6 Instanciações do *Framework* para Sistemas de Gerência de Análises de Biossequências

6.1 Introdução

Este capítulo apresenta instanciações do *framework* para sistemas de gerência de análises em biossequências (SGABio) no contexto dos ambientes de trabalho mais comuns dos pesquisadores, enfatizando as tarefas do sub-sistema SGWBio.

O ambiente de trabalho mais simples é o denominado aqui de *ambiente pessoal*, útil quando um pesquisador está realizando um trabalho em uma máquina pessoal, e comum em pequenos laboratórios onde poucos pesquisadores estão trabalhando em conjunto, compartilhando recursos disponibilizados em uma máquina.

O segundo ambiente de trabalho é denominado de *ambiente de laboratório*, utilizado normalmente em laboratórios onde o investimento financeiro dos projetos de pesquisa são maiores e, consequentemente, os pesquisadores possuem um parque de máquinas com mais recursos.

O terceiro ambiente mais comum é o chamado de *ambiente de comunidade*, normalmente utilizado por pesquisadores, pertencentes a diferentes laboratórios, que se unem formando uma comunidade. Este caso é comum em grandes projetos de pesquisa que necessitam da colaboração de várias pessoas e de recursos de diversos laboratórios.

As próximas seções apresentarão as arquiteturas de hardware destes ambientes de trabalho e as instanciações do SGABio nestes ambientes.

6.2 Arquiteturas de Hardware dos Ambientes de Trabalho dos Pesquisadores

6.2.1 Ambiente Pessoal

A arquitetura de hardware de um ambiente pessoal caracteriza-se por possuir um único nó processador, responsável por todas as tarefas solicitadas pelo usuário.

Neste tipo de arquitetura, naturalmente, o pesquisador tem um controle mais direto sobre o processamento, oferecendo-lhe mais confiança sobre a confidencialidade dos dados e a disponibilidade do sistema, por exemplo. A primeira questão é particularmente sensível já que, de fato, existem alguns projetos de pesquisa em que os pesquisadores não querem que seus dados sejam enviados para *sites* desconhecidos.

6.2.2 Ambiente de Laboratório

A arquitetura de hardware de um ambiente de laboratório caracteriza-se por ser tipicamente composta de vários processadores, interligados por uma rede de alta velocidade, onde cada um possui seu próprio armazenamento secundário. Este ambiente, ilustrado na Figura 17, é conhecido na literatura por arquitetura sem compartilhamento [Silberschatz, 1999]. Especificamente, este trabalho supõe que, neste tipo de ambiente, o custo de um processador acessar um arquivo armazenado em um disco local ou em um disco em um outro processador é praticamente o mesmo.

Brevemente, nesta arquitetura, diferentemente da centralizada, não é necessário a instalação de todos os serviços necessários para os pesquisadores em um único processador. Ou seja, cada processador pode ser responsável por parte das aplicações e dados.

De forma simples, é possível distribuir a execução de um workflow para diferentes processadores. De forma mais sofisticada, é viável ainda distribuir dados entre vários processadores, paralelizando a execução de tarefas, principalmente em um ambiente que contemple um número significativo de processadores.

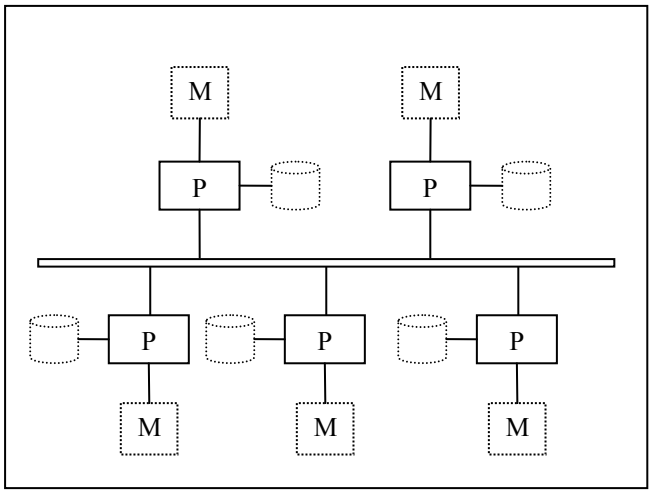

Figura 17. Arquitetura sem compartilhamento.

Esta arquitetura aumenta ainda a confiabilidade do sistema. Por exemplo, uma tarefa pode ser re-submetida a outro processador ao se detectar uma falha no que a estava executando. Em situações mais críticas, várias cópias da mesma tarefa podem ser submetidas a processadores para se aumentar as chances do sucesso de sua execução.

Outros aspectos que podem ser considerados são o planejamento, a reserva e a varredura. Sobre o planejamento, pode-se ter uma forma de se descobrir automaticamente qual processador é mais apropriado para executar uma tarefa. Quanto à varredura do sistema, um processador que estiver desocupado pode sinalizar seu estado a um processo de gerenciamento. Isto fará com que a próxima tarefa que o processador possa executar seja a ele alocada. Além disso, os recursos podem ser reservados com antecedência para determinados conjuntos de tarefas. Isto pode ser feito para se cumprir prazos e garantir qualidade de serviço.

Frequentemente, os processadores possuem grande parte de sua capacidade de armazenamento de dados em disco mal utilizada. Neste tipo de arquitetura é possível agregar armazenamento não utilizado em um grande espaço virtual de armazenamento, possivelmente configurado para se obter melhor desempenho e confiança do que em um único processador. Os dados podem ser replicados para servir como *backup* e podem ser armazenados perto de processadores que mais precisam acessá-los.

6.2.3 Ambiente de Comunidade

A arquitetura de hardware de um ambiente de comunidade também possui vários processadores, onde cada um possui seu próprio armazenamento principal e secundário. Entretanto, os processadores estão localizados em diferentes laboratórios e, normalmente, não são interligados por uma rede de alta velocidade. Desta forma, o custo de acesso aos discos torna-se dependente de suas localizações.

Os principais problemas deste ambiente são os custos de comunicação já que a rede, na maioria dos casos, possui baixa velocidade, e a transmissão de dados envolve interação por software em ambos os lados.

6.3 SGWBio no Ambiente Pessoal

Esta seção apresenta os aspectos do SGABio particulares ao ambiente pessoal. Para que um SGWBio (Figura 8) seja utilizado neste ambiente, a máquina existente deve conter todos os módulos do sistema, todos os processos instalados e todos os dados armazenados em um *data warehouse*, como esquematizado na Figura 18.

Considerando que o módulo assistente pode ser construído com uma interface desktop ou Web, o SGABio pode ser acessado também via rede. A máquina centralizadora pode estruturar os processos e dados que disponibiliza para o SGABio através de um sistema integrador de aplicativos e dados de Biologia Molecular, como o Bio-AXS [Seibel, 2002].

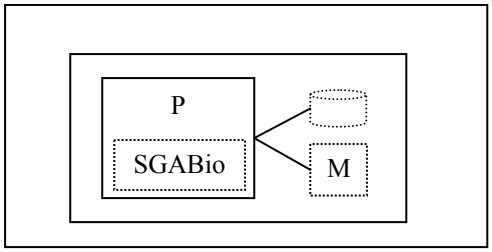

Figura 18. SGWBio em um ambiente de trabalho pessoal.

6.3.1 Implementação de Contêineres

Esta seção apresenta alternativas para implementação de contêineres, baseadas em uma análise dos tipos de contêineres.

Dada a natureza dos dados e processos, um contêiner pode potencialmente ocupar bastante espaço. Porém, dependendo do tipo do contêiner, é possível reduzir o espaço ocupado através de técnicas de *pipelining* ou *stream-based processing* [Elmasri, 1990]. De fato, suponha que o workflow contenha dois processos, p_1 e p_2, tais que a saída de p_1 seja um conjunto de itens de dados que precise ser lido por p_2. Então, dependendo da semântica de p_1 e p_2, os itens de dados gerados por p_1 poderão ser passados para p_2 assim que p_1 os produz, ou seja, sem que p_2 espere p_1 produzir o conjunto completo de saída.

Recorde que este comportamento está representado na ontologia através das propriedades das conexões e que os tipos de contêineres são definidos a partir das propriedades das conexões. Para facilitar a leitura, repetimos aqui as tabelas das seções 3.4 e 3.5 que definem as propriedades das conexões e os tipos de contêineres.

Tabela 4 – Propriedade gradativa para conexão de leitura.

Conexão de leitura	Definição
gradativa	consumidor pode ler item de dados assim que for liberado para leitura consumidor lê cada item de dados uma vez, e apenas uma vez
não-gradativa	consumidor começa a ler apenas depois de todos os itens de dados estarem liberados para leitura consumidor pode reler item de dados

Tabela 5 – Propriedade gradativa para conexão de escrita.

Conexão de escrita	Definição
gradativa	produtor libera item de dados para leitura imediatamente após escrevê-lo produtor escreve item de dados uma vez, e apenas uma vez
não-gradativa	produtor libera os itens de dados para leitura apenas depois de escrever todos os itens de dados produtor pode rescrever item de dados

Tabela 6 – Tipos de contêineres.

Tipo	Definição
gradativo	todas as conexões de entrada ou saída do contêiner são gradativas
não-gradativo	todas as conexões de entrada ou saída do contêiner são não-gradativas
misto	existe uma conexão de entrada ou saída do contêiner que é gradativa e existe uma conexão de entrada ou saída do contêiner que é não gradativa

Os contêineres serão implementados através das classes de objetos BufferLimitado, BufferIlimitado, Arquivo, ArquivoBufferLimitado e ArquivoBufferIlimitado, cujos comportamentos estão resumidos na Tabela 7, Tabela 8, Tabela 9, Tabela 10 e Tabela 11, respectivamente. A Tabela 12 apresenta estimativas dos tamanhos mínimo e máximo destas estruturas e, por conseguinte, das propriedades equivalentes definidas para os contêineres.

Brevemente, um buffer limitado oferece uma estrutura de armazenamento, com espaço limitado, para transferência de itens de dados entre processos produtores e processos consumidores. Oferece métodos para leitura e gravação de itens de dados semelhantes aos de uma fila, exceto que não há ordem entre os itens de dados, já que não é essencial para os processos de Bioinformática. Um item de dados é removido quando for lido por todas as conexões de leitura, e um item de dados é liberado para leitura imediatamente após ter sido gravado.

Um buffer ilimitado é semelhante a um buffer limitado, porém sem limite de espaço. Em particular, um item de dados também é removido quando for lido por todas as conexões de leitura, e um item de dados também é liberado para leitura imediatamente após ter sido gravado.

Um arquivo é uma estrutura de dados, sem limite de espaço, para armazenar as entradas e saídas de processos. Oferece métodos para leitura e gravação de itens de dados semelhantes aos de um buffer limitado, além daqueles típicos de um arquivo convencional. O espaço ocupado só é liberado pelo gerente de execução quando o contêiner é liberado. Note que a estrutura armazena itens de dados gerados por vários processos produtores, oferecendo tais itens de dados para processos consumidores.

Um arquivo com buffer limitado é uma estrutura de armazenamento que combina um arquivo de tamanho ilimitado com um buffer limitado. O arquivo

armazenará os itens de dados das conexões não-gradativas de escrita e o buffer limitado armazenará os itens de dados das conexões gradativas de escrita; ambos fornecerão itens de dados para os processos consumidores.

Um arquivo com buffer ilimitado é uma estrutura de armazenamento que combina um arquivo de tamanho ilimitado com um buffer ilimitado.

Tabela 7 – Especificação de BufferLimitado.

Organização
- estrutura de armazenamento implementada de forma semelhante a um buffer de tamanho limitado, com múltiplas conexões de leitura e de gravação, todas gradativas - um item de dados é removido quando for lido por todas as conexões de leitura - os itens de dados gerados pelas conexões de gravação não são necessariamente ordenados (não é uma fila no sentido usual) - não há prioridade entre as conexões (bloqueadas ou não)
Métodos
Open(c): - processo *abre* a conexão c com o buffer - topologia do workflow determina se a conexão é de leitura ou de gravação
Close(c): - processo fecha a conexão c com o buffer
Put(c,d): - a conexão c deve ser de gravação - se houver espaço no buffer para armazenar d, acrescenta d ao buffer - caso contrário, bloqueia a conexão c até que haja espaço no buffer para armazenar d - libera d para leitura por outra conexão
Get(c): - a conexão c deve ser de leitura - se houver algum item de dados d que ainda não tiver sido lido pela conexão c, retorna d caso contrário, bloqueia a conexão c até que o buffer possua algum item de dados que não tiver sido lido por c

Tabela 8 – Especificação de BufferIlimitado.

Organização
- estrutura de armazenamento implementada de forma semelhante a um buffer de tamanho ilimitado, com múltiplas conexões de leitura e de gravação, todas gradativas - políticas semelhantes ao BufferLimitado, exceto que o tamanho não é limitado
Métodos
- semelhantes ao BufferLimitado, exceto que Put(c,d) nunca bloqueia a conexão

Tabela 9 – Especificação de Arquivo.

Organização
- estrutura de armazenamento implementada de forma semelhante a um arquivo de tamanho ilimitado, com múltiplas conexões de leitura e de gravação, gradativas ou não-gradativas - o espaço ocupado só é liberado pelo gerente de execução quando o contêiner é liberado - os itens de dados recebidos de uma conexão não-gradativa de gravação são serializados de tal forma que é possível identificar o i-ésimo item de dado submetido pela conexão - não há prioridade entre as conexões (bloqueadas ou não)
Métodos
Open(c): - processo *abre* a conexão c com o arquivo - topologia do workflow determina se a conexão é de leitura ou de gravação
Close(c): - processo fecha a conexão c com o arquivo
Put(c,d): - a conexão c deve ser gradativa e de gravação - acrescenta d ao arquivo - libera d para leitura por outra conexão
Write(c,d) - a conexão c deve ser não-gradativa e de gravação - acrescenta d ao arquivo - d recebe uma *posição* no arquivo (um número inteiro), que é retornado - só libera d para leitura por outra conexão após c ser fechada
Rewrite(c,d,i) - a conexão c deve ser não-gradativa e de gravação - rescreve d na posição i do arquivo

Get(c) - a conexão c deve ser gradativa e de leitura - se houver algum item de dados d que ainda não tiver sido lido pela conexão c e que estiver liberado, retorna d - caso contrário, bloqueia a conexão c até que o contêiner possua algum item de dados que não tiver sido lido por c e que estiver liberado
Read(c,i) - a conexão c deve ser não-gradativa e de leitura - retorna o item de dados da posição i do contêiner c

Tabela 10 – Especificação de ArquivoBufferLimitado.

Organização
- estrutura de armazenamento implementada combinando um arquivo de tamanho ilimitado com um buffer limitado o o arquivo armazenará os itens de dados das conexões não-gradativas de escrita o o buffer limitado armazenará os itens de dados das conexões gradativas de escrita o ambos fornecerão itens de dados para os processos consumidores - o espaço ocupado pelo arquivo só é liberado pelo gerente de execução - os itens de dados recebidos de uma conexão não-gradativa de gravação são serializados de tal forma que é possível identificar o i-ésimo item de dado submetido pela conexão - não há prioridade entre as conexões (bloqueadas ou não)
Métodos
Open(c), Close(c): - semelhantes ao Arquivo
Write(c,d), Rewrite(c,d,i), Read(c,i): - a conexão c deve ser não-gradativa - semelhantes ao Arquivo
Put(c,d): - a conexão c deve ser gradativa e de gravação - se houver espaço no buffer para armazenar d, acrescenta d ao buffer - caso contrário, bloqueia a conexão c até que haja espaço no buffer para armazenar d - libera d para leitura por outra conexão

Get(c) - a conexão c deve ser gradativa e de leitura - se houver algum item de dados d que ainda não tiver sido lido pela conexão c e que estiver liberado, retorna d - caso contrário, bloqueia a conexão c até que o buffer ou o arquivo possua algum item de dados que não tiver sido lido por c e que estiver liberado

Tabela 11 – Especificação de ArquivoBufferIlimitado.

Organização
- estrutura de armazenamento implementada combinando um arquivo de tamanho ilimitado com um buffer ilimitado o o arquivo armazenará os itens de dados das conexões não-gradativas de escrita o o buffer ilimitado armazenará os itens de dados das conexões gradativas de escrita o ambos fornecerão itens de dados para os processos consumidores - o espaço ocupado pelo arquivo só é liberado pelo gerente de execução - os itens de dados recebidos de uma conexão não-gradativa de gravação são serializados de tal forma que é possível identificar o i-ésimo item de dado submetido pela conexão - não há prioridade entre as conexões (bloqueadas ou não)
Métodos
- semelhantes ao de ArquivoBufferLimitado, sem limitação de espaço para o buffer
Put(c,d): - a conexão c deve ser gradativa e de gravação - se houver espaço no buffer para armazenar d, acrescenta d ao buffer - caso contrário, bloqueia a conexão c até que haja espaço no buffer para armazenar d - libera d para leitura por outra conexão
Get(c) - a conexão c deve ser gradativa e de leitura - se houver algum item de dados d que ainda não tiver sido lido pela conexão c e que estiver liberado, retorna d - caso contrário, bloqueia a conexão c até que o buffer ou o arquivo possua algum item de dados que não tiver sido lido por c e que estiver liberado

Tabela 12 – Estimativas dos tamanhos dos contêineres

Implementação	Tamanho Mínimo	Tamanho Máximo
BufferLimitado	espaço necessário para armazenar o maior item de dados (ou seja, buffer para 1 item de dados) gerado pelos processos produtores	definido pelo gerente de execução
BufferIlimitado	somatório do espaço ocupado por todos os itens de dados gerados pelos produtores	(idêntica à estimativa do tamanho mínimo)
Arquivo	somatório do espaço ocupado por todos os itens de dados gerados pelos produtores	(idêntica à estimativa do tamanho mínimo)
ArquivoBufferLimitado	somatório do espaço ocupado por todos os itens de dados gerados pelos produtores com conexão não-gradativa, acrescido do espaço necessário para armazenar o maior item de gerado pelos processos produtores com conexão gradativa	somatório do espaço ocupado por todos os itens de dados gerados pelos produtores
ArquivoBufferIlimitado	somatório do espaço ocupado por todos os itens de dados gerados pelos produtores com conexão não-gradativa, acrescido do somatório do espaço ocupado por todos os itens de dados gerados pelos produtores	somatório do espaço ocupado por todos os itens de dados gerados pelos produtores

Conforme já foi dito, os tamanhos mínimo e máximo são importantes para a fase de otimização. Observando a Tabela 12, note que um buffer limitado sempre ocupa o menor espaço. Já um buffer ilimitado, para não bloquear processos produtores, requer que seja reservado espaço equivalente ao de um arquivo. Portanto, do ponto de vista de otimização, não se distingue de um arquivo. Porém, um buffer ilimitado, diferentemente de um arquivo, permite a liberação de um item de dados tão logo seja lido por todos os consumidores. Portanto, do ponto de vista da execução, não é equivalente a um arquivo.

