PROCESSAMENTO PARALELO PARA DETECÇÃO DE HOMOLOGIA REMOTA COM MAP/REDUCE

Carlos Eduardo Santanna da Silva

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

Orientador: Alexandre de Assis Bento Lima

Rio de Janeiro
Outubro de 2013

PROCESSAMENTO PARALELO PARA DETECÇÃO DE HOMOLOGIA REMOTA COM MAP/REDUCE

Carlos Eduardo Santanna da Silva

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

Examinada por:

____

Prof. Alexandre de Assis Bento Lima, D.Sc.

____

Prof. Marta Lima de Queirós Mattoso, D.Sc.

____

Prof. Fábio André Machado Porto, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

OUTUBRO DE 2013

Silva, Carlos Eduardo Santanna da

Processamento Paralelo para Detecção de Homologia Remota com Map/Reduce/ Carlos Eduardo Santanna da Silva. – Rio de Janeiro: UFRJ/COPPE, 2013.

XI, 104 p.: il.; 29,7 cm.

Orientador: Alexandre de Assis Bento Lima

Dissertação (mestrado) – UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 84-90.

1. Map/Reduce. 2. Hidden Markov Models. 3. *Workflow* Científico. I. Lima, Alexandre de Assis Bento. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedicatória

Dedico este trabalho a todas as pessoas que acreditaram em mim desde o início do meu curso de Mestrado, e me deram todo o apoio para realizar este sonho.

Agradecimentos

Quero agradecer primeiramente a Deus, pois me deu forças e sabedoria nesta longa jornada e a oportunidade de estudar e chegar até onde estou agora, na conclusão do meu curso de Mestrado.

Agradeço também aos meus pais e minhas irmãs, que me deram força e todo o apoio que precisei. Aos meus pais pela educação, experiência de vida e conselhos, às minhas irmãs pelo carinho, cuidado e incentivo especial que elas me propuseram.

Aos professores que me ajudaram a conhecer melhor a área de Computação e passarem seus conhecimentos da melhor forma possível, por me mostrarem que estudar com ética, responsabilidade, vontade e organização são as virtudes mais importantes para se crescer academicamente e profissionalmente. Gostaria de citar especialmente os professores Alexandre de Assis Bento Lima e Marta Lima de Queirós Mattoso.

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

PROCESSAMENTO PARALELO PARA DETECÇÃO DE HOMOLOGIA REMOTA COM MAP/REDUCE

Carlos Eduardo Santanna da Silva

Outubro/2013

Orientador: Alexandre de Assis Bento Lima

Programa: Engenharia de Sistemas e Computação

A detecção de homologia remota é uma das atividades mais importantes nos estudos de genética comparativa. Essas atividades demandam recursos computacionais de alto desempenho, em função do volume de dados que a ser processado. O paradigma de programação Map/Reduce vem sendo utilizado como alternativa às ferramentas atualmente existentes para a obtenção de processamento alto desempenho nesta área. O uso desse paradigma, no entanto, tem gerado problemas de balanceamento de carga durante a execução das atividades, gerando desperdício de recursos do *cluster* de computadores em que estão sendo executadas. Essa dissertação apresenta uma solução para a melhora do balanceamento de carga em atividades de detecção de homologia remota em ambientes Map/Reduce. Uma abordagem para fragmentação de dados é proposta e os resultados experimentais obtidos são bastante promissores.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PARALLEL PROCESSING FOR REMOTE HOMOLOGY DETECTION WITH MAP/REDUCE

Carlos Eduardo Santanna da Silva

October/2013

Advisor: Alexandre de Assis Bento Lima
Department: Systems and Computing Engineering

The remote homology detection is one of the most important tasks in comparative genomic studies. These tasks demand high performance computational resources due to the huge amounts of data to be processed. The Map/Reduce paradigm has been used as an alternative to the traditional tools employed in this area for achieving high performances. The use of this paradigm, however, has load balance issues when running such activities, wasting resources in computer clusters. This dissertation presents a solution for the improvement of load balancing on remote homology detection tasks in Map/Reduce environments. A new data partitioning approach is proposed and experimental results are very promising.

Índice

Capítulo 1 - Introdução 1

1.1 Motivação 4

1.2 Objetivo 5

1.3 Organização da Dissertação 7

Capítulo 2 – *Workflows Científicos*, Bioinformática e Map/Reduce 9

2.1 Workflows Científicos e Bioinformática 10

2.1.1 Workflows Científicos e SGWfC 10

2.1.2 Perfil do Modelo Escondido de Markov e HMMER 15

2.2 Map/Reduce 20

2.2.1 O Modelo de Programação Map/Reduce em Linguagens Funcionais .. 20

2.2.2. O Modelo de Programação Map/Reduce da Google 24

2.2.2.1 O *Framework* MapReduce 24

2.2.2.2 Mapeadores e Redutores 26

2.2.2.3 Arquitetura do *Framework* MapReduce 28

2.2.2.4 Sistema Distribuído de Arquivos 30

2.2.3 Hadoop 31

2.2.3.1 Mapeadores e Redutores 32

2.2.3.2 Arquitetura 33

2.2.3.3 Sistema de Arquivos Distribuídos do Hadoop 34

2.2.3.4 Hadoop Streaming 34

2.3 Uso de processamento paralelo em Bioinformática 35

2.3.1 CloudBLAST 35

Capítulo 3 - Alto Desempenho em Bioinformática: um Estudo de Caso 38

Capítulo 4 - Paralelização da Busca de Similaridades no SciHmm 45

4.1 Estratégia de Paralelização 45

4.2 Detalhes de Implementação 50

4.2.1 FastaInputFormat 51

4.2.2 FastaRecordReader 55

4.2.3 Função map 57

4.2.4 Função reduce 62

4.2.5 MultiFastaOutputFormat 62

Capítulo 5 - Resultados Experimentais 65
5.1 Ambiente de Execução 65
5.1.1 Hardware e Sistema Operacional 65
5.1.2 Softwares 66
5.1.3 Parâmetros de Execução do hmmsearch 66
5.1.4 Configuração da Execução da Tarefa Hadoop 67
5.1.5 Conjunto de Dados 68
5.1.6 Cenários 68
5.2 Resultados 69
2198 75
1149 75
642 75
427 75
290 75
Capítulo 6 - Conclusão 80
6.1 Contribuições 81
6.2 Limitações do Trabalho 81
6.3 Trabalhos Futuros 82
Referências 84
Apêndice 91

Índice de Figuras

Figura 1 - Arquitetura conceitual do SciCumulus (OCAÑA *et al.,* 2013) 15
Figura 2 - Modelo Escondido de Markov simples (EDDY, 1996)................................ 16
Figura 3 - Perfil do Modelo Escondido de Markov simples 19
Figura 4 - Exemplo de uma lista em Lisp... 21
Figura 5 - Exemplo de função em Lisp 21
Figura 6 - Processamento das listas L pela função f em L' 21
Figura 7 - Exemplo de uma função reduce em Lisp... 22
Figura 8 - Exemplo de uma função map e flodl em Haskell 22
Figura 9 - Função map em Haskell.. 23
Figura 10 - Função *foldl* 23
Figura 11 - Esquema de processamento com map e reduce (LIN *et al.*, 2010).............. 27
Figura 12 - Estrutura das atividades do workflow SciHmm (OCAÑA *et al.*, 2013)...... 41
Figura 13 - Esquema de paralelização do workflow SciHmm 46
Figura 14 - Processo de fragmentação do arquivo pelo InputFormat do Hadoop.......... 51
Figura 15 - Exemplo de arquivos FASTA originais.. 53
Figura 16 - Exemplo de arquivo com as sequências FASTA após o processo de concatenação.. 53
Figura 17 - Variação do tempo de execução (em segundos)... 71
Figura 18 - Variação do tempo de execução dos novos cenários (em segundos) 75
Figura 19 - Variação do fator de aceleração dos novos cenários 76
Figura 20 - Variação da taxa de utilização do processador (em porcentagem).............. 77
Figura 21 - Arquitetura conceitual de integração da solução de paralelização proposta com o SciCummulus... 83

Índice de Tabelas

Tabela 1 - Terminologias básicas utilizadas em estudos de genética comparativa........ 18
Tabela 2 - Tempos médios de execução em cada cenário (em segundos) 70
Tabela 3 - Fator de aceleração dos cenários.. 72
Tabela 4 - Fator de aceleração dos cenários sobre perspectiva do número de nós......... 73
Tabela 5 - Média dos tempos de execução com os novos cenários............................... 75
Tabela 6 - Média do fator de aceleração com os novos cenários 75
Tabela 7 - Média da taxa de utilização uso da cpu em cada cenário (em porcentagem) 77

Capítulo 1 - Introdução

A genética comparativa tem por finalidade comparar genomas com o objetivo de encontrar sequências homólogas através de similaridade. A detecção de homólogos é um dos principais campos de pesquisa nesta área (MILLER, *et. al*, 2004). Ela exerce um papel fundamental em pesquisas ligadas, por exemplo, à filogenia, que tem por objetivo detectar sequências homólogas entre organismos para o estudo da relação evolutiva entre os mesmos. Diversas ferramentas implementam algoritmos que realizam a detecção de homólogos utilizando diferentes abordagens para alcançar tal objetivo. Entre elas estão o HMMER (EDDY, 2011), o BLAST (ALTSCHUL, *et al.*, 1990) e o SAM (HUGHEY, 1995).

O HMMER é um dos softwares de análise de sequências mais utilizados na atualidade. Ele oferece uma série de ferramentas para diversas análises e construção de modelos a partir de sequências de proteínas ou nucleotídeos. Sua metodologia se baseia no Modelo Escondido de Markov (HMM, do inglês *Hidden Markov Model*). O HMM tem sido utilizado com grande sucesso na genética comparativa. Isso se deve ao fato de que este modelo tem conseguido executar atividades como as de detecção de homólogos com rapidez sem interferir na qualidade do resultado. Apesar de o HMMER ser um dos softwares de análise de sequências mais rápidos da atualidade, sua utilização com grandes massas de dados pode se tornar uma atividade que demanda várias horas de processamento.

De maneira geral, devido ao crescente aumento no volume de dados biológicos disponíveis para estudos, atividades como a de detecção de homólogos vem demandando muitos recursos computacionais. Tal fato tem estimulado pesquisas de novas metodologias ou paradigmas de implementação para melhorar seu desempenho.

Ambientes de Computação de Alto Desempenho (HPC, do inglês *High-Performance Computing*) têm sido utilizados para solucionar problemas de desempenho em genética comparativa. O uso de HPC acelera, ou até mesmo torna possível, em alguns casos, a solução de vários problemas na área de Bioinformática, inclusive na genética comparativa. HPC, atualmente, é realizada com o uso de computação paralela e distribuída, onde vários conjuntos de computadores e outros elementos de *hardware* são utilizados de forma coordenada para a execução de uma ou mais tarefas. Nesta área, é muito comum a utilização de *cluster*s de computadores, muitas vezes organizados em grades ou nuvens computacionais, o que permite a divisão de tarefas entre diversos nós para serem executadas em paralelo.

Devido a estas vantagens, diversas pesquisas têm sido feitas para tentar mapear problemas de genética comparativa - como o de detecção de homólogos - em ambientes de HPC. Uma das soluções conhecidas é o CloudBLAST (MATSUNAGA *et al*., 2008). O CloudBLAST consiste em um *framework* para executar o aplicativo BLAST em ambientes de HPC formados por *clusters* e nuvens computacionais. Trata-se de uma solução não-intrusiva, que utiliza a versão sequencial do BLAST em combinação com uma ferramenta de computação paralela para a obtenção de alto desempenho no processo de detecção de similaridades entre sequências de nucleotídeos e proteínas. Os resultados obtidos mostram que a utilização de ambientes de HPC para solucionar problemas de desempenho em atividades de genética comparativa pode ser utilizada com sucesso, possibilitando a obtenção de excelente desempenho (em relação ao tempo de execução) e bons resultados biológicos.

Para a obtenção de paralelismo, o CloudBLAST utiliza o *framework* Hadoop[1], que implementa o paradigma de programação Map/Reduce. O Hadoop é um software livre que implementa o *framework* MapReduce da Google (DEAN, GHEMAWAT, 2004). O MapReduce da Google não apenas implementa o paradigma de programação supracitado como também oferece um ambiente de execução paralela com suporte a falhas. Esta ferramenta vem ganhando bastante espaço na busca por alto desempenho em estudos de genética comparativa.

Outro tipo de solução que vem sendo largamente utilizado na área de computação científica, da qual a Bioinformática faz parte, são os W*orkflows* Científicos (HOLLINGSWORTH,1995). *Workflows* Científicos foram construídos com o objetivo principal de guardar informações importantes sobre proveniência de dados que são gerados a partir dos experimentos científicos, a fim de se ter informações relevantes sobre a linha de tempo (*timeline*) dos resultados gerados (DAVIDSON, FREIRE, 2008) durante os experimentos. Vários Sistemas de Gerência de Workflows Científicos (SGWfC) surgiram ao longo dos últimos anos e vêm sendo utilizados com sucesso em pesquisas de Bioinformática, como por exemplo o Vistrails (BAVOIL *et al.*, 2005) e o Swift (WILDE *et al.*, 2011). Muitos deles permitem a utilização de ambientes de HPC - sejam eles na nuvem, como o SciCumulus (OLIVEIRA *et al.*, 2010), ou utilizando *cluster* ou grades computacionais, como o Swift -, possibilitando que suas atividades sejam executadas em paralelo.

Dadas as vantagens desta tecnologia, o *workflow* SciHmm (OCAÑA *et. al*, 2013) foi desenvolvido com o objetivo de auxiliar cientistas no processo de testes de diferentes abordagens de Alinhamento Múltiplo de Sequências (MSA, do inglês *Multiple Sequence Allignment*) - MSA é o processo de alinhar um conjunto de mais de três

[1] http://hadoop.apache.org/

sequências biológicas - facilitando a escolha do método mais eficaz para um determinado estudo de filogenia que se deseje executar. Uma das principais atividades do *workflow* é a detecção de homologia remota, que é executada com a ferramenta HMMER, que utiliza HMM. OCAÑA *et al.* (2013) apresentam resultados relativos à execução do SciHmm em uma nuvem computacional. O desempenho obtido com o uso de HPC com processamento paralelo foi bastante superior ao obtido com a execução sequencial. Um alto grau de paralelismo foi conseguido na atividade de geração dos MSA por cinco ferramentas distintas. No entanto, durante a atividade de detecção de homologia remota utilizando os resultados produzidos pelas ferramentas de MSA, foi empregado um grau bastante reduzido de paralelismo: os resultados produzidos por cada ferramenta eram processados sequencialmente. Isto fez com que o tempo consumido em sua execução fosse bastante elevado. A investigação de uma estratégia para aumentar o desempenho obtido nesta fase em especial do *workflow* é a motivação desta dissertação.

1.1 Motivação

O *workflow* SciHMM possui cinco atividades. A primeira consiste em executar cinco diferentes ferramentas de MSA que geram os dados utilizados nas atividades seguintes. Nesta fase, atinge-se alto grau de paralelismo. No entanto, na terceira fase, que executa a detecção de homologia remota, o grau de paralelismo obtido é relativamente baixo. Como cinco ferramentas de MSA são utilizadas, cinco linhas de execução são criadas nesta fase, sendo, cada uma, destinada a processar os resultados produzidos a partir de uma ferramenta de MSA. Desta forma, cada subatividade da atividade de detecção de homologia é, na verdade, executada sequencialmente, o que faz com que ela seja uma das mais lentas do *workflow*.

O uso do paradigma de programação Map/Reduce vem sendo empregado com bastante sucesso em problemas de Bioinformática. Foi observado que o problema de baixo grau de paralelismo descrito acima poderia ser solucionado utilizando Map/Reduce. Essa percepção se deu ao fato de sua estrutura utilizar funções *map* e *reduce* de forma independente entre si e paralela, onde funções *map* recebem dados previamente fragmentados para serem processados e enviam seus resultados para funções *reduce*, que realizam seu agrupamento e produção do resultado final.

A motivação desta dissertação é, dada uma subatividade de detecção de homologia remota, investigar e propor estratégias para a sua execução com alto grau de paralelismo utilizando o paradigma de linguagem Map/Reduce, aproveitando o máximo de recursos disponíveis no ambiente de HPC onde se dá a execução do *workflow*.

1.2 Objetivo

Essa dissertação tem por objetivo principal propor uma solução para melhoria do desempenho da atividade de detecção de homologia remota do *workflow* SciHmm. Para isso, foram implementadas estratégias baseadas nas propostas do CloudBLAST. MATSUNAGA *et. al*. (2008) sinalizam que abordagem semelhante poderia ser utilizada com a ferramenta Hmmer. Nesta dissertação é avaliada a alternativa de se utilizar o HMMER em associação com Map/Reduce através do *framework* Hadoop.

O CloudBLAST fragmenta os dados de entrada do BLAST, mas nada é dito a respeito dos critérios de fragmentação, dos tamanhos dos fragmentos e da sua quantidade. Nesta dissertação, é feito um estudo neste sentido a fim de possibilitar a tomada de decisões a respeito das melhores estratégias de fragmentação a serem utilizadas com o HMMER para a detecção de homologia remota.

Uma das principais dificuldades surgidas em quase todas as atividades que empregam processamento paralelo é a obtenção de um bom balanceamento de carga, que consiste em fazer com que cada elemento computacional despenda aproximadamente a mesma quantidade de tempo na execução de suas tarefas, evitando a sobrecarga de alguns deles, bem como a subutilização de outros. O estudo realizado para avaliar os tamanhos dos fragmentos de dados processados por cada instância do HMMER bem como a quantidade de fragmentos tem por objetivo encontrar a configuração que proporcione o melhor balanceamento de carga durante a detecção de homologia remota.

Ao longo do desenvolvimento dessa dissertação, foram experimentadas outras soluções de paralelismo, como foi o caso do uso do paradigma da Interface de Passagem de Mensagens (MPI, do inglês *Message Passing Interface*) (GROPP *et al.*, 1999) - que é um padrão de troca de mensagens muito utilizado em HPC - implementada pelo próprio HMMER. Foi detectado um grande desbalanceamento de carga no uso da solução MPI do HMMER, afetando o desempenho e, por conseguinte, o tempo de execução da atividade. O problema relevante que foi tratado nesta dissertação foi como melhorar o desempenho e o tempo de execução de atividades de detecção de homologia remota sobre grandes massas de dados utilizando uma arquitetura HPC com o paradigma de programação Map/Reduce (através do *framework* Hadoop) e o HMMER.

Resumindo, o objetivo dessa dissertação é desenvolver uma estratégia que melhore o desempenho de execução de atividades de detecção de homologia remota no *workflow* SciHmm utilizando o HMMER e o framework Hadoop combinados com uma boa estratégia de fragmentação e distribuição de dados. Esta estratégia é uma extensão da proposta pelo CloudBLAST.

Com a finalidade de alcançar o objetivo proposto, foi realizada uma revisão na literatura, o que nos permitiu detectar que não existem atualmente soluções para a paralelização de detecção de homologia remota utilizando o HMMER e o paradigma de programação Map/Reduce. Posteriormente, foi verificado que o problema de balanceamento de carga ainda persistia em soluções próximas à que foi proposta por essa dissertação, como é o caso do CloudBLAST e do MPI do HMMER.

Foi elaborada uma estratégia personalizada de fragmentação dos dados de entrada a partir do uso do *framework* Hadoop. O processo de fragmentação foi embasado na arquitetura do próprio *framework*, assim também como no tamanho do *cluster* utilizado no que diz respeito ao número de nós. Além da solução através de um processo de fragmentação, foi analisada também uma segunda opção de paralelização através da distribuição automática dos dados entre os nós do *cluster* efetuado pelo próprio *framework*, sem a necessidade de fragmentação personalizada de dados.

Os resultados experimentais obtidos validam de forma positiva o uso de Map/Reduce e ambientes HPC para solucionar problemas de desempenho em detecção de homologia remota. Foi obtido um fator de aceleração de 22.88, e uma redução de 95.62% no tempo total de execução em comparação com melhor resultado da execução sequencial quando foi usado um *cluster* com 32 nós e um fator de fragmentação igual a 64 fragmentos. Além disso, vários outros cenários apresentarem fatores de aceleração linear e super linear.

1.3 Organização da Dissertação

Essa dissertação está organizada da seguinte forma. O Capítulo 2 apresenta uma breve definição dos principais conceitos utilizados, entre eles, *Workflows* Científicos, Bioinformática (perfil do Modelo Escondido de Markov e HMMER), *framework*

Hadoop, assim como o paradigma Map/Reduce, além de alguns trabalhos relacionados, que utilizam processamento paralelo para a execução de atividades de genética comparativa com o uso de arquitetura HPC. O Capítulo 3 descreve o workflow SciHmm e suas respectivas atividades, em especial a atividade de detecção de homologia remota, alvo dessa dissertação. O Capítulo 4 apresenta a estratégia de paralelização adotada e seus detalhes implementação. No Capítulo 5, são descritos e analisados os resultados experimentais obtidos a fim de validar a estratégia proposta. Por fim, o capítulo 6 apresenta a conclusão, evidenciando as principais contribuições, suas limitações e relacionando trabalhos futuros que porventura possam ser efetuados a partir dos resultados obtidos.

Capítulo 2 – *Workflows Científicos*, Bioinformática e Map/Reduce

Com o passar dos anos, a necessidade de se tratar grandes volumes de dados em várias áreas de pesquisa vem aumentando, o que demanda equipamentos de maior desempenho e novas metodologias de processamento. Técnicas de processamento paralelo têm sido usadas demasiadamente na tentativa de se obter resultados em um espaço de tempo aceitável (LEE, *et al.*, 2011). Com a popularização do uso de paralelismo no processamento de dados, modelos de programação como o Map/Reduce têm obtido bastante repercussão por seu sucesso, tanto no mundo acadêmico como no corporativo.

Além do fato de buscar formas de processar dados mais rapidamente, há também a necessidade de se organizar pesquisas, armazenar modificações feitas durante cada um desses processos e inferir sobre os resultados gerados. *Workflow*s científicos vêm se mostrando bastante versáteis neste propósito, agregando muito valor para a execução de atividades paralelas em ambientes propícios para isso.

O capítulo 2 está divido da seguinte forma. A seção 2.1 descreve a definição de *workflows* científicos, os principais atributos dos Sistemas Gerenciadores de *Workflows* Científicos, o perfil baseado no Modelo Escondido de Markov e o HMMER. A seção 2.2 descreve as principais funcionalidades e características do Map/Reduce, desde sua utilização em linguagens Funcionais, seguido do MapReduce da Google e mais tarde no Hadoop. A seção 2.3 apresenta alguns trabalhos recentes realizados em que fazem uso do processamento paralelo em soluções de problemas em Bioinformática.

