Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Alocação de Tarefas Paralelas Comunicantes em Ambientes Distribuídos Heterogêneos

Marcelo Nardelli Pinto Santana

Monografia apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Profª. Drª. Alba Cristina Magalhães Alves de Melo

Brasília
2006

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenadora: Profª. Drª. Alba Cristina Magalhães Alves de Melo

Banca examinadora composta por:

Profª. Drª. Alba Cristina Magalhães Alves de Melo (Orientadora) – CIC/UnB
Prof. Dr. Philippe Olivier Alexandre Navaux – UFRGS
Prof. Dr. Ricardo Pezzuol Jacobi – CIC/UnB

CIP – Catalogação Internacional na Publicação

Marcelo Nardelli Pinto Santana.

Alocação de Tarefas Paralelas Comunicantes em Ambientes Distribuídos Heterogêneos/ Marcelo Nardelli Pinto Santana. Brasília : UnB, 2006.
109 p. : il. ; 29,5 cm.

Tese (Mestre) – Universidade de Brasília, Brasília, 2006.

1. Programação Paralela, 2. Alocação de Tarefas, 3. Comparação de Seqüências

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro – Asa Norte
CEP 70910–900
Brasília – DF – Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Alocação de Tarefas Paralelas Comunicantes em Ambientes Distribuídos Heterogêneos

Marcelo Nardelli Pinto Santana

Monografia apresentada como requisito parcial
para conclusão do Mestrado em Informática

Profª. Drª. Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Philippe Olivier Alexandre Navaux
UFRGS

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Profª. Drª. Alba Cristina Magalhães Alves de Melo
Coordenadora do Mestrado em Informática

Brasília, 03 de novembro de 2006

Sumário

Resumo

Sistemas distribuídos têm sido cada vez mais utilizados na resolução de problemas que demandam grande quantidade de tempo de processamento, por permitirem a utilização simultânea de vários recursos computacionais. Diversas máquinas com arquiteturas distribuídas foram propostas ao longo dos anos. Entre essas arquiteturas, estão os *clusters* de computadores, que são sistemas distribuídos formados por estações de trabalho interligadas e que podem atingir um bom desempenho a um custo relativamente baixo.

Entretanto, para que a utilização de tais sistemas seja proveitosa, é necessário que a utilização dos recursos disponíveis seja feita de maneira a permitir a otimização de algum critério. A alocação de tarefas em um sistema distribuído visa determinar como serão utilizados os processadores do sistema de modo a otimizar um critério, que grande parte das vezes é o tempo de execução de uma aplicação. Diversas abordagens já foram propostas para o problema de alocação de tarefas, que é um problema NP-Completo, incluindo algoritmos heurísticos e estratégias específicas para determinadas aplicações.

Uma aplicação para qual existem diversas implementações em sistemas distribuídos é a comparação de seqüências biológicas, uma operação básica da biologia computacional que visa determinar o grau de similaridade entre seqüências. Os algoritmos ótimos existentes possuem complexidade de tempo e espaço de $O(n^2)$, sendo baseados na técnica de programação dinâmica e apresentando dependências de dados do tipo *wavefront*. O alto custo desses algoritmos justifica a utilização de sistemas distribuídos na resolução do problema, sendo que a maioria das implementações distribuídas busca utilizar todos os processadores disponíveis no sistema distribuído, de modo a minimizar a tempo de execução.

A presente dissertação propõe um *framework* de alocação de tarefas de aplicações de comparação de seqüências biológicas baseadas em programação dinâmica, além de quatro estratégias de alocação de tarefas. O *framework* e as estratégias de alocação foram implementados em um *cluster* de 10 máquinas. Os resultados mostram que, para seqüências relativamente pequenas, a utilização de todos os processadores disponíveis não é a opção mais vantajosa. Por isso mesmo, a utilização de políticas de alocação que levem em consideração o tamanho das seqüências e as características das máquinas disponíveis pode permitir a redução no tempo de execução da aplicação.

Palavras-chave: Programação Paralela, Alocação de Tarefas, Comparação de Seqüências

Abstract

Distributed systems have been widely used in the resolution of problems that demand a large amount of processing time, because they allow the simultaneous utilization of many computational resources. Many machines with a distributed architecture have been proposed during the years. Among these are computer clusters, which are distributed systems composed of interconnected workstations and that may achieve a good performance at a relatively low cost.

However, in order to take advantage of distributed systems, the available resources must be used in such a way that some criteria can be optimized. Task allocation in distributed systems aims at determine how the processors available in the system are going to be used, so that a criteria, which in many cases is the execution time of an application, is optimized. Many approaches have been proposed to the task allocation problem, which is NP-Complete, including heuristic algorithms and application specific strategies.

There are many proposed distributed implementations of the biological sequence comparison application, which is a basic operation in computational biology that determines the similarity degree between sequences. The optimal algorithms available have time and space complexities in $O(n^2)$, are based in the dynamic programming technique and present data dependencies that follow the wavefront pattern. The high costs of these algorithms justifies the utilization of distributed systems. Most of the known distributed implementations try to use all available processors in the system, so that the execution time can be minimized.

The present document proposes a framework for task allocation for biological sequence comparison applications based on dynamic programming, as well as four task allocation strategies. The framework and the strategies have been implemented in a 10 machine cluster. The results show that, when the sequences are relatively small, using all available processors is not the best decision. For this reason, the utilization of task allocation policies that take into account the sequences size and the machines characteristics may cause the execution time of the application to be reduced.

Keywords: Parallel Programming, Task Allocation, Sequence Comparison

Capítulo 1

Introdução

A idéia de se utilizar mais de um processador com a finalidade de se resolver um problema de maneira mais eficiente não é nova e justifica-se pela existência de aplicações que demandam intenso poder computacional a fim de serem executadas [19]. Além disso, a crescente utilização de computadores nas mais diversas áreas de conhecimento gerou a possibilidade de se desenvolver os mais variados tipos de aplicações, que passaram a incluir requisitos de compartilhamento de recursos e de acesso a recursos remotos além do tradicional requisito de tempo de execução.

Essas necessidades levaram ao desenvolvimento de diversos tipos de arquiteturas de sistemas distribuídos, nos quais vários recursos computacionais são interligados por redes de comunicação com a finalidade de permitir a resolução eficiente de problemas ou o compartilhamento de recursos distribuídos. Tais sistemas evoluíram de sistemas totalmente homogêneos, nos quais todos os componentes possuíam as mesmas características, até sistemas altamente heterogêneos, abrangendo uma ampla gama de recursos distintos, cada qual com suas próprias características. Além disso, a quantidade de recursos disponíveis nos sistemas distribuídos passou de alguns poucos processadores para milhares de recursos computacionais, que podem estar distribuídos através de longas distâncias geográficas [28].

A redução de preço dos processadores de uso geral usados nos computadores pessoais, levou ao desenvolvimento de uma arquitetura distribuída composta essencialmente desses elementos. Essas arquiteturas de computadores são conhecidas como *clusters* de computadores e possuem um certo número de computadores interligados por uma rede de interconexão [45]. Os *clusters* podem variar muito quanto ao número de computadores utilizados, podendo ir de alguns poucos computadores localizados numa sala até milhares de máquinas distribuídas em diferentes localizações geográficas. Uma das maiores vantagens de arquiteturas do tipo *cluster* é a possibilidade de construir, a partir de elementos de custo relativamente baixo (computadores pessoais) uma máquina de desempenho comparável a um supercomputador.

Dado este cenário, é importante o estudo das características dos sistemas distribuídos e dos problemas associados a eles. Uma das questões mais importantes e complexas quando se fala sobre sistemas distribuídos diz respeito a utilização eficiente dos recursos disponíveis. Se a utilização dos recursos não for adequada, os possíveis benefícios oriundos da utilização de sistemas distribuídos podem vir

a ser facilmente obscurecidos pela complexidade de se gerenciar mais de um recurso ao mesmo tempo e pelo *overhead* de se coordenar o trabalho conjunto dos recursos.

Deste modo, o gerenciamento dos recursos é de suma importância nos sistemas distribuídos. Um aspecto específico desse assunto é a distribuição de tarefas à processadores do sistema distribuído. Chama-se de escalonamento de tarefas (ou processadores) o processo de se determinar os processadores onde serão executadas as tarefas de uma aplicação em um sistema distribuído e os instantes de tempo em que essas tarefas serão iniciadas [23]. Em contraste à essa definição, chama-se de alocação de tarefas (ou de processadores) o processo de se determinar apenas os processadores nos quais as tarefas serão executadas.

O problema do escalonamento/alocação de tarefas é extremamente complexo, já tendo sido demonstrado que ele faz parte da classe de problemas NP-completos [29]. Por isso mesmo, quaisquer otimizações possíveis de serem feitas são importantes, fato que ressalta ainda mais a relevância do estudo dessa área. Nesse sentido, diversos tipos de algoritmos e sistemas de alocação de tarefas já foram propostos e estudados, vários dos quais têm por objetivo a alocação de tarefas independente do tipo de aplicação sendo executada. Em contrapartida, existem estratégias que buscam explorar as características específicas de diferentes tipos de aplicações a fim de tornar o processo de alocação mais eficiente, sendo consideradas portanto, estratégias específicas de aplicação (*application specific*).

Quando o sistema distribuído em questão é composto de elementos heterogêneos, o processo de alocação de tarefas torna-se ainda mais complexo, pois as diferentes características dos processadores do sistema ampliam as possibilidades de escolher como utilizá-los. Não obstante essa dificuldade, o uso de ambientes distribuídos heterogêneos é cada vez mais comum. Por esse motivo, sistemas de alocação de tarefas que levam em conta as características dos diferentes recursos disponíveis tendem a ser mais realistas nas suas decisões, quando comparados a sistemas que ignoram essas diferentes características.

Uma área de pesquisa que se beneficia da utilização de sistemas distribuídos é a comparação de seqüências na biologia computacional. A comparação de seqüências visa determinar o grau de similaridade entre seqüências e é uma operação básica na área de bioinformática. Existem algoritmos seqüenciais que fornecem resultados ótimos para esse problema [43][50]. Tais algoritmos baseiam-se na técnica de programação dinâmica e constroem a solução para o problema a partir do preenchimento de uma matriz, onde estão representadas soluções para sub-problemas do problema original. Nota-se nesses algoritmos a existência de dependências de dados que seguem o padrão *wavefront* [45], no qual o preenchimento de uma célula da matriz depende do preenchimento de três outras células. Apesar de fornecerem soluções ótimas, esses algoritmos possuem tempo de execução em $O(n^2)$. Para seqüências grandes, esse custo faz com que a utilização desses algoritmos não seja prática. Dessa forma, existem várias propostas de implementação desses algoritmos e de variações deles que foram projetadas para serem executadas em sistemas distribuídos [41][8][46], de modo a se conseguir uma redução no tempo de execução das comparações.

O objetivo da presente dissertação é investigar o problema de alocação de tarefas em *clusters* heterogêneos de computadores no contexto da aplicação de

comparações de seqüências biológicas. Nesse sentido, a dissertação apresenta uma proposta do projeto de um *framework* para alocação de tarefas de comparação de seqüências biológicas que se baseiam na programação dinâmica e apresentam a dependência de dados do tipo *wavefront*. O *framework* em questão tem por objetivo permitir que sejam utilizadas e comparadas diferentes estratégias de alocação de tarefas, de modo a se verificar os benefícios de cada uma.

Além do projeto do *framework* para alocação de tarefas, também são apresentadas quatro estratégias de alocação de tarefas de comparação de seqüências biológicas que levam em consideração o poder de processamento e os custos de comunicação dos processadores do sistema distribuído heterogêneo. Com isso, têm-se por objetivo investigar o comportamento dessas diferentes estratégias, especialmente no caso em que as seqüências a serem comparadas são pequenas, pois nesse caso, a simples utilização de todos os processadores disponíveis pode não ser a melhor opção, dadas às necessidade de comunicação e sincronização entre as tarefas.

Os resultados obtidos mostram primeiramente uma validação da arquitetura do *framework* proposto, onde foi possível se implementar as quatro estratégias de alocação de tarefas. Em segundo lugar, os resultados mostram que, no nosso ambiente de testes e para os casos em que as seqüências sendo comparadas são pequenas, é realmente mais vantajoso utilizar um número de processadores que seja menor do que o número total disponível. à medida em que as seqüências aumentam de tamanho essa vantagem diminui.

O presente documento está organizado da seguinte maneira. O capítulo 2 descreve os *clusters* de computadores e suas principais características. O capítulo 3 apresenta o problema de escalonamento e de alocação de tarefas e discute diferentes abordagens e algoritmos existentes para a resolução do problema. Já o capítulo 4 apresenta diversas abordagens para o escalonamento e a alocação de tarefas em ambientes distribuídos heterogêneos. No capítulo 5, descreve-se o problema da comparação de seqüências biológicas, incluindo os algoritmos que resolvem o problema e algumas propostas de algoritmos projetados para serem executados em sistemas distribuídos. O capítulo 6 apresenta o *framework* para escalonamento de tarefas em ambientes heterogêneos e detalha as estratégias de alocação de tarefas propostas. Em seguida, o capítulo 7 apresenta e discute os resultados obtidos em nosso ambiente de teste, pela utilização das estratégias de alocação em uma aplicação de comparação de seqüências biológicas. Por fim, o capítulo 8 apresenta as conclusões do trabalho e indica os possíveis trabalhos a serem desenvolvidos no futuro.

Capítulo 2

Computação em Cluster

2.1 Sistemas Distribuídos

Pode-se definir sistemas distribuídos como sistemas compostos por um certo número de processadores interligados por uma rede de interconexão [52]. Constrasta-se essa definição com a de sistemas centralizados, onde há apenas um elemento de processamento ao qual ligam-se terminais, memórias e periféricos [52]. Podemos citar como vantagens inerentes aos sistemas distribuídos a possibilidade de compartilhamento de equipamentos e dados, a possibilidade da distribuição de tarefas entre diferentes processadores e uma possível redução na relação custo/benefício entre dinheiro e poder computacional.

Apesar de tais vantagens, os sistemas distribuídos introduzem novos tipos de problemas que podem se tornar desvantagens quando comparados à sistemas centralizados. Todos esses problemas relacionam-se com a complexidade de se manter tais sistemas funcionando corretamente. Por um lado, o software desses sistemas deve ser capaz de lidar com os aspectos relativos à distribuição de dados e de tarefas, bem como ao compartilhamento de equipamentos, e deve fazer isso da maneira mais transparente possível. Por outro lado, o hardware de sistemas distribuídos introduz a rede de interconexão como uma complicação a mais, pois deve ser capaz de de interligar os processadores de maneira eficiente e confiável.

Não obstante a maior complexidade desses sistemas, seu uso têm se tornado cada vez mais comum, seja como tentativa de se aumentar o desempenho de aplicações por meio de sua execução distribuída, seja para possuir um ambiente com alto grau de disponibilidade, seja para permitir o acesso à informações compartilhadas. Exemplos de sistemas distribuídos são a Internet e as redes de computação móvel, além de máquinas com arquitetura paralela.

De maneira geral, observa-se nos sistemas distribuídos as seguintes características [17]: execução concorrente de atividades, ausência de relógio global que sincronize todos os componentes do sistema e possibilidade de haver falhas de determinados componentes que não sejam perceptíveis a outros componentes.

Diversas classificações já foram propostas para as arquiteturas de computadores, sendo que a taxonomia de Flynn [24] é talvez a mais citada. Ela baseia-se em duas características: o número de fluxos de dados acessados em um dado instante e o número de fluxos de instruções executadas no mesmo instante. As categorias

propostas por Flynn são:

- SISD (*Single Instruction Single Data*). Dentro dessa categoria encontram-se as arquiteturas tradicionais de computadores, baseadas na arquitetura de Von Neumann, onde, em um determinado momento, há apenas um fluxo de instruções que opera sobre um único fluxo de dados;

- SIMD (*Single Instruction Multiple Data*). Nessa categoria, econtram-se máquinas nas quais um único fluxo de instruções pode operar sobre vários fluxos de dados. Por exemplo, uma operação de adição de um valor constante pode ser executada sobre vários dados ao mesmo tempo, gerando vários resultados. Processadores vetoriais encaixam-se nessa categoria, sendo capazes de realizar operações sobre vários conjuntos de dados simultaneamente;

- MISD (*Multiple Instruction Single Data*). Nessa categoria, há a presença de vários fluxos de instruções sobre um único fluxo de dados. Em outras palavras, máquinas dessa categoria seriam capazes de realizar múltiplas operações simultaneamente sobre um mesmo fluxo de dados. Não há exemplos de arquiteturas reais que implementem essa característica.

- MIMD (*Multiple Instruction Multiple Data*). Máquinas na categoria MIMD apresentam vários fluxos de instruções que operam sobre vários fluxos de dados. Um exemplo de arquitetura que se encaixa nessa categoria, são os *clusters* (seção 2.2), formados a partir de vários computadores interligados em rede. Cada computador tem o seu fluxo de instruções, assim como seu fluxo de dado. O conjunto de computadores, portanto, possui vários fluxos de instruções e vários fluxos de dados.

Tanembaum [52] vai além nesse esquema e divide as máquinas MIMD entre máquinas fracamente acopladas e máquinas fortemente acopladas. As máquinas fracamente acopladas são chamadas por ele de Multicomputadores em que os elementos de processamento presentes na arquitetura possuem um espaço de memória privado, não compartilhado com outros elementos de processamento. Os *clusters* de computadores se encaixam nessa categoria, pois cada computador do *cluster* possui sua própria área de memória, que não pode ser diretamente acessada por outros computadores. Uma desvantagem dos sistemas fracamente acoplados é que o tempo de acesso à uma posição de memória tende a ser alto, visto que, se a memória não for a memória local do processador, este não poderá acessá-la diretamente. Entretanto, essa característica favorece o projeto de arquiteturas tolerantes à falha.

Já uma arquitetura fortemente acoplada é denominada por Tanembaum de Multiprocessadores e possui um espaço de memória compartilhado entre todos os elementos de processamento . Por esse motivo, o tempo de acesso à memória é muito menor, visto que sempre se pode acessar diretamente a memória compartilhada. Entretanto, a presença da memória compartilhada introduz um ponto único de falha na arquitetura.

Cada uma dessas categorias pode ser ainda divida de acordo com a rede interconexão dos processadores, que pode ser por barramento ou comutada. Em máquinas conectadas por barramento, todos os processadores estão ligados ao mesmo barramento de comunicação. Essa característica leva à possibilidade de um dos processadores, por exemplo, não conseguir usar o barramento, que pode estar sendo usado por outro processador. Além disso, o acréscimo de processadores ao barramento aumenta as situações de disputas pelo barramento, o que impossibilita que essa arquitetura tenha boa escalabilidade. Entretanto, o projeto da arquitetura é mais simples.

Em contraste, máquinas cuja rede de interconexão seja comutada (por exemplo, um *switch crossbar*) permitem que mais de um processador se comunique ao mesmo tempo. Dessa forma, reduz o número de disputas pela rede e aumenta a escalabilidade da arquitetura. Entretanto, o projeto de uma rede comutada é mais complexo e tende a ser mais caro.

2.2 Clusters de Computadores

O desenvolvimento de computadores pessoais com *chips* cada vez mais poderosos, conjugado ao desenvolvimento na tecnologia de redes de intercomunicação, levou à idéia de se construir uma infraestrutura computacional a partir de componentes de baixo custo, mas que ainda assim fosse capaz de obter um desempenho comparável ao de um supercomputador. Ao agrupamento de computadores pessoais (*Desktop Computers*) e discos por meio de uma rede de intercomunicação denomina-se *cluster* [30].

Clusters são sistemas distribuídos que apresentam algumas características próprias. Entre elas podemos citar [34]:

- Cada nó de processamento do *cluster* é um computador praticamente completo, com processador, memória e disco rígido, mas possivelmente sem alguns dos periféricos, tais como monitor ou teclado;
- Em geral, os nós de processamento são interligados por redes comuns de baixo custo;
- Os nós de processamento são fracamente acoplados à rede de interconexão;
- Cada nó de processamento possui um sistema operacional completo instalado e
- O agrupamento de computadores é visto como uma única máquina e não como muitas.

A idéia básica por trás do conceito de *cluster* é fazer com que um agregado de computadores se comporte como um único recurso cujo desempenho e capacidade sejam maiores do que cada um dos seus componentes isoladamente. Além disso, uma vantagem de um *cluster* sobre outros tipos de arquiteturas é a possibilidade de se ter uma máquina com alta disponibilidade. Visto que cada nó de processamento é uma máquina completa, a falha de um dos componentes (seja falha

de hardware ou de software) não impossibilita o funcionamento do *cluster* como um todo. Essa mesma característica se traduz em outra vantagem, relacionada à escalabilidade do sistema. Uma vez que o *cluster* é formado por componentes comuns (*commodities*), torna-se mais fácil adicionar ou remover componentes do *cluster* de modo a adequar o funcionamento. Por fim, a relação custo/benefício de um *cluster* tende a ser melhor do que a de outros tipos de sistemas distribuídos, já que componentes comuns são mais baratos do que componentes proprietários projetados para uma arquitetura específica.

Entretanto, assim como todo tipo de sistema distribuído, os *clusters* apresentam algumas desvantagens, das quais a mais significativa talvez seja a complexidade de administração do *cluster*. Já que os componentes do *cluster* são computadores completos, o custo e o esforço de administração de um cluster com n máquinas é próximo do custo de administração de n máquinas distintas. Apesar dessas dificuldades relacionadas à administração, os *clusters* tornaram-se uma das arquiteturas mais populares de sistemas distribuídos.

Observando-se as características de um *cluster* podemos classificá-los, de acordo com a taxonomia adotada por Tanenbaum [52], como sendo máquinas MIMD, fracamente acopladas, do tipo Multicomputadores. Em relação à rede de interconexão, podemos encontrar tanto *clusters* baseados em barramento quanto comutados.

Alguns outros esquemas de classificação de *clusters* utilizam outras características, tais como [34]:

- Tipo de nó de processamento do *cluster*: O *cluster* pode ser formado a partir de computadores completos, inclusive com os seus periféricos, ou então a partir de computadores que não possuem os periféricos e que estejam organizados lado a lado em *racks*. Essa característica pode influenciar no projeto da rede de interconexão a ser utilizada.

- Administração dos nós de processamento: Os nós de um *cluster* podem ser administrados e configurados de uma maneira centralizada ou não. Em geral, *clusters* onde os nós são computadores completos não são administrados centralizadamente. Essa característica influi na complexidade de se gerenciar um *cluster*.

- *Clusters* podem ser homogêneos ou heterogêneos: Em um *cluster* homogêneo, os nós possuem exatamente a mesma configuração, tanto de hardware quanto de software, enquanto que em um *cluster* heterogêneo, os nós possuem diferentes *hardwares* e estão configurados de maneira distinta uns dos outros. Essa característica influencia na complexidade do gerenciamento do *cluster* e também na maneira como algumas operações, tais como a distribuição de tarefas entre os nós, podem ser executadas.

- Segurança de comunicação: Os nós de um *cluster* podem ser acessados de fora do *cluster* por meio de protocolos padrão de comunicação ou não. No caso de isso não ser possível, o acesso ao *cluster* se dá geralmente por meio de um nó mestre, que centraliza toda a comunicação entre o *cluster* e o mundo externo.

- Um *cluster* pode ser dedicado ou não. Em um *cluster* dedicado, uma tarefa a ser executada costuma ter exclusividade quanto ao uso dos recursos disponibilizados, podendo utilizá-los sem a necessidade de competir com outras tarefas. Por outro lado, em um *cluster* não dedicado, várias tarefas de vários usuários podem estar sendo executadas concorrentemente, de maneira que a utilização dos recursos do *cluster* terá que ser gerenciada entre as diversas tarefas.

Além dessas características, é comum atualmente termos *clusters* cujo projeto visa priorizar uma de duas características: a alta disponibilidade ou o alto desempenho. Em um *cluster* de alta disponibilidade, todo o projeto de hardware e software é feito tendo por objetivo permitir que o *cluster* seja utilizado o maior tempo possível sem a ocorrência de falhas, do ponto de vista do usuário do *cluster*. Observa-se um alto grau de redundância dos componentes do *cluster*, de modo que se ocorrer uma falha em um dos componentes, outro pode assumir o seu lugar sem impedir o funcionamento do *cluster*. Já em um *cluster* de alto desempenho, o projeto é feito de modo que uma tarefa possa utilizar todos os recursos disponíveis de modo a obter o maior desempenho possível na sua execução.

2.3 Modelos de Programação

Para escrevermos um programa para resolver um determinado problema é preciso conhecermos as operações básicas permitidas para lidar com os dados do problema. Em um sistema monopocessado, as operações mais básicas que estão disponíveis costumam ser a leitura e a escrita de dados em memória. Diferentes linguagens de programação mapeiam essas operações básicas de diferentes maneiras, porém o conceito é o mesmo. Essas operações básicas de nível mais alto (em relação à forma como essas operações são implementadas em linguagens de programação) formam o chamado modelo de programação da máquina [19]. Em uma máquina monoprocessada, o modelo de programação universalmente aceito é conhecido como modelo de Von Neumann. Assim, um modelo de programação pode ser visto como sendo a arquitetura de alto nível de um sistema computacional ou, em outras palavras, como sendo a visão que uma aplicação tem do sistema em que está sendo executada [45].

Quando deixamos de lado o modelo monoprocessado e passamos a lidar com sistemas com vários processadores (ou seja, MIMD, de acordo com a taxonomia de Flynn [24]), onde as unidades de processamento estão distribuídas, introduzimos um novo elemento na maneira de se programar: o sistema de interconexão entre as unidades de processamento. Para se escrever um programa em um sistema MIMD, é geralmente necessário saber como se dá a comunicação no sistema. Diferentes linguagens ou APIs podem implementar a comunicação ente os elementos de processamento de maneiras diferentes, mas essas implementações estarão refletindo um conjunto de operações básicas. Assim, da mesma forma que as linguagens tradicionais usadas em sistemas monoprocessados mapeiam operações de um modelo de programação (o modelo de Von Neumann), as linguagens usadas em sistemas MIMD irão mapear as operações de um modelo de programação paralela (ou distribuída).

Dessa forma, no contexto dos sistemas MIMD, um modelo de programação especifica como as partes de um programa que está sendo executado de maneira distribuída e paralela transmitem informações entre si e quais operações de sincronização existem para coordenar o seu processamento [19]. Os modelos de programação mais adotados são o de memória compartilhada (também conhecido como modelo de espaço de endereçamento compartilhado) e o modelo de troca de mensagens. Além desses modelos, existem outros menos divulgados, como o modelo *data-parallel* [19] e o modelo de arrays sistólicos [19].

2.3.1 Memória Compartilhada

Em um sistema monoprocessado, podem existir diversas tarefas sendo executadas concorrentemente, cada uma utilizando o processador de forma exclusiva durante um determinado tempo. Todas essas tarefas acessam um único espaço de endereçamento. Em um sistema MIMD, ao invés de termos tarefas que usam o processador de maneira exclusiva, podemos ter várias tarefas sendo executadas de maneira verdadeiramente paralela. Desta forma, o modelo de programação de memória compartilhada pode ser visto como uma adaptação do modelo usado em um sistema monoprocessado para o contexto de um sistema multiprocessado. Todas as tarefas sendo executadas de forma verdadeiramente paralela acessam um único espaço de endereçamento, que forma a memória compartilhada usada por elas.

As operações básicas desse modelo continuam sendo as operações de leitura e escrita de dados na memória (*load* e *store*). Dessa forma, uma tarefa pode escrever um valor em uma posição de memória e esse valor estará disponível para leitura por outra tarefa.

Uma implementação do modelo de memória compartilhada assume um conjunto de processadores conectados a um conjunto de módulos de memória por meio de uma rede de interconexão. A capacidade de processamento pode ser aumentada pelo acréscimo de processadores, enquanto que a capacidade de memória pode ser aumentada pelo acréscimo de módulos de memória. A decisão do projeto da rede interconexão afeta significativamente as características da implementação do modelo. Três alternativas de rede de interconexão podem ser visualizadas [19]:

- Rede de interconexão baseada em um *switch crossbar*: nessa implementação, os processadores estão ligados aos módulos de memória por meio de *switch crossbar*, o que oferece um excelente tempo de acesso à memória. Entretanto, assim que o *switch* estiver totalmente utilizado, será necessário trocá-lo, caso seja necessário aumentar a capacidade do sistema. Dessa forma, o custo do *switch* torna-se um fator limitante na capacidade de expansão do sistema.

- Rede de interconexão multi-estágio: nesse esquema a rede de interconexão é formada por um *switch* multi-estágio, que conecta os processadores aos módulos de memória. Com isso, diminui-se o custo para se expandir o sistema. Porém, devido à natureza do *switch* multi-estágio, podem ocorrer colisões e disputas pelas linhas de comunicação, o que irá aumentar o tempo de acesso à memória.

- Rede de interconexão baseada em barramento: Nessa abordagem, os processadores e os módulos de memória estão ligados diretamente a um barramento central compartilhado. O custo de expansão do sistema é o menor possível, pois basta acrescentar novos processadores e módulos de memória ao barramento. Entretanto, pelo fato de toda a comunicação acontecer por meio de um barramento centralizado, as disputas pelo barramento serão cada vez mais freqüentes, à medida em que se aumentar o número de processadores, e cedo ou tarde o desempenho do sistema será seriamente comprometido. Assim, esse esquema acaba por impor uma limitação ao número de processadores devido ao barramento compartilhado.

Sempre que há disputa pelas linhas de comunicação que ligam os processadores à memória, a utilização inteligente das *caches* dos processadores é fundamental para aumentar o desempenho global do sistema, pois permite que um processador acesse uma informação sem ter disputar a linha de comunicação com a memória. Entretanto, gerenciar as *caches* dos processadores é uma tarefa consideravelmente complexa, pois uma vez que existem várias tarefas sendo executadas paralelamente, deve-se garantir que um valor escrito por um processador seja corretamente lido por outro processador. Em outras palavras, é preciso garantir que as *caches* estejam coerentes entre si e com a visão da memória compartilhada pelos processadores, o que é conhecido como problema de coerência de *caches*. Há diversos esquemas e protocolos propostos para garantir a coerência das *caches*.

Os esquemas apresentados acima podem ser implementados fisicamente. Entretanto, é possível ter um sistema que não possua uma memória fisicamente compartilhada, mas ainda assim implemente o modelo de programação por memória compartilhada. Nesse caso, o hardware ou o software do sistema será responsável por criar as abstrações necessárias para implementar corretamente as operações do modelo. Essa abordagem é chamada de memória compartilhada distribuída ou simplesmente DSM (*Distributed Shared Memory*) [17]. Dessa forma, um *cluster*, que é formado por nós de processamento que possuem suas próprias memórias locais, pode implementar um modelo de programação por memória compartilhada. Como dito anteriormente, existem tanto abordagens de hardware quanto de software para se implementar esse modelo.

Entretanto, o fato da memória não ser mais fisicamente compartilhada introduz um problema semelhante ao de coerência de caches, ou seja, deve-se garantir que as operações de leitura e escrita em memória feitas pelos processadores sejam consistentes entre si. A esse problema, dá-se o nome de problema de consistência de memória [17]. Para resolvê-lo, já foram propostos vários protocolos, que garantem diferentes níveis de consistência, desde uma consistência estrita até modelos relaxados de consistência, que assumem determinadas premissas e que não garantem a consistência da memória em todos os instantes de tempo.

2.3.2 Troca de Mensagens

No modelo de programação por troca de mensagens, assume-se que não existe uma memória compartilhada por todos os processadores. Ao contrário, assume que cada processador possui sua própria memória local e é apenas essa memória

que pode ser acessada diretamente pelo processador. Assim sendo, para que dois processadores troquem informações não é possível simplesmente ler ou escrever a informação em um local de memória, uma vez que um processador não pode nem ler e nem escrever na memória de outro processador.

Para viabilizar a comunicação entre processadores, o modelo de troca de mensagens provê duas operações básicas [19]: um envio de mensagem (*send*) e um recebimento de mensagem (*receive*). Para que haja comunicação entre dois processadores, é preciso que um deles execute um envio de mensagem e o outro execute um recebimento de mensagem. Assim, a combinação de um envio de mensagem com um recebimento de mensagem implementa a transferência de uma informação na memória de um processador para a memória de outro processador.

Ao contrário do modelo de memória compartilhada, que usa como operações básicas as mesmas operações disponíveis num modelo de programação de um sistema monoprocessador (leitura e escrita de memória), e por isso mesmo tende a ser, teoricamente, mais intuitivo, o modelo de troca de mensagem costuma ser implementado por meio de bibliotecas, APIs ou chamadas de sistema específicas para a troca de mensagens. As aplicações enxergam apenas as operações básicas do modelo, mas a implementação dessas operações realiza diversas outras atividades, tais como inicialização da rede e roteamento de mensagens, de modo a efetivar a comunicação.

Um exemplo de uma API para troca de mensagens é o MPI [1]. O MPI é uma especificação de biblioteca para troca de mensagens e se tornou a API mais usada para essa finalidade. Sendo uma especificação de biblioteca, MPI apenas define um conjunto de funções, tipos de dados e constates a serem usados em sistemas distribuídos baseados na troca de mensagem. Existem diversas implementações dessa especificação, entre as quais o MPICH [2] é um exemplo. Atualmente, o MPI encontra-se na versão 2.0 [3] e define funções para envio de mensagens, recebimento de mensagens e criação dinâmica de processos, entre outras.

Uma implementação do modelo de troca de mensagens leva em consideração as características da rede de interconexão, incluindo os protocolos de comunicação utilizados e a sua topologia. Já foram propostas máquinas baseadas no modelo de troca de mensagens cuja rede era implementada nas mais diversas topologias, como cubo, hipercubo ou anel, por exemplo [19]. A topologia da rede de interconexão é importante em algumas máquinas, pois podem haver restrições quanto à capacidade de um processador se comunicar com qualquer outro processador na topologia da rede. Atualmente, as implementações do modelo permitem que um processador se comunique com qualquer outro processador do sistema, seja por meio de esquema de nomes, números ou endereços.