Um arquivo com buffer limitado ocupa menos espaço do que um arquivo pois evita armazenar os itens de dados gerados através das conexões gradativas de escrita.

A comparação entre um arquivo com buffer ilimitado e um arquivo é paralela à comparação entre um buffer ilimitado e um arquivo.

Em resumo, do ponto de vista de uma estratégia de otimização que privilegia espaço ocupado por contêineres, poderíamos reduzir a escolha da implementação de um contêiner a três opções: buffer limitado, arquivo e arquivo com buffer limitado. Porém, do ponto de vista de gerência de memória em tempo de execução, as opções de buffer ilimitado e arquivo com buffer ilimitado são úteis.

A Tabela 13 descreve uma estratégia para escolher a implementação de um contêiner.

Para um contêiner gradativo, há três situações a analisar. Se a taxa agregada de consumo for maior ou igual à taxa agregada de produção, recomenda-se adotar um buffer limitado para implementar um contêiner do tipo gradativo. De fato, os consumidores poderão iniciar a leitura imediatamente após o buffer começar a ser abastecido, enquanto os produtores continuarão a escrever novos itens de dados no buffer, enquanto houver espaço disponível. Como a taxa agregada de consumo é maior ou igual à taxa agregada de produção, um buffer de tamanho limitado será suficiente para atender os processos. O tamanho do buffer depende do espaço de armazenamento disponível e de uma estimativa da variação das taxas de produção e consumo. Caso não seja possível estimar *a priori* o tamanho do buffer, o gerente de execução poderá escolher o tamanho do buffer adaptativamente, o que exige uma implementação mais sofisticada para buffer limitado.

Porém, neste caso (taxa agregada de consumo maior ou igual à taxa agregada de produção) se as taxas de consumo não forem aproximadamente iguais, os processos consumidores poderão ser sincronizados já que um dado armazenado no contêiner só poderá ser descartado quando todos os processos consumidores o tiverem lido. Logo, consumidores mais lentos atrasarão a execução dos consumidores mais rápidos. Portanto, recomenda-se a utilização de um buffer ilimitado para implementar o contêiner do tipo gradativo. De fato, como todos os dados gerados pelo produtor serão armazenados em um buffer

ilimitado, consumidores que lêem os dados gradativamente de forma mais lenta não atrasarão consumidores mais rápidos.

Se a taxa agregada de consumo for menor do que a taxa agregada de produção, recomenda-se adotar um buffer ilimitado para implementar um contêiner do tipo gradativo. De fato, como a taxa agregada de produção é maior do que a de consumo, um buffer limitado acabaria sendo completamente preenchido enquanto ainda está sendo consumido. Portanto, é preferível utilizar um buffer ilimitado. Poder-se-ia usar também um arquivo, que também permite iniciar o consumo assim que houver algum item de dados, através do método Get. Porém, o buffer ilimitado permite ao gerente de execução liberar cada item de dados assim que for lido por todos os processos consumidores (lembre-se que todas as conexões são gradativas), diferentemente de um arquivo, que mantém todos os itens de dados escritos até que o gerente de execução libere o contêiner.

Caso não seja possível estimar as taxas agregadas de produção e consumo, é recomendável a utilização de um buffer ilimitado, prevendo-se o pior caso.

Para um contêiner não-gradativo, só é possível usar um arquivo já que os processos (por definição de conexão não-gradativa) necessitam dos métodos *Read*, *Write* e *Rewrite*.

Para um contêiner misto, há três situações a analisar. Se todas as conexões de leitura são gradativas e a taxa agregada de consumo for maior ou igual à taxa agregada de produção das conexões gradativas de escrita, então é possível utilizar um arquivo com buffer limitado, economizando-se assim espaço. De fato, como todas as conexões de leitura são gradativas, os itens de dados enviados através das conexões gradativas de escrita podem ser repassados, através do buffer limitado, para as conexões gradativas de leitura.

Se todas as conexões de leitura são gradativas e a taxa agregada de consumo for menor do que a taxa agregada de produção das conexões gradativas de escrita, então é possível utilizar um arquivo com buffer ilimitado, economizando-se assim também algum espaço.

Se alguma conexão de leitura é não-gradativa, então é necessário utilizar um arquivo. De fato, se houver alguma conexão não-gradativa de leitura, só é possível usar um arquivo já que os processos com conexões não-gradativas necessitam dos métodos *Read*, *Write* e *Rewrite*.

Tabela 13 – Escolha da forma de implementação de um contêiner.

Implementação	Condições para escolha da implementação (conjunção das condições em cada célula)
BufferLimitado	- contêiner gradativo - taxa agregada de consumo é igual ou maior à taxa agregada de produção - taxas individuais de consumo aproximadamente iguais
BufferIlimitado	- contêiner gradativo - taxa agregada de consumo é menor do que a taxa agregada de produção
	- contêiner gradativo - não é possível estimar a taxa agregada de consumo ou de produção
	- contêiner gradativo - taxa agregada de consumo é igual ou maior à taxa agregada de produção - taxas individuais de consumo não são aproximadamente iguais
Arquivo	- contêiner não-gradativo
	- contêiner misto - alguma conexão de leitura é não-gradativa
ArquivoBufferLimitado	- contêiner misto - todas as conexões de leitura são gradativas - taxa agregada de consumo é maior ou igual à taxa agregada de produção das conexões gradativas de escrita
ArquivoBufferIlimitado	- contêiner misto - todas as conexões de leitura são gradativas - taxa agregada de consumo é maior ou igual à taxa agregada de produção das conexões gradativas de escrita
	- contêiner misto - todas as conexões de leitura são gradativas - não é possível estimar a taxa agregada de consumo ou de produção

6.3.2 Exemplos de estimativas

6.3.2.1 Exemplos de estimativas para o tamanho dos contêineres

A Tabela 14 apresenta exemplos de estimativas para o tamanho dos resultados gerados pelos programas mais utilizados em Bioinformática para

realizar nomeação de bases, montagem de fragmentos e comparação de sequências.

A nomeação de bases pode ser feita por programas como Phred, Abiview e Chromas, por exemplo. Em todos eles, a entrada são arquivos com cromatogramas e a saída são os *reads*, conforme já foi discutido no Capítulo 2. O tamanho dos *reads* pode variar de 500 a 1000 bases (ou nucleotídeos), limite imposto pelas máquinas de sequenciamento.

Tabela 14 – Exemplos para estimativas de tamanho de contêineres.

Tarefa	**Exemplo de Programas**	**Tamanho**	
		Entrada	**Saída**
Nomeação de Bases	Phred	N cromatogramas	N *reads* de aproximadamente 500 a 1000 nucleotídeos
	Abiview		
	Chromas		
Montagem de Fragmentos	Phrap	N *reads*	N *singletons,* no pior caso
	CAP3		
	TIGR Assembler		
Comparação de Sequências	BLAST	N sequências de consulta, $S_1,...S_N$, e 1 banco de dados	Somatório de A_i, sendo *i* variando entre 1 e N, e A_i o número máximo de alinhamentos da sequência S_i com as sequências do banco de dados
	FAST		
	ssearch		

Na montagem de fragmentos, os programas como Phrap, CAP3 e TIGR Assembler, recebem como entrada os *reads* e geram como saída os contigs. Em uma estimativa muito cautelosa, é possível considerar o pior caso de execução do programa, em que não é encontrado nenhuma sobreposição nos *reads* e, consequentemente, nenhum contig é gerado. Neste caso, a saída são os próprios *reads*, chamados de *singletons*. Nenhum destes programas possui parâmetros que permitam que seja feita uma estimativa menos cautelosa. Devido à natureza dos dados e à qualidade do trabalho de sequenciamento feito em laboratório, existirá um número maior ou menor de sobreposição entre os *reads*.

No caso da tarefa de comparação de sequências, que pode ser feita, por exemplo, pelos programas das classes BLAST, FAST e ssearch, os dados de

entrada são as sequências de consulta e um banco de dados. A saída será a apresentação dos alinhamentos encontrados para cada sequência s_i de entrada, cujo tamanho pode ser estimado conhecendo-se:

- *max_seq_bd*: o número máximo de sequências do banco de dados que o programa permite que seja apresentado em seu resultado;
- *max_alin_seq*: o número máximo de alinhamentos que podem ser apresentados para cada sequência do banco de dados; e
- *max_tam_alin*: o tamanho máximo de um alinhamento.

Desta forma, o tamanho total do resultado para a comparação de todas as N sequências de entrada existentes é dado pelo somatório de tam(A_i), para *i* variando de 1 a N, sendo tam(A_i) uma estimativa para o tamanho dos alinhamentos da sequência s_i com as sequências do banco de dados, que pode ser calculado através da equação

$$max\_seq\_bd * max\_alin\_seq * max\_tam\_alin(S_i).$$

Os programas BLAST, FAST e ssearch possuem parâmetros que definem *max_seq_bd* e *max_alin_seq*. Para o programa ssearch, [DDBJ, 2004] indica que estes parâmetros são chamados de *alignments* e *scores,* respectivamente, e possuem, ambos, 100 como valor *default.* Para o programa BLAST, [WU-BLAST, 2004a] indica que estes parâmetros são chamados de *B* e *hsp_max*, respectivamente, sendo 250 o valor *default* de *B* e 1.000 o valor defaul*t* de *hsp_max*.

No entanto, é preciso estimar *max_tam_alin* pois estes programas não possuem parâmetros para configurá-lo. De acordo com [Gish, 2004], uma estimativa razoável para o tamanho de um alinhamento é *log(KMN)/H*, sendo:

- *M* o tamanho (número de bases ou resíduos) da sequência de entrada;
- *N* o tamanho do banco de dados (número de bases ou resíduos de todas as sequências do banco de dados); e
- *K* e *H* são definidos através de fórmulas apresentadas em [Karlin, 1990] que estimam o quanto um alinhamento encontrado é realmente um resultado verdadeiro, e não um resultado falso positivo.

Os parâmetros *K* e *H* são calculados pelo BLAST. Por exemplo, [WU-BLAST, 2004a] mostra um exemplo de resultado do BLASTP, onde é possível

verificar, no final de sua apresentação, os valores 0,132 e 0,370 que são atribuídos a *K* e *H*, respectivamente.

6.3.2.2 Exemplo de estimativa para taxa de consumo

O estudo das taxas de consumo dos processos de Bioinformática é importante pois influencia a escolha do tipo de contêiner (ver Tabela 13). De fato, se vários processos consumidores compartilham um contêiner e possuem taxas de consumo (ou velocidade de execução) diferentes, os processos mais lentos atrasarão a execução dos processos que possuem taxa de consumo mais alta.

Sendo assim, nos casos em que é possível estimar a taxa de consumo dos processos, torna-se viável a escolha entre um buffer limitado e um buffer ilimitado, evitando a escolha *default* do buffer ilimitado, que gasta um espaço maior em disco.

Esta questão será ilustrada através de um estudo de caso que considera o compartilhamento de contêineres entre vários processos BLAST. Este caso foi escolhido devido à sua grande importância e popularidade nas análises das biossequências.

Família BLAST

Os programas que compõem a família BLAST são BLASTP, BLASTN, BLASTX, TBLASTN e TBLASTX. Eles foram feitos para se comparar sequências de consultas de DNA ou proteína com bancos de dados de DNA ou proteína, em qualquer combinação.

Existem dois tipos básicos de medidas de similaridades ou alinhamentos de sequências, classificadas como globais ou locais, como foi citado anteriormente. Algoritmos de similaridade global otimizam o alinhamento total das duas sequências. Algoritmos de similaridade local consultam apenas subsequências (também chamados de segmentos de sequências). Geralmente, medidas de similaridades locais são preferidas em procuras em bancos de dados. A família de programas BLAST utiliza o método de similaridade local.

O Algoritmo BLAST

Para ser possível entender a estimativa do tempo de execução do BLAST, é necessário uma explicação breve de seu funcionamento.

O BLAST possui como entradas uma sequência de consulta e um banco de dados de sequências, sendo que o seu algoritmo é descrito nos seguintes passos abaixo:

- **Passo 1**. Para cada segmento s_i de tamanho w da sequência de entrada, determina-se uma lista com todos segmentos que, alinhados com s_i, tem pontuação de no mínimo T. A pontuação de dois segmentos do mesmo tamanho e alinhados, é a soma dos valores das similaridades para cada par de caracteres (chamados de bases para sequências de nucleotídeos, ou resíduos, para sequências de aminoácidos) alinhados. Os valores das similaridades são dados por matrizes de pontuação, que também podem ser configurados como parâmetros de entrada do programa.
- **Passo 2**. Analisa-se todas as sequências do banco de dados a fim de encontrar alinhamentos entre os segmentos da lista obtida no passo 1 e os segmentos das sequências do banco de dados, cuja pontuação seja igual ou maior que T. Os alinhamentos encontrados são chamados de acertos ou *hits*.
- **Passo 3**. Para cada acerto encontrado é verificado se ele está dentro de um alinhamento cuja pontuação seja suficiente para ser notificada (maior ou igual a um valor S). Isto é feito estendendo o acerto em ambas direções até que a pontuação de alinhamento atinja um valor controlado pelo parâmetro X. X é um número inteiro que representa o declínio máximo permitido para a pontuação do alinhamento dos segmentos que estão sendo estendidos. Estes acertos estendidos são chamados de HSPs (HSP, de *High Score Segment Pair*, em inglês), gerados como saída deste passo e retornados como resultados para o pesquisador.

Sensibilidade e Tempo de Execução do BLAST

Para comparações de sequências de aminoácidos, geralmente é usada a matriz PAM-120 (uma variação de [Dayhoff, 1978]), enquanto que, para comparações de sequências de DNA, as pontuações das identidades são iguais a +5, e as desigualdades -4, sendo possível, logicamente, a utilização de outras pontuações.

Os parâmetros da família BLAST, como a matriz, *T*, *w* e *X,* são configurados com valores *default* para obter uma sensibilidade moderada ou alta, mas podem ser manualmente configurados pelo usuário do programa [WU-BLAST, 2004a].

Uma análise apresentada em [Altschul, 1998] revela que a complexidade computacional do tempo esperado para a execução do BLAST é aproximadamente

(1) $aW + bN + cNW/20^w$

para os programas BLASTP, BLASTX, TBLASTN e TBLASTX que analisam sequências de aminoácidos e

(2) $aW + bN + cNW/4^w$

para o programa BLASTN que analisa sequência de nucleotídeos. A única diferença é no último termo da equação, que considera o número de bases (4 nucleotídeos) e resíduos (20 aminoácidos) existentes. Desta forma, é possível generalizar as equações para

(3) $aW + bN + cNW/\mathrm{m}^w$

sendo m igual a 4 ou 20 (dependendo do BLAST), *W* o número de segmentos gerados no passo 1, *N* o número de bases ou resíduos no banco de dados, e *a*, *b* e *c* constantes. O termo *W* estima a compilação da lista de segmentos do passo 1, o termo *N* cobre a varredura do banco de dados feita no passo 2, e o termo *NW* considera a extensão dos acertos feito no passo 3. Apesar do número de segmentos gerados, *W*, aumentar exponencialmente com a diminuição de *T*, ele aumenta linearmente com o comprimento da sequência de consulta, o que significa dizer que ao dobrar o tamanho da sequência de consulta, dobra-se o tamanho de segmentos encontrados [Altschul, 1998].

É importante frisar que o conteúdo da sequência também influencia o tempo de execução do BLAST [Gish, 2004]. As sequências com regiões repetitivas muitas vezes não possuem relação filogenética (de herança), mas são bastante comuns em várias organismos. Isto faz com que o BLAST encontre muitos *hits,* mas que, na verdade, não representam relações de parentesco, ou seja, são resultados falsos positivos. Como exemplos de regiões repetitivas estão as homopoliméricas (e.g. AAAA...AAAAA), ou regiões com repetição de dinucleotídeo (como, por exemplo, o dinucleotídeo AG:

AGAGAGAG...AGAGAG) ou de trinucleotídeo (como, por exemplo, o trinucleotídeo AGG: AGGAGGAGGAGG...AGGAGGAGG), e assim por diante.

Compartilhamento de Contêineres

A equação (3) apresenta uma estimativa para o tempo de execução do BLAST e, consequentemente, torna possível definir melhor o tipo do contêiner.

Suponha o caso de dois processos BLAST, b_1 e b_2, comparando as mesmas sequências de entrada a bancos de dados diferentes, bd_1 e bd_2, respectivamente. As sequências de entrada estão no mesmo contêiner e, portanto, o contêiner será compartilhado por b_1 e b_2. Neste caso, o processo que tiver menor taxa de consumo das sequências de entrada poderá atrasar o outro processo, com taxa de consumo mais alta. A Figura 19 e a Figura 20 ilustram as possíveis situações que podem ocorrer, sendo que a diferença está na forma de produção das sequências de entrada do BLAST.

No caso delas serem geradas de forma não gradativa (representado pela linha contínua da conexão entre p_{p1} e c_1 na Figura 19) por p_{p1}, os processos b_1 e b_2 não interferirão na execução um do outro. Já no caso delas serem geradas de forma gradativa (Figura 20), b_1 ou b_2 poderá ter sua execução atrasada, caso b_1 e b_2 tenham velocidades de consumo diferentes. Sendo assim, se for possível calcular a Equação (3) para b_1 e b_2, o tipo do contêiner será um buffer limitado, caso os resultados sejam aproximadamente iguais, ou buffer ilimitado, caso contrário.

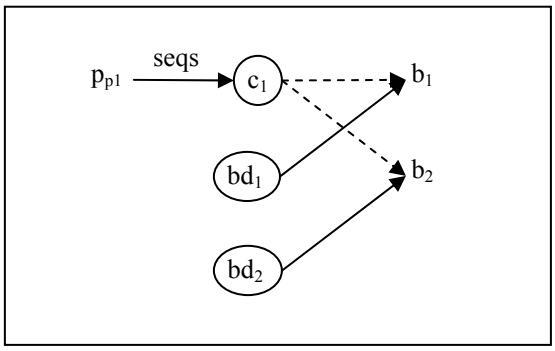

Figura 19. Exemplo 1 - BLASTs compartilhando contêineres.