2.1 Workflows Científicos e Bioinformática

2.1.1 Workflows Científicos e SGWfC

Workflow é uma tecnologia que vem sendo muito usada atualmente em diversas áreas da indústria e também vem ganhando muito espaço no meio acadêmico. O processo de *workflow* se baseia em automatizar um processo, onde as etapas do mesmo podem ser atividades dos mais variados tipos (HOLLINGSWORTH, 1995). Dentro do processo as atividades podem ser realizadas por uma pessoa, grupo de pessoas ou por um computador.

Baseado nessa lógica de trabalho fica bastante fácil tornar o processo - que está sob os moldes de *workflow* - em um conjunto de atividades distribuídas (BARKER, *et al.*, 2008) para que assim possam ser executadas em paralelo no caso de uma atividade não ser totalmente dependente da anterior. A partir deste princípio, o conceito de *workflow* vem sendo usado bastante no meio acadêmico para obter resultados científicos utilizando recursos e ambientes de computação paralela, e se tornou um forte tema de pesquisa desde então (BARKER, *et al.*, 2008). Apesar de ser um modelo tecnológico desenvolvido e aplicado primeiramente no mundo corporativo, ele tem conseguido bastante sucesso no meio acadêmico, para a condução de experimentos científicos. Para isso, foram necessárias algumas modificações na abordagem da implementação de *workflow*s.

Workflows tradicionais possuem o nível de abstração errado na visão de cientistas, se tornando muito difícil o aproveitamento deste modelo para os mesmos. Sendo assim, workflows científicos possuem diferentes níveis de abstração a fim de permitir ao cientista uma manipulação melhor dos problemas e de possíveis soluções através dos componentes disponibilizados pelo próprio *workflow*. *Workflows* científicos são

utilizados para a condução de experimentos científicos. Por isso, são desenvolvidos e executados com o objetivo de capturar a série de atividades desempenhadas, de modo a permitir a descrição bem sucedida de todo o processo computacional do experimento. Isso exige a disponibilização de um ambiente para se combinar gerência de dados científicos, suas análises, simulações e visualizações de dados e, assim, facilitar o processo da descoberta científica (BARKER, *et al.*, 2008).

Para organizar os *workflows* e gerenciá-los de maneira correta, existem os Sistemas de Gerência de *Workflows* Científicos (SGWfC). SGWfCs podem definir, gerenciar e executar um *workflow*. Através deles é possível elaborar a automação procedimental do *workflow*, invocando cada recurso humano e/ou eletrônico que esteja associado com uma ou mais atividades do mesmo. Além disso, alguns SGWfCs suportam a persistência dos dados resultantes dos experimentos e principalmente dos dados de proveniência que são de suma importância para a linha de tempo da pesquisa. Existem diversos SGWfCs, como Kepler (ALTINTAS *et al.*, 2004), Taverna (OINN *et al.*, 2004), Pegasus (DEELMAN *et al.*, 2003), VisTrails (BAVOIL *et al.*, 2005) e o Swift (WILDE *et al.*, 2011).

O VisTrails é um sistema de gerência de *workflows* científicos que fornece suporte a proveniência. Além disso, ele permite a visualização, exploração de dados e simulações do *workflow*. Um grande diferencial do VisTrails é a sua infraestrutura de proveniência, capaz de manter dados resultantes (ou seja, informações históricas) da atividade do *workflow* e a origem de cada dado gerado no processo, bem como suas respectivas transformações. Além disso, os dados são persistidos em arquivos XML e/ou banco de dados (ficando a critério das necessidades do pesquisador). Ele permite também navegar pelas versões anteriores do mesmo *workflow* a fim de realizar modificações.

O Swift é uma linguagem de script desenvolvida para modelar aplicações a serem executadas em paralelo em ambientes de processamento com múltiplos núcleos, grades computacionais, *cluster*, nuvens e até mesmo em supercomputadores. O Swift paraleliza automaticamente atividades computacionais independentes. Com o Swift é possível executar *workflows* científicos em ambientes HPC e gerenciar/persistir os dados de proveniência das execuções das atividades. Ele é implicitamente paralelo, ou seja, trabalha de forma paralela sempre, independente do que o usuário escreva no *script,* desde que a paralelização seja possível. Sendo assim, não é necessário que o usuário implemente paralelismo no seu código, deixando a cargo da ferramenta tarefas como garantia de sincronismo.

Em comparação com o VisTrails, o Swift fornece uma infraestrutura melhor no que diz respeito ao desempenho, uma vez que ele é preparado para executar atividades extremamente custosas computacionalmente e com grandes quantidades de dados em paralelo. O VisTrails fornece uma interface interativa orientada a componente - o que facilita o trabalho do pesquisador na elaboração do *workflow*.

Além dos SGWfCs apresentados acima, existem também outros mecanismos onde é possível definir *workflows* científicos. Além disso eles possuem suporte a HPC, o que permite que os *workflwos* definidos através desses mecanismos possam ser executados em paralelo. São eles o Chiron (OGASAWARA et al., 2013) e o SciCumulus (OLIVEIRA *et al*., 2010).

Chiron (OGASAWARA *et al.*, 2013) é um mecanismo de paralelização de *workflows* baseado em uma álgebra própria e desenvolvido utilizando a biblioteca MPI executado em ambientes HPC. O modelo algébrico utilizado pelo Chiron é baseado na álgebra relacional utilizada pelos bancos de dados Relacionais, e fornece um modelo de dados bastante uniforme, que expressa todos os dados do experimento como relações.

Desta maneira, a álgebra utilizada pelo Chiron consegue facilmente mapear experimentos em larga escala através de grafos acíclicos direcionados, principalmente os experimentos que permitem o uso de técnicas de varredura de parâmetros (ou seja, permite que o mesmo experimento seja executado repetidamente variando os dados de entrada e a configuração dos parâmetros). O modelo algébrico implementado pelo Chiron gera uma nova tupla para cada configuração e/ou combinação diferente de valores dos parâmetros. As atividades do *workflow* utilizam essas tuplas para a execução das atividades, assim também como são capazes de gerar novas tuplas. Esse tipo de sistema permite que se manipule uma grande quantidade de combinações e configurações de parâmetros, já que eles são vistos apenas como relações.

O mecanismo Chiron pode ser utilizado como um programa independente ou acoplado a um SGWfC. Ele consegue identificar como cada dado utilizado pelo *workflow* é estruturado e o que se deve esperar do resultado de uma determinada atividade no que diz respeito aos dados que serão utilizados e gerados por ela. A partir desta identificação, o Chiron consegue realizar otimizações algébricas sobre o plano de execução do workflow. Chiron também possui um gerenciamento de dados de proveniência único. Ele permite que os dados sejam explorados e/ou cruzados durante e após a execução do *workflow*. Este tipo de recurso não é visto em nenhum SGWfC atualmente, já que os mesmos guardam os dados de proveniência durante a execução em arquivos de *log* temporários, e somente após o término da execução do *workflow* os dados são disponibilizados para serem devidamente analisados, tornando bastante difícil de o cientista realizar qualquer tipo de busca durante a execução do mesmo. O banco de dados relacional PostgreSQL é utilizado pelo Chiron para armazenar os dados de proveniência.

SciCumulus é um *middleware* que tem como objetivo executar *workflows* científicos em ambientes de computação em nuvem e que suporta o paradigma de Computação de Múltiplas Tarefas (MTC, do inglês *Many-Task Computing*). MTC trabalha com a concepção de que os recursos computacionais podem ser utilizados para executar várias tarefas simultaneamente e em um curto espaço de tempo. O SciCumulus utiliza técnicas de fragmentação de dados e de varredura de parâmetros - que é um modelo de execução de *workflows* (WALKER *et. al*, 2007) - para a realização de atividades em paralelo. A varredura de parâmetros permite a execução de uma mesma atividade (ou parte dela) várias vezes, com diferentes entradas de dados e configurações. Com isso, um *workflow* científico consegue executar atividades em paralelo e de forma independente.

No contexto do SciCumulus, cada atividade do *workflow* pode gerar várias atividades que serão executadas em paralelo. Um exemplo é a atividade de construção dos modelos do SciHmm. Esta atividade pode ser mapeada em até 250 atividades paralelas, onde cada uma irá processar cinco conjuntos diferentes produzidos pelas ferramentas de MSA, totalizando assim 1250 atividades (OCAÑA, *et al.*, 2013). O SciCumulus é capaz de distribuir, monitorar e controlar cada uma dessas atividades, que são enviadas a partir da máquina do cientista para a nuvem (como, por exemplo, a Amazon EC2[2]). O *middleware* é composto por quatro camadas. A primeira é a camada do cliente, que é responsável por enviar solicitações de execução do *workflow* para a nuvem. A segunda é a camada de distribuição. Ela é responsável pela gerência das execuções das atividades nos nós envolvidos. A terceira é a camada de execução. Ela é responsável por executar todas as ferramentas que estão encapsuladas nas atividades e armazenar os dados de proveniência. Por último, há a camada de dados, que é responsável por armazenar os dados de entrada e os dados de proveniência que são

[2] http://aws.amazon.com/pt/ec2/

utilizados e gerados durante a execução. A figura 1 ilustra a arquitetura conceitual do SciCumulus. Em vista disso, o SciHmm pôde mapear todas as suas atividades através do SciCumulus e executá-las em um ambiente de HPC.

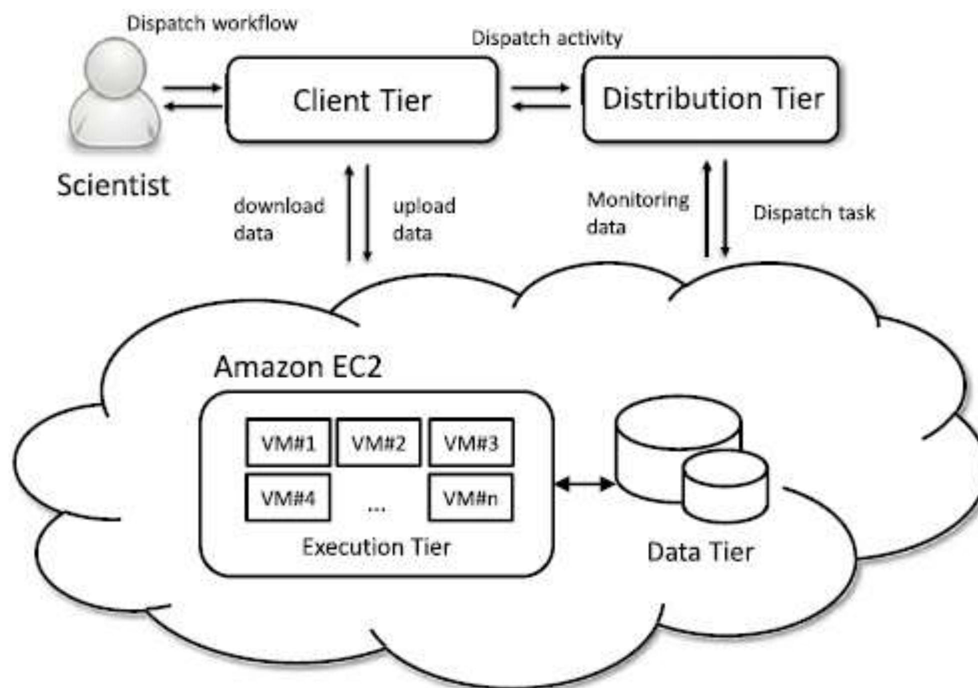

Figura 1 - Arquitetura conceitual do SciCumulus (OCAÑA *et al.,* 2013)

2.1.2 Perfil do Modelo Escondido de Markov e HMMER

HMMER (EDDY, 2011) é um pacote de soluções para análise de sequências construído a partir do modelo probabilístico chamado Modelo Escondido de Markov (HMM, do inglês *Hidden Markov Model*). HMM foi utilizado inicialmente para reconhecimento de padrões de linguagem (RABINER, 1990). O modelo foi introduzido no uso de aplicações de bioinformática no final dos anos 80 (CHURCHILL, 1989), e mais tarde pela primeira vez como modelos de perfis (KROGH *et al.*, 1994) para detectar distâncias entre sequências homólogas. Atualmente, vem conseguindo bastante sucesso entre cientistas e pesquisadores.

Um HMM é um modelo finito que descreve a distribuição de probabilidade de uma quantidade infinita de sequências, ou seja, representa a probabilidade de uma determinada sequência ocorrer dentro de um modelo (EDDY, 1996). Um HMM é composto por vários estados. Cada um desses estados corresponde a posições em uma estrutura 3D (como por exemplo, de uma proteína) ou colunas de um alinhamento múltiplo, dependendo do propósito com o qual o modelo será utilizado. Cada estado emite símbolos de acordo com as probabilidades emissão-símbolo (distribuição de probabilidade do modelo) estabelecidas, gerando como resultado uma sequência de símbolos (também chamados de símbolos visíveis). A figura 2 (a) ilustra um HMM simples. As probabilidades emissão-símbolo para cada símbolo em cada estado é ilustrada junto ao próprio símbolo, por exemplo, o símbolo "A" possui uma probabilidade de 0.4 de ser emitido pelo estado "1", enquanto que no estado "2", o mesmo símbolo tem uma probabilidade de apenas 0.05.

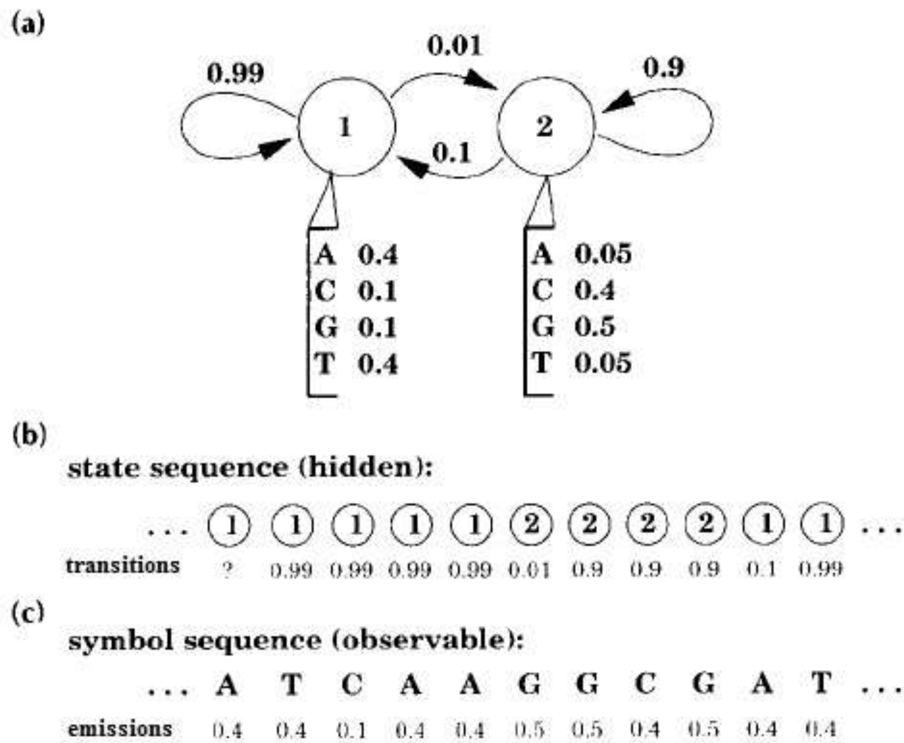

Figura 2 - Modelo Escondido de Markov simples (EDDY, 1996)

A figura 2 (c) ilustra a sequência de símbolos gerada pelo processo. Apesar dos símbolos serem visíveis, em uma cadeia de Markov comum, a sequência de estados que gerou esses símbolos permanece invisível. Neste caso, o HMM gera a sequência de estados (que antes era "escondida") que produziu a sequência de símbolos. A figura 2 (b) ilustra a sequência de estados que gerou a sequência de símbolos ilustrada na figura 2 (c). O estudo a ser feito é inferir a sequência de estados mais provável a partir do alinhamento HMM com a sequência observada (EDDY, 1996).

O uso de HMM tem como objetivo principal tentar solucionar três problemas básicos: o primeiro, a partir de um HMM e de sua sequência de símbolos, descobrir a probabilidade deste mesmo HMM gerar a sequência (conhecido como o problema de classificação); o segundo, descobrir a melhor sequência de estados que pode gerar uma determinada sequência de símbolos (conhecido como o problema de alinhamento); e em terceiro, a partir de um grande volume de dados, descobrir quais parâmetros e estrutura do HMM melhor representam os dados (conhecido como o problema de treinamento). A partir desses problemas, KROGH *et. al* (1994) analisaram que perfis podem ser reescritos de forma mais genéricas como HMM, já que os problemas descritos acima são análogos aos problemas de classificação de sequência com perfis, descoberta do melhor alinhamento sequência-perfil e a construção de perfis a partir de proteínas ou sequências de DNA alinhadas ou não.

Um perfil é uma matriz que contém informações estatísticas no que diz respeito às comparações realizadas em um determinado alinhamento de sequências. As colunas representam as posições nas sequências, e as linhas contêm a pontuação alcançada no alinhamento por posição em comparação com cada resíduo. Para entender melhor o que é um perfil em si, a tabela 1 apresenta uma associação de terminologias em estudos de genética comparativa de uma forma geral.

Tabela 1 - Terminologias básicas utilizadas em estudos de genética comparativa

Termo	Descrição
Domínio	Unidade estrutural de forma independente
Bloco	Alinhamento múltiplo de uma região conservada (sem inserção de lacunas) de sequências de proteínas
Padrão Conservado	Região altamente semelhante em um alinhamento de sequências de proteínas
Perfil	É a Matriz de Classificação de Posição Específica (MCPE) que é calculada a partir de cada bloco

Em HMM, de maneira mais específica, um perfil é utilizado como um modelo probabilístico que representa um conjunto ou grupo de sequências que são homólogas. Ele é chamado de perfil do Modelo Escondido de Markov (pHMM, do inglês *profile Hidden Markov Model*). Um perfil HMM é construído a partir de um alinhamento múltiplo de sequências com o objetivo de gerar um modelo (pHMM), e assim verificar o alinhamento através de uma comparação do próprio modelo gerado com outras sequências, a fim de encontrar sequências que sejam homólogas ao modelo. Essa etapa de comparação é chamada de Detecção de Homologia Remota.

Em pHMM, além dos estados normais já existentes em um HMM comum, são inseridos os estados de "Combinar", "Remover" e "Inserir". O estado Combinar identifica as regiões que obtiverem sucesso na comparação do alinhamento, ou seja, cada região em questão foi encontrada em todas as sequências do alinhamento. O estado de Inserir permite a inserção de um ou mais resíduos no alinhamento. O estado de Remover permite que resíduos desnecessários sejam removidos. A figura 3 ilustra um pHMM simples com todos os estados de transição descritos acima. O pHMM (à direita na figura) apresenta o que seria o modelo de um alinhamento múltiplo das quatro sequências à esquerda com três colunas cada uma. O estado Combinar é representado

pelo caractere “m”, enquanto os estados Remover e Inserir são representados pelos caracteres “d” e “i”, respectivamente.

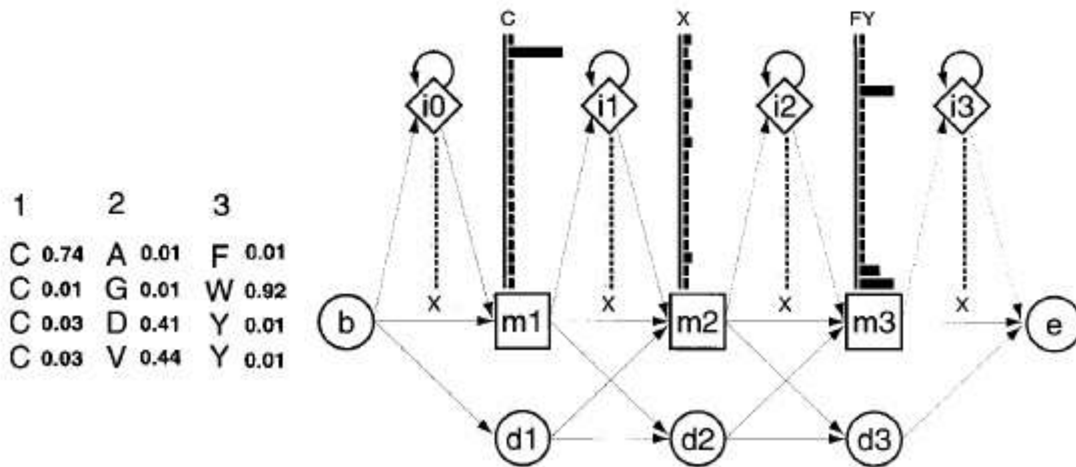

Figura 3 - Perfil do Modelo Escondido de Markov simples

O *software* HMMER se encontra hoje em sua terceira versão. Seus desenvolvedores afirmam que atualmente o HMMER é tão veloz quanto o BLAST (ALTSCHUL *et al.*, 1990) - umas das ferramentas mais famosas e usadas no estudo de sequências e em bioinformática -, principalmente com o uso de instruções de vetores para melhorar o desempenho de um dos principais algoritmos do HMMER (Viterbi), e, como consequência, aumentar o desempenho computacional e reduzir o tempo de processamento. De forma geral, o HMMER é bastante usado para identificar proteínas ou sequências de nucleotídeos que sejam homólogos. Todo esse processo de identificação é realizado através da comparação de um pHMM (onde cada pHMM é construído a partir de um Alinhamento Múltiplo de Sequências realizado antes desta etapa de comparação) com uma base de dados de sequências ou até mesmo com uma sequência simples. HMMER hoje é umas dos pacotes mais completos e de alto desempenho que existe na academia juntamente com o BLAST e vem produzindo resultados satisfatórios para cientistas e pesquisadores.

2.2 Map/Reduce

Map/Reduce é um modelo de programação funcional - popularizado primeiramente com o Google (DEAN, GHEMAWAT, 2004) - onde se é permitido manipular grandes massas de dados de forma paralela e distribuída através do uso de duas funções: *map* e *reduce*.

Este modelo é bastante antigo na literatura. Ele foi introduzido inicialmente em linguagens funcionais. A primeira a disponibilizá-lo foi a linguagem Lisp (na verdade por uma implementação específica chamada Common Lisp (STEELE, 1990)) em meados dos anos 1980. Mais tarde, Haskell (JONES, 2003) foi outra linguagem que também disponibilizou uma implementação do modelo no início dos anos 90. Porém, o modelo se popularizou bastante com a implementação da Google no fim dos anos 90 e início dos anos 2000. Esta implementação torna possível a execução de programas desenvolvidos segundo o modelo em ambientes paralelos de larga escala, heterogêneos, utilizando componentes computacionais não especializados e com tolerância automática a falhas.

