MODELAGEM DE DESEMPENHO DE SISTEMAS COM PARALELISMO PIPELINE

EMANUEL VIANNA DO VALLE

MODELAGEM DE DESEMPENHO DE SISTEMAS COM PARALELISMO PIPELINE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais. Deptartamento de Ciência da Computação. como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Virgílio Augusto Fernandes de Almeida
Co-orientadora: Jussara Marques de Almeida

Belo Horizonte
Julho de 2011

do Valle, Emanuel Vianna.

V181m Modelagem de desempenho de sistemas com paralelismo pipeline / Emanuel Vianna do Valle. — Belo Horizonte, 2011.
xvi, 110f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

Orientador: Virgílio Augusto Fernandes de Almeida.
Coorientadora: Jussara Marques de Almeida.

1. Computação - Teses. 2. Análise de Desempenho – Teses. 3. I.Orientador. II. Coorientadora. III. Título.

CDU 519.6*24 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Modelagem de desempenho de sistemas com paralelismo pipeline

EMANUEL VIANNA DO VALLE

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. VIRGÍLIO AUGUSTO FERNANDES ALMEIDA - Orientador
Departamento de Ciência da Computação - UFMG

PROFA. JUSSARA MARQUES DE ALMEIDA - Co-orientadora
Departamento de Ciência da Computação - UFMG

PROF. MAGNOS MARTINELLO
Departamento de Informática - UFES

PROF. PAULO ROMERO MARTINS MACIEL
Centro de Informática - UFPE

PROF. HUMBERTO TORRES MARQUES NETO
Departamento de Ciência da Computação - PUC/MG

Belo Horizonte, 29 de julho de 2011.

Agradecimentos

Este foi um trabalho que contou com o apoio de muita gente. Primeiramente, foi muito bom conviver com o Virgílio, exemplo de pessoa, que sempre me passou tranquilidade, me fez interessar um pouco mais por política e com sua ampla experiência em *performance* foi decisivo em vários momentos do trabalho, me fazendo sempre sair de sua sala com um *paper* e cheio de vontade de trabalhar. Agradeço também a Jussara, que tanto me ajudou neste trabalho, por sua disponibilidade, seu profissionalismo, sua competência e sua percepção. Na primeira metade do projeto, trabalhei junto com a Tat e a Marisa, todas duas super-dinâmicas, com quem aprendi bastante. Depois entrou o Giovanni, extremamente competente, que consegue fazer do trabalho uma diversão, tornando o trabalho mais produtivo. Me senti muito honrado de trabalhar com o grupo de pesquisa da HP, Umesh, Kevin e Harumi, com quem reunimos quase todas as sextas-feiras em um ambiente muito agradável. Agradeço enormemente ao Humberto, Magnos e Paulo, membros da banca, pela disponibilidade para participar da defesa e pela critériosa revisão da dissertação. Em particular ao Humberto, meu orientador da graduação, que me incentivou a entrar no mestrado e sempre me deu o maior apoio. Agradeço a Beth, minha namorada, que suportou a minha ausência; a meus pais, que me apoiaram e me aturaram neste período de muito estudo; a meus avós e a meu padrasto que contribuíram para minha formação; e aos velhos amigos de infância, do Dom Silvério e da PUC (em especial ao Dudu, Odon, Christiano e o Mário que entraram no mestrado junto comigo e a turma do Void-Main). O dia-a-dia no laboratório CAMPS foi muito agradável, o pessoal tem bastante opinião própria, tendo rendido boas discussões e boas risadas. Valeu Rauber, Giovanni, Matheus, Diego del Chile, Marisa, Tat, Fabrício, Adriano, Djim, Césão, Pesce, Saulo, Rapha, Douglas, Las Casas, Evandro! Neste período, fiz parte de um time de futebol, chamado Rebolation Futebol Clube (RFC), montado pela turma que entrou comigo no mestrado, para participar do campeonato do DCC, com quem dividi vitórias e derrotas e fiz boas amizades. A todos vocês, meu muito obrigado!

"Que o breve
seja de um longo pensar
Que o longo
seja de um curto sentir
Que tudo seja leve
de tal forma
que o tempo nunca leve."
(Alice Ruiz)

Resumo

Em uma era onde o paralelismo é imperativo (*commodity clusters*, multiprocessadores, múltiplos núcleos, GPGPU) o entendimento do consumo de recursos (CPU, disco, etc.) de uma aplicação paralela, em diferentes circunstâncias, é chave para tomar decisões em planejamento de capacidade e gerenciamento de carga de trabalho, tais como: (1) quantas e quais tipos de máquinas são necessárias, em um *cluster* de um *data center*, para atender os limites de tempos de resposta e *throughput* contratados no acordo de nível serviço (*Service Level Agreement* - SLA)? (2) como ajustar os parâmetros para tirar melhor proveito dos recursos e aumentar o desempenho e a escalabilidade do sistema? No entanto, técnicas de modelagem de desempenho tradicionais, como a Análise do Valor Médio (*Mean Value Analysis* - MVA) não podem ser aplicadas diretamente em cargas de trabalho que possuam relações de precedência, como as sincronizações entre tarefas produtoras e consumidoras em *jobs* que possuam paralelismo *pipeline*. Soluções exatas, utilizando cadeias de Markov para representar os possíveis estados do sistema foram propostas na literatura, mas não são escaláveis pois o espaço de estados cresce exponencialmente com o aumento do número de tarefas. A metodologia utilizada nesta dissertação é baseada em uma solução aproximada, mas eficiente, proposta em um trabalho anterior, por Liang e Tripathi (2000), o qual chamamos de modelo referência. A solução analítica do modelo referência utiliza um modelo hierárquico, onde o nível mais alto (*software*) é representado por um grafo de precedência e o nível mais baixo (*hardware*) por um modelo de rede de filas fechado. O grafo de precedência captura os atrasos de sincronização e a sobreposição média do tempo de execução de tarefas de um mesmo *job* (intra-job) e de tarefas de *jobs* diferentes (inter-job), enquanto que o modelo de rede de filas captura a contenção média nos recursos. O modelo de rede de filas é resolvido através da técnica Analise do Valor Médio aproximado (*approximate Mean Value Analysis* - aMVA). Para introduzir o efeito das regras de precedências no algoritmo aMVA, o tamanho médio das filas, que usualmente só captura a contenção dos recursos, foram inflacionados por fatores de sobreposição, estimados pelo grafo de precedência. O modelo referência permite

diversos tipos de relações de precedências entre as tarefas, referidos como operadores. Entretanto nenhum de seus operadores captura explícitamente as sincronizações de um *pipeline*, não sendo portanto aplicável diretamente à cargas de trabalho que possuam paralelismo *pipeline*. A nossa principal contribuição consiste em mostrar como construir o grafo de precedência, utilizando os operadores primitivos proposto no modelo referência, para capturar as sincronizações inerentes do paralelismo *pipeline*. Esta metodologia foi aplicada em dois estudos de casos: (1) consultas simples do banco de dados distribuído HP *Neoview*, que contém um *pipeline* na troca de mensagens entre duas tarefas; e (2) *jobs* do *Hadoop Online Prototype* (HOP), uma extensão do *Hadoop* (arcabouço de computação paralela inspirado no modelo de programação *MapReduce*) que provê um paralelismo *pipeline* entre múltiplas tarefas. Os principais resultados encontrados foram: (1) obtivemos uma boa aproximação na validação do modelo de consulta do HP *Neoview*, obtendo um erro relativo máximo de 11,5% para o tempo de resposta do *job* em relação a um simulador da rede de filas e a um emulador do sistema, para seis cenários diferentes (na literatura, o máxima aceito para tempo de resposta é 30%); (2) observamos que o paralelismo *pipeline* teve um impacto maior, com relação ao tempo médio de resposta do *job*, para carga leve, ou seja, antes do sistema saturar. Após a saturação o tempo médio de resposta do *job* é dominado pelos atrasos de filas e, um aumento no paralelismo *pipeline*, ao invés de aumentar o *speedup*, gera mais contenção; (3) no segundo estudo de caso, verificamos que as previsões de desempenho do modelo analítico para os *jobs* HOP tiveram boa acurácia para cenários com pouco paralelismo, no entanto, ao aumentar o paralelismo, suas previsões foram muito super-estimados. Avaliamos estes cenários, buscando identificar se havia alguma característica do sistema que não foi capturada, ou que não estava sendo bem modelada. Desenvolvemos simuladores específicos para avaliar as premissas adotadas no modelo e identificamos que a principal razão para não termos obtido uma boa acurácia nestes cenários foi devido a uma premissa do modelo referência que assume que a média do tempo de resposta das tarefas são exponencialmente distribuídas. Fizemos uma avaliação das limitações do modelo devido às implicações desta premissa e introduzimos um novo operador primitivo usado na construção do grafo de precedência, que utiliza o método *fork/join*, proposto anteriormente por Varki (1999), para estimar os fatores de sobreposição. A acurácia do modelo utilizando o método *fork/join* melhorou, ficando o erro relativo máximo do tempo de resposta do *job* abaixo de 15%.

