Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Curso de Bacharelado em Ciências da Computação

Daniel Haeser Rech
Rodrigo Pilatti

992ALIGN - UMA FERRAMENTA PARA ALINHAMENTO MÚLTIPLO DE SEQÜÊNCIAS DE DNA E PROTEÍNAS

Florianópolis
fevereiro de 2004

Daniel Haeser Rech

Rodrigo Pilatti

992ALIGN - UMA FERRAMENTA PARA ALINHAMENTO MÚLTIPLO DE SEQÜÊNCIAS DE DNA E PROTEÍNAS

Trabalho de conclusão de curso

do curso de Bacharelado em Ciências da Computação

Florianópolis

fevereiro de 2004

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Curso de Bacharelado em Ciências da Computação

992ALIGN - UMA FERRAMENTA PARA ALINHAMENTO MÚLTIPLO DE SEQÜÊNCIAS DE DNA E PROTEÍNAS

Daniel Haeser Rech

Rodrigo Pilatti

BANCA EXAMINADORA:

Prof. Dr. Antônio Augusto M. Fröhlich
Orientador

Prof. Dr. Edmundo Carlos Grisard
Banca

Charles I. Wust
Banca

Florianópolis
fevereiro de 2004

“Duas retas paralelas se cruzam no inferno!!”

Julio F. Szeremeta

Agradecimentos

Daniel

Aos meus pais.

Charles, Edmundo e Guto pela oportunidade e apoio dado.

Chuck Schuldiner (RIP) pela quebra da monotonia nas horas mais difíceis.

Rodrigo

Meus mais sinceros agradecimentos à todos os meus amigos, parentes e demais familiares que de uma forma ou de outra estiveram presentes durante toda a minha caminhada acadêmica.

Aos meus pais, Roseli e Ciro, e à minha irmã, Grasiele, pelo incontestável apoio, compreensão e carinho.

Ao professor Guto, por aceitar o desafio; Edmundo e Charles, pela atenção. dispensada à este trabalho.

Chuck Schuldiner, pelo exemplo de luta, garra e desempenho; suas músicas foram essenciais nas horas de maior estresse e solidão.

À uma pessoa em especial.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Gráficos

Resumo

As atuais ferramentas existentes para alinhamentos múltiplos de seqüências de DNA e proteínas apresentam em geral boas respostas, mas estão ainda muito longe de resultados perfeitos. Muitas vezes os biólogos precisam fazer intervenções manuais sobre os alinhamentos para obter uma solução que tenha sentido biológico. Também é necessário muitas vezes que os biólogos realimentem as ferramentas com diferentes parâmetros até que se obtenha uma boa solução. É necessário então aperfeiçoar os métodos existentes para alinhamentos múltiplos de forma que diminua o esforço humano para resolvê-los.

O presente trabalho apresenta uma implementação do método de alinhamento múltiplo progressivo. A implementação utiliza técnicas de programação paralela para aproveitar o poder de processamento de máquinas multiprocessadas. A paralelização é justificada pelo grande custo computacional demandado por alinhamentos múltiplos de seqüências.

Abstract

The existing tools for multiple alignments of DNA and protein sequences give good answers in general, but they are still far from giving perfect results. Often the biologists need to make manual interventions in the alignments to achieve a solution that makes biological sense. Also, often biologists need to feed these tools with different parameters until a good solution is achieved. So it's necessary to enhance the existing methods for multiple alignments in such a way that the human work to solve them is minimized.

The present work shows an implementation of the progressive multiple alignment method. The implementation uses techniques of parallel programming to take advantage of the computing power of multiprocessed computers. The parallelization is justified by the huge computational cost demanded by multiple sequence alignments.

1. Introdução

Nos últimos anos os estudos relacionados a genômica ganharam grande impulsão com o número crescente de genomas seqüenciados. A partir dessas informações, biólogos podem fazer as mais diversas análises com relação à evolução das espécies de seres vivos, e também no estudo de função de proteínas.

É preciso então que ferramentas sejam desenvolvidas para auxiliar o estudo e análise de toda essa informação gerada. Dado o grande volume de dados que compõem os genomas, é fundamental também que estas ferramentas apresentem resultados de forma rápida e eficiente. Torna-se necessário então o emprego de técnicas de programação paralela que aproveitem o poder da supercomputação.

O presente trabalho apresenta uma implementação de um alinhador múltiplo de seqüências de DNA e proteínas. Alinhamentos múltiplos são imprescindíveis para a comparação de genes e proteínas de espécies diferentes. Por este problema em geral demandar um grande tempo de processamento, a implementação tira proveito da supercomputação ao permitir que o programa seja executado em paralelo por vários processadores.

2. Biologia Computacional

As proteínas e os ácidos nucléicos são as moléculas fundamentais de todo os seres vivos. O ácido desoxirribonucléico (DNA) é o constituinte básico dos genes. Os genes de um organismo contêm todas as informações necessárias para o desenvolvimento de um organismo. O conjunto de genes de um organismo, por sua vez, recebe o nome de genoma. Se o genoma de um ser vivo for analisado, é possível saber as características que ele possui ou possuirá no futuro.

O problema é que a quantidade de informação contida nos genomas é muito grande. Assim, torna-se necessário aplicar a computação para ajudar a resolver os problemas relacionados ao estudo de genômica.

Biologia computacional é o estudo de técnicas matemáticas e computacionais aplicadas a problemas na área da biologia. É uma área cercada por algoritmos complexos e problemas ainda sem solução conhecida. Muitos problemas atualmente são resolvidos apenas por heurísticas que não apresentam soluções exatas, e muitas vezes estão bem longe da solução correta.

Entre os algoritmos mais conhecidos na área da ciência da computação utilizados na resolução de problemas da biologia, pode-se destacar a união de conjuntos disjuntos, árvores espalhadas, ordenação topológica, *hashing*, programação dinâmica, entre outros. A teoria da computação também tem sido utilizada para mostrar que os problemas advindos da área de biologia computacional são complexos e de difícil resolução.

Além da utilização de algoritmos clássicos da ciência da computação, diversos algoritmos novos foram criados para a resolução de problemas da biologia

molecular. A quantidade de artigos publicados sobre biologia computacional tem sido bastante considerável nos últimos anos.

Um problema freqüentemente encontrado em biologia computacional é o fato de que cientistas da computação na maioria das vezes estão mais interessados em resolver problemas abstratos, enquanto que biólogos estão mais interessados em resolver problemas práticos e concretos. Como conseqüência disto, muitas vezes os modelos criados pelos cientistas da computação ficam distantes da realidade dos problemas biológicos.

3. Biologia Molecular

Nesta parte, serão abordados alguns conceitos básicos de biologia molecular importantes para o entendimento dos aspectos e problemas computacionais presentes no decorrer deste trabalho.

Desde os tempos mais remotos, o ser humano observa que os seres vivos tendem a gerar seres com características semelhantes ao se reproduzirem. Cães produzem mais cães, e não produzem gatos. Os filhotes tendem a ser mais parecidos com os seus pais do que com outros indivíduos da mesma espécie.

Gregor Mendel, considerado o pai da genética moderna, obteve resultados significativos estudando este fenômeno. Após anos de trabalhos pacientes com ervilhas, ele observou que determinadas características da planta eram passadas por gerações, seguindo regras claras e bem determinadas. Como explicação às suas experiências, Mendel postulou que certos fatores, no caso, os *genes*, explicavam a determinação destas características.

As postulações de Mendel referentes a *genes* deixaram duas questões muito importantes em aberto: quais seriam as substâncias que compõem os genes e de que forma eles determinariam as características hereditárias de um ser vivo.

O primeiro questionamento começou a ser respondido em 1944, por Avery, MacLeod e McCarthy, através de um conjunto experiências envolvendo bactérias que causavam pneumonia em ratos. Estes pesquisadores puderam demonstrar que o *ácido desoxirribonucléico* (DNA) podia transformar as bactérias que tinham forma geneticamente estável, isto é, a transformação era passada de forma integral aos descendentes. Mais tarde, outros cientistas puderam comprovar que o DNA é o material

genético presente em quase todos os seres vivos. Cabe ressaltar que alguns vírus possuem seu material genético no RNA (ácido ribonucléico).

O segundo questionamento teve uma resposta parcial nos trabalhos de Garrod no início do século XX, e de outros pesquisadores que culminaram na hipótese um "gene – uma enzima". Segundo esta teoria, cada gene seria responsável por uma enzima no organismo. Enzimas são proteínas que catalisam as reações químicas nos seres vivos, tendo papel crucial no metabolismo. Esta hipótese em questão foi elaborada com base em diversas situações onde uma mutação causa falhas no metabolismo – e tais falhas são provocadas pela ausência de uma certa enzima. Hoje, sabe-se que todas as proteínas produzidas por um ser vivo são fabricadas a partir dos genes.

O papel fundamental dos ácidos nucléicos e proteínas nos processos vitais foi sendo conhecido dessa forma. A partir disso, esforços para se entender a estrutura química das moléculas foram sendo intensificados.

Tanto proteínas como ácidos nucléicos são *polímeros* – moléculas formadas pela ligação de várias unidades semelhantes – os *monômeros*.

No caso do DNA, os monômeros são os *nucleotídeos*. Quatro deles ocorrem naturalmente no DNA e são simbolizados pelas letras A, C, G, e T, que identificam as bases que os compõem. Dessa forma, uma molécula de DNA pode ser descrita completamente se for dada a seqüência de nucleotídeos que a compõem.

	Letra	***Nome***
1	A	Adenina
2	C	Citosina
3	G	Guanina
4	T	Timina

Tabela 1 - bases que compõem o DNA

Nas proteínas, os monômeros são os *aminoácidos*. Vinte aminoácidos ocorrem naturalmente em proteínas e encontram-se listados na tabela [2]. Como no caso dos ácidos nucléicos, grande parte das propriedades de uma proteína pode ser deduzida a partir da seqüência de aminoácidos que a compõe.

	Letra	***Abreviatura***	***Nome***
1	A	Ala	Alanina
2	C	Cys	Cisteína
3	D	Asp	Ácido Aspártico
4	E	Glu	Ácido Glutâmico
5	F	Phe	Fenilalanina
6	G	Gly	Glicina
7	H	His	Histidina
8	I	Ile	Isoleucina
9	K	Lys	Lisina
10	L	Leu	Leucina
11	M	Met	Metionina
12	N	Asn	Asparagina
13	P	Pro	Prolina
14	Q	Gln	Glutamina
15	R	Arg	Arginina
16	S	Ser	Serina
17	T	Thr	Treonina
18	V	Val	Valina
19	W	Trp	Triptofano
20	Y	Tyr	Tirosina

Tabela 2 - Os vinte aminoácidos naturais

A determinação da seqüência exata dos monômeros (ou *resíduos*) que formam uma molécula de ácido nucléico ou proteína chama-se *seqüenciamento* desta molécula. Apenas nas últimas décadas os processos laboratoriais de seqüenciamento foram sendo aperfeiçoados a ponto de poderem ser executados a custo e tempo

razoáveis. A partir daí houve uma grande produção de seqüenciamentos que causaram o aparecimento de *repositórios* (bases de dados biológicos) geralmente mantidos por entidades públicas, para o armazenamento de tais informações. Hoje em dia existem várias destas bases de dados disponíveis para consulta de forma eletrônica. Dentre os mais conhecidos pode-se citar o GenBank (NCBI-NIH, Estados Unidos), Protein Identification Resource (PIR, Estados Unidos), DNA Data Bank Of Japan (Japão), European Molecular Biology Laboratory (EMBL).

Paralelamente ao acúmulo de dados, a quantidade de métodos algorítmicos para o tratamento das informações moleculares também aumentou. A própria tecnologia dos computadores, que cresce de forma vertiginosa a cada dia, contribuiu para isso.

Atualmente, além de buscas em bases de dados por seqüências que satisfaçam a determinadas características, são também efetuados, com a ajuda dos computadores, estudos de moléculas para a determinação das suas características físico-químicas, sua configuração espacial, suas propriedades fisiológicas e sua história evolutiva.

3.1. Ácidos Nucléicos e Proteínas

Em 1953, Watson e Crick desvendaram as linhas gerais da conformação tridimensional do DNA, concluindo que o DNA é formado por duas fitas dispostas em forma de hélice – a *hélice dupla* do DNA. Cada uma destas fitas é um polinucleotídeo, conforme descrito anteriormente. Essas duas fitas são mantidas juntas através de pontes de hidrogênio que são ligações mais fracas que as ligações covalentes que mantém as fitas inteiras.

Agora, atendo-se um pouco ao aspecto da relação entre as fitas. Cada fita em particular, tem duas extremidades livres, chamadas de 3' e 5', numa alusão aos átomos de carbono que ficam livres no açúcar que compõem cada nucleotídeo. Neste contexto, duas observações se fazem importantes:

1) A extremidade 3' de uma fita corresponde à extremidade 5' da outra. Devido a esse fato, costuma-se dizer que as fitas são *antiparalelas.*

2) Um A numa fita corresponde a um T na outra fita, do mesmo modo que um C sempre corresponde a um G (vide figura [1]). Diz-se também que A e T são *bases complementares*, assim como C e G. Com base nisto, uma seqüência de nucleotídeos numa das fitas determina de forma completa uma molécula de DNA, propriedade esta que permite a replicação do DNA.

5' ----GGTACCAC---- 3'

3' ----CCATGGTG---- 5'

Figura 1 - Um exemplo de trecho de DNA de fita dupla

A convenção mundialmente adotada para se representar moléculas de DNA é se escrever apenas uma das fitas na direção 5' → 3'. A seqüência da outra fita pode ser facilmente determinada *invertendo* as ordens das bases e *complementando* cada uma delas. Ou seja: substitui-se A por T e C por G e vice-versa. Chama-se de *complemento reverso* de uma seqüência de bases *w* a seqüência w^{CR} obtida realizando-se estas operações em *w*. Como um exemplo: $CTATCG^{CR}$ = CGATAG.

Esta mesma convenção de se escrever na direção 5' → 3' é aplicada quando se trabalha com o RNA (ácido ribonucleico). O RNA difere do DNA por

apresentar-se geralmente em fita única e por usar a base U (uracil) em lugar de T (timina).

Em se tratando de proteínas, a convenção é iniciar a escrita partindo do radical amino (NH_3) livre. São as proteínas as moléculas responsáveis diretamente pela vida de uma célula. Algumas proteínas são classificadas como sendo *estruturais*, pois são responsáveis pela constituição das paredes celulares, cabelo, unhas e vários tipos de tecido. Outras agem como catalisadoras de reações específicas: são as *enzimas*.