A utilização do modelo de troca de mensagens costuma ser bastante utilizada em *clusters*, pelo fato de que um *cluster* costuma ser formado por máquinas completas, cada qual com sua própria memória e seu sistema de E/S. Assim, as operações de envio e recebimento de mensagem fazem bastante sentido dentro do contexto de uma máquina se comunicando com outra máquina distinta. Apesar disso, também podemos ter o sistema implementado em uma máquina que possua uma memória fisicamente compartilhada. Nesse caso, a operação de envio de mensagem corresponderia a escrever os dados da mensagem em uma determinada área da memória (um *buffer*, por exemplo), enquanto que a operação de leitura

iria ler essa área da memória.

2.3.3 Convergência entre os modelos

Atualmente, os principais modelos de programação paralela estão bastante independentes do hardware onde as tarefas são executadas.Assim, é possível ter máquinas com a memória fisicamente compartilhada, mas que mesmo assim expõem para as aplicações a visão de um modelo de troca de mensagens. Da mesma forma, é possível termos arquiteturas que sejam fisicamente incompatíveis com o modelo de memória compartilhada mas que mesmo assim proporcionam ao programador esse modelo de programação [19].

Essa independência dos modelos de programação quanto à implementação é benéfica, pois possibilita aos programadores escolher qual a melhor abordagem para resolver um determinado problema em um ambiente distribuído.

Capítulo 3

Escalonamento de Tarefas em Sistemas Distribuídos

Nos sistemas distribuídos, é importante que os recursos disponíveis sejam utilizados de maneira a permitir a otimização de certos critérios, como por exemplo, a minimização do tempo de resposta dos aplicativos ou a maximização do *throughput* do sistema. A maneira de se garantir a boa utilização dos recursos está em grande parte, associada ao escalonador de recursos [52]. O escalonador visa garantir que os recursos do sistema sejam atribuídos a tarefas (os consumidores dos recursos) de modo a otimizar um certo critério. No caso geral, os recursos são os mais variados possíveis, como por exemplo, dispositivos de armazenamento, unidades de processamento ou largura de banda.

Um caso particular muito importante do escalonamento de recursos é o escalonamento de tarefas. Nesse caso, os recursos a serem considerados são as unidades de processamento e os consumidores desses recursos são tarefas computacionais que devem ser executadas no sistema distribuído. O objetivo então é associar uma unidade de processamento para cada tarefa, visando a otimização de um critério especificado. Esse problema já foi muito estudado e ficou provado que, no caso geral (bem como em diversos casos específicos), é um problema NP-Completo [29].

Convém lembrar que o problema de escalonamento de tarefas não é exclusivo aos sistemas distribuídos. Um computador monoprocessado, por exemplo, também usa um escalonador a fim de determinar a maneira com que a CPU será utilizada pelas diversas tarefas que podem ser por ela executadas. Entretanto, em sistemas monoprocessados, o escalonamento deve determinar apenas quando uma tarefa irá ter a CPU à sua disposição [52]. Já nos sistemas distribuídos, o problema consiste em se determinar quando e onde as tarefas serão executadas.

3.1 Modelo de Escalonamento

Uma maneira de se caracterizar um sistema de escalonamento de tarefas é considerá-lo como sendo composto por um conjunto de tarefas de uma aplicação, um sistema distribuído e uma agenda de execução (*schedule*) na qual um determinado critério é otimizado [23]. As tarefas de uma aplicação podem ser

caracterizadas como sendo uma quádrupla (T, $<$, D, A), onde:

- $T = \{\tau_1, \tau_2, ..., \tau_n\}$ é o conjunto das tarefas,
- $<$ especifica uma relação de ordem parcial entre as tarefas (significando que se $\tau_i < \tau_j$ então τ_j não pode começar antes de τ_i terminar),
- D é uma matriz de comunicação entre as tarefas ($D(i, j)$ fornece uma noção do custo da comunicação entre τ_i e τ_j) e
- $A = (a_i, a_2, ..., a_n)$ é um vetor contendo uma medida da quantidade de computação a ser realizada em cada tarefa, onde a_n é a quantidade de computação a ser realizada pela tarefa τ_n. Se o sistema distribuído em questão for um sistema heterogêneo, ou seja, composto por elementos com diferentes características, então A não será um vetor, mas sim uma matriz, indicando o custo computacional de cada tarefa em cada um dos elementos do sistema distribuído.

Em relação à representação das tarefas, existe uma distinção que costuma ser feita. Essa distinção tem a ver com o fato de haver ou não relações de precedência entre as tarefas, no sentido que é descrito no parágrafo anterior. No caso de haver uma relação de precedência entre as tarefas, dizemos que tal relação é expressa por meio de um grafo direcionado acíclico $G(T, E)$ (DAG). Os nós representam as tarefas e as arestas do grafo capturam as restrições de precedência. Entretanto, pode ser o caso de não haver restrições explícitas de precedência entre as tarefas. Nesse caso, as tarefas podem ser representadas por um grafo não direcionado, chamado de grafo de interação de tarefas e as arestas desse grafo representam apenas as relações de comunicação entre as tarefas, sem impor uma relação de precedência. A figura 3.1 ilustra a diferença entre um DAG e um grafo de interação. Dá-se o nome de escalonamento de tarefas ao problema de alocar elementos de processamento às tarefas que obedecem a uma determinada relação de precedência; quando tal relação não existe, dizemos que trata-se de um problema de alocação de tarefas [23].

Um sistema distribuído, por sua vez, pode ser descrito como sendo o conjunto $P = \{p_1, p_2, ..., p_m\}$ de elementos de processamento. Cada um desses elementos possui uma velocidade de execução (*speed*) associada, denotada por S_i [23]. Os elementos de processamento estão conectados por uma rede de interconexão arbitrária. As ligações entre os elementos podem ser representadas por um grafo, onde cada nó é um elemento de processamento e cada aresta é um elo de ligação entre dois elementos na rede de interconexão. Associada ao sistema distribuído, há uma matriz R que indica a taxa de transferência de dados entre dois elementos (R_{ij} é a taxa de transferência entre os elementos p_i e p_j). Com essas definições, podemos dizer que uma agenda de execução, ou um escalonamento de tarefas, é uma relação f que mapeia cada tarefa em um elemento de processamento em um determinado instante de tempo, ou seja, $f : T \rightarrow P \times [0, \infty)$. Assim, $f(\tau_i) = (p_j, t)$ significa que a tarefa τ_i estará alocada no elemento de processamento p_j a partir do instante de tempo t.

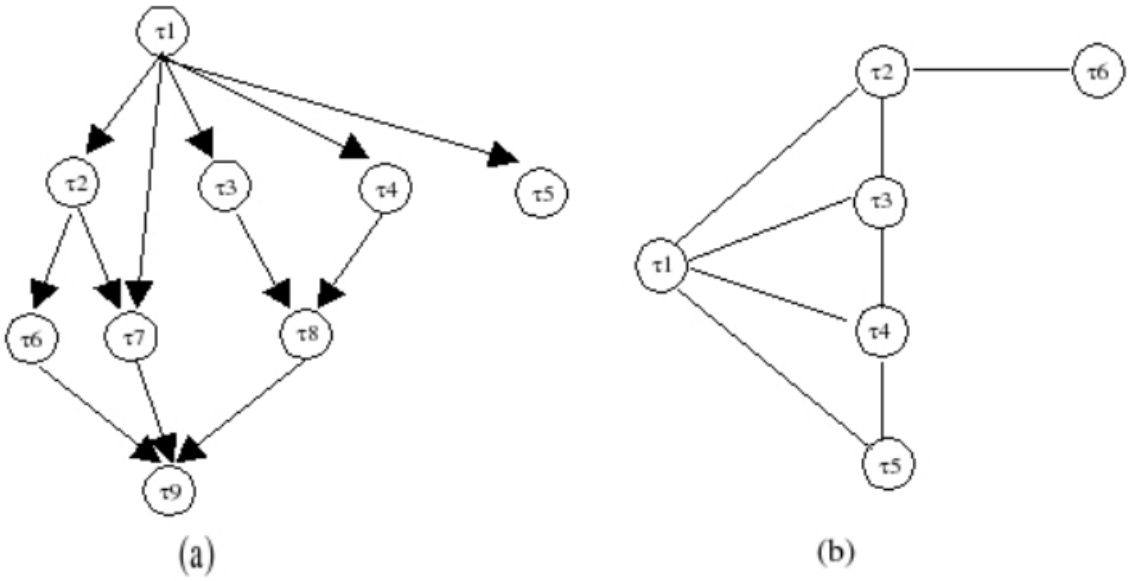


Figura 3.1: (a) Exemplo de um DAG. As arestas direcionadas indicam as relações de precedência entre as tarefas, que estão representadas pelos não. (b) Exemplo de um grafo de interação de tarefas. As arestas nesse grafo não são direcionadas, indicando que não há uma relação de precedência entre as tarefas.

Se todos os parâmetros do grafo de tarefas e do sistema distribuído forem conhecidos (ou estimados), é possível determinar (ou estimar) o tempo de execução de uma tarefa τ_i quando executada no elemento de processamento p_j, denotado pela equação 3.1 [23].

$$ET_{ij} = \frac{A_i}{S_i} \tag{3.1}$$

e o retardo de comunicação entre as tarefas τ_i e τ_j quando executadas nos elementos de processamento adjacentes p_k e p_l é denotado pela equação 3.2 [23]

$$CD_{ij,kl} = \frac{D_{ij}}{R_{kl}} \tag{3.2}$$

O objetivo da construção da agenda de execução é otimizar algum critério. Geralmente, esse critério é a redução do tempo total de execução das tarefas, ou de modo semelhante, a redução do comprimento da agenda de execução. Entretanto, existem outros critérios que podem ser utilizados como, por exemplo, aumentar o *throughput* do sistema.

3.2 Classificação de Sistemas de Escalonamento

Para que os sistemas de escalonamento possam ser melhor estudados, é preciso haver um meio de classificá-los de acordo com suas características. Dessa forma, é

possível estabelecer comparações entre escalonadores e fazer afirmações a respeito do que se pode esperar de um escalonador em questão.

Apesar disso, não há um esquema de classificação que seja totalmente aceito. Um dos esquemas mais aceitos é aquele proposto em [11]. Lá é descrita uma classificação essencialmente hierárquica de classes de escalonadores, embora haja partes desse esquema de classificação que descrevem características que não se encaixam em um nível específico da hierarquia. Embora a classificação seja discutida em termos de escalonadores de tarefas, seu autor defende que ela pode ser aplicada aos escalonadores de recursos em geral. A figura 3.2 ilustra a parte hierárquica do esquema de classificação.

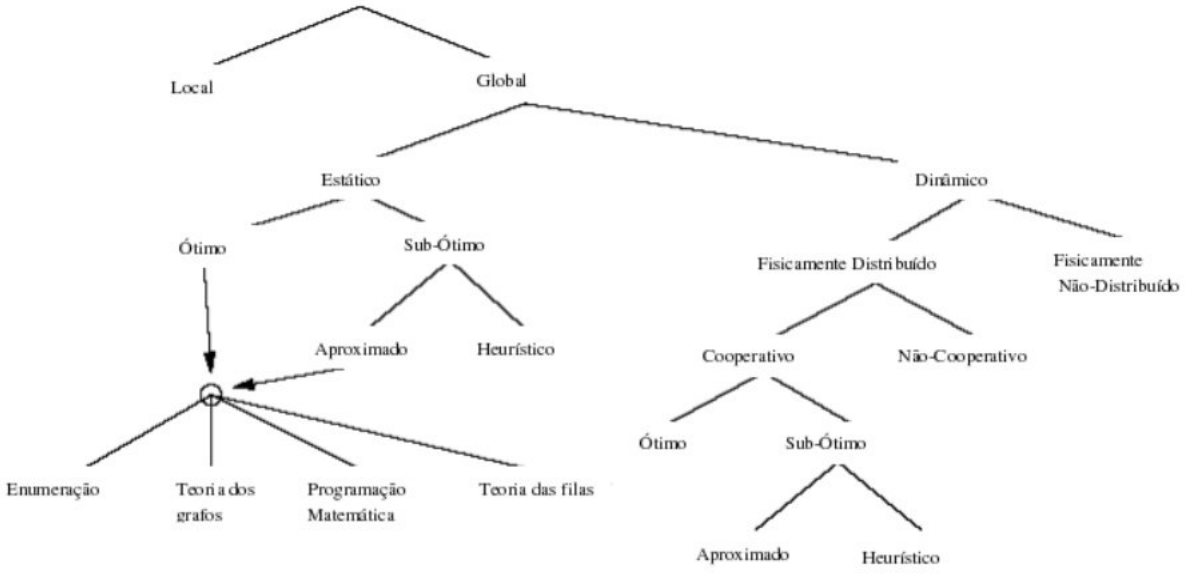


Figura 3.2: Classificação hierárquica de escalonadores (adaptado de [11])

Em primeiro lugar, um escalonador pode ser classificado como sendo local, quando visa alocar um único processador a uma ou mais tarefas, ou global, quando determina em que processador uma tarefa será executada. O segundo nível da hierarquia classifica os escalonadores em estáticos ou dinâmicos. No caso de escalonadores estáticos, assume-se que todas as informações sobre as necessidades de recursos das tarefas a serem executadas sejam conhecidas, bem como, possivelmente, a configuração do sistema. Dessa forma, a decisão do escalonador é feita antes das tarefas serem executadas. Toda vez que a tarefa for executada, ela será atribuída ao mesmo processador por uma agenda de execução já pré-determinada, a menos que as configurações do sistema ou as necessidades de recursos da tarefa mudem, caso em que a agenda deverá ser recalculada. Já no caso dinâmico, assume-se que há pouco conhecimento acerca das necessidades de uma tarefa e da configuração do sistema. Assim, as decisões do escalonador devem ser tomadas à medida que as tarefas forem executadas.

Os escalonadores estáticos podem ser classificados como ótimos ou sub-ótimos. Um escalonador ótimo usa um algoritmo que o possibilita chegar a uma solução ótima em relação a algum critério de otimização, dado que as características do

sistema e as necessidades das tarefas sejam conhecidas. Entretanto, dado que no geral o problema de escalonamento é NP-Completo, pode não ser viável obter uma solução ótima, salvo em alguns casos especiais. Nesse caso, um escalonador sub-ótimo pode ser usado, no intuito de encontrar uma solução que seja sub-ótima, em um tempo razoável. No nível abaixo dos escalonadores sub-ótimos, encontram-se os escalonadores aproximados, nos quais é usado o mesmo modelo computacional de um algoritmo ótimo para se encontrar uma solução boa em relação a algum critério, ao invés de ótima, e os escalonadores heurísticos, que utilizam informações não necessariamente exatas e que influenciam indiretamente o sistema.

Independentemente de o escalonador ser ótimo ou sub-ótimo e aproximado, as técnicas de alocação usadas são as mesmas. A classificação proposta apresenta escalonadores baseados em técnicas de [11]: enumeração e busca do espaço de soluções, teoria de grafos, programação matemática e teoria de filas.

Em relação aos escalonadores dinâmicos, duas classes são propostas, a saber, a dos escalonadores fisicamente distribuídos e a dos fisicamente não distribuídos. Na primeira classe, a responsabilidade de tomar as decisões é de mais de um processador, enquanto na segunda classe, as decisões devem ser centralizadas em um único processador. Os escalonadores fisicamente distribuídos podem ser cooperativos, onde os processadores envolvidos no processo de escalonamento cooperam trocando informações, ou não-cooperativos, onde os processadores tomam suas decisões de maneira independente uns dos outros. Nesse ponto da hierarquia, as mesmas considerações que foram feitas sobre escalonadores ótimos e sub-ótimos para o caso estático se aplicam.

Além dessas categorias dispostas de maneira hierárquica, a taxonomia também apresenta um conjunto de características que formam um esquema de classificação plano (não hierárquico). A justificativa para a existência dessa parte plana da taxonomia é a de que tais características podem aparecer em diversos níveis do esquema hierárquico, o que faz com que seja mais coerente colocá-las à parte.

Com relação a esse esquema plano, a primeira característica abordada é a da capacidade de adaptação. Um escalonador adaptativo é tal que, baseado no comportamento anterior e atual do sistema, utiliza diferentes algoritmos e/ou altera os parâmetros usados pela política de escalonamento. Um escalonador que não seja adaptativo, por sua vez, não modifica seu mecanismo básico de controle de acordo com o histórico do comportamento do sistema.

Uma segunda característica apresentada diz respeito ao balanceamento de carga. Um escalonador que faz o balanceamento de carga procura distribuir igualmente a carga computacional entre os processadores do sistema, de maneira que os processadores apresentem aproximadamente a mesma taxa de progresso. Para que esse esquema possa funcionar de maneira mais adequada, é preciso garantir que os processadores do sistema sejam capazes de fornecer informações sobre suas características dinâmicas, como a carga atual e a taxa de utilização de memória, entre outras. Se as informações forem pouco precisas a esse respeito, é provável que o escalonador não consiga tomar boas decisões.

Outra característica apresentada é a característica de *bidding* [11]. Escalonadores que se encontram nessa categoria utilizam um protocolo de negociação entre os processadores a fim de determinar onde uma tarefa será executada. Os

processadores do sistema podem assumir papéis de gerentes ou de contratados. Um processador que em um determinado momento esteja assumindo o papel de gerente irá representar uma tarefa que precisa de um local para ser executada. Esse gerente anunciará a existência dessa tarefa. Os processadores que desejarem assumir o papel de contratados farão ofertas para o gerente, que por sua vez, poderá escolher qual a melhor oferta.

Uma outra categoria que aparece na parte plana da taxonomia é a de escalonadores probabilísticos. Tais escalonadores, ao invés de examinarem analiticamente o espaço de soluções, utilizam a idéia de atribuir tarefas aos processadores de maneira randômica, de acordo com alguma determinada distribuição de probabilidade [11].

Por fim, há a categoria de escalonadores que permitem a redistribuição das tarefas, ou seja, permitem que as tarefas sejam retiradas de um processador e atribuídas a outro processador mesmo depois de terem sido iniciadas. A idéia por trás deste tipo de escalonador é a de que, uma vez que as características do sistema mudam dinamicamente, pode ser vantajoso mudar a atribuição de tarefas de maneira igualmente dinâmica. Obviamente, deve haver critérios bem definidos para se determinar quando é vantajoso fazer a migração de uma tarefa, caso contrário corre-se o risco de fazer com que as tarefas sejam migradas com muita freqüência, o que claramente irá degradar o desempenho do sistema.

Conforme visto, a classificação proposta por [11] é bem abrangente e visa servir de base para a comparação e estudo de sistemas de escalonamento. No trabalho em questão, diversos escalonadores foram classificados de acordo com esse esquema, numa tentativa de se mostrar a utilidade prática do mesmo. Convém lembrar que, apesar de muito discutida, essa não é a única taxonomia proposta para escalonadores. Por exemplo, em [38] encontramos uma outra taxonomia, menos completa do que aquela descrita em [11], além de uma descrição de vários algoritmos de escalonamento estático. Uma característica interessante da taxonomia proposta em [38] é o fato do problema de escalonamento de programas paralelos ser dividido em escalonamento de *jobs* (tarefas independentes) e escalonamento e mapeamento de tarefas interativas entre si. Tal taxonomia está ilustrada na figura 3.3.

Uma outra taxonomia de sistemas de escalonadores pode ser encontrada em [8], desta vez dando prioridade à caracterização de escalonadores dinâmicos. A principal diferença entre essa taxonomia e aquela proposta em [11] é que o processo de escalonamento de dinâmico é caracterizado como tendo duas fases: uma fase de previsão de estado, na qual o escalonador tem que fazer uma previsão do estado do sistema, e uma fase de tomada de decisão, na qual o escalonador deve decidir como escalonar as tarefas existentes em um determinado momento com base na previsão de estado do sistema feita anteriormente. Portanto, um algoritmo de escalonamento é caracterizado de acordo com as estratégias usadas em cada uma dessas fases. Assim como em [11], esse trabalho busca aplicar a taxonomia proposta a diversos algoritmos de escalonamento já existentes.

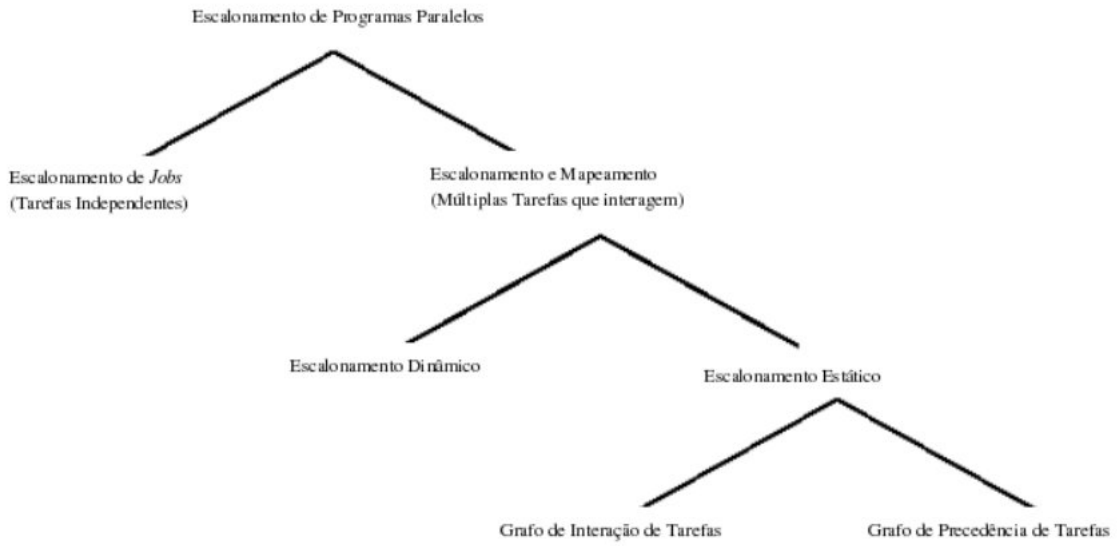


Figura 3.3: Taxonomia de escalonadores proposta em [38], que prioriza a caracterização de escalonadores estáticos

3.3 Uso de Informações dinâmicas

A utilização de escalonadores estáticos depende de termos conhecimento sobre todas as características relevantes da aplicação e do sistema. Na prática, tal situação não ocorre com freqüência, visto que os sistemas computacionais são utilizados por várias aplicações simultaneamente e possuem diversos serviços sendo executados ao longo do tempo. Ainda que existam certas informações que variam pouco ao longo do tempo, como o tipo de processador disponível ou a quantidade total de memória de uma máquina, também existem informações que não se mantêm estáticas, tais como a carga dos processadores e a utilização de memória e de largura de banda. Em outras palavras, podemos dizer que o nível de utilização dos recursos que estão disponíveis irá variar consideravelmente ao longo do tempo. Assim sendo, o uso de informações estáticas dificilmente irá refletir a configuração do sistema em um determinado momento.

O problema exposto leva à necessidade de se utilizar informações dinâmicas, ou seja, que variam ao longo do tempo. De fato, existem escalonadores que se baseiam nas informações dinâmicas obtidas do sistema para tomar suas decisões. Entretanto, o processo de obtenção das informações dinâmicas não deve ser um fator determinante para o desempenho de um escalonador. Por isso, existem sistemas desenvolvidos com a finalidade de disponibilizar, de maneira eficiente, informações dinâmicas coletadas periodicamente, tais como o NWS (*Network Weather Service*) [55].

Tendo à sua disposição as informações dinâmicas coletadas por algum sistema, um escalonador pode criar uma agenda de execução para uma determinada aplicação de modo a explorar a configuração dos recursos no momento em que a aplicação estiver para ser executada. Apesar de parecer interessante em um primeiro momento, esta idéia também pode ser considerada um tanto quanto in-

gênua, embora mais realística do que a utilização de informações estáticas. Pelo fato da natureza das informações utilizadas ser dinâmica, não há nenhuma garantia de que o estado do sistema, no momento em que a aplicação estiver sendo efetivamente executada, seja o mesmo estado que foi inferido a partir das informações coletadas previamente. Na pior das hipóteses, as informações usadas em um determinado momento podem refletir um estado completamente distinto do estado normal do sistema, de modo que a agenda de execução elaborada pelo escalonador pode resultar em um péssimo desempenho.

Essa observação leva à idéia de se tentar prever o estado do sistema em um determinado momento, ou período de tempo. Para que esse objetivo seja atingido, é preciso que haja uma maneira de armazenar de maneira persistente as informações dinâmicas coletadas no sistema, pois essas informações indicam as variações que aconteceram no estado do mesmo. Assim, se for possível armazenar as informações dinâmicas em diferentes momentos, é possível a utilização de métodos estatísticos a fim de se prever, com uma certa segurança, o estado do sistema em um momento futuro, de modo que a agenda de execução criada pelo escalonador poderá utilizar os recursos do sistema de forma mais otimizada.

Um sistema capaz de oferecer esse tipo de funcionalidade é complexo por si só, devendo atender a uma série de requisitos, tais como [7]:

- ser extensível, no sentido de que deve ser possível acrescentar novas categorias de informação a serem coletadas e armazenadas;
- ser flexível, no sentido de que as informações devem poder ser coletadas e acessadas de diferentes maneiras;
- prover informações úteis para aplicações executando simultaneamente em recursos distintos;
- prover tanto informações estáticas quanto dinâmicas para as aplicações;
- prover meta-informações que indiquem a qualidade da informações coletada (precisão, tempo em que a informação foi coletada, etc...); e
- ser acessado em tempo real por várias aplicações simultaneamente.

Um escalonador que tenha à sua disposição tais informações pode tomar decisões mais otimizadas por meio do emprego de técnicas estatísticas. Embora isso já seja suficientemente trabalhoso, ainda é possível pensar em outra utilização para essas informações. Um escalonador poderia refazer suas previsões enquanto a aplicação estivesse sendo executada e, caso considere apropriado, decidir mudar o escalonamento em tempo de execução. Essa seria a estratégia que mais se aproxima da maneira com que nós resolvemos problemas, visto que nos adaptamos à mudanças de condições para redefinirmos nossas estratégias. Porém, a mudança em tempo de execução das decisões de escalonamento traz novos problemas que acrescentam ainda mais complexidade ao processo de escalonamento.

Percebe-se claramente que a qualidade das informações disponíveis ao escalonador desempenha um papel importante na acurácia da agenda de execução a ser gerada e na complexidade do processo de escalonamento. A utilização de apenas

informações estáticas é a abordagem mais simples. Porém, pode levar a agendas que resultem em baixo desempenho, visto que a qualidade das informações pode não refletir o estado real do sistema. à medida em que aumentamos a precisão das informações, torna-se possível melhorar a qualidade das agendas de execução produzidas, ao custo de uma maior complexidade no processo de escalonamento.

3.4 Classes de Aplicações

Uma outra abordagem para o escalonamento de tarefas consiste no desenvolvimento de escalonadores especializados em determinados tipos de aplicações [7]. Existem escalonadores voltados para aplicações de um determinado domínio, tais como o que é descrito em [9]. Por outro lado, existem escalonadores que visam classes de aplicações que tenham uma estrutura em comum. Várias dessas classes já foram identificadas, tais como aplicações mestre-escravo, *parameter sweep*, *bag-of-tasks* e *workflow*.

A vantagem em se desenvolver um escalonador para um determinado tipo de programa é que é possível incorporar as características da aplicação no processo de escalonamento, o que por sua vez possibilita uma maior previsibilidade do comportamento da aplicação e o uso de políticas de escalonamento especializadas [7].

3.4.1 Aplicações Mestre-Escravo

Aplicações do tipo mestre-escravo possuem uma estrutura tal qual a representada pela figura 3.4. A aplicação é composta por uma tarefa mestre e várias tarefas escravas, subordinadas à tarefa mestre. As tarefas escravas costumam realizar o mesmo tipo de processamento, sendo iniciadas pela tarefa mestre. Após o término de seu processamento, cada tarefa escrava envia seus resultados para a tarefa mestre. Dessa forma, o processamento da aplicação é todo controlado pela tarefa mestre, que cria as tarefas escravas, inicia-as e coleta os seus resultados.

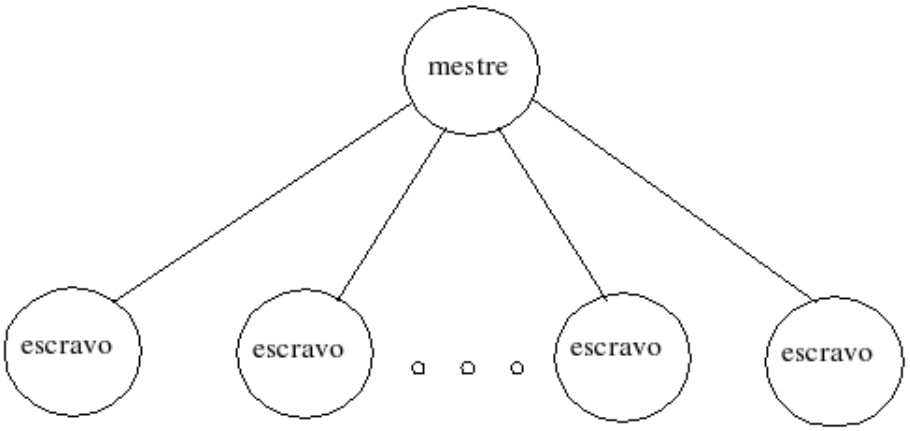


Figura 3.4: Estrutura de uma aplicação mestre-escravo

Um exemplo de um escalonador desenvolvido para essa classe de aplicações é o escalonador usado pelo MW [54], um ambiente de resolução de problemas incorporado ao Condor [54], sistema que tem por objetivo permitir a utilização oportunística de estações de trabalho ociosas na execução de tarefas. O MW cria um processo mestre contendo uma tarefa mestre que pode criar várias tarefas escravas e atribuir unidades de trabalho a elas. A tarefa mestre é composta por uma lista de trabalho, contendo as unidades de trabalho a serem desempenhadas pelas tarefas escravas, um módulo de acompanhamento, que atribui unidades de trabalho às tarefas escravas e acompanha o seu progresso, e um módulo de direcionamento, que examina resultados intermediários e guia a modificação da lista de trabalho e a criação das tarefas escravas.

Se considerarmos que a estrutura das aplicações do tipo mestre-escravo é feita de tarefas que executam ao mesmo tempo e se comunicam em vários instantes distintos, podemos dizer que tais aplicações são representadas por um grafo de interação de tarefas, ou seja, um grafo de tarefas que não possui a relação de precedência entre as tarefas. Entretanto, o número de tarefas que compõem a aplicação não é necessariamente fixo, sendo que as tarefas escravas podem ser criadas á medida em que houver trabalho para ser feito e processadores disponíveis para realizar o trabalho.

3.4.2 Aplicações do tipo *Parameter Sweep*

Uma aplicação do tipo *parameter sweep* pode ser considerada como sendo uma aplicação composta por n tarefas seqüenciais independentes, no sentido de que não existe nenhum tipo de comunicação entre tarefas ou de relação de precedência entre as mesmas [10]. Além disso, considera-se que todas as tarefas realizam exatamente o mesmo tipo de processamento. A única diferença entre o processamento realizado por duas tarefas quaisquer são os parâmetros de entrada usados pelas tarefas.

Aplicações *parameter sweep* são, em geral, aplicações desenvolvidas para explorar um grande espaço de possibilidades de resolução de um problema. São criadas várias tarefas, cada uma resolvendo o problema em questão com um determinado conjunto de parâmetros de entrada. Juntas, essas tarefas varrem o espaço de parâmetros possíveis. Por serem tarefas independentes entre si, muitos consideram que esse tipo de aplicação é ideal para ser executado em sistemas distribuídos como as grades computacionais, onde a distribuição geográfica dos recursos pode implicar em altos custos de comunicação. Um exemplo de um escalonador heurístico para aplicações do tipo *parameter sweep* é apresentado em [10].

O fato de que não existem relacionamentos de precedência entre as tarefas de uma aplicação do tipo *parameter sweep* indica que tais aplicações poderiam ser representadas por grafos de interação de tarefas. Porém, uma vez que as tarefas também são totalmente independentes entre si, a matriz de comunicação entre as tarefas é a matriz nula, que contém zero em todas as posições. Assim, o grafo que representaria essas aplicações seria um grafo composto apenas de vértices, sem nenhuma aresta conectando esses vértices.

3.4.3 Aplicações do tipo *Bag-of-Tasks*

Aplicações do tipo *bag-of-tasks* podem ser encaradas como uma espécie de generalização das aplicações do tipo *parameter sweep*. Assim como essas últimas, as aplicações *bag-of-tasks* são compostas por n tarefas totalmente independentes entre si [14], no sentido de que não existe relação de precedência entre as tarefas e também não há nenhuma comunicação entre as mesmas. Porém, não há nada que indique que as tarefas que compõem a aplicação executem o mesmo tipo de processamento. Duas tarefas quaisquer podem estar realizando processamentos completamente distintos uma da outra.

Uma vez que as tarefas que compõem uma aplicação *bag-of-tasks* são independentes entre si, é comum usar estratégias de replicação de tarefas para otimizar o desempenho dessas aplicações. Um algoritmo de escalonamento projetado para tarefas desses tipo que usa uma política de replicação de tarefas é apresentado em [14]. Assim como as aplicações *parameter sweep*, há quem afirme que as aplicações do tipo *bag-of-tasks* são as aplicações ideais para serem executadas em sistemas de grade computacionais, dada a independência entre as tarefas dessas aplicações.

3.4.4 Aplicações de *Workflow*

Uma aplicação de *workflow* é uma aplicação composta por uma coleção de componentes que devem ser executados de acordo com uma ordem parcial determinada por dependências de controle e de dados [15]. Essa classe de aplicações representa todas as aplicações que podem ser descritas por meio de um grafo acíclico dirigido (DAG).

O problema de se realizar o escalonamento de uma aplicação representada por meio de um DAG já foi e continua sendo muito estudado. Exemplos de escalonadores para essa classe de aplicações são aqueles descritos em [15] e [21].

O tipo de DAG também pode servir para se especializar o escalonador usado. Por exemplo, muitos dos algoritmos descritos em [23] foram projetados para aplicações representadas por determinados tipos de DAG's, como DAG's do tipo *in-forest* ou do tipo *out-forest*.