Abstract

In an era where the parallelism is imperative (commodity clusters, multi-processor multi-core, GPGPU) the understanding of resource consumption (CPU, disk, etc.) of a parallel application, in different circunstances, is key to decision-making in capacity planning and workload management, such as: (1) how many and what types of machines are needed, in a data center cluster, to support the agreed thresholds for response time and throughput in the service level agreement (SLA)? (2) how should system parameters be fine-tuned to increase system performance and scalability? However, traditional modeling techniques, such as Mean Value Analysis (MVA), can not be directly applied to workloads that have precedence constraints, such as the synchronizations among producers and consumers tasks of a job that present pipeline parallelism. Exact solutions, using Markov chains to represent the possible states of the system were proposed in literature, but are not scalable since the state space grows exponentially with the increasing of the number of tasks. The methodology used in this dissertation is based on an approximate, but efficient, solution, proposed in a previous work by Liang e Tripathi (2000), refereed as reference model. The analytical solution of the reference model uses a hierarchical model, where the highest level (software) is represented by a precedence graph and the lowest level (hardware) by a closed queuing network model. The precedence graph captures the synchronization delays and the average overlap of the execution time of tasks in the same job (intra-job) and tasks of different jobs (inter-job), while the queuing network model captures the average resource contention. The queuing network model is solved through the Approximate Mean Value Analysis technique (aMVA). To introduce the effect of the precedence rules in the aMVA algorithm, the average queue size, which usually only captures the resource contention, were inflated by overlap factors, estimated by the precedence graph. The reference model allows various types of precedence relationships, refereed as operators. Meanwhile none of its operators capture explicitely the synchronizations of a pipeline and therefore it is not directly applicable to the workloads that have pipeline parallelism. Our main contribution is show how to build the precedence graph, using

the primitive operators proposed in the reference model, to capture the inherent synchronization delay of pipeline parallelism. This methodology was applied in two case studies: (1) simple queries of the distributed database HP Neoview, which contains a pipeline in the message exchange between two tasks, and (2) jobs of the Online Hadoop Prototype (HOP), an extension of Hadoop (framework for parallel computing inspired by MapReduce programming model) to provide a pipeline parallelism among multiple tasks. The main findings were: (1) we obtained a good approximation in the validation of HP Neoview model, compared to a queuing netowrk simulator and an emulator of the system, achieving a maximum relative error of 11.5% for job average response time, for six different scenarios (in literature the maximum acceptable for response time is 30%), (2) we observed that the pipeline parallelism had a major impact, with respect to job average response time, for light load, i.e., before the system saturates. After the saturation, the job average response time is dominated by queue delays and an increase in the pipeline parallelism, instead of increase the speedup, generates more contention, (3) in the second case study, we found that the performance predictions of the analytical model for HOP jobs had a good accuracy for scenarios with little parallelism, however, when we increase the pipeline parallelism its predictions were far overestimated. We evaluate these scenarios in order to identify if there was a characteristic of the system that was not captured, or that was not well modeled. We develop specific simulators to evaluate the adopted model assumptions and we identify that the main reason to our model does not present a good accuracy in these scenarios was a premise of the reference model which assumes that the average response time of tasks are exponentially distributed. We evaluate the limitations of the model due to the implications of this assumption and introduce a new primitive operator which uses the fork/join method, previously proposed by Varki (1999), to estimate the overlap factors. The analytical model using the fork/join method produces better performance metrics estimates, where the maximum relative error of job response time was less than 15%.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

FESC Flow Equivalent Service Center

GBD Generalized Birth-Death

TI Tecnologia da Informação

GPGPU General-Purpose computation on Graphics Processing Units

CPU Central Processing Unit

SLA Service Level Agreement

MVA Mean Value Analysis

aMVA Approximated Mean Value Analysis

BI Business Intelligence

HOP Hadoop Online Prototype

PARSEC Princeton Application Repository for Shared-Memory Computers

MPL Multi Programmin Level

CV Coefficient of Variation

LI Load Independent

PASTA Poisson Arrivals See Time Averages

FCFS First Come First Served

OLTP Online Transaction Processing

CDF Cumulative Distribution Function

PS Processor Sharing

HDFS Hadoop File System

TAD Type Abstract of Data

v.a. Variável Aleatória

i.i.d Independente e igualmente distribuída

SPN Stochastic Petri Nets

Sumário

Capítulo 1

Introdução

O volume de dados produzidos na Internet vem crescendo consideravelmente. Por exemplo, a *Yahoo!* em 2007 já produzia diariamente um total de 25 TB (Kriegel et al., 2007). A extração de conhecimento destas grandes fontes de dados requer um processamento paralelo massivo por empresas que demandam serviços de TI e também na área acadêmica, para o desenvolvimento de pesquisas em áreas como banco de dados, mineração de dados, aprendizado de máquina e redes complexas. O crescimento do volume de dados produzido pela Internet e a dificuldade inerente para seu processamento é um dos grandes desafios da era da informação (Hopcroft, 2010).

Para suportar o processamento de grandes bases de dados, a indústria de arquiteturas paralelas teve um grande avanço em multi-processadores, múltiplos núcleos, GPGPU, etc. A computação em nuvem, onde os servidores de processamento e armazenamento são compartilhados através da Internet, surgiu como um paradigma robusto para suportar aplicações com intenso paralelismo (Wang et al., 2009). Neste contexto, novos modelos de computação emergiram como importantes modos de instanciar uma computação em nuvem e simplificar o desenvolvimento de aplicações, tais como *Grid* (Foster e Kesselman, 1999) e o *MapReduce* (Dean e Ghemawat, 2004).

Em termos de *software*, a computação paralela vêm sendo explorada em vários níveis. A independência funcional de tarefas de um *job* permite que múltiplas tarefas sejam executadas ao mesmo tempo (paralelismo de tarefas). Conforme o tipo de operação é possível particionar os dados e processá-los separadamente (paralelismo de dados). A existência de restrições de precedência tal como, uma tarefa B só pode ser executada após uma tarefa A, podem, caso a tarefa A demore muito tempo, engessar o processamento e comprometer o desempenho do sistema. Dependendo do grau de dependência entre as tarefas, pode-se dividir seu processamento em estágios, estabelecendo uma relação produtor-consumidor entre as tarefas (paralelismo em *pipeline*).

Em uma era onde o paralelismo é imperativo, o entendimento do consumo de recursos (CPU, disco, etc.) de uma aplicação paralela é chave para tomar decisões em gerenciamento de carga de trabalho e planejamento de capacidade. O tempo de resposta é uma das métricas mais importantes na avaliação de desempenho de um sistema. Embora os usuários só percebam o tempo final de uma execução (Raatikainen, 1989), o conhecimento do tempo médio de resposta do sistema, sob diferentes circunstâncias, é fundamental para a tomada de decisões, tais como: (1) quantas e quais tipos de máquinas são necessárias em um *cluster* de um *data center* para atender os limites de tempo de resposta e *throughput* contratados no acordo de nível serviço (*Service Level Agreement* - SLA)? (2) como ajustar os parâmetros (*tunning*) para tirar melhor proveito dos recursos e aumentar o desempenho e a escalabilidade do sistema?

Para suportar estas decisões é importante o desenvolvimento de modelos de desempenho confiáveis, que consigam capturar as principais características (comportamento médio) do sistema alvo, considerando os diferentes tipos de paralelismo presentes na carga de trabalho, de forma eficiente. Modelos de desempenho podem ser utilizados na fase de projeto do sistema, para auxiliar a encontrar qual a melhor configuração para implantação de um sistema (Menascé et al., 2004) ou, após sua implantação, para avaliar a eficiência (desempenho) de um sistema em produção (Menascé et al., 2004; Lazowska et al., 1984). Outra aplicação dos modelos de desempenho é auxiliar a configuração dos parâmetros (*tunning*) do sistema (Raman et al., 2008), sem ter que executar este sistema em uma enorme combinação de cenários.

1.1 Definição do Problema

Uma carga de trabalho é tipicamente representada por um conjunto de *jobs*, os quais são compostos por um conjunto de tarefas. Além do tempo gasto na execução, estas tarefas podem experienciar dois tipos de atraso: (1) atrasos de filas, devido ao compartilhamento de recursos e (2) atrasos de sincronização, devido a restrições de precedências entre tarefas que cooperam em um mesmo *job*. Por exemplo, suponha que um *job* J seja composto pelas tarefas A, B e C, sendo que C só pode iniciar após A e B finalizarem. Assim, mesmo que o sistema tenha recursos disponíveis para executar C, seu processamento não poderá ser iniciado enquanto suas dependências não tenham sido resolvidas, ou seja, A e B tenham terminado de processar.