A *estrutura primária* de uma molécula é composta pela seqüência de aminoácidos que compõem uma proteína. A *estrutura secundária* é dada por interações entre os aminoácidos, que podem formar hélices ou folhas planas em certos trechos da molécula. A configuração tridimensional completa, incluindo as pontes de hidrogênio e ligações fracas entre resíduos, constitui a *estrutura terciária* da proteína. Além disso tudo, certas proteínas são formadas por mais de uma cadeia de peptídeos; neste caso, a especificação de como as unidades sub-proteicas são ligadas entre si para a formação da proteína constitui a sua *estrutura quaternária*. Pode-se citar como um exemplo a hemoglobina, que possui quatro sub-unidades iguais duas a duas chamadas de alfa e beta. Embora as cadeias de alfa e beta sejam polipeptídeos, não se poderia chamá-las de proteínas, pois elas não apresentam nenhuma atividade metabólica. A função da hemoglobina é transportar oxigênio e gás carbônico no sangue, ajudando também a controlar o pH sanguíneo, mas nenhuma das suas sub-unidades desempenha estas funções por si só.

Ao longo dos anos, as proteínas vêm sendo usadas para investigar a evolução das espécies. Isso se deve ao fato de que é comum se encontrar proteínas exercendo um mesmo tipo de função em organismos diferentes. Por exemplo, todos os mamíferos possuem hemoglobina em seu sangue, embora não seja exatamente a mesma

molécula. Pode-se tirar muitas conclusões relacionadas a aspectos de origem através do estudo da seqüência de aminoácidos que constituem a hemoglobina de espécies diferentes.

Estruturas parecidas (sejam proteínas ou características morfológicas) podem ser *homólogas* ou *análogas* em seres vivos de espécies distintas. Quando estruturas originam-se de um ancestral comum, diz-se que estas estruturas são homólogas; caso o contrário, estas estruturas são análogas. Voltando ao exemplo utilizado da hemoglobina dos mamíferos, há fortes evidências de que estas proteínas sejam homólogas e não simplesmente análogas.

Mesmo entre proteínas homólogas, pode se encontrar grandes diversidades em sua estrutura primária. Apesar disso, existem trechos de seqüências que pouco ou nada mudam. Quando uma região de uma seqüência muda pouco de uma proteína pra outra, dizemos que esta região é uma região *conservada*. Enzimas que são responsáveis pela catalização de reações específicas possuem em sua estrutura tridimensional certas "bolsas" onde os reagentes se encaixam, facilitando a reação. Estes locais são chamados de *sítios ativos* e geralmente são bem conservados nas diferentes versões da proteína para diferentes espécies. Regiões de grande conservação, sejam elas ativas ou não, têm papéis fundamentais na atividade de uma proteína. Tais regiões são conhecidas como *motivos*.

3.2. Síntese de Proteínas

O *genoma* de qualquer organismo contém toda a informação genética necessária para a realização de todas as funções vitais. Em geral, ele é composto por *cromossomos* que são essencialmente moléculas de DNA extremamente longas empacotadas com a ajuda de certas proteínas. Toda a célula de qualquer organismo possui um determinado número de pares de cromossomos. As células que possuem estas características são chamadas de *diplóides*. Mas também existem células que se destinam a reprodução que possuem apenas um cromossomo em cada par; estas são chamadas de *haplóides*. A tabela [3] fornece alguns dados sobre o genoma de algumas espécies.

Espécie	**Número de Cromossomos (haplóide)**	**Tamanho Total do Genoma (pares de bases)**
Bacteriófago ? (vírus)	1	5×10^4
Escherichia coli (bactéria)	1	5×10^6
Saccharomyces cerevisiae (levedura)	16	1×10^7
Caenorhabditis elegans (verme)	6	1×10^8
Drosophila melanogaster (mosca)	4	2×10^8
Homo sapiens (homem)	23	3×10^9

Tabela 3 - o tamanho do genoma em algumas espécies.

Os *genes* – que são os trechos de DNA de um organismo, ou seja, são partes do cromossomo - são os responsáveis pela síntese de todas as proteínas de um ser vivo.

De forma mais geral, uma proteína é fabricada da seguinte forma (figura [2]): o mecanismo celular de síntese reconhece um certo gene dentro de um cromossomo, dando início à fase de *transcrição* deste gene. A transcrição por si consiste na produção de uma molécula de RNA, chamada de RNA *mensageiro* (mRNA) que é uma cópia da seqüência do gene, exceto pelo fato de se ter U no lugar de T no gene. O gene, sendo um trecho de DNA, possui duas fitas. Apenas uma delas é copiada.

O mRNA produzido agora será então usado na fase de *tradução*. Nesta parte, a seqüência de bases que forma o mRNA é lida para se formar a proteína. As bases são lidas 3 a 3, sendo que cada tripla (chamada de *codon*) especifica um aminoácido.

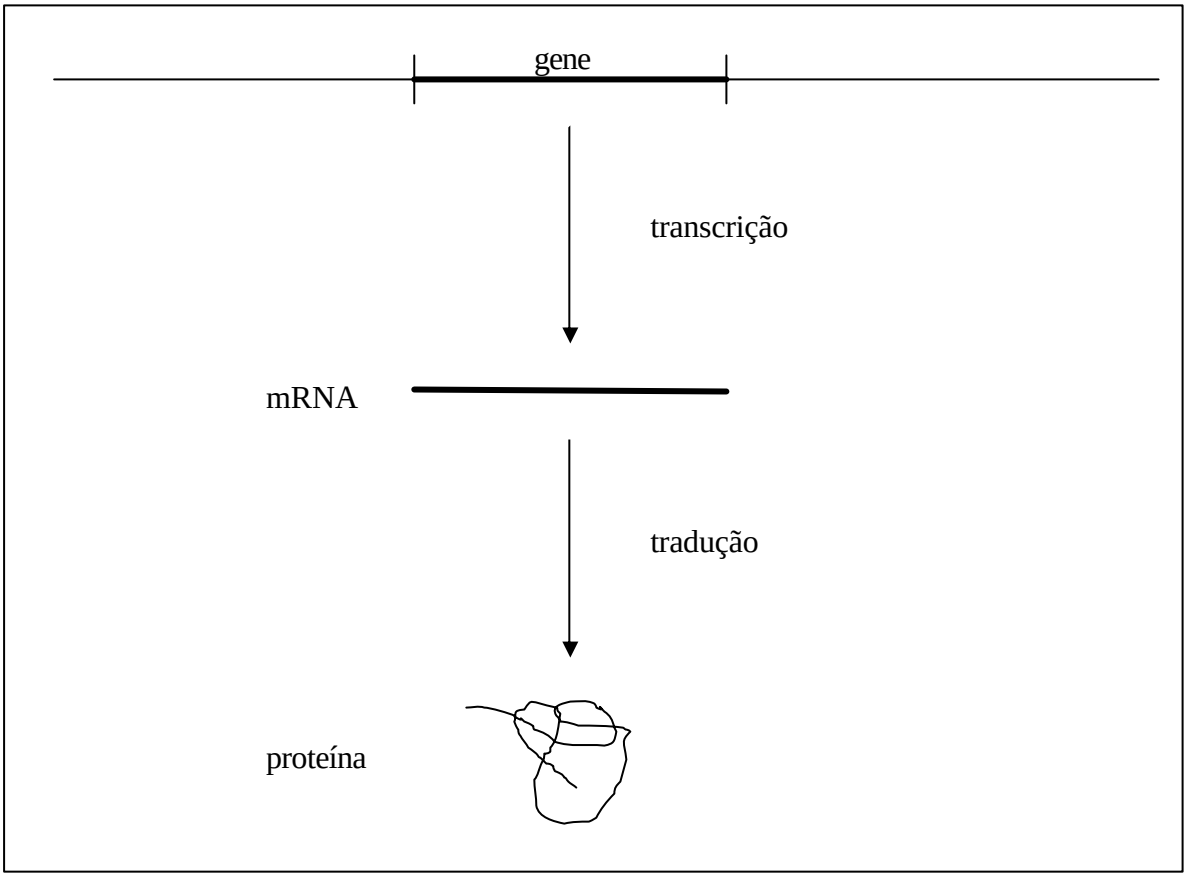

Figura 2 - Síntese proteica.

A tabela [4] abaixo mostra o *código genético*, que associa a cada códon um aminoácido correspondente. O mesmo código é usado em quase todos os seres vivos, com pequenas variações num pequeno número de organismos que se utilizam de um código diferente. Vale salientar que três dos códons não especificam nenhum tipo de aminoácido, sendo usados pela célula unicamente para sinalizar o final do processo de tradução. A figura [3] mostra um exemplo de tradução.

Primeira Posição	Segunda Posição				Terceira Posição
	G	A	C	T	
G	Gly	Glu	Ala	Val	G
	Gly	Glu	Ala	Val	A
	Gly	Asp	Ala	Val	C
	Gly	Asp	Ala	Val	T
A	Arg	Lis	Thr	Met	G
	Arg	Lis	Thr	Ile	A
	Ser	Asn	Thr	Ile	C
	Ser	Asn	Thr	Ile	T
C	Arg	Gln	Pro	Leu	G
	Arg	Gln	Pro	Leu	A
	Arg	His	Pro	Leu	C
	Arg	His	Pro	Leu	T
T	Trp	-	Ser	Leu	G
	-	-	Ser	Leu	A
	Cys	Tyr	Ser	Phe	C
	Cys	Tyr	Ser	Phe	T

Tabela 4 - O código genético.

ATG	GTG	CAC	CTG	ACT	GAT	GCT
Met	Val	His	Leu	Thr	Asp	Ala

Figura 3 - um exemplo de tradução.

O *ribossomo* é o responsável pela tradução do mRNA e a produção da proteína equivalente, com a participação do tRNA (RNA *transportadores*). Estes tRNA estão ligados, de um lado, a um ou mais códons específicos e, de outro, ao aminoácido correspondente, trazendo todos os aminoácidos para dentro do ribossomo na seqüência correta para a montagem da proteína.

Até hoje, pesquisadores estão intrigados sobre o fato de como a célula consegue reconhecer os genes dentro do cromossomo, pois nem todo o trecho de DNA é expresso como proteína. De fato, em organismos superiores, grande parte do DNA não tem função conhecida ou associada.

A questão do reconhecimento de genes está sendo amplamente estudada por pesquisadores do mundo inteiro, mas uma resposta em completo ainda não existe. Sabe-se que a região imediatamente anterior ao gene, denominada região promotora ou simplesmente *promotor*, é a parte que é reconhecida pelo mecanismo celular de transcrição. Partes desta região promotora podem ser mais conservadas do que outras, recebendo nomes especiais. Assim, segundo [3], a caixa de *Pribnow* é uma seqüência relativamente bem conservada em promotores de organismos *procariotos*, que são os organismos mais simples conhecidos, caracterizados pela ausência da membrana nuclear em suas células. Pode-se citar como exemplo de seres procariotos as bactérias e as algas azuis. Devido a sua simplicidade, facilidade de manejo, cultivo e ciclo de vida curto, as bactérias têm sido utilizadas de forma ampla nas pesquisas de engenharia genética.

Ao contrário dos organismos procariotos, existem os organismos eucariotos. Estes por sua vez são dotados de uma membrana nuclear responsável pela separação do núcleo – onde se encontram os cromossomos – do restante da célula. Todo o processo de metabolismo celular, incluindo-se a síntese de proteínas, é bem mais

complexo nos seres eucariotos. A presença de longos trechos não-traduzidos na maioria de seus genes e a existência de processos de modificação pós-tradução em certas proteínas constituem duas diferenças importantes. Respectivamente são usadas as denominações *exon* e *intron* para a designação de trechos traduzidos e não-traduzidos de um gene. O nome exon é uma alusão ao fato de que o trecho *sai* do núcleo para ser traduzido no ribossomo, enquanto os introns, por sua vez, ficam dentro do núcleo celular e não são expressos. Estimativas levam a crer que existe cerca de dez vezes mais DNA nos introns do que nos exons em um gene. Cabe salientar que, como nos procariotos, os eucariotos possuem reconhecimento de genes através de seus promotores.

Uma descoberta muito importante, feita por F. Jacob e J. Monod, foi a de que vários genes podem estar sob o controle de um mesmo promotor. O conjunto resultante da união de um promotor e seus genes associados recebe o nome de *operon.* Monod e Jacob estudaram o operon *lac* na bactéria intestinal *Escherichia coli*, relacionado à digestão de lactose por este microorganismo. Este com certeza foi um dos primeiros trabalhos a respeito de regulação gênica, e seus autores foram agraciados com o Prêmio Nobel por seus achados e contribuições.

4. Filogenia

No estudo da evolução das espécies de seres vivos, torna-se necessário procurar-se por homologias entre as espécies envolvidas. Homologias são similaridades resultantes da herança de um ancestral comum.

Filogenia é o estudo da evolução dos seres vivos. O seu objetivo é encontrar uma árvore evolucionária das espécies de seres vivos, assumindo que diferentes espécies derivam de um ancestral comum.

Os primeiros estudos de filogenia tentavam encontrar a árvore de evolução pelo fenótipo das espécies. Contudo, esse método pode levar a erros, tendo em vista que nada impede que duas espécies sem um ancestral comum venham a desenvolver uma característica em comum. Estudando somente o fenótipo, pode-se chegar a uma conclusão errada de que essas duas espécies teriam uma origem em comum.

Com a atual "onda" de seqüenciamento de genomas, o DNA inteiro de espécies está ficando disponível em bases de dados. Com toda essa informação gerada é possível abandonar o estudo de filogenia por fenótipo e passar a usar o genótipo dos seres. Embora isto faça muito mais sentido biologicamente, essa alternativa também trás um problema. A quantidade de informação a ser analisada é imensamente maior quando se estudam os genes dos seres vivos.

5. Alinhamentos Múltiplos

Um alinhamento de seqüências de DNA ou proteínas de diferentes espécies de seres vivos é uma hipótese de homologia entre as bases (ou nucleotídeos) que constituem os genes (ou proteínas) sendo comparados naqueles seres vivos. Alinhamentos podem ser tratados como modelos que podem ser usados para testar hipóteses evolucionárias e, portanto, são importantes para estudos de filogenia.

Encontrar um bom alinhamento entre as seqüências de várias espécies é geralmente uma tarefa difícil e envolve vários problemas. A similaridade pode ser alta em certos trechos das seqüências e baixa em outras partes. As seqüências podem ser de tamanhos variados ou muito diferentes entre si.

A maioria das técnicas de alinhamento múltiplo deriva do algoritmo de programação dinâmica.

5.1. Algoritmo de Programação Dinâmica

Para se encontrar um alinhamento entre duas seqüências de DNA, a técnica mais utilizada é baseada no algoritmo de programação dinâmica. Esta técnica busca o melhor alinhamento entre duas *strings* de caracteres. Ela se baseia na construção de uma matriz de comparação de prefixos das duas seqüências a serem alinhadas. Ao tentar alinhar sucessivamente os prefixos, o algoritmo atribui um *score* (valor de pontuação) para cada um dos prefixos. O *score* é computado de forma a penalizar as diferenças entre os prefixos e privilegiar as similaridades.

Por exemplo, dado duas seqüências S1 = TACCA e S2 = AGA, um possível alinhamento entre S1 e S2 seria:

```
S1)  TACCA
S2)  _AG_A
```

Para alinhar S1 e S2, foi preciso inserir *gaps* (espaços) para que o alinhamento fizesse sentido, produzindo um efeito negativo no *score* total do alinhamento. Este alinhamento também produziu um *mismatch* (alinhamento de dois caracteres diferentes) ao alinhar C com G na terceira posição. Assim como *gaps*, *mismatches* também produzem um efeito negativo sobre o *score* total do alinhamento.