3.4.5 Aplicações Representadas por Grafos de interação de Tarefas

As aplicações que são representadas por grafos de interação de tarefas, ou seja, aplicações que são formadas por tarefas que não apresentam relacionamentos de precedência, representam uma outra classe de aplicações. A alocação das tarefas dessas aplicações envolve a consideração dos custos de comunicação entre tarefas que vão estar sendo executadas simultaneamente e que estarão se comunicando em diversos momentos distintos.

O padrão de comunicação das tarefas pode servir para se estabelecer outras classes de aplicações. Existem, por exemplo, vários problemas que são resolvidos por meio da atualização de valores de uma matriz. Quando várias tarefas paralelas estão atualizando os valores dessa matriz, é preciso que elas se comuniquem de

maneira a fazer com o processo todo seja coordenado. Diferentes algoritmos estabelecem diferentes padrões de comunicação e atualização da matriz, como os padrões de *wavefront* (seção 5.4) ou *red-black* [45]. Esses padrões de comunicação podem servir de guia para as decisões de escalonamento da tarefas da aplicação.

3.5 Algoritmos de Escalonamento

Um sistema de escalonamento de tarefas é um sistema que interage com recursos (processadores) e consumidores (tarefas), utilizando uma certa política para garantir que os recursos disponíveis sejam usados de forma a garantir a otimização de algum critério [11]. A figura 3.5 ilustra essa situação.

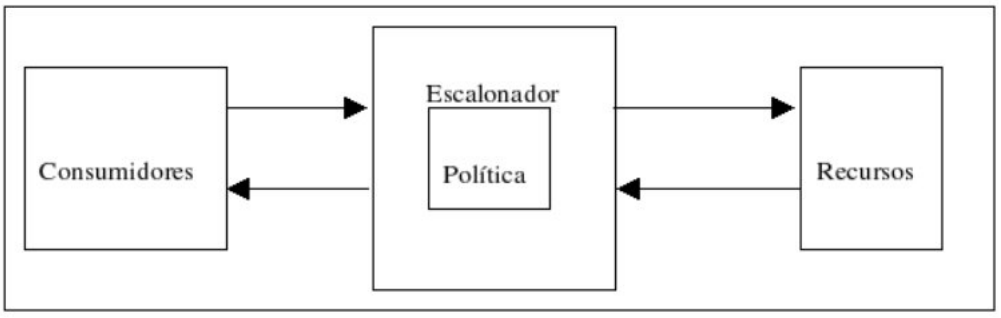


Figura 3.5: Um escalonador e seus relacionamentos com recursos e consumidores (adaptado de [11])

Ao longo dos anos, diversos algoritmos foram desenvolvidos de maneira a implementar as mais diferentes políticas de acesso aos processadores. A seguir, alguns desses algoritmos são descritos.

3.5.1 Escalonamento de grafos de tarefas do tipo *in-forest*

Uma das primeiras soluções ótimas em tempo polinomial para o problema de escalonamento de tarefas foi apresentada em [32]. O algoritmo em questão é ótimo sob certas restrições, apresentando um tempo de execução polinomial. Mais especificamente, o algoritmo assume que existe um grafo de tarefas com n tarefas, um sistema distribuído com m processadores, que o tempo de execução de cada tarefa é unitário e que o custo de comunicação entre qualquer par de tarefas é zero. Além disso, o algoritmo supõe que o grafo de tarefas seja do tipo *in-forest*, ou seja, cada nó possui no máximo um sucessor imediato. Assumindo essas restrições, o algoritmo encontra uma agenda de execução ótima que minimiza o tempo de execução das tarefas.

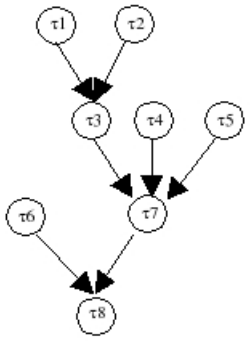
O algoritmo utiliza o conceito de nível de um nó no grafo. O nível de um nó x é o número máximo de nós (incluindo x) em qualquer caminho de x até um nó terminal. O algoritmo está ilustrado na figura 3.6.

Para ilustrar o funcionamento do algoritmo, vamos considerar o grafo de tarefas ilustrado na figura 3.7(a). Além disso, vamos considerar que as tarefas podem

Para cada nó do grafo, faça:
    Calcule o nível do nó
Construa uma lista de prioridades com base no nível de cada nó
Repita
    Quando houver um processador livre, atribua a ele
    a tarefa não executada de maior prioridade
até não restarem tarefas que não tenham sido atribuídas a processadores

Figura 3.6: Algoritmo de escalonamento para grafos do tipo *in-forest* ([23])

ser alocadas a 3 processadores (p_1, p_2 e p_3). O grafo é do tipo *in-forest*, no qual cada nó possui no máximo um sucessor. O algoritmo inicia calculando o nível de cada um dos nós do grafo. Nesse caso, o nível de um nó pode ser calculado como sendo o número de nós no caminho que inicia no nó em questão e termina no único nó terminal do grafo. Os valores calculados dos níveis dos nós do grafo estão na figura 3.7(b). Inicialmente, não há nenhuma tarefa alocada a nenhum processador. Assim, o algoritmo escolhe a tarefa de maior prioridade, τ_1, para ser executada em um processador (p_1, por exemplo), no instante de tempo 1. A tarefa τ_2 é escolhida em seguida para ser executada em p_2, no instante de tempo 1. Como existem três processadores disponíveis, o algoritmo ainda escolhe mais uma tarefa para ser executada no instante de tempo 1. A tarefa τ_3 não pode executar nesse instante pois depende das tarefas τ_1 e τ_2. Assim, é escolhida a tarefa τ_4, que é alocada ao processador p_3. O algoritmo continua executando esses passos e o resultado final (a agenda de execução) está ilustrada pelo gráfico Gantt da figura 3.7(c).



(a)

Nó	Nível
τ1	4
τ2	4
τ3	3
τ4	3
τ5	3
τ6	2
τ7	2
τ8	1

(b)

Tempo	p1	p2	p3
1	τ1	τ2	τ4
2	τ3	τ5	τ6
3	τ7		
4	τ8		

(c)

Figura 3.7: (a) grafo do tipo *in-forest* usado para ilustrar o funcionamento do algoritmo. (b) Valores dos níveis de cada nó do grafo calculados pelo algoritmo. (c) Mapa de Gantt representando a agenda de execução das tarefas.

O algoritmo proposto possui complexidade de tempo em $O(n)$. O algoritmo também funciona, com pequenas modificações para o caso de grafos do tipo *out-forest*, ou seja, grafos onde cada nó possui no máximo um predecessor imediato.

3.5.2 Escalonamento de grafos de tarefas do tipo *in-forest/out-forest* considerando comunicação

Esse algoritmo, proposto em [22], elimina a restrição de que o custo de comunicação entre as tarefas seja desprezível. O algoritmo baseia-se na idéia de transformar o grafo de tarefas, introduzindo novas relações de precedência a fim de compensar os custos de comunicação. Entretanto, esse algoritmo só produz resultados ótimos quando houver apenas dois processadores no sistema distribuído.

O algoritmo usa as seguintes definições: a profundidade de um nó é considerada como sendo o comprimento do maior caminho a partir de qualquer nó de profundidade zero até o nó em questão. Um nó cuja profundidade seja zero é um nó que não possua antecessores. Além disso, define-se a operação $swapall(f, x, y)$, onde f é a agenda de execução (*schedule*) e x e y são duas tarefas agendadas para começar a execução no tempo t nos processadores i e j, respectivamente. O resultado da operação é inverter todos os pares de tarefas agendados para serem executados nos processadores i e j no tempo t_1, onde $t_1 \geq t$. A figura 3.8 ilustra o algoritmo.

No grafo $G(V, A)$, do tipo *in-forest*, identificar os conjuntos de nós $S_1, .., S_k$, onde S_i é o conjunto de nós em V com um nó filho comum, $child(S_i)$
$A_1 \leftarrow A$
Para cada conjunto S_i
  Escolher um nó u de S_i com profundidade máxima
  $A_1 \leftarrow A_1 - (u, child(S_i)) \forall v \in S_i$ e $v \neq u$
  $A_1 \leftarrow A_1 \bigcup (v, u) \forall v \in S_i$ e $v \neq u$
  Obter a agenda de execução f
  aplicando o algoritmo visto anteriormente no grafo aumentado (V, A_1)
Para cada conjunto S_i do grafo original G
  Se o nó u (de profundidade máxima) for escalonado em f em um tempo
  imediatamente anterior ao nó denotado por $child(S_i)$
  mas em um processador diferente, aplicar a operação
  $swapall(f, child(S_i), x)$, onde x é a tarefa escalonada no tempo
  imediatamente após de u no mesmo processador.

Figura 3.8: Algoritmo de escalonamento para grafos do tipo *in-forest* considerando comunicação

Para ilustrar o funcionamento do algoritmo, consideramos o grafo da figura 3.9(a). Consideramos agora que o custo de comunicação entre cada par de tarefas que se comunicam seja unitário. O algoritmo inicia construindo os conjuntos de nós que possuem um nó filho em questão. Os conjuntos obtidos são $S_1 = \tau_1, \tau_2$, $S_2 = \tau_3, \tau_4$, $S_3 = \tau_5, \tau_6$, $S_4 = \tau_7, \tau_8$. Em seguida, para cada um desses conjuntos, o algoritmo seleciona o nó de maior profundidade. Quando houver empate, escolhe-se um nó do conjunto aleatoriamente. Assim, para S_1, escolhe-se τ_2, para S_2 escolhe-se τ_3, para S_3 escolhe-se τ_6 e para S_4, escolhe-se τ_7. Em seguida, constrói-se um grafo aumentado de acordo com o passo 3 do algoritmo. O grafo aumentado é exibido na figura 3.9(b). Esse grafo ainda é do tipo *in-forest*. Para esse grafo aumentado, pode-se considerar que a comunicação entre as tarefas é

desprezível e aplicar o algoritmo anterior para encontrar a agenda de execução. Aplicar o algoritmo anterior ao grafo aumentado desconsiderando os custos de comunicação é equivalente a considerar o grafo original considerando os custos de comunicação [23]. O resultado dessa operação é ilustrado na figura 3.9(c). Entretanto, existe uma comunicação entre as tarefas τ_2 e τ_3, que foram alocadas a diferentes processadores. Como o custo de comunicação não é mais desprezível, deveria haver um atraso de uma unidade de tempo, para que a comunicação se completasse. Porém, podemos usar a operação *swapall* para trocar de processador as tarefas que vêm depois de t_2. Assim, τ_2 e τ_3 serão alocadas no mesmo processador, o que implica em custo zero de comunicação. O resultado final, depois da aplicação da operação *swapall* estão na figura 3.9(d).

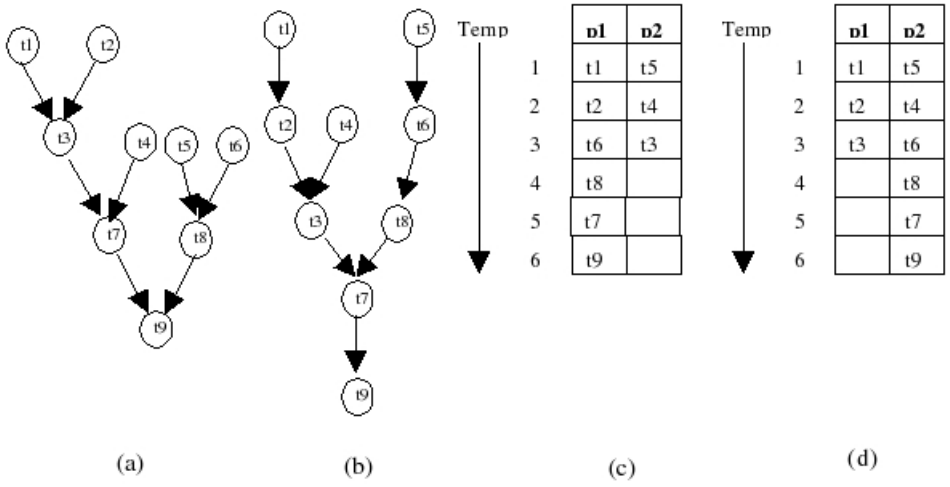


Figura 3.9: (a) grafo original usado como exemplo de aplicação do algoritmo. (b) Grafo aumentado. (c) Agenda de execução resultante obtida a partir do grafo aumentado e desconsiderando-se os custos de comunicação. (d) Agenda de execução depois da aplicação da operação *swapall*.

3.5.3 Escalonamento em dois processadores

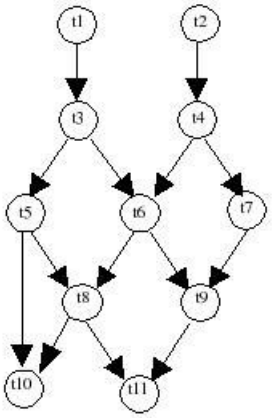
O trabalho apresentado em [36] apresenta um algoritmo ótimo para o problema de escalonamento com as seguintes restrições: há um grafo de tarefas de estrutura arbitrária contendo n tarefas; há um sistema distribuído com 2 processadores; cada tarefa tem custo de execução unitário; os custos de comunicação entre as tarefas são desprezíveis. Nesse contexto, o objetivo é minimizar o tempo total de execução. A figura 3.10 mostra o algoritmo.

O funcionamento do algoritmo será ilustrado pela sua aplicação ao grafo de tarefas da figura 3.11(a). Considera-se que existam dois processadores para serem usados. O algoritmo inicia atribuindo os rótulos 1 e 2 para os nós terminais τ_{10} e τ_{11}, respectivamente. O conjunto de tarefas com que não possuem sucessores ainda não rotulados é $\{\tau_8, \tau_9\}$. Para a tarefa τ_8, temos que $l(\tau_8) = \{2, 1\}$ e para a tarefa τ_9 temos $l(\tau_9) = 2$. Por isso, o algoritmo atribui o rótulo 3 para a tarefa

Atribua o rótulo 1 a uma das tarefas terminais
Repetir enquanto houver tarefas não rotuladas
Seja S o conjunto de tarefas não rotuladas. Assumindo que os rótulos $1, ..., j-1$ já tenham sido atribuídos, será selecionada uma tarefa de S para receber o rótulo j. Para cada x em S, define-se $l(x)$ como sendo a seqüência decrescente de inteiros formada pelo conjunto ordenado $\{L(y_1), ..., L(y_k)\}$ onde $L(y_1), ..., L(y_k)$ são os rótulos atribuídos aos sucessores imediatos de x. Seja x um elemento de S tal que para todo $x^{'}$ em S, $l(x) \leq l(x^{'})$ (em ordem lexicográfica). Deve-se atribuir j para x, ou seja, $L(x) = j$.
Use $L(v)$ como a prioridade da tarefa v, resolvendo empates de maneira arbitrária
Sempre que um processador se tornar disponível, atribua a ele a tarefa não executada de maior
prioridade, resolvendo conflitos de maneira arbitrária

Figura 3.10: Algoritmo de escalonamento ótimo em dois processadores

τ_9 e o rótulo 4 para a tarefa τ_8. O algoritmo continua a sua execução até que todas as tarefas tenham sido rotuladas. Uma tarefa a com maior rótulo é alocada primeiro ao processador p_1; depois uma tarefa b é alocada ao processador p_2 e assim por diante, até que todas as tarefas tenham sido alocadas.



(a)

Nó	Rótulo
t1	11
t2	10
t3	9
t4	8
t5	6
t6	7
t7	5
t8	4
t9	3
t10	1
t11	2

(b)

Tempo	p1	p2
1	t1	t2
2	t3	t4
3	t6	t5
4	t7	t8
5	t9	t10
6	t11	t9

(c)

Figura 3.11: (a) Grafo de tarefas usado para exemplificar o funcionamento do algoritmo. (b) rótulos atribuídos às tarefas. (c) Agenda de execução resultante da aplicação do algoritmo ao grafo de tarefas dado.

Esse algoritmo possui complexidade de tempo em $O(n^2)$. Uma vez que as tarefas possuem tempo de execução unitário, ambos os processadores ficarão livres

ao mesmo tempo. Por isso, assume-se que o processador 1 seja escalonado antes do processador 2.

3.5.4 Escalonamento de tarefas usando algoritmos genéticos

Diversas heurísticas são utilizadas na construção de algoritmos de escalonamento, a fim de se abordar o problema de escalonamento de tarefas no caso geral. O uso de heurísticas justifica-se pelo fato de que, no caso geral, o problema de escalonamento de tarefas é NP-Completo, o que inviabiliza a obtenção de soluções ótimas em tempo aceitável (polinomial). Com o uso de heurísticas, procura-se trocar a obtenção de uma solução ótima pela obtenção de uma solução sub-ótima em um tempo polinomial.

Uma das abordagens heurísticas que vem sendo usada no desenvolvimento de algoritmos de escalonamento de tarefas é a abordagem de algoritmos genéticos. Tais algoritmos utilizam conceitos inspirados pela teoria evolutiva de Darwin para implementar técnicas de busca global a fim de explorar diferentes regiões do espaço de busca de soluções simultaneamente, guardando um conjunto de soluções potenciais chamado de população [37]. As possíveis soluções de um problema que esteja sendo resolvido por um algoritmo genético são codificadas em uma estrutura de dados denominada de cromossomo e sobre ela aplicam-se operadores de mutação e de recombinação (*crossover*). Soluções que atendam um determinado critério de qualidade são mantidas para serem futuramente combinadas a fim de gerarem outras soluções. Aquelas que não atenderem ao critério de qualidade são geralmente descartadas.

Os operadores de recombinação e de mutação simulam a mistura de cromossomos que ocorre na natureza e a mutação que as espécies sofrem. Já o critério de qualidade, denominado de critério de *fitness*, permite a seleção das melhores soluções, o que tem por objetivo simular o processo de seleção natural, onde apenas os indivíduos mais bem adaptados sobrevivem. Esses passos são repetidos por um número pré-determinado de vezes que equivale ao número de gerações de população inicial.

Um algoritmo genético, em geral, possui a estrutura apresentada na figura 3.12 [37].

```
Gerar população inicial
Enquanto não atingir o número de gerações faça
    Para i = 1 até o tamanho da população faça
        Selecionar ao acaso dois cromossomos e aplicar o operador de
        recombinação
        Selecionar ao acaso um cromossomo e aplicar o operador de mutação
    Fim para
    Avaliar os cromossomos da população e realizar a seleção
Fim enquanto
Solução final ← melhor cromossomo
```

Figura 3.12: Estrutura geral de um algoritmo genético

Geralmente, a aplicação das operações de mutação e de recombinação não se dá a toda hora, mas ocorre com uma certa probabilidade, denominada fator de mutação, para o operador de mutação, e fator de recombinação, para o operador de recombinação.

A estrutura do algoritmo genético facilita a sua paralelização, onde vários nós computacionais trabalham em diferentes partes da população inicial e se comunicam a fim de determinar quais são os melhores indivíduos. Entre as estratégias de paralelização de algoritmos genéticos encontram-se os modelos de ilha isolada e de ilha conectada. Em ambos os modelos os nós computacionais se comunicam a fim de trocar informações sobre as soluções [37]. No modelo de ilha isolada, a comunicação só ocorre ao final do processo, onde a melhor solução dentre todas é escolhida. Já no modelo de ilha conectada, a comunicação é feita periodicamente e existe a migração de indivíduos de uma população para outra. O modelo de ilha conectada pode ser síncrono, onde todos os nós participam de uma fase de comunicação simultaneamente, ou guiado por eventos, onde os nós só se comunicam quando é considerado necessário. Este último esquema impõe um *overhead* de comunicação menor ao sistema, mas é mais difícil de ser implementado do que o esquema síncrono [37].

Em [37] é apresentado um algoritmo genético paralelo para resolver o problema de escalonamento de tarefas. O algoritmo é baseado no modelo de ilha conectada síncrona. Esse algoritmo também se baseia em técnicas de algoritmos de escalonamento de listas, uma outra abordagem heurística onde os nós do grafo de tarefas são ordenados em uma lista de acordo com alguma prioridade [38], de maneira que a ordem de precedência entre as tarefas não seja violada. Alguns possíveis valores que costumam ser usados como prioridade estão descritos em [38]. Depois que a lista é construída, os nós são removidos um a um e cada nó é alocado ao processador que minimiza um determinado critério, como por exemplo, o tempo do início de execução de uma tarefa.

Existem outros trabalhos que visam abordar o problema de escalonamento de tarefas por meio de algoritmos genéticos. Em [20], por exemplo, é apresentada uma outra formulação do problema de escalonamento, na qual a estrutura de um cromossomo é composta por dois vetores de genes. Um desses vetores é chamado de vetor de vértices e contém genes que representam os vértices do grafo de tarefas. O outro vetor é chamado de vetor de arestas e contém um gene para cada aresta do grafo. O algoritmo usa dois operadores de recombinação e visa tratar explicitamente o problema de escalonamento de tarefas levando em conta a contenção dos *links* de comunicação do sistema distribuído [20]. Em [5], é apresentado um algoritmo genético que combina várias heurísticas que costumam ser usadas em algoritmos de escalonamento de listas, de forma a obter a melhor agenda de execução.

3.6 Algoritmos de alocação de tarefas

O problema de alocação de tarefas costuma ser encarado de maneira diferente do problema de escalonamento de tarefas. Enquanto este preocupa-se em mapear as tarefas nos processadores e em um instante de tempo, aquele visa apenas mapear

as tarefas nos processadores, sem preocupar-se com o tempo em que cada tarefa deverá ser iniciada. Assim, o problema de alocação de tarefas está relacionado com a execução de tarefas que não possuem uma relação de precedência entre si. Portanto, as tarefas a serem alocadas, bem como as suas interações são representadas por meio de um grafo de interação de tarefas, onde cada nó representa uma tarefa e as arestas representam as interações entre as tarefas.

Uma vez que o problema de alocação de tarefas é tratado de maneira diferente do problema de escalonamento de tarefas, a sua modelagem também é diferente. Para modelar o problema de alocação de tarefas, assume-se o seguinte [23]:

- Existem n tarefas que deverão ser executadas em m processadores.
- Associado a cada tarefa, existe um vetor de m valores que indica o custo da execução da tarefa em cada um dos m processadores. Um tempo indicado como ∞ significa que a tarefa não pode ser executada no processador em questão.
- As tarefas são representadas por nós no grafo de interação.
- Associado a cada aresta do grafo existe um valor que corresponde ao custo de comunicação entre duas tarefas, no caso de serem alocadas a processadores distintos.

Tal modelo pode ser usado para estudar algoritmos de alocação estática. Entretanto, para algoritmos de alocação dinâmica, haverá informações previstas no modelo que não estarão disponíveis como, por exemplo, a estrutura do grafo de interações, o custo da execução das tarefas em cada processador ou as informações exatas sobre os custos de comunicação entre as tarefas.

Assim como no problema de escalonamento, existem algoritmos ótimos que resolvem o problema de alocação de tarefas em alguns casos particulares. Entretanto, sabe-se que no caso geral, bem como em muitos casos restritos, o problema de alocação de tarefas é NP-Completo [29].

3.6.1 Alocação de tarefas em dois processadores

Um dos algoritmos ótimos para o problema de alocação de tarefas é descrito em [51]. Esse algoritmo baseia-se em um algoritmo de fluxo de rede em grafos de rede de dois terminais. Em um grafo de rede de dois terminais, assume-se que existem dois nós especiais: um nó de origem (*source node*) e um nó de destino (*sink node*). Nesse tipo de grafo pode-se determinar um conjunto de corte, que é um conjunto de arestas C tais que, quando removidas, fazem com que o grafo se torne desconexo, particionado em um conjunto de origem (contendo o nó de origem) e um conjunto de destino (contendo o nó de destino). Cada corte de um grafo possui um peso associado, denotado por $W(C)$, que é igual é soma dos pesos de todas as arestas em C. Um conjunto de corte mínimo (ou ótimo) é um conjunto de corte cujo peso é menor do que o peso de qualquer outro conjunto de corte do grafo.

O algoritmo proposto assume a restrição em que existem apenas dois processadores no sistema distribuído. O algoritmo transforma o grafo de interação de tarefas em um grafo de rede de dois terminais, colocando um dos processadores como nó de origem e o outro como nó de destino, e encontra o conjunto de corte mínimo do grafo. As tarefas que ficarem no conjunto origem resultante deverão ser alocadas no processador correspondente ao nó de origem. As demais tarefas serão alocadas ao outro processador. A figura 3.13 mostra o algoritmo.

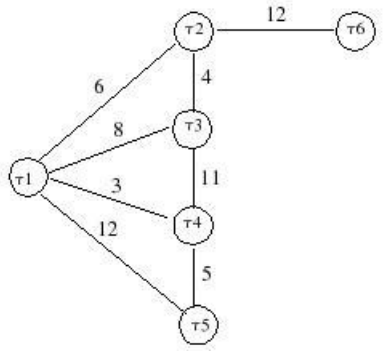
1 - Construir um grafo de rede de dois terminais: adicionar um nó de origem S_1 e um nó de destino S_2, representando os processadores P_1 e P_2, respectivamente. Para cada nó τ no grafo de interação de tarefas original, adicionar uma aresta indo de τ para S_1 e de τ para S_2. O peso da aresta (τ, S_1) é o custo de executar τ em P_2, enquanto que o peso da aresta (τ, S_2) é o custo de executar τ em P_1.
2 - Determinar o conjunto de corte mínimo do grafo, denominado de C.
3 - Alocar a tarefa τ ao processador P_i, se e somente se τ e S_i estiverem na mesma partição originada pelo conjunto de corte C.

Figura 3.13: Algoritmo de alocação de tarefas em dois processadores

O funcionamento do algoritmo será exemplificado pela sua aplicação ao grafo de interação da figura 3.14(a). Os números ao lado de cada aresta indicam o custo de comunicação entre as tarefas. A figura 3.14(b) mostra os tempos de execução de cada tarefa em dois processadores (p_1 e p_2). Inicialmente, o algoritmo constrói um grafo de rede de dois terminais acrescentando dois nós, S_1 e S_2, correspondendo aos processadores p_1 e p_2, e acrescenta arestas ligando esses dois nós a cada um dos outros nós, atribuindo os pesos das arestas de acordo com o que está explicado no passo 1 da descrição do algoritmo. O grafo resultante está ilustrado na figura 3.14(c). A determinação da alocação das tarefas pode ser completada depois de se descobrir o conjunto de corte mínimo desse grafo. A figura 3.14(d) ilustra qual seria esse conjunto de corte. Removendo-se as arestas do conjunto de corte, o grafo fica particionado e as tarefas que estão ligadas ao nó S_1 serão executadas em p_1, enquanto as tarefas que estão ligadas em S_2 serão alocadas ao processador p_2. Nesse exemplo, as tarefas τ_1, τ_2, τ_3, τ_4 e τ_5 serão alocadas ao processador p_1, enquanto a tarefa τ_6 será alocada ao processador p_2.

3.7 Classificação dos algoritmos apresentados

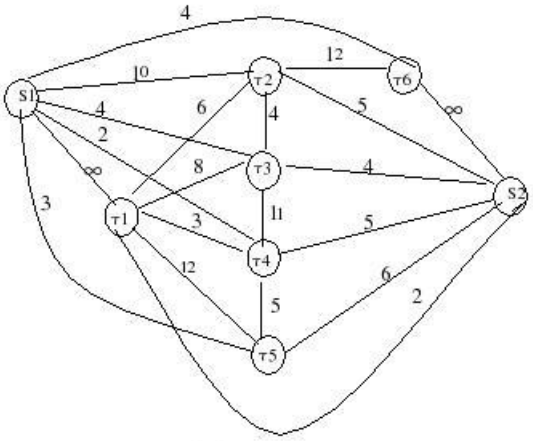
Podemos classificar os algoritmos apresentados anteriormente, com exceção do algoritmo genético paralelo, de acordo com a taxonomia apresentada em [11], como sendo algoritmos globais, estáticos e ótimos. Já para o algoritmo genético paralelo, a classificação indica que ele é um algoritmo global, estático, sub-ótimo e heurístico. Além disso, pode-se dizer que é um algoritmo adaptativo, pois os valores das taxas de mutação e de recombinação, bem como a freqüência da comunicação entre os processadores, varia com o tempo e o estado atual do algoritmo [37]. Deve-se notar que, para tal taxonomia, a distinção entre escalonamento de tarefas e alocação de tarefas não é relevante. Para o autor da taxonomia a única diferença entre esses dois problemas é que o escalonamento de tarefas representa



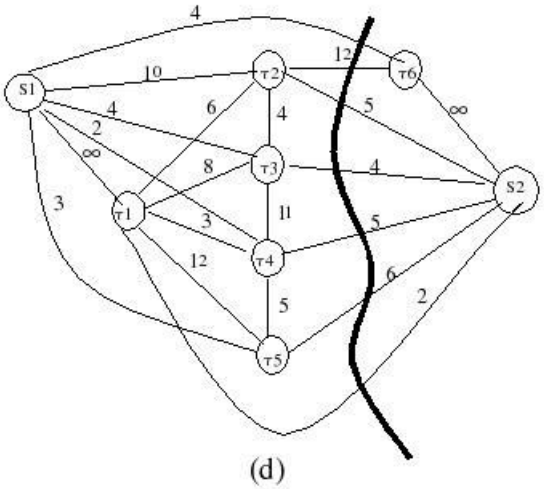
(a)

	Custo de execução em p1	Custo de execução em p2
τ1	2	∞
τ2	5	10
τ3	4	4
τ4	5	2
τ5	6	3
τ6	∞	4

(b)



(c)



(d)

Figura 3.14: (a) Grafo de interação de tarefas com os custos de comunicação exibidos. (b) Tempo de execução de cada tarefa nos processadores p_1 e p_2. (c) Grafo de rede de dois terminais construção pelo algoritmo. (d) Indicação do conjunto de corte ótimo obtido. As tarefas que continuam ligadas ao nó S_1 serão executadas no processador p_1, enquanto que as tarefas que ficaram ligadas ao nó S_2 serão executadas no processador p_2.

a visão dos consumidores de recursos (as tarefas que devem ser executadas nos processadores e os donos das tarefas), enquanto a alocação de tarefas representa a visão dos recursos que serão consumidos (no caso, os processadores do sistema distribuído).

Capítulo 4

Escalonamento de tarefas em Ambientes Heterogêneos

O gerenciamento e o escalonamento de recursos em ambientes heterogêneos é significativamente mais complexo do que em ambientes homogêneos, devido ao fato de que os diferentes componentes do sistema (processadores e *links* de rede, por exemplo) apresentam diferentes características. Ainda assim, existem diversos algoritmos que visam resolver esse problema levando em consideração as diferentes características do sistema.

4.1 Algoritmo para mapear tarefas comunicantes em recursos heterogêneos

Em ambientes heterogêneos nos quais os recursos podem estar geograficamente dispersos, o processo de se realizar o escalonamento de tarefas que se comunicam com freqüência deve levar em consideração os custos de comunicação e a capacidade de comunicação entre os diferentes recursos, de modo a evitar que os custos envolvidos nas trocas de mensagens entre as tarefas se tornem fatores limitantes do desempenho de uma aplicação. Em [53] é proposto um algoritmo que realiza o mapeamento de tarefas em recursos heterogêneos que podem estar geograficamente dispersos, levando em consideração as restrições de comunicação envolvidas. Dada uma aplicação composta de diversas tarefas que se comunicam, o algoritmo proposto busca maximar o *throughput* da execução das tarefas através do agrupamento das tarefas que apresentem maior custo de comunicação.

O algoritmo trabalha com dois grafos: um grafo de tarefas (*task graph*) e um grafo de recursos (*resource graph*). O grafo de tarefas possui vértices que representam as tarefas da aplicação e arestas que representam a comunicação entre tarefas. A cada vértice está associado um peso que indica a quantidade de computação que a tarefa tem a realizar. De maneira similar, a cada aresta está associado um peso que indica os requisitos de comunicação entre duas tarefas. O grafo de tarefas pode ser direcionado ou não. Se for direcionado, representa apenas a direção da comunicação, não implicando nenhuma relação de precedência entre as tarefas. O grafo de recursos, por sua vez, modela o sistema distribuído no qual as tarefas serão executadas (no caso, uma grade computacional). Os vértices

do grafo de recursos representam os recursos propriamente ditos e as arestas representam os elos de comunicação entre os recursos computacionais. Cada vértice possui um peso associado a ele, representando a capacidade computacional do recurso, e cada aresta possui um peso associado a ela, representando a capacidade de comunicação daquele elo.

Na explicação do algoritmo, a seguinte notação é usada [53]: seja G um grafo cujas arestas e os nós tenham pesos associados a eles. G_i é o peso associado ao vértice i, enquanto que $G_{i,j}$ é o peso da aresta que liga i à j. G^i é o grafo isomórfico a G cujo peso do vértice i é 1 e todos os outros vértices e arestas possuem peso 0. $G^{i,j}$ é o grafo isomórfico a G cujo peso da aresta entre i e j é 1 e os pesos de todas as outras arestas e dos vértices é 0. Dados dois grafos isomórficos, G e V, a soma entre deles, $G + V$, é definida como sendo um grafo isomórfico a ambos cujo peso de cada vértice é a soma dos pesos dos vértices correspondentes em G e V e o peso de cada aresta é a soma dos pesos das arestas correspondentes em G e V. De maneira similar, a subtração e a divisão de grafos isomórficos são definidas. Além disso, a multiplicação de grafo por uma constante escalar tem como resultado um grafo cujo peso dos vértices e arestas é igual ao peso anterior multiplicado pela constante.

O objetivo do algoritmo é maximizar a vazão (*throughput*) do sistema, ou seja, o número de tarefas completadas por unidade de tempo. Para isso, dado um mapeamento de tarefas em processadores, o algoritmo calcula a quantidade de processamento que cada processador deve realizar para que as tarefas executem uma unidade de trabalho, bem como a quantidade de dados que devem ser transferidos em cada *link* de comunicação para que as tarefas possam continuar seu processamento. Dividindo a quantidade obtida para cada processador e para cada *link* pela sua respectiva capacidade, temos um valor denominado de ocupação. Cada vértice e cada aresta têm sua ocupação e o algoritmo considera que o maior valor de ocupação de todo o grafo corresponde ao tempo necessário para que todas as tarefas executem uma unidade de trabalho. Assim, a ocupação é definida como sendo o inverso da vazão do sistema e o algoritmo busca minimizar a ocupação.