A literatura é rica em técnicas de modelagem de desempenho para cargas de trabalho que não possuem atrasos de sincronização (Menascé et al., 2004). No entanto, técnicas de modelagem de desempenho tradicionais, como a Análise do Valor Médio (MVA - *Mean Value Analysis*) (Reiser e Lavenberg, 1980) não podem ser aplicadas diretamente em cargas de trabalho que possuam restrições de precedência, tais como as sincronizações entre tarefas de um *job* que possuas paralelismo *pipeline*, pois violam algumas premissas de redes separáveis (descritas na Seção 2.1). Em particular, a premissa que assume que a taxa de processamento do sistema não depende do número de requisições, uma vez que o número de tarefas (unidade de carga em aplicações paralelas) variam durante a execução de um *job*.

Soluções exatas (Thomasian e Bay, 1986; Kruskal e Weiss, 1985) utilizaram cadeias de Markov (Papoulis, 1964) para representar os possíveis estados do sistema, dado pelo número de tarefas concorrentes a cada momento, e uma rede de filas para calcular as taxas de transição entre os estados. Entretanto estas soluções não são escaláveis pois o espaço de estados cresce exponencialmente com o número de tarefas.

O modelo analítico adotado nesta dissertação foi desenvolvido sob uma solução aproximada, mas eficiente, proposta anteriormente por Liang e Tripathi (2000), o qual chamamos de modelo referência. Esta solução utiliza um modelo hierárquico, onde o nível mais alto (*software*) é representado por um grafo de precedência e o nível mais baixo (*hardware*) por um modelo de rede de filas fechado.

O grafo de precedência captura os atrasos de sincronização e a sobreposição média do tempo de execução de tarefas de um mesmo *job* (*intra-job*) e de tarefas de *jobs* diferentes (*inter-job*), enquanto que o modelo de filas captura a contenção média nos recursos (CPU, disco, etc.). O modelo de filas é avaliado através da técnica Análise do Valor Médio aproximado (*approximate Mean Value Analysis* - aMVA) (Lavenberg e Reiser, 1980). Para introduzir o efeito das regras de precedências no algoritmo aMVA, o tamanho médio das filas, que usualmente só captura a contenção dos recursos, foram inflacionados por fatores de sobreposição, estimados pelo grafo de precedência.

O modelo referência (Liang e Tripathi, 2000) permite a representação de diversos tipos de relações de precedências (tais como serial, paralelo, etc.) e apresenta um operador para combinar a distribuição do tempo de resposta de um sub-conjunto de tarefas, para cada uma das precedências consideradas. Entretanto estes operadores representam relações primitivas de precedências, não havendo um operador específico que modele as inúmeras relações de precedências que podem haver entre tarefas produtoras e consumidoras em um paralelismo *pipeline*. Sendo assim, nenhum dos operadores do modelo referência capturam as sincronizações de um *pipeline*, não sendo portanto aplicável diretamente à cargas de trabalho que possuam paralelismo *pipeline*.

1.2 Parceria UFMG/DCC e HP Brasil R&D

Esta dissertação está inserida em um projeto maior originado da parceria entre DCC/UFMG e HP Brasil R&D. O projeto foi coordenado pelos professores Virgílio Almeida e Jussara Almeida e teve a colaboração do grupo de pesquisa da área de banco de dados da HP-Labs de Palo Alto. O grupo da HP foi coordenado pelo pesquisador Umeshwar Dayal (HP *fellow*) e também participaram os pesquisadores Kevin Wilkinson e Harumi Kuno. Os alunos da UFMG envolvidos foram Emanuel Vianna (mestrado, participou dos dois anos do projeto), Tatiana Pontes (graduação, participou dos dois anos do projeto), Marisa Vasconcelos (doutorado, participou do 1º ano do projeto) e Giovanni Commarela (mestrado, participou do 2º ano do projeto). Em 2009, recebemos a visita do pesquisador Umeshwar ao DCC/UFMG, em visita ao Brasil. Semanalmente, foram realizadas reuniões, junto ao grupo da HP, através de áudio-conferência, onde foram discutidos os desafios encontrados no decorrer do projeto.

1.3 Objetivos

O objetivo inicial desta dissertação era modelar a carga de trabalho do banco de dados distribuído HP *Neoview* (Winter, 2009), utilizado como armazém de dados (*Data Warehouse*) para processamento de consultas transacionais (OLTP - *Online Transaction Processing*) e consultas de relatório (BI - *Business Intelligence*). Modelamos e validamos, através de simulação e emulação, a carga de trabalho composta por consultas simples de BI, que possuem paralelismo *pipeline* na troca de mensagem entre duas tarefas. Os resultados encontrados, apresentados no Capítulo 4, foram sumarizados em um artigo intitulado "*Modeling Intra-Query Paralelism for Business Intelligence Workloads*"e submetido para a conferência IFIP *Performance* 2010. Embora não tenha sido aceito, pois os cenários avaliados eram relativamente simples, reconheceu-se que a modelagem do *pipeline* é desafiadora.

O grupo reavaliou a estratégia e optou por mudar a aplicação alvo para o *Hadoop Online Prototype* (HOP) (Condie et al., 2010), proposto recentemente, que estende o *Hadoop* (uma implementação *open source* inspirada no *MapReduce* (Dean e Ghemawat, 2004) amplamente utilizada, desenvolvida pela Apache (2011)), para prover um paralelismo *pipeline* entre múltiplas tarefas, possuindo, assim, relações de precedências mais complexas. Os resultados da validação por simulação do segundo estudo de caso, apresentados no Capítulo 5, foram sumarizados em um artigo intitulado "*Modeling the Performance of Hadoop Online Prototype*", publicado na conferência SBAC-PAD'11.

A modelagem destas duas cargas de trabalho propiciou o desenvolvimento do método de modelagem de sistemas com paralelismo *pipeline* proposto nesta dissertação. Sendo assim, o objetivo desta dissertação é desenvolver um modelo de desempenho razoavelmente confiável, ou seja, com acurácia dentro dos limites aceitos na literatura, para a previsão de métricas de desempenho, tais como, tempo médio de resposta do *job*, *throughput* e utilização de recursos.

1.4 Principais Contribuições e Resultados Encontrados

Esta dissertação desenvolveu um modelo de desempenho que estende o método proposto por Liang e Tripathi (2000), aqui referido como modelo referência, para abordar o paralelismo *pipeline* entre múltiplas tarefas. Este modelo analítico foi aplicado a dois estudos de casos: (1) paralelismo *pipeline* entre duas tarefas de consultas do banco de dados distribuído HP *Neoview* e (2) o paralelismo *pipeline* entre múltiplas tarefas dos *jobs* do *Hadoop Online Prototype* (HOP).

A modelagem de desempenho da carga de BI foi validada por um simulador da rede de filas (desenvolvido pela UFMG) e por um emulador do sistema (desenvolvido pela HP), enquanto que a modelagem da carga do HOP foi validado através de um simulador (desenvolvido pelo UFMG), e por experimentos reais (realizados pela HP). O meu papel neste projeto foi na derivação e implementação do modelo analítico desenvolvido, tendo colaborado também nas outras atividades, como no desenvolvimento do simulador (principalmente no 2º estudo de caso) e na validação do modelo.

As contribuições do modelo de desempenho desenvolvido nesta dissertação em relação ao modelo referência foram:

- O modelo referência permite diversos tipos de precedências entre as tarefas (apresentadas na Seção 2.1), referidos como operadores. Entretanto nenhum de seus operadores captura explícitamente as sincronizações de um *pipeline*. Foi desenvolvido um método para modelagem do paralelismo *pipeline*, que utilizar os operadores do modelo referência para construir o grafo de precedência correspondente.

- Diante de alguns cenários que foram super-estimados pelo modelo (Seção 5.4.2), avaliamos outros métodos para estimar o tempo médio de resposta de um subconjunto de tarefas em paralelo e utilizamos o método proposto por Varki (1999), que apresentou uma melhor acurácia na previsão das métricas de desempenho.

Os principais resultados encontrados nesta dissertação foram:

- Obtivemos uma boa aproximação na validação do modelo de consulta de BI, obtendo uma diferença relativa máxima de 11,5% para o tempo médio de resposta do *job* em relação a um simulador da rede de filas e ao um emulador do sistema, para seis cenários diferentes, onde o máximo aceito na literatura para o tempo de resposta é 30% (Menascé et al., 2004).