Existem extensões do algoritmo de programação dinâmica que introduzem alguns conceitos que aproximam mais os resultados da realidade biológica. Um exemplo é a penalização por abertura de *gaps*. Nessa extensão do algoritmo, é dada prioridade à ocorrência de *gaps* unidos. Isso é justificado pelo fato de ser muito mais fácil ocorrer uma mutação que acrescente ‘n’ genes ao mesmo tempo do que uma mutação que acrescente ‘n’ genes disjuntos em tempos diferentes. Para exemplificar, entre os seguintes alinhamentos, teoricamente o segundo é muito mais plausível biologicamente:

Alinhamento 1:

```
Seqüência A: ACCGTTA
Seqüência B: A_C_T_A
```

Alinhamento 2:

```
Seqüência A: ACCGTTA
Seqüência B: AC___TA
```

A complexidade de tempo do algoritmo de programação dinâmica puro para comparação de duas seqüências é n^2 com relação ao tamanho das seqüências. É possível generalizar o algoritmo para que ele resolva o alinhamento para um número 'n' fixo de seqüências. Contudo, a complexidade exponencial de memória e processamento torna a aplicação do algoritmo impraticável neste caso.

O problema de alinhar um número 'n' qualquer de seqüências é um NP-Completo se for utilizado o princípio do algoritmo de programação dinâmica. Os NP-Completos são problemas que não possuem solução conhecida que apresente uma resposta em um tempo polinomial ou menor com relação ao tamanho da entrada de dados. Na prática isto quer dizer que o tempo de processamento seria extremamente alto e impraticável.

Para resolver esse problema de complexidade, algumas heurísticas foram desenvolvidas para diminuir a complexidade de tempo de alinhamentos com mais de duas seqüências. A abordagem freqüentemente mais utilizada é o alinhamento múltiplo progressivo.

5.2 Alinhamento Múltiplo Progressivo

O alinhamento múltiplo progressivo[1] é uma abordagem para o alinhamento de várias seqüências que tenta inferir uma evolução entre as espécies sendo comparadas. Podemos dividi-lo em 3 fases principais:

5.2.1 Alinhamento par a par

Nesta fase é construída uma matriz de distâncias dos alinhamentos de todos os possíveis pares do conjunto de seqüências de entrada. A distância do alinhamento de cada par é obtida utilizando-se o algoritmo de programação dinâmica. Para um conjunto de 'n' seqüências de entradas, o algoritmo de programação dinâmica será invocado n(n-1)/2 vezes. Isto torna a fase de comparação par-a-par a mais crítica em relação ao tempo de processamento em alinhamentos múltiplos progressivos.

5.2.2. Construção da árvore guia

Obtida a matriz de distâncias, cria-se uma árvore guia que mais tarde conduzirá o alinhamento múltiplo propriamente dito. Existem diversos métodos para a construção desta árvore. O algoritmo neighbor-joining [5] (vide figura [4]) é geralmente considerado um dos métodos que apresentam resultados biológicos mais coerentes. Este algoritmo junta sucessivamente em uma árvore as seqüências mais próximas em ramos mais próximos, de forma iterativa, até que a árvore guia contenha todas as seqüências.

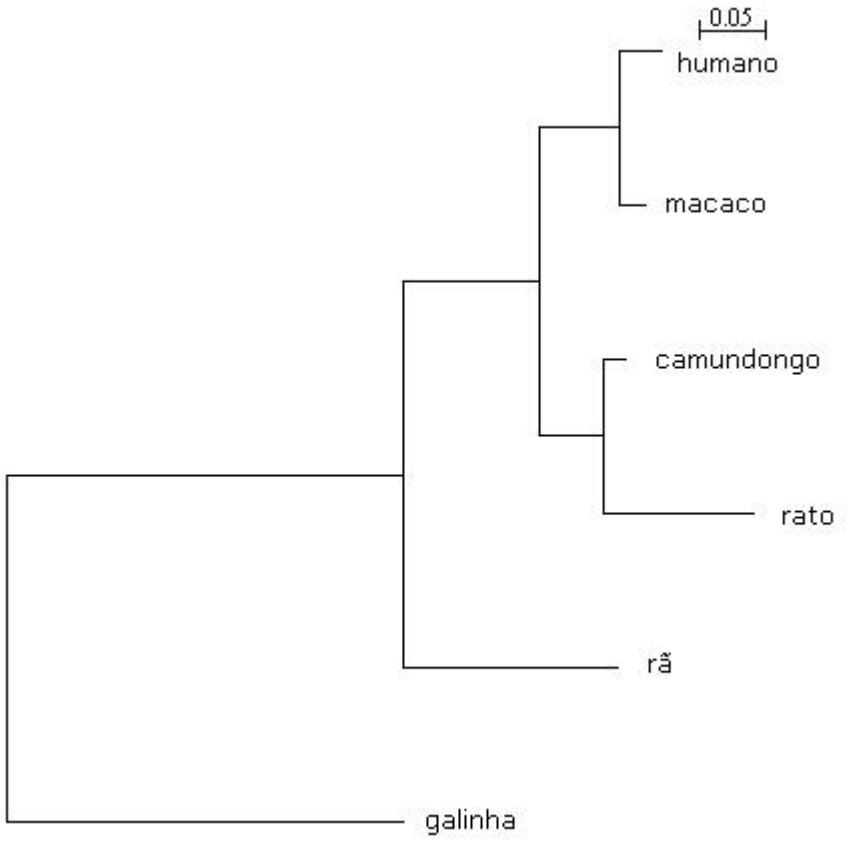

Figura 4 - Exemplo de árvore criada pelo algoritmo neighbor-joining.

5.2.3. Alinhamento Progressivo

O alinhamento progressivo produz o alinhamento múltiplo propriamente dito a partir da árvore guia criada. O alinhamento é feito progressivamente, de forma que as seqüências em ramos mais próximos sejam alinhadas primeiramente.

Para alinhar uma seqüência com um grupo de seqüências já alinhadas, os *gaps* inseridos nas seqüências já alinhadas são preservados. Nesta mesma situação, o alinhamento é feito entre a seqüência ainda não alinhada e a seqüência do grupo alinhado que tenha a menor distância em relação a ela. As demais seqüências do grupo são apenas ajustadas para acompanhar o alinhamento.

Para um número ‘n’ de seqüências no conjunto de entrada, são necessários n-1 alinhamentos nessa fase. Por conseqüência, o algoritmo de programação dinâmica é invocado n-1 vezes.

Há algumas questões que devem ser consideradas sobre este método de alinhamento múltiplo:

5.2.3.1. O alinhamento resultante possui sentido biológico?

O método de alinhamento múltiplo progressivo sempre retorna uma resposta, independentemente se ela for certa ou errada. Se forem introduzidas duas seqüências com nenhuma relação biológica entre elas, o resultado também não terá sentido biológico nenhum. Seqüências muito diferentes ou muito parecidas também podem retornar um resultado sem sentido biológico se for encontrado um mínimo local.

5.2.3.2. Mínimos locais

O alinhamento múltiplo progressivo não investiga todas as possibilidades de alinhamentos exaustivamente. Como conseqüência disto, um alinhamento obtido pode ser um mínimo local bem longe do mínimo global. A dificuldade de se conseguir um alinhamento perfeito aumenta imensamente com o aumento das seqüências de entrada.

Mesmo considerando este problema, as respostas deste método são em geral boas e próximas da solução final. É muito comum, porém, que após o alinhamento múltiplo seja necessário fazer correções ou ajustes manuais para melhorar o alinhamento.

5.3 Alinhamento Múltiplo Progressivo Processado em Paralelo

O fato do problema de alinhamento múltiplo ser NP-Completo torna o apelo ao uso de supercomputação muito grande. Mesmo as heurísticas que tornam possível a realização dos alinhamentos ainda podem consumir um tempo considerável. Entretanto, utilizando técnicas de programação paralela, é possível distribuir o processamento entre vários processadores para diminuir o tempo total de processamento.

Para paralelizar o método de alinhamento múltiplo progressivo, primeiramente foram definidas as partes críticas em tempo de processamento.

A fase de alinhamento par a par é de longe a parte de maior custo de processamento. Contudo, a sua paralelização é relativamente simples. A estratégia utilizada foi distribuir os diversos alinhamentos par a par entre vários processadores. Esta estratégia se encaixa na classe de programas paralelos SPMD (Single Program / Multiple Data).

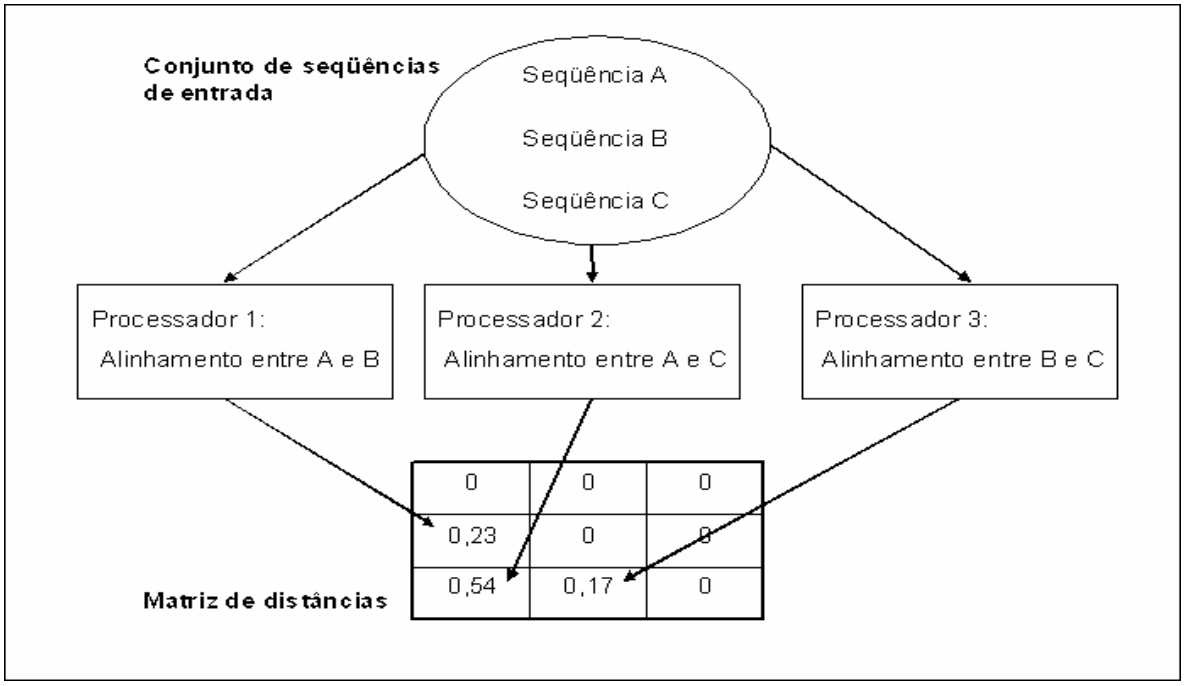

Figura 5 - Esquema de alinhamento par a par processado em paralelo.

Desconsiderando o tempo de comunicação entre os processos, o ganho de desempenho nesta fase deve ser praticamente proporcional ao aumento do poder de processamento. Como só existe comunicação entre o processo mestre e seus escravos uma única vez, o *overhead* de comunicação entre os processos torna-se insignificante na prática.

As fases seguintes do método de alinhamento múltiplo progressivo poderiam ser paralelizadas, mas o ganho de desempenho não seria muito vantajoso. Na construção da árvore guia, a busca dos dois ramos mais próximos na matriz de distâncias poderia ser facilmente paralelizada. Mas como a construção da árvore guia é um processo muito rápido em relação ao tempo total de processamento, o ganho de desempenho seria insignificante. Dividir a matriz de distâncias para processamento paralelo a cada iteração do algoritmo neighbor-joining poderia significar um overhead de comunicação muito grande.

A fase de alinhamento progressivo também poderia ser paralelizada, mas o ganho de desempenho seria muito relativo à estrutura da árvore guia criada. Como exemplo, o alinhamento progressivo guiado pela árvore da figura [4] poderia ser conduzido de forma que o alinhamento entre as seqüências humano e macaco e o alinhamento entre as seqüências camundongo e rato fossem processados paralelamente. Contudo, os demais alinhamentos teriam que ser conduzidos de forma seqüencial, já que há dependência de dados.

O programa que implementa o método de alinhamento múltiplo progressivo com processamento paralelo foi desenvolvido utilizando a linguagem de programação C++ em conjunto com MPI (Message Passing Interface).

MPI é uma interface para troca de mensagens entre processos. MPI dá suporte para comunicação entre processos trabalhando com o conceito de instruções

simples de *send* e *receive.* Há derivações destas instruções um pouco mais complexas para realização de tarefas coletivas entre os processos.

A MPI é utilizada na implementação do alinhador múltiplo no momento em que cada processo envia as distâncias dos alinhamentos par a par que cada um processou para o processo mestre. Como a fase de alinhamento par a par é a única processada em paralelo, os demais processos são finalizados enquanto o processo mestre processa as fases seguintes do alinhamento progressivo.

6. Resultados

Nesta seção será apresentada uma análise dos resultados obtidos com a implementação com relação à qualidade dos alinhamentos e ao desempenho obtido pela paralelização da ferramenta.

6.1 Análise de Qualidade

Para avaliar a qualidade dos alinhamentos, utilizou-se como referência o BAliBASE[6]. O BAliBASE é uma base de dados de alinhamentos múltiplos de seqüências de proteínas criada especificamente para a realização de *benchmarks* de qualidade. Todos os seus alinhamentos foram construídos manualmente a partir da comparação da estrutura de proteínas. Os alinhamentos do BAliBASE são divididos em grupos de características semelhantes que representam problemas comuns encontrados no alinhamento de seqüências de proteínas.

Para realizar o *benchmark* de qualidade, há um programa chamado bali_score. O bali_score compara um alinhamento de seqüências de proteínas da base de dados BAliBASE com um outro alinhamento que se deseja avaliar das mesmas seqüências.

O bali_score atribui um *score* para cada coluna do alinhamento sendo avaliado, e resume tudo em um valor percentual. Teoricamente um alinhamento que obtiver um *score* de 100% poderia ser considerado muito bom, ao passo que um *score* de 0% indicaria um alinhamento de péssima qualidade.

A tabela a seguir mostra os 5 grupos de seqüências do BAliBASE escolhidos para realizar o benchmark da implementação:

Conjunto:	1aab - ref 1	1csy – ref 1	1tvxA – ref 2	1dynA – ref 4	1thm2 – ref 5
Número de seqüências:	4	5	19	6	7
Tamanho do alinhamento:	82	110	78	893	252
Tamanho da maior seqüência:	79	104	69	848	229
Tamanho da menor seqüência:	67	100	51	89	172
Identidade média:	30%	30%	34%	18%	38%
Identidade máxima:	35%	38%	72%	27%	61%
Identidade mínima:	24%	27%	6%	8%	26%

Tabela 5 - Grupos de seqüências do BAliBASE utilizados no benchmarck da implementação.