Para formalizar esse conceito, define-se o conceito de grafo de carga (*load graph*) de um mapeamento. Dado um grafo de tarefas $G = \langle V_G, E_G \rangle$, um grafo de recursos (processadores) $P = \langle V_P, E_P \rangle$ e um mapeamento $m : G \to V$, o grafo de carga do sistema é definido como sendo

$$L(G, P, m) = \sum_{t \in V} G_t \cdot P^{m(t)} + \sum_{(s,t) \in E} G_{s,t} \cdot P^{m(s),m(t)}$$

O grafo de ocupação (*occupancy graph*) correspondente ao mapeamento m é definido como sendo a divisão da carga em cada vértice e aresta por sua respectiva capacidade, ou seja,

$$O(G, P, m) = \frac{L(G, P, m)}{P}$$

Considerando que, para um grafo X, $max(X)$ é igual ao maior peso entre todos os pesos de nós e arestas do grafo, o algoritmo visa achar o mapeamento

m tal que $max(O(G, P, m))$ seja mínimo. A figura 4.1 ilustra os conceitos de grafo de tarefas, grafo de recursos, grafo de carga e grafo de ocupação para o mapeamento $m = \{(a, p), (b, q), (c, q), (d, r), (e, s)\}$.

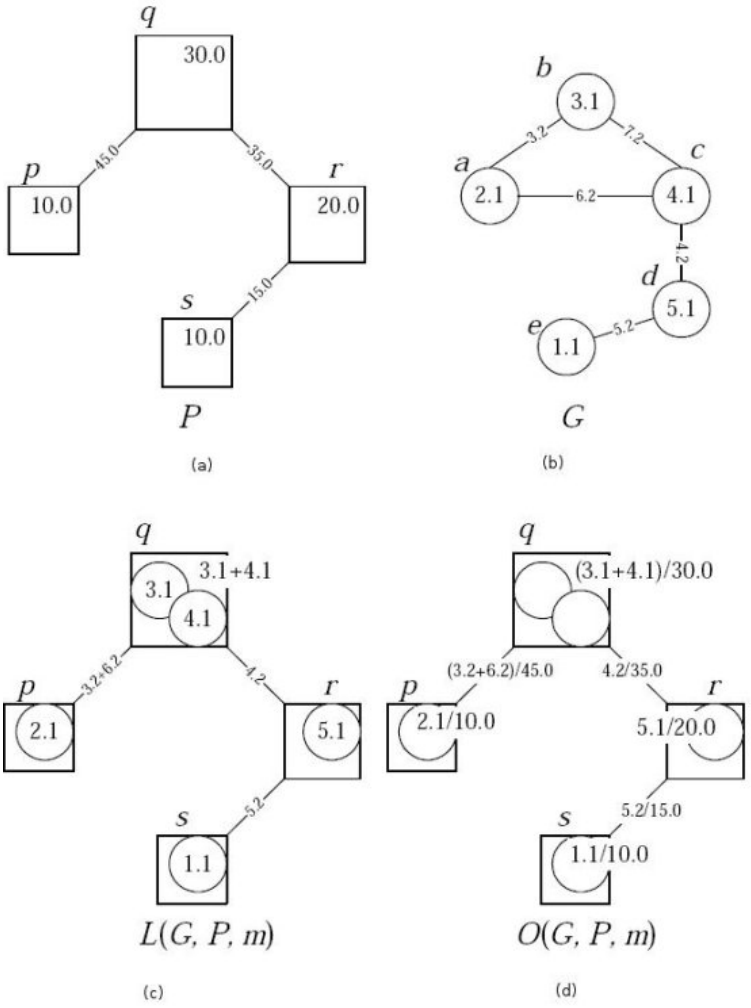


Figura 4.1: (a) Grafo de recursos P. (b) Grafo de tarefas G. (c) Grafo de carga para o mapeamento $m = \{(a, p), (b, q), (c, q), (d, r), (e, s)\}$. (d) Grafo de ocupação para o mapeamento $m = \{(a, p), (b, q), (c, q), (d, r), (e, s)\}$. (adaptado de [53])

A figura 4.2 define a estrutura geral do processo de escalonamento. O algoritmo inicia criando uma ordem total entre os vértices através do procedimento *agrupar*. Em seguida, cria um mapeamento inicial a partir de um mapeamento vazio, usando o procedimento *mapear_tarefas*. A partir daí, tenta fazer um melhoramento iterativo do mapeamento gerado, usando o procedimento *melhorar*.

O procedimento *agrupar* está definido no algoritmo 4.3. A fim de obter o ordenamento total entre os nós do grafo, o algoritmo da figura 4.3 usa uma heurística para criar grupos (*clusters*) de nós. As heurísticas de agrupamento de nós costumam ser bastante utilizadas em grafos de tarefas quando se tem por objetivo reduzir os custos de comunicação entre as tarefas, de modo que as tarefas que se comunicam muito tendem a ficar no mesmo grupo e serem alocadas a um mesmo

```
/* G = ⟨V_G, E_G⟩ é um grafo de tarefas
/* P = ⟨V_P, E_P⟩ é um grafo de recursos
<_G= agrupar(G);
m = {};
m = mapear_tarefas(m);
repetir {
   m' = m;
   m = melhorar(m');
} enquanto(O(G, P, m) < O(G, P, m'))
```

Figura 4.2: Estrutura geral do algoritmo proposto em [53]

processador [23].

```
agrupar(H) {
   T = agrupar_recursivo(H);
   <_H= ordenamento obtido através do percorrimento em profundidade do grafo;
    retornar <_H;
}
agrupar_recursivo(H) {
/*H = (V, E) é um subgrafo do grafo de tarefas
    Se( V possuir apenas um elemento v ) {
       retornar a folha v;
    } senão {
      H_1, ..., H_n = obter_grupos(H);
       retornar os nós dados por
         agrupar_recursivo(H_1), ..., agrupar_recursivo(H_n);
    }
}
obter_grupos(H) {
    repetir {
       Escolher ao acaso dois nós s e t do grafo;
       Encontrar o menor caminho entre s e t;
       Decrementar os pesos das arestas do caminho por uma constante pequena
△;
       Remover as arestas que ficarem com peso 0 ou negativo;
    } até que o grafo se torne desconectado;
    retornar os grupos obtidos quando o grafo se tornar desconectado;
}
```

Figura 4.3: Procedimento *agrupar*

O mapeamento das tarefas nos processadores é realizado pelo procedimento *mapear_tarefas*, mostrado pela figura 4.4. O procedimento tenta minimizar a ocupação do sistema, sendo que a ocupação é considerada como sendo o maior entre os seguintes valores:

- A ocupação atual do sistema, dada por $O(G, P, m)$

- Uma ocupação hipotética que é estimada assumindo-se que todas as tarefas que ainda não foram mapeadas sejam perfeitamente mapeadas nos processadores que ainda não receberam tarefas, ignorando-se os custos de comunicação, dada por $O_{comp}(G, P, m, Q)$, onde Q é o subconjunto de P formado pelo processadores ainda não utilizados

- Duas ocupações hipotéticas induzidas pela comunicação. Para um processador p e para um grafo de tarefas T, pode-se definir uma ocupação induzida pela comunicação denotada por $Q_{\rightarrow}(G, P, m, T, q)$ como sendo a quantidade total de comunicação das tarefas em T que chega até as tarefas alocadas ao processador q. De maneira similar, pode-se definir $Q_{\leftarrow}(G, P, m, q, T)$ como sendo a quantidade total comunicação que parte das tarefas em q e chega às tarefas em T.

---

mapear_tarefas(m) {
  $Q = V_P$;
  enquanto($Q \neq \{\}$) {
    q = um processador $\in Q$;
    $Q = Q - \{q\}$;
    $alocar\_tarefas(m, q, Q)$;
  }
}
alocar_tarefas(m,q,Q) {
/* Move algumas das tarefas ainda não mapeadas que estão em m para o processador q, levando em consideração a comunicação e o balanceamento da carga de q e Q */
  $U_{m_0} = \{t | t$ *é uma tarefa ainda não mapeada em* $m\}$
  para $i = 1$ até $|U_{m_0}|$ faça {
    t = a menor tarefa $\in U_{m_{i-1}}$ segundo a ordem dada pelo procedimento *agrupar*;
    $m_i = m_{i-1}[t/q]$ /*mapeia t em q;
    $U_{m_i} = U_{m_{i-1}} - \{t\}$;
    $O = O(G, P, m_i)$;
    $O_{comp} = O_{comp}(G, P, m_i, Q) = \frac{\sum_{t \in U_{m_i}} G_t}{\sum_{p \in Q} P_p}$; /*$O_{comp} = \infty$ se $Q = \{\}$
    $O_{\rightarrow} = O_{\rightarrow}(G, P, m_i, U_{m_i}, q) = \frac{\sum_{x \in U_{m_i}, m_i(y) = q} G_{x,y}}{\sum_{(p,q) \in E_p} P_{q,p}}$;
    $O_{\leftarrow} = O_{\leftarrow}(G, P, m_i, q, U_{m_i}) = \frac{\sum_{x \in U_{m_i}, m_i(y) = q} G_{y,x}}{\sum_{(p,q) \in E_p} P_{q,p}}$;
    $M_i = max(O, O_{comp}, O_{\rightarrow}, O_{\leftarrow})$;
  }
  Encontrar o i tal que M_i seja mínimo ($i = 1, \ldots, |U_{m_0}|$);
  Eventuais empates são resolvidos selecionando-se o maior i;
  retornar m_i;
}

Figura 4.4: Procedimento mapear_tarefas

Por fim, o algoritmo que tenta melhorar iterativamente o mapeamento m é dado na figura 4.5. Esse algoritmo inicia multiplicando a ocupação do sistema por valor pré-definido (o algoritmo usa o valor 0,75 [53]). A partir daí busca-se remover tarefas até que o mapeamento parcial obtido forneça uma ocupação menor do que a ocupação inicial multiplicada pelo valor pré-definido e então gerar um novo mapeamento. Nessas iterações, o algoritmo realiza os seguintes passos:

- Procura por vértices e arestas do grafo de recursos cuja ocupação seja maior do que a ocupação atual multiplicada pelo valor pré-definido;
- Caso encontre algum vértice, encontrar todas as tarefas que tenham sido mapeadas no processador representado por ele. Entre essas tarefas, remover a tarefa que tenha o maior custo;
- Caso encontre alguma aresta, encontrar os pares de tarefas s e t tais que o caminho entre os processadores que receberam s e t contenha a aresta encontrada e pelo menos uma das tarefas ainda não tenha sido removida. Entre todos os pares de tarefas encontradas, selecionar aquele que apresente o maior custo de comunicação e removê-lo.

---

melhorar(m) {
  $m = remover\_tarefas(m)$;
  $m = mapear\_tarefas(m)$;
}
remover_ tarefas(m) {
  $o = 0,75 \cdot max(O(G, P, m))$; /*O algoritmo usa o valor 0,75
  $D = \{\}$ /*Conjunto de tarefas removidas
  enquanto$(max(O(G, P, m - D)) > o)$ {
    $L = L(G, P, m - D)$;
    encontrar $p \in P$ e $q \in P$ tais que $L_{p,q}/P_{p,q} > o$;
    se encontrar {
      selecionar $s, t \in V_G$ tais que ($s \notin D$ ou $t \notin D$), $P_{p,q}^{m(s),m(t)} = 1$ e $G_{s,t}$ seja mínimo;
      $D = D + \{(s, m(s)), (t, m(t))\}$;
    } else {
      deve haver $p \in P$ tal que $L_p/P_p > o$;
      selecionar $s \in V_G$ tal que $s \notin D$, $m(s) = p$ e G_s seja mínimo;
      $D = D + \{(s, m(s))\}$;
    }
  }
  retornar $m - D$;
}

Figura 4.5: Procedimento *melhorar*

4.2 Alocação e balanceamento de carga de computações iterativas

Em [39] é apresentado um algoritmo que visa mapear as computações de algoritmos iterativos em *clusters* heterogêneos, ou seja, o trabalho é voltado para o mapeamento de tarefas que realizam computações independentes durante um período de tempo e então trocam informações entre si. Algoritmos paralelos que tratam da atualização de valores de uma matriz se encaixam nessa descrição.

O algoritmo assume uma plataforma alvo composta por processadores de diferentes velocidades que se comunicam por *links* de rede com diferentes larguras de banda. De maneira geral, o algoritmo proposto visa selecionar os processadores que serão usados, organizando-os em uma topologia de anel virtual, e atribuir rotas de comunicação para as mensagens que serão trocadas entre um processador e seu sucessor e/ou predecessor no anel virtual.

Duas observações podem ser feitas com base nessa descrição de alto nível. Em primeiro lugar, o algoritmo assume que o padrão de comunicação entre as tarefas é tal que, ao final do processo de mapeamento, um processador só precisará se comunicar com outros dois processadores: o seu sucessor e o seu predecessor no anel virtual. Em segundo lugar, o algoritmo assume que existe liberdade suficiente para se decidir por quais *links* de rede as mensagens trocadas pelos processadores deverão passar. Na prática, tal decisão é feita pelos algoritmos de roteamento implementados nos roteadores da rede. Para que um algoritmo de mapeamento de tarefas tome tais decisões será preciso a utilização de redes que implementem protocolos capazes de permitir maior flexibilidade na especificação de qualidade de serviço [39].

O algoritmo de Alocação assume as seguintes características:

- O poder computacional de cada processador é indicado por w_i e indica o custo de se processar uma tarefa de tempo unitário.

- A largura de banda de um *link* e entre dois processadores é denotada por b_e. O custo de se transmitir uma mensagem de tamanho D_e de p_i para p_j através de e é $\frac{D_e}{b_e}$. Se mais de uma mensagem compartilha um mesmo *link*, cada mensagem recebe uma fração x da largura de banda do *link*, sendo que a capacidade total do *link* não pode ser excedida.

- Assume-se que a cada passo da computação iterativa sendo mapeada exista uma quantidade D_w de trabalho a ser realizado. Um processador p_i realiza uma parte $\alpha_i.D_w$ dessa quantidade total, sendo que $\alpha_i \geq 0$ e $\sum_{i=1}^{|P|} \alpha_i = 1$.

- Assume-se que seja possível determinar como as mensagens serão roteadas através do sistema distribuído. Supõe-se que entre dois processadores p_i e p_j existem k *links* de comunicação $e_1, e_2, ..., e_k$. Uma mensagem de tamanho D_e que passe através de um *link* e_m receberá uma porção f_m da largura de banda b_{e_m} desse *link* e irá demorar $\frac{D_e}{b}$ unidades de tempo, onde $b = min_{1 \leq m \leq k} f_m$.

- Os processadores selecionados são organizados em um anel virtual e, após um processador realizar o trabalho necessário em uma iteração, ele comunica-se com seu sucessor e o seu predecessor no anel. O custo de comunicação entre um processador p_i e o seu sucessor no anel é denotado por $c_{i,succ(i)}$ e é igual a $\frac{1}{min(s_{i,m})}$, onde $s_{i,m}$ é a porção alocada da largura de banda de um *link* e_m que esteja no caminho entre o processador p_i e o seu sucessor no anel virtual. O custo de comunicação $c_{i,pred(i)}$ entre o processador p_i e o seu predecessor no anel virtual é definido de maneira semelhante.

Com base nessas definições, define-se o custo total de uma iteração da execução de um algoritmo iterativo de acordo com a equação 4.1:

$$T_{step} = max_{1 \leq i \leq |P|} \Pi i[\alpha.D_w.w_i + D_e.(c_{i,pred(i)} + c_{i,succ(i)})] \tag{4.1}$$

onde $\Pi i[x] = x$ se p_i estiver participando das computações iterativas ou 0 caso contrário. O objetivo do algoritmo de Alocação é determinar a melhor maneira de se selecionar q processadores entre os $|P|$ processadores disponíveis, atribuir computações a eles, organizá-los ao longo de um anel e compartilhar a largura de banda da rede entre as mensagens que serão trocadas, de modo a minimizar o tempo de execução total de um passo da execução de uma computação iterativa.

Assim, o problema é colocado sob a forma de um problema de otimização, onde dados $|P|$ processadores p_i, cada qual com um poder computacional dado por w_i, $|E|$ *links* de comunicação e_m, cada um com uma largura de banda total igual a b_{e_m}, uma quantidade total de trabalho igual a D_w e um volume de comunicação igual a D_e, a cada passo de uma computação iterativa deve-se determinar

$$T_{step} = min_{1 \leq q \leq |p|, \sigma \in \Theta_{q,p}, \sum_{i-1}^{q} \alpha_{\sigma(i)} = 1} \{max_{1 \leq i \leq q} (\alpha_{\sigma(i)}.D_w.w_{\sigma(i)} + D_e.(c_{\sigma(i),\sigma(i-1 mod q)} + c_{\sigma(i),\sigma(i+1 mod q)}))\} \tag{4.2}$$

onde $\Theta_{q,p}$ denota o conjunto de funções $\sigma : [1..q] \rightarrow [1..p]$ que indexam os q processadores selecionados para formar o anel virtual. Demonstra-se que esse problema de otimização é NP-completo [39] e por isso o algoritmo de Alocação proposto utiliza heurísticas para construir o anel virtual e alocar a largura de banda para as mensagens trocadas entre os processadores do anel.

A heurística de construção do anel inicia selecionando o melhor par de processadores entre os $|P|$ processadores existentes e, iterativamente, adiciona outros processadores. Levando em consideração a equação 4.1 e o fato de que $\sum_{i=1}^{q} \alpha_i = 1$, define-se

$$w_{cummul} = \frac{1}{\sum_{i-1}^{q} \frac{1}{w_i}} \tag{4.3}$$

e pode-se reescrever a equação 4.1 da seguinte forma:

$$T_{step} = D_w.w_{cummul}(1 + \frac{D_e}{D_w} \sum_{i=1}^{q} \frac{c_{i,i-1} + c_{i,i+1}}{w_i}) \tag{4.4}$$

Para cada processador p_i a ser iterativamente adicionado ao anel deve-se descobrir onde ele deve ser inserido. Para cada par de processadores p_j e p_k, deve-se calcular o custo de inserir p_i entre p_j e p_k, de maneira que são escolhidos o processador p_i e o par p_j, p_k que minimizem o custo, atualizando então o valor de T_{step}, conforme definido na equação 4.4.

Para que seja calculado o custo de se inserir p_i entre p_j e p_k, utiliza-se uma outra heurística para se determinar a alocação de largura de banda dos *links* entre esses processadores. A heurística substitui os custos de comunicação entre p_j e p_k, nos dois sentidos, pelos custos de comunicação entre p_j e p_i e entre p_i e p_k. Como a comunicação ocorre nos dois sentidos, existem 4 custos que são calculados pela heurística. O primeiro passo da heurística é devolver a largura de banda que já havia sido alocada para o caminho entre p_j e p_k. Para isso, aplica-se um algoritmo de menor caminho no grafo que representa o sistema distribuído (processadores e *links*) [39]. Se existem *links* compartilhados, a heurística usa um método analítico para computar as porções da largura de banda que minimizem a equação 4.4 e que serão alocadas a cada novo caminho. Atualizam-se então os novos custos de comunicação.

Depois que os novos custos de comunicação são calculados, atualiza-se o valor da equação 4.3 adicionando-se o processador novo na fórmula. Por fim, atualiza-se a equação 4.4, retirando-se do somatório os dois termos correspondentes ao caminho entre os processadores antigos e incluindo os termos novos calculados anteriormente.

4.3 Escalonamento de aplicações *parameter sweep* em grades computacionais

Em [10], é proposto o escalonamento de aplicações do tipo *parameter sweep* em ambientes de grades computacionais. O escalonamento dessas aplicações é abordado através de heurísticas integradas em um algoritmo adaptativo que faz parte de um *framework* chamado PST (*Parameter Sweep Template*).

O trabalho proposto considera que as aplicações *parameter sweep* são aplicações compostas por tarefas independentes, ou seja, que não se comunicam e nem possuem relação de precedência (seção 3.4.2). Porém, admite que essas tarefas podem compartilhar arquivos de entrada. Por outro lado, uma grade computacional é modelada como sendo um conjunto de *clusters* que são acessíveis ao usuário por meio de *links* de rede distintos. Esse modelo de grade computacional é bastante simplificado, em especial porque:

- não leva em consideração a contenção dos *links* de rede;
- não leva em consideração a contenção no acesso a arquivos dentro de um mesmo *cluster*; e
- desconsidera a topologia de rede, na medida em que considera que os *clusters* que o usuário acessa não estão ligados entre si, o que impede certas otimizações no que diz respeito à transferência de arquivos entre os clusters.

Apesar dessas limitações, os autores do trabalho acreditam que os resultados por eles obtidos permanecem relevantes mesmo em condições de uso mais reais.

O algoritmo proposto gera um plano para atribuir transferências de arquivos para os *links* de rede e de tarefas computacionais para os computadores dos *clusters*. Os momentos em que o algoritmo é executado são chamados de *eventos de escalonamento*. Em cada evento, assume-se que o escalonador tem o conhecimento de:

- estado da topologia da grade computacional (número de *clusters*, número de computadores em cada *cluster* e carga nas CPUs e na rede);
- número e localização de cópias dos arquivos de entrada; e
- lista das transferências de arquivos e de tarefas que já foram completadas ou que estão em andamento

O algoritmo proposto é apresentado na figura 4.6. O ponto principal do algoritmo está nas linhas 7 à 9, ou seja, no *loop* em que ocorre a atribuição heurística de tarefas.

```
1. Determinar o próximo evento de escalonamento
2 .Criar um gráfico de Gantt, G
3 .Para cada tarefa e transferência de arquivo em andamento
4.      Estimar o seu tempo de término
5.      Preencher os espaços correspondentes no gráfico de Gantt G
6. Selecionar um subconjunto T de tarefas que ainda não tenham sido iniciadas
7. Repetir
8.      Atribuir heuristicamente tarefas, preenchendo os espaços em G
9. até que cada computador possua trabalho suficiente
10. Converter G em um plano
```

Figura 4.6: Esqueleto do algoritmo de escalonamento de aplicações *parameter sweep*

Para que as tarefas sejam atribuídas aos recursos computacionais existentes, são propostas 4 heurísticas diferentes. Três delas são adaptações de heurísticas já existentes, enquanto uma delas é uma extensão de uma outra heurística. Todas as heurísticas trabalham computando o tempo mínimo de término de execução das tarefas, ou MCT (*Minimum Completion Time*).

A primeira heurística adaptada é a heurística *min-min*. No trabalho em questão, essa heurística usa o mínimo MCT como métrica, de modo a privilegiar a execução de tarefas que possam ser completadas mais cedo. A segunda heurística apresentada, chamada de *max-min*, utiliza como métrica o máximo MCT, de modo a tentar permitir a sobreposição da execução de tarefas longas com tarefas curtas. Já a terceira heurística, *sufferage* utiliza, para cada tarefa, a diferença entre o melhor MCT e o segundo melhor MCT, de maneira a favorecer a execução de tarefas que seriam mais prejudicadas se não fossem executadas em um determinado recurso. Essas três heurísticas já haviam sido apresentadas em um

trabalho anterior [40]. Em [10], elas foram adaptadas para levar em conta as transferências de arquivos de entrada e saída, no momento do cálculo do MCT, e o fato de que alguns dos arquivos necessários podem já estar disponíveis em determinado local, não sendo necessária a sua transferência.

A quarta heurística descrita pelo trabalho em questão é uma extensão da heurística de *sufferage*, motivada pelos resultados pouco satisfatórios obtidos. Essa nova heurística é chamada de *XSufferage* e trabalha com o conceito de MCT no nível do *cluster*, ou seja, computa-se o menor MCT de todos os recursos de uma determinado *cluster*. Para cada tarefa e para cada *cluster*, computa-se o MCT levando-se em consideração apenas os recursos do *cluster* em questão, chamando esse valor de MCT de *cluster*. Em seguida, computa-se o valor da diferença entre o melhor e o segundo melhor MCT do *cluster*. Esse resultado é chamado de valor de *sufferage* de *cluster*. A tarefa que possua o maior valor de *sufferage* de *cluster* é atribuída ao recurso que permita o menor MCT dentro do *cluster* que obtenha o menor MCT de *cluster*.

4.4 Escalonamento de tarefas considerando contenção de comunicação

Em [49] propõe-se uma abordagem para o escalonamento de tarefas que leve em consideração um modelo mais realístico do sistema distribuído. Em particular, recorre-se ao conceito de um grafo de topologia, que modela a rede do sistema e permite a avaliação de questões relativas à contenção dos *links* de comunicação. O algoritmo de escalonamento proposto é voltado para aplicações que sejam representadas por um grafo acíclico dirigido (DAG) e, além da atribuição de tarefas aos processadores, busca também atribuir as arestas do DAG aos *links* de comunicação existentes.

O desenvolvimento de um algoritmo que leve em consideração a contenção da comunicação nos *links* de comunicação é justificado pelo fato de que um modelo mais realístico permite que se tome decisões mais precisas. Os modelos tradicionais usados para abordar o problema de escalonamento de tarefas apresentam uma série de simplificações que não são percebidas na prática, incluindo:

- A existência de processadores dedicados à tarefas sendo escalonadas;
- Custo de comunicação local igual a zero;
- Comunicação entre processadores feita por subsistemas dedicados, sem o envolvimento do processador;
- Ausência de contenção nos *links* de comunicação, com todas as comunicações ocorrendo concorrentemente; e
- Topologia de rede totalmente conectada.

O modelo de rede proposto em [49] elimina as duas últimas restrições acima, estabelecendo o conceito de um grafo de topologia, capaz de modelar de maneira

mais precisa diferentes topologias de rede. O grafo de topologia pode representar, além dos processadores da rede, *switches* de comunicação e *links* unidirecionais, bidirecionais e barramentos.

Entretanto, a maior precisão na modelagem do sistema acarreta uma necessidade de maior controle quanto às decisões a serem tomadas. Em particular, para que a contenção nos *links* de comunicação seja levada em consideração, o algoritmo proposto assume que seja possível controlar o roteamento das mensagens no sistema, algo que não é usual.

O algoritmo proposto é na verdade uma adaptação da heurística de escalonamento de lista (*list scheduling*) de modo a considerar a contenção nos *links* de comunicação. No escalonamento de lista, as tarefas, representadas por nós no DAG, são ordenadas de acordo com algum valor de prioridade. Em seguida, as tarefas são atribuídas aos processadores que permitam os menores tempos de término de execução das tarefas. Esse procedimento genérico é ilustrado na figura 4.7.

Ordenar as tarefas $\tau_i, i = 1, ..., |T|$ de acordo com algum esquema de prioridade e com as relações de precedência
Para cada tarefa τ_i
  Encontrar o processador p_j que permita o menor tempo de término de execução para τ_i
  Atribuir τ_i a p_j

Figura 4.7: Procedimento genérico de escalonamento de lista

A partir do momento em que se leva em consideração a contenção dos *links* de comunicação, a operação de atribuição de uma tarefa a um processador deve levar em conta esse fator. Por isso, o procedimento genérico de escalonamento de listas, ilustrado na figura 4.7, é adaptado de maneira que a sua última operação (atribuir τ_i a p_j) seja feita de acordo com o procedimento listado na figura 4.8.

Para cada tarefa τ_k que seja antecessora de τ_i, de acordo com alguma ordem definida, faça:
  Se o processador ao qual τ_k foi atribuída for diferente de p_j, então
    Encontrar uma rota R a partir do processador ao qual τ_k foi atribuída até p_j
    Atribuir a aresta $e_{k,j}$ do DAG para a rota R
Atribuir τ_i a p_j

Figura 4.8: Escalonamento da tarefa τ_i no processador p_j levando em consideração a contenção nos *links* de comunicação

No algoritmo da figura 4.8, a rota R é encontrada através da aplicação de um algoritmo de roteamento da rede. Assim, o tempo de execução do algoritmo vai estar atrelado ao tempo de execução desse algoritmo de roteamento. O trabalho de [49] estabelece as definições técnicas necessárias à modelagem do sistema distribuído e da modelagem da contenção nos *links* de comunicação e estabelece que a complexidade do algoritmo de escalonamento proposto é de

$O(V^2 + PE^2O(Roteamento))$. Obviamente, uma desvantagem desse algoritmo é a dependência quanto ao algoritmo de roteamento usado, que não costuma ser controlado pelo escalonador.

4.5 Distribuição de trabalho em ambientes heterogêneos para algoritmos paralelos de programação dinâmica

Programação dinâmica é uma técnica de projeto de algoritmos usada para resolver problemas através da combinação de soluções de subproblemas [16]. Em algoritmos de programação dinâmica, usa-se uma matriz bidimensional para se armazenar as soluções dos subproblemas e a solução final do problema original é armazenada na última entrada da matriz.

Algoritmos paralelos de programação dinâmica funcionam atribuindo a vários processadores a responsabilidade de processar diferentes partes da matriz bidimensional usada. Assim, o trabalho total de se calcular a solução do problema é dividido entre os processadores, que processam as suas partes da matriz e comunicam seus resultados para os outros processadores.

O trabalho apresentado em [18] visa analisar a possibilidade de se incluir técnicas de otimização automatizadas no projeto de algoritmos paralelos de programação dinâmica, determinando de maneira automática alguns parâmetros como o número de processadores a serem usados e o número de processos por processador, por exemplo. Para isso, considera-se que o tempo de execução de um algoritmo paralelo é dado por uma função do tamanho do problema e de alguns parâmetros do sistema e do algoritmo. Entre os parâmetros de sistema, podemos citar o custo de determinada operação nos processadores do sistema (t_c), o tempo de *start-up* da rede (t_s) e o tempo de transmissão de uma palavra (t_w). Já entre os parâmetros do algoritmo, encontram-se valores como o número de processadores usados e o tamanho de sub-problema que será atribuído a cada processador. Assume-se também que o valor dos parâmetros de sistema pode ser influenciado pelos parâmetros do algoritmo. Assim, o tempo de execução de um algoritmo paralelo é modelado pela função 4.5

$$t(s, t_c(s,p,b,d), t_s(s,p,b,d), t_w(s,p,b,d)) \tag{4.5}$$

onde s é o tamanho do problema, p é o número de processadores usados, b é o tamanho de bloco (sub-problema) distribuído para os processadores e $d = (d_1, ..., d_p)$ é um vetor contendo o número de processos por processador (por exemplo, d_1 processos executando no processador p_1). Uma vez que o sistema sendo considerado é heterogêneo, os valores dos parâmetros de sistema, são representados por vetores ou então por matrizes. Por exemplo, $t_c = (t_{c1}, ..., t_{cn})$ e t_w e t_s são matrizes de tamanho pxp.

Propõe-se que se determine o número de processadores, o número de processos por processador e a atribuição de processos a processadores através da construção de uma árvore de soluções [18]. Nessa árvore, cada nível (a partir do nível 1) representa um processo e cada nó representa um processador, tal qual ilustra

a figura 4.9. Assim, um caminho da raiz da árvore até um determinado nó representa uma solução que determina os valores acima mencionados.

Sabendo-se que o problema de se encontrar uma solução ótima é NP-completo [29], propõe-se a utilização de estratégias que eliminem nós na árvore de solução. Entretanto, para esse problema, o trabalho apresentado possui duas desvantagens:

- Não apresenta um método genérico de eliminação de nós. O trabalho apresenta dois experimentos e a estratégia de exploração da árvore de soluções construída é dependente do sistema sendo utilizado, de maneira que, para descrever a estratégia, gasta-se uma boa parte do tempo explicando as características dos sistemas usados nos experimentos; e

- é aplicado apenas a um problema específico, chamado de "problema das moedas", que visa determinar como se obter uma quantidade C usando-se o menor número de moedas, dadas a quantidade C e moedas de n tipos $(v_1, ..., v_n)$, com uma certa quantidade de moedas de cada tipo [18]. Essa característica é limitante porque são feitas comparações com estimativas de tempo de execução baseadas em um modelo para esse problema.

Entretanto, a idéia de se automatizar a determinação de certos parâmetros de algoritmos paralelos é interessante e existem diversos problemas reais que podem ser resolvidos por meio de algoritmos de programação dinâmica.

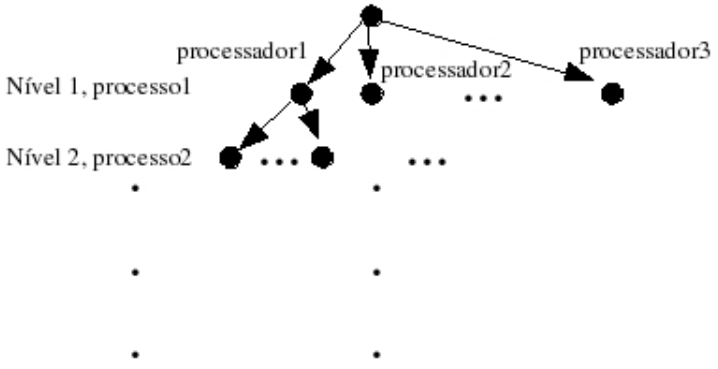


Figura 4.9: Exemplo de árvore de soluções

4.6 Comparação dos algoritmos apresentados

A tabela 4.1 mostra um quadro comparativo com diversas características dos algoritmos apresentados, sendo que destaca-se o fato de que todos buscam utilizar informações sobre a aplicação e/ou sobre o sistema para determinar as estratégias de alocação de tarefas.