2.2.1 O Modelo de Programação Map/Reduce em Linguagens Funcionais

O conceito do modelo de programação Map/Reduce foi proposto inicialmente na linguagem Lisp. Lisp itera sobre uma sequência de dados aplicando sobre cada um deles a função *map*. O resultado do processo é armazenado em uma nova sequência, que é tratada como um dado do tipo lista. A figura 4 ilustra uma lista simples em Lisp. As funções são anotadas em forma de prefixo, onde o tipo de operação a ser executada antecede uma ou várias listas. A figura 5 ilustra uma função simples em Lisp.

```
'(5 6 1 -3 -9 7)
'(a b c)
```

Figura 4 - Exemplo de uma lista em Lisp

```
('abs '(5 6 1 -3 -9 7)) => (5 6 1 3 9 7)
('cons '(a b c) '(5 7 4)) => ((a . 5) (b . 7) (c . 4))
```

Figura 5 - Exemplo de função em Lisp

A figura 6 ilustra o método de atuação das funções *map* em Lisp. A lista "L" sofre a alteração definida na função *map* "f" pelo usuário, e, como resultado, é produzida outra lista " L' ", que pode ser usada para outras finalidades ou para sofrer outro(s) processamento(s), por exemplo, sofrer alteração pela função *reduce*.

Figura 6 - Processamento das listas L pela função f em L'

A função *reduce* em Lisp combina ou reduz (do termo *reduce* em inglês) os resultados (armazenados em uma lista) provenientes das execuções do *map* e guarda o novo resultado em uma outra lista. A figura 7 ilustra algumas operações realizadas a partir de uma função *reduce* em Lisp. A figura 7 (a) apresenta a soma dos valores da lista. A figura 7 (b) ilustra a subtração da lista a partir do final da mesma, ou seja, da direita para a esquerda. As figuras 7 (c) e 7 (d) ilustram a concatenação dos caracteres

"abc" em suas respectivas listas. No primeiro caso, os caracteres são adicionados no início da lista, já no segundo eles adicionados no final.

(*reduce* #'+ '(5 9 3 2)) => 19 **(a)**

(*reduce* #'- '(5 9 3 2) *:from-end* t) => (-5 (-9 (- 3 2))) => -15 **(b)**

(*reduce* #'list '(5 9 3 2) *:initial-value* 'abc) => (((('abc 5) 9) 3) 2) **(c)**

(*reduce* #'+ '(5 9 3 2) *:from-end* t *:initial-value* 'abc) => (5 (9 (3 (2 abc)))) **(d)**

Figura 7 - Exemplo de uma função reduce em Lisp

Haskell possui uma abordagem um pouco diferente de Lisp. Ele é uma linguagem que trabalha através de programação funcional fortemente tipificada para a definição das funções de *map* e reduce. Esse tipo de linguagem torna um pouco mais fácil a definição e o entendimento das funções (JONES, 2003). Outra diferença é encontrada é com relação a função de *reduce*. Em Haskell ela é expressa pela função *foldl*. A figura 8 ilustra alguns exemplos de como funciona uma função em *map* e *foldl* em Haskell. Em ambos os casos apresentados na figura 8 foram realizadas operações simples, mas como pode ser visto produziram resultados diferentes. No primeiro caso, foi feita a multiplicação de todos os elementos por cinco, e como resultado foi gerada outra lista com os resultados das operações. No segundo caso, foram subtraídos os valores da lista, e como resultado foi gerado um resultado único - neste caso, um número - e não uma lista.

map ((*) 5) [3 1 4] → [15 5 20] (a)

fold (-) 0 [3 1 4] → -8 (b)

Figura 8 - Exemplo de uma função map e flodl em Haskell

Em Haskell, a definição de *map* e foldl é bastante diferente, assim também como a expressão de seus respectivos resultados. A figura 9 ilustra a definição de uma função *map* em Haskell. Como pode ser visto acima, uma função do tipo "a→b" que incide

sobre uma lista, gera outra lista como resultado. Caso a lista esteja vazia, irá gerar como resultado outra lista vazia, independente do tipo da função. Para entender melhor as funções que um *map* pode assumir - ou seja, “a→b” - na figura 8 (a), o argumento “a” seria o tipo de operação, neste caso “(*)”, e o argumento “b” seria o valor que irá incidir na lista junto o com o tipo de operação, neste caso, “5”. Ainda assim, a expressão “a→b” pode ser qualquer função definida pelo usuário (JONES, 2003).

map (f) [] → []

assumir que “f” seja uma função “a → b”, ou seja, “a” sobre “b”

map (a→b) [] → []

assumir uma lista inicial [l] e como resultado assumir uma lista final [l’]

map (a→b) [l] → [l’]

Figura 9 - Função map em Haskell

A definição de *foldl* é um pouco diferente, e toma como argumentos três itens e não dois como na função *map*. A figura 10 ilustra a definição da função. A função *foldl* como foi visto acima, faz uso apenas de operações binárias e não de uma função. O resultado da função *foldl* sempre será um valor combinado pela operação desejada. Na figura 8 (b) pode ser facilmente identificado cada um dois argumentos citados na definição. A operação binária - argumento “b” - corresponde a “(-)”, o valor fixo - argumento “a” - corresponde a “0”, e a lista está claramente expressa.

foldl (b) a [] → c

assumir uma lista inicial [l]

fold (b) a [l] → c

onde “b” é um valor de operação binária, “a” um valor fixo e “c” o resultado da operação sobre a lista

Figura 10 - Função *foldl*

2.2.2. O Modelo de Programação Map/Reduce da Google

O modelo proposto pela Google, no seu *framework* denominado MapReduce, utiliza vários princípios de Haskell e Lisp. A definição de *reduce* usada em Lisp serve de ponto de partida para a mesma função em MapReduce. Já com relação à função de *map*, MapReduce parte do princípio básico do uso dos três argumentos usado na mesma função em Haskell.

Outro diferencial encontrado em MapReduce é o fato de as funções *map* e *reduce* serem bem mais flexíveis do que as que são implementadas em Lisp e Haskell. A possibilidade do uso de funções definidas pelo usuário em todas as etapas (*map* e *reduce*) permite uma flexibilidade muito maior para a manipulação dos dados. Deste modo, os dados podem ter uma estrutura mais complexa e mesmo assim serem processados pelo modelo, pois o próprio usuário define a melhor estratégia para manipulá-los.

2.2.2.1 O *Framework* MapReduce

Com o grande aumento da quantidade de dados e suas respectivas taxas de crescimento, abordagens diferentes precisam ser usadas para processá-los em um tempo adequado às necessidades dos usuários. Um conceito antigo e bastante fundamentado na ciência é a técnica de "Dividir para conquistar", usado pela primeira no algoritmo de Karatsuba (KARATSUBA *et al.*, 1962). A ideia básica do algoritmo consiste em dividir um problema grande em vários problemas menores. Partindo do princípio de que essas tarefas menores são independentes (AMDAHL, 1967), elas poderiam ser processadas em paralelo dentro de um ambiente HPC por várias linhas de execução, núcleos de processadores, múltiplos processadores ou até mesmo em vários nós de um *cluster*.

MapReduce possui muitas vantagens para lidar com grandes volumes de dados. Seu nível de abstração permite que o usuário não se preocupe com nenhuma implementação em nível de sistema. Sendo assim, o usuário manter o foco no desenvolvimento de algoritmos para processamento do dado em si, não se preocupando com questões inerentes à implementação processamento paralelo em seu código, como comunicação entre processos e tolerância a falhas. A exemplo do OpenMP (CHAPMAN *et al.*, 2007) - que é uma interface de programação paralela com multi-processos em memória compartilhada - e da Interface de Passagem de Mensagem (MPI, do inglês *Message Passing Interface*) (WILLIAM *et al.*, 1994) - que é um padrão para comunicação de processos através de troca de mensagens em computação paralela -, MapReduce também fornece soluções prontas para a distribuição dos dados e dos recursos computacionais em um *cluster* ou grade computacional.

A distribuição e a gerência de grandes massas de dados, possui um custo muito alto na maioria das soluções que envolvem paralelismo de dados. O MapReduce porém, possui um grande diferencial em relação à maioria dessas soluções. A gerência de dados, assim como a distribuição dos mesmos, é feita por um sistema de distribuição de arquivos que trabalha juntamente com o *framework* MapReduce. O sistema de arquivos distribuídos trabalha com a estratégia de dividir os dados de entrada entre os discos locais dos nós do *cluster* envolvidos. O sistema de arquivos distribuídos será descrito mais adiante.

O modelo de programação trabalha basicamente com cinco conceitos simples; o primeiro deles é a fragmentação dos dados de entrada; o segundo é a criação de pares chave/valor a partir dos fragmentos separados dos dados de entrada; o terceiro é o agrupamento dos valores intermediários com uma mesma chave; o quarto é o processo

de iteração sobre os grupos de mesma chave gerados; o quinto é o processo de redução (agrupamento) de cada grupo.

O uso dos pares chave/valor são de suma importância para a estrutura de programação do MapReduce. Chaves e valores podem assumir diferentes tipos, desde os mais simples (como texto, número inteiro ou de ponto flutuante), e até os mais complexos (como matrizes associativas, listas ou tuplas). O MapReduce ainda oferece o suporte de personalização de tipos de dados. Logo, caso um problema não consiga mapear algum tipo de dado de seu experimento nos predefinidos, um novo tipo pode facilmente ser criado, desde que o mesmo esteja dentro dos padrões do paradigma de programação. Esse tipo de situação ocorre muito com dados não estruturados provenientes de estudos sobre Astronomia, Biologia, Física, entre outros.

Por fim, para processar todos os dados que foram submetidos ao *framework* usando o paradigma de programação Map/Reduce, duas funções são responsáveis pela manipulação/organização desses dados, as tradicionais *map* e *reduce*.

2.2.2.2 Mapeadores e Redutores

A associação entre as funções de *map* e *reduce* se dá através dos pares chave/valor. Ela segue conforme abaixo:

map (ch, v) → *lista* (ch’, v’)

reduce (ch’, lista(v’)) → *lista*(ch’, v’’)

A função *map* é chamada a cada par de chave/valor proveniente do processamento dos dados de entrada. O *map* gera como resultado pares intermediários de chave/valor, cujos valores são agrupados de acordo com a chave, ou seja, todos os valores que possuem a mesma chave são colocados no mesmo grupo, por uma operação intermediária entre as

funções *map e reduce*, implementada pelo próprio *framework*. Após esse grupamento, os pares gerados são enviados para a função *reduce*.

Uma instância da função *reduce* recebe uma chave juntamente com seu respectivo conjunto de valores. Ela agrega o conjunto de valores relativos à chave recebida. Como resultado desses novos grupamentos, são gerados novos pares chave/valor que serão escritos nos arquivos de saída. Os dados de saída são escritos novamente no sistema de arquivos distribuído para futuro acesso do usuário. A figura 11 ilustra todo esse processo entre os dados de entrada, *map*, *reduce* e os dados de saída.

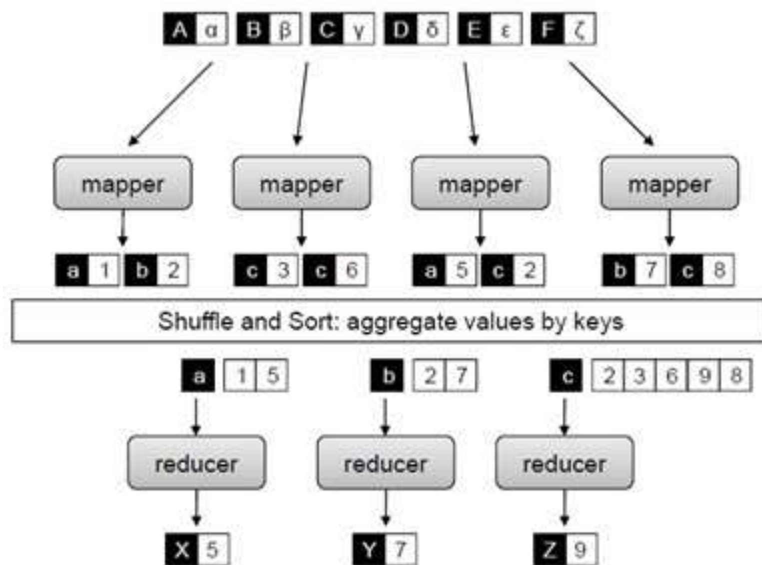

Figura 11 - Esquema de processamento com map e reduce (LIN *et al.*, 2010)

Durante o processo de execução paralela, vários Mapeadores e Redutores são criados. O número de Mapeadores e Redutores varia de acordo com a configuração do *framework*. Cada um deles executa respectivamente a mesma função map e reduce, mas sobre porções diferentes dos dados de entrada. Cada Mapeador pode executar uma ou mais funções *map* (uma de cada vez) em cada execução. Da mesma forma funciona o Redutor com relação as funções *reduce*.

Como dito anteriormente, vários problemas podem ser expressos, ou seja, resolvidos usando o paradigma de MapReduce. DEAN e GHEMAWAT (2004) apresentam alguns cenários que são perfeitamente modelados usando MapReduce.

2.2.2.3 Arquitetura do *Framework* MapReduce

Uma atividade dentro do *framework* é referenciada como uma tarefa. Uma tarefa consiste em um pacote com os códigos das funções *map* e *reduce*, todos os parâmetros necessários para a execução dos programas da tarefa em si e parâmetros como o caminho dos dados de entrada, que se encontram no sistema distribuído de arquivos, e o local de gravação dos dados de saída da tarefa (também no sistema distribuído de arquivos).

O usuário submete uma tarefa ao *framework* para ser executada a partir do nó principal - chamado de nó Mestre, que também é responsável por armazenar os metadados do *cluster* - que irá coordenar toda a execução da tarefa no *cluster*. O *framework* a partir desse momento toma toda a responsabilidade da execução. Com isso, ele inicia um grande processo de manutenção, recuperação e balanceamento do sistema durante a execução da tarefa. Dentre as responsabilidades do *framework*, estão as principais descritas a seguir.

Escalonamento

Em função da fragmentação dos dados de entrada, uma tarefa MapReduce também é dividida em unidades menores. O escalonamento - que é executado pelo nó Mestre - é o responsável por dividir a tarefa em unidades menores e alocá-las aos nós de processamento. Essas unidades são chamadas de tarefas de *map* e tarefas de *reduce*. Cada tarefa de *map* é responsável por um fragmento do dado, sendo assim uma tarefa pode ter centenas ou milhares de tarefas de *map* a serem executadas no *framework*. O *reduce* também possui unidades menores chamadas de tarefas de *reduce*. Cada tarefa de

reduce está relacionada a uma porção de chaves intermediárias geradas pelas tarefas de *map*.

Dependendo do número de tarefas (tanto de *map* como de *reduce*), o escalonador gerencia uma fila para a organização das mesmas. As tarefas que não estão em execução se encontram em estado de "espera". O escalonador precisa monitorar o progresso das tarefas que se encontram em execução, submetendo as que estão em "espera" conforme as demais vão sendo finalizadas. Além de "espera", ou outros estados possíveis de uma tarefa de *map* e uma tarefa de *reduce* são "em execução" e "concluída". Cada tarefa - seja ela de *map* ou *reduce* - também possui uma identificação dizendo qual o nó responsável pela sua execução (para as tarefas em estado "em execução").

Tolerância a falhas e Recuperação

Em ambientes típicos de execução de tarefas MapReduce há grandes quantidades de nós para processar massas de dados, uma premissa deve ser assumida: falhas de execução são muito comuns. A maneira de lidar e corrigir tais erros é o grande diferencial que as soluções devem possuir para obter sucesso no uso de arquiteturas como a do MapReduce.

A primeira estratégia para tolerância a falhas do MapReduce é a replicação dos dados. No processo de fragmentação, o *framework* envia mais de uma cópia dos fragmentos para serem armazenados em diferentes nós do *cluster*. Esse recurso ajuda o MapReduce em caso de falha de algum nó, pois dessa maneira ele não deixará de executar tarefas de *map* que precisam de um fragmento que esteja em um nó que falhou, pois as mesmas podem ser executadas em nós que contenham réplicas do fragmento. O MapReduce utiliza como padrão o fator replicação três. Porém ele pode ser configurado facilmente de acordo com as necessidades do usuário.

Localização dos dados

A localização dos dados é um fator muito importante para o bom funcionamento e desempenho do MapReduce. Em vista disso, foi necessário elaborar um esquema de armazenamento que reduzisse o custo de comunicação entre os nós a fim de manter a ideia de levar o código até os dados e não o contrário. Para solucionar esse problema, a Google elaborou um sistema de arquivos distribuídos chamado de Arquivos de Sistemas Google (GHEMAWAT *et al.*, 2003) (GFS, do inglês *Google File System*).

2.2.2.4 Sistema Distribuído de Arquivos

O Sistema de Arquivos Google (GFS) foi desenvolvido para prover alto desempenho, escalabilidade, confiabilidade e disponibilidade no armazenamento de dados. O GFS consiste em um *middleware* para o armazenamento de dados em uma grande quantidade de nós. Essas máquinas fazem parte do *cluster* MapReduce ou podem ser um *cluster* a parte. Na maioria das situações, o primeiro caso é o mais frequente. Assim sendo, os nós exercerão duas responsabilidades: a do *framework* MapReduce e a do GFS.

O GFS armazena os dados de entrada nos nós que estão no *cluster*. Ele fragmenta cada arquivo em blocos de 64MB. Os fragmentos são distribuídos e armazenados nos nós do *cluster*, e o nó Mestre armazena o endereço de todos os fragmentos. Com essas informações, o escalonador distribui as tarefas de *map* de acordo com a localização dos fragmentos. Como cada tarefa de *map* é responsável por um fragmento, o escalonador consegue enviar a tarefa de *map* ao nó que possui o fragmento necessário para sua execução. Isso evita a transferência de dados entre os nós durante a execução das tarefas de *map*.

Houve a necessidade de assumir o princípio de que independentemente da qualidade e da quantidade de nós, sempre vai haver máquinas que não estarão funcionando ou

algumas que conseguirão se recuperar de erros. O monitoramento constante, a detecção de erros, a tolerância a falhas e a recuperação automática são recursos essenciais e de suma importância em ambientes como esses e devem ser parte integrante do sistema.

Um fator importante neste tipo de arquitetura é a escrita dos dados de resutlado, uma vez que eles precisam ser escritos no *middleware* conforme forem gerados, podendo haver assim uma grande concorrência por conta da grande quantidade de nós envolvidas. Para ajudar no esquema de concorrência, o GFS implementou um sistema de operação chamado *append* atômico. O *append* atômico permite que vários nós, conforme forem finalizando suas execuções, possam escrever (*append*) no mesmo arquivo sem a necessidade de se ter um gerenciamento de fila/sincronismo entre os nós.

2.2.3 Hadoop

Criado pelos Laboratórios da Yahoo![3], e mais tarde se tornando um dos projetos mantidos pela Apache Software Foundation[4], Hadoop é um *framework* open-source que oferece recursos e suporte para aplicações distribuídas de processamento de dados intenso (WHITE, 2012) utilizando o paradigma de programação Map/Reduce. O *framework* foi baseado no MapReduce da Google. Assim como no MapReduce da Google, Hadoop também é capaz de processar grandes massas de dados (terabytes, petabytes) em um ambiente de HPC com centenas ou milhares de nós.

Dentre os subprojetos fundamentais do Hadoop estão o Sistema de arquivos distribuídos do Hadoop (SHVACHKO *et al.*, 2010) (HDFS, do inglês *Hadoop Distributed File System*), que é o sistema de arquivos distribuído feito para o Hadoop e similar ao GFS, e o *framework* Hadoop MapReduce, que é bastante similar ao

[3] http://labs.yahoo.com/
[4] http://hadoop.apache.org/

framework MapReduce da Google, e fornece recursos e fontes para o processamento dos dados paralelos em *cluster*s usando o paradigma de programação Map/Reduce.

Apesar de o Hadoop ser baseado no *framework* da Google, ele possui algumas diferenças com relação a ele. Nesta seção iremos explorar essas diferenças e em que elas implicam na estrutura/arquitetura e também no desempenho do *framework* Hadoop.

2.2.3.1 Mapeadores e Redutores

No Hadoop podem se encontrar algumas diferenças em relação ao *framework* da Google. Primeiramente as abordagens com relação às funções *map* e *reduce*. No Hadoop, a função *reduce* possui uma implementação um pouco diferente. Ela é apresentada com uma chave e um valor para iterar (*iterator*) sobre os dados, onde esse valor vai iterar sobre todos os valores de uma mesma chave. Neste caso, os valores são organizados e ordenados de forma arbitrária. Na implementação da Google o *framework* permite ao usuário especificar uma ordenação secundária (o que não é um requisito obrigatório) sobre a chave, a fim de apresentar os valores de forma ordenada para a função de *reduce*.

Outra diferença bastante interessante é com relação à manipulação das chaves na função *reduce*. No MapReduce não é possível mudar a chave durante o processamento da função *reduce*, ou seja, o valor da chave dos dados de saída deve ser o mesmo do valor da chave dos dados de entrada. No Hadoop não existe tal limitação. O usuário pode implementar uma função *reduce* em que possa produzir novas chaves e até mesmo novos pares chave/valor com diferentes chaves.

No MapReduce o número de *map tasks* está diretamente relacionado com o número de fragmentos gerados e que serão processados. Geralmente o número de fragmentos e o de tarefas de *map* são iguais. Salvo em situações em que foi necessário a reexecução

de alguma tarefa devido a alguma falha. Com isso, o usuário não possui controle do número de *map tasks*. No Hadoop, porém, isso não acontece. O usuário tem a possibilidade de configurar o número mínimo e/ou máximo de tarefas de *map*. Caso o número de fragmentos seja menor que o número mínimo exigido pelo usuário, os fragmentos são refeitos para que se tenha pelo menos o número mínimo configurado.

No caso da função *reduce* se apresenta o mesmo fato descrito acima. No MapReduce o número de tarefas i é diretamente relacionado com o número de chaves heterogêneas, não permitindo a intervenção do usuário com relação a isso. No Hadoop o usuário pode configurar o número exato de tarefas de *reduce* que serão usadas para processar os pares chave/valor intermediários gerados pela função *map*.

2.2.3.2 Arquitetura

Hadoop possui uma arquitetura bastante similar ao MapReduce *framework*. As estratégias os mecanismos de *scheduling*, localização de dados, e tolerância a falhas são praticamente os mesmos. Uma diferença encontrada no Hadoop é no processo de transição das chaves intermediárias e dos seus respectivos conjuntos de valores - ambos gerados a partir do processamento da função *map* - para o *reduce*. No MapReduce, a função *reduce* precisa esperar todas as funções *map* sejam finalizadas para ter a garantia de que todos os pares intermediários foram gerados e ordenados corretamente para ai sim manipulá-los todos de uma vez. No Hadoop a abordagem usada é de que os valores podem ser facilmente enviados conforme forem sendo gerados, sendo assim, conforme as execuções das tarefas de *map* forem sendo finalizadas, elas podem enviar seus pares chave/valor intermediários para os nós que estão executando as tarefas de *reduce* (a única diferença aqui é que não será possível nem ordenar as chaves e nem os diferentes valores de uma mesma chave). Assim sendo, os nós que irão executar as funções *reduce*

não ficarão ociosos aguardando a finalização completa das tarefas de *map* como é feito no MapReduce. Como resultado final desta modificação, foi uma boa melhora do desempenho do tempo de execução da tarefa no *framework* como um todo.