A seguir são apresentados os resultados do *benchmark* para os 5 grupos de seqüências testados. Para efeito de contraste, há também os resultados obtidos com alguns dos programas de alinhamentos múltiplos mais comuns.

	992Align	**ClustalX**	**Saga**	**Dialign**	**PileUp8**	**HMMT**
1aab	0,836	1,000	0,823	1,000	1,000	0,214
1csy	0,822	1,000	0,969	0,980	0,935	0,779
1tvxA	0,688	0,552	0,448	0,000	0,345	X
1dynA	0,230	0,000	0,000	0,600	0,000	X
1thm2	0,776	0,774	0,774	1,000	0,645	X

Tabela 6 - Resultados do benchmark para os 5 grupos de seqüência testados.

Comparando os resultados, verificou-se que o 992Align teve um desempenho mais regular em relação aos outros programas. As demais ferramentas oscilaram mais fortemente entre bons e péssimos resultados. Essa regularidade pode ser atribuída a maior generalização das heurísticas utilizadas no 992Align. Os demais programas parecem ter heurísticas especializadas em alinhamentos múltiplos com

características mais específicas. Conclui-se que o 992align parece ser uma boa opção quando se desconhecem as características do alinhamento múltiplo que se deseja realizar.

6.2 Análise de Desempenho

A plataforma de hardware utilizada para a realização do *benchmark* de desempenho foi um cluster de computadores com as seguintes características:

- 1 servidor bi-processado, com processadores AMD Athlon 1900 MHz e 1 GB de memória RAM;
- 8 nodos bi-processados com as mesmas características do servidor;
- Comunicação entre servidor e nodos através de um switch Myrinet.

Para avaliar o ganho de desempenho proporcionado pela paralelização do alinhador múltiplo, foram realizados testes com dois conjuntos de seqüências. O primeiro conjunto é formado por seqüências da proteína CFTR (Cystic Fibrosis Transmembrane Conductance Regulator) de diferentes organismos. Já o segundo conjunto é um grupo de seqüências de DNA.

O conjunto CFTR contém 34 seqüências variando entre 103 e 1581 aminoácidos. Cada seqüência possui em média 1121 aminoácidos. O conjunto db10 contém 10 seqüências variando entre 4979 e 5488 nucleotídeos, e cada seqüência possui em média 5180 nucleotídeos.

Para visualizar o ganho de desempenho, os testes foram realizados com diferentes números de processadores, conforme ilustrado nos gráficos [1] e [2]:

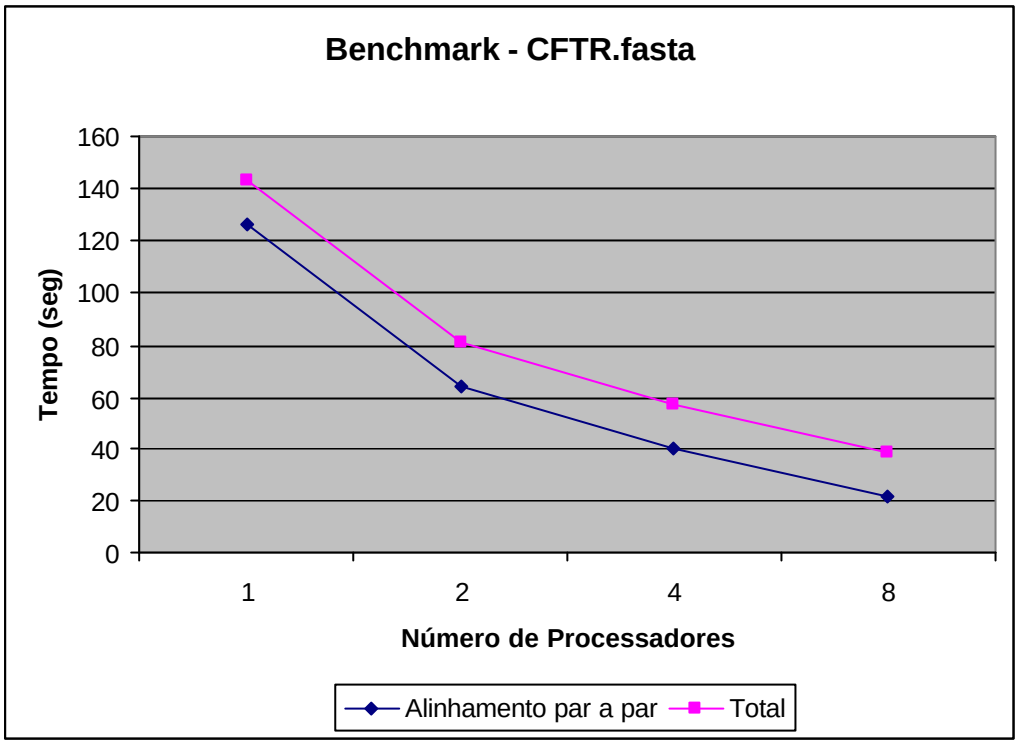

Gráfico 1 – Benchmark do conjunto de seqüências CFTR.fasta

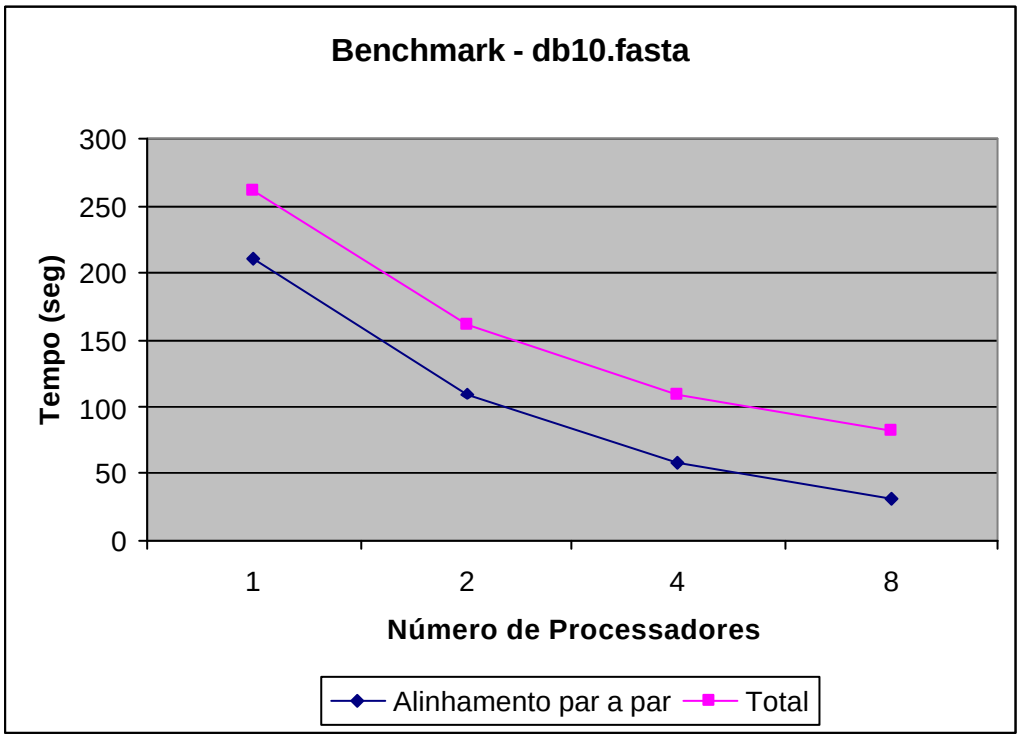

Gráfico 2 - Benchmark do conjunto de seqüências db10.fasta

Como se pode observar nos dois casos analisados, o ganho de tempo na fase de alinhamento par a par foi praticamente proporcional ao aumento do poder de processamento. Ou seja, conforme o número de processadores era dobrado, o tempo de processamento nesta fase caía pela metade.

A fase de construção da árvore guia tem um custo de tempo insignificante. A fase de alinhamento progressivo, por não ser paralelizada, passa a ter um custo significativo para um número ‘n’ de processadores, dependendo do número de seqüências sendo alinhadas. Quanto maior for o número de seqüências, maior é a vantagem de se ter um grande número de processadores.

O tempo total dos alinhamentos múltiplos acompanha a queda de tempo de processamento da fase de alinhamento par a par até certo ponto. A partir daí, o alinhamento progressivo passa a ser o fator determinante do custo total de processamento. Este ponto é influenciado pelo número de seqüências de entrada.

7. Conclusão

A implementação de alinhadores múltiplos de seqüências de DNA e proteínas esta longe de ser uma tarefa trivial. Como só é possível resolvê-los com ajuda de heurísticas, os resultados estão sujeitos a mínimos locais que podem estar longe de uma solução ótima. No entanto, o método de alinhamento múltiplo progressivo costuma apresentar resultados satisfatórios. Isto ficou caracterizado pelo bom desempenho apresentado pelo 992align no *benchmark* de qualidade.

Parece cada vez mais óbvia a convergência entre biologia computacional e computação paralela. Isto deve-se ao grande custo computacional necessário para processar a enorme quantidade de dados gerada por laboratórios de seqüenciamento.

Este projeto apresentou uma implementação do método de alinhamento múltiplo progressivo que tira proveito da supercomputação para diminuir o custo de processamento de alinhamentos com grandes quantidades de dados. Esta característica é essencial para tornar mais eficiente o estudo de filogenia e outros problemas relacionados à comparação de seqüências de DNA e proteínas.

8. Trabalhos Futuros

Na realização deste projeto ficaram evidentes algumas questões que poderiam ser abordadas futuramente de modo a melhorar o processo de alinhamento múltiplo:

- Formatação de dados / XML: atualmente há uma falta de padronização nos formatos de arquivos utilizados pelas ferramentas de bioinformática. Seria interessante se existisse uma formatação padrão de forma a acabar com esse problema.

- Alinhamento progressivo iterativo: muitos autores sugerem que sejam criados métodos iterativos que processem alinhamentos múltiplos com diversos parâmetros de forma automatizada, evitando que seja preciso realimentar as ferramentas manualmente diversas vezes.

- Programação dinâmica com espaço de memória linear: existe uma versão do algoritmo de programação dinâmica que ocupa espaço de memória linear, mas a sua implementação é bastante complexa. Devido ao curto tempo de projeto, optou-se por implementar apenas a versão de complexidade de memória quadrática.

Referências Bibliográficas

[1] FENG, Da-Fei; DOOLITTLE, Russell F. *Progressive sequence alignment as a prerequisite to correct phylogenetic trees*. J. Mol. Evol., 60:351-360, 1987.

[2] KEPPLER, Karl J.; STUDIER, James A. *A note on the neighbor-joining algorithm of saitou and nei*. Molecular Biology and Evolution 6, págs. 729-731. 1988.

[3] MEIDANIS, João; SETÚBAL, João Carlos. *Introduction to computational biology*. Brooks/Cole Publishing Company. 1997.

[4] MPI Forum, The. *MPI: A Message-Passing Interface Standard*. Disponível em http://www-unix.mcs.anl.gov/mpi . Acessado em 25/01/2004.

[5] NEI, Masatoshi; SAITO, Naruya. *The neighbor-joining method: a new method for reconstructing phylogenetic trees.* Molecular Biology and Evolution 4, págs. 406-425. 1987.

[6] THOMPSON, Julie D.; PLEWNIAK, Frédéric; POCH, Olivier. *BAliBASE: A benchmark alignments database for the evaluation of multiple sequence alignment programs.* Bioinformatics vol. 15, págs. 87-88, 1999.

[7] THOMPSON, Julie D.; PLEWNIAK, Frédéric; POCH, Olivier. *A comprehensive comparison of multiple sequence alignment programs.* Laboratoire de Biologie Structurale, Institut de Génétique et de Biologie Moléculaire et Cellulaire. 1999.

[8] TRELLES, Oswaldo. *On the parallelization of bioinformatics applications.* Briefings in Bioinformatics vol. 2, n° 2; págs. 181-219, 2001.

[9] WATERMAN, Michael S. *Introduction to computational biology: maps, sequences and genomes*. Chapman and Hall, London, 1995.

Anexos – A: Artigo

992ALIGN - UMA FERRAMENTA PARA ALINHAMENTO MÚLTIPLO DE SEQÜÊNCIAS DE DNA E PROTEÍNAS

Rodrigo Pilatti[1], Daniel Haeser Rech[2]

[1,2] Curso de Bacharelado em Ciências da Computação
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC), Brasil, 88040-900
Fone (0XX48)333-9999, Fax (0XX48)333-9999
pilatti@inf.ufsc.br, haeser@inf.ufsc.br

RESUMO

As ferramentas existentes para alinhamentos múltiplos apresentam em geral boas respostas, mas ainda estão muito longe de resultados perfeitos. Muitas vezes os biólogos precisam fazer intervenções manuais sobre os alinhamentos para obter uma solução que tenha sentido biológico, bem como realimentar as ferramentas com diferentes parâmetros. É necessário então aperfeiçoar os métodos existentes para alinhamentos múltiplos de forma que se diminua o esforço humano para resolvê-los.

Este artigo apresenta uma solução para alinhamento múltiplo progressivo de seqüências de DNA e proteínas. Esta utiliza técnicas de programação paralela para aproveitar o poder de processamento de máquinas multiprocessadas. A paralelização é justificada pelo grande custo computacional demandado por alinhamentos múltiplos de seqüências.

Palavras-chave: Alinhamentos Múltiplos, DNA, Proteínas, Paralelismo, Biologia Computacional

ABSTRACT

The existing tools for multiple alignments give good answers in general, but they are still far from giving perfect results. Often the biologists need to make manual interventions in the alignments to achieve a solution that makes biological sense, also needing to feed these tools with different parameters. So it's necessary to enhance the existing methods for multiple alignments in such a way that the human work to solve them is minimized.

This article presents a solution for the progressive multiple alignment method for DNA and protein sequences. It uses techniques of parallel programming to take advantage of the computing power of multiprocessed computers. The parallelization is justified by the huge computational cost demanded by multiple sequence alignments.

Keywords: Multiple Alignment, DNA, Proteins, Paralelism, Computational Biology.

Introdução

Nos últimos anos os estudos relacionados a genômica ganharam grande impulsão com o número crescente de genomas seqüenciados. A partir dessas informações, biólogos podem fazer as mais diversas análises com relação à evolução das espécies de seres vivos, e também no estudo de função de proteínas.

É preciso então que ferramentas sejam desenvolvidas para auxiliar o estudo e análise de toda essa informação gerada. Dado o grande volume de dados que compõem os genomas, é fundamental também que estas ferramentas apresentem resultados de forma rápida e eficiente. Torna-se necessário então o emprego de técnicas de programação paralela que aproveitem o poder da supercomputação.

O presente trabalho apresenta uma implementação de um alinhador múltiplo de seqüências de DNA e proteínas. Alinhamentos múltiplos são imprescindíveis para a comparação de genes e proteínas de espécies diferentes. Por este problema em geral demandar um grande tempo de processamento, a implementação tira proveito da supercomputação ao permitir que o programa seja executado em paralelo por vários processadores.