- Foram avaliados também os parâmetros da carga de trabalho de BI, onde observamos que o paralelismo *pipeline* teve um ganho maior (no tempo de resposta do *job*) com carga leve, ou seja, antes do sistema saturar. Após a saturação o tempo de resposta do *job* é dominado pelos atrasos de filas e, um aumento no paralelismo *pipeline*, ao invés de aumentar o *speedup*, gera mais contenção

- As previsões de desempenho do modelo utilizando os operador propostos no modelo referência obtiveram um bom acordo, em relação ao simulador e ao experimento real, para cenários com pouco paralelismo, entretanto superestimou as métricas de desempenho ao aumentar o número de tarefas em paralelo, atingindo uma diferença relativa de 68%.

- Após incorporar o método *fork/join*, proposto por Varki (1999), para estimar o tempo de resposta de um sub-conjunto de tarefas em paralelo, as previsões do modelo obtiveram um melhor acordo, ficando o erro relativo máximo do tempo de resposta do *job* abaixo de 15%, para todos os cenários avaliados.

1.5 Organização da Dissertação

Esta dissertação está organizada como a seguir. O Capítulo 2 revisa diversas técnicas de modelagem de desempenho de cargas de trabalho de computações paralelas na Seção 2.1. A metodologia que foi o ponto de partida para a nossa modelagem, a qual chamamos de modelo referência, foi descrita na Seção 2.2. Com o intuito de tornar a leitura mais corrente, as equações utilizadas no nível lógico do modelo referência (Liang e Tripathi, 2000), para estimar a média da distribuição do tempo de resposta do *job*, foram introduzidas no Apêndice A. O Capítulo 3 descreve a estratégia utilizada para modelar o paralelismo *pipeline*. A modelagem do paralelismo *pipeline* entre duas tarefas de consultas simples do banco de dados HP *Neoview*, primeiro estudo de caso desenvolvido, é apresentado no Capítulo 4. O segundo estudo de caso, modelagem do paralelismo de $m \times n$ tarefas dos *jobs* do HOP, é apresentado no Capítulo 5. Uma

vez que o algoritmo da modelagem do paralelismo *pipeline* do HOP possui muitos detalhes, que podem dificultar a compreensão da metodologia, optamos por explicar o algoritmo em alto nível, aplicado a um exemplo, no Capítulo 5 e descrever o algoritmo em mais detalhes nos Apêndices B e C. Na avaliação de desempenho por experimento real do HOP focamos principalmente na avaliação do tempo médio de resposta do *job*, os resultados da validação do tempo médio de resposta das tarefas foram mostrados no Apêndice D. A conclusão desta dissertação, sumarizando as contribuições e os principais resultados encontrados nos dois estudos de caso realizados, são apresentados no Capítulo 6, juntamente como os trabalhos futuros.

Capítulo 2

Referencial Teórico

Este Capítulo apresenta os trabalhos relacionados a avaliação de desempenho de sistemas composto de execução de tarefas em paralelo. A Seção 2.1 apresenta várias metodologias de modelagem de sistemas paralelos que foram estudadas até se chegar na estratégia adotada nesta dissertação. O trabalho sob o qual desenvolvemos nossa solução foi o modelo analítico proposto por Liang e Tripathi (2000), o qual chamamos de modelo referência, descrito na Seção 2.2.

2.1 Métodos para Modelagem de Aplicações Paralelas

Esta seção apresenta várias abordagens para modelagem de desempenho de aplicações paralelas. A Seção 2.1.1 descreve trabalhos que utilizaram modelos paramétricos. Uma metodologia para modelagem do paralelismo *pipeline* através de modelos probabilísticas é mostrada na Seção 2.1.2. A Seção 2.1.3 apresenta técnicas de modelagem baseadas em Redes de Petri. A estrategia de modelagem utilizada nesta dissertação é baseada em modelos hierárquicos, sumarizada na Seção 2.1.4.

2.1.1 Modelos Paramétricos

Alguns modelos de desempenho bem sucedidos são simples modelos paramétricos que estimam a *performance* do sistema com base em poucos parâmetros, que descrevem o paralelismo e a comunicação de uma aplicação paralela. Em Adve e Vernon (2004), os autores propõe usar um grafo de tarefas determinístico, ou seja, sem considerar as distribuições de probabilidade do tempo de resposta das tarefas, para modelar o desempenho de programas paralelos.

Entretanto, a solução proposta não considera a variabilidade causada pela contenção nos recursos e requer uma especificação detalhada do tempo de execução individual de cada tarefa, enquanto que modelos estocásticos (solução utilizada), onde são consideradas as distribuições do tempo de resposta das tarefas, requer o conhecimento apenas de estatísticas básica, como média e variância do tempo de execução das tarefas. O modelo de Adve e Vernon (2004) focou na avaliação de execuções de *jobs* separadamente, não permitindo modelar a execução de múltiplos *jobs* em paralelo, como múltiplas consultas a um banco de dados, avaliados no primeiro estudo de caso.

Embora este método não capture a variância dos atrasos de fila nos recursos físicos, ele captura a contenção do nível lógico (como uma fila para um conjunto de *threads* ou um *locking* de recursos), inserindo um *overhead* no tempo de execução das tarefas. O método para estimar este *overhead* foi baseado no trabalho de Tsuei e Vernon (1990), mas ao invés de utilizarem uma cadeia de *Markov* para representar a combinação de requisição em uma fila lógica utilizaram um modelo de filas com um *delay center* Flow Equivalent Service Center (FESC) (*flow-equivalent service center*), onde o tempo de serviço varia em função da taxa de requisição.

Em Adve e Vernon (1993), os mesmos autores, fizeram um estudo de quando é melhor usar modelos estocásticos ou determinísticos. Observaram que modelos estocásticos foram bem sucedidos em programas com *fork-join*, entretanto em cenários não *fork-join* a premissa do tempo de resposta das tarefas ser exponencialmente distribuído adotado em diversos trabalhos (Thomasian e Bay, 1986; Mak e Lundstrom, 1990; Liang e Tripathi, 2000) leva os modelos estocásticos a não obterem uma boa acurácia (estimativas super-estimados).

2.1.2 Modelos Probabilísticos

Um trabalho com propósito bastante similar ao desenvolvido nesta dissertação foi o método de Navarro et al. (2009), que utilizou modelos probabilísticos para modelar o paralelismo *pipeline* da carga de trabalho de dois *benchmarks* (geradores de carga sintéticos) do *suite* PARSEC (Bienia et al., 2008): (1) o *ferret*, para recuperação de imagens por similaridade, cujo processo se divide em seis estágios; e (2) o *dedup* para compressão de *streams* cujo método (utilizando de-duplicaçao) foi decomposto em um *pipeline* de cinco estágios, que podem executar em paralelo.

Para um modelo fechado, os autores consideraram que em *steady state* [1] cada estágio i (ou seja, onde há um número máximo de *jobs* concorrentes), se comporta como um sistema de filas $M/M/c_i/N/K$ interligadas, onde o tempo entre chegadas e o tempo de serviço são exponenciais, há c_i servidores (*threads*) no estágio i, as filas têm capacidade para N requisições e a população é de K requisições. Em caso de um modelo aberto (onde as requisições chegam com a uma taxa de chegada λ), cada estágio foi modelado como uma rede de filas $M/M/c_i$.

O método recebe como entrada o tempo de serviço de cada estágio e assume que o tempo entre chegada dos estágios são iguais, dado pelo maior tempo de serviço e resolve a rede de filas de cada estágio, calculando os respectivos *throughputs* (λ_i). Em seguida o tempo de resposta é derivado pela Lei de Little (Little, 1961), considerando que o *throughput* do sistema é dado pelo menor *throughput* dos estágios ($\min \lambda_i$).

Este método modela a contenção lógica da fila de *threads* de cada estágio do *pipeline*, capturando a pressão (*throughput*) que um estágio exerce sobre o estágio seguinte. Entretanto não foi possível utilizar este método pois ele assume que as tarefas são independentes, ou seja, assim que uma tarefa termina de ser executada em um estágio i ela vai imediatamente para o estágio $i + 1$, não possibilitando modelar um *fork/join*, onde o *job* só prossegue após um sub-conjunto de tarefas finalizarem.

2.1.3 Redes de Petri

Redes de Petri (*Petri Nets* - PN) vêm sendo amplamente utilizadas para modelagem analítica de aplicações paralelas (Hu e Gorton, 1997). Uma das principais vantagens é que ela suporta uma representação gráfica que auxilia o entendimento do comportamento do sistema. Florin e Natkin (1991) fez uma analogia entre PN e Redes de Fila. Em PN, uma fila é equivalente a um local, representado por círculos. Os clientes em uma fila são modelados por *tokens* em um local. E os centros de serviços correspondem à transições, representado por barras horizontais. Sendo assim, a chegada de clientes (*tokens*) à fila de um centro de serviço é dado por um arco entre uma transição (centro de serviço) e um local (fila). Similarmente, a partida de clientes (*tokens*) é representada por um arco entre uma local e uma transição. Outra diferença é que Redes de Filas modelam apenas a exclusão mútua nos recursos (nível físico) mas não modela as sincronizações, enquanto que PN aborda ambos.