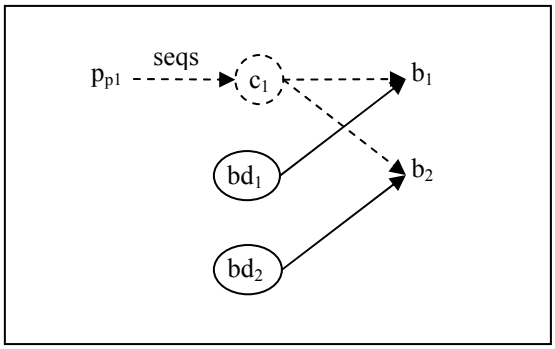

Figura 20. Exemplo 2 - BLASTs compartilhando contêineres.

Extensão para programas diferentes da família BLAST

No exemplo anterior, foi citado que os programas da família BLAST deveriam ser do mesmo tipo. Portanto, a comparação de BLASTP e TBLASTX, por exemplo, não deve ser feita aplicando-se a Equação (3) diretamente. Isto ocorre pois alguns programas da família BLAST não implementam apenas o algoritmo descrito anteriormente, mas também a tradução das sequências de nucleotídeos (de consulta ou do banco de dados) que recebe como entrada para sequências de aminoácidos. No caso em que há tradução, como BLASTX, TBLASTN e TBLASTX (Tabela 15), é necessário calcular *W* de acordo com a sequência de entrada traduzida, e calcular *N* de acordo com o banco de dados traduzido.

Tabela 15 – Tipos de BLAST.

	Sequência de Consulta	**Banco de Dados**	**Equação**
BLASTN	nt	nt	$aW + bN + cNW/4^w$
BLASTP	aa	aa	$aW + bN + cNW/20^w$
BLASTX	nt – aa	aa	$6\ (aW + bN + cNW/20^w)$
TBLASTN	aa	nt – aa	$aW + bN + cNW/20^w$
TBLASTX	nt – aa	nt – aa	$6\ (aW + bN + cNW/20^w)$

Considere o caso do BLASTX, que recebe uma sequência de consulta de nucleotídeos e um banco de dados de aminoácidos, traduz a sequência de consulta para aminoácidos nos 6 frames (conforme foi descrito no capítulo 2) e compara estas 6 sequências traduzidas com o banco de dados de aminoácidos. Isto quer dizer que os 3 passos do algoritmo descrito anterioriormente serão executados 6

vezes, uma para cada sequência de entrada de aminoácidos obtida por tradução da seqüência de nucleotídeos de entrada.

Já o TBLASTN recebe uma sequência de consulta de aminoácidos e um banco de dados de nucleotídeos, traduz as sequências de nucleotídeos do banco de dados para sequências de aminoácidos nos seis frames (isto quer dizer que o número de sequências do banco de dados será multiplicado por seis), e compara a sequência de consulta de aminoácidos contra todas as sequências de aminoácidos obtidas por tradução das seqüências de nucleotídeos do banco de dados. Isto quer dizer que os três passos do algoritmo descrito anteriormente serão executados uma vez somente para a sequência de entrada de aminoácidos, mas o parâmetro N terá que refletir as sequências de aminoácidos traduzidas, e não as sequências de nucleotídeos recebidas como entrada.

O TBLASTX é uma combinação dos dois casos anteriores.

6.3.3 Otimização Dinâmica por Pipelining

Esta seção apresentará o funcionamento do gerente de execução no ambiente pessoal. A idéia principal do algoritmo de gerência de execução é a otimização dinâmica por pipelining, ou seja identificar os processos do workflow que podem ser executados concorrentemente via pipelining, em um determinado instante. Algumas vezes, devido ao limite de espaço em disco, a execução concorrente de todos os processos será inviabilizada, e o gerente de execução terá que administrar estes casos adiando a execução de processos que não puderam ser executados.

O algoritmo se inicia com o recebimento do documento de especificação do workflow e a inicialização das seguintes estruturas de dados básicas, que serão descritas ao longo desta seção.

Estruturas de Dados Básicas

- grafo_bipartido_workflow: grafo bipartido representante do workflow, que possui os seguintes objetos:
  - nó-processo: objeto representante de um processo do workflow. Possui como atributos:

- id: identificação
- estado: estado do processo, que possui os seguintes valores: “inicial”, “pendente”, “executando”, “finalizado”

- nó-contêiner: objeto representante do local onde os nós-processo armazenarão e lerão dados. Possui como atributos:
  - id: identificação
  - temporário: booleano que é verdadeiro quando os dados armazenados no contêiner são temporários, e falso quando os dados são finais, ou seja, terão que ser armazenados em disco.
  - tamanho: espaço ocupado em disco pelos dados gerados por todos os processos p_i que armazenam dados no contêiner.
  - tamanho_máximo: espaço máximo ocupado em disco pelos dados gerados por todos os processos p_i que armazenam dados no contêiner.
- arco(nó-processo, nó-contêiner)
  - id: identificação
  - tipo: tipo de produção de dados de um nó-processo em um nó-contêiner (gradativo, não-gradativo)
  - estado: estado de produção de dados de um nó-processo em um nó-contêiner (inativo, aberto, fechado).
- arco(nó-contêiner, nó-processo)
  - id: identificação
  - tipo: tipo de consumo de dados de um nó-processo em um nó-contêiner (gradativo, não-gradativo)
  - estado: estado de consumo de dados de um nó-processo em um nó-contêiner (inativo, aberto, fechado).

Os processos que podem ser executados concorrentemente são organizados em uma lista, denominada *lista_disparos*. A identificação é feita em dois passos. No primeiro passo são encontrados componentes conexos do grafo bipartido do workflow que podem ser executados concorrentemente de acordo com suas definições no documento de especificação do workflow. A coleção destes componentes conexos é chamada de *estágio_ideal*. No segundo passo, considera-se uma situação real, onde haverá um limite de espaço em disco, que pode inviabilizar a execução concorrente de todos os processos pertencentes ao *estágio_ideal*. Sendo assim, este passo trata da descoberta do *estágio_real* que, através da poda de nós-contêiner e nós-processo do *estágio_ideal*, constrói

componentes conexos viáveis de serem executados, de acordo com a limitação do espaço em disco. Aqui, alguns processos definidos no *estágio_ideal* ficarão pendentes para serem executados em um próximo estágio.

A variável global *k* indica o espaço em disco que pode ser alocado para a execução de processos do workflow, em um determinado instante. Durante a execução do workflow, esta variável é atualizada.

Estruturas de Dados Auxiliares

- lista_disparos: lista de nós-processo que devem iniciar sua execução;
- estágio_ideal: coleção de componentes conexos (do grafo bipartido do workflow) que possuem nós-processo que podem ser disparados quando um nó-processo p_i termina sua execução, em um caso em que não é considerado limitação de espaço em disco.
- estágio_real: coleção de componentes conexos (do grafo bipartido do workflow) criados a partir de estágio_ideal, considerando a limitação de espaço em disco.
- k: espaço livre em disco para a execução do workflow em um determinado instante, sendo $k \geq 0$. É atualizado durante a execução do workflow, dependendo dos espaços utilizados pelos contêineres.

Inicialmente, inicializa-se as estruturas de dados apresentadas anteriormente. Assume-se a existência de um processo fictício p_0 que simplificará a definição do método principal deste algoritmo, pois simulando sua finalização, tem-se o evento que dispara a execução do primeiro processo do workflow.

Inicialização

- lista_disparos ← {}
- estágio_ideal ← {}
- estágio_real ← {}
- cria_grafo_bipartido_workflow(documento_especificação_wf)
- Assume-se que o processo fictício p_0 terminou sua execução.

Em seguida, cria-se o grafo bipartido do workflow (representado pelo objeto *grafo_bipartido_workflow*), a partir do documento de especificação do workflow.

Cada nó-processo terá sua *identificação* atribuída de acordo com o documento de especificação do workflow, e seu *estado* configurado como *inicial*, indicando que ainda não começou sua execução. Durante a execução do workflow, os nós-processo poderão se encontrar em outros estados, como *pendente*, indicando que o processo não foi disparado por limitação de espaço em disco (estado atribuído a um processo quando ele é podado de um *estágio_ideal*), como *executando*, indicando que o processo está sendo executado (estado configurado sempre que um processo pertence a um *estágio_real* completamente definido, ou seja, após o *estágio_ideal* ter sido podado o suficiente para poder ter seus nós-processo executados, de acordo com a limitação do espaço em disco), e como *finalizado*, indicando que o processo terminou sua execução.

Cada nó-contêiner terá os atributos *identificação*, *tipo*, *temporário, tamanho* e *tamanho_máximo* atribuídos de acordo com o documento de especificação do workflow. O *tipo* indica a forma de implementação do contêiner. O *temporário* indica se os dados que estão armazenados no contêiner são finais ou temporários. No caso de contêineres que possuem dados já armazenados em disco, como os recursos (por exemplo, bancos de dados), o atributo *temporário* será configurado como 'falso', já que estes dados não são temporários e, consequentemente, não poderão ser retirados do disco.

O atributo *tamanho* dos nós-contêiner é o espaço em disco que ele precisa para armazenar os seus dados em um determinado instante. Inicialmente o contêiner terá o atributo *tamanho* configurado com o valor mínimo necessário para armazenar seus dados, dependendo do método de implementação escolhido. Durante a execução, devido à podas de nós-processo, pode ser necessário materializar mais dados em um nó-contêiner, para que estes dados fiquem disponíveis para nós-processo que tiveram sua execução adiada. Nestes casos, o atributo *tamanho* será configurado com o valor do atributo *tamanho_máximo*.

Método cria_grafo_bipartido_workflow

- Cria-se o grafo_bipartido_workflow(documento_especificação_wf), sendo que
- Para cada nó-processo p_i existente

- Configura identificação do nó-processo de acordo com documento de especificação do workflow;
- configura_estado_processo (p_i, "inicial")

- Para cada nó-contêiner c_i existente, configura seus atributos (identificação, temporário, tamanho e tamanho_máximo) de acordo com o documento de especificação do workflow e sua forma de implementação de acordo com a Tabela 13.
- Para cada arco a_i existente, configura seus atributos (identificação, tipo, estado) de acordo com o documento de especificação do workflow. A configuração dos estados dos arcos é apresentada no método configura_estados_arcos(), descrita mais a frente.

Os contêineres que armazenam dados de recursos (como bancos de dados), ou outros dados que não são produzidos por nenhum processo do workflow, são configurados com tamanho igual a zero. Isto é feito porque estes dados já estão armazenados em disco e não influenciarão o cálculo de espaço necessário para a execução do workflow.

Os atributos de tamanho de um contêiner podem ser determinados pelo gerente de otimização no documento de especificação do workflow, quando é possível calcular o tamanho antes da execução do workflow. Entretanto, na maioria das vezes, este cálculo é feito pelo gerente de execução durante a execução do workflow. Isto acontece porque os dados gerados como saída dos processos são dependentes dos dados obtidos como entrada pelos processos. Como trata-se de uma composição de processos, o cálculo do tamanho de contêiner que contém dados de entrada de um processo *p* pode depender dos resultados de vários outros processos que devem ser executados antes dele.

O tamanho de um contêiner está associado à sua forma de implementação (Tabela 12), que é definida no método *configura_implementação_contêiner*. Este método aplica as regras definidas na Tabela 13. Em caso de poda, e consequente mudança no tamanho do contêiner, configura-se também outra forma de implementação do contêiner, que esteja de acordo com o seu novo tamanho. Este caso está ilustrado no método *poda_estágio_real*, que será apresentado mais a frente.

Na criação do grafo, os arcos também têm seus atributos *identificação*, *tipo* e *estado* configurados. O atributo *tipo* indica a forma de produção ou consumo dos dados e, por isso, pode possuir os valores *gradativo* ou *não-gradativo*. O

atributo *estado* é configurado na criação do grafo como *inativo*, indicando que o arco ainda não está sendo utilizado para a comunicação de dados. Durante a execução do workflow, os nós-processo poderão se encontrar em outros estados, como *aberto*, indicando que está aberto para produção ou consumo, e *fechado*, indicando que está fechado para produção ou consumo.

O método principal do algoritmo está definido a seguir. Trata-se de uma iteração que é feita sempre que um processo termina sua execução.

Método principal

- Loop: Aguarda um nó-processo terminar sua execução
  - Quando nó-processo p_i termina sua execução
    - configura_estado_processo (p_i, "finalizado")
    - libera_espaço_contêineres ()
    - configura_estados_arcos (p_i)
    - lista_disparos ← obtém_próximos_disparos()
    - Para cada nó-processo p_j da lista_disparos
      - configura_estado_processo (p_j, "executando");
      - configura_estados_arcos (p_j)
      - Dispare thread para executar p_j
      - Remova p_j da lista_disparos;

Espera-se que um processo p_i termine sua execução porque, neste momento, pode ser liberado um espaço do disco, que pode ser alocado para execução de outros processos. Além disso, os dados gerados por p_i podem estar sendo aguardados por outros processos, que estão em estado *pendente* ou *inicial*.

No momento do término da execução de um processo, seu estado é configurado como *finalizado* através do método *configura_estado_processo*.

Sobre a liberação do espaço em disco, é feita uma varredura por todos os contêineres no método *libera_espaço_contêineres* que identifica aqueles que possuem todos os seus arcos de entrada e de saída em estado *fechado*. Esta situação mostra que, se o nó-contêiner armazena resultados intermediários, seu espaço pode ser desalocado. Neste momento o espaço livre para execução do workflow é aumentado.

Método libera_espaço_contêineres

Definição

- Libera possíveis espaços alocados à contêineres.

Entrada

- -

Saída

- Espaços (possíveis) dos contêineres desalocados.

Método

- Forme lista l_contêineres com todos os contêineres existentes no grafo_bipartido_workflow
- Para cada c de l_contêineres
  - Forme lista l_arcos_consumo com os arcos arco(c,p_i) onde p_i é um nó-processo que consome dados armazenados em c.
  - Forme lista l_arcos_produção com os arcos arco(p_i,c) onde p_i é um nó-processo que produz dados em c.
  - lista_arcos ← l_arcos_consumo ∪ l_arcos_produção
  - libera_espaço ← verdadeiro
  - Para cada arco a_i de lista_arcos
    - se a_i.estado ≠ "fechado" então
      - libera_espaço ← falso
  - Se libera_espaço = verdadeiro E c.temporário = verdadeiro então
    - Libera o espaço deste contêiner no disco. Isto quer dizer que k ← k + c.tamanho.

Quando um processo inicia ou termina sua execução, os estados dos arcos podem se modificar. O método *configura_estados_arcos* implementa as modificações nestes estados. Quando este método é chamado sem passar como parâmetro nenhum nó-processo, ele trata da inicialização dos estados de todos os arcos do grafo bipartido do workflow. Neste caso, todos os arcos são configurados como *inativos*, exceto aqueles que conectam nós-contêiner a nós-processo tais que os nós-contêiner possuem dados já armazenados em disco, ou seja, que não serão produzidos por nenhum nó-processo do workflow. Nesta situação, os arcos já ficam configurados com estado *aberto*.

Quando o método é chamado com um nó-processo *p* que terminou sua execução, ou seja, seu estado é *finalizado*, os arcos que conectam nós-contêiner a *p* (indicando o consumo de dados de p) e os arcos que conectam *p* a nós-contêiner (indicando produção de dados) são configurados como *fechado,* já que ele não precisa mais consumir e nem irá mais produzir nenhum dado. Neste momento, se o nó-processo *p* for o último nó a produzir dados em um determinado nó-contêiner *c*, os arcos do tipo *não-gradativo*, que conectam *c* a nós-processo consumidores de dados em c, devem ser configurados como *aberto*.

Quando o método é chamado por um nó-processo *p* que iniciou sua execução, ou seja, seu estado é *executando*, os arcos que conectam nós-contêiner a *p* (indicando o consumo de dados de *p*) já devem estar no estado *aberto*. Os arcos que conectam *p* a nós-contêiner (indicando produção de dados) são configurados como *aberto*. Neste momento, se o nó-processo *p* for o primeiro nó a produzir dados em um determinado nó-contêiner *c*, os arcos do tipo *gradativo*, que conectam *c* a nós-processo consumidores de dados em *c*, devem ser configurados como *aberto*.

Método configura_estados_arcos

Definição

- Configura o estado dos arcos quando um processo termina ou inicia sua execução.

Entrada

- p: nó-processo que terminou ou iniciou sua execução

Saída

- estados dos arcos atualizados.

Método

- se p = nulo // Evento: inicialização dos arcos
- Para todos os arcos a_i de grafo_bipartido_workflow
    - a_i.estado ← "inativo"
    - Se a_i for um arco de consumo, ou seja (nó-contêiner c, nó-processo p), forme uma lista l de arcos a_j de produção de dados em c, ou seja, (p_i, c).
    - Se l = {} então // deixa aberto os arcos dos contêineres que não tem produtor
        - a_i.estado ← "aberto"
    - Retorne;

- se p.estado = "executando" então // Evento: p iniciou execução

// Tratando dos arcos (p,c_i) onde p produz dados em c_i, que mudam de estado inativo para aberto

- Forme uma lista l com todos os possíveis arcos (p, c_i)
- Para cada arco a_i de l // Tanto faz o tipo de a_i, "gradativo" ou "não-gradativo"
  - a_i.estado ← "aberto"

  // Tratando dos arcos de consumo dos dados gerados por p, que mudam de estado inativo para aberto
  - Forme uma lista l_2 com todos os possíveis arco(c_i,p_j), sendo arco(c_i,p_j).tipo = "gradativo"
  - Para cada arco a_j de l_2
    - a_j.estado ← "aberto"

- se p.estado = "finalizado" então // Evento: p terminou execução

  // Tratando dos arcos (c_i, p) onde p consome dados em c_i, que mudam de estado aberto para fechado
  - Forme uma lista l com todos os possíveis arcos (c_i, p)
  - Para cada arco a_i de l // Tanto faz o tipo de a_i, "gradativo" ou "não-gradativo"
    - a_i.estado ← "fechado"

  // Tratando dos arcos (p,c_i) onde p produz dados em c_i, que mudam de estado aberto para fechado
  - Forme uma lista l com todos os possíveis arcos (p,c_i)
  - Para cada arco a_i de l // Tanto faz o tipo de a_i, "gradativo" ou "não-gradativo"
    - a_i.estado ← "fechado"

    // Tratando dos arcos de consumo de dados gerados por p, que mudam de estado inativo para aberto
    - Forme uma lista l_2 com todos os possíveis arco(c_i,p_j), sendo arco(c_i,p_j).tipo = "não-gradativo"
    - Para cada arco a_j de l_2

      // Testa se p é o último nó-processo a produzir dados em c_i
      - Forme uma lista l_3 com todos os possíveis arco(p_x,c_i)
      - todos_fechados ← verdadeiro;
      - Para cada a_x de l_3
        - Se a_x.estado ≠ "fechado"
          - todos_fechados ← falso;
      - Se todos_fechados = verdadeiro então
        - a_j.estado ← "aberto"

Retornando ao método principal do algoritmo, após a configuração do estado do processo para *finalizado*, configuração dos estados dos arcos e a possível liberação do espaço em disco, o algoritmo trata da descoberta dos possíveis processos que podem ser executados neste momento. Isto é feito através do método *obtém_próximos_disparos*, que retorna uma lista de processos que podem ser executados. Ao receber esta lista, o gerente de execução dispara as *threads* para tratar da execução destes processos, configura os estados destes processos para *executando,* e configura os estados dos arcos que são afetados com a inicialização de tais processos.