Alinhamentos Múltiplos

Um alinhamento de seqüências de DNA ou proteínas de diferentes espécies de seres vivos é uma hipótese de homologia entre as bases (ou nucleotídeos) que constituem os genes (ou proteínas) sendo comparados naqueles seres vivos. Alinhamentos podem ser tratados como modelos que podem ser usados para testar hipóteses evolucionárias e, portanto, são importantes para estudos de filogenia.

Encontrar um bom alinhamento entre as seqüências de várias espécies é geralmente uma tarefa difícil e envolve vários problemas. A similaridade pode ser alta em certos trechos das seqüências e baixa em outras partes. As seqüências podem ser de tamanhos variados ou muito diferentes entre si.

A maioria das técnicas de alinhamento múltiplo deriva do algoritmo de programação dinâmica.

Algoritmo de Programação Dinâmica

Para se encontrar um alinhamento entre duas seqüências de DNA, a técnica mais utilizada é baseada no algoritmo de programação dinâmica. Esta técnica busca o melhor alinhamento entre duas *strings* de caracteres. Ela se baseia na construção de uma matriz de comparação de prefixos das duas seqüências a serem alinhadas. Ao tentar alinhar sucessivamente os prefixos, o algoritmo atribui um *score* (valor de pontuação) para cada um dos prefixos. O *score* é computado de forma a penalizar as diferenças entre os prefixos e privilegiar as similaridades.

A complexidade de tempo do algoritmo de programação dinâmica puro para comparação de duas seqüências é n^2 com relação ao tamanho das seqüências. É possível generalizar o algoritmo para que ele resolva o alinhamento para um número ‘n’ fixo de seqüências. Contudo, a complexidade exponencial de memória e processamento torna a aplicação do algoritmo impraticável neste caso.

O problema de alinhar um número ‘n’ qualquer de seqüências é um NP-Completo se for utilizado o princípio do algoritmo de programação dinâmica. Os NP-Completos são problemas que não possuem solução conhecida que apresente uma resposta

em um tempo polinomial ou menor com relação ao tamanho da entrada de dados. Na prática isto quer dizer que o tempo de processamento seria extremamente alto e impraticável.

Para resolver esse problema de complexidade, algumas heurísticas foram desenvolvidas para diminuir a complexidade de tempo de alinhamentos com mais de duas seqüências. A abordagem freqüentemente mais utilizada é o alinhamento múltiplo progressivo.

Alinhamento Múltiplo Progressivo

O alinhamento múltiplo progressivo [1] é uma abordagem para o alinhamento de várias seqüências que tenta inferir uma evolução entre as espécies sendo comparadas. Podemos dividi-lo em 3 fases principais:

1) Alinhamento par a par: Nesta fase é construída uma matriz de distâncias dos alinhamentos de todos os possíveis pares do conjunto de seqüências de entrada. A distância do alinhamento de cada par é obtida utilizando-se o algoritmo de programação dinâmica. Para um conjunto de 'n' seqüências de entradas, o algoritmo de programação dinâmica será invocado n(n-1)/2 vezes. Isto torna a fase de comparação par-a-par a mais crítica em relação ao tempo de processamento em alinhamentos múltiplos progressivos.

2) Construção da árvore guia: obtida a matriz de distâncias, cria-se uma árvore guia que mais tarde conduzirá o alinhamento múltiplo propriamente dito. Existem diversos métodos para a construção desta árvore. O algoritmo neighbor-joining [5] (vide figura [1]) é geralmente considerado um dos métodos que apresentam resultados biológicos mais coerentes. Este algoritmo junta sucessivamente em uma árvore as seqüências mais próximas em ramos mais próximos, de forma iterativa, até que a árvore guia contenha todas as seqüências.

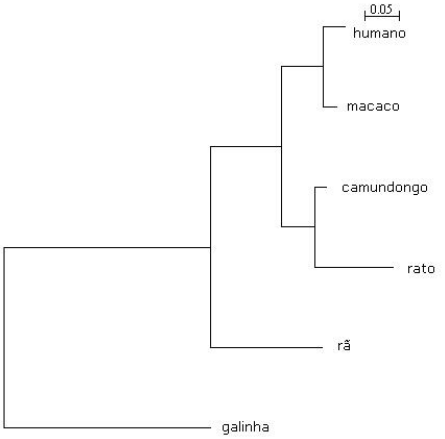

Figura 1 - Exemplo de árvore criada pelo algoritmo neighbor-joining.

3) Alinhamento Progressivo: O alinhamento progressivo produz o alinhamento múltiplo propriamente dito a partir da árvore guia criada. O alinhamento é feito progressivamente, de forma que as seqüências em ramos mais próximos sejam alinhadas primeiramente. Para alinhar uma seqüência com um grupo de seqüências já alinhadas, os *gaps* inseridos nas seqüências já alinhadas são preservados. Nesta mesma situação, o alinhamento é feito entre a seqüência ainda não alinhada e a seqüência do grupo alinhado que tenha a menor distância em relação a ela. As demais seqüências do grupo são apenas ajustadas para acompanhar o alinhamento.

Para um número 'n' de seqüências no conjunto de entrada, são necessários n-1 alinhamentos nessa fase. Por conseqüência, o algoritmo de programação dinâmica é invocado n-1 vezes.

Há algumas questões que devem ser consideradas sobre este método de alinhamento múltiplo:

- O alinhamento resultante possui sentido biológico? O método de alinhamento múltiplo progressivo sempre retorna uma resposta,

independentemente se ela for certa ou errada. Se forem introduzidas duas seqüências com nenhuma relação biológica entre elas, o resultado também não terá sentido biológico nenhum. Seqüências muito diferentes ou muito parecidas também podem retornar um resultado sem sentido biológico se for encontrado um mínimo local.

- **Mínimos locais:** O alinhamento múltiplo progressivo não investiga todas as possibilidades de alinhamentos exaustivamente. Como conseqüência disto, um alinhamento obtido pode ser um mínimo local bem longe do mínimo global. A dificuldade de se conseguir um alinhamento perfeito aumenta imensamente com o aumento das seqüências de entrada. Mesmo considerando este problema, as respostas deste método são em geral boas e próximas da solução final. É muito comum, porém, que após o alinhamento múltiplo seja necessário fazer correções ou ajustes manuais para melhorar o alinhamento.

Alinhamento Múltiplo Progressivo Processado em Paralelo

O fato do problema de alinhamento múltiplo ser NP-Completo torna o apelo ao uso de supercomputação muito grande. Mesmo as heurísticas que tornam possível a realização dos alinhamentos ainda podem consumir um tempo considerável. Entretanto, utilizando técnicas de programação paralela, é possível distribuir o processamento entre vários processadores para diminuir o tempo total de processamento.

Para paralelizar o método de alinhamento múltiplo progressivo, primeiramente foram definidas as partes críticas em tempo de processamento.

A fase de alinhamento par a par é de longe a parte de maior custo de processamento. Contudo, a sua paralelização é relativamente simples. A estratégia utilizada foi distribuir os diversos alinhamentos par a par entre vários processadores. Esta estratégia se encaixa na classe de programas paralelos SPMD (Single Program / Multiple Data).

Desconsiderando o tempo de comunicação entre os processos, o ganho de desempenho nesta fase deve ser praticamente proporcional ao aumento do poder de processamento. Como só existe comunicação entre o processo mestre e seus escravos uma única vez, o *overhead* de comunicação entre os processos torna-se insignificante na prática.

As fases seguintes do método de alinhamento múltiplo progressivo poderiam ser paralelizadas, mas o ganho de desempenho não seria muito vantajoso. Na construção da árvore guia, a busca dos dois ramos mais próximos na matriz de distâncias poderia ser facilmente paralelizada. Mas como a construção da árvore guia é um processo muito rápido em relação ao tempo total de processamento, o ganho de desempenho seria insignificante. Dividir a matriz de distâncias para processamento paralelo a cada iteração do algoritmo neighbor-joining poderia significar um overhead de comunicação muito grande.

A fase de alinhamento progressivo também poderia ser paralelizada, mas o ganho de desempenho seria muito relativo à estrutura da árvore guia criada. Como exemplo, o alinhamento progressivo guiado pela árvore da figura [1] poderia ser conduzido de forma que o alinhamento entre as seqüências humano e macaco e o alinhamento entre as seqüências camundongo e rato fossem processados paralelamente. Contudo, os demais alinhamentos teriam que ser conduzidos de forma seqüencial, já que há dependência de dados.

O programa que implementa o método de alinhamento múltiplo progressivo com processamento paralelo foi desenvolvido utilizando a linguagem de programação C++ em conjunto com MPI (Message Passing Interface).

MPI é uma interface para troca de mensagens entre processos. MPI dá suporte para comunicação entre processos trabalhando com o conceito de instruções simples de *send* e *receive.* Há derivações destas instruções um pouco mais complexas para realização de tarefas coletivas entre os processos.

Análise de Qualidade

Para avaliar a qualidade dos alinhamentos, utilizou-se como referência o BAliBASE[6]. O BAliBASE é uma base de dados de alinhamentos múltiplos de seqüências de proteínas criada especificamente para a realização de *benchmarks* de qualidade. Todos os seus alinhamentos foram construídos manualmente a partir da comparação da estrutura de proteínas. Os alinhamentos do BAliBASE são divididos em grupos de características semelhantes que representam problemas comuns encontrados no alinhamento de seqüências de proteínas.

Para realizar o *benchmark* de qualidade, há um programa chamado bali_score. O bali_score compara um alinhamento de seqüências de proteínas da base de dados BAliBASE com um outro alinhamento que se deseja avaliar das mesmas seqüências.

O bali_score atribui um *score* para cada coluna do alinhamento sendo avaliado, e resume tudo em um valor percentual. Teoricamente um alinhamento que obtiver um *score* de 100% poderia ser considerado muito bom, ao passo que um *score* de 0% indicaria um alinhamento de péssima qualidade. A tabela a seguir mostra os 5 grupos de seqüências do BAliBASE escolhidos para realizar o benchmark da implementação:

Conjunto:	1aab - ref 1	1csy – ref 1	1tvxA – ref 2	1dynA – ref 4	1thm2 – ref 5
Número de seqüências:	4	5	19	6	7
Tamanho do alinhamento:	82	110	78	893	252
Tamanho da maior seqüência:	79	104	69	848	229
Tamanho da menor seqüência:	67	100	51	89	172
Identidade média:	30%	30%	34%	18%	38%
Identidade máxima:	35%	38%	72%	27%	61%
Identidade mínima:	24%	27%	6%	8%	26%

Tabela 1 - Grupos de seqüências do BAliBASE utilizados no benchmarck da implementação.

A seguir são apresentados os resultados do *benchmark* para os 5 grupos de seqüências testados. Para efeito de contraste, há também os resultados obtidos com alguns dos programas de alinhamentos múltiplos mais comuns.

	992Align	**ClustalX**	**Saga**	**Dialign**	**PileUp8**	**HMMT**
1aab	0,836	1,000	0,823	1,000	1,000	0,214
1csy	0,822	1,000	0,969	0,980	0,935	0,779
1tvxA	0,688	0,552	0,448	0,000	0,345	X
1dynA	0,230	0,000	0,000	0,600	0,000	X
1thm2	0,776	0,774	0,774	1,000	0,645	X

Tabela 2 - Resultados do benchmark para os 5 grupos de seqüência testados.

Comparando os resultados, verificou-se que o 992Align teve um desempenho mais regular em relação aos outros programas. As demais

ferramentas oscilaram mais fortemente entre bons e péssimos resultados. Essa regularidade pode ser atribuída a maior generalização das heurísticas utilizadas no 992Align. Os demais programas parecem ter heurísticas especializadas em alinhamentos múltiplos com características mais específicas. Conclui-se que o 992align parece ser uma boa opção quando se desconhecem as características do alinhamento múltiplo que se deseja realizar.

Análise de Desempenho

A plataforma de hardware utilizada para a realização do *benchmark* de desempenho foi um cluster de computadores com as seguintes características:

- 1 servidor bi-processado, com processadores AMD Athlon 1900 MHz e 1 GB de memória RAM;
- 8 nodos bi-processados com as mesmas características do servidor;
- Comunicação entre servidor e nodos através de um switch Myrinet.

Para avaliar o ganho de desempenho proporcionado pela paralelização do alinhador múltiplo, foram realizados testes com dois conjuntos de seqüências. O primeiro conjunto é formado por seqüências da proteína CFTR (Cystic Fibrosis Transmembrane Conductance Regulator) de diferentes organismos. Já o segundo conjunto é um grupo de seqüências de DNA.

O conjunto CFTR contém 34 seqüências variando entre 103 e 1581 aminoácidos. Cada seqüência possui em média 1121 aminoácidos. O conjunto db10 contém 10 seqüências variando entre 4979 e 5488 nucleotídeos, e cada seqüência possui em média 5180 nucleotídeos.