[1] *Steady state*: estado em que o sistema se encontra em equilíbrio de fluxo, ou seja, número de requisições que chegam é igual ao que sai.

Redes de Petri estocásticas (*Petri Nets* - PN) são um sub-conjunto da PN, onde cada transição de disparo t remove um *token* do estado atual e deposita um *token* em um estado i com uma probabilidade p ou em um estado j com probabilidade $1 - p$. O tempo de disparo (*firing*), associado a cada transição, pode ter uma distribuição arbitrária ou exponencial. Neste contexto, um trabalho bastante relevante é o modelo de Ferscha (1992), onde foi descrito como utilizar PN para representar o fluxo de execução de aplicações paralelas, cujas tarefas que podem ter quatro tipos de precedência, mostradas na Figura 2.1. A estratégia de modelagem consiste em percorrer o grafo de estados e substituir um sub-conjunto de tarefas por uma única transição correspondente de acordo com o tipo de precedência. Os métodos de análise das Redes de Petri podem ser agrupados em três grupos (Murata e Shatz, 1989): (1) método da árvore de cobertura ("alcançabilidade"), (2) abordagem da matriz de incidência e equações de estado e (3) por técnicas de redução e decomposição.

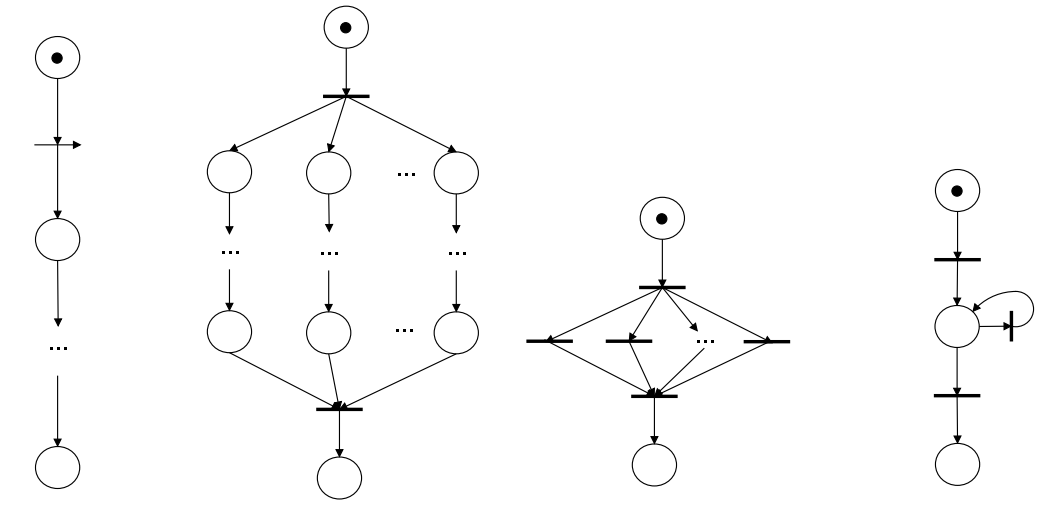


Figura 2.1. Modelo de Redes de Petri de diferentes tipos de precedência.

As Redes de Petri são também utilizadas para modelar a contenção no nível lógico, tais como *locking de um recurso* ou *deadlocks* (Murata e Shatz, 1989). Embora as Redes de Petri possuam ofereçam vários benefícios elas também possuem sérios problemas. O principal problema associado é a complexidade computacional para resolver as Cadeias de Markov subjacentes. Para reduzir o espaço de estados, há vários trabalhos que exploram as simetrias no nível de *software* (tarefas com demandas iguais) e *hardware* (centros de serviço com mesma velocidade), através de Redes de Petri com cores, onde estruturas com comportamento similares (mesma cor) são agrupadas ("dobradas").

2.1.4 Modelos Hierárquicos

Vários trabalhos utilizaram modelos hierárquicos (Thomasian e Bay, 1986; Mak e Lundstrom, 1990; Liang e Tripathi, 2000) para modelagem de carga de trabalho com execução de tarefas em paralelo. O nível mais alto captura a sincronização entre as tarefas (nível lógico) através de um grafo de tarefas, enquanto que o nível mais baixo captura a contenção dos recursos físicos utilizando um modelo de rede de filas. Esta combinação favorece o *trade-off* entre a eficácia (acurácia) e a eficiência (complexidade computacional) para os modelos analíticos. Separados, estes métodos perdem expressividade de prover modelos confiáveis para sistemas paralelos, pois os modelos de rede de filas não capturam os atrasos de sincronizações entre as tarefas, enquanto que os grafos de tarefas não capturam os atrasos de filas da exclusão mútua dos recursos. Quando as duas abordagens são utilizadas em conjunto, os dois tipos de atrasos são cobertos (Jonkers, 1994a).

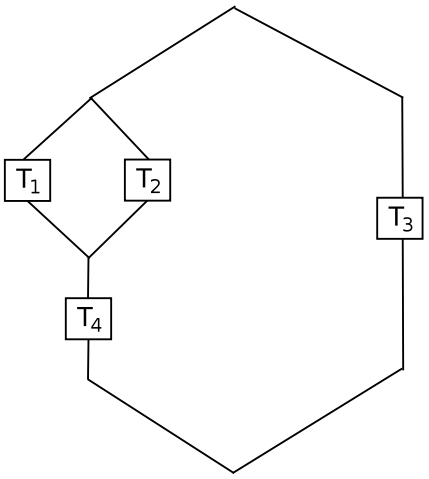
Estes modelos hierárquicos utilizam modelos de redes de filas separáveis, que são um sub-conjunto das redes de filas gerais. As redes de filas separáveis podem ser analisadas em tempo polinomial, enquanto que o pior caso das redes de filas gerais são exponenciais. Para utilização de modelos de redes de filas separáveis é necessário satisfazer seis premissas (Lazowska et al., 1984): (1) a taxa de chegada de um centro é igual a sua taxa de saída (equilíbrio de fluxo); (2) o tempo de serviço não varia com o número de requisições (homogeneidade do tempo de serviço); (3) apenas um passo é dado (chegada ou término da visita a um centro) na mudança de um estado para o outro (base para a cadeia de *Markov*); (4) a taxa de chegada externa não depende do número de requisições ou de suas localizações na rede de filas; (5) a taxa de processamento de um centro varia com o número de requisições no centro, mas não dependem do número de requisições ou de suas localizações na rede de filas; (6) o tempo entre o fim de uma requisição em um centro j e o seu início em um o centro k é independente do tamanho da fila dos centros, o que leva a um aspecto surpreendente: o padrão de roteamento é irrelevante para a *performance* do sistema.

Uma metodologia para modelagem de aplicações *multi-thread* (memória compartilhada) em ambientes multi-camadas, contendo vários servidores (servidor *web*, servidor de aplicações, servidor de banco de dados, servidor de e-mail) foi proposta por (Menascé e Bennani, 2006). Devido ao número limitado de *threads*, as requisições que, quando chegam no sistema, encontram todas as *threads* ocupadas são colocadas em uma fila. Logo, as requisições podem experienciar dois tipos de atraso: (1) atrasos de *software* dado pelo tempo que as requisições devem esperar pela liberação de uma *thread* e (2) atraso de contenção, devido a competição por recursos com as ou-

tras *threads* em execução na mesma máquina. O sistema alvo foi modelado por um modelo hierárquico onde os atrasos de *software* foram modelado por cadeias de *Markov* (Papoulis, 1964), resolvida utilizando a generalização do processo de nascimento e morte (*Generalized Birth-Death* - GBD) (Kendall, 1948) e os atrasos de contenção nos recursos foram modelados por um modelo de rede de filas aberto Menascé et al. (2004), onde as requisições chegam a uma taxa λ. Os autores propuseram um modelo multi-classe (onde requisições à servidores diferentes possuem demandas diferentes) e com ou sem controle de admissão (ou seja, com fila de *threads* finita ou infinita). A principal desvantagem deste modelo em relação ao modelo referência (Liang e Tripathi, 2000), utilizado na modelagem adotada, é que este modelo não considera as relações de precedência presente no fluxo de execução de uma requisição, que são necessárias para modelar o paralelismo *pipeline*.