Referência	**Classe de aplicação**	**heurística**	**Sistemas heterogêneos**	**Tipo de Sistema**	**parâmetros necessários**
[53] seção 4.1	grafo de iteração	*clustering*	Sim	LAN MAN WAN	número de *hosts*; capacidade dos *hosts*; topologia da rede; capacidade dos links; número de tarefas; custo das tarefas; comunicação entre tarefas;
[39] seção 4.2	grafo de iteração	algoritmo *greedy*	Sim	LAN MAN WAN	número de *hosts*; capacidade dos *hosts*; topologia da rede; capacidade dos *links*; algoritmo de roteamento; custo de cada iteração; custo de comunicação
[10] seção 4.3	*Parameter Sweep*	Max-min; Min-min; Sufferage; XSufferage	Sim	Grid	Número de tarefas; Número de *hosts*; Duração estimada das tarefas; Capacidade dos *links*
[49] seção 4.4	*workflow* (DAG)	*List Scheduling*	Sim	LAN	DAG da aplicação; custo das tarefas; comunicação entre tarefas; número de *hosts*; capacidade dos *hosts*; topologia da rede; capacidade dos *links*; algoritmo de roteamento
[18] seção 4.5	grafo de iteração	Percorrimento de árvore de soluções	Sim	LAN	número de *hosts*; capacidade dos *hosts*; tempo de uma operação; *start-up time*; *word sending time*

Tabela 4.1: Quadro comparativo entre alguns dos algoritmos apresentados

Capítulo 5

Comparação de Seqüências na Biologia Computacional

5.1 Introdução

A comparação de seqüências é considerada uma das operações mais importantes da área de biologia computacional, sendo usada como base para muitas outras operações. A comparação de seqüências de DNA busca determinar o grau de similaridade entre duas seqüências dadas, identificando quais partes dessas seqüências são mais similares. Dentre os problemas da biologia computacional que fazem uso da comparação de seqüências, podemos citar [47]:

- Comparação dos resultados do seqüenciamento de um gene realizado por dois laboratórios diferentes: neste caso, há duas seqüências praticamente iguais, com apenas algumas pequenas diferenças isoladas, e busca-se os locais dessas pequenas diferenças.
- Montagem de fragmentos de DNA: dadas duas seqüências, determinar se existe algum prefixo de uma delas que seja similar a um sufixo da outra.
- Busca por similaridades locais usando grandes bancos de dados de seqüências: dados uma seqüência e um banco de dados contendo várias outras seqüências, comparar a seqüência dada com todas as outras a fim de determinar se existe uma *substring* comum entre a seqüência dada e alguma das outras seqüências.

O uso dos computadores neste tipo de problema pode ser justificado pelo fato de que boa parte das comparações de seqüências de DNA são feitas usando-se um grande volume de dados. Além disso, mesmo nos casos em que seja viável a realização de uma comparação manual, o uso de computadores, além de tornar o trabalho mais eficiente, pode ajudar a descobrir resultados que não tenham sido notados manualmente.

Para se abordar o problema das comparações de seqüências, as seguintes definições costumam ser utilizadas [47]:

- Uma seqüência de DNA é representada por uma *string* sobre o conjunto finito A, C, G, T (correspondendo às bases Adenina, Citosina, Guanina e Timina).

- Um alinhamento entre duas seqüências é definido como sendo a inserção de espaços em posições arbitrárias das seqüências, de modo a torná-las do mesmo tamanho e permitindo que possam ser colocadas uma acima da outra de modo a criar uma correspondência entre bases (ou espaços) em uma das seqüências e bases (ou espaços) na outra. Além disso, assume-se que não pode haver uma correspondências entre um espaço em uma seqüência e um espaço na outra seqüência. Por exemplo, dadas as seqüências GACGG e GATCGG, um possível alinhamento entre elas seria:

```
GA_CGG
GATCGG
```

- O escore de um alinhamento é definido como sendo a soma dos escores das colunas do alinhamento, onde o escore de uma coluna é um valor atribuído a ela dependendo de seu conteúdo. Por exemplo, pode-se assumir que, se uma coluna do alinhamento possuir dois caracteres iguais, então tal coluna terá um escore de valor $+1$. Se a coluna possuir caracteres diferentes, então o valor será de -1. Se houver um espaço em uma das colunas, valor será de -2. Neste, podemos dizer que o alinhamento do item anterior tem um escore total de $1 + 1 - 2 + 1 + 1 + 1 = 3$.

- Similaridade entre duas seqüências: é definida como sendo o maior escore total obtido dos alinhamentos entre duas seqüências. O alinhamento que fornecer o maior escore é chamado de alinhamento ótimo.

5.2 Comparação Global de Seqüências

O alinhamento global de seqüências tem por objetivo obter um alinhamento (e conseqüentemente a similaridade) entre duas seqüências por inteiro. Existe um algoritmo ótimo que resolve o problema do alinhamento global, chamado de algoritmo de Needleman-Wunsch [43]. O algoritmo utiliza-se da técnica de programação dinâmica, uma vez que o espaço de solução do problema é muito grande (exponencial) para permitir uma solução eficiente baseada em uma enumeração exaustiva das soluções. Então, ao invés de tentar achar explicitamente todos os alinhamentos possíveis entre as seqüências dadas para depois selecionar o melhor, o algoritmo resolve o problema a partir da resolução de instâncias menores do mesmo problema. Mais especificamente, busca-se as similaridades entre prefixos arbitrários das duas seqüências, começando com prefixos menores e usando os resultados computados anteriormente para achar as similaridades entre prefixos maiores.

Considerando duas seqüências s e t, de tamanhos m e n, respectivamente, temos que o número de prefixos de cada uma dessas seqüências é dado por $n+1$ (para t) e $m+1$ (para s), incluindo o prefixo vazio. A matriz de similaridade A a ser construída possuirá dimensões $(m+1)$X$(n+1)$ e cada posição $A[i,j]$ irá conter a similaridade entre os prefixos $s[1..i]$ de s e $t[1..j]$ de t. O preenchimento da matriz utiliza regras de atribuição de escore, conforme discutido anteriormente e se baseia na observação de que, tendo alinhado os prefixos $s[1..i]$ e $t[1..j]$, para se construir o alinhamento do prefixo $s[1..i+1]$ com o prefixo $t[1..j+1]$, precisamos apenas considerar o caractere $s[i+1]$ de s e o caractere $t[j+1]$ de t. Para se alinhar esses caracteres, temos apenas 3 alternativas:

- Alinhar $s[i+1]$ com $t[j+1]$
- Inserir um espaço em s e alinhar esse espaço com $t[j+1]$
- Inserir um espaço em t e alinhar esse espaço com $s[i+1]$

Levando em conta a matriz A de similaridade entre s e t, temos então que a posição $A[i,j]$, ou seja a similaridade entre os prefixos $s[1..i]$ e $t[1..j]$, depende apenas das posições $A[i,j-1]$, $A[i-1,j]$ e $A[i,j]$. Existem as seguintes situações:

- Se a posição $A[i,j]$ for calculada a partir da posição $A[i-1,j-1]$, isso corresponde a ter alinhado $s[i]$ com $t[j]$
- Se a posição $A[i,j]$ for calculada a partir da posição $A[i-1,j]$, isso corresponde a ter alinhado $s[i]$ com um espaço inserido em t
- Se a posição $A[i,j]$ for calculada a partir da posição $A[i,j-1]$, isso corresponde a ter alinhado um espaço que foi inserido em s com $t[j]$

Dessa maneira, a fórmula 5.1 é utilizada para o preenchimento da matriz de similaridade.

$$A[i,j] = max \begin{cases} A[i,j-1]+p \\ A[i-1,j-1]+h(i,j) \\ A[i-1,j]+p \end{cases} \tag{5.1}$$

Na fórmula 5.1, p é a penalidade decorrente de se alinhar um caractere de uma seqüência com um espaço inserido na outra seqüência e $h(i,j)$ é uma função que retorna valores com base na comparação dos caracteres $s[i]$ e $t[j]$. Como mencionado anteriormente, alguns valores que costumam ser usados são -2 para p, -1 para $h(i,j)$ se $s[i] \neq s[j]$ e 1 para $h(i,j)$ se $s[i] = s[j]$.

Para o preenchimento da primeira linha da matriz, usa-se apenas a fórmula $A[i,j] = A[i,j-1]+p$, enquanto que para preencher a primeira coluna da matriz, usa-se apenas a fórmula $A[i,j] = A[i-1,j]+p$. A justificativa é que, tanto para a primeira linha quanto para a primeira coluna, estamos criando alinhamentos entre caracteres de uma das seqüências com o prefixo vazio da outra seqüência. Conseqüentemente, o valor da similaridade nas posições da primeira linha e da

primeira coluna da matriz sempre serão múltiplos da penalidade atribuída ao alinhamento entre um caractere e um espaço inserido.

Como exemplo do funcionamento do algoritmo, assumindo duas seqüências $s = GACGG$ e $t = GATCGG$, de tamanhos 5 e 6, respectivamente, teremos uma matriz $6X7$, tal qual é mostrado pela figura 5.1.

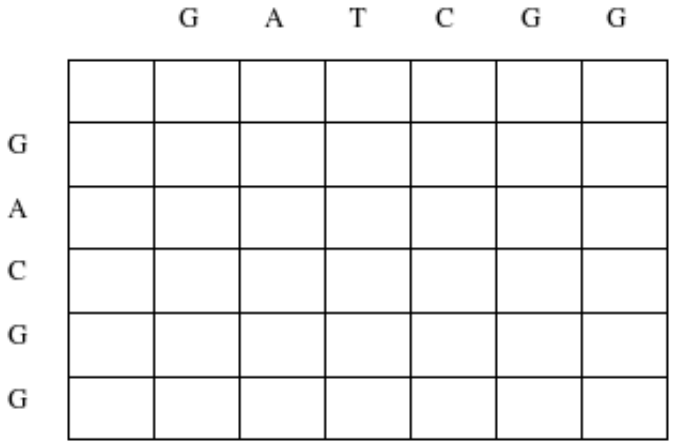


Figura 5.1: Matriz de similaridade entre $GACGG$ e $GATCGG$ (ainda não preenchida)

O preenchimento dessa matriz se dará obedecendo as regras descritas anteriormente. Primeiramente, serão preenchidas a primeira linha e a primeira coluna da matriz, de modo a apresentar a similaridade entre o alinhamento dos prefixos de uma das seqüências com o prefixo vazio da outra seqüência. A situação da matriz, após o preenchimento da primeira linha e da primeira coluna é mostrado na figura 5.2.

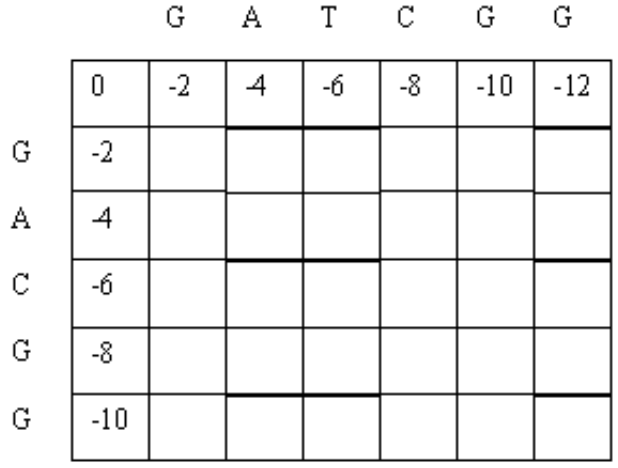


Figura 5.2: Matriz de similaridade entre $GACGG$ e $GATCGG$ (após o preenchimento da primeira linha e da primeira coluna)

Assume-se o valor de -2 para p (a penalidade por se alinhar um caractere com um espaço vazio). Nesse ponto é possível aplicar a fórmula para o preenchimento

das posições restantes da matriz. Considerando -1 o valor de $h(i,j)$ quando $s[i] \neq t[j]$ e 1 quando $s[i] = t[j]$, teremos a matriz representada na figura 5.3

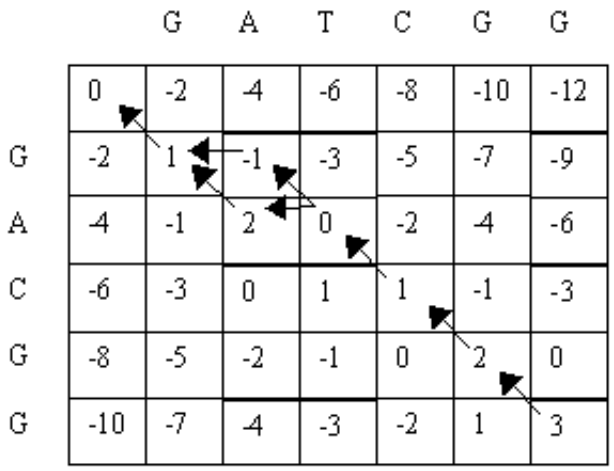


Figura 5.3: Matriz de similaridade entre $GACGG$ e $GATCGG$ (preenchida)

Cada posição $A[i,j]$ da matriz possui o valor da similaridade entre $s[1..i]$ e $t[1..j]$. Portanto, para se saber a similaridade entre as duas seqüências inteiras, basta observar o valor da posição $A[m+1, n+1]$, que no caso do exemplo em questão é 3. A figura 5.4 ilustra o procedimento para o preenchimento da matriz de similaridade.

$m \leftarrow |s|$
$n \leftarrow |t|$
for $i \leftarrow 0$ to m do
    $A[i,0] \leftarrow i * p$
for $j \leftarrow 0$ to n do
    $A[0,j] \leftarrow j * p$
for $i \leftarrow 0$ to m do
    for $j \leftarrow 0$ to n do
        $A[i,j] \leftarrow max(A[i-1,j]+p, A[i-1,j-i]+h(i,j), A[i,j-1]+p)$
return $a[m,n]$

Figura 5.4: Algoritmo para encontrar a similaridade entre duas seqüências (comparação global)

De posse da matriz de similaridade pode-se reconstruir o alinhamento entre as duas seqüências. Para isso, parte-se da posição $A[m+1, n+1]$ e caminha-se na matriz até a posição $A[1,1]$. As posições da matriz que farão parte do caminho podem ser deduzidas da fórmula que foi utilizada no preenchimento da mesma. Por exemplo, o valor 3 da posição $A(6,7)$ da matriz utilizada no exemplo foi originado, de acordo com a fórmula, a partir do valor 2 da posição $A(5,6)$. Isso significa que o caractere G de s foi alinhado com o caractere G de t. Continuando o processo obtém-se o alinhamento mostrado na figura 5.5.

GA_CGG
GATCGG

Figura 5.5: Alinhamento ótimo entre $GACGG$ e $GATCGG$

5.3 Comparação Local de Seqüências

A comparação local de seqüências, ao contrário da comparação global, tem por objetivo encontrar diversos alinhamentos entre subseqüências de duas seqüências dadas. Assim como no problema da comparação global, o espaço de soluções é grande demais para ser explorado por uma enumeração exaustiva das soluções. O algoritmo ótimo para resolução do problema é chamado de algoritmo de Smith-Waterman [50] e também utiliza o conceito de programação dinâmica, sendo na verdade uma adaptação do algoritmo para comparação global.

Todos os conceitos utilizados no algoritmo de comparação global, como alinhamentos e similaridades, também são utilizados na comparação local de seqüências, assim como toda a discussão a respeito da matriz de similaridade e o significado de suas posições.

A única diferença entre os dois algoritmos é que na comparação local, a fórmula usada no preenchimento da matriz de similaridade não aceita valores negativos, sendo, portanto, o zero o menor valor existente na matriz. A fórmula 5.2 é a fórmula usada no preenchimento da matriz.

$$A(i,j) = max \begin{cases} A(i,j-1) + p \\ A(i-1,j-1) + h(i,j) \\ A(i-1,j) + p \\ 0 \end{cases} \tag{5.2}$$

Como decorrência dessa nova fórmula, a primeira linha e a primeira coluna da matriz não mais serão preenchidas com múltiplos do valor da penalidade associada ao alinhamento entre um caractere e um espaço vazio. Ao invés disso, a primeira linha e a primeira coluna serão preenchidas com o valor 0.

Aplicando essa fórmula para o problema da comparação local das duas seqüências utilizadas no exemplo anterior, obtém-se ao final uma matriz igual a que é mostrada na figura 5.6.

O algoritmo usado para preencher a matriz é o algoritmo 1 acrescido de uma condição que não permita que uma posição da matriz seja preenchida com um valor negativo.

Utilizando um procedimento semelhante ao que foi descrito para reconstruir o alinhamento global entre as seqüências, pode-se encontrar diversos alinhamentos entre subseqüências de $GACGG$ e $GATCGG$. Para encontrar um desses alinhamentos basta partir de uma posição que tenha um valor maior do que zero e caminhar até encontrar uma posição cujo valor seja igual a zero, através de um caminho composto por posições deduzidas a partir da regra de preenchimento da

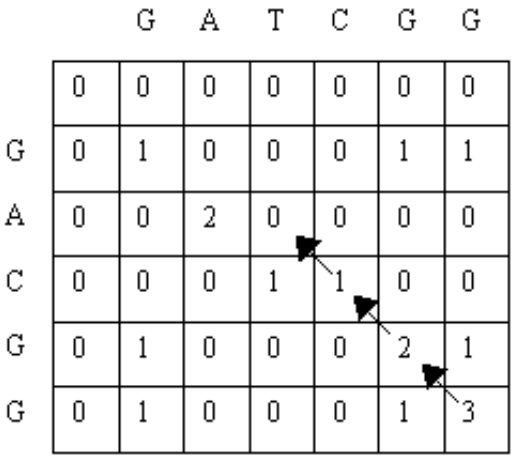


Figura 5.6: Matriz de similaridade entre $GACGG$ e $GATCGG$ (preenchida)

matriz. No exemplo, podemos constatar que existe um alinhamento de similaridade igual a 3, entre as subseqüências CGG de s e CGG de t, além de dois alinhamentos de similaridade igual 2 e nove alinhamentos de similaridade igual a 1.

5.4 Padrão *Wavefront* de acesso aos dados

Durante o procedimento de preenchimento da matriz de similaridade, tanto no caso da comparação local quanto no caso da comparação global, existem dependências entre os dados das células da matriz. Essas dependências são decorrentes das fórmulas usadas no preenchimento da matriz. Em outras palavras, para se garantir que uma resposta correta seja encontrada, é preciso respeitar o padrão de acesso às células da matriz decorrentes das dependências entre os dados.

De maneira mais concreta, as fórmulas de preenchimento da matriz dizem que uma célula $A[i, j]$ depende dos valores de três outras células: a que está imediatamente acima dela ($A[i-1, j]$), a que está imediatamente à esquerda dela ($A[i, j-1]$) e a que está na diagonal superior esquerda dela ($A[i-i, j-1]$). Assim, para preencher a posição $A[0, 0]$ não haverá nenhuma espécie de dependência, visto que não há nenhuma célula antes dela. Porém, se quisermos preencher a posição $A[1, 1]$, devemos obrigatoriamente ter preenchido as posições $A[0, 1]$ e $A[1, 0]$, além da posição $A[0, 0]$. Se, em seguida, quisermos calcular as posições $A[1, 2]$ e $A[2, 1]$, devemos ter calculado, além das células anteriores, as células $A[0, 2]$ e $A[2, 0]$. Esse mesmo raciocínio se aplica ao cálculo de todas as outras células da matriz. Dessa forma, pode-se dizer que o cálculo da matriz segue da esquerda para a direta como se fosse uma onda percorrendo as anti-diagonais da matriz [45], conforme exemplifica a figura 5.7. Na figura em questão, a seta indica a direção do processamento da matriz, ou seja, o processamento acontece através das anti-diagonais da matriz.

Esse padrão de acesso aos dados permite que o cálculo da matriz seja paralelizado. Entretanto, devido às dependências entre as posições da matriz, o grau

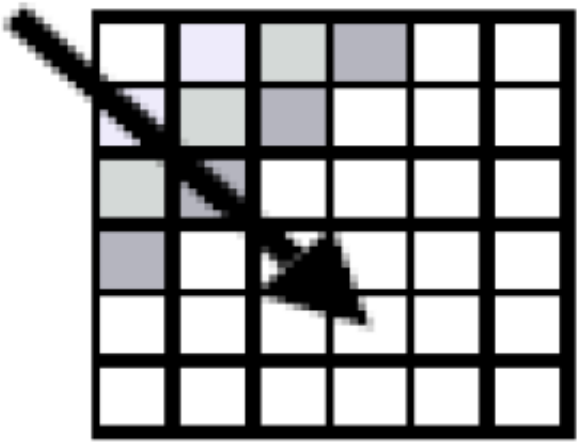


Figura 5.7: Exemplo do padrão wavefront

de paralelismo obtido não é uniforme ao longo do preenchimento da matriz. No início do processamento, ou seja, no preenchimento das primeiras posições da matriz, o grau de paralelismo é baixo. Entretanto, à medida em que mais posições são calculadas, torna-se possível calcular mais posições em paralelo, ou seja, o grau de paralelismo aumenta. Na anti-diagonal principal da matriz, o grau de paralelismo que pode ser obtido é máximo. Depois da anti-diagonal principal, o grau de paralelismo volta a diminuir.

5.5 Comparação de Seqüências em Sistemas Distribuídos

Sendo uma operação básica da biologia computacional, é interessante que o desempenho da comparação de seqüências seja o melhor possível. Tanto o algoritmo de Smith-Waterman para comparação local (seção 5.3) quanto o algoritmo de Needleman-Wunsch para comparação global (seção 5.2), apesar de ótimos, possuem complexidade em $O(m * n)$, onde m e n são os tamanhos das seqüências, devido à necessidade de se calcular a matriz de similaridade. Para duas seqüências de tamanhos iguais a n, a complexidade torna-se quadrática em n. Tal complexidade se traduz em um custo alto, tanto de processamento quanto de memória, especialmente quando as seqüências em questão são grandes.

Para diminuir os requisitos de tempo e espaço da comparação de seqüências, diversas abordagens já foram propostas. Por um lado, podemos ter abordagens heurísticas que apresentam custos de tempo e espaço lineares, ou próximos de lineares, mas que não garantem resultados ótimos. Existem várias alternativas de ferramentas que possuem boa aceitação e são úteis pois são capazes de gerar resultados em tempo bem menores do que os algoritmos ótimos baseados em programação dinâmica. Exemplos dessas ferramentas são o BLAST [4] e o FASTA [44].

Por outro lado pode-se tentar reduzir o tempo de execução dos algoritmos ótimos através de técnicas de programação paralela. A análise da aplicação e do

padrão de acesso às posições da matriz, que segue o padrão *wavefront* (seção 5.4), como discutido anteriormente, garantem que é possível implementar a comparação de seqüências de maneira paralela.

5.5.1 Comparação Global usando uma implementação paralela *multithreaded*

Em [41], é proposta uma implementação paralela com múltiplas *threads* de execução da comparação global de seqüências. Essa implementação é baseada no modelo EARTH de execução [33]. Esse modelo disponibiliza um ambiente de execução em que uma aplicação é vista como uma coleção de *threads* cuja ordem de execução é determinada pelas dependências de dados e controle identificadas na aplicação. Além disso, cada *thread* desse modelo é considerada como sendo composta por entidades chamadas de *fibers*. Uma *fiber* é uma linha de execução não preemptiva e que é escalonada de acordo com regras que garantem que ela só será executada quando todos os dados necessários à sua execução estiverem disponíveis.

Nessa proposta, as linhas da matriz de similaridade são divididas entre os processadores. Cada processador recebe um conjunto de linhas, que deverão ser processadas por ele. O processador possui uma *thread* que irá ser responsável por esse processamento. A thread de cada processador possui duas *fibers*. Uma delas é responsável por transmitir/receber informações de sincronização para/do processador que possui uma linha da matriz vizinha à sua própria linha, de modo a garantir que as dependências do padrão *wavefront* sejam respeitadas. A outra *fiber* da *thread* de um processador será responsável pelo cálculo das posições daquela linha. Essa *fiber*, também receberá informações de sincronização da outra *fiber* e, além disso, irá enviar/receber os valores calculados nas células da linha da matriz.

Dessa forma, a matriz de similaridade é processada linha à linha, da esquerda para a direita, com cada processador responsável por uma linha. Cada vez que um processador preenche uma posição da sua linha, ele envia o valor calculado para o processador responsável pela próxima linha, além de informações de controle e de sincronização. Uma vantagem desse esquema é que não é necessário alocar espaço para toda a matriz, que pode ser grande no caso de seqüências longas, pois cada processador irá trabalhar apenas com uma linha. Outra vantagem é que, com a utilização de duas *fibers* por *thread*, uma para sincronização e outra para processamento da matriz, é possível sobrepor a comunicação entre as *threads* e o processamento da matriz.

Um ponto a ser notado é que a implementação em questão procura utilizar todos os processadores disponíveis para realizar o processamento da matriz. Em outras palavras, é utilizada uma abordagem fixa de mapeamento, onde cada processador recebe um conjunto de linhas, e que procura atribuir tarefas a todos os processadores.

5.5.2 Comparação local utilizando DSM (*Distributed Shared Memory*)

Em [8][42], uma variação heurística do algoritmo de comparação local é implementado baseando-se no modelo de *Distributed Shared Memory* (DSM) (seção 2.3.1). Esse modelo visa tornar a programação em ambientes distribuídos mais próxima da programação em sistemas monprocessados, através de operações de leitura e escrita em memória. Em poucas palavras, um sistema DSM funciona como um sistema de memória virtual, onde as páginas de memória podem ser encontradas localmente ou em outras máquinas na rede. Apesar de buscar fazer com que o modelo de programação seja mais intuitivo, por ser mais próximo do modelo de programação de sistemas monoprocessados, existe um *overhead* de se manter as páginas de memória coerentes entre si, uma vez que elas podem estar duplicadas em diferentes máquinas. Diversos sistemas buscam implementar modelos de consistência de memória relaxados, a fim de diminuir esse *overhead*.

Um desses sistemas DSM é o JIAJIA [31], que implementa um modelo de consistência de memória de escopo [35]. A implementação paralela da comparação local em questão utiliza o JIAJIA para fazer o processamento da matriz. A implementação proposta possui duas etapas. Na primeira delas, a matriz de similaridade é calculada, de modo a obter os escores dos alinhamentos. Na segunda etapa, os alinhamentos selecionados são reconstruídos.

Na etapa de processamento da matriz de similaridade, a paralelização dos cálculos obedece ao padrão *wavefront* (seção 5.4). Entretanto, ao contrário de [41], o trabalho é atribuído aos processadores não com base nas linhas da matriz, mas sim com base nas colunas. Cada processador do sistema é responsável por calcular um número de colunas contíguas da matriz. A comunicação entre os processadores ocorre nas posições que representam as fronteiras entre as partes da matriz alocadas a cada processador. Outra diferença entre as implementações é que, uma vez que está sendo utilizado um sistema DSM, para se garantir que o padrão *wavefront* seja obedecido, a implementação sugerida utiliza operações de *lock/unlock* e de barreiras para fazer a sincronização entre os processadores.

Ao final da primeira etapa do processamento, haverá uma lista contendo as posições iniciais e finais de cada alinhamento selecionado. A segunda etapa do processamento, que visa recuperar esses alinhamentos, baseia-se em um modelo mestre-escravo (seção 3.4.1) de processamento. Cada um dos processadores disponíveis recupera suas tarefas a partir da lista e reconstrói o alinhamento correspondente. A fim de minimizar as operações de sincronização nesse etapa, utiliza-se um mapeamento do tipo *scather-gather* [25].

Cabe observar que, na primeira etapa do processamento, a política de alocação de tarefas também é fixa, onde cada processador do sistema recebe um número igual de colunas a serem processadas. Esse número é igual à $\frac{n}{p}$, onde n é o tamanho da seqüência e p é o número total de processadores. Todos os processadores recebem tarefas.

5.5.3 Comparação local de seqüências em *grid* (*culster* de *clusters*)

Em [12][13] é descrita uma implementação paralela da comparação local de seqüencias em um ambiente de *grid* computacional (*cluster* de *clusters*). Essa implementação baseia-se no modelo de troca de mensagens e utiliza o MPICH [2] e o MPICH-G2 [26] para realizar a comunicação entre os processadores envolvidos. O MPICH é uma implementação da API MPI (seção 2.3.2), de troca de mensagens. Já o MPICH-G2 é uma implementação dessa API voltada para o ambiente de *grid* e desenvolvida no âmbito do projeto globus [27], um *toolkit* para construção de *grids* computacionais. Um detalhe a ser considerado é que esse trabalho apresenta uma implementação da comparação local que visa obter alinhamentos quase ótimos, mas não ótimos.

O ambiente de execução para a implementação proposta é formado por vários *clusters* interligados entre si. A comunicação entre as máquinas de um *cluster* é eficiente e atinge altas taxas de transmissão de dados. Entretanto, a comunicação entre os *clusters* apresenta taxas de transmissão consideravelmente piores. Assim sendo, considera importante utilizar essas informações para determinar como o processamento será feito. A comunicação entre os *clusters* é feita por meio do MPICH-G2, enquanto que a comunicação mais eficiente dentro de um *cluster* é feita através do MPICH.

A implementação apresentada possui duas etapas de processamento. Na primeira, calcula-se a matriz de similaridade, obtendo o escore e as posições de alinhamentos quase ótimos. A distribuição do trabalho é feita com base nas colunas da matriz. Na segunda etapa, esses alinhamentos são reconstruídos. São investigadas diferentes estratégias de alocação de tarefa para a execução da primeira etapa do algoritmo proposto.

Uma dessas estratégias é a alocação estática das colunas aos processadores. Primeiramente, as colunas da matriz são divididas entre os *clusters* do sistema, de acordo com o poder de processamento de cada um deles. Em seguida, cada *cluster* divide as colunas alocadas para ele entre os seus processadores. Um fato importante de ser observado é que, apesar da alocação ser estática, o número de colunas atribuídas a cada processador não é igual, sendo que existirão processadores responsáveis por mais colunas. Essa estratégia é simples de ser implementada, porém é suscetível a uma degradação de desempenho da aplicação caso existam outras aplicações sendo executadas.

Uma estratégia alternativa proposta é a alocação baseada no paradigma mestre-escravo (3.4.1). Nessa estratégia, a matriz de similaridade é divida em blocos retangulares e os processadores solicitam blocos da matriz para um processo mestre. Quando terminam o processamento, enviam para o mestre a fronteira dos blocos processados. Essa abordagem tem a vantagem de permitir a alocação de mais tarefas para os processadores que estiverem apresentando um melhor desempenho. Porém, o custo de comunicação é elevado, visto que trata-se de um ambiente *multi-cluster*.

Por fim, é proposta uma nova estratégia baseada no modelo mestre-escravo. Nessa estratégia, a matriz é inicialmente dividida de maneira semelhante à política estática. Entretanto, os processadores periodicamente informam estatísticas

de desempenho para um processo mestre, que monta uma nova distribuição de tarefas com base nessas informações. As informações dessa nova distribuição de tarefas são transmitidas a todos os processadores. Entretanto, a redistribuição das tarefas propriamente ditas é feita pelos próprios processadores, que trocam dados (no caso, pedaços da matriz sendo calculada) com seus processadores vizinhos. Dessa forma, busca-se um meio termo entre a capacidade de alocar mais trabalho aos processadores de maior desempenho e o custo de comunicação envolvido no processo.

Na segunda etapa do processamento, depois do cálculo da matriz de similaridade e da identificação dos alinhamentos, todos os processadores participam da reconstrução desses alinhamentos. é importante notar que, tanto na primeira etapa quanto na segunda etapa, busca-se utilizar todos os processadores disponíveis.

5.5.4 Comparação Paralela Exata em *clusters* de computadores

Uma outra variação do algoritmo de comparação local é mostrada em [46] e implementada utilizando-se o MPICH [2], implementação da biblioteca MPI (seção 2.3.2). A variação do algoritmo apresenta duas fases de processamento, uma de cálculo da matriz de similaridade e outra de reconstrução dos alinhamentos encontrados. Na primeira fase, ocorre o preenchimento da matriz de similaridade, de modo a encontrar os melhores escores de alinhamentos. Nessa implementação, assim como em todas as outras discutidas até o momento, o algoritmo leva em consideração o padrão de acesso às posições da matriz, onde a posição $A(i, j)$ depende das posições $A(i-1, j)$, $A(i, j-1)$ e $A(i-1, j-1)$. Dessa forma, não é necessário trabalhar com a matriz inteira em memória, bastando apenas duas linhas e lembrando-se de guardar as posições dos melhores escores encontrados a cada momento.

A implementação paralela da primeira fase de processamento (preenchimento da matriz de similaridade) é feita levando-se novamente em consideração o padrão de acesso às posições da matriz (Método *wavefront*, seção 5.4) e baseia-se na distribuição de colunas da matriz aos processadores disponíveis. As figuras 5.8 à 5.12 ilustram o preenchimento da matriz em paralelo. Cada processador envolvido recebe um grupo contíguo de colunas da matriz, que ele terá que processar. A título de ilustração, a matriz é mostrada por inteiro, mas pode-se trabalhar apenas com duas linhas de cada vez, conforme discutido anteriormente. Na figura 5.8, tem-se que, ao início do processamento, apenas o processador p_1 pode trabalhar, de modo a respeitar o padrão de acesso às posições da matriz.

Ao término do preenchimento da primeira linha por parte de p_1, este processador envia para o processador p_2 os dados necessários para que ele possa iniciar o processamento (figura 5.9). Nesse momento, conforme mostra a figura 5.10, tanto p_1 quanto p_2 podem trabalhar paralelamente, um deles (p_1) preenchendo o início da segunda linha da matriz e o outro (p_2) preenchendo o final da primeira linha. Quando p_1 termina a sua parte, ele novamente envia para p_2 dados necessários para que este processador continue seu processamento (figura 5.11).

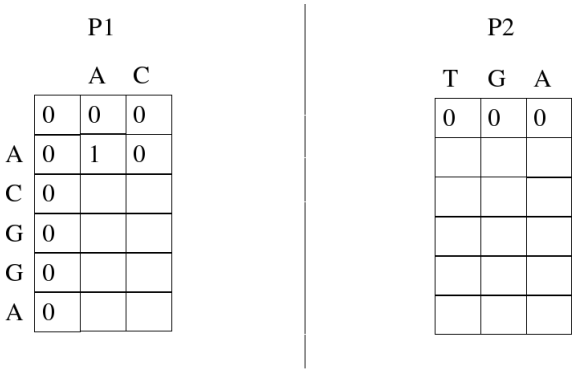


Figura 5.8: Início da comparação paralela: apenas o processador p_1 pode processar. O processador p_2 deve aguardar.

Esse procedimento é repetido até que a matriz seja totalmente processada (figura 5.12).