2.2.3.3 Sistema de Arquivos Distribuídos do Hadoop

O Hadoop utiliza seu próprio esquema de armazenamento de arquivos chamado Sistema de arquivos distribuídos do Hadoop (HDFS, do inglês *Hadoop Distributed File System*). Assim como o *framework* Hadoop, ele também foi construído pelos laboratórios do Yahoo!, é mantido hoje pela Apache Software Foudation. O HDFS é uma implementação open-source do GFS e que dá suporte ao armazenamento de dados para o Hadoop.

O HDFS é bastante similar ao GFS. Ele possui essencialmente as mesmas atribuições e provêm das mesmas funções que o GFS apresenta, mudando apenas as terminologias. Seu sistema de distribuição e armazenamento de arquivos continua o mesmo, assim como os recursos de replicação e de fragmentação dos dados.

2.2.3.4 Hadoop Streaming

Hadoop *Streaming* (WHITE, 2012) é um utilitário fornecido pelo próprio *framework* Hadoop. O Hadoop *Streaming* tem como objetivo possibilitar a escrita das funções *map* e *reduce* em outras linguagens que não o Java. Para isso, o Hadoop *Streaming* utiliza os padrões do fluxo (entrada e saída) do Unix para a comunicação do Hadoop com as funções. Com isso, qualquer linguagem que faça uso de leituras de entradas e escritas das saídas padrões do Unix podem ser usadas no Hadoop através do *Streaming*. Por natureza o *Streaming* foi desenvolvido para processamento de texto - porém em versões mais recentes ele também pode ser usado para processar dados binários -, onde os dados são recebido/lidos linha a linha pela entrada padrão e processadas pelo *map* uma a uma.

A escrita na saída padrão também é feita linha a linha pelas funções *map* e *reduce*. Cada linha possui um par chave/valor separados por um delimitador. No caso do reduce, ele também faz a leitura através da entrada padrão e recebe os dados de forma ordenada por *key* (WHITE, 2012), e também realiza as escritas na saída padrão.

Uma pequena desvantagem em relação ao Hadoop Streaming é que o seu uso gera uma pequena sobrecarga no processamento, com isso o desempenho do *framework* como um todo fica um pouco penalizado e onerando assim o tempo de processamento (DING *et al.*, 2011). Devido a essa situação, o seu uso sempre deve vir acompanhado de um bom estudo do problema e de uma boa elaboração para a solução do mesmo.

2.3 Uso de processamento paralelo em Bioinformática

Várias pesquisas já desenvolvidas mostram, com excelentes resultados, que é possível usar técnicas de paralelismo junto a aplicações de Bioinformática. Muitas delas são dedicadas a melhorias de desempenho, visando principalmente à redução do tempo de processamento.

Com o crescimento da massa de dados, o uso de ambientes HPC vem se tornando um caminho natural em pesquisas de Bioinformática para resolver problemas de desempenho. A utilização do paradigma de linguagem Map/Reduce também vem ganhando espaço nos últimos anos como uma boa alternativa de desempenho em Bioinformática. Soluções como o CloudBLAST (MATSUNAGA *et al.*, 2008.) é um exemplo bastante similar ao que foi realizado neste trabalho e que utiliza ambas as abordagens acima.

2.3.1 CloudBLAST

CloudBLAST (MATSUNAGA *et al.*, 2008.) é um *middleware* para execução paralela da ferramenta de alinhamento BLAST em *clusters* e nuvens computacionais. O

BLAST é utilizado para alinhamento de conjuntos de sequências de proteínas ou de nucleotídeos a fim de detectar similaridades. Para entender melhor, essas comparações são feitas a partir de um conjunto de sequências de proteínas "S" (por exemplo, S = {s1, s2,... sn}), com um banco de dados "D" com k sequências (por exemplo, D = {d1, d2,..., dk}), e partir deste processo gerados dados estatísticos de acordo com os acertos produzidos, validando suas qualidades e relevância para o experimento.

Para se utilizar o BLAST em um ambiente distribuído e paralelo, duas abordagens podem ser adotadas: a primeira é fragmentar as sequências de entrada, onde cada fragmento de "S" é processado em um nó diferente; a segunda é fragmentar tanto as sequências de entrada como o banco de dados "D". A segunda abordagem necessita de uma alta velocidade na troca de mensagens entre os nós, para que ao final do processo seja gerado um único resultado com a junção de cada saída de cada nó. Um fator importante a ser considerado nessa abordagem é que os dados estatísticos resultantes das comparações entre as sequências são extremamente dependentes dos tamanhos das mesmas e do banco de dados, sendo assim, particionar tanto o banco de dados como as sequências de entrada poderia claramente afetar os resultados.

A primeira abordagem (fragmentar apenas as sequências de entrada) que é utilizada pelo CloudBLAST, no entanto, pode ser facilmente executada em paralelo, exigindo um simples *script* para fragmentar "S", executar o BLAST sobre cada fragmento e combinar todos os resultados. Esta abordagem pode ser implementada com o uso do *framework* Hadoop.

CloudBLAST utiliza o Hadoop para gerenciar a execução em paralelo do BLAST. A função *map* é responsável pela execução do BLAST em cada nó. Cada fragmento é processado por seu respectivo nó através da função *map*. O banco de dados "D" é replicado em cada nó para que cada fragmento possa ser comparado com o banco de

dado completo e não fragmentado, garantindo assim uma execução mais confiável e mais apurada com relação aos resultados.

Para garantir uma boa distribuição das sequências de entrada entre os nós, CloudBLAST também utiliza o HDFS do Hadoop. Por padrão o HDFS gera partições dos arquivos em tamanhos iguais de 64MB, sendo assim, como a divisão é feita levando em consideração apenas o tamanho (em bytes) do arquivo, se teriam fragmentos com sequências incompletas, gerando assim um dado inconsistente. Para resolver este problema, CloudBLAST desenvolveu seu próprio fragmentador de sequências FASTA no Hadoop através das bibliotecas que o mesmo disponibiliza e permite a modificação para as necessidades do usuário. Desta maneira, foi possível garantir que durante o processo de fragmentação um fragmento nunca teria uma sequência incompleta.

Os resultados obtidos com o CloudBLAST mostram ganho em comparação com execuções sequenciais e com soluções que utilizam outras técnicas para execução paralela do BLAST, como o mpiBLAST. Com este último, por exemplo, o CloudBLAST conseguiu uma aceleração de 57 contra 52.4 do mpiBLAST em relação à execução sequencial, quando ambos são executados em um ambiente com 64 processadores.

Capítulo 3 - Alto Desempenho em Bioinformática: um Estudo de Caso

SciHmm é um *workflow* desenvolvido por OCAÑA *et al.* (2013) com o objetivo de auxiliar o processo de escolha de métodos de MSA para filogenia. Trata-se de um *workflow* científico de alto desempenho baseado em arquitetura de computação em nuvem e também em perfis de Modelos Escondidos de Markov (pHMM, do inglês *profile Hidden Markov Model*). O SciHmm foi desenvolvido para facilitar a análise e a comparação entre diferentes métodos de MSA. Durante sua execução, cada um dos métodos utilizados é avaliado e classificado de acordo com os seus respectivos resultados, a fim de determinar o mais adequado para o experimento de filogenia do pesquisador que utiliza o SciHmm.

Com o grande aumento das massas de dados na área de Biologia, principalmente devido ao sucesso de novas descobertas do código genético, as análises de filogenia têm se tornando atividades extremamente demoradas. Filogenia é o estudo da relação evolutiva entre grupos de organismos. Essa relação é descoberta através de sequenciamento de dados moleculares e também com o uso de matrizes de dados morfológicos. Como resultado dessas análises, é gerada a história evolutiva (ou seja, a filogenia) dos grupos de organismos envolvidos, criando assim as chamadas árvores filogenéticas. O desempenho das análises de filogenia está diretamente relacionado ao uso do alinhamento múltiplo de sequências, já que é necessário fazer dezenas, centenas ou até milhares de alinhamentos para conseguir construir as árvores filogenéticas.

Os estudos de genômica comparativa, que analisam relações de estrutura e funções entre os genomas de diferentes espécies, como Alinhamento Múltiplo de Sequências (MSA, do inglês *Multiple Sequence Alignment*), detecção de homologia remota e

filogenia, vêm sendo influenciados diretamente pelo aumento constante das massas de dados com o decorrer dos anos. Esse fato tem aumentado a complexidade maior destas atividades. Com isso, elas têm demandado ambientes com capacidade computacional cada vez maior a fim de processarem grandes volumes de dados em um espaço de tempo aceitável.

Um dos campos mais importantes nos estudos de genômica comparativa é o MSA. Existem hoje diferentes métodos e abordagens para realizá-lo. Cada uma dessas abordagens pode gerar diferentes resultados. Com isso, ao serem usados como, por exemplo, em análises filogenéticas, diferentes árvores filogenéticas podem ser geradas ao final do processo. A partir dessa visão, é de suma importância para o cientista saber qual o melhor método a ser aplicado para realizar um MSA a fim de se ter a melhor análise filogenética possível. O único modo de se escolher o melhor método é realizar os devidos testes com todos os métodos de MSA, validar e comparar cada um deles, escolhendo assim o de melhor resultado. Essa abordagem é importante pelo fato de o uso maciço de MSA estar diretamente relacionado com análises filogenéticas, pois influencia fortemente a construção das árvores (OCAÑA *et al.*, 2013). Atividades de MSA são extremamente custosas, podendo demorar várias horas ou dias para serem finalizadas, dependendo da massa de dados. Em vista de todos os problemas citados acima, OCAÑA *et al.* (2013) desenvolveram o *workflow* SciHmm.

O *workflow* SciHmm é constituído por três etapas principais. A primeira é a construção dos MSA, a segunda é a construção dos pHMMs, e a terceira é a comparação desses modelos com uma base de dados de sequências local. O objetivo principal do *workflow* é fazer uma validação da sensibilidade de cada um dos métodos de MSA utilizados para a detecção de sequências homólogas.

O SciHmm compara cinco ferramentas para MSA: MAFFT (KATOH *et al.*, 2008), Muscle (EDGAR, 2004), Probcons (DO *et al.*, 2005), Kalign (LASSMANN *et al.*, 2005) e ClustalW (THOMPSON *et al.*, 1994). Cada uma delas oferece uma solução diferente para alinhamento múltiplo de sequências. MAFFT oferece três tipos de abordagens para MSA: método progressivo, refinamento iterativo e refinamento iterativo com auxílio do uso de contagens (*scores*) consistentes; cada um dessas abordagens pode gerar resultados diferentes no que diz respeito ao desempenho e à qualidade. Muscle utiliza o algoritmo de estimativa rápida de distância juntamente com o auxílio do contador *k*mer. ProbCons é um algoritmo que realiza o alinhamento progressivo através do par baseado em Modelo Escondido de Markov, onde, ao invés de fazer uso do algoritmo de alinhamento Viterbi, utiliza precisão máxima esperada e transformação de consistência probabilística para o alinhamento múltiplo (DO *et al.*, 2005). Kalign utiliza o algoritmo correspondência textual Wu-Manber para garantir suas métricas de qualidade de precisão e velocidade. O ClustalW é uma das ferramentas de MSA mais utilizadas em Bioinformática. Para melhorar seu desempenho e sua sensibilidade, o ClustalW vem sofrendo diversas melhorias em seu código e estruturas de dados. Atualmente, possui dois métodos para realização de MSA. O primeiro é o esquema de pontuação de posição específica e o segundo é um sistema de baixa ponderação para grupos de sequências representadas. Cada uma dessas abordagens é explorada pelo *workflow* SciHmm, e seus respectivos resultados são analisados de forma comparativa entre si, a fim de se obter o melhor resultado possível para o seguimento do *workflow* e principalmente para a construção das árvores filogenéticas.

A especificação detalhada do *workflow* é mostrada pela figura 12. Como pode ser observado, o *workflow* é composto por cinco atividades, são eles: (1) construção dos MSA, (2) construção dos pHMM, (3) comparação dos pHMM com um banco de dados

de sequências local, (4) análise de validação cruzada, e (5) geração das curvas de Características Operacionais do Receptor (ROC, do inglês *Receiver-Operating Characteristic*).

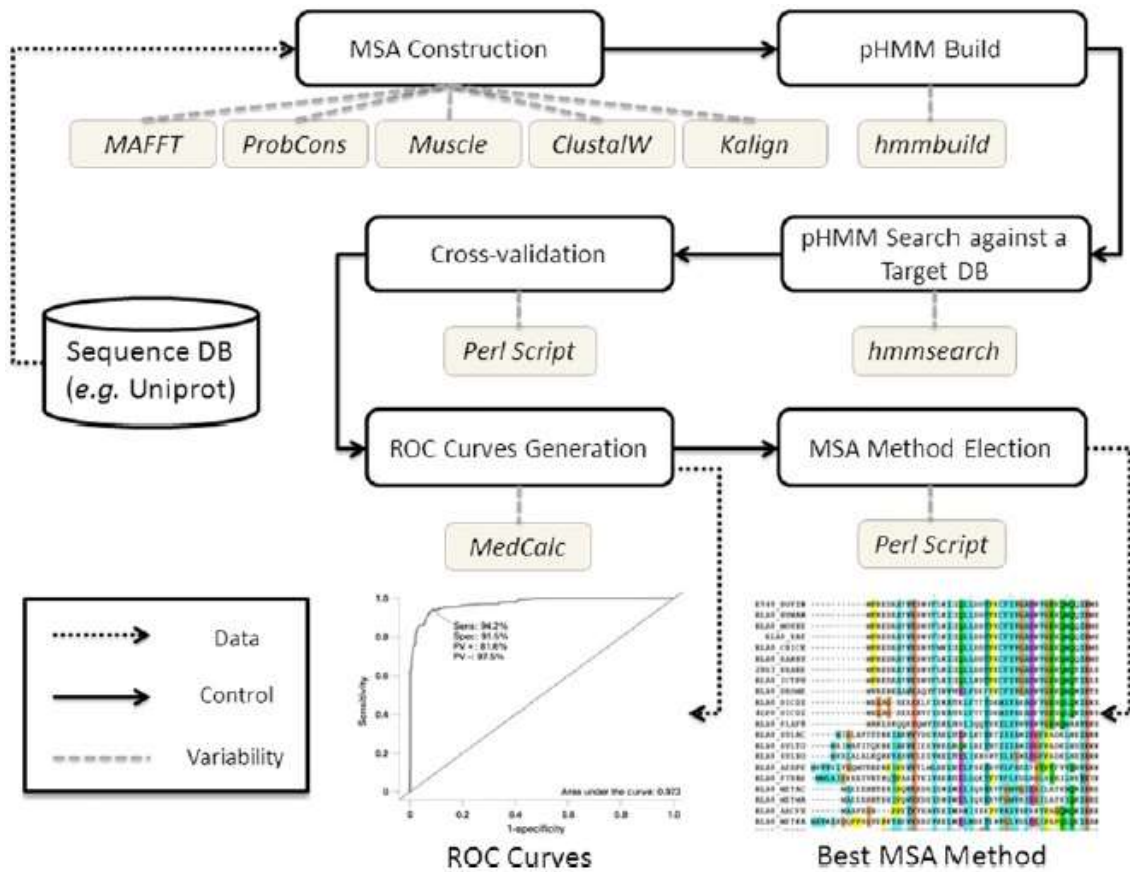

Figura 12 - Estrutura das atividades do workflow SciHmm (OCAÑA *et al*., 2013)

A primeira atividade corresponde à construção do MSA. Como entrada de dados, são utilizados vários arquivos de sequências em formato FASTA. Como dito anteriormente, o SciHmm emprega cinco ferramentas de MSA. Os MSA são gerados a partir dos arquivos de entrada por cada uma das cinco ferramentas, ou seja, cada ferramenta irá gerar um MSA diferente. Cada arquivo FASTA causa a geração de um MSA. O SciHmm executa esta etapa em paralelo, ou seja, os cinco métodos são executados simultaneamente em diferentes nós da nuvem. Além disso, como cada arquivo FASTA é processado de maneira independente dos demais, o SciCumulus executa paralelamente uma instância de cada ferramenta para cada arquivo de entrada,

gerando um grau de paralelismo bastante alto. Contudo, a execução individual de cada ferramenta é feita de forma sequencial. A atividade é considerada concluída apenas quando todas as ferramentas finalizam suas respectivas execuções.

A segunda atividade é a construção dos pHMM. Ela recebe os dados de saída gerados por cada um dos métodos de MSA. O software *hmmbuild* - que faz parte do pacote HMMER - é o responsável por gerar perfis HMM (pHMM) a partir de alinhamentos múltiplos. Sendo assim, o *hmmbuild* gera um pHMM para cada um dos MSA gerado pela atividade anterior. Esta atividade também é executada em paralelo, porém não da mesma forma que a primeira. Apenas cinco instâncias do *hmmbuild* são executadas em paralelo, cada uma destinada a processar os MSA gerados por uma das ferramentas empregadas na fase anterior. A execução individual do *hmmbuild* é sequencial.

A terceira atividade é a execução do *hmmsearch*. O *hmmsearch* - que faz parte do pacote HMMER - é o responsável por realizar a detecção de homologia remota. Ele realiza a leitura de cada modelo (pHMM) e busca por similaridades relevantes no conjunto de sequências alvo (neste caso, um banco de sequências de proteínas). O banco de sequências de proteínas utilizado como alvo das buscas do *hmmserach* é composto por vários arquivos FASTA obtidos a partir do banco de sequências Uniprot[5]. O *hmmsearch* toma cada modelo gerado pelo *workflow* e o compara com todos os arquivos FASTA do banco de sequências de proteínas. A comparação de um modelo com um arquivo FASTA é feita de forma sequencial pelo *workflow*. A busca realizada pelo *hmmsearch* gera como resultado um conjunto de "acertos", que são sequências ou partes de sequências (também chamados de domínio) que foram encontradas no banco de sequências de proteínas. Cada "acerto" é acompanhado por vários resultados

[5] http://www.uniprot.org/

estatístico oriundos dos cálculos efetuados durante a execução da ferramenta, onde o mais importante deles é o *E-value*. O *E-value* é uma estimativa de erro para o resultado, ou seja, ele representa a relevância do "acerto" para a sequência em questão considerando o banco de sequências de proteínas. Quanto menor o *E-value*, maior a relevância do "acerto". O paralelismo implementado para a execução do *hmmsearch* é o mesmo empregado para o *hmmbuild*: uma execução para cada uma das atividades de alinhamento executada pelo *workflow*. Sendo assim, há, no máximo, cinco instâncias do *hmmsearch* em execução em paralelo. Cada uma é responsável por todos os pHMM gerados a partir dos MSA de uma ferramenta de alinhamento. A atividade de detecção de homologia remota efetuada pelo *workflow* é bastante custosa. Isso leva a uma grande demora no processo de execução desta atividade, devido principalmente ao fato de haver uma grande quantidade tanto de modelos a serem comparados como de arquivos FASTA, estando o desempenho da busca ser diretamente ligado ao tamanho de ambos os arquivos. O objetivo desta dissertação é implementar uma estratégia para acelerar a execução desta fase do *workflow*, beneficiando-se do ambiente paralelo de execução.

A quarta atividade do *workflow* é a validação cruzada, que confirma - ou não - os "acertos" encontrados na etapa anterior. Seu objetivo é identificar os "acertos" que são verdadeiros positivos, ou seja, os que realmente são válidos e verdadeiros, e os falsos positivos, ou seja, os que são relatados como verdadeiros, mas na verdade não são. Com isso, é possível se chegar a uma conclusão sobre qual modelo de alinhamento é o melhor para o experimento de filogenia.

A quinta atividade é a geração das curvas ROC. Elas são usadas como complemento da validação dos resultados da validação cruzada.

O objetivo desta dissertação é acelerar a execução da terceira etapa do *workflow*, que demanda grande tempo de processamento. As estratégias adotadas para este fim são descritas no próximo capítulo.

Capítulo 4 - Paralelização da Busca de Similaridades no SciHmm

Como descrito no capítulo anterior, a terceira atividade do *workflow* SciHmm é uma das que oneram bastante o seu tempo total de execução. Em vista disto, esta dissertação destina-se à melhoria desta atividade específica, tentando reduzir o seu tempo de execução. A estratégia utilizada é não-intrusiva, permitindo que o mesmo *software* de busca de similaridades utilizado em um ambiente sequencial possa ser utilizado em um ambiente paralelo. Tal característica facilita a escolha de ferramentas por parte do usuário do workflow, uma vez que não são necessárias versões desenvolvidas especialmente para execução com paralelismo.

As seções a seguir descrevem a estratégia de paralelização adotada bem como os detalhes de implementação.

4.1 Estratégia de Paralelização

A estratégia de paralelização adotada é apresentada na figura 13. Nela, pode-se notar que a terceira atividade do *workflow* é implementada através de uma série de subatividades, que, em alguns casos, podem ser executadas em paralelo. A paralelização não-intrusiva é conseguida através do uso do *framework* Hadoop.

A execução da detecção de homologia remota com o *hmmsearch* se dá através da comparação de cada arquivo do conjunto de modelos (pHMM) com cada arquivo do banco de sequências de proteínas, que são os arquivos FASTA da base de dados do Uniprot.