Para retornar os processos que podem ser executados, o método *obtém_próximos_disparos* cria o *estágio_ideal* através do método *constrói_estágio_ideal* e cria o *estágio_real* passando como parâmetro o *estágio_ideal* para o método *constrói_estágio_real*. Depois, analisa o *estágio_real* e retorna os nós-processo que devem ser disparados.

Método obtém_próximos_disparos

Objetivo

- Cálculo dos próximos nós-processo que podem ser executados, quando um nó-processo p_i termina sua execução.

Entrada

- -

Saída

- lista_disparos.

Método

- estágio_ideal ← constrói_estágio_ideal
- estágio_real ← constrói_estágio_real(estágio_ideal)
- Forme lista_disparos com os nós-processo de estágio_real;
- Retorne lista_disparos

O método *constrói_estágio_ideal* cria todos os componentes conexos a partir do grafo_bipartido_workflow que possuem nós-processo que estão liberados para serem executados. Um processo está liberado para ser executado quando encontra-se no estado *inicial* ou *pendente* e possui seus dados de entrada

liberados. A liberação dos dados de entrada é verificada através do método *está_liberado*.

Método constrói_estágio_ideal

Definição

- Cálculo de estágio_ideal.

Entrada

- -

Saída

- estágio_ideal

Método

- Forme uma lista l com todos os nós-processo p que (obtém_estado_processo(p)="inicial") OU (obtém_estado_processo(p)= "pendente")
- lista_processos_liberados ← {}
- Para cada processo p de l:
    - Se está_liberado(p, lista_processos_liberados) então
        - lista_processos_liberados ← lista_processos_liberados ∪ {p}
- Crie o estágio_ideal a partir da lista_processos_liberados. O estágio_ideal é a coleção de componentes conexos que possuem nós-processo que estão liberados para serem executados de acordo com suas definições no documento de especificação do workflow.
- Retorne estágio_ideal

Método está_liberado

Definição

- Testa se um processo p está liberado para ser executado.

Entrada

- p: nó-processo que deseja descobrir se pode ser executado

Saída

- liberado: variável booleana que é verdadeiro quando p pode ser executado

Método

- Forme uma lista l com todos os arcos (c_i,p), sendo c_i os nós-contêiner que p possui dados de entrada
- liberado ← verdadeiro
- Enquanto liberado = verdadeiro E (l ≠ {})
    - Retira próximo arco(c_i,p) de l
    - Se (arco(c_i,p).estado = "aberto") // tanto faz o tipo do arco: gradativo ou não-gradativo

- liberado ← verdadeiro

- Se (arco(c_i,p).tipo = "gradativo")E (arco(c_i,p).estado = "inativo")
  - Forme lista l_2 com todos os nós-processo p_j que são produtores de c_i
  - estão_lista_para_disparar ← verdadeiro
  - Para cada p_j de l_2
    - Se $p_j \notin$ lista_processos_liberados então
      - estão_lista_para_disparar ← falso
  - Se estão_lista_para_disparar = falso
    - liberado ← falso
  - Senão
    - liberado ← verdadeiro
- Senão
  - liberado ← falso

- Retorna liberado

O método *constrói_estágio_real* poda nós-processo e nós-contêiner dos componentes conexos do *estágio_ideal* e constrói componentes conexos que sejam viáveis de serem executados, de acordo com a limitação do espaço em disco. O primeiro passo deste método é testar se não é possível executar todos os nós-processo do *estágio_ideal*. Somente se não for possível, é iniciado o procedimento de poda.

Considerando-se que o espaço livre em disco para a execução dos próximos nós-processo é k (sendo $k > 0$) e que o espaço que o *estágio_ideal* ocupa é k' (definido através do método *calcula_tamanho_estágio_ideal*), se $k' \leq k$, então o *estágio_ideal* pode ser executado. Neste caso, *estágio_real* passa ser o *estágio_ideal*. Contudo, se $k' > k$, calcula-se $s = k' - k$, espaço a mais ocupado pelo *estágio_ideal*, que o impede de ser executado. Neste caso, realiza-se a poda de nós-processo e nós-contêiner dos componentes conexos do *estágio_ideal*, diminuindo-se k', até que s fique menor ou igual a zero.

A poda é feita nos nós-contêiner que se localizam na fronteira (ou seja, que são sorvedouros, ou *sinks*, em inglês) do grafo. A escolha ótima dos nós para serem podados exige a construção da composição de todos os nós, em conjuntos de tamanho 1 a n, sendo n o número de nós da fronteira. Calcula-se o ganho de espaço em disco ao podar cada um destes conjuntos e encontra-se a melhor

solução. Este método leva a um problema semelhante ao problema de *bin packing,* que é NP-Completo [Garey, 1978] e, portanto, não será proposto aqui como solução.

A maneira escolhida aqui foi obter todos os nós-contêiner localizados nas fronteiras dos componentes conexos (e organizar na lista *l_contêineres_fronteira*), calcular o ganho de espaço ao retirar cada um deles (através do método *calcula_poda*), e podar o de menor ganho de espaço. Caso ainda não haja espaço em disco, obtém-se, novamente, os nós-contêiner das fronteiras, que serão diferentes após a poda, e reinicia-se o procedimento. Esta estratégia está baseada no método do guloso [Horowitz et al., 1996].

A escolha da poda de nós da fronteira é uma tentativa de manter o pipelining com o maior número de processos possível. Isto é feito porque há uma maior chance de se gastar menos espaço com dados temporários em contêineres utilizando-se esta estratégia, já que o pipelining é construído por nós-contêiner com tamanhos mínimos. Se for feita uma poda em algum nó-contêiner pertencente a um pipelining, os seus dados terão que ser materializados para que possam ser acessados por processos que tiverem sua execução adiada.

Método constrói_estágio_real

Definição

- Cálculo de estágio_real.

Entrada

- estágio_ideal

Saída

- estágio_real

Método

- Teste de necessidade de poda do estágio ideal
  - k' ← calcula_tamanho_estágio_ideal(estágio_ideal)
  - Se k'< k
    - estágio_ real ← estágio_ideal
    - Retorne estágio_real
- Inicialização
  - s ← k' – k
  - estágio_ real ← estágio_ideal

- l_contêineres_fronteira ← lista com todos os contêineres das fronteiras dos componentes conexos do estágio_real
- (ganho,l_processos_podar, l_contêineres_podar, l_contêineres_mudar_tamanho) ← calcula_poda(l_contêineres_fronteira);
- terminou ← falso;

- Enquanto terminou = falso
  - s ← s - ganho
  - poda_estágio_real (l_processos_podar, l_contêineres_podar, l_contêineres_mudar_tamanho);
  - se s ≤ 0 então
    - terminou ← verdadeiro
    - k ← k – s
  - senão
    - l_contêineres_fronteira ← lista com todos os contêineres das fronteiras dos componentes conexos do estágio_real
    - (ganho, l_processos_podar, l_contêineres_podar, l_contêineres_mudar_tamanho) ← calcula_poda(l_contêineres_fronteira);
- Retorne estágio_real

Método calcula_tamanho_estágio_ideal

Definição

- Calcula o espaço ocupado pelo estágio_ideal.

Entrada

- estágio_ideal

Saída

- tam_estágio_ideal.

Método

- tam_estágio_ideal ← 0
- Forme uma lista l com todos os contêineres c de estágio_ideal
- Para cada contêiner c de l:
  - tam_estágio_ideal ← tam_estágio_ideal + c.tamanho
- Retorne tam_estágio_ideal

A escolha da melhor poda é feita pelo método *calcula_poda*. Ele chama, para cada nó-contêiner *c* das fronteiras dos componentes conexos, o método

calcula_ganho, que calcula o ganho de espaço em disco ao se podar um nó-contêiner. Note que, ao retirar um nó-contêiner, os seus nós-processo produtores de dados e consumidores de dados deverão ser removidos. Ao retirar estes nós-processo, seus outros nós-contêiner também terão que ser removidos. Portanto, a retirada de um nó-contêiner provoca a retirada de nós-processo e nós-contêiner de forma direta ou indireta, como definido no método *calcula_ganho*. Sendo assim, quando se deseja retirar um determinado nó-contêiner, este método retorna o ganho de espaço em disco (variável *ganho*), os nós-processo (lista *l_processos_podar*) e os nós-contêiner (lista *l_contêineres_podar*) que terão que ser podados, e os nós-contêiner que terão seus tamanhos modificados (lista *l_contêineres_mudar_tamanho*), pois, com a poda, tais nós-contêiner terão que materializar dados para serem lidos por nós-processo que tiveram sua execução postergada para outra etapa da execução.

Após o cálculo do ganho de espaço, o método *calcula_poda* retorna a opção de menor ganho de espaço.

Método calcula_poda

Definição

- Calcula quais nós-processo e quais nós-contêiner devem ser podados do estágio_real.

Entrada

- l_contêineres_fronteira: lista com contêineres que estão na fronteira dos componentes conexos do estágio_real.

Saída

- l_processos_podar: lista de nós-processo que é necessário podar do estágio_real.
- l_contêineres_podar: lista de nós-contêiner que é necessário podar do estágio_real.
- l_contêineres_mudar_tamanho: lista de nós-contêiner que terão seus tamanhos alterados se for feita a poda dos elementos de l_processos_podar e l_contêineres_podar do estágio_real.
- ganho: ganho de espaço se for feita a poda dos elementos de l_processos_podar e l_contêineres_podar do estágio_real.

Método

- l ← {}
- Para cada contêiner c de l_contêineres_fronteira:
    - (ganho,l_processos_podar,l_contêineres_podar, l_contêineres_mudar_tamanho) ← calcula_ganho(c);

- $l \leftarrow l \cup$ {(ganho,l_processos_podar, l_contêineres_podar, l_contêineres_mudar_tamanho)}

- Ordene l por ordem crescente de ganho
- Retorne o primeiro elemento de l

Método calcula_ganho

Definição

- Cálculo do ganho de espaço de armazenamento em disco, dos nós-contêiner e nós-processo que devem ser retirados do estágio-real ao se decidir pela retirada de um contêiner da fronteira dos componentes conexos do estágio-real.

Entrada

- c: contêiner que se deseja retirar do componente conexo

Saída

- l_processos_podar: lista de nós-processo que é necessário podar do estágio_real ao se retirar o contêiner c
- l_contêineres_podar: lista de nós-contêiner que é necessário podar do estágio_real ao se retirar o contêiner c
- l_contêineres_mudar_tamanho: lista de nós-contêiner que terão seus tamanhos alterados se for feita a poda dos elementos de l_processos_podar e l_contêineres_podar do estágio_real.
- ganho: ganho de espaço se for feita a poda dos elementos de l_processos_podar e l_contêineres_podar do estágio_real.

Método

- Forme uma lista l_processos_podar com todos os nós-processo que produzem dados em c
- Para cada processo p de l_processos_podar:
  - Forme uma lista l_contêineres de todos os contêineres $c_i \neq c$ que p produz dados:
    - Para cada c_i de l_contêineres
      - Forme uma lista lp_1 de processos $p_i \neq p$ que produzem dados em c_i
      - Forme uma lista lp_2 de processos $p_i \neq p$ que consomem dados em c_i
      - l_processos_podar $\leftarrow$ l_processos_podar $\cup$ lp_1 $\cup$ lp_2
- Forme uma lista l_contêineres_podar com todos os nós-contêiner que os processos p $\in$ l_processos_podar produzem dados
- Forme uma lista l_contêineres_mudar_tamanho com todos os nós-contêiner que os processos p $\in$ l_processos_podar consomem dados,

- Retire contêineres de l_contêineres_mudar_tamanho que estão em l_contêineres_podar
- ganho ← 0
- Para cada $c_i \in$ l_contêineres_podar
    - ganho ← ganho + c_i.tamanho
- perda ← 0
- Para cada $c_i \in$ l_contêineres_mudar_tamanho (Só existe perda quando um contêiner precisa materializar todos os seus dados, sendo que antes ele não precisava. Em outros casos c_i.tamanho_máximo = c_i.tamanho.)
    - perda ← perda + c_i.tamanho_máximo – c_i.tamanho
- ganho ← ganho - perda
- Retorne (ganho, l_processos_podar, l_contêineres_podar, l_contêineres_mudar_tamanho)

A poda é feita através do método *poda_estágio_real*. Ele recebe como entrada uma lista de nós-processo e uma lista de nós-contêiner que devem ser retirados do *estágio_real*. Após estas remoções, os nós-contêiner pertencentes à lista *l_contêineres_mudar_tamanho* terão seus tamanhos atualizados. Além disso, os nós-processo que foram podados devem ter seus estados atualizados para *pendente*. Em uma versão mais complexa deste algoritmo, estes nós-processo podem ganhar prioridade de execução, não podendo ser escolhidos novamente para serem podados.

Método poda_estágio_real

Definição

- Faz a poda de nós-contêiner e nós-processo, criando um novo estágio_real. Atualiza o estado dos nós-contêiner que ficaram nas fronteiras do novo estágio_real

Entrada

- l_processos_podar: lista de nós-processo que se deseja podar do estágio_real
- l_contêineres_podar: lista de nós-contêiner que se deseja podar do estágio_real
- l_contêineres_mudar_tamanho: lista de nós-contêiner que terão seus tamanhos alterados se for feita a poda dos elementos de l_processos_podar e l_contêineres_podar do estágio_real.

Saída

- estágio_real atualizado.

Método

- Retire todos os nós-processo p ∈ l_processos_podar e todos os nós-contêiner c ∈ l_contêineres_podar do estágio_real
- Para todos os nós-contêiner c ∈ l_contêineres_mudar_tamanho
  - configura_implementação_contêiner(c)
  - c.tamanho ← c.tamanho_máximo
- Para todos os nós-processo p ∈ l_processos_podar
  - configura_estado_processo (p_j, "pendente");
- Retorne estágio_real atualizado

Alguns métodos auxiliares são apresentados a seguir.

Método configura_tipo_contêiner

Definição

- Configura o tipo de um contêiner.

Entrada

- c: nó-contêiner que se deseja configurar tipo.
- novo_tipo: tipo que se deseja configurar ao nó-contêiner.

Saída

- tipo do contêiner c atualizado.

Método

- c.tipo ← novo_tipo

Método obtém_tipo_contêiner

Definição

- Obtém o tipo de um contêiner c.

Entrada

- c: contêiner.

Saída

- c.tipo: tipo do contêiner.

Método

- Retorne c.tipo

Método configura_estado_processo

Definição

- Configura o estado de um processo p.

Entrada

- p: processo

Saída

- estado do processo atualizado.

Método

- p.estado ← novo_estado

Método obtém_estado_processo

Definição

- Obtém o estado de um processo p.

Entrada

- p: processo

Saída

- p.estado: estado do processo.

Método

- Retorne p.estado

6.3.4 Exemplo de Otimização Dinâmica por Pipelining

Como ilustração do método de otimização apresentado na Seção 6.3.3, considere o *grafo_bipartido_workflow* da Figura 21. Os nós-contêiner estão identificados por c_i, os nós-processo por p_i, e os arcos por a_{ij}, sendo i e j números inteiros. Os arcos do tipo *gradativo* são definidos por linhas pontilhadas e os arcos do tipo *não-gradativo*, por linhas contínuas.

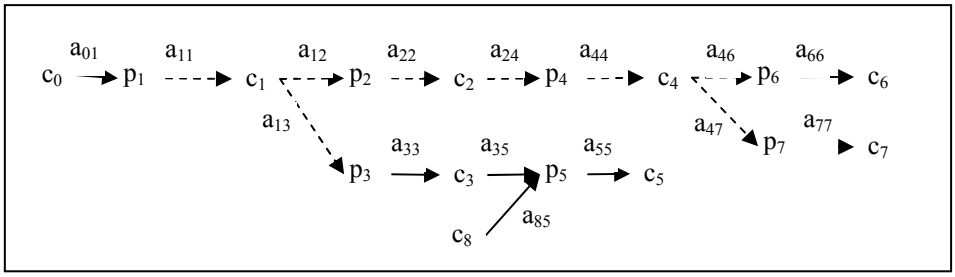

Figura 21. Exemplo: grafo bipartido.

A Figura 22, mais completa, mostra também os estados dos arcos identificados por e_a e os estados dos processos por e_p. Na inicialização das estruturas de dados, o estado dos arcos é *inativo* (representado por $e_a=i$) ou *aberto* (representado por $e_a=a$) e o estado dos processos é *inicial* (representado por $e_p=i$).