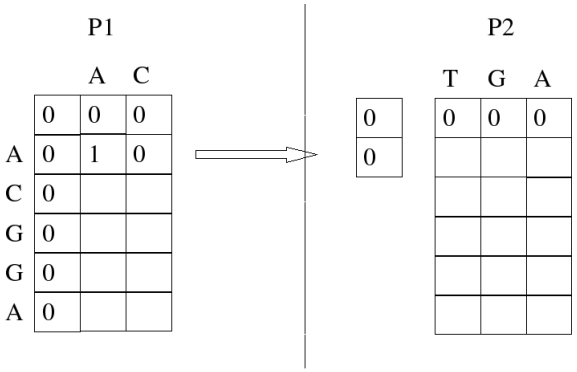


Figura 5.9: p_1 envia dados para que p_2 possa iniciar seu processamento.

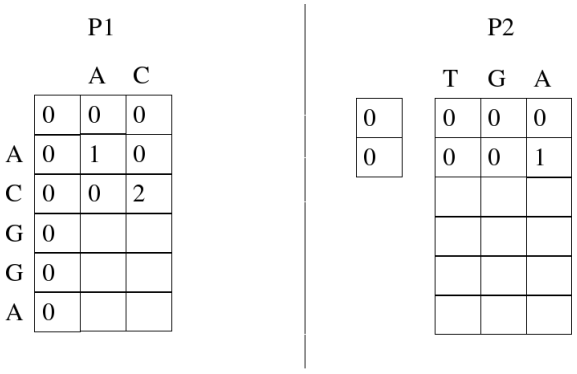


Figura 5.10: p_1 e p_2 podem trabalhar em paralelo.

A distribuição das colunas para os processadores disponíveis é feita em [46] dividindo-se o tamanho de uma linha da matriz (seqüência s), que supomos ser n, pelo número total de processadores, que supomos ser k. Assim, cada processador terá uma subseqüência de s de tamanho aproximadamente igual a $\frac{n}{k}$. Em outras

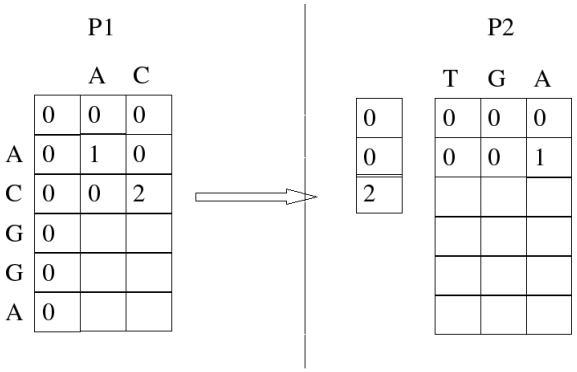


Figura 5.11: p_1 envia dados para p_2 continuar seu processamento.

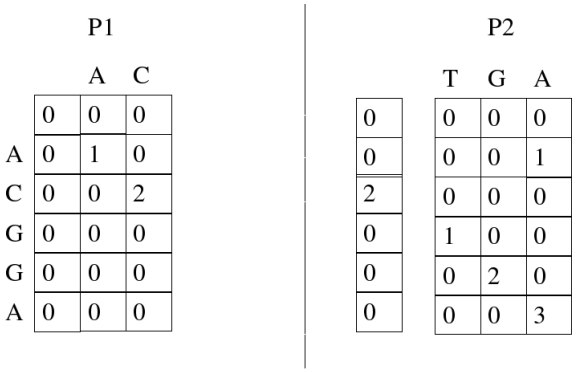


Figura 5.12: Fim do processamento paralelo

palavras, a política de alocação usada é fixa e busca fazer com que todos os processadores disponíveis recebam algum trabalho.

5.5.5 Comparação entre os algoritmos de comparação paralela de seqüências

A tabela 5.1 compara as características de cada uma das propostas apresentadas nas seções anteriores.

Pode-se notar que existem abordagens em diferentes modelos de programação, como troca de mensagens e DSM. Apenas uma das implementações, a descrita em [13], se preocupa explicitamente em abordar a questão dos sistemas heterogêneos. Coincidentemente, é a única que apresenta uma política de alocação dinâmica e que não seja fixa. Entretanto, todas utilizam o número total de processadores disponíveis, ou seja, procuram utilizar todos os processadores do sistema.

Referência	**Tipo de Comparação**	**Modelo de Programação**	**Sistemas Heterogêneos**	**Política de Alocação**	**Procs. usados**
[41] seção 5.5.1	Global	Troca de Mensagens (EARTH)	Não	Estática/fixa	Todos
[8][42] seção 5.5.2	Local	DSM (JIAJIA)	Não	Estática/fixa	Todos
[12][13] seção 5.5.3	Local (quase ótimo)	Troca de Mensagens (MPICH /MPICHG2)	Sim	Estática/Dinâmica (Mestre-escravo)	Todos
[46] seção 5.5.4	Local	Troca de Mensagens (MPICH)	Não	Estática/fixa	Todos

Tabela 5.1: Quadro comparativo entre alguns as implementações propostas

Capítulo 6

Projeto de um *Framework* de Alocação de Tarefas para Ambientes Heterogêneos

O objetivo do presente capítulo é apresentar o projeto de um *framework* para a alocação de tarefas em ambientes distribuídos heterogêneos para aplicações que se encaixem no padrão *wavefront* (seção 5.4). Mais especificamente, de maneira similar a [18], o *framework* proposto tem como aplicações-alvo as aplicações de comparação de seqüências biológicas que utilizam programação dinâmica (capítulo 5).

A decisão de se projetar um *framework* foi tomada porque, depois de analisarmos diversos alocadores de tarefas (capítulos 3 e 4), verificamos que não existe política de alocação que seja melhor adequada a um grande conjunto de aplicações e plataformas de execução. Sendo assim, é interessante que diversas políticas sejam oferecidas ao programador para que o mesmo possa escolher a que lhe seja mais adequada.

Em segundo lugar, optamos, como em [39], [10] e [18], projetar um alocador que leva em consideração as características da aplicação. Essa decisão foi tomada porque possibilita que mais informações sejam levadas em conta, resultando em um alocador de tarefas mais realístico. Como o foco desse trabalho é em ambientes heterogêneos, que por sua natureza são bastante complexos, consideramos que um alocador específico da aplicação (*application-specific*) seria mais apropriado.

Em terceiro lugar, observamos que a maioria das estratégias de alocação projetadas para aplicações de comparação de seqüências biológicas utiliza o número total de processadores disponíveis para a alocação. No entanto, observamos que, quando o tamanho do problema a ser resolvido é pequeno ou quando o número de processadores disponíveis é muito grande , a utilização de todos os processadores pode causar um aumento do tempo de execução da aplicação paralela, quando comparado com o tempo de execução com menos processadores. Sendo assim, o objetivo do alocador de tarefas proposto na dissertação é determinar quantos e quais processadores devem ser utilizados na execução da aplicação.

6.1 Arquitetura do *framework* de alocação de tarefas

O alocador de tarefas proposto apresenta uma arquitetura modular, onde observa-se a existência de módulos específicos para tratar determinadas ações. A figura 6.1 apresenta os módulos do sistema. São eles:

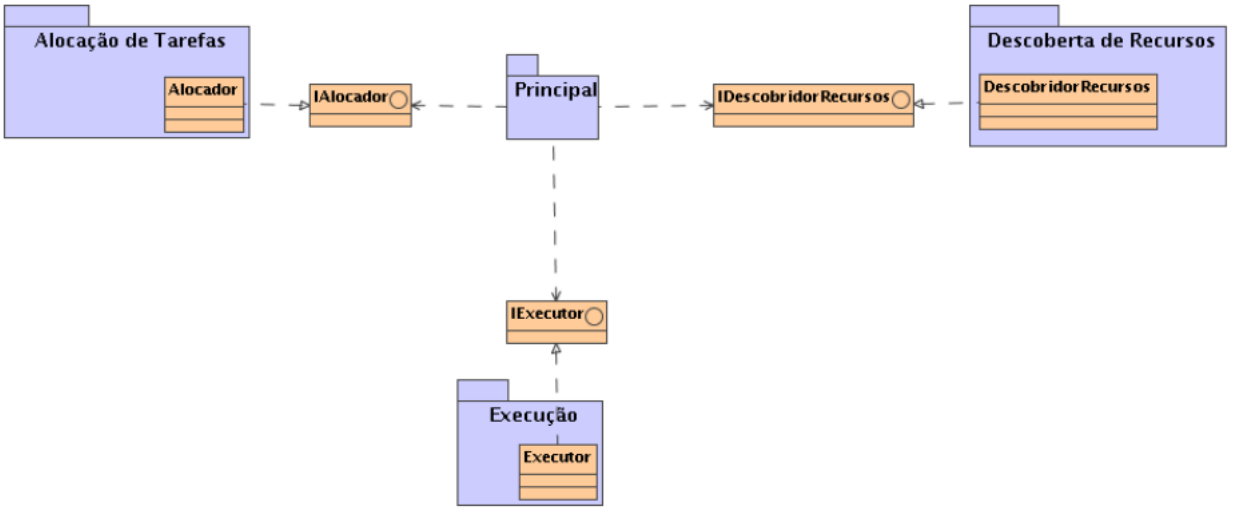


Figura 6.1: Arquitetura proposta

- Módulo de Descoberta de Recursos. Esse módulo é responsável por recuperar informações sobre o sistema que auxiliem na tomada de decisões de alocação. Exemplos de tais informações incluem: quantidade de elementos de processamento disponíveis, poder computacional de cada elemento de processamento e custos de comunicação entre os elementos de processamento;

- Módulo de Alocação de Tarefas. Esse módulo é responsável por implementar estratégias de alocação que podem, ou não, utilizar as informações obtidas pelo módulo de descoberta de recursos. Esse módulo fornece a informação de quantas tarefas deverão ser criadas e em quais elementos de processamento elas serão executadas.

- Módulo de Execução. Esse módulo é responsável por implementar a execução da aplicação. é com base nesse módulo que as tarefas irão realizar o seu processamento.

- Módulo Principal. Esse módulo é responsável por controlar todo o processo, realizando chamadas aos outros módulos quando necessário e criando as tarefas a serem executadas com base no que o módulo de mapeamento/alocação determinar.

Essa arquitetura modular caracteriza um *framework* para a alocação de tarefas. A criação de um módulo específico para a descoberta de recursos, por exemplo, isola essa funcionalidade dos outros módulos e possibilita a implementação de diferentes alternativas para essa atividade, indo desde a consulta a arquivos texto até a consulta a sistemas especializados na descoberta de recursos. Da mesma maneira, a existência de um módulo específico para o mapeamento/alocação de tarefas permite a implementação de diferentes estratégias e algoritmos de alocação, que podem ser comparados quanto ao desempenho e a utilização dos recursos disponíveis, por exemplo. Por fim, a definição de um módulo de execução possibilita a instanciação do *framework* para diferentes aplicações, sendo que o processamento específico da aplicação é encapsulado nesse módulo.

A partir dessa estrutura definida, serão criados dois componentes. Um deles, chamado de Componente Mestre, representa a tarefa inicial a ser executada no framework. Esse componente será o responsável pelo início de toda a execução, determinando onde serão criadas as outras tarefas e iniciando o processo de criação das mesmas. O outro componente, chamado de Componente Worker, representa as tarefas que foram criadas pelo Componente Mestre. Cada uma dessas tarefas é responsável pelas operações da aplicação a ser executada.

Uma vez que o Componente Mestre é o responsável por guiar a criação das outras tarefas, ele é construído a partir dos elementos definidos em cada um dos módulos da arquitetura, conforme mostra a figura 6.2. Dessa forma, ele é capaz de fazer a descoberta de recursos, criar o mapeamento entre tarefas e processadores, criar essas tarefas nos processadores adequados e ainda participar da computação a ser realizada.

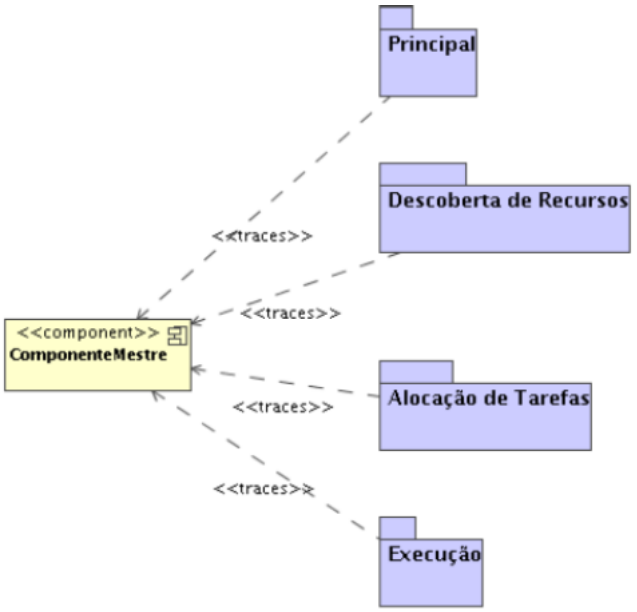


Figura 6.2: Componente Mestre

Já o Componente Worker, por representar as tarefas que serão criadas, é criado a partir dos elementos definidos no módulo Principal e no módulo de Execução, conforma mostra a figura 6.3, pois ele não precisa fazer descoberta de recursos

e nem criar mapeamentos entre tarefas e processadores. O componente precisa apenas participar das computações a serem realizadas.

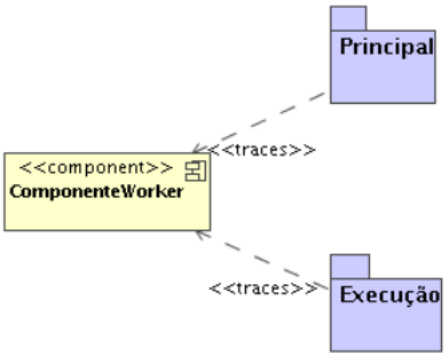


Figura 6.3: Componente Worker

A figura 6.4 mostra um exemplo de um ambiente de execução desses componentes. Entre as máquinas disponíveis à aplicação, uma delas será chamada de Máquina Mestre. Essa máquina contém um Componente Mestre e um Componente Worker. As outras máquinas serão chamadas de Máquinas Worker e possuem apenas instâncias do Compoente Worker. O processamento é iniciado a partir da Máquina Mestre, que executa o Componente Mestre. A execução desse componente obedece o esquema apresentado na figura 6.10, que será discutida mais adiante. No momento em que esse componente criar as outras tarefas da aplicação, a Máquina Mestre irá se comunicar com as Máquinas Worker para que as instâncias do Componente Worker sejam executas. A execução desses componentes obedece o esquema da figura 6.11.

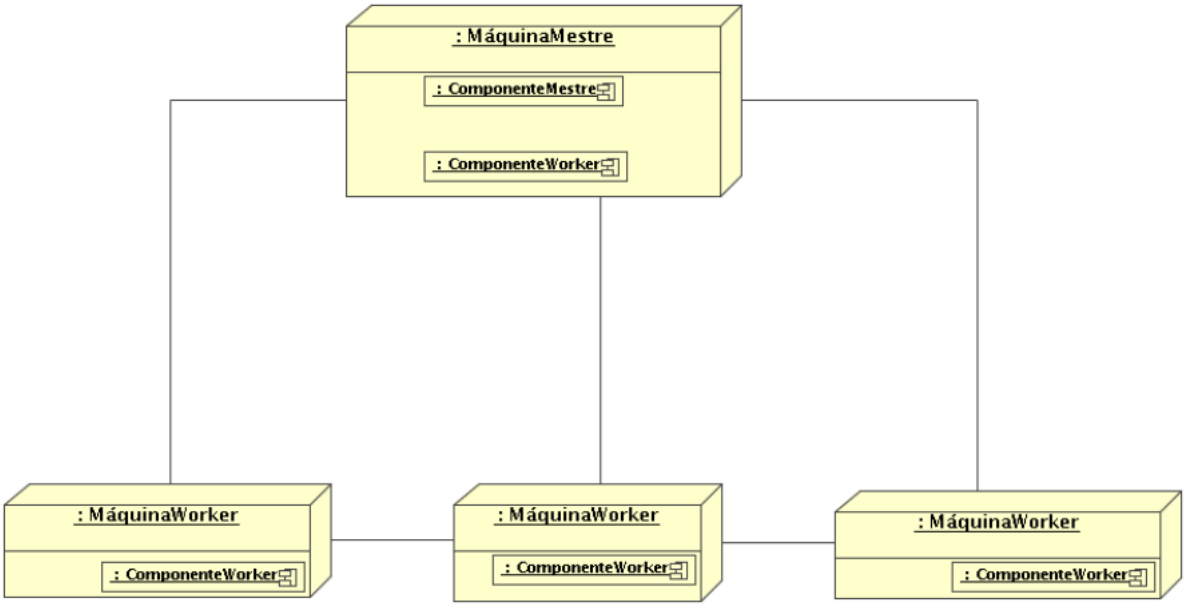


Figura 6.4: Possível cenário de execução dos componentes do *framework*

6.2 Módulo de Descoberta de Recursos

Esse módulo tem como objetivo encapsular os mecanismos existentes para descoberta de recursos. As informações que um mecanismo de descoberta de recursos pode recuperar podem variar bastante, indo desde a quantidade de elementos de processamento disponíveis até os custos de comunicação entre os elementos de processamento. Da mesma forma, a maneira de se recuperar tais informações também pode variar, podendo ser feita, por exemplo, através da leitura de um arquivo texto, da consulta a um serviço de diretório, da consulta direta aos elementos de processamento do sistema, entre outras alternativas.

No contexto do *framework* proposto, as informações recuperadas por este módulo são as seguintes:

- Quantidade de processadores existente no sistema
- Poder de processamento dos processadores. Uma vez que, no contexto desse trabalho, estamos lidando com aplicações de programação dinâmica, essa informação é expressa através de uma estimativa do custo de um processador preencher uma posição da matriz de similaridade (seção 5.2) durante o processamento de uma aplicação. Essa estimativa pode ser feita através da execução de uma versão seqüencial da aplicação. Executa-se a aplicação e divide-se o tempo gasto no preenchimento da matriz pelo tamanho da matriz, de modo a se obter a estimativa em questão. Quanto menor o tempo gasto, maior o poder de processamento.
- Custo de comunicação entre os processadores. Indica o custo de um processador enviar uma mensagem para outro processador através da rede de comunicação. Esse custo pode ser obtido através da execução de um *ping* entre duas máquinas. O tempo total do *ping*, que inclui o percurso de ida e volta de uma mensagem entre duas máquinas, é dividido por dois para se obter a estimativa do tempo de ida da mensagem. Quanto maior o tempo, maior o custo de comunicação.

Uma vez que o foco do presente trabalho é a alocação de tarefas, e não a descoberta de recursos, a implementação desse módulo foi simplificada, de modo que as informações necessárias são obtidas através da leitura de um arquivo texto previamente criado. Esse arquivo contém uma lista dos nomes das máquinas disponíveis (indica quantas e quais são as máquinas), uma lista de valores que indicam a estimativa do tempo gasto para se preencher uma posição de uma matriz em cada um dos processadores (indica o poder de processamento) e uma lista de valores que formam uma matriz com os tempo gastos na transmissão de uma mensagem entre cada um dos processadores (indica o custo de comunicação).

Apesar dessa simplificação, é preciso que o módulo de descoberta de recursos possua uma interface bem definida, de modo a permitir a alteração do mecanismo usado na recuperação das informações, bem como a alteração nas próprias informações recuperadas, causando o menor impacto possível na aplicação. Para permitir a variação das informações recuperadas, é definido um tipo de dado estruturado que guarda todas essas informações, como mostra a figura 6.5.

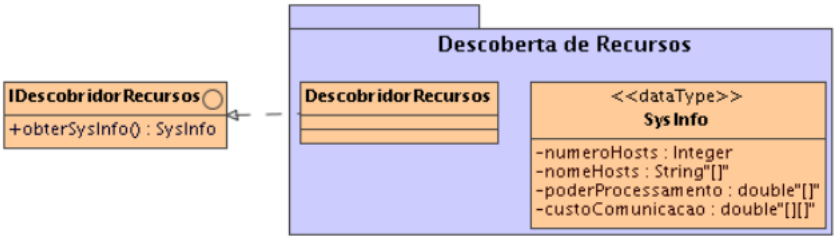


Figura 6.5: Módulo de Descoberta de Recursos

A interface do módulo de descoberta de recursos possui então uma função *obterSysInfo* que retorna o tipo de dado estruturado com as informações recuperadas (figura 6.5). Se for necessário mudar o mecanismo de descoberta de recursos, acessando-se um serviço de diretório, por exemplo, as alterações ficam confinadas ao módulo de descoberta de recursos, pois a interface permanece a mesma. De modo semelhante, caso sejam acrescentadas novas informações a serem recuperadas, basta acrescentá-las no tipo de dado estruturado. A interface continua a mesma, porém, nesse caso, é necessário implementar o tratamento dessas informações nos lugares que farão uso delas, como por exemplo, no módulo de alocação de tarefas.

6.3 Módulo de Alocação de Tarefas

O propósito do módulo de alocação de tarefas é isolar a política de alocação sendo usada, permitindo que esta varie sem causar impacto no código do restante da aplicação. Para permitir esse nível de isolamento, é preciso que o módulo de alocação de tarefas possua uma interface bem definida. A figura 6.6 mostra essa interface.

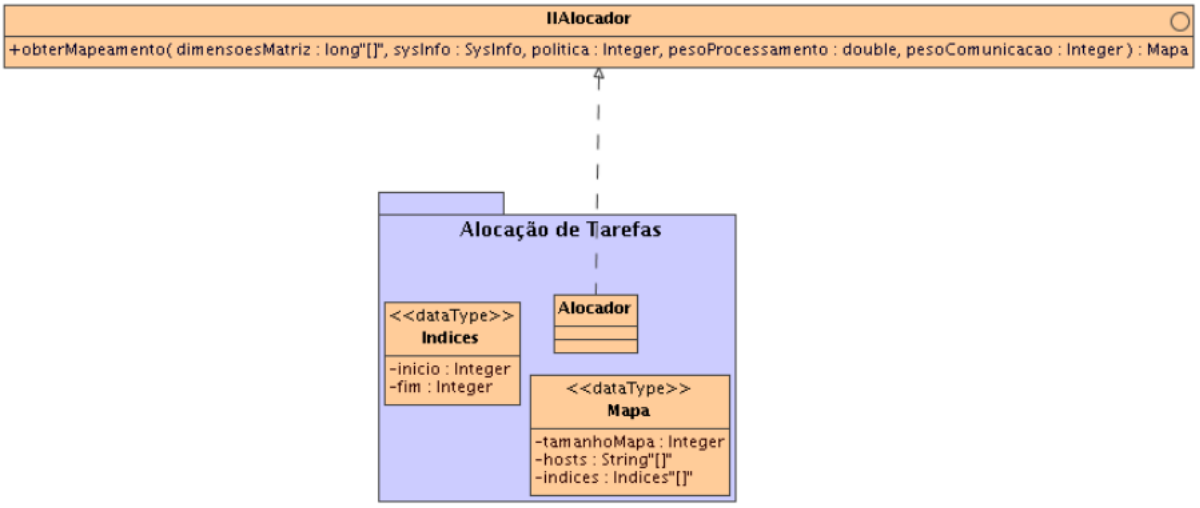


Figura 6.6: Módulo de Alocação de Tarefas

A função *obterMapeamento*, da interface do módulo Alocador, recebe os se-

guintes parâmetros:

- Tamanho da matriz a ser processada
- Informações sobre o sistema
- Um *flag* indicando qual a política de alocação a ser usada
- Valores representando o peso a ser dado para o poder de processamento e para o custo de comunicação no processo de alocação

O retorno dessa função é um tipo de dado estruturado representando um mapeamento entre tarefas e processadores (figura 6.6). Na estrutura retornada estão presentes o tamanho do mapeamento (indicando quantas máquinas serão usadas), uma lista de nomes de máquinas (indicando quais máquinas serão usadas) e uma lista de índices da matriz a ser processada, indicando quais colunas serão processadas em cada uma das máquinas.

O funcionamento geral do módulo de alocação de tarefas é mostrado na figura 6.7. Ao ser ativado, o módulo primeiramente verifica o *flag* de indicação de política de alocação, de modo a determinar qual será a política utilizada. Essa política, por sua vez, pode utilizar as informações do sistema, bem como os pesos atribuídos ao poder processamento e ao custo de comunicação, para ordenar as máquinas do sistema. Por fim, a política obtém o mapeamento, utilizando mecanismos próprios e baseando-se nas informações do sistema e da aplicação (tamanho da matriz).

---

obterMapeamento(tamanhoMatriz, SysInfo, políticaDeAlocação, pesoProcessamento, pesoComunicação)
    Determinar qual a política de alocação a ser usada
        Caso seja a 1: executarPolitica1(dimensoesMatriz, SysInfo, pesoProcessamento, PesoComunicação)
        Caso seja a 2: executarPolitica2(dimensoesMatriz, SysInfo, pesoProcessamento, PesoComunicação)
        Caso seja a 3: executarPolitica3(dimensoesMatriz, SysInfo, pesoProcessamento, PesoComunicação)
        ...
    retornar o mapeamento obtido por uma das políticas.

Figura 6.7: Estrutura da função *obterMapeamento*

6.4 Módulo Executor

O módulo executor tem como objetivo isolar os detalhes da aplicação propriamente dita das outras partes do *framework*. Em outras palavras, o módulo fornece um interface simples para as tarefas, que visa esconder dos outros módulos as operações da aplicação. Essa interface serve de "ponto de entrada"para as

tarefas executarem a aplicação, no sentido de que a função fornecida pela interface irá realizar as chamadas para as operações da aplicação.

Para atingir o seu objetivo, a interface do módulo executor deve fornecer uma função que receba como parâmetro as informações que irão permitir a execução da aplicação. No contexto do *framework* apresentado, essas informações são aquelas que permitirão a construção da matriz a ser processada. No caso de uma aplicação de comparação de seqüências, que utilize a MPI, por exemplo, a interface é igual a que está sendo mostrada na figura 6.8. Essa interface define uma função que recebe as seqüências a serem comparadas e o tamanho dessas seqüências, informações necessárias para a construção da matriz, além de argumentos passados pela linha de comando e um comunicador MPI que será utilizado para a comunicação das tarefas. Dentro dessa função, são feitas chamadas à outras funções, que implementam a lógica necessária à execução da aplicação, conforme exemplifica a figura 6.9.

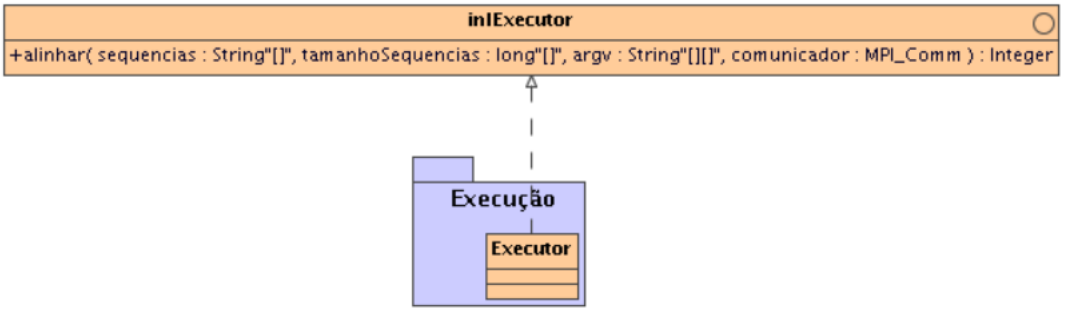


Figura 6.8: Módulo de Execução

```
alinhar()
    Se rank da tarefa for igual 0
        Distribui trabalho
        Espera resultados
    Senão
        Recebe trabalho
        chama funcao1()
        chama funcao2()
        chama funcaoN()
        Envia resultados
```

Figura 6.9: Estrutura da função *alinhar*, do módulo de execução

Dentro do módulo de execução, o Componente Mestre (seção 6.1) pode, por exemplo, distribuir trabalho para as tarefas criadas e esperar. Os Componentes Worker (seção 6.1), por sua vez, recebem o trabalho enviado do Componente Mestre, realizam o processamento necessário e devolvem os resultados obtidos.

6.5 Módulo Principal

O módulo principal representa o ponto de entrada do *framework* proposto, ou seja, é por esse módulo que se inicia uma aplicação que instancie o *framework*, utilizando sua estrutura. Dentro desse módulo, são feitas chamadas para todos os outros módulos, a fim de se obter as informações do sistema, criar o mapeamento entre tarefas e processadores e executar a aplicação. Além disso, é nesse módulo que são efetivamente criadas as tarefas que irão executar a aplicação, com base no mapeamento gerado pelo módulo de alocação de tarefas.

6.6 Funcionamento Básico do *Framework*

A figura 6.10 é um diagrama de seqüência que ilustra o funcionamento básico do framework proposto. Uma aplicação começa a ser executada com uma única tarefa, que inicia sua execução pelo módulo principal. Depois de eventuais procedimentos de inicialização que precisem ser executados, é feita uma chamada ao módulo de descoberta de recursos. Esse módulo recupera as informações do sistema e as devolve para o módulo principal encapsuladas em um tipo de dado estruturado. Em seguida, o módulo principal aciona o módulo de alocação de tarefas, passando como parâmetro as informações sobre o sistema. O módulo de alocação de tarefas retorna um tipo de dado estruturado que representa o mapeamento sugerido entre tarefas a serem criadas e processadores do sistema. Com base nesse mapeamento, o módulo principal cria dinamicamente as tarefas da aplicação nos respectivos processadores. Em seguida, faz uma chamada ao módulo de execução, para que possa tomar parte na execução da aplicação.

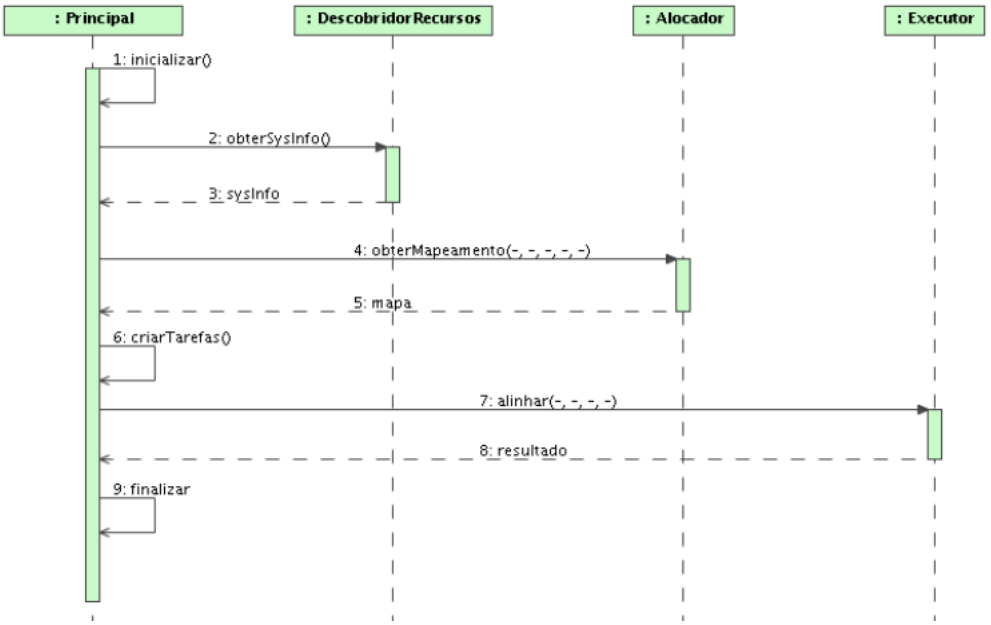


Figura 6.10: Funcionamento básico do *framework* proposto

Cada uma das tarefas criadas pelo módulo principal, também irá começar

sua execução pelo módulo principal. A diferença é que essas tarefas não irão fazer chamadas aos módulos de descoberta de recursos e de alocação de tarefas. Ao invés disso, as tarefas criadas, depois de executarem seus procedimentos de inicialização, irão realizar uma chamada ao módulo de execução, para que possam tomar parte na execução da aplicação. Esse comportamento está ilustrado no diagrama de seqüência da figura 6.11.

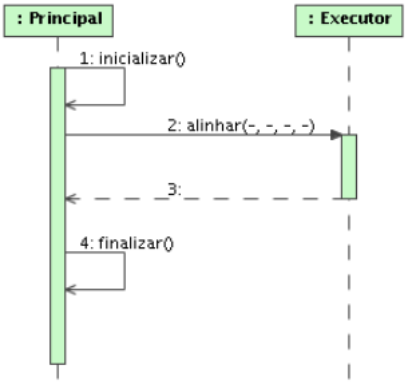


Figura 6.11: Funcionamento básico das tarefas criadas pelo *framework* proposto

6.7 Políticas de Alocação de Tarefas

Como dito anteriormente, o módulo de alocação de tarefas do framework proposto tem como objetivo criar um mapeamento entre as tarefas da aplicação e os processadores disponíveis no sistema. Quando se pensa em uma implementação paralela de uma aplicação baseada em programação dinâmica, a maneira mais direta e intuitiva de se alocar tarefas é possivelmente através da distribuição uniforme do trabalho entre todos os processadores disponíveis, estratégia que chamamos de política fixa de alocação de tarefas [48]. Dessa forma, cada processador torna-se responsável pelo processamento de um número igual de colunas da matriz e todos eles teriam trabalho a ser feito.

6.7.1 Política Fixa

Durante a discussão sobre políticas de alocação, usaremos como exemplo uma aplicação de comparação local de seqüências de DNA, que, conforme visto no capítulo 5, é resolvido de maneira ótima através de um algoritmos de programação dinâmica e que apresenta dependências de dados de acordo com o padrão *wavefront*. Quando se compara duas seqüências de tamanho m e n, têm-se uma matriz mXn. Utilizando a política fixa de alocação de tarefas, essa matriz será dividida igualmente entre os processadores do sistema, de modo que cada processador possuirá uma matriz mX$(\frac{n}{k})$, onde k é o número total de processadores. Apesar dessa abordagem ser a mais direta e intuitiva, alguns problemas devem ser considerados:

- Não são levadas em consideração as informações sobre os processadores disponíveis. Se os processadores fizerem parte de um sistema heterogêneo, as diferentes características de cada processador, como poder de processamento e custo de comunicação, por exemplo, podem ser levadas em conta para ajudar a alocação de tarefas. De acordo com as características do sistema, o desempenho da aplicação pode ser melhor se determinadas máquinas forem escolhidas para receber as tarefas, deixando outras sem nenhuma tarefa a ser executada.