Um outro trabalho para modelagem de sistemas paralelos que utiliza um modelo hierárquico é a metodologia *Glamis* proposta inicialmente em (Jonkers, 1993) e estendida em (Jonkers, 1994b). Uma das vantagens deste método, que difere de outras abordagens (Heidelberger e Trivedi, 1983; Mak e Lundstrom, 1990; Thomasian e Bay, 1986; Liang e Tripathi, 2000), é que ele não assume que o tempo de resposta das tarefas são exponencialmente distribuídos. O nível físico é modelado por uma rede de filas separáveis e o nível lógico por um grafos de tarefas. Assim como em Mak e Lundstrom (1990) a modelagem se concentrou em grafos série-paralelos. A solução proposta consiste em um algoritmo iterativo relativamente simples, mostrado na Figura 2.2, que toma como entrada o número de tarefas, as demandas das tarefas, a matriz de precedência e as características da arquitetura (número de centros de serviço, políticas de escalonamento). O algoritmo inicia a execução do *job* com um sub-conjunto A de tarefas ativas, que não possuem predecessoras. Executa o MVA (exato ou aproximado) estimando o tempo médio de resposta destas tarefas. O número de classes de *jobs* é igual ao número de tarefas ativas. Obtém o menor tempo de resposta, R_{min}, e finaliza todas as tarefas que terminaram em R_{min}, inserindo-as em um sub-conjunto de tarefas completadas B. Em seguida insere em A todas as tarefas sucessoras das tarefas que acabaram de terminar. Este processo se repete até que todas as tarefas tenham sido executadas. O tempo médio de resposta do *job* é dado pela soma de R_{min}.

O processo para composição do tempo de resposta do *job* é similar a trabalhos anteriores (Adve e Vernon, 1993; Liao et al., 2005), que considera um grafo de tarefas determinístico, onde ambos negligenciam a variância dos *jobs*. Outra contribuição do trabalho de (Jonkers, 1994a) é que ele explora as simetrias do modelo físico (como processadores idênticos, bancos de memória) e do nível lógico (tarefas em paralelo, com mesma demanda e mesmo sucessores e predecessores) agrupando recursos ou tarefas



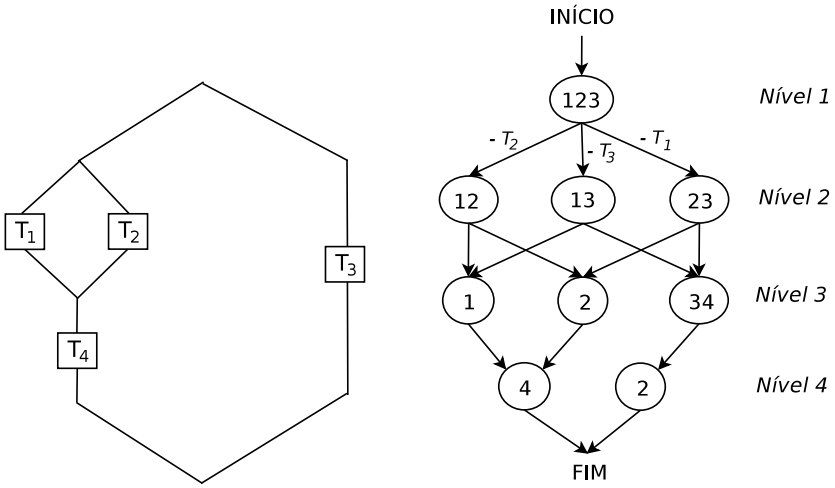
Algoritmo 1: Metodologia *Glamis*

```
1  R_job = 0          /* tempo de resposta do job */
2  C = 4                        /* total de tarefas */
3  A = {T1, T2, T3} /* tarefas em execução */
4  B = { }                      /* tarefas finalizadas */
5  enquanto |B| < C faça
6  |  A = MVA(A), R_min = min(A), R_job+ =
   |  R_min
7  |  para cada a em A faça
8  |  |  se a == R_min então
9  |  |  |  A − {a}
10 |  |  |  B + {a}
11 |  |  senão R_a − R_min
```

Figura 2.2. Exemplo de aplicação da metodologia *Glamis*

e reduzindo a complexidade computacional do modelo. Para isso utilizou o método proposto por (Courtois, 1977), que substitui (agrega) o conjunto de recursos por um *delay center* FESC (*flow-equivalente sevice center*), onde o tempo de serviço varia em função da taxa de requisição do *delay center*, apresentando a derivação das equações para estimar a taxa de requisição. A agregação das tarefas é feita pelo nível de multi-programação (MPL - *multiple programming level*). Assim como (Adve e Vernon, 1993) este método não considera múltiplos *jobs* em paralelo.

Um método de modelagem de desempenho aproximado, mas de alta acurácia, que considera as relações de precedência, necessárias para modelar o paralelismo *pipeline*, foi proposto por Thomasian e Bay (1986). Não é possível utilizar diretamente um algoritmo MVA para modelar a execução paralela de tarefas devido a variação nos estados do sistema. Sendo assim, o método proposto utilizou um modelo hierárquico onde o nível de *software* foi modelado através de cadeias de *Markov* e o nível de *hardware* por um modelo de rede de filas separáveis fechado, utilizado para computar as taxas de transição dos estados. A Figura 2.3 apresenta o mapeamento do grafo acíclico de tarefas (forma mais natural de representar uma computação paralela) para uma cadeia de *Markov*. Cada nó indica quais tarefas estão executando em paralelo no sistema. Uma transição significa o término de uma tarefa (*throughput*) e o disparo da tarefa seguinte (se houver), mudando para o estado correspondente. Uma das restrições deste método é que ele considera um único *job* no sistema, não sendo aplicável para modelar múltiplas consultas de um banco de dados (1º estudo de caso). Este método permite capturar as sincronizações entre as tarefas do paralelismo *pipeline*, entretanto, o espaço de estados cresce exponencialmente com o aumento do número de



(a) Grafo de tarefas (b) Cadeia de Markov

Figura 2.3. Cadeia de *Markov* para um grafo de quatro tarefas

tarefas, impedindo de ser usado para modelar *jobs* com muitas tarefas, como os *jobs* de aplicações do HOP (2º estudo de caso).

Para reduzir a complexidade do modelo, Mak e Lundstrom (1990) propuseram uma solução em tempo polinomial para composição de um grafo de tarefas serial-paralelo, assumindo que o tempo de resposta das tarefas são exponencialmente distribuídos, com base em Salza e Lavenberg (1981). A contenção dos recursos é expressa através de um modelo de rede de filas resolvida usando um MVA aproximado, enquanto que as sincronizações das tarefas é modelado por um grafo de tarefas. Assumiram que o tempo de fila de uma tarefa i causado por uma tarefa j é proporcional a sobreposição das mesmas. Estes fatores de sobreposição foram utilizados para inflacionar o tamanho das filas de um MVA de uma única classe de *job*. Liang e Tripathi (2000) estenderam este trabalho para várias outras classes de grafos de tarefas e proporam um método para calcular os fatores de sobreposição entre tarefas do mesmo *job* e de *jobs* diferentes. O modelo referência permite diversos tipos de relações de precedências, entretanto nenhum de seus operadores capturam as sincronizações de um *pipeline*, não sendo portanto aplicável diretamente à cargas de trabalho que possuam paralelismo *pipeline*. Uma descrição mais detalhada deste modelo, sob a qual nossa solução foi construída, é apresentado na próxima Seção.

2.2 Modelo Referência

Esta Seção apresenta o modelo de desempenho proposto por Liang e Tripathi (2000), baseado no trabalho de Mak (1987), para previsão de métricas de *performance* de aplicações paralelas que contenham restrições de precedência. Este trabalho foi o ponto de partida para a modelagem de desempenho adotada nesta dissertação. Implementamos e estendemos este método para modelar as dependências entre produtores e consumidores inerentes ao paralelismo *pipeline*.

A solução analítica proposta por Liang e Tripathi (2000), como se pode ver na Figura 2.4, consiste em um modelo hierárquico que combina um grafo de tarefas, para capturar os atrasos de sincronização do fluxo de execução das tarefas (nível lógico), com um modelo de rede de filas fechado (onde é assumido que há um número máximo de *jobs* no sistema), que captura a contenção nos recursos (nível físico). O método proposto é computacionalmente eficiente, possui complexidade polinomial, dominado pela resolução da rede de filas (aMVA), que possui complexidade $O(C^2K^3)$ de tempo e espaço, onde C é o número de classes de tarefas (unidade de carga do modelo) e K é o número de centros de serviço da rede de filas.