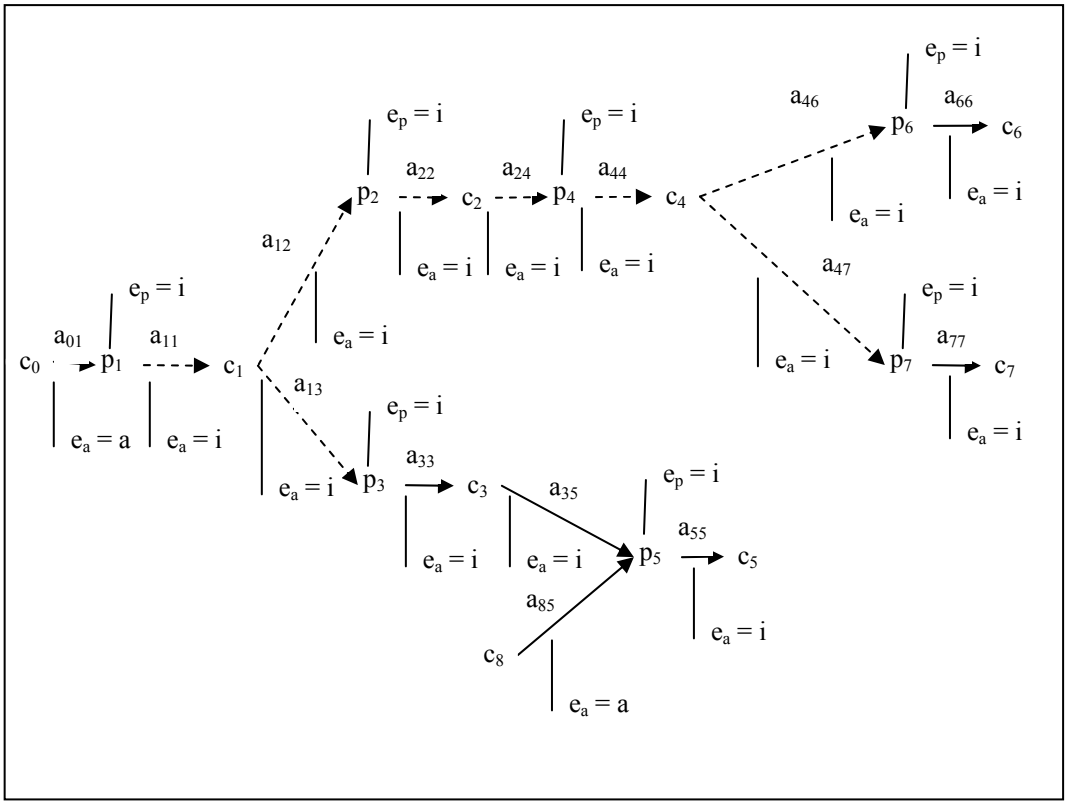

Figura 22. Exemplo: grafo bipartido após inicialização.

Suponha a existência de um processo fictício p_0 que esteja inicialmente executando e, portanto, no estado *executando*. Após a finalização de p_0, seu estado passa a ser *finalizado,* o estado do arco a_{00} que associa p_0 a c_0 passa a ser *aberto,* e o estado do arco a_{01} também passa a ser *aberto*. Observe que p_5 possui um contêiner c_8 que armazena dados do disco, não produzidos por processo do workflow. Este contêiner inicia seu estado como *aberto*, e possui os atributos *tamanho* e *tamanho_máximo* configurados como zero, e o atributo *temporário* como *falso*.

Suponha que *k* seja o espaço livre em disco para a execução do workflow. O algoritmo obtém os próximos processos a disparar através da execução do método *obtém_próximos_processos*.

Primeiro constrói-se o *estágio_ideal*. Os processos que estão liberados para serem disparados são os da lista *lista_processos_liberados* = {p_1, p_2, p_3, p_4, p_6, p_7}. Somente p_5 não pode ser disparado, pois ele precisa que p_3 produza todos os seus dados, para que ele possa consumí-los de forma não-gradativa (arco a_{35} do

tipo *não-gradativo* e estado atual igual a *inativo*). O método *constrói_estágio_ideal* constrói o componente conexo que representa o *estágio_ideal*, ilustrado na Figura 23.

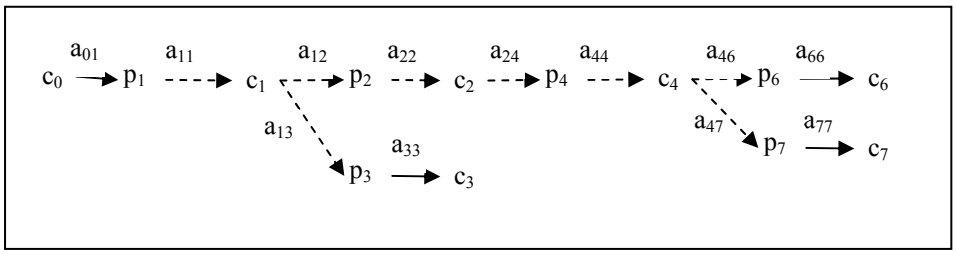

Figura 23. Exemplo: *estágio_ideal* 1.

Em seguida, o gerente de execução constrói o *estágio_real*. Suponha que o tamanho do *estágio_ideal, k',* seja maior do que *k*. Neste caso, é necessário iniciar o processo de poda. No primeiro passo, o *estágio_real* é uma cópia do *estágio_ideal*. Os nós-contêiner da fronteira do *estágio_real* corrente são *l_contêineres_fronteira* = {c_3, c_6, c_7}.

O método *calcula_poda* chama o método *calcula_ganho(c_3), calcula_ganho(c_6)* e *calcula_ganho(c_7)*. Suponha que o ganho da retirada de c_3 seja g_3, da retirada de c_6 seja g_6, da retirada de c_7 seja g_7, e que $g_7 < g_6 < g_3$. Neste caso, opta-se pela poda de c_7. Para ilustrar o cálculo do ganho, veja o método *calcula_ganho(c_7)*. A lista de processos (*l_processos_podar*) que devem ser podados inicia-se como {p_7}, já que p_7 é o único que produz dados em c_7. Como p_7 não produz mais dados em nenhum outro contêiner, nenhum outro nó-processo é acrescentado à lista *l_processos_podar*. A lista de nós-contêiner a podar (*l_contêineres_podar*) fica apenas com {c_7}. Ao retirar este nó-processo e este nó-contêiner, o nó-contêiner c_4, que tinha seus dados consumidos por p_7 de forma não-gradativa (arco a_{47} é do tipo *gradativo*), terá que materializar todos os seus dados para que p_7 possa ser iniciado em outra etapa da execução. Neste caso, a lista *l_contêineres_mudar_tamanho* fica igual a {c_4}.

O ganho de espaço ao retirar c_7 é o tamanho corrente de c_7 (atributo *tamanho* de c_7). A perda representa os dados que terão que ser armazenados a mais em c_4, ou seja, a diferença entre o valor do atributo *tamanho_máximo* e o valor do atributo *tamanho* de c_4. O ganho total é a diferença entre o ganho ao retirar c_7 e a perda pelo armazenamento extra de c_4.

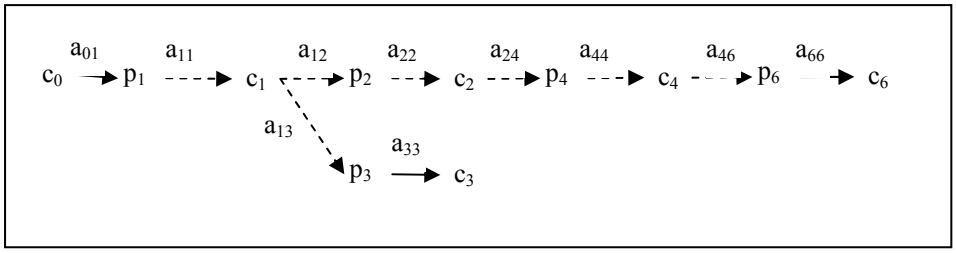

Figura 24. Exemplo: *estágio_real* 1.

Faz-se então a poda de c7 e o *estágio_real* passa a ser o apresentado na Figura 24.

Suponha que ainda não seja possível executar o *estágio_real* calculado. Neste caso, repete-se o procedimento com a nova lista de contêineres da fronteira, que é $\{c_3, c_6\}$. Suponha que o ganho na retirada de c_3 seja g_3 e na retirada de c_6 seja g_6, e que $g_6 < g_3$. Neste caso, opta-se pela poda de c_6. No método *calcula_ganho*(c_6), a lista de processos (*l_processos_podar*) é $\{p_6\}$, e a lista de contêineres a podar (*l_contêineres_podar*) é $\{c_6\}$. Ao retirar este nó-processo e este nó-contêiner, o nó-contêiner c_4 não terá nenhum armazenamento extra. Com a poda anterior de p_7, o nó-contêiner c_4 já teve que materializar todos os seus dados para que p_7 pudesse ser iniciado em outra etapa da execução. Com isso, o tamanho de c_4 passou a ser igual ao seu *tamanho_máximo*. Sendo assim, a perda que seria a diferença entre o valor do atributo *tamanho_máximo* e o valor do atributo *tamanho* de c_4 é igual a zero. O ganho total é o ganho ao retirar c_6, já que não há perda pelo armazenamento extra de c_4.

Faz-se então a poda de c_6 e o *estágio_real* passa a ser o apresentado na Figura 25.

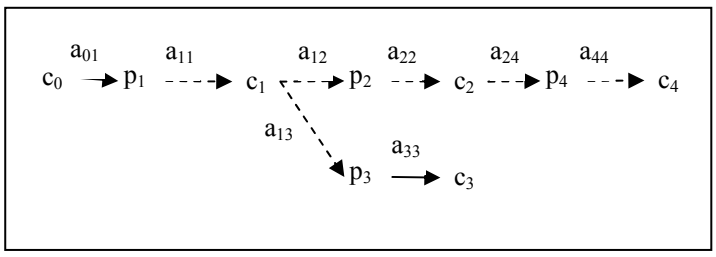

Figura 25. Exemplo: *estágio_real* 2.

Suponha que este *estágio_real* ainda é podado mais uma vez (retirada de p_3 e c_3) e torna-se viável de ser executado. Neste caso, a Figura 26 apresenta e

estágio_real finalizado e a Figura 27 ilustra o cenário da execução neste momento.

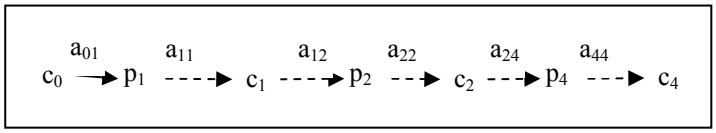

Figura 26. Exemplo: *estágio_real* 3.

Após o disparo de p_1, p_2 e p_4, os estados dos arcos e dos processos são atualizados, como mostra a Figura 27. Alguns nós-processo passaram de estado *inicial* para *executando* (representado por e_p=*e*) ou *pendente* (representado por e_p=*p*) e alguns arcos passaram de estado *inativo* para *aberto* (representado por e_a=*a*).

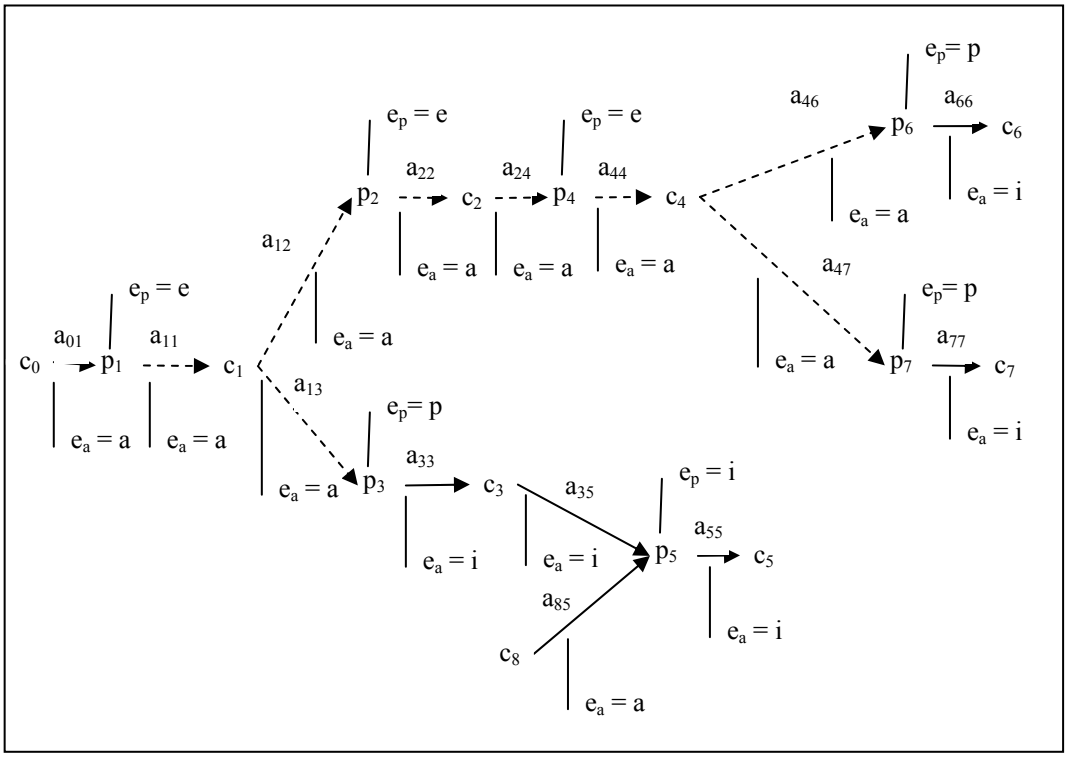

Figura 27. Exemplo: grafo bipartido do workflow no cenário 1.

Suponha que p_1 termine sua execução. Neste momento, o estado de p_1 passa a ser *finalizado* e os estados dos arcos a_{01} e a_{11} passam a ser *fechado*.

Na construção do *estágio_ideal*, a *lista_processos_liberados* é igual a {p_3, p_6, p_7}. O *estágio_ideal* é composto por dois componentes conexos, ilustrados na Figura 28.

Continua-se o algoritmo, a partir daqui, repetindo-se as chamadas dos métodos já ilustrados anteriormente.

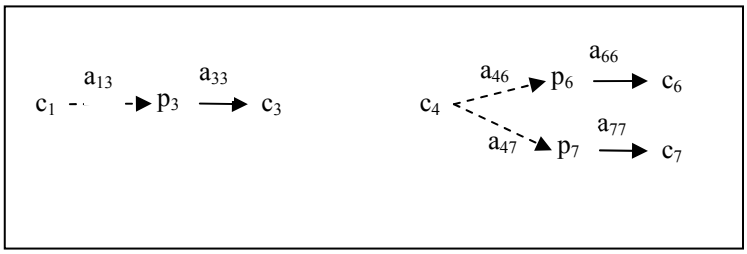

Figura 28. Exemplo: *estágio_ideal* 2.

6.4 SGWBio no Ambiente de Laboratório

Esta seção discute características da implementação do SGWBio no ambiente de trabalho de laboratório.

Neste ambiente, suporemos que:

1. Todos os módulos do SGWBio executarão em um único processador, como ilustrado na Figura 29.
2. Cada programa de Bioinformática está instalado em um ou mais (mas não necessariamente em todos) processadores que compõem o ambiente.
3. Para todo processador, o acesso a dados armazenados em outros processadores (dados remotos) é tão rápido quanto o acesso aos dados locais.

O assistente poderá ser implementado de forma a ser acessado apenas pela máquina que tem o SGWBio instalado, ou também por outras máquinas via rede, como representado na Figura 29.

A otimização via pipelining e formas de implementação de contêineres, consideradas no ambiente de trabalho pessoal, continuam sendo aplicáveis a este tipo de arquitetura. Entretanto, devido a características específicas deste ambiente, o algoritmo de gerência de execução, definido para o ambiente pessoal, deve ser adaptado através da extensão e criação de novos métodos e novas formas de

implementação de contêineres. Além disso, outras otimizações também podem ser aplicadas neste ambiente. As próximas seções apresentam as adaptações e os outros tipos de otimizações.

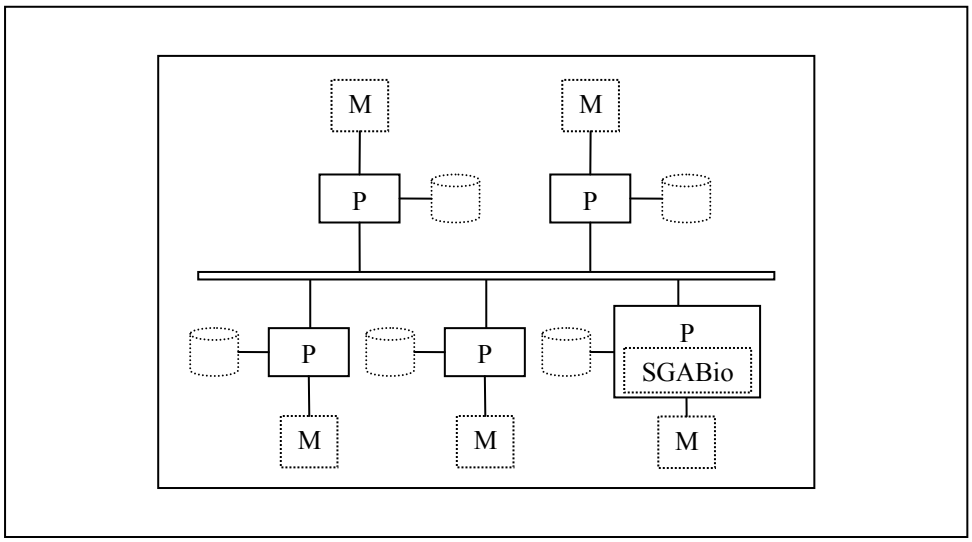

Figura 29. SGWBio em um ambiente de trabalho laboratório.

6.4.1 Implementação de Contêineres

No ambiente pessoal, argumentou-se que os nós-contêiner poderiam ser implementados de acordo com a Tabela 13, ou seja como um *BufferLimitado*, *BufferIlimitado*, *Arquivo*, *ArquivoBufferLimitado* e *ArquivoBufferIlimitado*.

Foi descrito que um *BufferLimitado* e um *BufferIlimitado* oferecem métodos para leitura e gravação de itens de dados semelhantes aos de uma fila, exceto que não há ordem entre os itens de dados, já que não é essencial para os processos de Bioinformática. Um item de dados é removido quando for lido por todas as conexões de leitura, e um item de dados é liberado para leitura imediatamente após ter sido gravado. Estas características também se aplicam ao *ArquivoBufferLimitado* e *ArquivoBufferIlimitado*.

No ambiente de laboratório, existirão casos (definidos a seguir) onde um item de dados é removido assim que for lido por alguma conexão de leitura (diferentemente dos casos anteriores), e é liberado para leitura imediatamente após ter sido gravado.

Existem duas políticas de consumo dos dados destes contêineres: *round robin* e *sob demanda*. Na política *round robin*, os dados são distribuídos para os

processos consumidores de forma cíclica. Na política *sob demanda*, cada processo consumidor, ao terminar de processar um dado de entrada, retira o próximo dado do contêiner.

Desta forma, os nós-contêiner também poderão ser implementados como *ChaveamentoLimitado*, *ChaveamentoIlimitado*, *ArquivoChaveamentoLimitado* e *ArquivoChaveamentoIlimitado*. A palavra chaveamento indica o momento da leitura de um dado do contêiner por um processo.

