER, A., YATES, A., HEUER, M. e SCHREIBER, M. Biojava: an open-source framework for bioinformatics. *Bioinformatics*, 24(18):2086–2097, 2008.

INTEL. Intel® parallel composer parallelization guide, 2008.

JIANG, X., ZHANG, P., LIU, X., S., S. e YAU, T. Survey on index based homology search algorithms. *The Journal of Supercomputing*, 40(2):185–212, may 2007. ISSN 0920-8542.

KALAFUS, K. J., JACKSON, A. R. e MILOSAVLJEVIC, A. Pash: Efficient genome-scale sequence anchoring by positional hashing. *Genome Research*, 14:672–678, 2004.

KENT, W. J. Blat - the blast-like alignment tool. *Genome Research*, 12:656–664, 2002.

KIM, JUNG, Y., BOYD, ANDREW, ATHEY, D., B., PATEL e M., J. miblast: Scalable evaluation of a batch of nucleotide sequence queries with blast. *Nucleic Acids Research*, 33(18):4335–4344, 2005.

KONGETIRA, P., AIGARAN, K. e OLUKOTUN, K. Niagara: A 32-way multithreaded sparc processor. *Hot Chips*, 16:21–29, 2005.

KORF, I., YANDELL, M. e BEDELL, J. *BLAST*. O'Reilly, Julho 2003.

MA, B., TROMP, J. e LI, M. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

MEIDANIS, J. e SETÚBAL, J. C. *Uma Introdução à Biologia Computacional*. UFPE-DI, Recipe, PE, 1994.

MORGULIS, A., COULOURIS, G., RAYTSELIS, Y., MADDEN, T. L., AGARWALA, R. e SCHÄFFER, A. A. Database indexing for production megablast searches. *Bioinformatics*, 24(18):1757–1764, 2008.

NCBI. Genbank, 2006. URL `http://www.ncbi.nlm.nih.gov/Genbank_aa/`.

NCBI. Ncbi human genome sequence databases, 2008a. URL `ftp://ftp.ncbi.nih.gov/genbank/genomes/H_sapiens/hs_phase3.fna.gz`.

NCBI. Ncbi reference sequence project (refseq), 2008b. URL `http://www.ncbi.nih.gov/RefSeq/`.

NEEDLEMAN, S. e WUNSCH, C. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3): 443–53, mar. 1970.

NING, Z., COX, A. J. e MULLINKIN, J. C. Ssaha: A fast search method for large dna databases. *Genome Research*, 11:1725–1729, 2001.

OF HEALTH, N. I. Public collections of dna and rna sequence reach 100 gigabases, 2005. URL `http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html`.

PEARSON, W. e LIPMAN, D. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci*, 85:2444–2448, 1988.

PHILLIPS, A. J. Homology assessment and molecular sequence alignment. *Journal of Biomedical Informatics*, 39:18–33, 2005.

RENEKER, J. e SHYU, C.-R. Refined repetitive sequence searches utilizing a fast hash function and corss species information retrievals. *BMC Bioinformatics*, 6(111):111–10, 2005.

SMITH, T. e WATERMAN, M. Identification of common molecular subsequences. *J.Mol.Biol.*, 147:195–197, 1981.

SPRACKLEN, L. e ABRAHAM, S. G. Chip multithreading: Opportunities and challenges. Em *Proceedings...*, San Francisco, California, Fevereiro 2005. Int¿l Symposium on High-Performance Computer Architecture.

STRACHAN, T. e READ, A. P. *Genética molecular humana*. Artmed, Porto Alegre, 2 edition, 2002.

TAN, G., XU, L., JIAO, Y., FENG, S., BU, D. e SUN, N. An optimized algorithm of high spatial-temporal efficiency for megablast. *Proceedings of the 2005 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, 2005.

THORSEN, O., SMITH, B., SOSA, C. P., JIANG, K., LIN, H., PETERS, A. e CHUN FENG, W. Parallel genomic sequence-search on a massively parallel system. *ACM International Conference on Computing Frontiers*, may 2007.

W3C. Web services activity, 2007. URL `http://www.w3.org/2002/ws/`.

WANG, C. e LLIOT J LEFKOWITZ. Ss-wrapper: a package of wrapper applications for similarity searches on linux clusters. *BMC Bioinformatics*, 5:171 – 9, 2004.

WHEELER, D., CHURCH, D., EDGAR, R., FEDERHEN, S., W., H. e MADDEN, T. Database resources of the national center for biotechnology information: update. *Nucleic Acids Res*, 32:35–40, 2004.

WITTEN, I. H., MOFFAT, A. e BELL, T. C. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann Publishers, 1999.

ZHANG, Z., SCHWARTZ, S., WAGNER, L. e MILLER, W. A greedy algorithm for aligning dna sequences. *JOURNAL OF COMPUTATIONAL BIOLOGY*, 7(1/2):203–214, 2000.