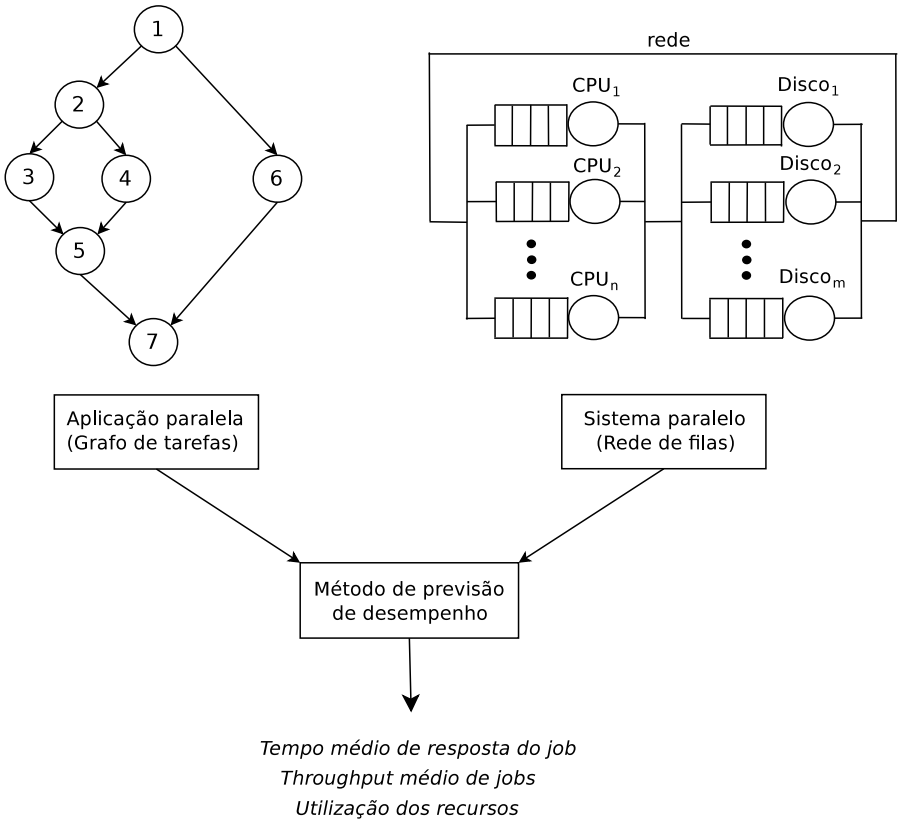


Figura 2.4. Figura adaptada de Liang e Tripathi (2000)

A ordem de execução das tarefas é estabelecida pelas relações de precedências. A Figura 2.5 mostra uma abstração que ilustra o papel do modelo de precedências. O centro de controle do *job* mantém as regras de dependências (de acordo com os tipos de operação que serão apresentados na Seção 2.2.3) e dispara as tarefas apenas quando elas estão "prontas para executar". Se ignorarmos as relações de precedências presentes entre as tarefas, ou seja, se nós ignorarmos a função do centro de controle do *job* o sistema se torna uma rede de filas padrão separável. A estratégia proposta consiste em introduzir tais regras de precedência no algoritmo Análise do Valor Médio aproximado (aMVA - *approximated Mean Value Analysis*) para inflacionar o tamanho médio das filas (contenção nos recursos) através de fatores de sobreposição (computados a partir da árvore de precedência) que representam o paralelismo médio entre tarefas do mesmo *job* (*intra-job*) e entre *jobs* diferentes (*inter-job*).

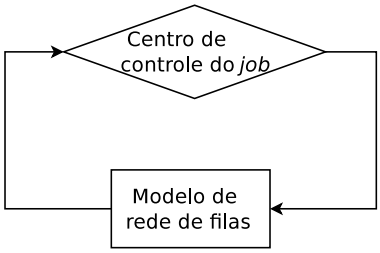


Figura 2.5. Abstração da estratégia de modelagem de Liang e Tripathi (2000).

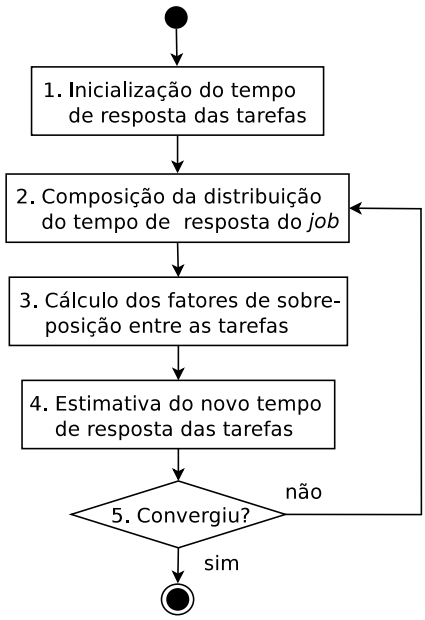


Figura 2.6. Algoritmo iterativo do método de Liang e Tripathi (2000)

Os principais passos do algoritmo são apresentados na Figura 2.6. Toma-se como entrada os parâmetros: número de centros de serviço K, número de tarefas C, demandas de serviço D_{ik}, níveis de multiprogramação N e a árvore de precedência, apresentados na Tabela 2.1. O algoritmo começa inicializando, para cada tarefa, o tempo médio de residência de cada centro de serviço e o tempo médio de resposta (passo 1). Uma forma de fazê-la é considerar que o sistema está sem contenção e inicializar o tempo de residência de cada tarefa i em cada centro k com a demanda correspondente.

$$R_{i,k} = D_{i,k}, \quad \forall\ i,k, \qquad\qquad R_i = \sum\nolimits_{k=1}^{K} D_{i,k}, \quad \forall\ i \tag{2.1}$$

Assim, o tempo médio de resposta de cada tarefa i é inicializado pela soma das demandas da tarefa i em todos os centros k. No passo 2, a distribuição do tempo médio de resposta do *job* assim como os fatores de sobreposição *intra* e *inter-job* são estimados. Estes fatores de sobreposição são introduzidos em uma solução de MVA aproximada no passo 3 para produzir novas estimativas do tempo de resposta médio das tarefas. No passo 4, as novas estimativas do tempo de resposta de cada tarefa i (R'_i) são comparadas com as da iteração anterior (R_i) para verificar se o algoritmo convergiu (dentro do limite de tolerância ϵ, onde foi considerado $\epsilon = 10^{-4}$ nos cenários avaliados):

$$\frac{|R_i - R'_i|}{R_i} < \epsilon, \quad i = \{1, 2, .., C\} \tag{2.2}$$

Em caso de convergência, as novas estimativas do tempo de resposta são usadas para produzir a estimativa final do tempo de resposta do *job* (assim como foi feito no passo 2). O *throughput* de *jobs* e a utilização de recursos são calculados usando a Lei de Little (Little, 1961) aplicada às demandas (parâmetro de entrada) das tarefas. Se o teste de convergência falhar, o algoritmo retorna ao passo do 2 para uma nova iteração.

O método proposto assume as seguintes premissas:

1. As tarefas circulam várias vezes pelos centros de serviço, permitindo aproximar o tempo de resposta para uma exponencial (Salza e Lavenberg, 1981).
2. O atraso de filas é diretamente proporcional a sobreposição entre as tarefas (Mak e Lundstrom, 1990).
3. Consideraram apenas uma classe de *job*, embora o modelo possa ser estendido para múltiplas classes de *jobs*.
4. Ao invés de calcular a convolução entre as distribuições das tarefas (alto custo computacional), assume-se que, se o $CV^2 \leq 1$ aproxima-se para uma *Erlang* (série

[2]O coeficiente de variação (CV) é uma medida de dispersão, dado por $CV = \frac{S}{m}$, onde S é o desvio padrão e m a média. Ele mede o quanto que o desvio padrão está distante da média. O

de exponenciais com médias idênticas), dado por sua baixa variância (Trivedi, 1982), caso contrário, aproxima-se para uma distribuição hiper-exponencial (que possui duas ou mais fases e cada fase tem uma probabilidade p de ocorrer (de forma paralela ou exclusiva).

5. Além das premissas da rede de fila separáveis.

Tabela 2.1. Notação do modelo referência.