6.4.2 Otimização Dinâmica por Pipelining

Esta seção discute como o algoritmo de gerência de execução, apresentado para o ambiente pessoal, pode ser adaptado para o ambiente de laboratório.

Como no ambiente pessoal, o algoritmo recebe o documento de especificação do workflow definido da forma explicada anteriormente, constrói o grafo bipartido do workflow e inicializa os atributos dos objetos que compõem o grafo, ou seja, os nós-processo, nós-contêiner e arcos. Feito isso, o gerente de execução aguarda que um nó-processo termine sua execução. Após o término de um nó-processo, calcula-se o *estágio_ideal* e o *estágio_real*, ou seja, determina-se o será possível executar. Este enfoque é idêntico ao do ambiente pessoal. O *estágio_ideal* continua sendo construído baseado nos nós-processo que estão liberados para serem executados segundo suas entradas, mantendo as estratégias de pipelining e de compartilhamento de contêineres. Portanto, mantém-se a estratégia de que, quanto maior for o número de nós-processo em pipelining, menor a possibilidade de se armazenar resultados temporários e, consequentemente, menor o gasto de espaço. A diferença encontra-se no método de obtenção do *estágio_real*.

Diferentemente do ambiente de trabalho pessoal, no ambiente de laboratório os programas de Bioinformática, dados e recursos estão disponíveis e, mesmo replicados, em diversos processadores. Desta forma, após a definição do *estágio_ideal* (como no ambiente pessoal), é necessário descobrir qual(is) processador(es) serão responsáveis pela execução de um processo e pela disponibilização dos dados e recursos. A alocação dos processos aos processadores é uma tarefa que depende de alguns fatores.

Os processadores possuem propriedades que os diferenciam quanto à capacidade, ocupação, programas de Bioinformática instalados e dados disponíveis. Como capacidade de uma máquina, tem-se fatores fixos, como velocidade de CPU, tamanho de disco e de memória principal. Quanto à ocupação da máquina, tem-se os fatores que dependem do estado da máquina em um instante particular, como o consumo de memória (principal e virtual), os processos em execução, o consumo de CPU, a utilização do espaço em disco e a quantidade de operações de I/O.

Os programas de Bioinformática não estão necessariamente instalados em todos os processadores que compõem o ambiente. Além disso, os seus dados de entrada e recursos podem estar espalhados nos diversos discos do ambiente.

Outra característica importante deste ambiente é a forma de acesso aos dados (entrada, saída ou recurso) feita pelos processos. Alguns processos necessitam ler ou gerar dados no disco local do processador em que estão sendo executados. Por exemplo, alguns processos paginam dados entre a memória principal e o disco local. O acesso aos dados nesta situação é chamado de *acesso local*.

Por outro lado, existem processos que lêem ou geram dados em qualquer disco remoto como se fosse em um disco local. Neste caso, os discos presentes no ambiente de laboratório podem ser vistos como um único disco, formando um único sistema de arquivos. O acesso aos dados nesta situação é chamado de *acesso global*.

No *grafo_bipartido_workflow,* o acesso aos dados feito pelos nós-processo é representado pelos arcos e, assim, no ambiente de laboratório, o arco possui ainda o atributo *acesso*, que pode possuir os valores *local* ou *global*.

Todos estes fatores devem ser utilizados para a alocação de um processo a um processador existente no ambiente, o que é feito durante a construção do *estágio_real*, conforme será apresentado e justificado a seguir.

O algoritmo de gerência de execução recebe como entrada o documento de especificação do workflow do workflow e inicializa as mesmas estruturas de dados básicas do ambiente pessoal e as estruturas de dados representantes dos processadores, como definido a seguir. Conforme já foi dito, no ambiente de laboratório, o arco possui a mais o atributo *acesso*, destacado no quadro abaixo.

Estruturas de Dados Básica

- grafo_bipartido_workflow
  - nó-processo
  - nó-contêiner
  - arco(nó-processo, nó-contêiner)
    - ... (mesmos atributos definidos no ambiente pessoal)
    - **acesso**: forma de acesso ao nó-contêiner feita pelo nó-processo (local, global)
  - arco(nó-contêiner, nó-processo)
    - ... (mesmos atributos definidos no ambiente pessoal)
    - **acesso**: forma de acesso ao nó-contêiner feita pelo nó-processo (local, global)
- processador: objeto representante de um processador do ambiente de laboratório
  - identificação: identificação do processador.
  - k_i: espaço livre em disco local para a execução do workflow em um determinado instante, sendo $k_i \geq 0$. É atualizado durante a execução do workflow, dependendo dos espaços utilizados pelos contêineres que possui alocado.
- processadores_estado_corrente: vetor que armazena a situação corrente de alocação dos nós-processo e nós-contêiner aos processadores. Cada posição armazena os seguintes objetos:
  - m: processador (ou máquina).
  - lista_processos: lista de nós-processo p em execução em m.
  - lista_contêineres_acesso_local: lista de nós-contêiner c que possuem acesso local pelos nós-processo p alocados à m.
  - lista_contêineres_acesso_global: lista de nós-contêiner c que possuem acesso global pelos nós_processo alocados a qualquer processador do ambiente.
- K: espaço total livre em todos os discos dos processadores do ambiente de laboratório.
- processadores_estado_próximo: vetor com a mesma estrutura do vetor processadores_estado_corrente, e que gerencia a alocação dos nós-processo e nós-contêiner aos processadores em um determinado instante da execução.

Como foi dito anteriormente, a criação do *estágio_ideal* é feita da mesma forma que no ambiente pessoal, sendo aqui omitida. Entretanto, a obtenção do *estágio_real* é feita de uma maneira particular a este ambiente.

No ambiente pessoal, existia um único processador e o algoritmo tratava da limitação de espaço em disco deste processador. Isto era feito através da poda dos nós-processo que se localizavam na fronteira do *estágio_real*, até que os nós-processo restantes pudessem ser executados concorrentemente.

No ambiente de laboratório, esta tarefa é estendida pois existem vários processadores, cada um com sua limitação de espaço em disco e com suas características próprias. Desta forma, o algoritmo necessita escolher um processador para cada processo pertencente ao *estágio_ideal*. Caso isso não seja possível, o algoritmo podará os nós-processo e os nós-contêiner do *estágio_ideal* até que ele se torne viável para ser executado. A alocação de um processo a um processador não deve ser feita caminhando-se nos componentes conexos a partir de seus nós localizados nas fronteiras (sorvedouros) até os seus nós iniciais (ou seja, as suas fontes, ou *sources*, em inglês). Caso esta estratégia fosse adotada, há risco de não existirem processadores disponíveis para alocar os nós-processo iniciais.

A estratégia da construção do *estágio_real* no ambiente de laboratório é baseada no caminhamento no grafo a partir dos seus nós iniciais até os seus nós de fronteira. Obtém-se os nós-processo do primeiro nível dos componentes conexos do *estágio_real*, e calcula-se os espaços necessários para a execução de cada um deles. Este espaço é calculado pelo método *calcula_espaço*, que faz a soma dos espaços necessários para o armazenamento dos dados dos contêineres acessados de forma local e global por um processo p.

Cria-se uma fila com estes nós-processo ordenados de forma crescente de acordo com o espaço calculado. Enquanto houver espaço livre em algum processador do ambiente, tenta-se alocar o primeiro nó-processo da fila a algum processador, conforme apresentado no método *lab_constrói_estágio_real*.

Método lab_constrói_estágio_real

Definição

- Cálculo de estágio_real.

Entrada

- estágio_ideal

Saída

- estágio_real
- processadores_estado_corrente atualizado.

Método

- estágio_real ← estágio_ideal
- processadores_estado_próximo ← processadores_estado_corrente
- Forme uma lista l com pares (nó-processo, espaço) com os nós-processo localizados no primeiro nível dos componentes conexos de estágio_real. Atribua o valor zero a variável espaço de cada par.
- Para cada par (p, esp_p) ∈ l
    - esp_p ← calcula_espaço(p)
- Ordene l de forma crescente de acordo com esp_p.
- Para cada par (p, esp_p) ∈ l
    - Retire o par de l
    - Enquanto K ≥ 0
        - aloca_processos_pipelining(p)
- processadores_estado_corrente ← processadores_estado_próximo
- Retorne estágio_real e processadores_estado_corrente atualizado.

O método *aloca_processos_pipelining* trata da alocação de um determinado processo *p* e de todos os processos p_i que consomem, de forma gradativa, dados gerados por *p*, e assim recursivamente. Ou seja, *aloca_processos_pipelining* trata da alocação de todos os processos que formam um pipelining a partir de p.

Método aloca_processos_pipelining

Definição

- Alocação de um nó-processo p a um processador e de seus nós-contêiner e nós-processo associados de forma direta ou indireta, via pipelining.

Entrada

- p: nó-processo

Saída

- processadores_estado_próximo atualizado.

Método

- $l \leftarrow \{p\}$
- Para cada nó-processo p_i de l
    - Forme uma lista com os nós-contêiner c_i que p_i armazena dados, ou seja, c_i tal que exista arco (p_i,c_i)
    - Para cada nó-processo p_j que lê dados de c_i
        - $l \leftarrow l \cup \{p_j\}$
- Para cada nó-processo p_i de l
    - aloca_ou_poda_processo (p_i)
- Retorne.

A alocação de cada processo *p* é feita pelo método *aloca_ou_poda_processo*. Inicialmente, tenta-se alocar processadores para os nós-contêiner que *p* acessa de forma global através do método *aloca_contêineres_globais*. Sendo possível, tenta-se alocar um processador para *p* e para os nós-contêiner que *p* acessa de forma local através do método *aloca_contêineres_locais*.

Método aloca_ou_poda_processo

Definição

- Alocação de um processador para o nó-processo p e seus nós-contêiner acessados de forma local e alocação de processadores para os nós-contêiner acessados de forma global. Caso não seja possível alocar, poda o estágio_ideal.

Entrada

- p: nó-processo

Saída

- processadores_estado_próximo atualizado.

Método

- conseguiu ← aloca_contêineres_globais(p)
- Se conseguiu = verdadeiro então
    - conseguiu ← aloca_contêineres_locais(p)
- Forme lista l_c com todos os nós-contêiner c_i que armazenam dados de entrada de p, ou seja c_i tal que exista arco (c_i, p)

- Se conseguiu = verdadeiro então
    - libera_espaço_extra(l_c)
- Senão
    - lab_poda_estágio_real(l_c)
- Retorne

O método *aloca_contêineres_globais* cria uma lista com todos os nós-contêiner que o processo *p* armazena dados de saída de forma global. Para cada um deles é necessário alocar um processador que possa armazená-lo, tarefa feita pelo método *encontra_processador_contêiner_global*.

Não é relevante considerar os nós-contêiner com dados de entrada acessados de forma global, já que no momento da execução deste método, estes nós-contêiner já foram alocados a algum processador e *p* acessa-os de forma global.

Método aloca_contêineres_globais

Definição

- Alocação de processadores para os nós-contêiner acessados de forma global.

Entrada

- p: nó-processo

Saída

- processadores_estado_próximo atualizado
- variável booleana conseguiu indicando verdadeiro quando foi possível alocar contêineres.

Método

- Forme uma lista l com os arcos a_i (p,c_i) que conectam p a nós-contêiner c_i, tal que a_i.acesso = global // arcos que representam saídas de p, pois as entradas globais de p já foram alocadas anteriormente e não precisam ser alocadas novamente
- Para cada a_i de l
    - conseguiu ← verdadeiro
    - Enquanto conseguiu = verdadeiro
        - esp ← c_i.tamanho_máximo
        - conseguiu ← encontra_processador_contêiner_global(p, c_i, esp)
- Retorne conseguiu

O método *aloca_contêineres_locais* cria uma lista com todos os nós-contêiner que o processo *p* acessa (para leitura ou geração de dados) de forma local, calcula o espaço necessário para armazenar estes nós-contêiner, e chama o método *encontra_processador_contêineres_locais*. Este método é responsável pela alocação de um processador para *p* e para todos os seus nós-contêiner acessados de forma local.

Diferente dos nós-contêiner de entrada acessados de forma global, os dados de entrada acessados de forma local devem ser considerados. No momento da execução deste método, estes nós-contêiner já foram alocados a algum processador, mas eles precisarão ser alocados ao mesmo processador em que *p* for alocado. Como, neste ambiente, está sendo considerado que o acesso ao disco e a transferência de dados entre os discos não possui um custo alto, em comparação com o tempo de execução dos nós-processo, a estratégia apresentada aqui opta, sempre que possível, pela alocação de um nó-processo a um processador que ainda não está sendo utilizado por nenhum nó-processo. Desta forma, mesmo em um pipelining, os nós-processo serão alocados preferencialmente em processadores distintos, mesmo que isto signifique replicar nós-contêiner em diversos processadores. Entretanto, sempre que um nó-contêiner não tiver mais sendo acessado, seus dados poderão ser liberados, assim como no ambiente pessoal.

As estimativas dos tamanhos dos nós-contêiner nos métodos *aloca_contêineres_globais* e *aloca_contêineres_locais* são feitas de forma cautelosa, pois considera-se que é necessário encontrar um processador que seja capaz de armazenar todos os nós-contêiner, que armazenam dados gerados por p, com seu tamanho máximo. Isto é assumido mesmo para os nós-contêiner que terão seus dados consumidos apenas por nós-processo de forma gradativa e que, portanto, não necessitariam ser alocados a um processador com capacidade de armazenamento para o seu tamanho máximo, mas somente para o seu tamanho mínimo.

Uma outra forma de se fazer o cálculo é através de uma estratégia otimista, que considere o tamanho mínimo dos nós-contêiner c_is para o caso em que eles fossem consumidos apenas por nós-processo de forma gradativa. Neste caso, como os nós-processo p_is, consumidores de c_is, são analisados depois da alocação dos c_is, é possível que não existam processadores para alocá-los, e

consequentemente, eles terão que ser podados. Nesta situação, será necessário recalcular as alocações já definidas, pois $c_i s$ terão que armazenar mais dados que o seu tamanho mínimo, já que estes dados serão consumidos em uma outra etapa da execução pelos nós-processo $p_i s$ que foram podados.

No caso da estratégia cautelosa, caso nenhum processo consumidor de um nó-contêiner c_i tenha sido podado, é possível liberar o espaço extra alocado no processador para armazenar c_i.

Como na estratégia otimista, o procedimento para recalcular as alocações já definidas não é uma tarefa trivial. Como na estratégia cautelosa é possível liberar espaços extras alocados (apresentado no método *libera_espaço_extra*), optou-se pela utilização desta estratégia.

Método aloca_contêineres_locais

Definição

- Alocação de um processador para o nó-processo p e seus nós-contêiner acessados de forma local.

Entrada

- p: nó-processo

Saída

- processadores_estado_próximo atualizado.
- variável booleana conseguiu indicando verdadeiro quando foi possível alocar contêineres e processo.

Método

- Forme uma lista l com os arcos a_i (p,c_i) ou (c_i,p) que conectam p a nós-contêiner c_i e vice-versa , tal que a_i.acesso = local
- esp ← 0
- l_c_i ← {}
- Para cada a_i de l
    - l_c_i ← l_c_i ∪ $\{c_i\}$
    - Se a_i for (c_i, p) // arco que representa entrada de dados de p
        - esp ← esp + c_i.tamanho
    - Senão // a_i é (p,c_i): arco que representa saída de p
        - esp ← esp + c_i.tamanho_máximo
- conseguiu ← encontra_processador_contêineres_locais(p, l_c_i, esp)
- Retorne conseguiu

Os métodos *encontra_processador_contêiner_global* e *encontra_processador_contêineres_locais* são responsáveis pela alocação de processador.

No algoritmo de execução do workflow no ambiente pessoal existia uma variável *k* que controlava o espaço em disco da única máquina do ambiente. No ambiente de laboratório, cada processador m_i possui seu espaço livre representado por k_i, sendo denominado *K* o somatório de todos os k_i. Para alocar um nó-processo p_i a um processador m_i é necessário que m_i tenha instalado o programa que o processo p_i executa e que haja espaço suficiente para armazenar os dados dos nós-contêiner que p_i acessa de forma local. Esta alocação é feita pelo método *encontra_processador_contêineres_locais*. Além disso, é necessário que existam outros processadores no ambiente que tenham espaço para armazenar os dados dos nós-contêiner que p_i acessa de forma global, tarefa do método *encontra_processador_contêiner_global*.

Quando mais de um processador atende a estes requisitos para alocar nós-contêiner (local ou global), é necessário escolher o processador mais adequado segundo alguma estratégia. Uma possível estratégia seria baseada na capacidade do processador, outra seria baseada no espaço (por exemplo, escolher o processador que tem o menor espaço necessário para armazenar os nós-contêiner), e uma mais complexa, poderia considerar a combinação dos dois fatores anteriores. Em todas as estratégias, neste ambiente, será dada prioridade aos processadores que não possuem algum nó-processo alocado ou que possuem nós-processo com baixa complexidade de tempo de execução.

É interessante notar que, no caso dos nós-contêiner com dados gerados de forma global por seus nós-processo, eles podem ser re-alocados durante a construção do *estágio_real*, caso seja necessário liberar espaço local no processador em que ele foi alocado. Isto é possível porque o local de armazenamento deles não é relevante para os nós-processo que consomem seus dados.

Ao alocar um nó-contêiner c_i (com acesso local ou global) a um processador m_i, é necessário atualizar k_i de acordo com o tamanho de c_i.

Retornando ao método *aloca_ou_poda_processo*, caso tenha sido feita a alocação de um nó-processo, este método chama o método *libera_espaço_extra*

para, se possível, liberar os espaços a mais que foram alocados aos nós-contêiner de forma cautelosa.

Método libera_espaço_extra

Definição

- Libera espaço extra alocado aos nós-contêiner devido a estratégia cautelosa de alocação de nós-processo e nós-contêiner a processadores.

Entrada

- lista_contêineres: lista de nós-contêiner cujos espaços serão verificados para saber se é possível liberar espaço extra.

Saída

- Espaço liberado.

Método

- Para cada c_i de lista_contêineres
  - Forme lista l_p com todos os nós-processo p_i que lêem dados de c_i, ou seja p_i tal que exista arco $a_i(c_i,p)$
  - todos_consumidores_alocados ← verdadeiro
  - Para cada p_i de l_p
    - Verifique em processadores_estado_próximo se p_i está alocado a algum processador. Se não tiver sido alocado então todos_consumidores_alocados ← falso.
  - Se todos_consumidores_alocados = verdadeiro
    - Atualize o tamanho necessário para alocação de c_i para seu tamanho_mínimo em processadores_estado_próximo. Isto liberará um espaço no processador m_i em que c_i foi alocado de forma cautelosa, ou seja, com seu tamanho máximo.
- Retorne.