- Não está sendo levado em conta o *overhead* causado pela necessidade de comunicação e sincronização entre as tarefas. A necessidade de comunicação e sincronização entre as tarefas, decorrente do padrão *wave-front*, traz consigo alguns custos e limita a quantidade de processamento que pode ser feita em paralelo. Se considerarmos uma situação em que exista uma grande quantidade de processadores disponíveis e as seqüências sendo comparadas são relativamente pequenas, a distribuição uniforme de trabalho entre os processadores pode não ser a melhor opção de alocação, devido aos custos de comunicação, sendo que a utilização de menos processadores pode tornar o processamento menos demorado.

6.7.2 Políticas que determinam o número de processadores

Assim sendo, propomos a criação de políticas de alocação que busquem determinar o melhor conjunto de máquinas a serem utilizadas, dadas as informações sobre o sistema e as estimativas de custos de processamento e de comunicação entre as tarefas e considerando que o tempo total de processamento da aplicação será dado pela soma entre o tempo de processamento gasto pelos processadores e o tempo gasto com a comunicação entre esses processadores. A estrutura das políticas de alocação proposta é ilustrada na figura 6.12.

---

politicaDeAlocacao(InfoSobreSistema, InfoSobreAplicação)

Ordenar as máquinas do sistema com base em algum critério (poder de processamento, custo de comunicação, etc...)

Calcular uma estimativa do tempo de execução usando apenas uma máquina (a melhor, considerando o ordenamento obtido)

para $i \leftarrow 2$ até o número de máquinas disponíveis faça:

Calcular uma estimativa do tempo de execução usando um número i de máquinas (considerando o ordenamento obtido), com base em uma fórmula pré-determinada.

Se a estimativa obtida for pior do que a estimativa para $i - 1$ máquinas, então:

Considerar a alocação de tarefas para $i - 1$ máquinas e sair do *loop*

retornar o mapeamento obtido entre tarefas e processadores.

Figura 6.12: Estrutura geral das políticas de alocação que levam em conta as informações do sistema e os custos de comunicação entre tarefas

Dentro dessa estrutura, a cada vez que consideramos mais um processador,

calculamos a estimativa do processamento considerando esse novo processador e assumindo que a matriz a ser processada é dividida igualmente entre os processadores. Essa estrutura admite variações em dois pontos: no ordenamento (ou não) das máquinas de acordo com algum critério e na fórmula usada para fazer as estimativas do tempo de execução. Ambas as operações utilizam informações sobre o sistema e sobre a aplicação para tomar as decisões necessárias. Por isso, assumimos que temos acesso às seguintes informações sobre o sistema:

- k: número de processadores disponíveis.
- e_i: tempo gasto para que o processador p_i preencha uma posição da matriz
- $c_{i,j}$: tempo gasto para que o processador p_i se comunique com o processador p_j

Essas informações são fornecidas pelo módulo de descoberta de recursos, através de um tipo de dado estruturado, conforme visto na seção 6.2. No contexto do framework proposto e em relação à fórmula usada para o cálculo da estimativa de tempo de execução, implementamos cinco políticas de alocação. Uma delas é a política fixa de alocação, que já discutimos anteriormente. Essa política não faz o ordenamento das máquinas e não calcula estimativas do tempo de execução. Ao invés disso, apenas divide igualmente as colunas da matriz entre todos os processadores do sistema. As outras políticas não são fixas, porque o número de processadores usados no mapeamento pode ser menor do que o número total de processadores do sistema.

6.7.2.1 Política ProcCom

A primeira diferença entre a política ProcCom e a política fixa, é que a política ProcCom ordena as máquinas do sistema de acordo com o poder de processamento delas. Assim, as máquinas que apresentarem um maior poder de processamento receberão as primeiras tarefas. Justificamos a opção por ordenar as máquinas dessa forma porque estamos lidando com aplicações baseadas em programação dinâmica, que apresentam as dependências do padrão *wave-front* e, assim, assumirmos que a maior parte do tempo de execução da aplicação será gasta no processamento das colunas da matriz atribuídas a cada processador. Desta forma, privilegiando as máquinas com maior poder de processamento, visa-se reduzir o tempo de execução total da aplicação.

A política ProcCom considera que o tempo total de execução da aplicação é composto pelo tempo de processamento gasto por cada processador e pelo tempo gasto na comunicação entre eles. Essa política assume que a parte relativa ao tempo de processamento será limitada pelo tempo gasto pelo processador mais lento, entre os que estão sendo utilizados. Quanto ao tempo de comunicação, considera-se que ele será função do volume total de dados transmitidos entre os processadores e o custo de comunicação entre eles.

Quando faz-se a estimativa para um processador, o custo de comunicação é zero. Já o custo de processamento é dado pelo tempo gasto pelo processador

preencher a matriz inteira. Considerando que a matriz possui dimensões nXm, o custo total com um processador é dado pela equação 6.1.

$$t_1 = e_1 * m * n \tag{6.1}$$

à medida em que passamos a fazer as estimativas com mais processadores, passamos a levar em conta o custo de comunicação entre eles. Pelos exemplos vistos nas figuras 5.8 à 5.12 é possível perceber que um processador, ao término da aplicação, terá enviado ao outro processador uma coluna da matriz. Lembrando que a matriz tem dimensões nXm, o processador terá enviado m dados para o outro processador. Quanto ao custo de processamento, sempre que adicionamos mais um processador, a matriz é dividida igualmente entre eles, o que significa dizer que cada processador possui uma matriz de dimensões nX$(\frac{m}{i}$, onde i é o número de processadores sendo considerados. Na política ProcCom, o custo de processamento será dado unicamente pelo tempo gasto pelo processador mais lento preencher a sua matriz. Assim, o custo total quando consideramos i processadores, com i podendo variar entre 2 e k, é dado pela fórmula 6.2.

$$t_i = e_i * (\tfrac{m}{i}) * n + n * \sum_{j=1}^{i-1} c_{j,j+1} \tag{6.2}$$

Assim, de acordo com essa abordagem, calculamos uma estimativa para o tempo de execução usando apenas um processador, por meio da equação 6.1 e, iterativamente, adicionamos mais processadores, calculando uma estimativa de tempo usando esses novos processadores, através da equação 6.2. Quando a estimativa calculada for pior do que a estimativa anteriormente encontrada, significa que já temos o melhor número de processadores a ser utilizado.

6.7.2.2 Política ProcComSinc

Apesar da política ProcCom levar em consideração tanto o custo de processamento quanto o custo de comunicação entre os processadores, ela ignora a necessidade de sincronização entre os processadores em instantes específicos. Em outras palavras, ela assume que todos os processadores trabalham em paralelo durante todo o tempo, o que não é verdade. Em especial, durante o início do processamento (preenchimento da primeira linha da matriz), bem como no final dele, o grau de paralelismo é baixo, devido às dependências originadas do padrão *wave-front* (seção 5.4).

Da mesma maneira que na política ProcCom, na política ProcComSinc as máquinas também são primeiramente ordenadas de acordo com o poder de processamento. Além disso, busca-se levar em consideração, além do tempo de processamento e do tempo de comunicação, a necessidade de sincronização da aplicação. Obviamente, para a estimativa inicial, que considera apenas um processador, o custo total será estimado da mesma forma que na primeira abordagem.

Entretanto, quando utilizamos mais processadores, a estimativa será calculada de forma diferente. O funcionamento dessa segunda abordagem é melhor explicado por meio de um exemplo. As figuras 6.13 à 6.16 mostram o processo considerando-se 3 processadores.

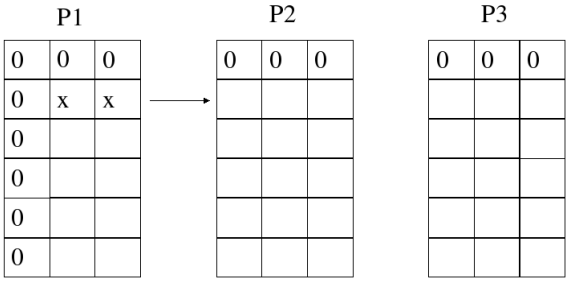


Figura 6.13: Inicialmente, apenas p_1 pode trabalhar. Quando p_1 terminar de preencher a sua parte da primeira linha, ele envia dados para p_2

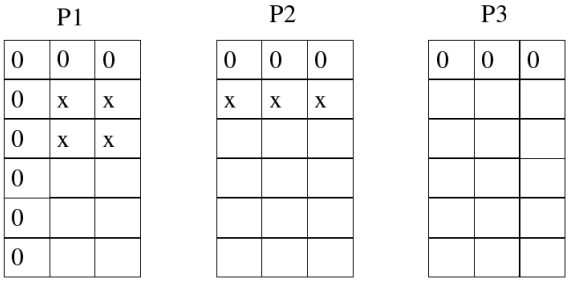


Figura 6.14: p_1 e p_2 podem trabalhar em paralelo. p_3, entretanto, deve continuar esperando

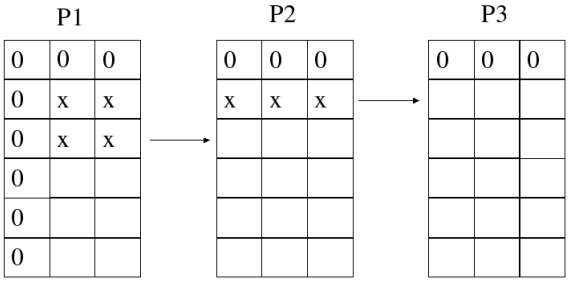


Figura 6.15: p_2 termina de preencher a sua parte da primeira linha e envia dados para p_3

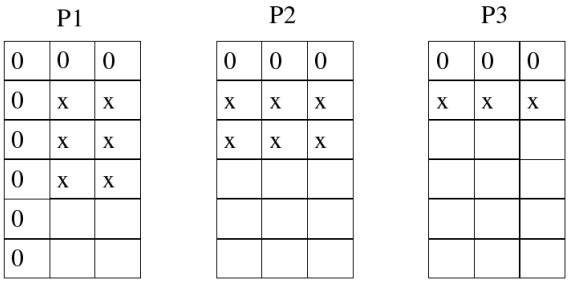


Figura 6.16: Os três processadores podem trabalhar em paralelo

Inicialmente, apenas o processador p_1 pode trabalhar. Os outros processadores não podem começar porque dependem de informações a serem recebidas. O processador p_1 demora um tempo igual a $e_1 * \frac{m}{3}$ para preencher a sua parte da

linha da matriz. Depois que p_1 processar uma linha da matriz, ele envia dados para p_2, com um custo igual a $C_{1,2}$ (figura 6.13). Nesse momento, tanto p_1 quanto p_2 podem trabalhar em paralelo. O processador p_3, entretanto, deve continuar esperando (figura 6.14). Assumimos agora que o custo de processamento nesse momento é limitado por p_2, pois tanto os processadores que estão antes quanto os que estão depois dependem do término de seu processamento para prosseguir. O tempo gasto por p_2 para preencher a linha será $e_2 * \frac{m}{3}$. Depois de preencher a linha, p_2 enviará dados para p_3, com um custo igual a $C_{2,3}$ (figura 6.15). Neste momento, os 3 processadores estarão trabalhando em paralelo (figura 6.16). Da mesma forma que fizemos com p_2 anteriormente, assumimos agora que o tempo de processamento será limitado pelo tempo gasto por p_3, pois todos os processadores dependem dele. O tempo gasto por p_3 será dado por $e_3 * \frac{m}{3}$. Uma vez que p_3 é o último processador, a partir desse momento assumimos que o tempo de processamento sempre dependerá dos custos envolvendo p_3. Como a matriz possui n linhas, teremos $n * C_{2,3}$ para o custo de comunicação e $n * e_3 * \frac{m}{3}$ para o custo de processamento. Com base nesse exemplo, chega-se à fórmula 6.3 para a estimativa com mais de um processador.

$$t_i = \frac{m}{i} * \sum_{j=1}^{i-1}(e_j) + \sum_{j=1}^{i-2}(c_{j,j+1}) + n * (c_{i-1,i} + e_i * \frac{m}{i}) \tag{6.3}$$

Assim como na política ProcCom, calculamos uma estimativa para o tempo de execução usando apenas um processador, por meio da equação 6.1 e, iterativamente, adicionamos mais processadores, calculando uma estimativa de tempo usando esses novos processadores, através da equação 6.3. Quando a estimativa calculada for pior do que a estimativa anteriormente encontrada, significa que já temos o melhor número de processadores a ser utilizado.

Outro ponto em comum entre a política ProcComSinc e a política ProcCom é que ambas, na hora de criar o mapeamento, dão preferência aos processadores com maior poder de processamento, ou seja, as máquinas são ordenadas de acordo com o poder de processamento de cada uma.

6.7.2.3 Política ComProcSinc

A política ComProcSinc é, na verdade uma variação da política ProcComSinc. A fórmula a ser utilizada no cálculo das estimativas ainda é a da equação 6.3. Entretanto, nessa política, as máquinas são inicialmente ordenadas pelo custo de comunicação entre elas, de modo que as máquinas que apresentarem um menor custo de comunicação terão preferência no recebimento de tarefas. Essa opção foi feita para que fosse possível comparar esse critério de ordenação com o critério usado pela política ProcComSinc.

6.7.3 Política ComProc

A política ComProc é, na verdade uma variação da política ProcCom. A fórmula a ser utilizada no cálculo das estimativas ainda é a da equação 6.2. Entretanto,

nessa política, as máquinas são inicialmente ordenadas pelo custo de comunicação entre elas, de modo que as máquinas que apresentarem um menor custo de comunicação terão preferência no recebimento de tarefas.

Capítulo 7

Resultados Experimentais

Este capítulo tem por objetivo descrever e comentar os resultados obtidos na execução de uma aplicação de comparação local de seqüências utilizando a estrutura do *framework* proposto e as políticas de alocação propostas.

A primeira seção descreve o ambiente de testes da aplicação, incluindo os recursos de hardware e software utilizados. A segunda seção apresenta a metodologia utilizada para coletar as informações necessárias à execução da aplicação. Por fim, a terceira seção apresenta os resultados obtidos na execução da aplicação.

7.1 Ambiente de Execução

O ambiente de testes da aplicação possui 10 computadores, dos quais sete são AMD e três são Pentium 4, de 1.7 GHz. A Tabela 7.1 lista as características de cada máquina.

Em relação ao software utilizado, cada uma das máquinas executa o Linux como sistema operacional. Os programas foram compilados pelo gcc. Além disso, foi utilizada uma implementação da interface MPI2, para permitir a comunicação entre tarefas nos computadores e para permitir a criação dinâmica de tarefas. A tabela 7.2 relaciona os softwares utilizados.

As 10 máquinas disponíveis estão dispostas em dois laboratórios distintos, Laico e Labpós. conforme mostra a figura 7.1. As máquinas pos-03, pos-04, pos-06, pos-08, pos-09, pos-10 e pos-14 ficam no Labpós, enquanto que as máquinas fau, magicien e carbona ficam no Laico. As máquinas de um laboratório se comunicam com as máquinas do outro laboratório através de um *switch*.

Nome	**Processador**	**Memória**	**HD**
pos-03, pos-04 pos-06, pos-08 pos-09, pos-10 pos-14	AMD Athlon, 1 GHz	256 MB	18GB
fau, magicien carbona	Pentium 4, 1.7GHz	256 MB	28 GB

Tabela 7.1: Características dos computadores utilizados

Nome	Versão	Descrição
RedHat Linux	3.2.2-5	Sistema Operacional
gcc	3.2	Compilador C
mpich2	1.0.3	Implementação do MPI2

Tabela 7.2: Características dos softwares utilizados

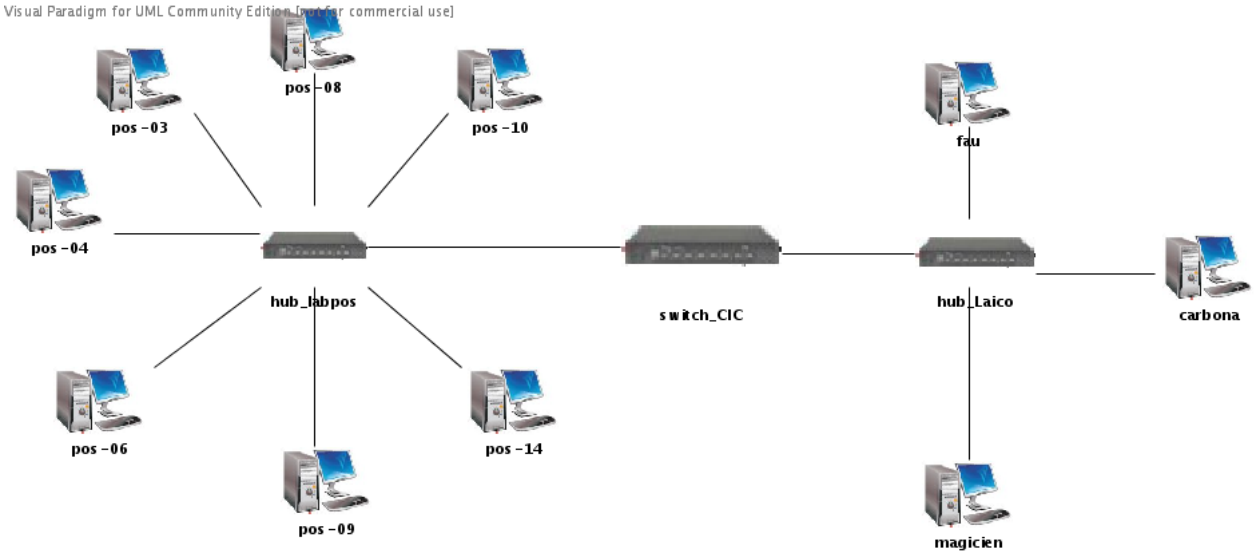


Figura 7.1: Esquema do ambiente de testes

7.2 Coleta de Informações sobre o Sistema

Para que fosse possível utilizar a estrutura do *framework* e as políticas de alocação definidas nele, era preciso obter informações sobre o sistema distribuído em questão. Conforme foi visto na seção 6.2, é preciso que o módulo de descoberta de recursos do *framework* seja capaz de indicar quantas e quais máquinas estão disponíveis, além de fornecer uma medida do poder computacional de cada uma delas e uma medida do custo de comunicação entre elas.

O assunto de descoberta de recursos é vasto, com vários trabalhos específicos sobre o tópico. Uma vez que o foco desse trabalho não é na descoberta de recursos em si, mas sim na alocação de tarefas no sistema, optou-se por uma abordagem simples para a descoberta de recursos: as informações necessárias foram previamente coletadas e armazenadas em um arquivo texto, que foi lido em cada execução da aplicação, de modo a fornecer as informações em questão.

Para obter a medida do poder computacional de cada máquina, optou-se por determinar uma estimativa do tempo necessário para cada uma delas preencher uma posição da matriz de similaridade criada na comparação local de seqüências (capítulo 5). A fim de se conseguir essa estimativa, preparou-se uma versão seqüencial da aplicação, com seqüências pequenas, que foi executada 25 vezes em cada uma dessas máquinas, registrando-se o tempo total de cada execução. Ao final, tirou-se a média desses tempos de execução e dividiu-se o valor obtido pelo tamanho da matriz. Desse modo, temos uma estimativa do tempo médio necessário para um processador preencher uma única posição da matriz, uma

operação básica da aplicação. Quanto menor esse valor, mais rápido a máquina em questão é capaz de preencher a matriz, o que equivale a dizer que ela possui um maior poder computacional. A figura 7.2 mostra os valores obtidos (em segundos) no ambiente de testes usado.

Quanto à medida do custo de comunicação entre duas máquinas, utilizou-se o tempo necessário para realizar um *ping* entre duas máquinas. Assim, foi executado o *ping* de cada uma das máquinas para cada uma das outras máquinas. Foram feitas 25 execuções, sempre registrando-se o tempo demorado. Em seguida, cada um desses tempo foi divido por dois, uma vez que o tempo do *ping* envolve a ida e a volta de uma mensagem entre duas máquinas e o interesse para o contexto desse trabalho estava apenas no tempo de ida da mensagem. Os novos valores, para cada par de máquinas, foram somados e tirou-se a média deles. Dessa forma, foi possível obter uma estimativa do tempo médio da transmissão de uma mensagem entre duas máquinas quaisquer e, quanto maior esse valor, maior o custo de comunicação entre as máquinas. A figura 7.3 mostra os valores obtidos (em segundos) no ambiente de testes usado.

Máquina	Tempo para preencher uma posição da matriz de similaridade
pos-03	0,000426895
pos-04	0,000459714
pos-06	0,000463029
pos-08	0,000460648
pos-09	0,000462419
pos-10	0,000461371
pos-14	0,000462152
fau	0,000270248
magicien	0,000261410
carbona	0,000261333

Figura 7.2: Tempos de preenchimento de posição da matriz de similaridade

	pos-08	pos-06	pos-04	pos-03	pos-14	pos-10	pos-09	fau	magicien	carbona
pos-08	0.0	0.08618	0.0844	0.0721	0.09376	0.0944	0.08964	0.19358	0.18994	0.1885
pos-06	0.0991	0.0	0.07174	0.08526	0.07812	0.08182	0.0807	0.1918	0.18722	0.18588
pos-04	0.08618	0.08938	0.0	0.08502	0.0806	0.08256	0.08166	0.19208	0.18958	0.18598
pos-03	0.14556	0.16884	0.16692	0.0	0.1868	0.18872	0.18792	0.38864	0.37372	0.37528
pos-14	0.17692	0.15152	0.1442	0.17376	0.0	0.17204	0.16512	0.38992	0.3894	0.37704
pos-10	0.0892	0.07754	0.07648	0.08784	0.0802	0.0	0.08446	0.19556	0.19146	0,19700
pos-09	0.09288	0.0784	0.0731	0.09362	0.0817	0.0822	0.0	0.18564	0.18308	0.18972
fau	0.19798	0.20926	0.19526	0.20276	0.20962	0.21208	0.19938	0.0	0.15866	0.15152
magicien	0.20634	0.21182	0.21222	0.2042	0.21044	0.2107	0.20352	0.18162	0.0	0.14902
carbona	0.20472	0.20564	0.20516	0.21158	0.20014	0.20872	0.20346	0.18208	0.1622	0.0

Figura 7.3: Tempos de comunicação entre as máquinas

7.3 Avaliação das Políticas de Alocação

Baseamos os nossos testes na estrutura do framework do proposto e na implementação paralela da comparação local de seqüências pelo algoritmo de *Smith-Waterman* que é descrita em [46] e que foi comentada na seção 5.5.4. Para que essa implementação fosse usada no *framework*, foram necessárias pequenas alterações no seu código, de modo a fazer com que ela ficasse encapsulada no módulo de execução do *framework* e pudesse ser chamada a partir do módulo principal.

As comparações foram feitas com seqüências sintéticas. Em todas as execuções dos testes, o Componente Mestre do *framework* inicia o processamento, recuperando as informações do sistema e criando o mapeamento entre tarefas e processadores. Logo em seguida, esse componente cria dinamicamente as tarefas trabalhadoras (Componente Worker), via uma chamada MPI_Comm_spawn_multiple, definida no MPI2.

7.3.1 Primeiro experimento

No primeiro experimento realizado, utilizamos seqüências de 1 KB, 2 KB e 4 KB, fazendo as seguintes comparações:

- Um arquivo com 100 seqüências de 1K e um arquivo com uma seqüência de 1K;
- Um arquivo com 100 seqüências de 1K e um arquivo com uma seqüência de 2K;
- Um arquivo com 100 seqüências de 2K e um arquivo com uma seqüência de 1K;
- Um arquivo com 100 seqüências de 2K e um arquivo com uma seqüência de 2K;
- Um arquivo com 100 seqüências de 4K e um arquivo com uma seqüência de 1K;
- Um arquivo com 100 seqüências de 4K e um arquivo com uma seqüência de 2K;

Nesse primeiro experimento, verifica-se o tempo de cada comparação sendo feita e, a cada grupo de cem comparações, somava-se esses tempos, de modo a obter o tempo gasto para se fazer 100 comparações. Além disso, verificamos também o tempo total de se executar a aplicação, registrando nesse total não apenas o tempo de se fazer as 100 comparações, mas também o tempo de se iniciar a aplicação, gerar o mapeamento, criar as tarefas e finalizar as tarefas criadas. Esse procedimento foi executado para cada uma das políticas de alocação descritas no capítulo 6, de maneira a ser possível comparar o desempenho da aplicação utilizando quantidades de processadores diferentes e validar a hipótese de que, para seqüências pequenas, a utilização de um número reduzido de processadores

pode ser mais vantajosa do que a utilização do número total de processadores disponíveis.

Um ponto importante a ser observado é que, nesse experimento, à cada comparação a ser feita, era gerada uma nova alocação de tarefas e essas novas tarefas eram criadas no *framework*. Uma vez que cada execução da aplicação fazia 100 comparações, esse procedimento (gerar a alocação e criar tarefas) era executado também 100 vezes.

A figura 7.4 exibe os resultados obtidos para a comparação entre 100 seqüências de 1K e uma seqüência de 1K. Nessa figura, temos a informações sobre a política utilizada, o número de processadores escolhidos pelo política, o tempo de fazer as 100 comparações, o tempo total da execução da aplicação, o *speedup* do tempo das comparações em relação à política fixa e o *speedup* do tempo total em relação à política fixa. Estamos utilizando termo *speedup* para denominar o ganho relativo de desempenho entre um determinada política de alocação e a política fixa, ou seja, o *speedup* está sendo dado pela divisão do tempo gasto utilizando-se a política fixa pelo tempo gasto utilizando-se uma outra política.

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	7,539	155,20184	1	1
ProcCom	3	4,028	47,74052	1,871648461	3,250945737
ProcComSinc	3	4,529	47,06171	1,664605873	3,297836819
ComProcSinc	5	6,312	86,20786	1,194391635	1,800321224
ComProc	8	5,135	120,52839	1,468159688	1,287678695

Figura 7.4: Resultados obtidos na comparação entre 100 seqüências de 1K e uma seqüência de 1K

A primeira linha de resultados da figura 7.4 diz respeito à utilização da política fixa de alocação, que sempre irá utilizar o número máximo de processadores (que é igual a 10, no caso). As outras linhas dizem respeito à utilização de outras políticas de escalonamento. Pela análise dos resultados, podemos observar que:

- Tanto o tempo das comparações quanto o tempo total são reduzidos quando se utiliza um número de processadores menor do que o número alocado pela política fixa;
- As políticas que resultaram no melhor desempenho foram a ProcCom (seção 6.7.2.1) e a ProcComSinc (seção 6.7.2.2), o que valida a idéia de que, apesar do custo de comunicação envolvido, o tempo de processamento é determinante para o desempenho da aplicação, no nosso ambiente de testes.

Essas conclusões podem ser observadas na figura 7.5, que mostra, no lado esquerdo, uma comparação gráfica entre os tempos obtidos e, no lado direito, o *speedup* obtido em relação à política fixa.

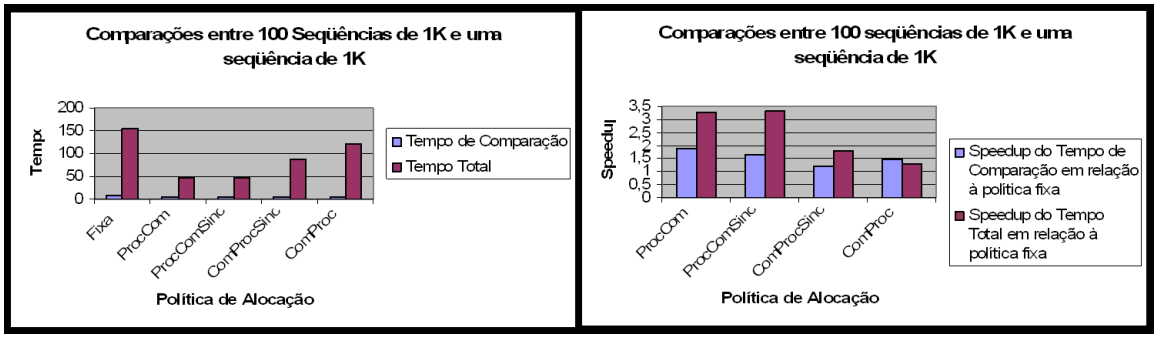


Figura 7.5: Resultados obtidos na comparação entre 100 seqüências de 1K e uma seqüência de 1K (comparação dos tempo de processamento e speedup em relação à política fixa

Política	**Tempo gasto (s)**
fixa	0,0000519753
procCom	0.0025441647
procComSinc	0.0011198521
comProcSinc	0.0016481876
comProc	0.0004179478

Tabela 7.3: Tempos gastos pelo módulo de alocação de tarefas para obter o mapeamento entre processadores e tarefas

Pela análise das figuras, torna-se claro que as políticas ProcCom e ProcComSinc exibiram um melhor desempenho. Por sua vez, os tempos obtidos pela utilização das políticas ComProc e ComProcSin, que utilizam o custo de comunicação para escolher os processadores a serem utilizados, não foram tão bons, apesar de ainda serem um pouco melhores do que os obtidos pela utilização da política fixa.

Um ponto importante a ser notado é que o tempo total da execução da aplicação, que envolve a criação e finalização das tarefas, é muito maior do que o tempo gasto apenas nas comparações. Além disso, a utilização de menos máquinas possibilitou um *speedup* no tempo total que é significativamente maior do que o *speedup* do tempo das comparações apenas. Isso nos leva à crer que o custo envolvido nos processos de criação, inicialização e finalização de tarefas é bastante alto.

Para confirmar que o *overhead* observado no tempo total de execução é decorrente dos processos de criação, inicialização e finalização de tarefas e não da geração do mapeamento entre tarefas e processadores, medimos o tempo gasto pelo módulo de alocação de tarefas na operação de obter o mapeamento. A tabela 7.3 mostra esses tempos.

A tabela 7.3 mostra que os tempos de obtenção do mapeamento são muito baixos. Podemos multiplicar esses tempo por 100, a fim de saber o tempo total gasto na obtenção dos mapeamento de 100 comparações, mas mesmo assim, esse tempo é baixo, em relação ao tempo total da aplicação. Sendo assim, podemos concluir que as operações de criação, inicialização e finalização de tarefas são as

responsáveis pela diferença entre o tempo gasto nas comparações e o tempo total da aplicação.

A figura 7.6 mostra os resultados do experimento para as comparações entre 100 seqüências de 1K e uma seqüência de 2K.

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	10,529	161,81416	1	1
ProcCom	3	5,268	56,3354	1,998671222	2,872335334
ProcComSinc	3	5,217	57,35509	2,018209699	2,821269394
ComProcSinc	5	9,416	91,40552	1,118203059	1,770288709
ComProc	8	6,614	129,64637	1,591926217	1,248119481

Figura 7.6: Resultados obtidos na comparação entre 100 seqüências de 1K e uma seqüência de 2K

Podemos observar que os resultados obtidos seguem o mesmo padrão daqueles da figura 7.4, com as políticas ProcCom e ProcComSinc apresentando os melhores desempenhos. Os gráficos da figura 7.7 também mostram uma tendência similar aos da figura 7.5.

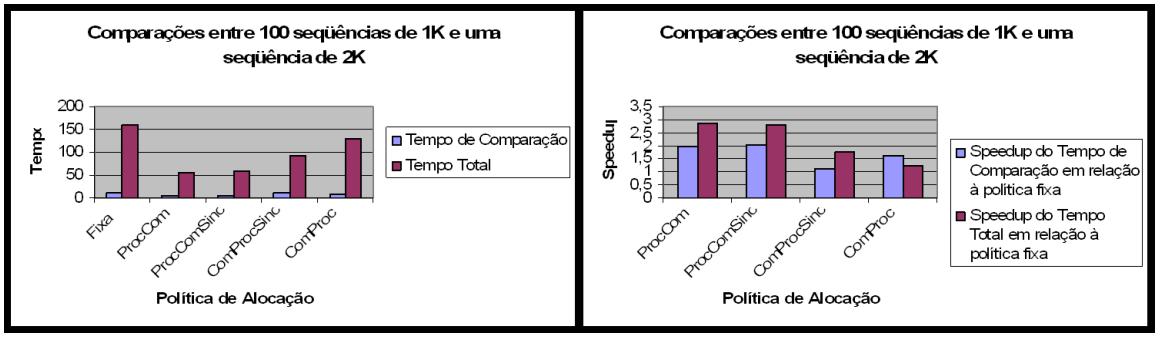


Figura 7.7: Resultados obtidos na comparação entre 100 seqüências de 1K e uma seqüência de 2K (comparação dos tempo de processamento e speedup em relação à política fixa

Nos gráficos em questão, continua-se a observar que o *speedup* do tempo total de execução é significativamente maior do que o *speedup* do tempo das comparações, o que reforça a idéia de que os custos de criação, inicialização e finalização das tarefas são altos.

As figuras 7.8 à 7.11 mostram os resultados obtidos no experimento 1 para as comparações entre 100 seqüências de 2 K e uma seqüência de 1K e para as comparações entre 100 seqüências de 2K e uma seqüência de 2K.