Para visualizar o ganho de desempenho, os testes foram realizados com diferentes números de processadores, conforme ilustrado nos gráficos [1] e [2]:

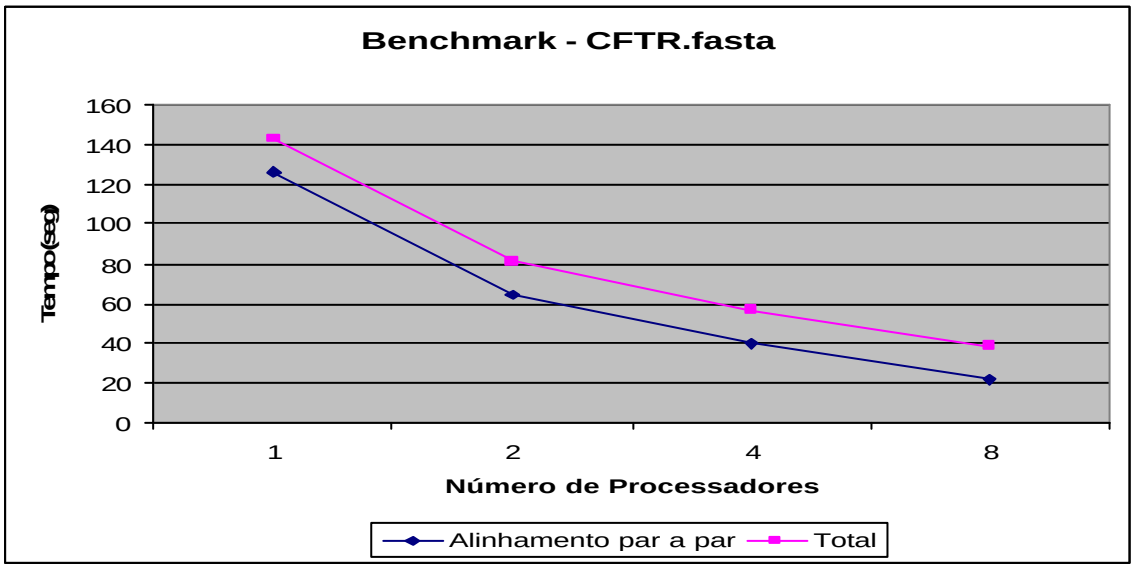

Gráfico 1 – Benchmark – CFTR.fasta

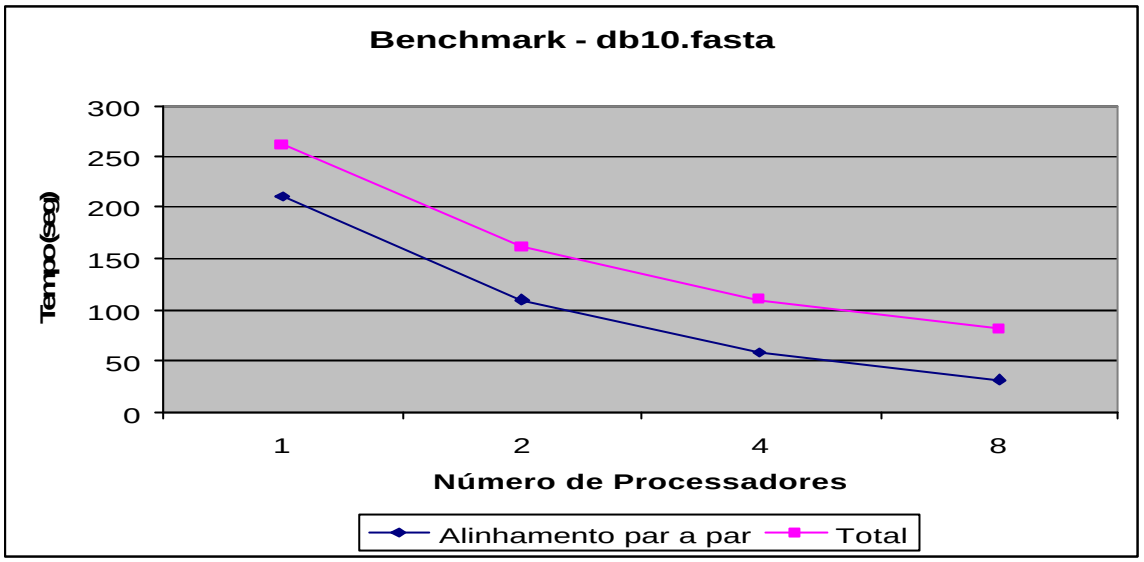

Gráfico 2 – Benchmark – db10.fasta

Como se pode observar nos dois casos analisados, o ganho de tempo na fase de alinhamento par a par foi praticamente proporcional ao aumento do poder de processamento. Ou seja, conforme o número de processadores era dobrado, o tempo de processamento nesta fase caía pela metade.

A fase de construção da árvore guia tem um custo de tempo insignificante. A fase de alinhamento progressivo, por não ser paralelizada, passa a ter um custo significativo para um número 'n' de processadores, dependendo do número de seqüências sendo alinhadas. Quanto maior for o número de seqüências, maior é a vantagem de se ter um grande número de processadores.

O tempo total dos alinhamentos múltiplos acompanha a queda de tempo de processamento da fase de alinhamento par a par até certo ponto. A partir daí, o alinhamento progressivo passa a ser o fator determinante do custo total de processamento. Este ponto é influenciado pelo número de seqüências de entrada.

Conclusão

A implementação de alinhadores múltiplos de seqüências de DNA e proteínas esta longe de ser uma tarefa trivial. Como só é possível resolvê-los com ajuda de heurísticas, os resultados estão sujeitos a mínimos locais que podem estar longe de uma solução ótima. No entanto, o método de alinhamento múltiplo progressivo costuma apresentar resultados satisfatórios. Isto ficou caracterizado pelo bom desempenho apresentado pelo 992align no *benchmark* de qualidade.

Parece cada vez mais óbvia a convergência entre biologia computacional e computação paralela. Isto deve-se ao grande custo computacional necessário para processar a enorme quantidade de dados gerada por laboratórios de seqüenciamento.

Este projeto apresentou uma implementação do método de alinhamento múltiplo progressivo que tira proveito da supercomputação para diminuir o custo de processamento de alinhamentos com grandes quantidades de dados. Esta característica é essencial para tornar mais eficiente o estudo de

filogenia e outros problemas relacionados à comparação de seqüências de DNA e proteínas.

Referências Bibliográficas

[1] FENG, Da-Fei; DOOLITTLE, Russell F. *Progressive sequence alignment as a prerequisite to correct phylogenetic trees*. J. Mol. Evol., 60:351-360, 1987.

[2] KEPPLER, Karl J.; STUDIER, James A. *A note on the neighbor-joining algorithm of saitou and nei*. Molecular Biology and Evolution 6, págs. 729-731. 1988.

[3] MEIDANIS, João; SETÚBAL, João Carlos. *Introduction to computational biology*. Brooks/Cole Publishing Company. 1997.

[4] MPI Forum, The. *MPI: A Message-Passing Interface Standard*. Disponível em http://www-unix.mcs.anl.gov/mpi . Acessado em 25/01/2004.

[5] NEI, Masatoshi; SAITO, Naruya. *The neighbor-joining method: a new method for reconstructing phylogenetic trees.* Molecular Biology and Evolution 4, págs. 406-425. 1987.

[6] THOMPSON, Julie D.; PLEWNIAK, Frédéric; POCH, Olivier. *BAliBASE: A benchmark alignments database for the evaluation of multiple sequence alignment programs*. Bioinformatics vol. 15, págs. 87-88, 1999.

[7] THOMPSON, Julie D.; PLEWNIAK, Frédéric; POCH, Olivier. *A comprehensive comparison of multiple sequence alignment programs*. Laboratoire de Biologie Structurale, Institut de Génétique et de Biologie Moléculaire et Cellulaire. 1999.

[8] TRELLES, Oswaldo. *On the parallelization of bioinformatics applications.* Briefings in Bioinformatics vol. 2, n° 2; págs. 181-219, 2001.

[9] WATERMAN, Michael S. *Introduction to computational biology: maps, sequences and genomes*. Chapman and Hall, London, 1995.