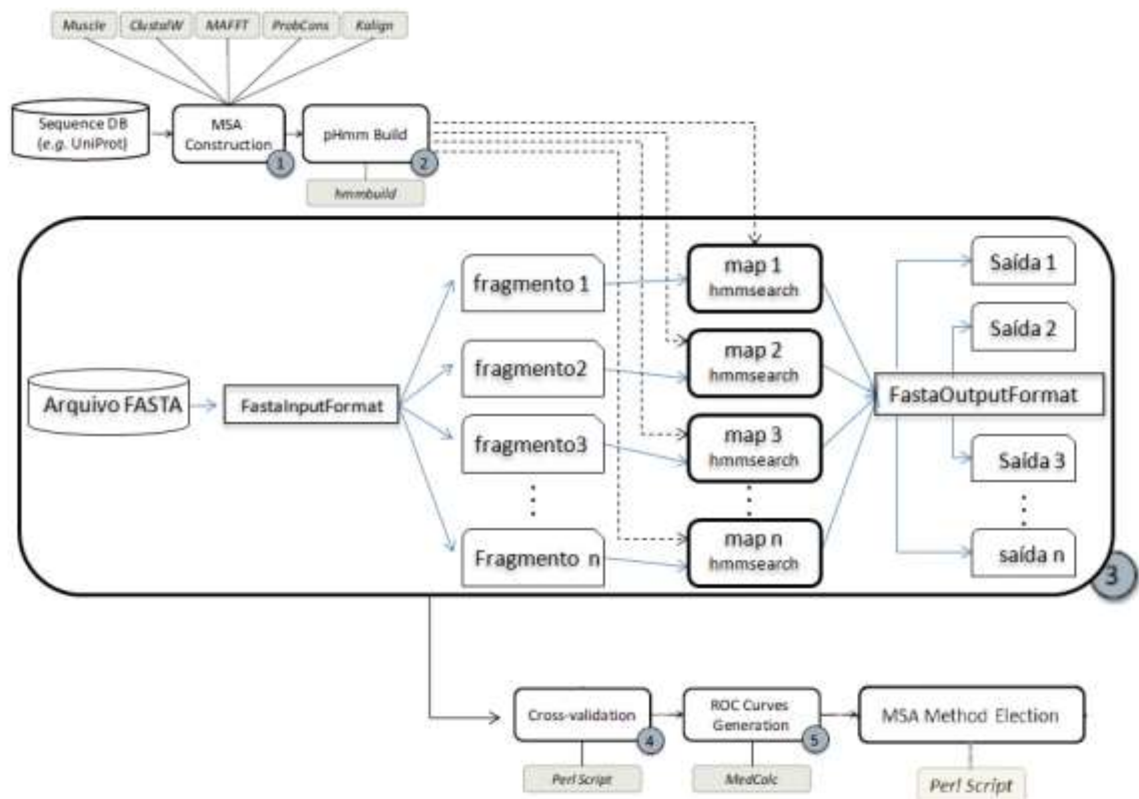

Figura 13 - Esquema de paralelização do workflow SciHmm

Para realizar a paralelização da busca dos modelos pHMM no banco de sequências de proteínas, foram propostas duas estratégias. A mais simples consiste em passar diretamente para o Hadoop o conjunto de modelos (pHMM) e os arquivos FASTA. Os modelos são replicados em todos os nós do *cluster* (já que eles não serão fragmentados), enquanto os arquivos FASTA são utilizados como entrada de dados para a tarefa do Hadoop. Essa estratégia deixa sob responsabilidade do Hadoop a fragmentação e a distribuição dos arquivos FASTA.

Geralmente, o SciHmm não faz uso de arquivos extensos (*terabytes* ou *petabytes* de tamanho) em pequena quantidade. Ao contrário, na maioria das vezes, ele utiliza muitos arquivos relativamente pequenos (se comparados a arquivos extensos como os citados acima). O Hadoop não realiza nenhum tipo de análise prévia com relação ao tamanho dos arquivos de entrada, tratando cada um deles de forma independente. Dependendo do

experimento, um arquivo de entrada pode não preencher por completo toda uma partição padrão do HDFS, pelo fato de ser menor que o tamanho padrão adotado por ele (64MB). Com isso, como há vários arquivos FASTA de entrada (o HDFS trata cada um deles de forma individual), durante a execução da tarefa do Hadoop poderá haver tarefas *map* com partições completas (com os 64MB preenchidos) ou não. O primeiro caso é uma situação ideal, pois não há desperdício de espaço na partição. Já no segundo caso, não só há desperdício de espaço, como há também a possibilidade de desperdício de recursos, pois, como a tarefa *map* estará manipulando uma porção pequena de dados, ela tem grandes chances de terminar sua execução antes das tarefas *map* que estão manipulando partições completas. Caso não haja outras partições a serem processadas, o nó que estava executando esta tarefa *map* ficará ocioso, gerando desbalanceamento de carga entre os nós.

Outro problema relacionado ao tamanho aos arquivos de entrada está no fato de que, como o HDFS armazena cada arquivo separadamente, caso o número de arquivos ou o número de partições gerado em função dos arquivos não seja pelo menos igual ao número de nós do *cluster*, haverá também subutilização de recursos em função da não utilização de todos os nós disponíveis.

Em vista dos problemas relatados acima, existe uma grande probabilidade de ocorrer desbalanceamento de carga durante a execução da tarefa. Desta forma, a primeira estratégia apresenta um processo que consiste em combinar todos os arquivos FASTA de entrada em um arquivo unificado, e então fragmentá-los através de uma estratégia personalizada que leva em consideração sua estrutura, não deixando este processo a cargo do Hadoop. A concatenação dos arquivos de entrada e a estratégia de fragmentação buscam distribuir de forma igual a carga de trabalho entre todas as tarefas de *map* a serem executadas pelos nós do *cluster*.

A estratégia de fragmentação personalizada se baseia na quantidade de fragmentos em que os dados de entrada devem ser divididos. O tamanho total do arquivo é dividido pela quantidade de fragmentos desejada. O número de fragmentos nunca pode ser menor do que o número de nós do *cluster*. Com isso, teremos todos os fragmentos com aproximadamente o mesmo tamanho em *bytes* (a quantidade de sequências dentro de um determinado fragmento pode variar de acordo com tamanhos das sequências). Logo, a chance de se obter balanceamento de carga será muito maior do que se for utilizada a forma automática de fragmentação implementada pelo Hadoop.

Essa estratégia busca a melhor configuração e ajuste na geração dos fragmentos a fim de se ter o melhor aproveitamento possível dos recursos disponíveis no *cluster*, independente do número de nós. Esses fragmentos são então passados para os nós do *cluster* Hadoop para serem processados em paralelo pelas funções *map* do *framework*. Cada função *map* irá tomar cada um dos modelos (os arquivos pHMM provenientes da execução da atividade anterior) para ser comparado com seu respectivo fragmento FASTA. Para isso, cada tarefa *map* executa uma instância do *hmmsearch* (figura 13). Este trabalho tem como princípio fragmentar e distribuir o banco de sequências para a realização do processamento paralelo.

A segunda estratégia utiliza justamente a abordagem mais simples descrita anteriormente. Ela faz uso da fragmentação padrão do *framework* Hadoop, ou seja, daquela produzida a partir da fragmentação automática realizada pelo HDFS. Nenhum processo personalizado de fragmentação é utilizado nesta abordagem. O processo padrão de fragmentação se baseia apenas no tamanho do arquivo que foi enviado previamente ao HDFS, onde, a cada 64MB, é criada uma nova partição.

Ambas as estratégias não utilizam a função *reduce*. Sua não utilização é justificada mais adiante. Além disso, foi criado também um processo personalizado de gravação

dos dados de saída proveniente dos resultados das funções *map*. O Hadoop não é flexível quando se trata de criação e gravação dos arquivos de saída, não permitindo nenhuma alteração nessa configuração quando se utiliza o seu processo padrão. Porém, com a necessidade de se criar vários arquivos diferentes a fim de organizar os dados de saída, foi construído um processo personalizado para sua gravação. Esse processo se baseia em uma tabela de dispersão criada a partir das chaves produzidas pelos *maps*.

A tabela de dispersão utilizada é parte do processo personalizado de gravação dos arquivos de saída. O processo mapeia os arquivos a serem gravados no HDFS através das chaves geradas pelos *maps*. Através das chaves, ele cria uma tabela de dispersão que permite que todos os valores com a mesma chave sejam mapeado para o mesmo arquivo a ser gravado. Desta forma, sempre que um par chave/valor a ser gravado em algum arquivo no HDFS é produzido, a chave é recolhida e passada para a tabela de dispersão. Quando a mesma identifica o ponteiro relativo ao arquivo da chave, ela repassa o endereço correto do arquivo para a gravação do valor.

Em se tratando de uma comparação com a abordagem adotada pelo SciCumulus para a paralelização da terceira etapa do *workflow* SciHmm, o SciCumulus possui um grau de paralelismo baixo, pois, neste caso, ele gera apenas cinco atividades para serem executadas em paralelo. Cada uma destas atividades corresponde a cada conjunto de modelos pHMM gerado em função das ferramentas de MSA. Cada atividades é responsável pela comparação do seu respectivo conjunto de modelos com todos os arquivos FASTA do banco de dados de sequências. Desta forma, o grau de paralelismo do SciCumulus fica restrito ao número de ferramentas de MSA utilizado pelo SciHmm. Outra diferença em relação ao SciCumulus é que o mesmo não faz uso de um sistema distribuídos de arquivos, enquanto o Hadoop utiliza o HDFS. O uso do HDFS, facilita a

fragmentação e a localização dos dados dentro do *cluster*, aumentando assim a eficácia do paralelismo implementado pelo Hadoop.

4.2 Detalhes de Implementação

Este trabalho utiliza o Apache Hadoop para executar tarefas (*jobs*) em paralelo em um *cluster* de computadores com memória distribuída. O Hadoop implementa o sistema de arquivos distribuído HDFS utilizando os discos rígidos dos nós do *cluster*. Ao ser armazenado no HDFS, um arquivo é automaticamente dividido em partições (*chunks*) de tamanhos iguais e estas são distribuídas entre os nós. Caso o tamanho do arquivo seja menor do que o configurado para uma partição, ele não é dividido.

Assim como o CloudBLAST, a estratégia proposta por esta dissertação usa fragmentação de dados para implementação de paralelismo. Os dados de entrada usados no SciHmm são sequências contidas em vários arquivos no formato FASTA, cada um correspondendo a uma espécie biológica diferente. No experimento, foi permitido que o Hadoop utilizasse a criação de partições a partir da gravação dos arquivos no HDFS. Como o HDFS efetua a partição baseado apenas no tamanho dos arquivos, foi necessário que o processo de fragmentação realizasse ajustes específicos para corrigir inconsistências de dados que porventura existissem em função da partição do HDFS, evitando assim que se fragilize a execução do processo. O Hadoop trabalha com classes que implementam as interfaces *InputFormat* e *RecordReader* para ler, fragmentar e distribuir os dados de entrada (uma vez que já se encontram armazenados no HDFS) que serão processados pelas funções *map*. O usuário pode facilmente implementar essas interfaces a fim de criar classes que se comportem de acordo com suas necessidades.

A figura 14 ilustra o que ocorre com os fragmentos dos arquivos FASTA quando não se realiza os ajustes necessários no momento de se submeter os mesmos para a uma

tarefa. Isso ocorre porque pois as classes padrões do Hadoop que implementam a *InputFormat* apenas executam a leitura das partições previamente criadas pelo HDFS e as repassam para os *maps*. Dessa maneira, ele não respeita os limites (de início e fim) das sequências, podendo, com isso, gerar fragmentos inconsistentes. As partições das sequências “xxxxxx” e “yyyyyy” na figura 14 foram geradas de forma incorreta pelo processo padrão do Hadoop. Durante o processo de execução do *workflow* essa inconsistência vai gerar resultados errados e acarretar erros de execução da tarefa. Tal problema foi observado também na implementação do CloudBLAST. A fim de tornar possível a leitura sem erros de arquivos FASTA armazenados no HDFS, foram desenvolvidas as classes *FastaInputFormat* e *FastRecordReader*.

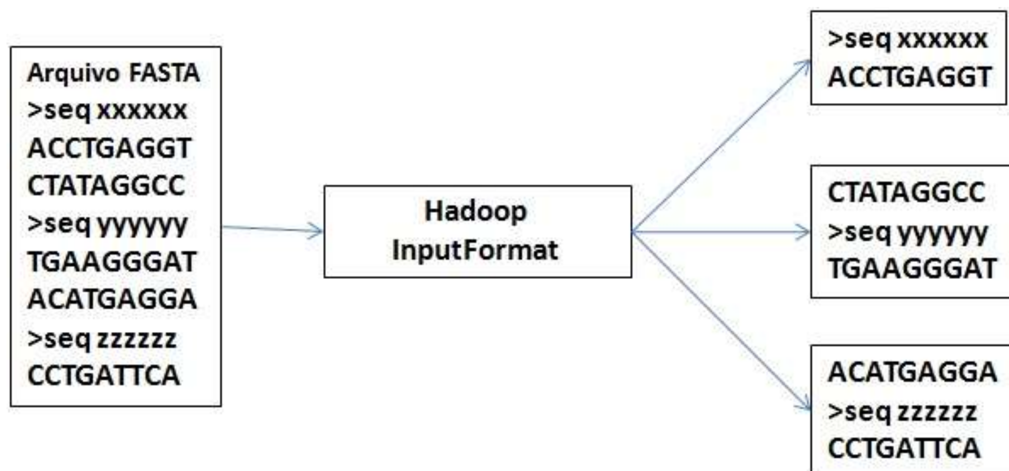

Figura 14 - Processo de fragmentação do arquivo pelo InputFormat do Hadoop

São descritas a seguir as classes *FastaInputFormat* e *FastRecordReader*, assim como a função *map*, a fase de não utilização da função *reduce* e a classe personalizada de saída *MultiFastaOutputFormat*.

4.2.1 FastaInputFormat

A classe *FastaInputFormat* foi desenvolvida para corrigir os possíveis erros de consistência dos dados ocasionados por leituras dos arquivos de entrada armazenados no HDFS. Através dessa classe também foi implementado um processo de

fragmentação onde são produzidos fragmentos de mesmo tamanho do arquivos de entrada de acordo com o número de nós do *cluster*, a fim de se obter uma melhor distribuição dos dados e balanceamento de carga. Duas estratégias para tratamento dos dados de entrada estão sendo propostas e analisadas nesta dissertação: a primeira com a concepção de que apenas um arquivo FASTA - criado a partir da combinação dos arquivos FASTA que constituem o banco de sequências para busca - é usado como entrada. Este arquivo é dividido em partições pelo HDFS no momento de seu armazenamento e refragmentado no momento de sua leitura a fim de produzir fragmentos de tamanho uniforme. A segunda estratégia baseia-se na ideia de se utilizar vários arquivos como entrada de dados. Eles são armazenados de forma independente no HDFS. Foi construída uma classe para cada estratégia. No primeiro caso, a classe é denominada *MultiFastaFileInputFormat*, e, no segundo caso, *SingleFastaInputFormat*. Ambas são representadas pelo *FastaInputFormat* na figura 13.

A primeira estratégia se vale do fato de que o Hadoop trabalha melhor com arquivos muito grandes, o que de fato é garantido na documentação do próprio *framework*. Sendo assim, como o banco de sequências possui vários arquivos FASTA, foi desenvolvido um *script* de pré-processamento para ser executado sobre os arquivos FASTA usados como dados de entrada. Esse *script* cria um arquivo unificado de entrada e o armazena no HDFS, que o divide em partições seguindo sua estratégia padrão. O *script* foi desenvolvido usando Python. Este processo consiste em nada mais do que simplesmente concatenar todos os arquivos em um arquivo unificado. Porém, para manter a organização das sequências, e não causar inconsistência nos dados, cada linha do arquivo possui a seguinte configuração: nome-do-arquivo; NTS; sequência-FASTA. NTS é o Número Total de Sequências do arquivo original da sequência em questão. Seu uso é esclarecido mais adiante. As figuras 15 e 16 ilustram, respectivamente, trechos de

arquivos FASTA comuns e do arquivo gerado pelo processo de concatenação. O arquivo unificado criado é então armazenado no HDFS.

A_Aepe_aa.fasta

```
>638190487 APE2618a conserved hypothetical protein [Aeropyrum pernix K1
MVVAVSKVIRAKYEKGMLKPLEPLDLEEGEELILEIKERSGSKGVRRFFG
IVKVRKRETGEEDYYEYISERGSVPG

>638190488 APES075 99aa long hypothetical protein [Aeropyrum pernix K1]
MGLDVEWIPDTGYRGISDKQVARIANDSRRIILTRDSDFLKPYLRKDAKY
GIIYIAEPVRKDNLDKLARNIVKALELLKEKPRLIIITSSTIESYPLTS

>638190489 APES076 84aa long hypothetical protein [Aeropyrum pernix K1]
VNAVMVLLPGYKWLEVVPGRRGGRPTVKGTRITVDDILEALANGWSVEEV
ADNYRIPIEAVYEALRYALETLRKVEVVAVESTS
```

E_Sotu_aa.fasta

```
>PGSC0003DMP400000001 PGSC0003DMT400000001
MVNPWKDYVKLSTVLMGSHGLTKGSLAFLNYPAQIMFKSAKVLPVMVMGA
FVPGLRRKYPPHEYVSAVLLVAGLILFTLADAQTSPNFSLIGVLMISGAL
VMDSFVGNYQEAIFTSNPNTTQVYLMCFCIQYDA*
>PGSC0003DMP400000002 PGSC0003DMT400000002
MVNPWKDYVKLSTVLMGSHGLTKGSLAFLNYPAQIMFKSAKVLPVMVMGA
FVPGLRRKYPPHEYVSAVLLVAGLILFTLADAQTSPNFSLIGVLMISGAL
VMDSFVGNYQEAIFTSNPNTTQMEMLYCSTIVGFPILFVAMIVTGELRIA
WPACAQHPYVYGVLVFEACATFVGQVSVLSLVAIFGAATTTMITTARKAV
TLLLSYLIFTKPLTEQHGTGLLLMCMCIIMKMLPENKPPHPRPQKIIPLQ
QTEKPRETEDDRFQIGIEEEEEKRPLV*
```

Figura 15 - Exemplo de arquivos FASTA originais

```
A_Aepe_aa.fasta 2763  >638190487 APE2618a conserved hypothetical protein [Aeropyrum pernix K1]
A_Aepe_aa.fasta 2763  MVVAVSKVIRAKYEKGMLKPLEPLDLEEGEELILEIKERSGSKGVRRFFG
A_Aepe_aa.fasta 2763  IVKVRKRETGEEDYYEYISERGSVPG
A_Aepe_aa.fasta 2763
A_Aepe_aa.fasta 2763  >638190488 APES075 99aa long hypothetical protein [Aeropyrum pernix K1]
A_Aepe_aa.fasta 2763  MGLDVEWIPDTGYRGISDKQVARIANDSRRIILTRDSDFLKPYLRKDAKY
A_Aepe_aa.fasta 2763  GIIYIAEPVRKDNLDKLARNIVKALELLKEKPRLIIITSSTIESYPLTS
A_Aepe_aa.fasta 2763
A_Aepe_aa.fasta 2763  >638190489 APES076 84aa long hypothetical protein [Aeropyrum pernix K1]
A_Aepe_aa.fasta 2763  VNAVMVLLPGYKWLEVVPGRRGGRPTVKGTRITVDDILEALANGWSVEEV
A_Aepe_aa.fasta 2763  ADNYRIPIEAVYEALRYALETLRKVEVVAVESTS
A_Aepe_aa.fasta 2763
E_Sotu_aa.fasta 56218 >PGSC0003DMP400000001 PGSC0003DMT400000001
E_Sotu_aa.fasta 56218 MVNPWKDYVKLSTVLMGSHGLTKGSLAFLNYPAQIMFKSAKVLPVMVMGA
E_Sotu_aa.fasta 56218 FVPGLRRKYPPHEYVSAVLLVAGLILFTLADAQTSPNFSLIGVLMISGAL
E_Sotu_aa.fasta 56218 VMDSFVGNYQEAIFTSNPNTTQVYLMCFCIQYDA*
E_Sotu_aa.fasta 56218 >PGSC0003DMP400000002 PGSC0003DMT400000002
E_Sotu_aa.fasta 56218 MVNPWKDYVKLSTVLMGSHGLTKGSLAFLNYPAQIMFKSAKVLPVMVMGA
E_Sotu_aa.fasta 56218 FVPGLRRKYPPHEYVSAVLLVAGLILFTLADAQTSPNFSLIGVLMISGAL
E_Sotu_aa.fasta 56218 VMDSFVGNYQEAIFTSNPNTTQMEMLYCSTIVGFPILFVAMIVTGELRIA
E_Sotu_aa.fasta 56218 WPACAQHPYVYGVLVFEACATFVGQVSVLSLVAIFGAATTTMITTARKAV
E_Sotu_aa.fasta 56218 TLLLSYLIFTKPLTEQHGTGLLLMCMCIIMKMLPENKPPHPRPQKIIPLQ
E_Sotu_aa.fasta 56218 QTEKPRETEDDRFQIGIEEEEEKRPLV*
```

Figura 16 - Exemplo de arquivo com as sequências FASTA após o processo de concatenação

Para resolver o problema de distribuição e balanceamento de carga, o *MultiFastaFileInputFormat* fragmenta o arquivo de acordo com o número de fragmentos que o usuário previamente escolheu. Com isso, o número de fragmentos é sempre configurado de acordo com o número de nós disponíveis no *cluster*. Cada fragmento tem o mesmo tamanho, porém esse tamanho pode variar de acordo com a configuração do número de fragmentos feita pelo usuário. Como exemplo, suponha um arquivo FASTA com tamanho de 20MB. No processo padrão de armazenamento do HDFS, esse arquivo iria gerar apenas uma partição, já que ele não possui mais de 64MB (o que causaria a não paralelização da tarefa). Com a classe *MultiFastaFileInputFormat* porém, o usuário pode configurá-la para que gere 4 fragmentos. Com isso, cada fragmento terá aproximadamente 5MB. Neste momento, o *MultiFastaFileInputFormat* não se preocupa em ultrapassar barreiras das sequências, ele simplesmente foca na divisão de fragmentos de mesmo tamanho para proporcionar balanceamento de carga. Essa estratégia implica em que, dependendo do tamanho dos arquivos FASTA, pode acontecer de vários arquivos FASTA inteiros estarem em um mesmo fragmento. Como cada arquivo FASTA - ou mesmo parte dele - precisa ser processado de forma independente dos outros, essa separação será feita na função *map* através dos pares chave/valor gerados pelo *FastaRecordReader*.

A segunda estratégia não realiza nenhum tipo de pré-processamento. Os arquivos com as sequências FASTA são armazenados independentemente no HDFS e distribuídos de forma automática. O *SingleFastaInputFormat* neste caso não realiza nenhum tipo de divisão específica. A divisão neste caso é feita de forma padrão pelo Hadoop, ou seja, os fragmentos têm o tamanho de até 64MB. A quantidade de fragmentos que um arquivo irá gerar está diretamente relacionada com o seu tamanho. Diferentemente do *MultiFastaFileInputFormat*, o *SingleFastaInputFormat* é chamado

para cada arquivo FASTA em particular que será usado como entrada de dados, ou seja, cada fragmento terá apenas dados de um mesmo arquivo FASTA - seja ele inteiro ou não, caso seu tamanho seja maior do que 64MB - e não de vários, como ocorre na abordagem anterior.

Em ambas as soluções os fragmentos são passados diretamente para o *FastRecorReader* para finalizar o processamento de fragmentação antes de enviá-los para o *map*.

4.2.2 FastaRecordReader

A interface *RecordReader* é nativa do Hadoop e tem como objetivo principal gerar o par chave/valor para cada linha do arquivo ou fragmento de entrada recebido. A classe *FastaRecordRearder* implementa a interface *RecordRearder*. Ela é responsável não só por gerar pares chave/valor, como também por ajustar os limites de início e fim das sequências que foram fragmentadas pela classe *MultiFastaFileInputFormat*. Tanto a classe *MultiFastaFileInputFormat* como o *SingleFastaInputFormat* utilizam a classe *FastaRecorReader*. A classe *SingleFastaInputFormat* no entanto, não utiliza o ajustes de limites dos fragmentos, já que a mesma não efetua fragmentação dos arquivos de entrada.