Caso não seja possível fazer a alocação de um processo *p*, o método *aloca_ou_poda_processo* chama o método *lab_poda_estágio_real*, que chama o método *calcula_poda* para cada um dos nós-contêiner de *p*.

O método *calcula_poda* está baseado na mesma idéia do método *calcula_ganho* apresentado no ambiente pessoal. Ele retorna os nós-processo (lista *l_processos_podar*) e os nós-contêiner (lista *l_contêineres_podar*) que terão que ser podados ao se retirar *p*, e os nós-contêiner que terão seus tamanhos modificados (lista *l_contêineres_mudar_tamanho*), pois, com a poda, terão que

materializar mais dados para que os nós-processo que tiveram sua execução postergada tenham acesso aos seus dados de entrada em outra etapa da execução.

No fim da iteração feita em *lab_poda_estágio_real*, tem-se todos os nós-contêiner e nós-processo que terão que ser podados do *estágio_real*. Chama-se então o método *poda_estágio_real*, idêntico ao apresentado no ambiente pessoal.

Método lab_poda_estágio_real

Definição

- Gerência da poda de nós-contêiner e nós-processo do ambiente de laboratório.

Entrada

- lista_contêineres: lista de nós-contêiner que devem ser podados.

Saída

- estágio-real podado.

Método

- l_processos_podar ← {}
- l_contêineres_podar ← {}
- l_contêineres_mudar_tamanho ← {}
- Para cada c_i de l_c
    - l_processos_podar_2, l_contêineres_podar_2, l_contêineres_mudar_tamanho_2 ← calcula_poda(c_i)
    - l_processos_podar ← l_processos_podar ∪ l_processos_podar_2
    - l_contêineres_podar ← l_contêineres_podar ∪ l_contêineres_podar_2
    - l_contêineres_mudar_tamanho ← l_contêineres_mudar_tamanho ∪ l_contêineres_mudar_tamanho_2
- poda_estágio_real (l_processos_podar, l_contêineres_podar, l_contêineres_mudar_tamanho);
- Retorne .

Método calcula_poda

Definição

- Cálculo dos nós-contêiner e nós-processo que devem ser retirados do estágio-real ao se decidir pela retirada de um contêiner dos componentes conexos do estágio-real.

Entrada

- c: contêiner que se deseja retirar do componente conexo

Saída

- l_processos_podar: lista de nós-processo que é necessário podar do estágio_real ao se retirar o contêiner c

- l_contêineres_podar: lista de nós-contêiner que é necessário podar do estágio_real ao se retirar o contêiner c
- l_contêineres_mudar_tamanho: lista de nós-contêiner que terão seus tamanhos alterados se for feita a poda dos elementos de l_processos_podar e l_contêineres_podar do estágio_real.

Método

- Forme uma lista l_processos_podar com todos os nós-processo que produzem dados em c
- Para cada processo p de l_processos_podar:
    - Forme uma lista l_contêineres de todos os contêineres $c_i \neq c$ que p produz dados:
        - Para cada c_i de l_contêineres
            - Forme uma lista lp_1 de processos $p_i \neq p$ que produzem dados em c_i
            - Forme uma lista lp_2 de processos $p_i \neq p$ que consomem dados em c_i
            - l_processos_podar $\leftarrow$ l_processos_podar $\cup$ lp_1 $\cup$ lp_2
- Forme uma lista l_contêineres_podar com todos os nós-contêiner que os processos p $\in$ l_processos_podar produzem dados
- Forme uma lista l_contêineres_mudar_tamanho com todos os nós-contêiner que os processos p $\in$ l_processos_podar consomem dados,
- Retire contêineres de l_contêineres_mudar_tamanho que estão em l_contêineres_podar
- Retorne (l_processos_podar, l_contêineres_podar, l_contêineres_mudar_tamanho)

6.4.3 Otimização Dinâmica por Paralelização

A otimização proposta até agora para o ambiente de laboratório baseia-se na execução concorrente de processos através de pipelining de dados e de compartilhamento de contêineres.

Como no ambiente pessoal, pipelining reduz o número de arquivos temporários e o tempo de apresentação dos resultados intermediários do workflow ao pesquisador. Já no ambiente de laboratório, esta estratégia também reduz o tempo de execução do workflow. Sobre o compartilhamento dos contêineres, a vantagem continua sendo, como no ambiente pessoal, evitar a replicação desnecessária dos dados.

Além deste tipo de otimização, é possível adotar outras estratégias no ambiente de laboratório, como o de distribuição de dados de entrada e a fragmentação horizontal do banco de dados. Estas duas estratégias, discutidas a seguir, diminuem o tempo de execução do workflow. Elas são implementadas por variantes do método *encontra_processador_contêineres_locais*.

Como foi dito anteriormente, para alocar um nó-processo *p* é necessário que:

- os nós-contêiner de entrada acessados de forma global (c_{eg}) já estejam alocados a algum processador;
- os nós-contêiner de entrada acessados de forma local (c_{el}) sejam alocados ao mesmo processador que *p*;
- os nós-contêiner de saída acessados de forma local (c_{sl}) sejam alocados ao mesmo processador que p; e
- os nós-contêiner de saída acessados de forma global (c_{sg}) sejam alocados a algum processador.

De acordo com a estratégia de alocação de processadores apresentada anteriormente, os c_{eg} e c_{sg} são os mais flexíveis, ou seja, eles podem ser alocados a qualquer processador. Já os c_{el} e c_{sl} devem ser alocados ao mesmo processador em que *p* executa, o que restringe o número de processadores capacitados. Além disso, a estratégia cautelosa restringe ainda mais este número.

Distribuição de Dados de Entrada

A existência de um arco conectando *p* a c_{sl} do tipo *gradativo* indica que *p* gera resultados independentes com relação aos dados de entrada. Nesta situação, se *p* analisa seus dados de entrada em c_{el} ou c_{eg} também de forma independente, ou seja, se *p* lê seus dados de entrada por arcos do tipo *gradativo*, é possível alocar cópias de *p* a vários processadores, sendo que cada uma lê uma parte dos dados de entrada e, consequentemente, gera uma parte dos dados de saída. Este procedimento diminui o espaço necessário para um processador executar uma cópia de *p* pois, apesar dos c_{el} ou c_{eg} (lidos de forma gradativa) terem que ser alocados com o mesmo tamanho em qualquer um dos processadores, os c_{sl} diminuirão de tamanho.

Como um exemplo, considere o workflow de genoma completo definido na Figura 1. É viável distribuir um conjunto de contigs para vários processadores e executar o Glimmer em cada um deles a fim de paralelizar a tarefa de predição de

genes. O resultado final é simplesmente a união dos resultados locais. Isto é possível porque o Glimmer lê os contigs que estão nos nós-contêiner de entrada de forma gradativa e gera os seus resultados (possíveis genes) de forma gradativa. O nó-contêiner que armazena os contigs deve ser alocado com o seu tamanho mínimo, ou seja, o tamanho do maior contig, independente se for usado um ou vários processadores. Cada um deles precisa ter este espaço disponível. O nó-contêiner que armazena os possíveis genes terá tamanho diferente, dependendo se um ou vários processadores forem selecionados. No caso de um único processador, o método cauteloso reservará espaço para armazenar todos os possíveis genes de todos os contigs, como se eles não fossem ser lidos por um outro nó-processo em pipelining. No caso de vários processadores, o método cauteloso continuará sendo aplicado, mas reservará espaço para armazenar todos os possíveis genes para um número reduzido de contigs. Quanto menor o número de contigs que a cópia do Glimmer executando em um processador tiver que analisar, menor o espaço que será reservado de forma cautelosa.

A execução do BLAST é similar. É possível distribuir um conjunto de sequências de entrada entre diferentes processadores e executar o BLAST para comparar cada sequência a um banco de dados de sequências, que deve ser replicado em cada processador [WU-BLAST, 2004b]. Os resultados das comparações são independentes e, portanto, é possível unir todos para apresentar ao usuário.

Em geral, analisando a semântica de cada processo, é possível decidir se é viável particionar um conjunto de dados de entrada e aplicar a cada partição uma cópia do processo, executando em um processador diferente. Se for viável, é necessário incluir processos de controle interno para particionar o conjunto de entrada, e processos de controle interno para combinar os resultados gerados. Ao decidir aplicar esta regra de otimização, o gerente de execução deve adicionar automaticamente tais processos de controle interno no workflow.

Uma outra forma de particionar um conjunto de dados de entrada seria implementar o contêiner com estes dados como *ChaveamentoLimitado*, *ChaveamentoIlimitado*, *ArquivoChaveamentoLimitado* ou *ArquivoChaveamentoIlimitado*, dependendo de seus tipos de conexões de entrada e saída.

O particionamento e distribuição dos dados de entrada podem ser feitos de diversas maneiras. O particionamento e distribuição *a priori* aplica-se quando os dados de entrada encontram-se em um contêiner implementado como um arquivo, ou seja, quando todos os dados de entrada existem antes do início da execução. Neste caso, os dados de entrada podem ser particionados e distribuídos entre as cópias do processo *p*, executando em processadores distintos, antes do início da execução propriamente dita.

Já o particionamento e distribuição *a posteriori* caracteriza-se por distribuir os dados de entrada às cópias do processo em tempo de execução. O particionamento e distribuição pode adotar uma política de *round robin,* ou seja, distribuindo os dados de entrada entre os processadores de forma cíclica. Pode ainda ser sob demanda, em que cada cópia do processo, ao terminar de processar um dado de entrada, retira o próximo do contêiner.

No caso da política de *round robin* distribui-se melhor o espaço gasto para armazenar os resultados, mas pode se perder em tempo de execução. Já se a distribuição for por demanda, pode-se ganhar em tempo de execução, mas é possível que o disco de um processador que possui velocidade de execução maior seja completamente preenchido, impedindo-o de executar.

Fragmentação Horizontal do Banco de Dados

No segundo caso, quando existe um arco que conecta c_{el} a *p* do tipo *não-gradativo*, geralmente c_{el} representa um banco de dados que será utilizado por *p* para analisar seus dados de entrada, localizados em um outro nó-contêiner. Este caso acontece em vários programas de Bioinformática que acessam banco de dados, como o BLAST, FAST e o HMMPFam. Todos estes programas comparam suas sequências de entrada a um determinado banco de dados.

Neste contexto é possível:

1. Fragmentar horizontalmente o banco de dados e distribuir os fragmentos entre os processadores que irão executar cópias de *p*.
2. Executar as cópias de *p* em cada processador, utilizando o fragmento local.
3. Mover os resultados locais para um processador central, que combinará os resultados locais para obter o resultado final.

Assim, como na estratégia anterior, é necessário incluir processos de controle interno para particionar o banco de dados, e processos de controle interno para combinar os resultados gerados. Ao decidir aplicar esta regra de otimização,

o gerente de execução deve adicionar automaticamente tais processos de controle interno no workflow. A ontologia deve conter a definição destes processos de controle interno.

Esta estratégia possui, como desvantagem, a necessidade de executar um procedimento adicional para construir o resultado final a partir da combinação dos resultados parciais. Como vantagem, possibilita a redução do tempo de execução do workflow.

Por exemplo, no caso do BLAST, cada sequência de entrada deve ser enviada para todos os *n* processadores. Cada cópia de processo BLAST deve realizar a comparação da sequência de entrada com o fragmento local do banco de dados. Os resultados parciais serão enviados para um processador central que deve executar um processo para intercalar os resultados parciais e formar o resultado final.

Uma estratégia de fragmentação horizontal do banco de dados, viável para ser utilizada com o BLAST, consiste em criar os fragmentos com aproximadamente o mesmo tamanho (número de bases ou resíduos), sendo a soma dos tamanhos de cada fragmento o tamanho total do banco de dados. Esta idéia está baseada na equação de complexidade de tempo do BLAST [Altschul, 1990], que mostra que, fixando-se a sequência de entrada e os parâmetros do BLAST, o tamanho do banco de dados é o principal fator de influência na velocidade de execução do BLAST. Considerando que os processadores da arquitetura possuam aproximadamente a mesma capacidade e ocupação, esta seria então uma boa estratégia de fragmentação horizontal do banco de dados para o caso do BLAST.

6.5 SGWBio no Ambiente de Comunidade

Esta seção discute características da implementação do SGWBio no ambiente de trabalho de comunidade.

Neste ambiente, suporemos que:

1. Todos os módulos do SGWBio executarão em um único processador, como ilustrado na Figura 30.
2. Cada programa de Bioinformática está instalado em um ou mais (mas não necessariamente em todos) processadores que compõem o ambiente.

3. Para todo processador, o acesso a dados armazenados em outros processadores (dados remotos) não é tão rápido quanto o acesso aos dados locais.
4. Existem bancos de dados cujo acesso é restrito, ou seja, cuja política de acesso não permite transferir seus dados para outros processadores do ambiente.

De acordo com a suposição 3, existe um custo para a transferência dos dados de um determinado nó-contêiner entre processadores. Este custo é definido como o tempo de transmissão de uma unidade de dados de um processador para outro, multiplicado pelo número de bytes a serem transmitidos, somado ao custo fixo de inicialização de uma mensagem, multiplicado pela quantidade de mensagens.

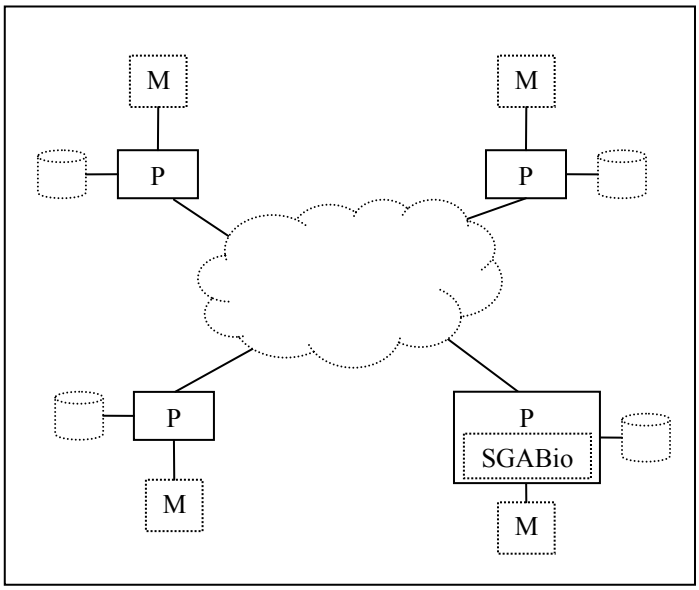

Figura 30. SGWBio em um ambiente de trabalho de comunidade.

O custo de inicialização é o intervalo de tempo máximo para que uma instância de processo em um determinado processador seja inicializado. Inclui o tempo necessário para a autenticação de usuários, verificação de permissões e inicialização dos itens de hardware e software necessários [Azevedo, 2003].

Este custo de inicialização se faz necessário porque o compartilhamento de recursos nesta arquitetura é condicional: cada processador, dono de um recurso, define restrições sobre quando, onde e o que pode ser feito com seu recurso. A implementação de tais restrições necessita de mecanismos para expressar políticas

para o estabelecimento de identidade para os consumidores dos recursos (autenticação) e para determinação se uma operação é consistente com os relacionamentos de compartilhamento (autorização).

O assistente poderá ser implementado de forma a ser acessado apenas pela máquina que tem o SGWBio instalado, ou também por outras máquinas via rede, como representado na Figura 30.

A otimização via pipelining, formas de implementação de contêineres, distribuição de dados de entrada e fragmentação dos bancos de dados, consideradas no ambiente de trabalho de laboratório, continuam sendo aplicáveis a este tipo de arquitetura. Entretanto, devido a características específicas deste ambiente, o algoritmo de gerência de execução, definido para o ambiente pessoal e laboratório, deve ser adaptado através da extensão e criação de métodos e formas de implementação de contêineres. As próximas seções apresentam estas adaptações.

6.5.1 Implementação de Contêineres

No ambiente pessoal, argumentou-se que os nós-contêiner poderiam ser implementados de acordo com a Tabela 13.

Como foi descrito anteriormente, o nó-contêiner, no ambiente pessoal, pode ser implementado como um *BufferLimitado*, *BufferIlimitado*, *Arquivo*, *ArquivoBufferLimitado* e *ArquivoBufferIlimitado*. Em um ambiente de comunidade, estas estruturas também devem ser utilizadas, mas é necessário ter um maior cuidado com a implementação dos buffers. O buffer é ineficiente neste ambiente quando produtor(es) e consumidor(es) de um mesmo nó-contêiner, estiverem sendo executados em processadores diferentes, pois haverá um gargalo para se realizar a transferência na rede de cada elemento do buffer, individualmente.

Para minimizar este gargalo é possível implementar uma estratégia de otimização. Em lugar de transferir pela rede um dado por vez, assim que o produtor gerá-lo, aguarda-se a produção de um conjunto de dados, formando-se a noção de *pacote*, que é enviado pela rede. Desta forma os nós-contêiner seriam implementados pelas estruturas apresentadas na Seção 6.3.1 e por novas estruturas

– *PacoteBufferLimitado*, *PacoteBufferIlimitado*, *ArquivoPacoteBufferLimitado* e *ArquivoPacoteBufferIlimitado* – que incorporam esta noção de pacote.

6.5.2 Otimização Dinâmica por Pipelining

Esta seção discute como o algoritmo de gerência de execução, apresentado para os ambientes pessoal e de laboratório, pode ser adaptado para o ambiente de comunidade.

Como nos ambientes pessoal e de laboratório, o algoritmo recebe o documento de especificação do workflow, constrói o grafo bipartido do workflow e inicializa os atributos dos objetos que compõem o grafo, ou seja, os nós-processo, nós-contêiner e arcos. Feito isso, o gerente de execução aguarda que um nó-processo termine sua execução. Após o término de um nó-processo, o gerente calcula o *estágio_ideal* e o *estágio_real*, ou seja, determina o que será possível executar. Este enfoque é idêntico ao do ambiente pessoal e de laboratório. A diferença encontra-se, assim como no ambiente de laboratório, no método de obtenção do *estágio_real*.

De forma semelhante ao ambiente de laboratório, no ambiente de comunidade, os programas de Bioinformática, dados e recursos estão disponíveis e, mesmo replicados, em diversos processadores. Desta forma, após a definição do *estágio_ideal* (como no ambiente pessoal), é necessário descobrir qual(is) processador(es) serão responsáveis pela execução de um processo e pela disponibilização dos dados e recursos.