Anexos – B: Código Fonte

```
//
// 992Align.cpp
//

#include <ctime>
#include <stdlib.h>
#include "mpi.h"
#include "util.h"

double *AA[MAX_SEQ_SIZE];
double *BB[MAX_SEQ_SIZE];
double *CC[MAX_SEQ_SIZE];


SequencesSet *inputSequences;

int matrixSize;
double distancesMatrix[128][128];
double distancesMatrix2[128][128];

double     gapPenalty = 2;
double openGapPenalty = 10;


#include "fileIO.h"
#include "scoringfunction.h"
#include "pairalign.h"
#include "njtree.h"
#include "progressivealign.h"

int main(int argc, char *argv[]) {
       std::clock_t start;
       std::clock_t pair;
    std::clock_t tree;
       std::clock_t prog;

    int i, j, k, l;

    for(i = 0; i < MAX_SEQ_SIZE; i++) {
           AA[i] = new double[MAX_SEQ_SIZE];
           BB[i] = new double[MAX_SEQ_SIZE];
           CC[i] = new double[MAX_SEQ_SIZE];
    }

    int myrank, np;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &np);

       if(myrank == 0) {
              if(argc == 1) {
                     printf("Digite: 992align arquivo_sequencias\n");
                     return 0;
              }
       }

       inputSequences = read_file(argv[1]);

       if(inputSequences->nSequences == 0) {
              if(myrank == 0)
```

```
                printf("Erro abrindo arquivo!\n");
        return 0;
   }

FILE *dndFile;
FILE *alnFile;

   if(myrank == 0) {
        char *dndFilename = new char[128];
        char *alnFilename = new char[128];


        strcpy(dndFilename, "files/");
        strcat(dndFilename, argv[1]);
        strcat(dndFilename, ".dnd");
        dndFile = fopen(dndFilename, "w");

        strcpy(alnFilename, "files/");
        strcat(alnFilename, argv[1]);
        strcat(alnFilename, ".aln");
        alnFile = fopen(alnFilename, "w");
   }


   matrixSize = inputSequences->nSequences;
   for(j = 0; j < matrixSize; j++) {
        for(i = j; i < matrixSize; i++) {
                distancesMatrix2[i][j] = 0;
        }
   }

   setScoringFunction(DEFAULT);


   start = std::clock();


   //
   //     Pairwise Alignment
   //
   SequencesSet *globalAlignment;
   double score;

   // Computation of the workload for each processor
   int numberOfPairAlignments = (matrixSize * (matrixSize - 1)) / 2;
   int *workload = new int[np];
   int *workstart = new int[np];
   int na = numberOfPairAlignments;
   int npTmp = np;
   int wl;
   int wlTmp = 0;

   for(i = 0; i < np; i++) {
        wl = na / npTmp;
        workload[i] = wl;
        na = na - wl;
        workstart[i] = wlTmp;
           wlTmp = wlTmp + wl;
        npTmp--;
   }

   double *myDistances = new double[workload[myrank]];
   double *allDistances;
   if(myrank == 0)
        allDistances = new double[numberOfPairAlignments];

   k = 0;
   l = 0;
```

```
	// Starts Pairwise Alignments
	for(j = 1; j < matrixSize; j++) {
		for(i = 0; i < j; i++) {
			if((k >= workstart[myrank]) && (k < (workstart[myrank] +
workload[myrank]))) {
				printf("P%d comparing sequences %d e %d\n", myrank,
j + 1, i + 1);
				bestPairAlignScore(inputSequences->sequences[i],
inputSequences->sequences[j]);
				globalAlignment = bestPairAlignment(inputSequences-
>sequences[i], inputSequences->sequences[j]);
				myDistances[l] = distance(globalAlignment-
>sequences[0], globalAlignment->sequences[1]);
				l++;
			}
			k++;
		}
	}

	// Processor 0 receives and organize distances matrix
	MPI_Gatherv(myDistances, workload[myrank], MPI_DOUBLE, allDistances,
workload, workstart, MPI_DOUBLE, 0, MPI_COMM_WORLD);

	if(myrank == 0) {
		k = 0;
		for(j = 1; j < matrixSize; j++) {
			for(i = 0; i < j; i++) {
				distancesMatrix[i][j] = *(allDistances + k);
				distancesMatrix[j][i] = *(allDistances + k);
				distancesMatrix2[i][j] = *(allDistances + k);
				k++;
			}
		}


		pair = std::clock();


		//
		//	Create NJTree
		//
		printf("\nCreating guide tree...");
		njtree *theNJtree = createNJTree();
		printf(" Ok\n");


		tree = std::clock();


		//
		//	Progressive alignment
		//
		printf("\nMaking progressive alignment...");
		SequencesSet *multipleAlignment = progressiveAlign(theNJtree);
		printf(" Ok\n");

		prog = std::clock();

		printDND(theNJtree, dndFile);
		int alignmentLength = strlen(multipleAlignment->sequences[0]);
		for(i = 0; i < multipleAlignment->nSequences; i++) {
			k = strlen(multipleAlignment->names[i]);
			while(k < 16) {
				multipleAlignment->names[i][k] = ' ';
				k++;
			}
			multipleAlignment->names[i][16] = '\0';
```

```
            }
            int *check = new int[multipleAlignment->nSequences];
            for(i = 0; i < multipleAlignment->nSequences; i++)
                  check[i] = 0;
            int m;
            i = 0;
            fprintf(alnFile, "//\n\n");
            while(i < alignmentLength) {
                  j = 0;
                  m = i;
                  while(j < multipleAlignment->nSequences) {
                        i = m;
                        fprintf(alnFile, "%s", multipleAlignment->names[j]);
                        k = 0;
                        while(k < 5) {
                              l = 0;
                              while((i < alignmentLength) && (l < 10)) {
                                    fprintf(alnFile, "%c",
multipleAlignment->sequences[j][i]);
                                    check[j] = check[j] +
multipleAlignment->sequences[j][i];
                                    l++;
                                    i++;
                              }
                              fprintf(alnFile, " ");
                              k++;
                        }
                        fprintf(alnFile, "\n");
                        j++;
                  }
                  fprintf(alnFile, "\n\n");
            }
            fclose(dndFile);
            fclose(alnFile);
            printf("\n Pair: %f seconds\n", double(pair - start) /
CLOCKS_PER_SEC);
            printf(" Tree: %f seconds\n", double(tree - pair) /
CLOCKS_PER_SEC);
            printf(" Prog: %f seconds\n", double(prog - tree) /
CLOCKS_PER_SEC);
            printf("Total: %f seconds\n", double(prog - start) /
CLOCKS_PER_SEC);
      }
      MPI_Finalize();
      return 1;
}
```

```
//
// pairalign.h
//

#ifndef _pairalign_
#define _pairalign_

	#include "util.h"
	#include "scoringfunction.h"

	double bestPairAlignScore(char *sequence1, char *sequence2);
	SequencesSet* bestPairAlignment(char *seq1, char *seq2);
	double distance(char *sequence1, char *sequence2);

#endif
```

```
//
// pairalign.cpp
//

#include "pairalign.h"


extern double *AA[MAX_SEQ_SIZE];
extern double *BB[MAX_SEQ_SIZE];
extern double *CC[MAX_SEQ_SIZE];


extern double gapPenalty;
extern double openGapPenalty;

double bestScore;

double bestPairAlignScore(char *sequence1, char *sequence2) {
	int sequence1Size = strlen(sequence1);
	int sequence2Size = strlen(sequence2);
	int i, j;

	// SEMI-GLOBAL
	for(i = 1; i <= sequence1Size; i++) {
		AA[i][0] = 0; //MOST_NEGATIVE_INT; // 0;
		BB[i][0] = MOST_NEGATIVE_INT;
		CC[i][0] = -(openGapPenalty + (gapPenalty * i));
	}
	for(j = 1; j <= sequence2Size; j++) {
		AA[0][j] = 0; //MOST_NEGATIVE_INT; // 0;
		BB[0][j] = -(openGapPenalty + (gapPenalty * j));
		CC[0][j] = MOST_NEGATIVE_INT;
	}
	AA[0][0] = 0;
	BB[0][0] = MOST_NEGATIVE_INT;
	CC[0][0] = MOST_NEGATIVE_INT;

	for(i = 1; i <= sequence1Size; i++) {
		for(j = 1; j <= sequence2Size; j++) {
			AA[i][j] = p(sequence1[i - 1], sequence2[j - 1]) +
max3(AA[i - 1][j - 1], BB[i - 1][j - 1], CC[i - 1][j - 1]);

			BB[i][j] = max3(-(openGapPenalty + gapPenalty) + AA[i][j -
1], -gapPenalty + BB[i][j - 1], -(openGapPenalty + gapPenalty) + CC[i][j -
1]);

			CC[i][j] = max3(-(openGapPenalty + gapPenalty) + AA[i -
1][j], -(openGapPenalty + gapPenalty) + BB[i - 1][j], -gapPenalty + CC[i -
1][j]);
		}
	}
```

```
	bestScore = max3(AA[i - 1][j - 1], BB[i - 1][j - 1], CC[i - 1][j - 1]);
	return bestScore;
}

SequencesSet* bestPairAlignment(char *sequence1, char *sequence2) {
	int i = strlen(sequence1);
	int j = strlen(sequence2);
	int ib = i;
	int jb = j;
	int currentMatrix, k;
	SequencesSet *alignment = new SequencesSet();

	alignment->nSequences = 2;
	alignment->sequences[0] = new char[MAX_SEQ_SIZE * 2];
	alignment->sequences[0][0] = '\0';
	alignment->sequences[1] = new char[MAX_SEQ_SIZE * 2];
	alignment->sequences[1][0] = '\0';

	currentMatrix = 1;
	for (k = 1; k <= i; k++) {
		if(AA[k][j] > AA[ib][jb]) {
			ib = k;
			jb = j;
		}
	}
	for (k = 1; k < j; k++) {
		if(AA[i][k] > AA[ib][jb]) {
			ib = i;
			jb = k;
		}
	}
	while(i > ib) {
		strlcat(sequence1[i - 1], alignment->sequences[0]);
		strlcat('-', alignment->sequences[1]);
		i--;
	}
	while(j > jb) {
		strlcat('-', alignment->sequences[0]);
		strlcat(sequence2[j - 1], alignment->sequences[1]);
		j--;
	}

	while ((i > 0) && (j > 0)) {
		int tmpMatrix = currentMatrix;
		switch (tmpMatrix) {
			case 1: {
				if(AA[i][j] == p(sequence1[i - 1], sequence2[j - 1])
+ AA[i - 1][j - 1]) {
					currentMatrix = 1;
				} else {
				if(AA[i][j] == p(sequence1[i - 1], sequence2[j - 1])
+ BB[i - 1][j - 1]) {
					currentMatrix = 2;
				} else {
				if(AA[i][j] == p(sequence1[i - 1], sequence2[j - 1])
+ CC[i - 1][j - 1]) {
					currentMatrix = 3;
				}
				}
				}
				strlcat(sequence1[i - 1], alignment->sequences[0]);
				strlcat(sequence2[j - 1], alignment->sequences[1]);
				i--;
				j--;
			} break;
			case 2: {
				if(BB[i][j] == (-(openGapPenalty + gapPenalty) +
AA[i][j - 1])) {
```

```
                              currentMatrix = 1;
                        } else {
                        if(BB[i][j] == (-gapPenalty + BB[i][j - 1])) {
                              currentMatrix = 2;
                        } else {
                        if(BB[i][j] == (-(openGapPenalty + gapPenalty) +
CC[i][j - 1])) {
                              currentMatrix = 3;
                        }
                        }
                        }
                        strlcat('-', alignment->sequences[0]);
                        strlcat(sequence2[j - 1], alignment->sequences[1]);
                        j--;
                  } break;
                  case 3: {
                        if(CC[i][j] == (-(openGapPenalty + gapPenalty) +
AA[i - 1][j])) {
                              currentMatrix = 1;
                        } else {
                        if(CC[i][j] == (-(openGapPenalty + gapPenalty) +
BB[i - 1][j])) {
                              currentMatrix = 2;
                        } else {
                        if(CC[i][j] == (-gapPenalty + CC[i - 1][j])) {
                              currentMatrix = 3;
                        }
                        }
                        }
                        strlcat(sequence1[i - 1], alignment->sequences[0]);
                        strlcat('-', alignment->sequences[1]);
                        i--;
                  }
            }
      }
      while(i > 0) {
            strlcat(sequence1[i - 1], alignment->sequences[0]);
            strlcat('-', alignment->sequences[1]);
            i--;
      }
      while(j > 0) {
            strlcat('-', alignment->sequences[0]);
            strlcat(sequence2[j - 1], alignment->sequences[1]);
            j--;
      }
      return alignment;
}

double distance(char *sequence1, char *sequence2) {
      int i;
      int prox = 0;

      int sizeOfAlignment = strlen(sequence1);
      int nonGappedAlignmentSize = sizeOfAlignment;
      for(i = 0; i < sizeOfAlignment; i++) {
            if((sequence1[i] == '-') || (sequence2[i] == '-'))
                  nonGappedAlignmentSize--;
            if(sequence1[i] == sequence2[i])
                  prox++;

      }
      return 1 - (double(prox) / double(nonGappedAlignmentSize));
}
```

```
//
// njtree.h
//


#ifndef _njtree_
#define _njtree_

	#include "util.h"

	typedef struct njnode {
		njnode *parent;
		njnode *left;
		njnode *right;
		double branchLength;
		int nSequence;
		char *sequenceName;
	} njnodeType;

	typedef struct {
		njnode *root;
	} njtree;

	void printDND(njtree *theNJtree, FILE *dndFile);
	void printDNDRecursive(njnode *theNode, FILE *dndFile);
	bool isLeaf(njnode *theNode);
	njtree* createNJTree();

#endif
```

```
//
// njtree.cpp
//

#include "njtree.h"

extern SequencesSet *inputSequences;
extern int matrixSize;
extern double distancesMatrix2[128][128];

bool isLeaf(njnode *theNode) {
	if((theNode->left == NULL) && (theNode->right == NULL))
		return true;
	return false;
}

void printDND(njtree *theNJtree, FILE *dndFile) {
	printDNDRecursive(theNJtree->root, dndFile);
	fprintf(dndFile, ";");
	fclose(dndFile);
}

void printDNDRecursive(njnode *theNode, FILE *dndFile) {
	if(isLeaf(theNode)) {
		fprintf(dndFile, "%s:%f\n", theNode->sequenceName, theNode-
>branchLength);
		return;
	}
	fprintf(dndFile, "(\n");
	printDNDRecursive(theNode->left, dndFile);
	fprintf(dndFile, ",\n");
	printDNDRecursive(theNode->right, dndFile);
	fprintf(dndFile, ")\n");
	fprintf(dndFile, ":%f\n", theNode->branchLength);
}

njtree* createNJTree() {
```

```
	double M[128][128];
	double divergences[128];
	njnode *theNodes[128];
	njnode *newnode;
	double distanceBetweenNodes;
	int x, y, i, j, k, l;

	for(i = 0; i < matrixSize; i++) {
		theNodes[i] = new njnode();
		theNodes[i]->parent = NULL;
		theNodes[i]->left = NULL;
		theNodes[i]->right = NULL;
		theNodes[i]->nSequence = i;
		theNodes[i]->sequenceName = inputSequences->names[i];
	}
	int N = matrixSize - 2;

	while(matrixSize > 1) {
		for(k = 0; k < matrixSize; k++) {
			divergences[k] = 0;
			for(i = 0; i < k; i++)
				divergences[k] = divergences[k] +
distancesMatrix2[i][k];
			for(j = k + 1; j < matrixSize; j++)
				divergences[k] = divergences[k] +
distancesMatrix2[k][j];
		}

		for(j = 1; j < matrixSize; j++) {
			for(i = 0; i < j; i++) {
				M[i][j] = distancesMatrix2[i][j] - ((divergences[i]
+ divergences[j]) / N);
			}
		}

		x = 0;
		y = 1;
		for(j = 2; j < matrixSize; j++) {
			for(i = 0; i < j; i++) {
				if(M[i][j] < M[x][y]) {
					x = i;
					y = j;
				}
			}
		}
		i = x;
		j = y;

		theNodes[i]->branchLength = (distancesMatrix2[i][j] / 2) +
((divergences[i] - divergences[j]) / (2 * N));
		theNodes[j]->branchLength = distancesMatrix2[i][j] - theNodes[i]-
>branchLength;

		newnode = new njnode();
		newnode->parent = NULL;
		newnode->left = theNodes[i];
		newnode->right = theNodes[j];
		newnode->left->parent = newnode;
		newnode->right->parent = newnode;
		distanceBetweenNodes = distancesMatrix2[i][j];
		if(i > j) {
			k = j;
			l = i;
		}
		else {
			k = i;
			l = j;
		}
```

```
            theNodes[k] = newnode;

            for(i = 0; i < k; i++)
                  distancesMatrix2[i][k] = (distancesMatrix2[i][k] +
distancesMatrix2[i][l] - distanceBetweenNodes) / 2;
            for(j = l; j < matrixSize; j++)
                  distancesMatrix2[k][j] = (distancesMatrix2[k][j] +
distancesMatrix2[l][j] - distanceBetweenNodes) / 2;
            for(j = l + 1; j < matrixSize; j++)
                  for(i = l; i < matrixSize; i++)
                        distancesMatrix2[i][j] = distancesMatrix2[i + 1][j];
            for(j = l; j < matrixSize; j++)
                  for(i = 0; i < matrixSize; i++)
                        distancesMatrix2[i][j] = distancesMatrix2[i][j + 1];

            for(j = l; j < (matrixSize - 1); j++)
                  theNodes[j] = theNodes[j + 1];

            matrixSize--;
      }
      njtree *theNJtree = new njtree();
      theNJtree->root = newnode;
      return theNJtree;
}
```

```
//
// progressivealign.h
//

#ifndef _progressivealign_
#define _progressivealign_

	#include "util.h"
	#include "scoringfunction.h"
	#include "pairalign.h"
	#include "njtree.h"

	SequencesSet* progressiveAlign(njtree *theNJtree);
	SequencesSet* progressiveAlignRecursive(njnode *theNode);
	double bestProfileAlignScore(SequencesSet *profile1, SequencesSet
*profile2, int m, int n);
	SequencesSet* bestProfileAlignment(SequencesSet *profile1, SequencesSet
*profile2, int m, int n);
	SequencesSet* alignProfiles(SequencesSet *profile1, SequencesSet
*profile2, int m, int n);

#endif
```

```
//
// progressivealign.cpp
//

#include <stdlib.h>
#include "progressivealign.h"


extern SequencesSet *inputSequences;
extern double distancesMatrix[128][128];

extern double gapPenalty;
extern double openGapPenalty;

int i, j, x, y;
double currentDistance;
double shortestDistance;

SequencesSet* progressiveAlign(njtree *theNJtree) {
	return progressiveAlignRecursive(theNJtree->root);
}

SequencesSet* progressiveAlignRecursive(njnode *theNode) {
	SequencesSet *profile1;
	SequencesSet *profile2;
	if(isLeaf(theNode)) {
		profile1 = new SequencesSet();
		profile1->nSequences = 1;
		profile1->sequences[0] = new char[MAX_SEQ_SIZE];
		strcpy(profile1->sequences[0], inputSequences->sequences[theNode-
>nSequence]);
		profile1->names[0] = inputSequences->names[theNode->nSequence];
		profile1->index[0] = theNode->nSequence;
		return profile1;
	} else {
		profile1 = progressiveAlignRecursive(theNode->left);
		profile2 = progressiveAlignRecursive(theNode->right);
		x = 0; y = 0;
		shortestDistance = distancesMatrix[profile1->index[0]][profile2-
>index[0]];
		for(j = 0; j < profile2->nSequences; j++) {
			for(i = 0; i < profile1->nSequences; i++) {
				currentDistance = distancesMatrix[profile1-
>index[i]][profile2->index[j]];
```

```
                        if(currentDistance < shortestDistance) {
                              shortestDistance = currentDistance;
                              x = i; y = j;
                        }