Cada fragmento que chega a classe *FastaRecorReader* é processado linha a linha e assim são gerados os pares chave/valor. A chave neste caso é o nome do arquivo FASTA em questão, e o valor é o restante da informação da linha. Como as classe *MultiFastaFileInputFormat* e *SingleFastaInputFormat* processam os arquivos de maneira diferente, elas terão o par chave/valor configurado de forma diferente. No primeiro caso o valor será composto pelo Número Total de Sequências (NTS) - que indica a quantidade de sequências do arquivo FASTA original sem fragmentação - e

pela sequência FASTA em si. No segundo caso, terá apenas a sequência FASTA. Os fragmentos possuem delimitadores de início e fim criados pelo *FastaInputFormat*. Esses delimitadores são representados da seguinte forma: o de início, pelos bytes do ponto de partida do fragmento; e o de final pelo somatório dos bytes iniciais com a quantidade de bytes a serem consumidos na leitura. No início da leitura de cada fragmento é feita uma validação para verificar se a primeira sequência do fragmento está completa ou não. Caso não esteja, o delimitador de início irá avançar até o início da primeira sequência completa do fragmento. Ao final da leitura de cada fragmento também é feita uma verificação a fim de se detectar se a última sequência do fragmento está completa ou não. Caso não esteja, um reajuste é feito no delimitador final para estender a leitura do fragmento além do que estava previamente estabelecido até que se tenha a sequência completa no fragmento em questão. Com os novos valores dos delimitadores configurados, a classe *FastaRecordReader* os utiliza para efetuar a leitura dos dados e criar os pares de chave/valor correspondentes. As informações de modificação dos delimitadores são armazenadas em variáveis globais e verificadas no início e no término do processo de leitura de um fragmento e atualizadas de acordo com as novas modificações. Dessa forma, evita-se que um fragmento efetue a leitura de dados que já foram processados.

Todo esse processo de ajuste dos fragmento feito pela classe *FastaRecorReader* foi possível graças à própria arquitetura do *framework* Hadoop. Os delimitadores de fragmentos, apesar de serem gerados no início do processo de leitura dos dados de entrada pelas classes *FastaInputFormat*, são apenas informações lógicas, não rígidas. Sendo assim, é possível realizar leituras além dos limites preestabelecidos. A leitura efetiva dos dados contidos no HDFS só é realizada pela classe *FastaRecorReader*. Após

a geração dos pares chave/valor, os fragmentos são enviados para serem de fato processados pela função *map*.

4.2.3 Função map

A função *map* foi desenvolvida em Python e não em Java, que é a linguagem nativa do Hadoop. A linguagem de programação Python foi escolhida devido ao uso do Hadoop Streaming para submeter tarefas para o *framework*. A escolha do Hadoop Streaming se deu devido à necessidade do uso de aplicações externas ao Hadoop - neste caso, o *hmmsearch* do Hmmer -, já que o Streaming possui um suporte natural para essas execuções. Devido à diversidade apresentada nas classes *MultiFastaFileInputFormat* e *SingleFastaInputFormat*, foi necessário construir duas funções *map*. Elas possuem duas seções principais, e diferem apenas em dois pontos que serão discutidos adiante.

A função *map* para a classe *MultiFastaFileInputFormat* é um pouco mais complexa (o código fonte da função encontra-se no Apêndice). Essa complexidade é devida ao fato de haver a necessidade de separação dos arquivos FASTA, já que neste caso o mesmo fragmento pode possuir vários arquivos FASTA, conforme descrito anteriormente.

A primeira seção do *map* consiste apenas em separar e armazenar os dados. Como a captura dos dados é feita pela entrada padrão, o *map* irá processar uma linha do fragmento de entrada por vez. No primeiro momento, o *map* separa a chave e o valor através da função *split* do Python (*split*, neste caso, é uma função de divisão de texto nativa do Python). Por consequência, as informações que estão no valor (o NTS e a sequência) também são separados pela mesma função *split*. Sendo assim, ao final desse processo inicial, o *map* terá o nome do arquivo correspondente da sequência em

questão, o número total de sequências (NTS) do mesmo arquivo e a sequência em si, todos separados. Cada uma dessas informações possui um propósito. O nome do arquivo é usado para nomear o arquivo que será gerado - no disco local do nó - para armazenar todas as sequências deste mesmo arquivo que estão nesse fragmento. O NTS será armazenado em um dicionário com o nome do arquivo correspondente para ser usado mais adiante. Por último, temos a sequência, que será gravada no arquivo previamente criado. Como o fragmento enviará todas as sequências desse mesmo arquivo em ordem (já que o fragmento é criado a partir da leitura sequencial do arquivo de entrada), esse processo se repetirá até que o marcador de texto Fim-De-Arquivo-Fasta (EOFF, do inglês *End-Of-Fasta-File*) seja recebida, ou que se chegue ao final da leitura do fragmento. No primeiro caso, todas as sequências do arquivo correspondente presentes nesse fragmento foram lidas. Com isso, o arquivo corrente é concluído, fechado e salvo no disco local do nó. Logo após, um novo arquivo é criado com o novo nome de arquivo FASTA recebido. A leitura continua até que se atinja o final do fragmento ou aparecer novamente o EOFF.

Ao final desta primeira seção, ter-se-ão todos os arquivos FASTA presentes no fragmento separados e um dicionário com o NTS de cada arquivo. O NTS é usado para corrigir o *E-value* calculado pelo hmmserch. Por padrão, o *hmmsearch* se baseia no tamanho do banco de sequências de proteínas que está sendo processado para calcular o *E-value*. Porém, como o arquivo FASTA original pode estar separado (ou seja, ter partes em fragmentos diferentes), o número de sequências em que o *hmmsearch* se baseará para calcular o *E-value* estará errado, gerando assim resultados que não são verdadeiros. O NTS é usado para fornecer ao *hmmsearch* o número real de sequências do arquivo FASTA original, e não o número de sequências do arquivo que ele está processando.

Com a primeira seção concluída, inicia-se a preparação da segunda seção com a configuração de alguns parâmetros. Primeiramente é configurado o endereço dos arquivos dos modelos (pHMM gerados na atividade de número 2 do *workflow* SciHmm) que serão usados para comparação com os arquivos FASTA armazenados no disco local. Os arquivos pHMM são replicados em todos os nós do *cluster*. Essa replicação é feita através do parâmetro "-cacheArchive", que permite que arquivos comprimidos (.zip, .tar, .tar.gz) que estão armazenados no HDFS sejam replicados e automaticamente descomprimidos localmente nos nós do *cluster*. Para facilitar o acesso ao diretório que contém os arquivos descompactados, o Hadoop permite a criação de um link simbólico ao final do parâmetro "-cacheArchive". Em seguida, é feita a configuração do endereço do local onde os arquivos FASTA foram salvos. Por último, é criada uma variável para armazenar os dados de saída.

Com as configurações iniciais preparadas, dois ciclos se iniciam. O primeiro ciclo é feito em relação aos arquivos FASTA a serem comparados. O segundo ciclo é feito em relação aos arquivos pHMM, ainda dentro do primeiro ciclo. Assim sendo, para cada arquivo FASTA do primeiro ciclo será feita uma comparação com todos os modelos pHMM do segundo ciclo.

O *hmmsearch* possui vários parâmetros de configuração. Este trabalho faz uso de alguns desses parâmetros. Eles foram usados para organizar os arquivos que podem ser gerados como resultados da execução, e, principalmente, para ajustar o processamento a fim de se ter uma qualidade melhor nos resultados.

Primeiramente foram usados os parâmetros "-A" e "-o". O *hmmsearch* realiza um alinhamento múltiplo entre as sequências que obtiveram "acerto" na busca. O parâmetro "-A" é responsável por configurar o local de gravação do resultado desse alinhamento. Nesse experimento, o parâmetro "-A" foi configurado para que o nome do arquivo

possua sempre o sufixo “*.outalignm*”. O *hmmsearch* também gera um conjunto de dados com as informações pertinentes da busca, principalmente os resultados dos cálculos estatísticos. O parâmetro “-o” é o responsável por configurar o local de gravação desses resultados. Nos experimentos, o parâmetro “-o” foi configurado para que o nome do arquivo possua sempre o sufixo “*.outsearch*”. Ambos os parâmetros possuem prefixos para os nomes dos arquivos a seguinte forma: nome-arquivo-FASTA_nome-arquivo-pHMM.

Outros parâmetros de suma importância utilizados foram os “-E”, “--domE” e “-Z”. Os dois primeiros têm como objetivo controlar a estimativa de erro da pesquisa, ou seja, o *E-value*. Ambos os parâmetros controlam a relevância de cada “acerto” encontrado, porém eles atuam em campos diferentes. O parâmetro “-E” ajusta o *E-value* para o “acerto” de sequências, enquanto o “--domE” ajusta o *E-value* para o “acerto” dos domínios das sequências. Com o valor configurado para cada um deles - ambos os parâmetros aceitam qualquer valor racional entre 0 e 1 -, o *hmmsearch* consegue reduzir o número de resultados falso-positivos, apurando assim os resultados finais reportados. O parâmetro “-Z” ajusta a informação correta com relação ao número total de sequências que estão contidas no arquivo FASTA original das sequências em questão. Este é o momento em que se utiliza a variável NTS, sendo utilizada para configurar o parâmetro “-Z” com a informação correta. Com isso, o calculo do *E-value* é feito adequadamente.

Com todos os parâmetros configurados, o *map* realiza o comando *os.system*, que efetua uma tarefa de *execute* do sistema operacional sobre o comando passado em forma de texto. Ao final do processo, os arquivos com os resultados são salvos conforme os parâmetros “-A” e “-o”. Para evitar que seja feita a validação de resultados em arquivos que não contenham acertos, uma simples verificação é feita: averiguar se o arquivo de

saída do alinhamento (".*outalignm*") foi criado ou não. Caso não tenha sido criado, ele descarta a validação e segue para a próxima execução.

O processo de validação dos acertos carrega o arquivo com sufixo ".*outsearch*" para dar início à sua leitura. Em seguida, é configurada uma variável usada como chave para toda saída (os "acertos") gerada por este arquivo. A chave é configurada com o prefixo do arquivo em questão configurado anteriormente (nome-arquivo-FASTA+nome-arquivo-pHMM), ou seja, contém o nome do arquivo FASTA e do arquivo pHMM usados na busca. Essa chave irá auxiliar o *framework* Hadoop na escrita dos resultados que é explicada mais adiante. Logo em seguida, é iniciado um processo de leitura do arquivo para capturar todos os "acertos" e suas respectivas informações para futura avaliação do pesquisador. Quando todos os "acertos" são processados, ele simplesmente abandona a leitura do arquivo, não precisando atingir o marcador de Fim-De-Arquivo (do inglês, EOF) do mesmo. Para cada linha de informação, ou seja, para cada "acerto" encontrado é concatenada a chave (que foi criada previamente ao processo) auxiliado pelo delimitador "\t", formando assim o par chave/valor de saída. Após o termino do processo de validação, o *map* envia os dados para a saída padrão. Eles são gravados nos devidos arquivos de saída sem a necessidade do uso de uma função *reduce*.

O segundo tipo *map* (utilizado na segunda estratégia) se diferencia apenas em duas partes em relação primeiro. A primeira seção do primeiro *map*, que se destina a separar e gravar os arquivos FASTA, não se faz necessária no segundo *map* simplesmente porque no caso do *SingleFastaInputFormat* não haverá mais de um arquivo FASTA por fragmento. Desta forma a única tarefa na primeira seção do segundo *map* é gravar os dados que chegam pela entrada padrão no disco local. Porém, o par chave/valor *SingleFastaInputFormat* não possui o nome arquivo que está sendo processado, e que será usado para criar o arquivo no disco local. No entanto, o Hadoop fornece vários

metadados que podem ser utilizados durante a execução. Esses metadados são obtidos através do comando *os.environ*, que permite a leitura das variáveis do ambiente. Sendo assim, o segundo *map* faz uso deste comando para capturar o nome do arquivo FASTA que está dentro do fragmento que está sendo processado.

A segunda diferença consiste no fato de que, como cada *map* irá processar somente um arquivo FASTA - e não vários como no *map* anterior -, não é necessário o uso de um segundo ciclo (que iteraria sobre os arquivos FASTA). Sendo assim, apenas um ciclo é necessário para iterar sobre os arquivos pHMM.

4.2.4 Função reduce

Esta dissertação se baseia na abordagem de apenas-map (do inglês, *map-side-only*) do Hadoop, onde não se faz uso da função *reduce*. A função *reduce* trabalha essencialmente na organização e na ordenação os resultados provenientes do *map*. Nesta dissertação, este tipo de ordenação e organização não são necessários, pois as estratégias de paralelização utilizadas precisam apenas de um controle com relação a gravação dos resultado em seus respectivos arquivos. Por esta razão, foi necessário desenvolver uma classe que implementasse a interface *OutputFormat* para a organização dos arquivos de saída.

4.2.5 MultiFastaOutputFormat

A interface *OutputFormat* do Hadoop é responsável pela escrita dos dados de saída provenientes do *reduce* ou diretamente do *map* (caso desta dissertação). Assim como a interface *InputFormat*, ela também pode ser implementada e personalizada pelo usuário de acordo com suas necessidades. Dessa forma, foi construída a classe *MultiFastaOutputFormat*.

Mesmo com a de fragmentação dos arquivos FASTA realizada no início do processo, é necessário que seus resultados sejam agrupados novamente para melhor entendimento do usuário. Para isso, foi utilizado o conceito de Saídas Múltiplas (do inglês, *Multiple Outputs*) fornecido pelo próprio Hadoop. Esse conceito trabalha com a ideia de que muitas vezes as interfaces padrões de saída do Hadoop (*OuputFormat*) não são suficientes para escrever os dados de saída de problemas solucionados através do uso do *framework*. Existem diversas limitações ligadas à nomenclatura dos arquivos e também à quantidade de arquivos de saída que podem ser escritos no HDFS. A partir disso, o Hadoop disponibiliza o conceito de Saídas Múltiplas, que permite que vários arquivos possam ser gravados no HDFS, além de permitir o controle da nomenclatura de cada um deles.

A classe *MultiFastaOutputFormat* foi desenvolvida fundamentalmente com base no conceito de Saídas Múltiplas. Ela é invocada toda vez que um par chave/valor de saída é enviado para ser escrito no HDFS. Ela utiliza um fragmentador de texto por delimitador, separando assim a chave do valor. O atributo chave neste caso contém a informação nome-arquivo-FASTA+nome-arquivo-pHMM. A classe *MultiFastaOutputFormat* utiliza essa informação para criar um arquivo com esse mesmo nome no HDFS, e, em seguida, gravar o valor nesse mesmo arquivo. Assim, para todos os pares chave/valor com a mesma chave, o valor será simplesmente gravado no arquivo já criado. Esse processo é realizado toda vez que um par chave/valor for recebido com uma chave que não possui um arquivo correspondente para ser gravado. Portanto, todos os resultados de um mesmo arquivo FASTA-pHMM produzidos em nós diferentes podem ser agrupados em um mesmo arquivo, organizando todas as informações de saída. Todo o controle de ponteiros, tabela de dispersão (baseado na chave) e uso do disco é realizado pelo próprio *framework*.

No capítulo a seguir descrevemos os resultados obtidos durante a execução de cada um dos experimentos abordados por esta dissertação que são a execução tradicional, e as duas estratégias de paralelização. Os resultados apresentados a seguir visam comprovar o bom desempenho do uso de Map/Reduce e HPC em atividades de detecção de homologia remota através das estratégias de paralelização descritas acima.

Capítulo 5 - Resultados Experimentais

Neste capítulo, são descritos os experimentos realizados a fim de verificar o desempenho das estratégias de paralelização propostas. São analisados os resultados obtidos e as conclusões sobre cada um deles. No primeiro experimento, a terceira atividade do *workflow* SciHmm (busca dos modelos sobre o banco de dados de proteínas efetuada pelo *hmmsearch*) é executada sequencialmente. No segundo experimento, as estratégias propostas nesta dissertação para paralelização da referida atividade são avaliadas. A primeira estratégia utiliza um arquivo unificado com as sequências FASTA como banco de dados para buscas efetuadas pelo *hmmsearch*; a segunda utiliza os arquivos FASTA em sua forma original como banco de dados.

A seção 5.1 apresenta as configurações de *hardware*, *softwares*, parâmetros e dados utilizados no experimento. A seção 5.2 apresenta e discute os resultados obtidos.

5.1 Ambiente de Execução

5.1.1 Hardware e Sistema Operacional

O *cluster* utilizado para realização dos experimentos faz parte do da plataforma Grid'5000 (CAPPELLO *et al.*, 2005). O Grid'5000 é uma infraestrutura computacional que disponibiliza e interliga 5000 processadores distribuídos através de *clusters* localizados em diversos pontos da França. Ele é mantido pela INRIA[6], pelo CNRS[7], por um conjunto de universidades e também por conselhos regionais daquele país.

O *cluster* utilizado nos experimentos possui um total de 50 nós disponíveis, cada um com dois processadores AMD Opteron 2218 (2.6GHz, 1MB de cache e dois núcleos), 4GB de RAM, 250GB de disco rígido do tipo SATA, e duas conexões de rede do tipo

[6] Instituto Nacional de Pesquisas em Ciência da Computação e Controle
[7] Centro Nacional de Pesquisa Científica

Gigabit Ethernet (NVIDIA MCP55 Pro). Cada nó foi configurado com o sistema operacional o Debian Wheezy (ou Debian 7).

5.1.2 Softwares

Como plataforma Java, foi utilizada a versão 7 do Open JDK[8]. Para monitorar o uso de recursos do *cluster*, foi utilizado o software Ganglia[9], em sua versão cliente. O Grid'5000 disponibiliza o servidor Ganglia como serviço. Os nós do *cluster* enviam suas informações ao servidor, que as disponibilizam aos usuários. A linguagem de programação Python em sua versão 2.7.3 foi utilizada para a implementação da função *map* e para a utilização da mesma pelo Hadoop Streaming. O pacote HMMER foi instalado em sua versão 3.0 com sua configuração padrão.

O *framework* Hadoop foi instalado em sua versão 0.22.0. A configuração foi feita de forma a ajustá-lo às necessidades específicas do experimento e assim aumentar seu desempenho. Por padrão, o Hadoop determina fator de replicação três para os dados, ou seja, cada item de dado possui réplicas em três diferentes nós. Em se tratando de um experimento em que o ambiente de execução é bastante estável, e pelo número de nós utilizados no experimento não ser muito grande, diminui-se bastante a probabilidade de falhas. Em consequência disso, a replicação de dados do *framework* não foi utilizada, sendo o fator de replicação configurado em um.

5.1.3 Parâmetros de Execução do *hmmsearch*

Dentro da função *map*, os parâmetros "-E" e "-domE" do *hmmsearch* foram configurados com o mesmo valor de 1e-20. O parâmetro "-Z" é usado apenas na primeira estratégia do segundo experimento.

---

[8] http://openjdk.java.net/
[9] http://ganglia.sourceforge.net/

5.1.4 Configuração da Execução da Tarefa Hadoop

Como esta dissertação faz uso do Hadoop Streaming, a submissão da atividade para o Hadoop precisa fazer referência ao uso do Streaming através do parâmetro "jar" com a localização do arquivo "hadoop-0.22.0-streaming.jar" Como o experimento também faz uso de classes personalizadas que implementam as interfaces *InputFormat*, *RecordReader* e *OutputFormat*, é preciso informar o caminho do arquivo ".jar" que possui essas classes compiladas através do parâmetro "-libjars".

O número de tarefas *reduce* que esta atividade irá gerar é configurado pelo parâmetro "-D mapred.reduce.tasks". Como este trabalho não faz uso de *reduce*, este parâmetro foi configurado com o valor zero.

O número de fragmentos é informado pelo parâmetro "mapred.line.input.format.linespermap". Esse parâmetro é acompanhado da classe *NLineTextInputFormat* do Hadoop, que gera os fragmentos baseado no número de linhas que um fragmento deve possuir. Esse número é informado pelo usuário através parâmetro "mapred.line.input.format.linespermap". Porém, sua finalidade foi alterada para as necessidades desta dissertação: ele passou a representar a quantidade de fragmentos que devem ser gerados pela classe *MultiFastaFileInputFormat* para a atividade. As classes de leitura dos arquivos de entrada (*MultiFastaFileInputFormat* e *SingleFastaInputFormat*) e a classe de escrita para os arquivos de saída *MultiFastaOutputFormat* são informadas, respectivamente, pelo parâmetros "-inputformat" e "-outputformat".

O parâmetro "-cacheArchive" configura o endereço do arquivo ".tar.gz" com os modelos a serem replicados e descompactados em cada nó do *cluster*, assim também como o link simbólico para ser referenciado dentro da função *map*.

O diretório dos arquivos com as sequências FASTA de entrada e o diretório de saída dos dados resultantes da atividade são informados respectivamente pelos parâmetros "-input" e "-output".

Por fim, o parâmetro "-mapper" configura o caminho para o arquivo que contém o código da função *map*.

5.1.5 Conjunto de Dados

Os arquivos com os modelos HMM (pHMM) foram gerados pelo próprio *workflow* SciHmm nas atividades anteriores. Estes modelos são comparados com um banco de dados de sequências, que neste trabalho são arquivos FASTA. Os arquivos foram extraídos do banco de dados Uniprot. Foi utilizado um total de 214 arquivos de sequências FASTA de diferentes tamanhos. Para o experimento que avalia a primeira estratégia de paralelização, eles foram concatenados em um arquivo unificado com um tamanho total de 1.2 GB. Em seguida, este arquivo foi armazenado no HDFS, sendo dividido em partições de acordo com a estratégia padrão adotada por esse sistema. Os arquivos com os modelos pHMM foram compactados em um arquivo no formato ".tar.gz" de 3.7 MB. Foram utilizados 166 modelos. Em se tratando dos arquivos pHMM, o *workflow* pode gerar esses modelos (com estatísticas diferentes) em até cinco formas diferentes, em função das opções das ferramentas de alinhamento múltiplo. Os modelos usados neste trabalho, porém, foram construídos a partir do MSA feito pelo MAFFT.