Os processadores possuem propriedades que os diferenciam quanto à capacidade, ocupação, programas de Bioinformática instalados e dados disponíveis.

De fato, os programas de Bioinformática não estão necessariamente instalados em todos os processadores que compõem o ambiente. Além disso, os seus dados de entrada e recursos podem estar em diversos processadores, conectados pela rede que compõe o ambiente.

Diferentemente do ambiente de laboratório, no ambiente de comunidade, só será considerado o *acesso local* já que o custo de acesso a dados remotos é tipicamente bastante alto. Ou seja, não será considerado *acesso global*, onde os

processos lêem ou geram dados em qualquer disco remoto como se fosse um disco local.

Todos estes fatores devem ser considerados para a alocação de um processo a um processador existente no ambiente, o que é feito durante a construção do *estágio_real*, conforme será apresentado e justificado a seguir.

O algoritmo de gerência de execução recebe como entrada o documento de especificação do workflow e inicializa as mesmas estruturas de dados básicas do ambiente pessoal e as estruturas de dados representantes dos processadores, como definido no ambiente de laboratório. Entretanto, no ambiente de comunidade, um arco não possuirá o atributo *acesso*.

Como a criação do *estágio_ideal* é feita da mesma forma que no ambiente pessoal, ela será omitida aqui. Entretanto, a obtenção do *estágio_real* é feita de uma maneira particular a este ambiente.

No ambiente de comunidade, assim como no ambiente de laboratório, esta tarefa deve considerar a existência de vários processadores, cada um com sua limitação de espaço em disco e com suas características próprias. Desta forma, o algoritmo necessita escolher um processador para cada processo pertencente ao *estágio_ideal*. Caso isso não seja possível, o algoritmo podará os nós-processo e os nós-contêiner do *estágio_ideal* até que ele se torne viável para ser executado.

A estratégia de construção do *estágio_real* no ambiente de comunidade também será baseada no caminhamento no grafo a partir dos seus nós iniciais até os seus nós de fronteira (como no ambiente de laboratório). Obtém-se os nós-processo do primeiro nível dos componentes conexos do *estágio_real*, e calcula-se os espaços necessários para a execução de cada um deles. Este espaço é calculado pelo método *calcula_espaço*, que soma os espaços necessários para o armazenamento dos dados dos contêineres por um processo *p*. Neste caso, todos os contêineres serão acessados de forma local.

Cria-se uma fila com estes nós-processo ordenados de forma crescente de acordo com o espaço calculado. Enquanto houver espaço livre em algum processador do ambiente, tenta-se alocar o primeiro nó-processo da fila a algum processador. Este tarefa é feita pelo método *com_constrói_estágio_real*, que é idêntico ao método *lab_constrói_estágio_real*, e que, portanto, sua apresentação será omitida. A mudança do nome do método justifica-se apenas para indicar o ambiente em que será utilizado.

O método *com_constrói_estágio_real* chama o método *aloca_processos_pipelining* para tratar da alocação de um determinado processo *p* e de todos os processos p_i que consomem, de forma gradativa, dados gerados por *p*, e assim recursivamente. Ou seja, *aloca_processos_pipelining* trata da alocação de todos os processos que formam um pipeline a partir de *p*.

A alocação de cada processo *p* é feita pelo método *aloca_ou_poda_processo*. Entretanto, este método não tentará alocar processadores para os nós-contêiner que *p* acessa de forma global através do método *aloca_contêineres_globais*, já que estes nós-contêiner não existem neste ambiente. Sendo assim, o método será idêntico ao do ambiente de laboratório, exceto pela retirada da primeira linha e, portanto, não será apresentado aqui. A tarefa principal do método é a alocação de um processador para *p* e para (todos) os nós-contêiner que *p* acessa (de forma local) através do método *aloca_contêineres_locais* (o mesmo do ambiente de laboratório).

O método *aloca_contêineres_locais* cria uma lista com todos os nós-contêiner que o processo *p* acessa (para leitura ou geração de dados), calcula o espaço necessário para armazenar estes nós-contêiner, e chama o método *encontra_processador_contêineres_locais*, responsável pela alocação de um processador para *p* e para todos os seus nós-contêiner. Caso a alocação seja feita, chama-se o método *libera_espaço_extra* para, se possível, liberar os espaços a mais que foram alocados aos nós-contêiner de forma cautelosa. Caso contrário, o método *aloca_ou_poda_processo* chama o método *grad_poda_estágio_real*, que chama o método *calcula_poda* para cada um dos nós-contêiner de *p*. O método *grad_poda_estágio_real* é idêntico ao *lab_poda_estágio_real*. O nome foi mudado apenas para ficar de acordo com o ambiente em que está sendo considerado.

No momento da execução do método *encontra_processador_contêineres_locais*, os nós-contêiner com os dados de entrada já foram alocados a algum processador. Porém, eles precisarão ser alocados ao mesmo processador em que *p* e seus nós-contêiner com dados de saída forem alocados.

As estimativas dos tamanhos dos nós-contêiner continuam sendo feitas de forma cautelosa, pois considera-se que é necessário encontrar um processador que

seja capaz de armazenar todos os nós-contêiner que armazenam dados gerados por *p*, com seu tamanho máximo.

O método *encontra_processador_contêineres_locais* é responsável pela alocação de um processador m_i a um nó-processo p_i. Para isso é necessário que m_i tenha instalado o programa que o processo p_i executa e que haja espaço suficiente para armazenar os dados dos nós-contêiner que p_i acessa.

Quando mais de um processador atende a estes requisitos para alocar nós-contêiner, é necessário escolher o processador mais adequado, segundo alguma estratégia.

Como neste ambiente de comunidade, diferentemente do ambiente de laboratório, está sendo considerado que o acesso a dados não locais (e a conseqüente transferência de dados entre os processadores) possui um custo alto, a estratégia apresentada aqui opta, sempre que possível, pela alocação de um nó-processo a um processador tal que o custo da transferência de dados de entrada do processo *p* seja o menor possível. Desta forma, em um pipeline, a estratégia tentará alocar os nós-processo preferencialmente nos mesmos processadores, já que isto evita a transferência de dados entre os processadores. Este tipo de otimização, dividirá o workflow em sub-workflows, de forma que cada sub-workflow será executado por um processador.

Sendo assim, o método *encontra_processador_contêineres_locais* calculará o custo para a transferência dos dados de entrada, conforme definido anteriormente, para cada um dos processadores que podem executar p_i e escolherá o de menor custo.

Além de todos estes fatores já discutidos, que são relevantes para a alocação de processador, antes de calcular o custo de transferência dos nós-contêiner de entrada de p_i, é necessário verificar, no caso de um nó-contêiner que armazena um banco de dados, se o processador de origem dos dados permite que seja feita a transferência do banco de dados para um outro processador. Caso não seja permitido, o recurso é chamado de *âncora*, e será obrigatória a execução de p_i no mesmo processador em que a âncora está, não importando o custo da transferência dos outros nós-contêiner.

Caso seja possível transferir os dados do banco, uma estratégia que pode ser considerada para diminuir o custo da transferência é a execução de um filtro nos dados, antes de sua transferência, diminuindo a quantidade de dados que devem

ser transferidas, mas não influenciando nos resultados de p_i. Esta idéia está baseada no fato de que é menos custoso executar um filtro nos dados no seu processador de origem, transferir estes dados e executar p_i sobre eles, a transferir todos os dados para então executar um filtro e p_i sobre eles [Özsu, 1999].

Caso ainda exista mais de um processador capaz de executar p_i, assim como no ambiente de laboratório, é possível adotar outras estratégias baseadas na capacidade do processador, no espaço dos nós-contêiner, ou na combinação destes dois fatores.

Uma sofisticação no método *encontra_processador_contêineres_locais,* responsável pela alocação de um processador m_i a um nó-processo p_i, seria, em lugar de deixá-lo escolher um processador analisando somente os nós-contêiner de p_i, permitir que esta escolha fosse feita analisando também os nós-processo (e seus respectivos nós-contêiner) que são executados após p_i de acordo com o pipeline.

6.5.3 Otimização Dinâmica por Paralelização

Um processador de um ambiente de comunidade pode ser simples, como os apresentados na Figura 30, ou complexo, representando um ambiente de laboratório interno. Neste último caso, elege-se um dos processadores deste ambiente de laboratório interno para ser uma interface para o ambiente de comunidade. Desta forma, o ambiente de comunidade não precisa ter conhecimento deste ambiente interno e, consequentemente, continuará executando os algoritmos definidos anteriormente como se todos os processadores do ambiente fossem simples. Contudo, os processadores complexos permitem a utilização das estratégias de otimização dinâmica por paralelização, que devem ser gerenciadas pela interface do ambiente de laboratório interno. Com isso, diminui-se o tempo de execução do workflow.

Uma outra estratégia que permite a diminuição do tempo de execução do workflow é baseada na idéia da fragmentação do banco de dados, apresentada no ambiente de laboratório. Em um ambiente de comunidade, alguns bancos de dados estarão replicados, sendo possível executar cópias de um mesmo processo para analisar fragmentos do banco de dados. Os resultados parciais serão enviados para

um processador central que deve executar um processo para analisar os resultados parciais e formar o resultado final.

Por exemplo, no caso do BLAST, cada sequência de entrada deve ser enviada para todos os *n* processadores que possuem réplicas do banco de dados. Cada cópia de processo BLAST deve ser configurada para realizar a comparação da sequência de entrada com um determinado fragmento do banco de dados. Por *default*, o BLAST faz a comparação da sequência de entrada com todas as sequências do banco de dados. Entretanto, é possível restringir a comparação em um fragmento do banco de dados, configurando os parâmetros *dbrecmin* e *dbrecmax*, que indicam a primeira e a última sequência que deve ser analisada do banco de dados, respectivamente [WU-BLAST, 2004a].

Como exemplos de bancos de dados replicados tem-se o SwissProt e o TrEMBL que podem ser encontrados em [EMBL-EBI, 2004] e em [Expasy, 2004].

6.6 Comentários Finais

As biossequências, principais dados analisados pelos programas de Bioinformática, são representadas por cadeias de caracteres (tirados de um alfabeto de 4 caracteres, no caso de DNA, e de 20 caracteres, no caso de proteínas) de diferentes tamanhos ou por estruturas bidimensionais ou tridimensionais. Estas biossequências podem conter informações biológicas importantes, mas que, na maioria dos casos, são difíceis de serem descobertas devido à complexidade de suas representações. Assim, os programas de análise destes dados também são complexos, baseando-se normalmente em heurísticas, já que a obtenção do resultado ótimo é quase sempre computacionalmente inviável.

Este é o caso, por exemplo, dos programas de comparação de sequências em bancos de dados, como o BLAST e FAST, que foram criados devido à complexidade de tempo quadrática do algoritmo Smith-Waterman, que encontra o alinhamento ótimo entre um par de sequências. Da mesma forma, os programas mais utilizados para montagem de fragmentos, como Phrap e CAP3, são baseados em heurísticas, já que a solução ótima é construída através do alinhamento múltiplo de sequências, um caso ainda mais intratável do que o alinhamento entre pares de sequências. Outros exemplos, como os algoritmos Pratt e Teireisias de

descobrimento de padrões em biossequências, o Glimmer de predição de genes e o Modeller de obtenção da estrutura tridimensional de proteínas, podem ser citados, pois também são baseados em heurísticas pelo mesmo motivo dos outros.

A definição de funções de custo para estimar *a priori* o tempo e o espaço necessários para a execução destes programas de Bioinformática é tarefa bastante complexa. Consequentemente, torna-se praticamente inviável implementar estratégias de otimização *a priori* para workflows de Bioinformática que sejam baseadas em funções de custo sofisticadas. Portanto, a estratégia proposta neste capítulo estima *a posteriori* (durante a execução) os tamanhos dos contêineres de forma bastante cautelosa, aloca *a posteriori* os processos aos processadores, e está baseada em heurísticas de pipelining e de paralelização de processos, que não dependem de funções de custo complexas, e que são vantajosas quanto a espaço e tempo de execução, dentro de expectativas razoáveis.

No caso do ambiente pessoal, a estratégia de pipelining não diminui o tempo final de execução do workflow, pois os recursos da máquina centralizadora serão divididos para atender aos processos concorrentes. Entretanto, esta estratégia contribui para reduzir o número de arquivos temporários e para que os resultados intermediários sejam fornecidos ao pesquisador mais rapidamente. Ao acessar os resultados intermediários, o pesquisador pode decidir continuar ou não a execução do workflow.

Nos ambientes de laboratório e de comunidade, o pipelining reduz o número de arquivos temporários e o tempo de apresentação dos resultados intermediários do workflow ao pesquisador e, ainda, reduz o tempo de execução do workflow.

As formas de implementação dos contêineres, em todos os ambientes, é vantajosa pois evita a replicação desnecessária dos dados e a sincronização dos processos (produtores e consumidores). Evitando-se a sincronização, no ambiente pessoal, reduz-se o tempo de apresentação dos resultados intermediários do workflow, e, nos ambientes de laboratório e comunidade, além desta redução, há também uma melhora no tempo total de execução do workflow.

Além do pipelining, é possível adotar outras estratégias de otimização nos ambientes de laboratório e de comunidade, baseadas na paralelização de cópias de processos através de distribuição de dados de entrada e da distribuição de recursos. Estas estratégias diminuem o tempo de execução do workflow. Entretanto esta idéia também pode ser utilizada para permitir maior confiança no

sistema. Em situações críticas, é interessante paralelizar cópias de processos para se ter garantia do sucesso de sua execução. Além disso, caso um processo de inspeção descubra que ocorreu algum erro de execução ou ocorra alguma falha em um processador, o gerente de execução pode alocar uma outra máquina que possua uma cópia do processo que estava sendo executada. Isto deve ser feito caso não seja detectado um erro de programação e que, consequentemente, se repetirá durante uma próxima execução com a cópia do processo, mesmo que instalada em outra máquina. No caso de erro de programação, é possível utilizar um outro programa de Bioinformática que desempenhe a mesma tarefa.

A otimização do workflow dividindo-o em sub-workflows foi proposta apenas no ambiente de comunidade. No ambiente pessoal, como só existe um processador, esta estratégia não se aplica. Em um ambiente de laboratório, esta estratégia poderia ser utilizada, mas como o custo de execução de um processo neste ambiente é muito mais alto que o da transferência dos dados, optou-se pela alocação de cada processo em processadores distintos enquanto fosse possível, não priorizando, consequentemente, a criação de sub-workflows. No ambiente de comunidade esta estratégia foi adotada pois era necessário contar com o custo da transferência dos dados.

Outra otimização que pode ser aplicada em ambiente de comunidade e de laboratório é a denominada otimização global. Por otimização global, entende-se a otimização de vários workflows que estejam trabalhando no mesmo dado, aproximadamente ao mesmo tempo, para reduzir o volume de dados acessado. Esta otimização pode envolver acesso simultâneo a dados em cache ou até mesmo o *caching* de dados entre diferentes transações, como implementado em bancos de dados orientado a objetos.

A otimização global traria, como vantagem, a redução do volume de dados acessado em disco e, consequentemente, a diminuição do tempo de execução dos workflows.

No ambiente de comunidade, a ontologia poderia cobrir ainda uma descrição dos processadores, incluindo outras propriedades de qualidade, como confiabilidade e disponibilidade. A confiabilidade é a razão entre o número de requisições atendidas pelo processador para executar um processo ou disponibilizar um dado ou recurso sobre o número de requisições efetuadas. Em uma arquitetura distribuída, as máquinas podem estar disponíveis, mas não serem

capazes de atender um pedido devido a problemas de congestionamento de rede ou de erros de implementação [Azevedo, 2003]. A disponibilidade é a porcentagem de tempo em que uma máquina está disponível para atender uma requisição de instância de processo, dado ou recurso [Azevedo, 2003]. Estas características também poderiam ser utilizadas pelo SGWBio para classificar qual é o melhor processador a ser alocado.

É interessante ressaltar que os algoritmos de gerência de execução apresentados nesta tese não utilizaram técnicas conhecidas por *Queueing Networks* e *Petri Nets* pois elas são baseadas em modelos de comunicação de processos mais simples que os definidos nesta tese (através da Tabela 6) para os workflows em Bioinformática.

Os algoritmos de gerência de execução são naturalmente executados pelo módulo gerente de execução. No caso do ambiente pessoal será implementado pela classe *PersonalExecutionManager*, no ambiente de laboratório será implementado pela classe *LaboratoryExecutionManager*, e no ambiente de comunidade será implementado pela classe *CommunityExecutionManager*. Em todos os casos, o gerente de execução recebe o documento de especificação do workflow, que é elaborado pelo gerente de otimização.

O gerente de otimização é então o responsável pela criação deste documento, considerando informações definidas na ontologia, que permitem que ele inclua automaticamente processos de transformação de formatos, de inspeção, de armazenamento dos resultados no *data warehouse* do sistema, além de permitir a definição de tipos de contêineres adequados para que o gerente de execução possa realizar a execução de forma otimizada.

Os tipos de contêineres (gradativo, não-gradativo e misto) estão associados às suas formas de implementação e às escolhas das otimizações que serão feitas durante a execução. Desta forma, o gerente de otimização é responsável pela otimização *a priori* e o gerente de execução pela otimização *a posteriori*.

A ontologia apresentada no Capítulo 3 deve ser estendida para contemplar os processadores, suas propriedades e seus relacionamentos com os processos e dados. O gerente de ontologia deve informar ao gerente de execução a capacidade de um processador (ou seja, suas características fixas, como velocidade de CPU e tamanho de disco) e os programas de Bioinformática instalados e dados disponíveis. Com isso, o gerente de execução, conhecendo a ocupação do

processador (ou seja, suas características em um determinado momento como espaço em disco disponível), pode aplicar os métodos definidos neste capítulo para escolher qual é o melhor processador para executar um determinado processo.

As estratégias apresentadas para todos os ambientes estão focadas no espaço em disco que será necessário para a execução dos workflows, tendo em vista o enorme volume de dados que é utilizado pelos workflows em Bioinformática. Contudo, é interessante também a adoção de técnicas de otimização visando melhorar o gerenciamento de memória principal, diminuindo a paginação e, consequentemente, aumentando a velocidade de execução dos processos. Um exemplo deste tipo de estratégia pode ser encontrado em [Lemos, 2000a], que apresenta uma proposta para melhorar o gerenciamento de memória feito pelos programas da classe BLAST, e que pode ser adotada em conjunto com as otimizações propostas nesta tese.