                  }
            }
            return alignProfiles(profile1, profile2, x, y);
      }
}

SequencesSet* alignProfiles(SequencesSet *profile1, SequencesSet *profile2,
int m, int n) {
      SequencesSet *alignment, *pairAlignment;
      int i, j, k, l;
      int prof1Size, prof2Size, pairalignSize;

      bestPairAlignScore(inputSequences->sequences[profile1->index[m]],
inputSequences->sequences[profile2->index[n]]);
      pairAlignment = bestPairAlignment(inputSequences->sequences[profile1-
>index[m]], inputSequences->sequences[profile2->index[n]]);

      prof1Size = strlen(profile1->sequences[0]);
      prof2Size = strlen(profile2->sequences[0]);
      pairalignSize = strlen(pairAlignment->sequences[0]);

      alignment = new SequencesSet();
      alignment->nSequences = profile1->nSequences + profile2->nSequences;
      for(i = 0; i < alignment->nSequences; i++) {
            alignment->sequences[i] = new char[MAX_SEQ_SIZE * inputSequences-
>nSequences];
            alignment->sequences[i][0] = '\0';
      }
      for(i = 0; i < profile1->nSequences; i++) {
            alignment->names[i] = new char[MAX_NAME_SIZE];
            strcpy(alignment->names[i], profile1->names[i]);
            alignment->index[i] = profile1->index[i];
      }
      for(i = 0; i < profile2->nSequences; i++) {
            alignment->names[i + profile1->nSequences] = new
char[MAX_NAME_SIZE];
            strcpy(alignment->names[i + profile1->nSequences], profile2-
>names[i]);
            alignment->index[i + profile1->nSequences] = profile2->index[i];
      }

      i = 0; j = 0; k = 0;
      while((i < prof1Size) || (j < prof2Size) || (k < pairalignSize)) {
            if(j < prof2Size) {
                  if((k >= pairalignSize) && (profile2->sequences[n][j] == '-
')) {
                        if(i < prof1Size) {
                              if(profile1->sequences[m][i] == '-') {
                                    for(l = 0; l < profile1->nSequences;
l++)
                                          strrcat(alignment->sequences[l],
profile1->sequences[l][i]);
                                    i++;
                                    j++;
                                    continue;
                              }
                        }
                        for(l = 0; l < profile1->nSequences; l++)
                              strrcat(alignment->sequences[l], '-');
                        j++;
                        continue;
                  } else {
                  if((pairAlignment->sequences[1][k] != '-') && (profile2-
>sequences[n][j] == '-')) {
```

```
                        if(i < prof1Size) {
                              if(profile1->sequences[m][i] == '-') {
                                    for(l = 0; l < profile1->nSequences;
l++)
                                          strrcat(alignment->sequences[l],
profile1->sequences[l][i]);
                                    i++;
                                    j++;
                                    continue;
                              }
                        }
                        for(l = 0; l < profile1->nSequences;l++)
                              strrcat(alignment->sequences[l], '-');
                        j++;
                        continue;
                  }
                  }
            }
            if(k < pairalignSize) {
                  if((i >= prof1Size) && (pairAlignment->sequences[0][k] ==
'-')) {
                        for(l = 0; l < profile1->nSequences; l++)
                              strrcat(alignment->sequences[l], '-');
                        k++; j++;
                        continue;
                  } else {
                  if((pairAlignment->sequences[0][k] == '-')  && (profile1-
>sequences[m][i] != '-')) {
                        for(l = 0; l < profile1->nSequences; l++)
                              strrcat(alignment->sequences[l], '-');
                        k++; j++;
                        continue;
                  }
                  }
            }
            if(i < prof1Size) {
                  if(((pairAlignment->sequences[0][k] != '-') && (profile1-
>sequences[m][i] != '-'))
                  || ((pairAlignment->sequences[0][k] == '-') && (profile1-
>sequences[m][i] == '-'))) {
                        if(((pairAlignment->sequences[1][k] != '-') &&
(profile2->sequences[n][j] != '-'))
                        || ((pairAlignment->sequences[1][k] == '-') &&
(profile2->sequences[n][j] == '-')))
                              j++;
                        k++;
                  } else {
                  if(((profile1->sequences[m][i] != '-') && (profile2-
>sequences[n][j] != '-'))
                  || ((profile1->sequences[m][i] == '-') && (profile2-
>sequences[n][j] == '-')))
                        j++;
                  }
                  for(l = 0; l < profile1->nSequences; l++)
                        strrcat(alignment->sequences[l], profile1-
>sequences[l][i]);
                  i++;
            }
      }

      i = 0; j = 0; k = 0;
      while((i < prof1Size) || (j < prof2Size) || (k < pairalignSize)) {
            if(i < prof1Size) {
                  if((k >= pairalignSize) && (profile1->sequences[m][i] == '-
')) {
                        if(j < prof2Size) {
                              if(profile2->sequences[n][j] == '-') {
```

```
                                        for(l = 0; l < profile2->nSequences;
l++)
                                              strrcat(alignment->sequences[l +
profile1->nSequences], profile2->sequences[l][j]);
                                        j++;
                                        i++;
                                        continue;
                                  }
                            }
                            for(l = 0; l < profile2->nSequences; l++)
                                  strrcat(alignment->sequences[l + profile1-
>nSequences], '-');
                            i++;
                            continue;
                      } else {
                      if((pairAlignment->sequences[0][k] != '-') && (profile1-
>sequences[m][i] == '-')) {
                            if(j < prof2Size) {
                                  if(profile2->sequences[n][j] == '-') {
                                        for(l = 0; l < profile2->nSequences;
l++)
                                              strrcat(alignment->sequences[l +
profile1->nSequences], profile2->sequences[l][j]);
                                        j++;
                                        i++;
                                        continue;
                                  }
                            }
                            for(l = 0; l < profile2->nSequences; l++)
                                  strrcat(alignment->sequences[l + profile1-
>nSequences], '-');
                            i++;
                            continue;
                      }
                      }
                }
                if(k < pairalignSize) {
                      if((j >= prof2Size) && (pairAlignment->sequences[1][k] ==
'-')) {
                            for(l = 0; l < profile2->nSequences; l++)
                                  strrcat(alignment->sequences[l + profile1-
>nSequences], '-');
                            k++; i++;
                            continue;
                      } else {
                      if((pairAlignment->sequences[1][k] == '-') && (profile2-
>sequences[n][j] != '-')) {
                            for(l = 0; l < profile2->nSequences; l++)
                                  strrcat(alignment->sequences[l + profile1-
>nSequences], '-');
                            k++; i++;
                            continue;
                      }
                      }
                }
                if(j < prof2Size) {
                      if(((pairAlignment->sequences[1][k] != '-') && (profile2-
>sequences[n][j] != '-'))
                      || ((pairAlignment->sequences[1][k] == '-') && (profile2-
>sequences[n][j] == '-'))) {
                            if(((pairAlignment->sequences[0][k] != '-') &&
(profile1->sequences[m][i] != '-'))
                        || ((pairAlignment->sequences[0][k] == '-') &&
(profile1->sequences[m][i] == '-')))
                                  i++;
                            k++;
                      } else {
```

```
                        if(((profile2->sequences[n][j] != '-') && (profile1-
>sequences[m][i] != '-'))
                            || ((profile2->sequences[n][j] == '-') && (profile1-
>sequences[m][i] == '-')))
                                i++;
                    }
                    for(l = 0; l < profile2->nSequences; l++)
                        strrcat(alignment->sequences[l + profile1-
>nSequences], profile2->sequences[l][j]);
                    j++;
            }
      }
      delete(pairAlignment);
      delete(profile1);
      delete(profile2);
      return alignment;
}
```

```
//
// scoringfunction.h
//

#ifndef _scoringfunction_
#define _scoringfunction_

	#include "util.h"

	void setScoringFunction(int type);
	void setMatrix(int type);
	double p(char a, char b);
	double getOpenGapPenalty();
	double getGapPenalty();

#endif
```

```
//
// scoringfunction.h
//

#include "scoringfunction.h"

extern double gapPenalty;
extern double openGapPenalty;

int function;

char *aminoacids = "ABCDEFGHIKLMNPQRSTVWXYZ";
char *nucleicAcids = "ABCDGHKMNRSTUVWXY";

//                    A  B  C  D  E  F  G  H  I  J  K   L   M   N   O   P   Q
R   S   T   U   V   W   X   Y   Z
int indexAA[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 0, 10, 11, 12, 13, 0 , 15, 16, 17,
18, 19, 0 , 21, 22, 23, 24, 25};

double matrix[23][23];

int blosum30[]={
  4,
  0,  5,
 -3, -2, 17,
  0,  5, -3,  9,
  0,  0,  1,  1,  6,
 -2, -3, -3, -5, -4, 10,
  0,  0, -4, -1, -2, -3,  8,
 -2, -2, -5, -2,  0, -3, -3, 14,
  0, -2, -2, -4, -3,  0, -1, -2,  6,
  0,  0, -3,  0,  2, -1, -1, -2, -2,  4,
 -1, -1,  0, -1, -1,  2, -2, -1,  2, -2,  4,
  1, -2, -2, -3, -1, -2, -2,  2,  1,  2,  2,  6,
  0,  4, -1,  1, -1, -1,  0, -1,  0,  0, -2,  0,  8,
 -1, -2, -3, -1,  1, -4, -1,  1, -3,  1, -3, -4, -3, 11,
  1, -1, -2, -1,  2, -3, -2,  0, -2,  0, -2, -1, -1,  0,  8,
 -1, -2, -2, -1, -1, -1, -2, -1, -3,  1, -2,  0, -2, -1,  3,  8,
  1,  0, -2,  0,  0, -1,  0, -1, -1,  0, -2, -2,  0, -1, -1, -1,  4,
  1,  0, -2, -1, -2, -2, -2, -2,  0, -1,  0,  0,  1,  0,  0, -3,  2,  5,
  1, -2, -2, -2, -3,  1, -3, -3,  4, -2,  1,  0, -2, -4, -3, -1, -1,  1,  5,
 -5, -5, -2, -4, -1,  1,  1, -5, -3, -2, -2, -3, -7, -3, -1,  0, -3, -5, -3,
20,
  0, -1, -2, -1, -1, -1, -1, -1,  0,  0,  0,  0,  0, -1,  0, -1,  0,  0,  0, -
2, -1,
 -4, -3, -6, -1, -2,  3, -3,  0, -1, -1,  3, -1, -4, -2, -1,  0, -2, -1,  1,
5, -1,  9,
  0,  0,  0,  0,  5, -4, -2,  0, -3,  1, -1, -1, -1,  0,  4,  0, -1, -1, -3, -
1,  0, -2,  4};
```

```
int blosum62[]={
  4,
 -2,  4,
  0, -3,  9,
 -2,  4, -3,  6,
 -1,  1, -4,  2,  5,
 -2, -3, -2, -3, -3,  6,
  0, -1, -3, -1, -2, -3,  6,
 -2,  0, -3, -1,  0, -1, -2,  8,
 -1, -3, -1, -3, -3,  0, -4, -3,  4,
 -1,  0, -3, -1,  1, -3, -2, -1, -3,  5,
 -1, -4, -1, -4, -3,  0, -4, -3,  2, -2,  4,
 -1, -3, -1, -3, -2,  0, -3, -2,  1, -1,  2,  5,
 -2,  3, -3,  1,  0, -3,  0,  1, -3,  0, -3, -2,  6,
 -1, -2, -3, -1, -1, -4, -2, -2, -3, -1, -3, -2, -2,  7,
 -1,  0, -3,  0,  2, -3, -2,  0, -3,  1, -2,  0,  0, -1,  5,
 -1, -1, -3, -2,  0, -3, -2,  0, -3,  2, -2, -1,  0, -2,  1,  5,
  1,  0, -1,  0,  0, -2,  0, -1, -2,  0, -2, -1,  1, -1,  0, -1,  4,
  0, -1, -1, -1, -1, -2, -2, -2, -1, -1, -1, -1,  0, -1, -1, -1,  1,  5,
  0, -3, -1, -3, -2, -1, -3, -3,  3, -2,  1,  1, -3, -2, -2, -3, -2,  0,  4,
 -3, -4, -2, -4, -3,  1, -2, -2, -3, -3, -2, -1, -4, -4, -2, -3, -3, -2, -3,
11,
  0, -1, -2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -2, -1, -1,  0,  0, -1, -
2, -1,
 -2, -3, -2, -3, -2,  3, -3,  2, -1, -2, -1, -1, -2, -3, -1, -2, -2, -2, -1,
2, -1,  7,
 -1,  1, -3,  1,  4, -3, -2,  0, -3,  1, -3, -1,  0, -1,  3,  0,  0, -1, -2, -
3, -1, -2,  4};

int gonnet250[]={
  24,
   0,   0,
   5,   0, 115,
  -3,   0, -32,  47,
   0,   0, -30,  27,  36,
 -23,   0,  -8, -45, -39,  70,
   5,   0, -20,   1,  -8, -52,  66,
  -8,   0, -13,   4,   4,  -1, -14,  60,
  -8,   0, -11, -38, -27,  10, -45, -22,  40,
  -4,   0, -28,   5,  12, -33, -11,   6, -21,  32,
 -12,   0, -15, -40, -28,  20, -44, -19,  28, -21,  40,
  -7,   0,  -9, -30, -20,  16, -35, -13,  25, -14,  28,  43,
  -3,   0, -18,  22,   9, -31,   4,  12, -28,   8, -30, -22,  38,
   3,   0, -31,  -7,  -5, -38, -16, -11, -26,  -6, -23, -24,  -9,  76,
  -2,   0, -24,   9,  17, -26, -10,  12, -19,  15, -16, -10,   7,  -2,  27,
  -6,   0, -22,  -3,   4, -32, -10,   6, -24,  27, -22, -17,   3,  -9,  15,
47,
  11,   0,   1,   5,   2, -28,   4,  -2, -18,   1, -21, -14,   9,   4,   2,  -
2,  22,
   6,   0,  -5,   0,  -1, -22, -11,  -3,  -6,   1, -13,  -6,   5,   1,   0,  -
2,  15,  25,
   1,   0,   0, -29, -19,   1, -33, -20,  31, -17,  18,  16, -22, -18, -15, -
20, -10,   0,  34,
 -36,   0, -10, -52, -43,  36, -40,  -8, -18, -35,  -7, -10, -36, -50, -27, -
16, -33, -35, -26, 142,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,   0,   0,   0,   0,   0,
 -22,   0,  -5, -28, -27,  51, -40,  22,  -7, -21,   0,  -2, -14, -31, -17, -
18, -19, -19, -11,  41,   0,  78,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,   0,   0,   0,   0,   0,   0,   0};

int clustalvdnamt[]={
 10,
  0,  0,
  0,  0, 10,
  0,  0,  0,  0,
  0,  0,  0,  0, 10,
```

```
  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 10,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 10,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0};

void setScoringFunction(int type) {
      int k = 0;
      function = type;
      for(int j = 0; j < 23; j++) {
            for(int i = 0; i <= j; i++) {
                  switch(type) {
                        case BLOSUM30: {
                              matrix[i][j] = blosum30[k];
                              matrix[j][i] = blosum30[k];
                        } break;
                        case BLOSUM62: {
                              matrix[i][j] = blosum62[k];
                              matrix[j][i] = blosum62[k];
                        } break;
                        case GONNET250: {
                              matrix[i][j] = gonnet250[k] / 10;
                              matrix[j][i] = gonnet250[k] / 10;
                        }
                  }
                  k++;
            }
      }
}

double p(char a, char b) {
      if (a >= 97 && a <= 122)
            a = a - 32;
      if (b >= 97 && b <= 122)
            b = b - 32;
      if(function == DEFAULT) {
            if(a == b) {
                  return 10;
            }
            else
                  return 0;
      }
      return matrix[indexAA[a - 65]][indexAA[b - 65]];
}
```

```
//
// fileIO.h
//

#ifndef _fileIO_
#define _fileIO_

	#include "util.h"

	SequencesSet* read_file(char *nomeArquivo);

#endif
```

```
//
// fileIO.cpp
//

#include "fileIO.h"

SequencesSet* read_file(char *nomeArquivo) {
	SequencesSet *seqset = new SequencesSet();
	seqset->nSequences = 0;

	FILE *sequencesFile = fopen(nomeArquivo, "r");
	if (sequencesFile == NULL) {
		return seqset;
	}

	char c;
	c = fgetc(sequencesFile);
	while (!feof(sequencesFile)){
		while ((c != '>') && !feof(sequencesFile)) {
			c = fgetc(sequencesFile);
		}

		seqset->sequences[seqset->nSequences]    = new
char[MAX_SEQ_SIZE];
		seqset->sequences[seqset->nSequences][0] = '\0';
		seqset->names[seqset->nSequences]        = new
char[MAX_NAME_SIZE];
		seqset->names[seqset->nSequences][0]     = '\0';

		c = fgetc(sequencesFile);
		while ((c != 13) && (c != 10) && !feof(sequencesFile)) {
			if(strlen(seqset->names[seqset->nSequences]) <
(MAX_NAME_SIZE - 1))
				strrcat(seqset->names[seqset->nSequences], c);
			c = fgetc(sequencesFile);
		}

		c = fgetc(sequencesFile);
		while ((c != '>') && !feof(sequencesFile)) {
			if((c != 13) && (c != 10) && (strlen(seqset-
>sequences[seqset->nSequences]) < (MAX_SEQ_SIZE - 1)))
				strrcat(seqset->sequences[seqset->nSequences], c);
			c = fgetc(sequencesFile);
		}
		seqset->nSequences++;
	}
	fclose(sequencesFile);
	return seqset;
}
```

```
//
// util.h
//

#ifndef _util_
#define _util_

	#define	MOST_NEGATIVE_INT	-2147483647

	#define		MAX_SEQ_SIZE	3001 // should be desired value + 1
	#define			MAX_NAME_SIZE	257  // should be desired value
+ 1

	// Scoring Matrices
	#define			DEFAULT	0
	#define		BLOSUM30	1
	#define		BLOSUM62	2
	#define		GONNET250	3


	#define max(a,b) ((b)>(a)?(b):(a))
	#define min(a,b) ((b)<(a)?(b):(a))

	#define max3(a, b, c) max(a, max(b, c))

	#include <stdio.h>
	#include <string.h>

	typedef struct {
		int nSequences;
		char *names[128];
		char *sequences[128];
		int index[128];
	} SequencesSet;

	char *strlcat(char c, char *string);
	char *strrcat(char *string, char c);

#endif
```

```
//
// util.cpp
//

#include "util.h"

char *strlcat(char c, char *string) {
	char *newstring = new char[strlen(string) + 2];
	newstring[0] = c;
	newstring[1] = '\0';
	strcat(newstring, string);
	strcpy(string, newstring);
	delete(newstring);
	return string;
}

char *strrcat(char *string, char c) {
	char *newstring = new char[strlen(string) + 2];
	strcpy(newstring, string);
	newstring[strlen(string)] = c;
	newstring[strlen(string) + 1] = '\0';
	strcpy(string, newstring);
	delete(newstring);
	return string;
}
```