5.1.6 Cenários

Diferentes cenários foram elaborados para se avaliar o desempenho das estratégias adotadas de acordo com a utilização do número de nós do *cluster*. Os cenários variaram de acordo com a estratégia abordada em cada experimento. No primeiro experimento, o número de nós utilizados foi apenas um, já que é uma execução sequencial. Neste caso,

todos os 214 arquivos FASTA são armazenados no mesmo nó. No segundo experimento, o tamanho do *cluster* foi variado (em ambas as estratégias) de acordo com a quantidade de nós disponíveis. Sendo assim, foram construídos *clusters* com os seguintes números de nós: 2, 4, 8, 16 e 32. Nos experimentos em que é avaliada a primeira estratégia de paralelização, o número de fragmentos gerados a partir do banco de dados de sequências também foi configurado para variar. A quantidade de fragmentos variou da seguinte forma: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 e 2048. Neste caso, os 214 arquivos FASTA foram concatenados, armazenados no HDFS e fragmentados, no momento de sua leitura, de acordo com a configuração de cada cenário. A justificativa para a variação da quantidade de fragmentos utilizada, principalmente quando essas quantidades passam a ser muito maiores do que o número de nós do cluster, é a tentativa de se reduzir o tempo de execução de cada tarefas Com isso, pretendia-se dar ao Hadoop a oportunidade alocar tarefas não executadas para os nós que se tornassem disponíveis antes dos demais e, assim, conseguir balanceamento de carga. Nos experimentos que avaliam a segunda estratégia de paralelização, os 214 arquivos FASTA são armazenados no HDFS de maneira independente. Para cada cenário, cada busca foi executada três vezes.

5.2 Resultados

Os resultados aqui mostrados visam a avaliar a execução paralela da atividade de detecção de homologia remota do *workflow* SciHmm utilizando as estratégias propostas. São apresentados e analisados resultados referentes ao tempo de execução, à taxa de utilização do processador, ao fator de aceleração e aos dados biológicos gerados por cada experimento.

A tabela 2 apresenta as médias dos tempos de execução para cada experimento em cada cenário testado (na tabela, a coluna com o número de fragmentos em 214,

representa os cenários onde não houve fragmentação dos arquivos FASTA, que correspondem ao experimento sequencial e à avaliação da segunda estratégia de paralelização). Cada linha representa o número de processadores utilizados para o experimento. Este número sempre é igual ao dobro do número de nós utilizados, pois cada nó possui dois processadores. Por este motivo, a execução sequencial aparece com dois processadores, apesar de não empregar paralelismo: ela é executada em apenas um nó, mas não há como diminuir o número de processadores. A primeira linha da coluna mais à direita (célula na cor verde) mostra o tempo da execução sequencial (1 nó – dois processadores – com 214 arquivos FASTA formando o banco de dados de busca). As demais linhas da mesma coluna (na cor azul) trazem os tempos das execuções que empregam a segunda estratégia de paralelização (que utiliza os 214 arquivos FASTA separadamente, sem concatená-los em um arquivo unificado), de acordo com o número de nós empregados. As demais colunas (32 a 2048) representam, a partir da segunda linha (na cor laranja), os tempos das execuções que utilizam a primeira estratégia de paralelização, que é a que concatena os arquivos FASTA utilizados para busca e o fragmenta no momento da leitura. Neste caso, cada coluna representa o número de fragmentos em que o arquivo foi dividido no momento da leitura para distribuição entre as funções *map*. Não há resultados na primeira linha, pois são execuções que empregam sempre mais de um processador. A figura 17 apresenta os dos dados da tabela 2 em um gráfico.

Tabela 2 - Tempos médios de execução em cada cenário (em segundos)

#proc.	32	64	128	256	512	1024	2048	214
2								6636
4	2605	2581	2856	3360	4670	6579	11472	2628
8	1360	1416	1520	1786	2347	3219	2695	1309
16	754	746	803	925	1169	1831	2794	694
32	427	515	484	537	650	962	1501	403
64	397	290	309	347	421	555	802	340

Verde: execução sequencial; Laranja: execuções com a primeira estratégia de paralelização; Azul: execuções com a segunda estratégia de paralelização.

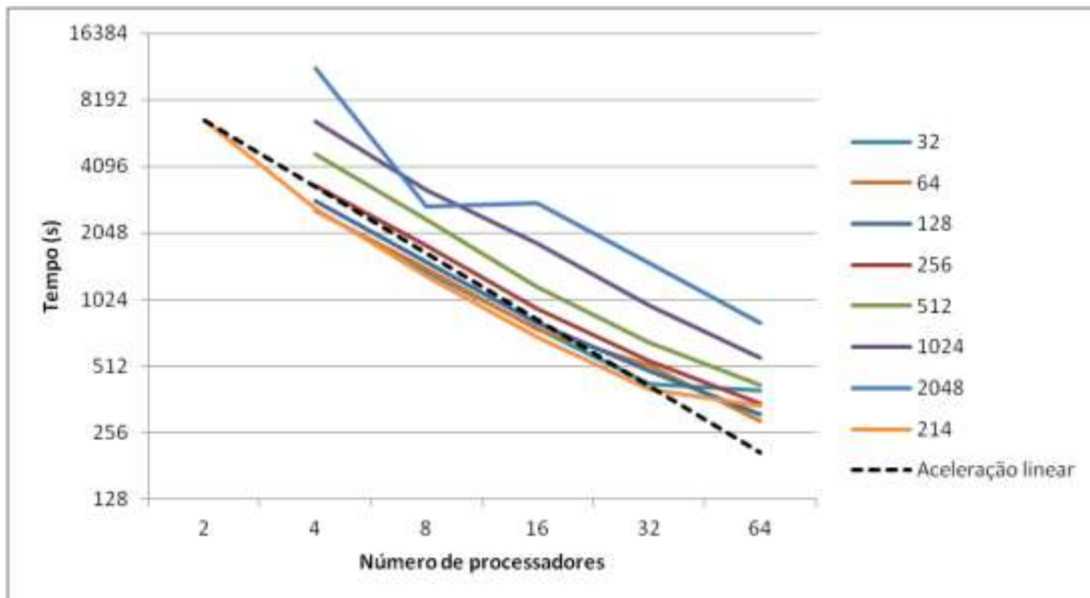

Figura 17 - Variação do tempo de execução (em segundos)

O tempo médio obtido com a execução sequencial foi de 6636 segundos. Com a primeira estratégia de paralelização, o melhor tempo obtido foi de 290 segundos, no cenário que emprega 64 processadores e divide o banco de dados de busca em 64 fragmentos. Com os novos resultados, foi observado que o tempo de execução sofreu redução conforme o número de processadores empregados foi aumentando. Porém, pode-se notar que o emprego de um número de fragmentos muito superior ao número de processadores apresentou desempenho ruim. Essa análise é feita em seguida.

Como pode ser observado, na primeira estratégia, a tentativa de se fragmentar muito o arquivo de entrada não se mostrou muito eficaz. No caso extremo, obtido com 4 processadores e 2048 fragmentos, houve uma piora significativa em relação ao próprio tempo sequencial. Durante a execução de uma atividade no Hadoop, quanto maior o número de fragmentos, maior o número de funções *map* executadas. O custo de se construir/destruir uma tarefa *map* é alto. Esse custo se dá à necessidade de se executar vários acessos ao disco em função da gravação dos resultados da execução, e das informações do *log* necessárias para a gerência do ambiente pelo Hadoop. O tráfego de

dados gerado por esse processo repetitivo e a geração de arquivos em disco oneram bastante o tempo de construção/destruição das tarefas *map*. Sendo assim, com o número de tarefas *map* muito grande em função da quantidade de fragmentos, o tempo total de execução ficou muito alto. Estes fatores explicam o desempenho ruim obtido com o emprego de um número de fragmentos muito superior ao número de processadores.

Para a segunda estratégia de paralelização, o melhor tempo obtido foi o de 340 segundos, no cenário com 64 processadores. Podemos notar que a adição de nós sempre provoca aceleração na execução. Porém, dado um número de processadores, os tempos obtidos sempre foram inferiores ao melhor tempo obtido com a primeira estratégia, exceto para a configuração com 32 processadores. Este resultado indica que a primeira estratégia é a que apresenta melhores resultados, desde que o número de fragmentos escolhido seja apropriado. Esta escolha é analisada mais adiante.

A tabela 3 apresenta a aceleração obtida em todos os cenários das estratégias de paralelização. Foi obtida uma aceleração de 22.88 e uma redução de 95.62% no tempo total de execução nos resultados da primeira estratégia em comparação com o tempo da execução sequencial, no melhor caso. O melhor resultado conseguido com a segunda estratégia foi uma aceleração de 19.52 e uma redução de tempo de 94.87% também em comparação com o tempo da execução sequencial.

Tabela 3 - Fator de aceleração dos cenários

#proc	32	64	128	256	512	1024	2048	214
4	2,55	2,57	2,32	1,98	1,42	1,01	0,58	2,53
8	4,88	4,69	4,36	3,71	2,83	2,06	2,46	5,07
16	8,80	8,89	8,26	7,17	5,68	3,62	2,38	9,56
32	15,55	12,89	13,70	12,37	10,21	6,90	4,42	16,47
64	16,72	22,88	21,50	19,11	15,77	11,95	8,27	19,52

Laranja: execuções com a primeira estratégia de paralelização; Azul: execuções com a segunda estratégia de paralelização.

Considerando o número de processadores, os fatores de aceleração apresentados na tabela 3 ficaram um pouco distantes do ideal. Mesmo no melhor caso, em cada linha da tabela, o fator de aceleração atingiu no máximo a metade do que seria o fator ideal (com

exceção do caso com 64 processadores que ficou abaixo desse valor) em comparação com um cenário de aceleração linear. Da mesma forma que a tabela da média dos tempos, o fator de aceleração vai diminuindo conforme o número de fragmentos vai aumentando.

No entanto, analisando os fatores de aceleração sobre a perspectiva do número de nós ao invés do número de processadores, as conclusões são um pouco diferentes. Conforme apresentado na tabela 4, o que pode ser observado com relação à primeira estratégia de paralelização, é que existem fatores de aceleração que foram super-lineares, enquanto outros ficaram bem próximos disso. Nos casos com 16 e 32 nós, não existem fatores de aceleração linear nem super-linear (percebe-se também que, nos casos com 16 nós, os fatores de aceleração ficaram mais próximos do linear do que nos casos com 32 nós), apesar de serem os casos com os menores tempos de execução. Os casos com o número de fragmentos maior que 128 não obtiveram fator de aceleração linear, enquanto que, quando o número de fragmentos é menor que 128, a aceleração é sempre super-linear (com exceção dos casos com 16 e 32 nós, conforme falado anteriormente). Já com relação à segunda estratégia de paralelização, todos os casos obtiveram fator de aceleração super-linear com exceção do caso com 32 nós.

Tabela 4 - Fator de aceleração dos cenários sobre perspectiva do número de nós

#nós	32	64	128	256	512	1024	2048	214
2	2,55	2,57	2,32	1,98	1,42	1,01	0,58	2,53
4	4,88	4,69	4,36	3,71	2,83	2,06	2,46	5,07
8	8,80	8,89	8,26	7,17	5,68	3,62	2,38	9,56
16	15,55	12,89	13,70	12,37	10,21	6,90	4,42	16,47
32	16,72	22,88	21,50	19,11	15,77	11,95	8,27	19,52

Laranja: execuções com a primeira estratégia de paralelização; Azul: execuções com a segunda estratégia de paralelização.

As acelerações lineares e super-lineares obtidas se analisarmos os resultados considerando o número de nós e não apenas de processadores é relevante e indica um ótimo desempenho das estratégias de paralelização propostas. Cada nó possui dois

processadores. No entanto, há apenas um disco rígido por nó. Considerando então o total de recursos e o tempo empregado em operações de acesso a disco, as acelerações obtidas são lineares e super-lineares em vários casos. Em uma dissertação que analisa processamento de dados, tal resultado é relevante.

A análise final com relação a esses primeiros resultados é que o desempenho é dependente da quantidade de fragmentos utilizada. Observando os resultados, intui-se que melhor tempo sempre é obtido quando o número de fragmentos é igual ao número de processadores. Para embasar esta conclusão, foram testados outros cenários: 2, 4, 8 e 16 fragmentos. A tabela 5 apresenta as médias dos tempos de execução e a tabela 6 os fatores de aceleração obtidos com os novos cenários. Os resultados oferecem indícios para a hipótese da relação entre o número de fragmentos e o tempo de execução. O Hadoop configura automaticamente a quantidade de tarefas de *map* que um nó pode executar simultaneamente. Essa quantidade é baseada no número de processadores que o nó possui. Como os nós do *cluster* do experimento possuem dois processadores, duas tarefas de *map* puderam ser executadas ao mesmo tempo. Em vista disso, quando o número de fragmentos se iguala ao número de processadores, todos os fragmentos são processados de uma só vez em paralelo; dois em cada nó. Isso oferece forte indício de que a fragmentação inicial do banco de dados usada pela primeira estratégia de paralelização proporciona uma divisão de tarefas balanceada entre os nós. Os melhores cenários são aqueles em que cada *map* executa apenas uma tarefa. Tal fato mostra também uma característica do tipo de processamento executado pelo *hmmsearch*: buscas em bancos de dados diferentes, mas com aproximadamente o mesmo tamanho em bytes, têm tempos de processamento semelhantes para um mesmo conjunto de modelos pHMM. As figuras 18 e 19 ilustram respectivamente a média do tempo de execução e o fator de aceleração apenas dos cenários em que o número de fragmentos é

igual ao número de processadores, além do cenário da segunda estratégia de paralelização.

Tabela 5 - Média dos tempos de execução com os novos cenários

#proc.	2	4	8	16	32	64	128	256	512	1024	2048	214
2												6636
4	3628	2198	2321	2471	2605	2581	2856	3360	4670	6579	11472	2628
8	3627	1304	1149	1289	1360	1416	1520	1786	2347	3219	2695	1309
16	3612	1337	932	642	754	746	803	925	1169	1831	2794	694
32	3603	1328	984	766	427	515	484	537	650	962	1501	403
64	3641	1338	989	782	397	290	309	347	421	555	802	340

Cores: Verde: execução sequencial; Laranja: execuções com a primeira estratégia de paralelização; Azul: execuções com a segunda estratégia de paralelização.

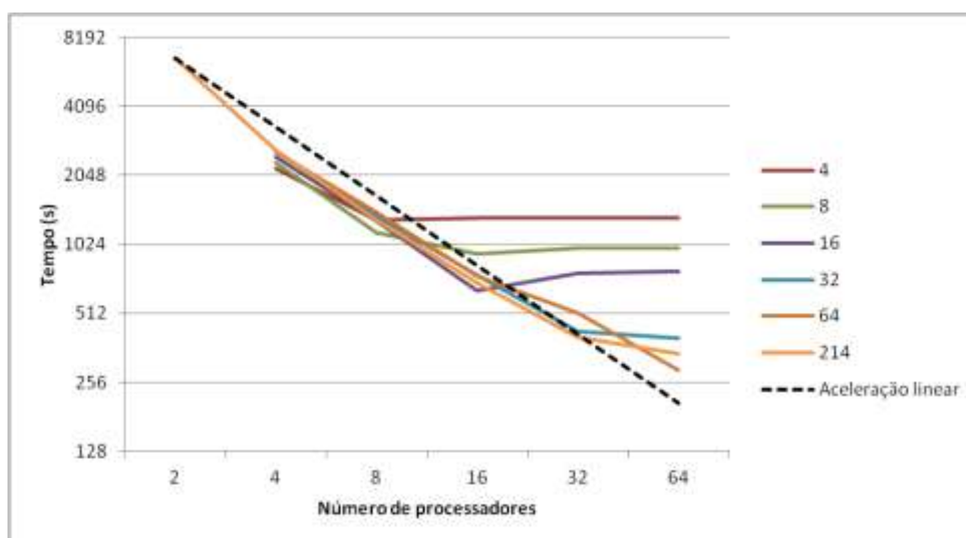

Figura 18 - Variação do tempo de execução dos novos cenários (em segundos)

Tabela 6 - Média do fator de aceleração com os novos cenários

#proc.	2	4	8	16	32	64	128	256	512	1024	2048	214
4	1,83	3,02	2,86	2,69	2,55	2,57	2,32	1,98	1,42	1,01	0,58	2,53
8	1,83	5,09	5,78	5,15	4,88	4,69	4,36	3,71	2,83	2,06	2,46	5,07
16	1,84	4,96	7,12	10,34	8,80	8,89	8,26	7,17	5,68	3,62	2,38	9,56
32	1,84	5,00	6,74	8,66	15,55	12,89	13,70	12,37	10,21	6,90	4,42	16,47
64	1,82	4,96	6,71	8,49	16,72	22,88	21,50	19,11	15,77	11,95	8,27	19,52

Cores: verde: execução sequencial; laranja: execuções com a primeira estratégia de paralelização; azul: execuções com a segunda estratégia de paralelização.

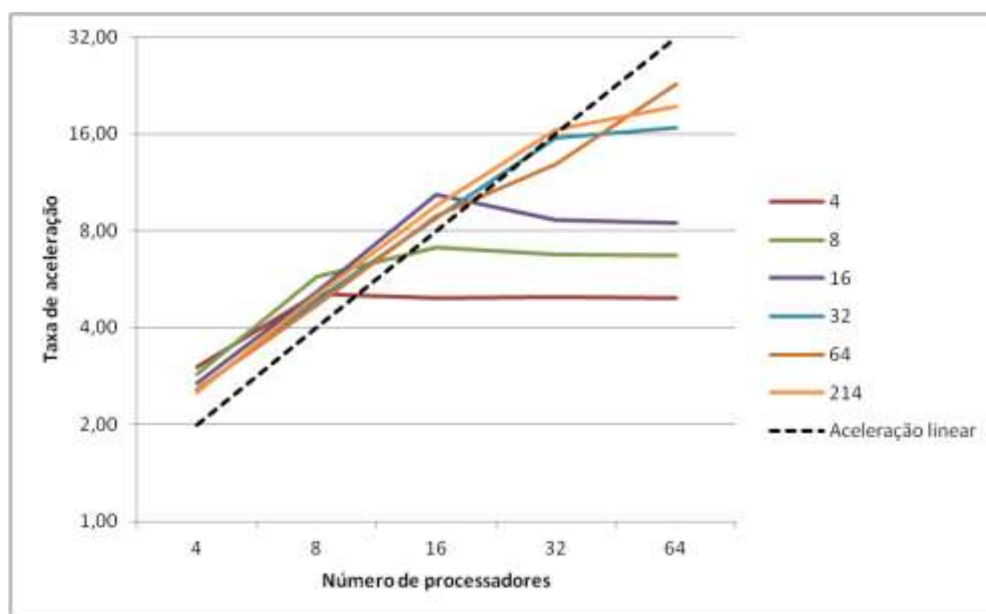

Figura 19 - Variação do fator de aceleração dos novos cenários

Acreditava-se que um número grande de fragmentos seria a melhora escolha para o processamento paralelo, pois, dessa forma, um nó com carga de trabalho menor devido a características do fragmento a ele associado poderia executar rapidamente sua tarefa e ser alocado novamente para execução de uma segunda tarefa, reduzindo o desbalanceamento de carga. Porém, de acordo com o que os resultados obtidos, as características da aplicação *hmmsearch* não provocam desbalanceamento caso a primeira estratégia de paralelização seja adotada. Para melhor entendimento do balanceamento de carga, foi analisada também a taxa de utilização do processador.

A taxa de utilização do processador é uma métrica importante a ser analisada. A tabela 7 apresenta a média da porcentagem da taxa de utilização do processador. A figura 20 mostra os dados da tabela 7 em um gráfico. Os dados relativos às execuções realizadas com a primeira estratégia de paralelização mostram que a taxa de utilização do processador é inversamente proporcional ao número de fragmentos: quanto menor o número de fragmentos, maior é a taxa de utilização do processador. Isso se explica pelo

fato de que fragmentos menores são processados mais rapidamente e mais tempo proporcional é gasto em operações de criação e destruição de funções *map* e em operações de entrada e saída, tanto em relação à rede quanto ao disco.

Tabela 7 - Média da taxa de utilização uso da cpu em cada cenário (em porcentagem)

	Número de fragmentos							
# nós	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**214**
2	x	X	x	x	x	x	x	90.46
4	81,67	83,24	78,14	68,14	60,72	43,71	38,76	89,28
8	80,52	82,32	74,38	66,13	60,87	45,12	32,94	88,95
16	84,64	83,13	71,69	67,64	60,78	42,09	31,76	86,04
32	80,62	88,87	72,96	71,32	56,22	41,83	34,12	81,47
64	75,84	86,58	82,14	71,57	58,91	41,28	35,52	60,91

Verde: execução sequencial; Laranja: execuções com a primeira estratégia de paralelização; Azul: execuções com a segunda estratégia de paralelização.

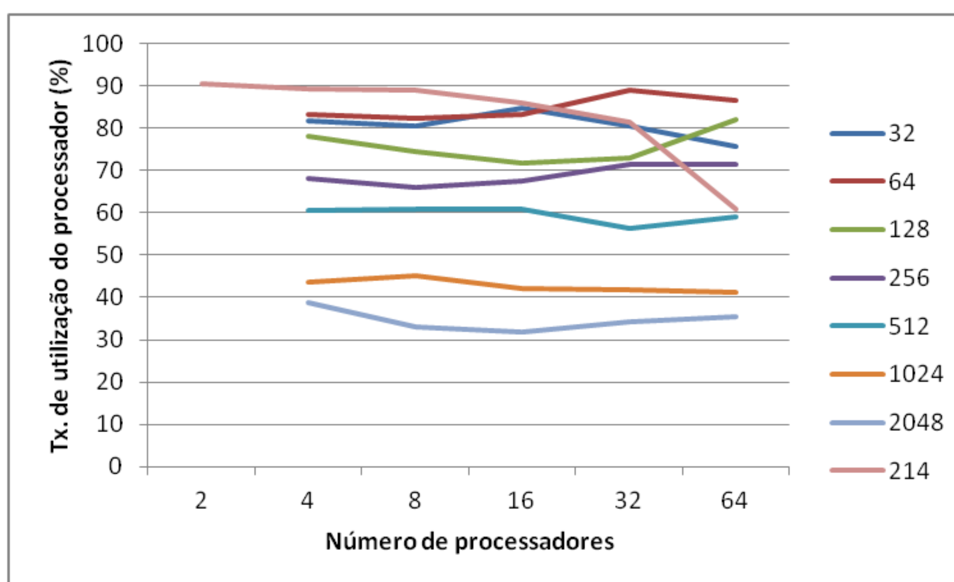

Figura 20 - Variação da taxa de utilização do processador (em porcentagem)

O balanceamento de carga foi realmente comprovado quando analisamos a taxa de utilização do processador a partir do número de fragmentos em função do número de processadores. Quando fixamos uma determinada quantidade de fragmentos e comparamos as taxas de utilização do processador entre os diferentes números de processadores, elas se mostraram bastante estáveis e equilibradas, variando muito pouco o valor de uma para outra. Sendo assim, esses resultados comprovam que houve um

bom balanceamento de carga, e que a ferramenta *hmmsearch* leva praticamente o mesmo tempo para processar buscas em banco de dados de mesmo tamanho, dado um mesmo conjunto de modelos.