As figuras mais uma vez exibem o mesmo padrão, ou seja, a utilização de um número de processadores menor do que o número total é vantajosa; as políticas

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	8,858	162,38552	1	1
ProcCom	3	5,275	49,62863	1,679241706	3,272012949
ProcComSinc	3	5,326	48,51497	1,663161848	3,34712193
ComProcSinc	5	7,563	91,28648	1,171228349	1,77885619
ComProc	8	5,151	128,20412	1,719666084	1,266617017

Figura 7.8: Resultados obtidos na comparação entre 100 seqüências de 2K e uma seqüência de 1K

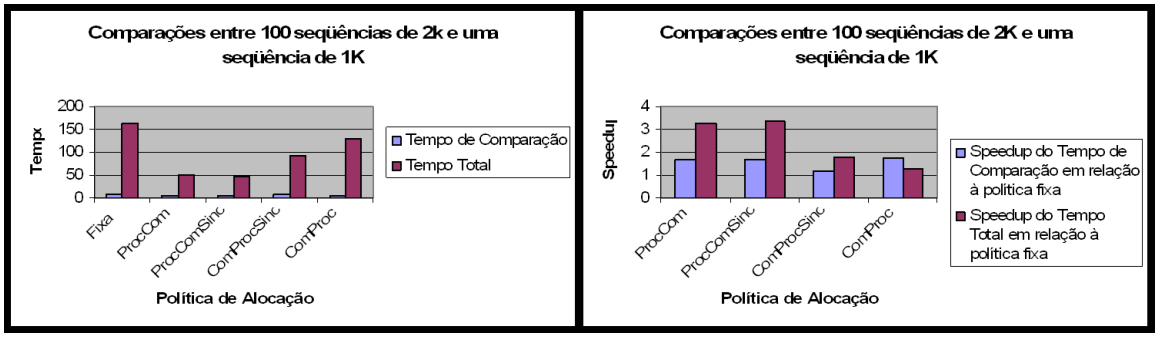


Figura 7.9: Resultados obtidos na comparação entre 100 seqüências de 2K e uma seqüência de 1K (comparação dos tempo de processamento e speedup em relação à política fixa

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	11,053	166,86152	1	1
ProcCom	3	7,075	50,95674	1,562261484	3,274572117
ProcComSinc	3	7,07	50,07842	1,563366337	3,332004484
ComProcSinc	5	10,695	101,54622	1,033473586	1,643207596
ComProc	8	6,868	131,06667	1,609347699	1,273104139

Figura 7.10: Resultados obtidos na comparação entre 100 seqüências de 2K e uma seqüência de 2K

ProcCom e ProcComSinc apresentam melhor desempenho; e o tempo total de execução da aplicação é bastante superior ao tempo gasto apenas nas comparações. Um outro ponto interessante a ser notado é que, para as políticas que baseiam a escolha dos processadores nos custos de comunicação (ComProc e Com-

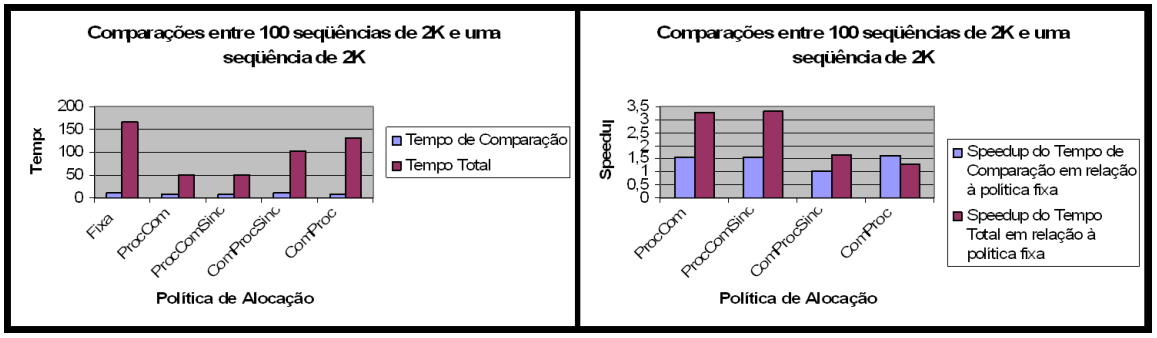


Figura 7.11: Resultados obtidos na comparação entre 100 seqüências de 2K e uma seqüência de 2K (comparação dos tempo de processamento e speedup em relação à política fixa

ProcSinc), a utilização de 8 máquinas (política ComProcSinc) é mais vantajosa do que a utilização de 5 máquinas (política ComProc).

Por fim, as figuras 7.12 à 7.15 mostram os resultados obtidos no experimento 1 para as comparações entre 100 seqüências de 4 K e uma seqüência de 1K e para as comparações entre 100 seqüências de 4K e uma seqüência de 2K.

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	9,146	165,43298	1	1
ProcCom	3	6,862	48,20798	1,332847566	3,431651357
ProcComSinc	3	6,844	49,96264	1,33635301	3,311133679
ComProcSinc	8	5,353	130,87753	1,708574631	1,264028898
ComProc	8	5,53	133,80547	1,653887884	1,236369335

Figura 7.12: Resultados obtidos na comparação entre 100 seqüências de 4K e uma seqüência de 1K

Os resultados das figuras reforçam os padrões observados nos resultados anteriores. Uma outra conclusão que pode ser feita com base na análise desses resultados é que, à medida em que o tamanho das seqüências aumenta, o ganho pela utilização de menos processadores do que o número máximo disponível diminui. Em outras palavras, os resultados mostram a tendência de que, quando as seqüências são grandes, é mais vantajoso utilizar um número maior de máquinas. Essa tendência mostra-se especialmente clara ao percebermos que, para as seqüências de 4K, as políticas comProc e comProcSinc, que escolhem as máquinas com base nos custos de comunicação, apresentaram, pela primeira vez até agora, um resultado melhor do que o obtido pelas políticas procCom e procComSinc, mesmo tendo selecionado mais processadores.

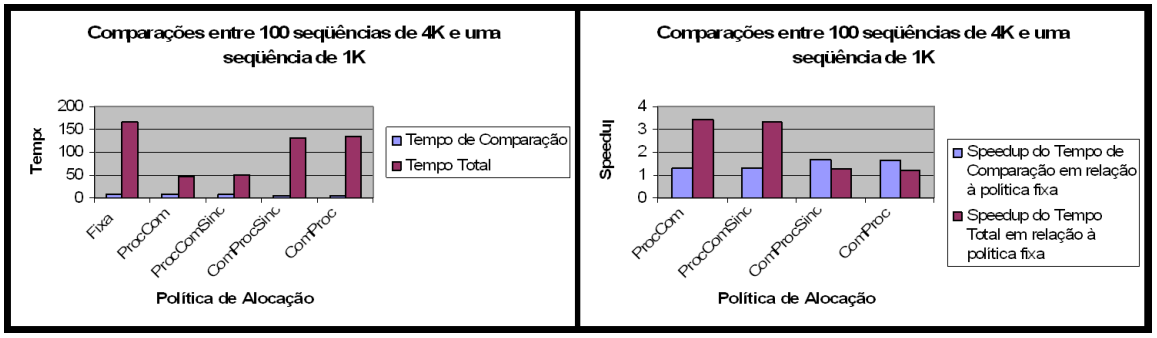


Figura 7.13: Resultados obtidos na comparação entre 100 seqüências de 4K e uma seqüência de 1K (comparação dos tempo de processamento e speedup em relação à política fixa

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	11,655	171,88619	1	1
ProcCom	3	9,9	55,75832	1,177272727	3,082700304
ProcComSinc	3	10	55,917	1,1655	3,073952286
ComProcSinc	8	7,828	139,58709	1,48888605	1,23139031
ComProc	8	8,001	135,32657	1,456692913	1,270158477

Figura 7.14: Resultados obtidos na comparação entre 100 seqüências de 4K e uma seqüência de 2K

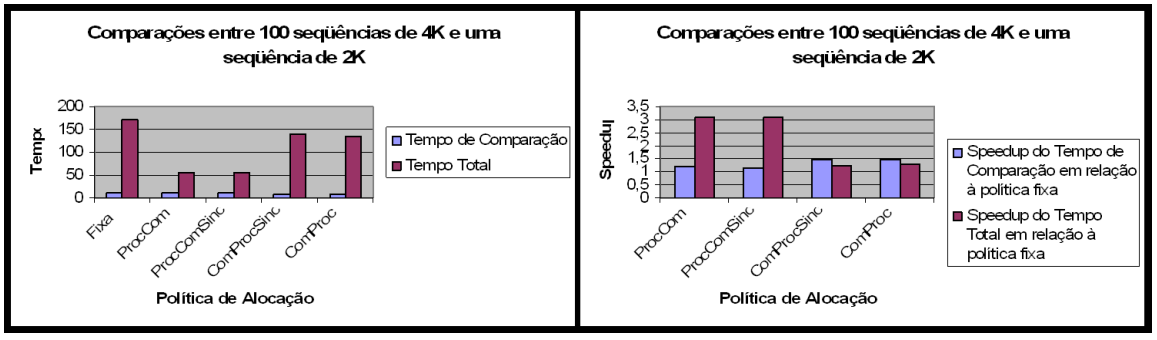


Figura 7.15: Resultados obtidos na comparação entre 100 seqüências de 4K e uma seqüência de 2K (comparação dos tempo de processamento e speedup em relação à política fixa

7.3.2 Segundo Experimento

Os resultados do primeiro experimento, no qual a cada comparação eram criadas novas tarefas, mostram que o custo de criação, inicialização e finalização de tarefas é bastante elevado, visto que o tempo total de execução medido é significativamente maior do que o tempo gasto apenas nas comparações. A fim de

investigar esse resultado, procedeu-se com um segundo experimento. Nesse segundo experimento, também utilizamos seqüências sintéticas de 1 KB, 2 KB e 4 KB, fazendo as seguintes comparações:

- Cem comparações entre um arquivo com uma seqüência de 1K e um arquivo com uma seqüência de 1K;
- Cem comparações entre um arquivo com uma seqüência 1K e um arquivo com uma seqüência de 2K;
- Cem comparações entre um arquivo com uma seqüência 2K e um arquivo com uma seqüência de 1K;
- Cem comparações entre um arquivo com uma seqüência 2K e um arquivo com uma seqüência de 2K;
- Cem comparações entre um arquivo com uma seqüência 4K e um arquivo com uma seqüência de 1K;
- Cem comparações entre um arquivo com uma seqüência 4K e um arquivo com uma seqüência de 2K;

De maneira similar ao primeiro experimento, medimos o tempo de se executar apenas as comparações e o tempo total gasto desde o início da aplicação até o final. Esse procedimento também foi executado para cada uma das políticas de alocação descritas no capítulo 6.

Entretanto, ao contrário do primeiro experimento, não havia a criação de novas tarefas a cada comparação a ser feita. Ao invés disso, quando a aplicação era iniciada, gerava-se o mapeamento entre tarefas e processadores, criava-se as tarefas e então, a mesma comparação era executada 100 vezes, uma após a outra, pelas mesmas tarefas criadas anteriormente. Dessa forma, esperava-se que o tempo total da execução da aplicação ficasse mais próximo do tempo gasto nas comparações, o que comprovaria o *overhead* das operações de criação, inicialização e finalização de tarefas.

As figuras 7.16 à 7.19 mostram os resultados obtidos no experimento 2 para as comparações entre uma seqüência de 1K com outra seqüência de 1K e para as comparações entre uma seqüência de 1K com uma seqüência de 2K.

De fato, podemos observar nos resultados exibidos nas figuras 7.16 à 7.19 que, agora que não estão sendo criadas novas tarefas a cada comparação a ser feita, a diferença entre o tempo total da aplicação e o tempo gasto nas comparações é significativamente menor, sendo que os dois tempos são próximos um do outro. Esses resultados levam a crer, portanto, a hipótese de que a criação, inicialização e finalização de tarefas a cada comparação, feita no experimento 1, implica em um considerável *overhead* na execução da aplicação.

Outras conclusões que podem feitas a partir da análise dos resultados apresentados 7.16 à 7.19 são similares às conclusões do experimento 1:

- é mais vantajoso, no caso de seqüências pequenas, utilizar um número de processadores menor do que o número total disponível;

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	4,393	5,56762	1	1
ProcCom	3	2,017	2,77067	2,17798711	2,00948507
ProcComSinc	3	2,09	2,55028	2,101913876	2,183140675
ComProcSinc	5	3,917	5,13092	1,121521573	1,085111442
ComProc	8	3,508	5,17538	1,252280502	1,075789604

Figura 7.16: Resultados obtidos na comparação uma seqüência de 1K e outra seqüência de 1K

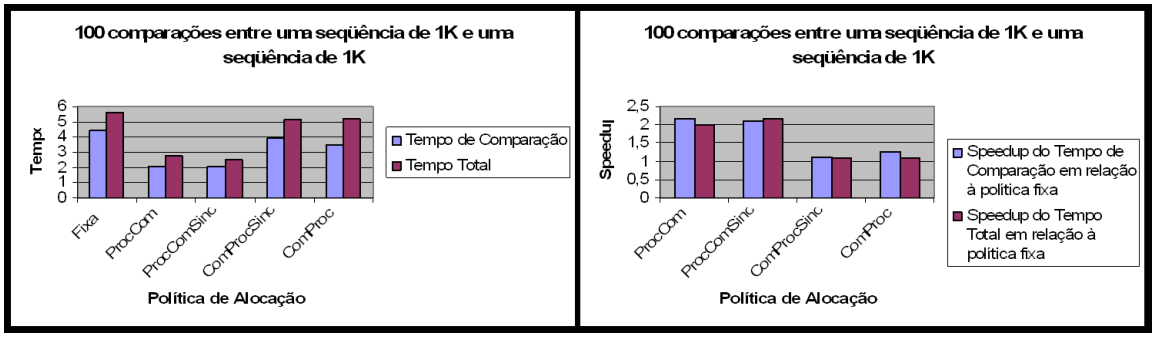


Figura 7.17: Resultados obtidos na comparação uma seqüência de 1K e outra seqüência de 1K (comparação dos tempo de processamento e speedup em relação à política fixa

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	7,247	8,92732	1	1
ProcCom	3	3,717	4,8297	1,949690611	1,848421227
ProcComSinc	3	3,921	4,57064	1,848252997	1,953188175
ComProcSinc	5	7,283	10,45485	0,995056982	0,853892691
ComProc	8	7,271	8,45453	0,996699216	1,0559215

Figura 7.18: Resultados obtidos na comparação uma seqüência de 1K e uma seqüência de 2K

- As políticas de alocação que selecionam os processadores com base no poder de processamento (ProcCom e ProcComSinc) apresentam melhor desempenho, o que comprova que o custo de processamento é o fator dominante no tempo da aplicação no ambiente de testes utilizado;

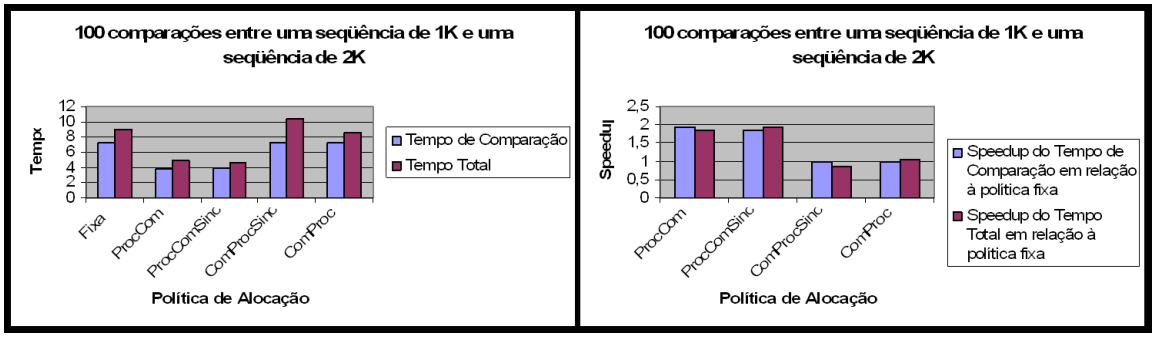


Figura 7.19: Resultados obtidos na comparação uma seqüência de 1K e uma seqüência de 2K (comparação dos tempo de processamento e speedup em relação à política fixa

- As políticas de alocação que selecionam os processadores com base no custo de comunicação (ComProc e ComProcSinc) apresentam ganhos pequenos em relação à política fixa, sendo que observou-se, em alguns casos, um desempenho pior do que o obtido com a política fixa de alocação.

Para as comparações entre seqüências de 2K e 1K e entre seqüências de 2K e 2K, os resultados obtidos estão ilustrados nas figuras 7.20 à 7.23.

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	4,464	5,84596	1	1
ProcCom	3	2,64	3,97017	1,690909091	1,472470952
ProcComSinc	3	2,685	3,12009	1,662569832	1,873651081
ComProcSinc	5	5,656	6,81346	0,789250354	0,858001661
ComProc	8	5,594	6,83342	0,797997855	0,855495491

Figura 7.20: Resultados obtidos na comparação uma seqüência de 2K e uma seqüência de 1K

Esses resultados mais uma vez comprovam as observações feitas até o momento, com o ganho de desempenho na utilização das políticas ProcCom e ProcComSinc. Uma vez mais, observa-se casos em que a utilização das políticas ComProc e ComProcSinc resulta em uma perda de desempenho, em relação à política fixa.

Por fim, realizamos o experimento 2 para as comparações entre uma seqüência de 4K e uma seqüência de 1K e para as comparações entre uma seqüência de 4K e uma seqüência de 2K. Os dados obtidos são mostrados nas figuras 7.24 à 7.27.

Podem ser observadas as mesmas tendências observadas nos outros resultados. Além disso, torna-se claro, pela análise das figuras 7.24 e 7.27, que a utilização de poucos processadores, à medida em que o tamanho das seqüências aumenta, não

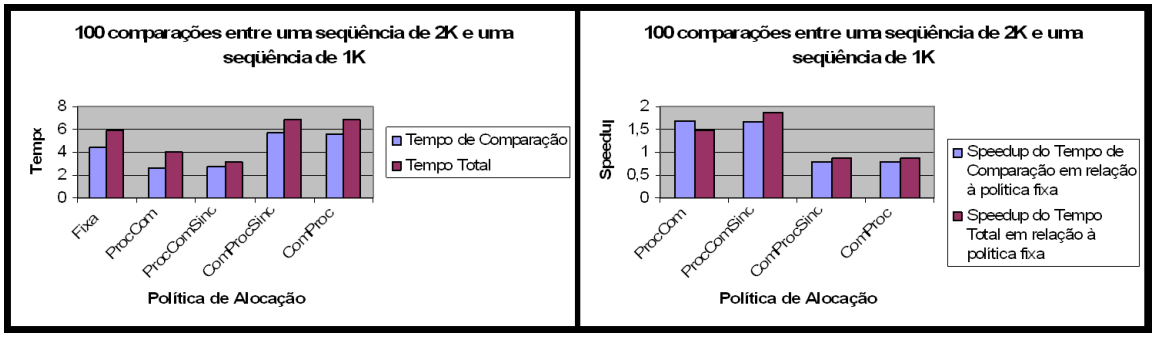


Figura 7.21: Resultados obtidos na comparação uma seqüência de 2K e uma seqüência de 1K (comparação dos tempo de processamento e speedup em relação à política fixa

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	8,099	10,13574	1	1
ProcCom	3	4,758	5,6849	1,702185792	1,782923182
ProcComSinc	3	5,097	5,46135	1,588973906	1,85590376
ComProcSinc	5	8,184	9,60431	0,989613881	1,05533245
ComProc	8	7,851	9,48003	1,031588333	1,069167503

Figura 7.22: Resultados obtidos na comparação uma seqüência de 2K e outra seqüência de 2K

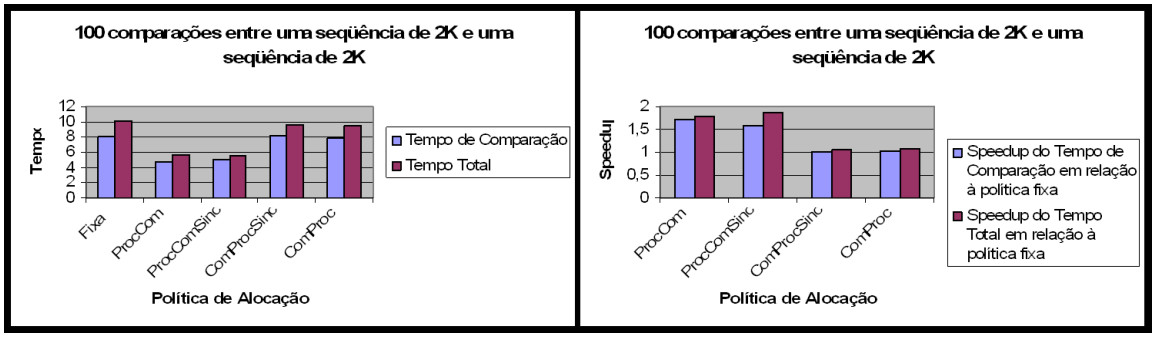


Figura 7.23: Resultados obtidos na comparação uma seqüência de 2K e outra seqüência de 2K (comparação dos tempo de processamento e speedup em relação à política fixa

possibilita muitos ganhos no desempenho da aplicação, ou seja, para seqüências grandes, a melhor estratégia é utilizar um número grande de processadores.

Os resultados obtidos, tanto no experimento 2 quanto no experimento 1, permitem a conclusão que, caso o número de máquinas disponíveis no ambiente de testes fosse maior, as diferenças entre os desempenhos das políticas de alocação

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	6,135	7,20741	1	1
ProcCom	3	4,835	5,83235	1,268872802	1,235764315
ProcComSinc	3	5,29	5,44427	1,15973535	1,323852417
ComProcSinc	8	5,992	7,82348	1,023865154	0,921253713
ComProc	8	5,638	7,09054	1,088151827	1,016482525

Figura 7.24: Resultados obtidos na comparação uma seqüência de 4K e uma seqüência de 1K

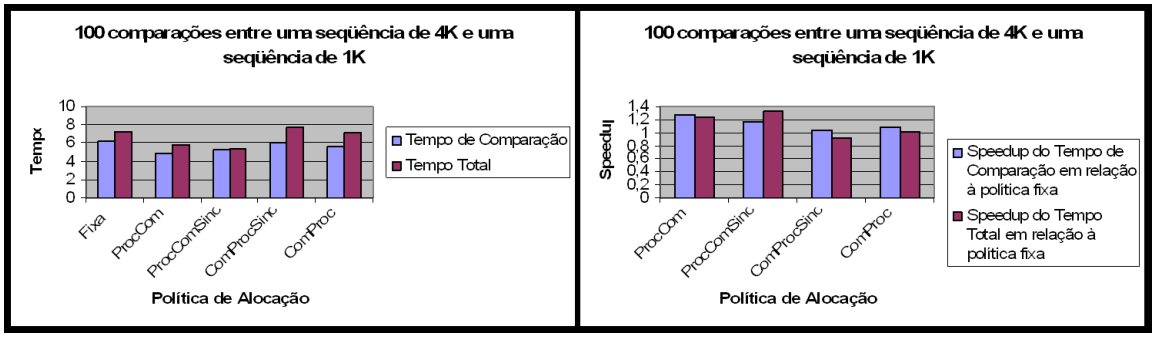


Figura 7.25: Resultados obtidos na comparação uma seqüência de 4K e uma seqüência de 1K (comparação dos tempo de processamento e speedup em relação à política fixa

Política	Número de máquinas	Tempo de Comparação	Tempo Total	Speedup do Tempo de Comparação em relação à política fixa	Speedup do Tempo Total em relação à política fixa
Fixa	10	10,695	11,90984	1	1
ProcCom	3	9,882	10,61539	1,082270795	1,12194088
ProcComSinc	3	10,08	10,43429	1,061011905	1,141413551
ComProcSinc	8	11,541	12,94902	0,926696127	0,919748367
ComProc	8	11,313	12,93106	0,94537258	0,921025809

Figura 7.26: Resultados obtidos na comparação uma seqüência de 4K e uma seqüência de 2K

em relação à política fica seria ainda maior, uma vez que a política fixa sempre atribui tarefas a todos os processadores disponíveis. No caso das seqüências pequenas, é provável que, se houvesse mais máquinas disponíveis, a diferença entre o desempenho obtido pela utilização da política fixa e o desempenho obtido pela

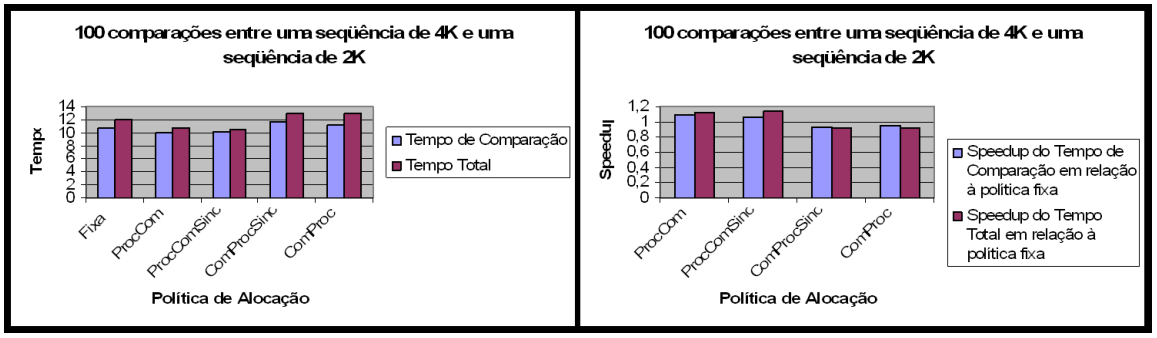


Figura 7.27: Resultados obtidos na comparação uma seqüência de 4K e uma seqüência de 2K (comparação dos tempo de processamento e speedup em relação à política fixa

utilização das políticas ProcCom e ProcComSinc seria ainda maior, deixando ainda mais claro o fato de que, para seqüências pequenas, a opção mais vantajosa não é utilizar todos os processadores disponíveis.

Capítulo 8

Conclusão

A presente dissertação apresentou o projeto, a implementação e a avaliação de um *framework* de alocação de tarefas de aplicações de comparação de seqüências biológicas baseadas em programação dinâmica para ambientes heterogêneos, além de quatro políticas de alocação de tarefas com essas características. Utilizamos um ambiente de teste heterogêneo composto por dez máquinas distribuídas em dois laboratórios diferentes e realizamos experimentos envolvendo baterias de cem comparações com seqüências de 1K, 2K e 4K, de modo a observar o comportamento das políticas de alocação para a comparação dessas seqüências, quando contrastadas com uma política fixa de alocação de tarefas.

Os resultados obtidos mostram que, no ambiente de testes que utilizamos, o tempo necessário para se realizar as comparações das seqüências e o tempo total de execução da aplicação, nos casos em que as seqüências são pequenas, são reduzidos quando se utiliza um número de processadores menor do que o número total de processadores disponível. Além disso, as políticas que possibilitaram o melhor desempenho, quando as seqüências eram pequenas, foram as políticas que atribuíam tarefas primeiro aos processadores que possuíam um melhor poder de processamento. Esse resultado mostrou que, apesar de existir um custo de comunicação e sincronização entre as tarefas, o tempo de execução é dominado pelo custo de processamento das tarefas nos processadores selecionados.

Um outro resultado observado mostrou que, na execução de uma aplicação que realiza várias comparações entre seqüências, o custo de criação, inicialização e finalização das tarefas exerce forte influência no tempo total de execução da aplicação. Nos experimentos que realizamos em que, à cada comparação a ser feita, eram criadas novas tarefas, o tempo total de aplicação foi consideravelmente maior do que o tempo necessário apenas para se fazer as comparações. Foi possível comprovar esse fato no experimento em que só eram criadas tarefas para a primeira comparação a ser feita e então repetia-se essa comparação várias vezes usando as mesmas tarefas criadas anteriormente.

Existem diversos trabalhos que podem ser realizados de modo a dar continuidade às atividades desenvolvidas e descritas nessa dissertação. Entre esses trabalho, podemos citar:

- Trabalhar no módulo de descoberta de recursos do *framework*, integrando-o com algum sistema de descoberta de recursos já existente ou desenvolvendo

um novo sistema;

- Avaliação das políticas de alocação de tarefas em outro ambientes que possuam mais processadores e redes de características diferentes;
- Projetar, implementar e avaliar novas políticas de alocação dentro da arquitetura do *framework* proposto;
- Instanciar o *framework* com outras aplicações baseadas em programação dinâmica, de modo a validar melhor a arquitetura proposta;
- Instanciar o *framework* para outras aplicações paralelas de comparação de seqüências biológicas, como aquela proposta em [6]; e
- Adaptar o *framework* para ambientes *multi-cluster* e propor políticas de alocação de tarefas específicas para ambientes desse tipo, de modo similar ao trabalho descrito em [13].

Referências

[1] http://www-unix.mcs.anl.gov/mpi/.

[2] http://www-unix.mcs.anl.gov/mpi/mpich/.

[3] http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 214:403–410, 1990.

[5] A. Auyeung, I. Gondra, and H. K. Dai. Multi-heuristic list scheduling genetic algorithm for task scheduling. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 721–724. ACM Press, 2003.

[6] R. B. Batista and A. C. M. A. de Melo. Z-align: An exact parallel strategy for biological sequence alignment in user-restricted memory space. In *IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2006.

[7] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The grid: blueprint for a new computing infrastructure*, pages 279–309. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[8] A. Boukerche, A. C. M. A. de Melo, M. E. T. Walter, R. C. F. Melo, M. N. P. Santana, and R. B. B. A performance evaluation of a local dna sequence alignment algorithm on a cluster of workstations. In *18th International Parallel and Distributed Processing Symposium*, 2004.

[9] J. R. Budenske, R. S. Ramanujan, and H. J. Siegel. On-line use of off-line derived mappings for iterative automatic target recognition tasks and a particular class of hardware platforms. In *6th Heterogeneous Computing Workshop (HCW '97)*, pages 96–. IEEE Computer Society, 1997.

[10] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, pages 349–363, Cancun, Mexico, 2000. IEEE Computer Society.

[11] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[12] C. Chen and B. Schmidt. Computing large-scale alignments on a multi-cluster. In *2003 IEEE International Conference on Cluster Computing*, pages 38–45. IEEE Computer Society, 2003.

[13] C. Chen and B. Schmidt. An adaptive grid implementation of dna sequence alignment. *Future Gener. Comput. Syst.*, 21(7):988–1003, 2005.

[14] W. Cirne, D. P. da Silva, L. Costa, E. Santos-Neto, F. Brasileiro, J. P. Sauvé, F. A. B. Silva, C. O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. In *32nd International Conference on Parallel Processing (ICPP 2003)*, pages 407–. IEEE Computer Society, 2003.

[15] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, C. Mendes, Z. Shi, A. YarKhan, and J. Dongarra. New grid scheduling and rescheduling methods in the grads project. In *Proceedings of NSF Next Generation Systems Program Workshop*, pages 199–206, Santa Fe, New Mexico, April 2004. in conjunction with IPDPS'2004.

[16] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[17] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison Wesley, 3rd edition, 2001.

[18] J. Cuenca, D. Giménez, and J. P. Martínez. Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. In *3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004), 3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogenous Networks (HeteroPar 2004)*, pages 354–361. IEEE Computer Society, 2004.

[19] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998.

[20] Y. Dandass. A genetic algorithm for sceduling directed acyclic graphs in the presence of communication contention.

[21] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Workflow management in griphyn. In J. Nabrzyski, J. M. Schopf, and J. Weglarz, editors, *Grid Resource Management*. Kluwer Academic Publishers, 1st edition, 2003.

[22] H. El-Rewini and H. Ali. On considering communication in scheduling task graphs on parallel processors. *Journal of Parallel Algorithms and Applications*, 3:177–191, 1994.

[23] H. El-Rewini and T. G. Lewis. *Distributed and Parallel Computing*. Manning Publications Co., 1997.

[24] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. on Computers*, (C-21):948–960, 1972.

[25] I. Foster. *Designing and Building ParallelPrograms*. Addison-Wesley, 1995.

[26] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-area implementation of the message passing interface. *Parallel Comput.*, 24(12-13):1735–1749, 1998.

[27] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[28] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NPCompleteness*. W.H. Freeman and Company, 1979.

[30] J. L. Hennessy and D. A. Patterson. *Arquitetura de Computadores: Uma abordagem Quantitativa*. Editora Campus, 3rd edition, 2003.

[31] S. Hu, W. Shi, and Z. Tang. Jiajia: an svm system based on a new cache coherence protocol. In *High Performance Computing and Networking (HPCN)*, pages 463–472. Springer-Verlag, 1999.

[32] T. Hu. Parallel sequencing and assembly line problems. *Operations Research*, (9):841–848, 1961.

[33] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the earth-manna multithreaded system. *Int. J. Parallel Program.*, 24(4):319–348, 1996.

[34] K. Hwang and Z. Xu. *Scalable Paralel Computing: Technology, Architecture, Programming*. McGraw-Hill, 1st edition, 1998.

[35] L. Iftode, J. P. Singh, and K. Li. Scope consistency: a bridge between release consistency and entry consistency. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 277–287. ACM Press, 1996.

[36] E. G. Coffman Jr. and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[37] Y. Kwok and I. Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 47(1):58–77, 1997.

[38] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[39] A. Legrand, H. Renard, Y. Robert, and F. Vivien. Mapping and load-balancing iterative computations. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):546–558, 2004.

[40] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, page 57, Washington, DC, USA, 1998. IEEE Computer Society.

[41] W. S. Martins, J. del Cuvillo, W. Cui, and G. R. Gao. Whole genome alignment using a multithreaded parallel implementation. In *Proc. of the 13th Symposium on Computer Architecture and High Performance Computing*, 2001.

[42] A. C. M. A. Melo, M. E. M. T. Walter, R. C. F. Melo, M. N. P. Santana, and R. B. Batista. Local dna sequence alignment in a cluster of workstations: Algorithms and tools. *Journal of the Brazilian Computer Society*, 10(2):69–76, 2004.

[43] S . Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[44] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence analysis. *Proc. Natl. Acad. Sci. U.S.A.*, 85:2444–2448, 1988.

[45] Gregory F. Pfister. *In search of clusters (2nd ed.)*. Prentice-Hall, Inc., 1998.

[46] E. Sandes, A. C. M. A. de Melo, and M. Ayala-Rincon. Comparação paralela exata de seqüências biológicas em clusters de computadores. In *4th International Information and Telecommunication Technologies Symposium*, 2005.

[47] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS, 1997.

[48] Gary Shao. Adaptive scheduling of master/worker applications on distributed computational resources, 2001.

[49] O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, 2005.

[50] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[51] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 3(1):85–93, 1977.

[52] A. S. Tanembaum. *Sistemas Operacionais Modernos*. Prentice Hall do Brasil, 1995.

[53] K. Taura and A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 102. IEEE Computer Society, 2000.

[54] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 0:0–20, 2004.

[55] R. Wolski, N. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999.