Os resultados biológicos também foram analisados e verificados, pois a busca executada em paralelo deve apresentar resultados iguais - ou bastante próximos - aos da execução sequencial. As execuções que utilizaram a primeira estratégia de paralelização produziram, na maioria dos cenários, resultados biológicos iguais aos obtidos com a execução sequencial, havendo poucas variações em alguns deles. Essas variações sempre geravam mais resultados, ou seja, geravam mais "acertos" do que o resultado sequencial. O número de "acertos" obtidos em paralelo, no entanto, não excederam em mais do que cinco o número obtido com a execução sequencial. Essa porção de resultados em excesso foi analisada e concluiu-se que eles não representam um erro de execução, mas sim um resultado um pouco mais apurado do que a execução sequencial, não se tratando de nada que venha a interferir de forma brusca nos resultados produzidos pelo *workflow*.

Nas execuções que utilizaram a segunda estratégia de paralelização, os resultados biológicos não apresentaram nenhuma variação, principalmente devido ao fato de não haver concatenação e posterior fragmentação do arquivo de sequências sobre os quais as buscas foram realizadas. As execuções são idênticas à sequencial, do ponto de vista do algoritmo de busca.

Com todos os resultados apresentados nesta seção, concluímos que as estratégias de paralelização aceleraram o processo de busca no banco de sequências efetuado pelo *hmmsearch*, apresentando boa aceleração no tempo de execução, principalmente quando consideramos o número de nós utilizados. A estratégia de unificar os arquivos, armazená-los no HDFS e fragmentá-los durante sua leitura para execução mais

balanceada utilizando o Hadoop se mostrou a melhor solução. Essa solução distribui a trabalho de forma bastante equilibrada, e faz um bom balanceamento de carga entre os nós do cluster, o que o Hadoop não conseguiu efetuar de forma automática. Além disso, esta dissertação contribui também com a heurística de que, para esta aplicação (*hmmsearch*), a melhor estratégia é produzir fragmentos de tamanhos semelhantes e em quantidade tal que cada processador execute apenas uma tarefa *map* sobre um único fragmento durante toda a atividade.

Capítulo 6 - Conclusão

A genética comparativa tem exercido um papel bastante importante em pesquisas de Biologia. Porém, com o aumento do volume de dados e da complexidade dos problemas, os problemas da área vêm exigindo novas estratégias de processamento para que se possa alcançar bom desempenho em termos de tempo de execução sem comprometer a qualidade dos resultados e sem o emprego de *hardware* especializado de alto custo. A detecção de homologia remota é um dos campos da genética comparativa que vem enfrentando problemas com relação ao desempenho de suas ferramentas. Ela exerce um papel muito importante em estudos de filogenia, um dos principais campos de pesquisa e estudo atualmente. O SciHmm é um *workflow* que auxilia o processo de escolha de métodos de MSA para filogenia. Esta dissertação teve por objetivo melhorar o desempenho da atividade de detecção de homologia remota utilizando o paradigma Map/Reduce com o auxílio do *framework* Hadoop.

Tecnologias como o *framework* Hadoop (que implementa o paradigma de programação Map/Reduce) vêm sendo utilizadas na tentativa de solucionar problemas de desempenho através de processamento paralelo de larga escala em diversas áreas de pesquisa. Há vários casos de sucesso, inclusive na Bioinformática. Durante os experimentos desta dissertação, porém, foi descoberto que o procedimento padrão de fragmentação de dados implementado pelo Hadoop não se mostrou ser a melhor escolha para se alcançar um bom resultado no que diz respeito ao tempo de execução. Ou seja, utilizar diretamente as partições geradas pelo HDFS como dados de entrada para atividades de *map* se mostrou uma escolha ruim em termos de desempenho para a atividade de detecção de homologia remota analisada nesta dissertação. Com isso, foi necessário desenvolver uma nova estratégia de fragmentação de dados para sua

distribuição entre as atividades de busca realizadas paralelamente por várias atividades de *map*. Com essa nova abordagem de fragmentação de dados, os fragmentos são gerados em função do tamanho do *cluster*. Desse modo, se alcançou não só um excelente desempenho computacional como também um ótimo balanceamento de carga durante a execução, permitindo assim que os recursos não fossem subutilizados.

O uso do paradigma de programação Map/Reduce implementado pelo Hadoop em um ambiente de HPC se mostrou uma boa alternativa em termos de desempenho para problemas de detecção de homologia remota.

As seções a seguir descrevem a contribuição, as limitações do trabalho e trabalhos futuros.

6.1 Contribuição

Esta dissertação mostra um exemplo de utilização do paradigma de programação Map/Reduce em um ambiente de HPC para melhorar o desempenho de uma atividade de detecção de homologia remota no *workflow* científico SciHmm. Sua principal contribuição é a proposta e análise de desempenho de uma estratégia de fragmentação de dados que proporciona balanceamento de carga a ser utilizada neste contexto.A fragmentação inicial implementada pelo *framework* Hadoop não se mostrou satisfatória e teve desempenho inferior ao obtido com a estratégia aqui proposta.

6.2 Limitações do Trabalho

Algumas limitações foram detectadas na estratégia proposta por essa dissertação. A primeira se refere à configuração do fator de fragmentação dos dados de entrada. Apesar de os resultados mostrarem fortes indícios de que este número deve ser configurado como igual ao número de processadores, essa hipótese não foi comprovada de maneira conclusiva. Há a possibilidade de a estratégia possuir outros padrões de configuração

para o fator de fragmentação, e que apresentem resultados melhores do que os apresentados nessa dissertação.

A segunda limitação se deve ao fato de o *framework* Hadoop não ser muito flexível na intervenção do usuário no balanceamento de carga. Apesar de haver melhora significativa no balanceamento de carga nas tarefas de detecção de homologia remota, esse balanceamento poderia ser ainda melhor caso o *framework* fosse menos restrito nessa abordagem.

6.3 Trabalhos Futuros

Como trabalhos futuros, podem ser feitos estudos mais específicos no que diz respeito à determinação do fator de fragmentação dos arquivos de entrada, levando em consideração não só o número de processadores disponíveia, mas também heterogeneidade de processadores, quantidade de memória em cada nó, latência de rede, localização de dados, entre outros.

Há também a possibilidade de distribuição dos modelos pHMM entre as tarefas *map* no lugar das sequências FASTA, ou até mesmo dos modelos e das sequências simultaneamente.

Outro trabalho futuro seria a integração da estratégia apresentada por esta dissertação com o SciCumulus para a execução do workflow SciHmm. A figura 21 apresenta uma ilustração conceitual desta possibilidade. Como pode ser observado, a integração pode ser feita de forma não-intrusiva, permitindo que ambos os ambientes sejam executados de forma independente, havendo apenas a necessidade da transferência dos modelos pHMM gerados para dentro do HDFS, assim também como a transferência dos resultados do processo de detecção de homologia remota de volta para o SciCumulus para a continuidade do processamento do *workflow*. Por fim, há também

a possibilidade do próprio SciCumulus implementar a nova abordagem de fragmentação de dados desenvolvida por essa dissertação dentro do seu processo de distribuição e paralelismo de dados.

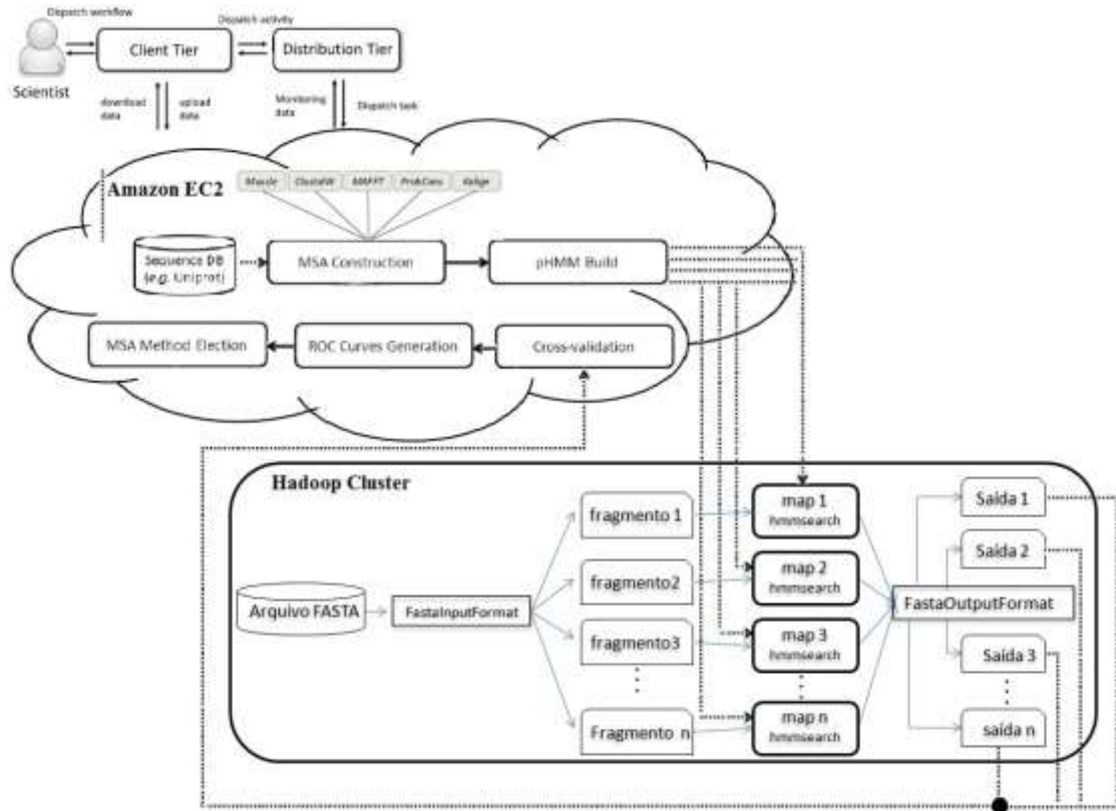

Figura 21 - Arquitetura conceitual de integração da solução de paralelização proposta com o SciCummulus

Referências

ALTINTAS, I., BERKLEY, C., JAEGER, E., JONES, M., LUDASCHER, B., MOCK, S., “Kepler: an extensible system for design and execution of scientific workflows”, in: **Proceedings of Scientific and Statistical Database Management**, pp. 423–424, Jun. 2004.

ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, E. W., LIPMAN, D. J., "Basic local alignment search tool", **Journal of molecular biology** v. 215, n. 3, pp. 403-410, Out. 1990.

AMDAHL, G.M., "Validity of the single processor approach to achieving large scale computing capabilities". In: **Proceedings of the April 18-20, 1967, spring joint computer conference**, pp. 483-485, Atlantic City, New Jersey, Abr. 1967.

ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., WANG, R. Y., "Serverless network file systems". In: **Proceedings of the fifteenth ACM symposium on Operating systems principles**, pp.109-126, Copper Mountain, Colorado, Dez. 1995.

BARKER, A., HEMERT, J. V., "Scientific workflow: a survey and research directions". In: **Proceedings of the 7th international conference on Parallel processing and applied mathematics**, pp. 746-753, Gdansk, Poland, Set. 2008.

BAVOIL, L., CALLAHAN, CROSSNO, S., P., FREIRE, J., SCHEIDEGGER, C., SILVA, C., VO, H.. "VisTrails: Enabling Interactive Multiple-View Visualizations". In: **Proceedings of IEEE Visualization**, pp. 135-142, Minneapolis, Out. 2005.

CABRERA, L., LONG, D. D. E., **SWIFT: USING DISTRIBUTED DISK STRIPING TO PROVIDE HIGH I/O DATA RATES**, Santa Cruz, CA, University of California at Santa Cruz, 1991.

CAPPELLO, F., CARON, E., DAYDE, M., DESPREZ, F., JEGOU, Y., PRIMET, JEANNOT, P., E., LANTERIM S., LEDUC, J., MELAB, N., MORNET, G., NAMYST, R., QUETIER, B., RICHARD, O., “Griade'5000: A large scale and highly reconfigurable grade experimental testbed”. In: **Proceedings of the 6th IEEE/ACM International Workshop on Grade Computing**, pp. 99-106, Washington, DC, Nov. 2005

CHAPMAN, BARBARA, JOST, GABRIELE AND RUUD VAN DER PAS, **Using OpenMP - Portable Shared Memory Parallel Programming**, Cambridge, MA, MTI Press, 2007.

CHURCHILL, G. A., "Stochastic models for heterogeneous DNA sequences", **Bulletin of Mathematical Biology** v. 51, n. 1, pp. 79-94, Jan. 1989.

COUTINHO, F., OGASAWARA, E., OLIVEIRA, D., BRAGANHOLO, V., LIMA, A. A. B., DÁVILA, A. M. R., MATTOSO, M., "Data parallelism in bioinformatics workflows using Hydra". In: **Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing**, pp.507-515, Chicago, Jun. 2010.

DAVIDSON, S. B., FREIRE, J., “Provenance and scientific workflows: challenges and opportunities”, In: **Proceedings of the ACM SIGMOD international conference on Management of data**, pp. 1345-1350 Vancouver, Canada, Jun. 2008.

DEAN, J., GHEMAWAT, S., "MapReduce: simplified data processing on large clusters", **Symposium on Operating Systems Design and Implementation** v.6, n.1, pp.137-150, Mar. 2004.

DEELMAN, E., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA, G., VAHI, K., BLACKBURN, K., LAZZARINI, A., ARBREE, A., CAVANAUGH, R., KORANDA, S., "Mapping Abstract Complex Workflows onto Grid Environments", **Journal of Grid Computing**, v.1, n. 1, pp. 25-39, Mar. 2003.

DING, M., ZHENG, L., LU, Y., LI, L., GUO, S., GUO, M., "More convenient more overhead: the performance evaluation of Hadoop streaming". In: **Proceedings of the 2011 ACM Symposium on Research in Applied Computation**, pp. 307-313, Nova York, Mar. 2011.

DO, C. B., MAHABHASHYAM, M. S. P., BATZOGLOU, S., "ProbCons: Probabilistic consistency-based multiple sequence alignment," **Genome Research** v. 15, n. 2, pp. 330 -340, Fev. 2005.

EDDY, S. R., "Hidden Markov Models", **Current Opinion in Structural Biology** v. 6, n. 3, pp. 361-365, Jun. 1996.

EDDY, S. R., "A new generation of homology search tools based on probabilistic inference". In: **Genome Informatics 2009 - Proceedings of the 20th International Conference**, pp. 205-211, Ashburn, Out. 2009.

EDDY, S .R., "Accelerated profile HMM searches", **PLoS Comput. Biol.**, Outubro 2011.

EDGAR, R. C., "MUSCLE: multiple sequence alignment with high accuracy and high throughput," **Nucleic Acids Research** v. 32, n. 5, pp. 1792 -1797, Mar. 2004.

GHEMAWAT, S., GOBIOFF, H., LEUNG, S.-T., "The Google File System", **Symposium on Operating Systems Principles** v. 37, n. 5, pp. 29-43, Out. 2003.

GROPP, W., LUSK, E., SKJELLUM, A., **Using MPI: Portable Parallel Programming with the Message-Passing Interface**. 2 ed. Cambridge, Massachusetts, MIT Press, 1999.

HOLLINGSWORTH , D., **The Workflow Reference Model**, Winchester, Workflow Management Coalition, 1995.

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., WEST, M. J., "Scale and performance in a distributed file system", **ACM Transactions on Computer Systems (TOCS)** v.6 n.1, p.51-81, Fev. 1988.

HUGHEY, R., KROGH, A., **Sam: Sequence Alignment And Modeling Software System**, UCSC-CRL-95-7, University of California at Santa Cruz, Santa Cruz, CA, 1995

JONES, P., SIMON, **Haskell 98 Language and Libraries: The Revised Report**, Cambridge, Cambridge University Press, 2003.

KARATSUBA, A., OFMAN, Y., “Multiplication on many-digital numbers by automatic computers”, In: **Proceedings of the USSR Academy of Sciences** v. 145, pp. 293–294, 1962.

KATOH, K., TOH, H., “Recent developments in the MAFFT multiple sequence alignment program,” **Briefings in Bioinformatics** v. 9, n. 4, pp. 286-298, Jul. 2008.

KROGH A., BROWN, M., MIAN, I. S, SJOLANDER, K., HAUSSLER, D., "Hidden Markov models in computational biology: Applications to protein modeling", **Journal of Molecular Biology** v. 235, n. 5, pp. 1501–1531, Fev. 1994

LASSMANN, T., SONNHAMMER, E. L. L., "Kalign--an accurate and fast multiple sequence alignment algorithm," **BMC Bioinformatics** v. 6, pp. 298-298, Dez. 2005.

LEE, K.-H., LEE, Y.-J., CHOI, H., CHUNG, Y. D., MOON, B., "Parallel data processing with MapReduce: a survey", **ACM SIGMOD Record** v.40 n.4, pp. 11-20, Dez. 2011.

LIN, J., DYER, C., **Data-Intensive Text Processing with MapReduce**, California, Morgan and Claypool Publishers, 2010.

MATSUNAGA, A., TSUGAWA, M., FORTES, J., "CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications", In: **Proceedings of the 2008 Fourth IEEE International Conference on eScience**, pp.222-229, Washington, DC, Dez. 2008

MILLER, W., MAKOVA, K. D., NEKRUTENKO, A., HARDISON, R. C., "Comparative Genomics," **Annual Review of Genomics and Human Genetics**, vol. 5, no. 1, pp. 15–56, Sep. 2004

OCAÑA, K. A. C. S., OLIVEIRA, D., DIAS, J., OGASAWARA, E., MATTOSO, M., "Designing a parallel cloud based comparative genomics workflow to improve phylogenetic analyses ", v. 29, n. 8, pp. 2205‑2219, Out. 2013.

OGASAWARA, E., DIAS, J., SILVA, V., CHIRIGATI, F., OLIVEIRA, D., PORTO, F., VALDURIEZ, P., MATTOSO, M., "Chiron: a parallel engine for algebraic scientific workflows", **Journal of Concurrency and Computation: Practice and Experience** v. 1, pp. 2337-2341, Mai 2013.

OGASAWARA, E., OLIVEIRA, D., CHIRIGATI, F., BARBOSA, C.E., ELIAS, R., BRAGANHOLO, V., COUTINHO, A., MATTOSO M., "Exploring many task computing in scientific workflows". In: **Proceedings of the 2nd Workshop on Many-Task Computing on Grades and Supercomputers**, pp. 1-10, Portland, Nov. 2009.

OINN, T., ADDIS, M., FERRIS, J., MARVIN, D., SENGER, M., GREENWOOD, M., CARVER, T., GLOVER, K., POCOCK, M. R., WIPAT, A., LI, P., “Taverna: a tool for the composition and enactment of bioinformatics workflows”, **Bioinformatics** v.20, n.17, pp.3045-3054, Nov. 2004.

OLIVEIRA, D., OGASAWARA, E., BAIÃO, F., MATTOSO, M., "SciCumulus: A Lightweight Cloud Middleware to Explore Many Task Computing Paradigm in Scientific Workflows". In: **Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing**, pp. 378-385, Washington, Jul. 2010.

RABINER, L. R., "A tutorial on hidden Markov models and selected applications in speech recognition". In: *Readings in speech recognition*, v. 1, Morgan Kaufmann Publishers Inc., pp. 257 - 286, 1990.

SHVACHKO, K., KUANG, H., RADIA, S., CHANSLER, R., "The Hadoop Distributed File System". In: **Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)**, pp. 1-10, Washington, Mai. 2010.

STEELE, G.L., **Common Lisp - The Language**. 2 ed. Lexington, Digital Press, 1990.

SZYPERSKI, C., **Component Software: Beyond Object-Oriented Programming**. 1 ed. Londres, Addison-Wesley Professional, 1997.

THOMPSON, J. D., HIGGINS, D. G., GIBSON, T. J., “CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence

weighting, position-specific gap penalties and weight matrix choice," **Nucleic Acids Research** v. 22, n. 22, pp. 4673-4680, Nov. 1994.

WALKER, E., GUIANG, C., "Challenges in executing large parameter sweep studies across widely distributed computing environments", In: **Workshop on Challenges of large applications in distributed environments**, p. 11-18, Monterey, California, USA, Jun. 2007.

WHITE, T., **Hadoop: The Definitive Guide**. 3 ed. Massachusetts, O'Reilly Media, 2012.

WILDE, M., HATEGAN, M., WOZNIAK, J. M., CLIFFORD, B., KATZ, D. S., FOSTER, I., "Swift: A language for distributed parallel scripting", **Parallel Computing** v.37, n.9, pp. 633-652, Set. 2011.

WILLIAM, G., EWING, L., ANTHONY, S., **Using MPI: portable parallel programming with the message-passing interface**, Cambridge, MA, MIT Press Scientific And Engineering Computation Serie, 1994.

Apêndice

```
import sys

import subprocess

import os

import tempfile

from fnmatch import fnmatch


f = open("temp.txt", 'w')

os.makedirs("outfiles")

os.makedirs("input")

numSeqFastaFile = {}

numSeqFile = ""

fileSeqName = ""

value = ""

controle = 0


#input comes from STDIN (standard input)

for line in sys.stdin:

    sequence = line.split('\t')


    try:

       fileSeqName = sequence[1]

       numSeqFile = sequence[2]

       value = sequence[3]

    except:

       continue


    if controle == 0:

       f.close()

       f = open("input/" + fileSeqName, 'w')

              numSeqFastaFile[fileSeqName] = numSeqFile

       f.write(value)

       controle = -1
```

```
    else:
        if value.find("EOFF") >= 0:
            controle = 0
        else:
            f.write(sequence[3])
f.close()

pathHmm = "hmm"
pathFasta = "input"
dirFile = os.listdir(pathHmm)
dirFasta = os.listdir(pathFasta)
lineOutput = ""

for fasta in dirFasta:
    for hmmFile in dirFile:
        if fnmatch(hmmFile, "*.hmm"):
            prefixoArq = "outfiles/" + fasta.strip() + "_" + hmmFile
            outAlignm = "-A " + prefixoArq + ".outalignm "
            outSearch = "-o " + prefixoArq + ".outsearch "
            params = "-E 1e-20 --domE 1e-20 -Z " + numSeqFastaFile[fasta] + " "
            hmm = pathHmm + "/" + hmmFile + " "
            fastaFile = pathFasta + "/" + fasta.strip()

            os.system("./hmmsearch " + outSearch + outAlignm + params + hmm +
fastaFile)
            tempdirFile = os.path.exists(prefixoArq + ". outalignm")

            if tempdirFile == True:
                search = open(prefixoArq + ". outalignm")
                key = prefixoArq.replace(pathHmm + "/", '')
                seqLine = 0
                lineOutput = ""

                for value in search:
```

```
if lineOutput == "":
 if ((value.find('E-value') >= 0) and (value.find('score') >=
0)):
  seqLine = 1
  lineOutput += key + '\t' + value + "\n"
 else:
  if ((value.find('.') >= 0) or value.find('----') >= 0):
      lineOutput += key + '\t' + value + "\n"
  else:
      if ((value.find('\n') >=0) and (len(value) < 2)):
         break
print lineOutput + "\n"
```