Parâmetros de entrada	
T	Árvore com as precedência das tarefas
N	Número de *jobs* concorrentes no sistema (MPL)
C	Número total de classes de tarefas
K	Número total de centros de serviço
$D_{i,k}$	Demanda média de serviço de cada tarefa i em cada centro k
ϵ	Erro mínimo aceito para convergência
I	Número máximos de iterações para convergir
Variáveis do modelo de precedência	
$\Re_0$	Média da distribuição do tempo de resposta do *job*
$L_X(T_i, T_j)$	tempo médio de sobreposições de a tarefa i e j do mesmo *job*
$L_I(T_i, T_j)$	Tempo médio de sobreposição entre as tarefa i e j de *jobs* diferentes
α_{ij}	Fator de sobreposição entre as tarefa i e j do mesmo *job*
β_{ij}	Fator de sobreposição entre as tarefa i e j de *jobs* diferentes
Variáveis do modelo de rede de filas (médias)	
$R_i(\vec{N})$	Tempo de resposta da tarefa i
$R_{ik}(\vec{N})$	Tempo de residência da tarefa i no centro k
$A_{ik}(\vec{N})$	Tamanho da fila no centro k visto pela tarefa i no instante de chegada
$Q_{jk}(\vec{N}\text{-}\vec{1}_i)$	Tempo da fila no centro k visto pela tarefa i com uma tarefa i a menos
$R_{jk}(\vec{N}\text{-}\vec{1}_i)$	Tempo de residência da tarefa j no centro k com uma tarefa i a menos
Métricas de desempenho de saída	
R_0	Tempo médio de resposta do *job*
X_0	*Throughput* de *jobs* médio
U_k	Utilização média do centro k

O grafo de tarefas é uma das formas mais naturais de se representar as precedências de uma aplicação paralela. Um *job* é representado por um conjunto de n nós, onde cada nó corresponde a uma tarefa e os arcos as relações de dependência entre elas. A Figura 2.7 mostra um exemplo de um grafo de tarefas e a árvore de composição correspondente. As folhas da árvore são as tarefas e os nós internos são a composição de dois *jobs* por uma das operações acima. Dois *jobs* são ditos de uma mesma classe se possuem a mesma árvore de composição e a mesma matriz de demandas.

CV é utilizado para caracterizar uma distribuição de probabilidade. Por exemplo, uma distribuição exponencial é um caso especial onde o $CV = 1$. Uma série de exponenciais (Erlang) possui baixa variância e consequentemente um $CV < 1$. Já um *fork* de exponenciais (hiper-exponencial), onde cada distribuição tem uma probabilidade p de ser escolhida, possui uma variância maior ($CV > 1$).

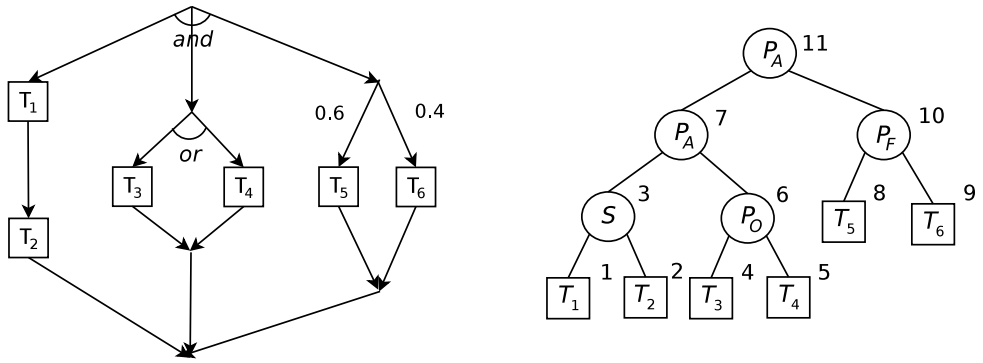


Figura 2.7. Mapeamento de um grafo de tarefas para uma árvore de composição

Os autores consideraram cinco classes de *jobs* denominadas serial (S), *parallel-and* (P_A), *parallel-or* (P_O), *probabilistic-fork* (P_F) e *leaf* (L), ilustrados na Figura 2.8.

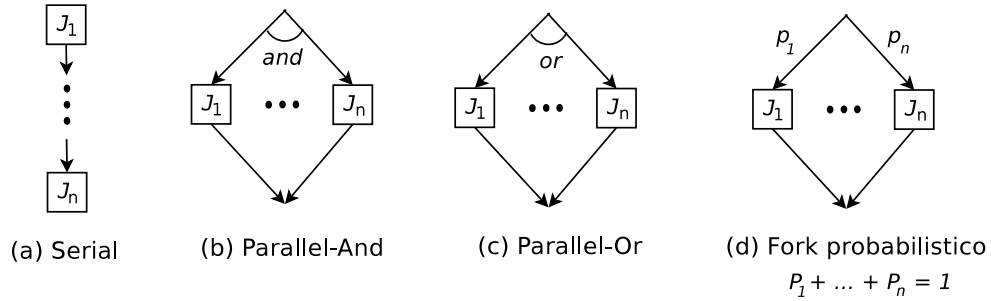


Figura 2.8. Tipos de grafos de tarefas propostos por Liang e Tripathi (2000)

Considere J_1 e J_2 dois *jobs* distintos iniciando em s_1 e s_2 e finalizando em t_1 e t_2 respectivamente. Então um *job* J do tipo k, $k \in \{L, S, P_A, P_O, P_F\}$ é recursivamente definido por:

A seguir será explicado com mais detalhes os passos 2 a 4 do algoritmo do modelo referência, solução sob a qual a modelagem de desempenho desta dissertação foi desenvolvida. Para simplificar a explicação, a ordem da apresentação dos passos será invertida (*bottom-up*). Primeiramente será descrito, na Seção 2.2.1, o passo 4, que produz a estimativa do novo tempo de resposta através do algoritmo aMVA modificado. O cálculo dos fatores de sobreposição *intra-job* e *inter-job*, utilizados no passo 4 para introduzir as restrições de precedência, são realizados no passo 3, mostrado na Seção 2.2.2. Para calcular estes fatores é necessário conhecer o tempo médio de resposta do *job*, estimado pelo passo 2, descrito na Seção 2.2.3. Referimos o leitor ao trabalho original para uma descrição detalhada.

L: uma tarefa simples é um *job* do tipo L

S: o *job* $J = J_1 + J_2$ é um *job* do tipo S com terminais s_1 e t_2 , onde o símbolo + significa a união disjunta de J_1 e J_2. J_2 deve iniciar imediatamente após o fim de J_1, desta forma o tempo de resposta de J é a soma dos tempos de resposta de J_1 e J_2;

P_A: o *job* $J = J_1 \wedge J_2$ é um *job* do tipo P_A. J_1 e J_2 iniciam concorrentemente com J e J termina quanto ambos J_1 e J_2 terminam. Assim, o tempo de resposta de J é o máximo dos tempos de resposta de J_1 e J_2;

P_O: o *job* $J = J_1 \vee J_2$ é um *job* do tipo P_O. J_1 e J_2 iniciam concorrentemente com J e J termina assim que J_1 ou J_2 terminar. O tempo de resposta de J é o mínimo dos tempos de resposta de J_1 e J_2. É assumido neste modelo que após J_1 ou J_2 terminar todas as outras tarefas do outro *job* saem do sistema;

P_F: o *job* $J = J_1 \backslash_f J_2$ é um *job* do tipo P_F. A cada vez que J inicia, ou J_1 ou J_2 iniciam, sendo que J_1 inicia com probabilidade f e J_2 com probabilidade $1 - f$. O tempo de resposta de J será $f \times [tempoderespostadeJ_1] + (1 - f) \times [tempoderespostadeJ_2]$.

2.2.1 Estimativa do Novo Tempo de Resposta da Tarefa

O tempo médio de resposta de cada tarefa i é estimado através do algoritmo da Análise do Valor Médio aproximado (aMVA) (Schweitzer, 1978), estendido para incorporar as restrições de precedência. Para um sistema composto de N *jobs*, representa-se a carga de trabalho pelo vetor $\vec{N} =< N_1, \ldots, N_C >$, onde N_i é o número de tarefas da classe i no sistema. Considera-se que todos os *jobs* possuem uma tarefa de cada classe i, havendo N tarefas de cada classe, ou seja, $N_i = N, \ \forall \ i$.

O cálculo do tempo de resposta de cada tarefa i (R_i) é dado pela soma de seus tempos de residência ($R_{i,k}$), mostrado abaixo na equação 2.3, considerando que os centros de serviço são visitados em série (não importando a ordem), ou em caso de acesso paralelo a probabilidade de se visitar cada centro k está embutida em R_{ik}.

$$R_i = \sum_{k=1}^{K} R_{i,k}, \quad k = \{1, 2, ..., K\} \tag{2.3}$$

O algoritmo MVA exato para redes fechadas separáveis (Kleinrock, 1975) estimam o tempo médio de residência da classe da tarefa i no centro k pela equação[3] abaixo:

$$R_{i,k}(\vec{N}) = \begin{cases} D_{i,k}(1 + A_{i,k}(\vec{N})), & se\ for\ um\ load\ dependent \\ D_{i,k}, & se\ for\ um\ delay\ center \end{cases} \tag{2.4}$$

onde $D_{i,k}$ é a demanda média de serviço da classe de tarefa i no centro k. Pela

[3]Um centro de serviço que não depende da carga (*load independent - LI*) é um centro cuja demanda não varia à medida que aumentamos a taxa de chegada de requisições.

aproximação de Lavenberg e Reiser (1980), $A_{i,k}(\vec{N})$ pode ser estimado como a soma do tamanho das filas quando o sistema tem uma tarefa da classe i a menos ($Q_{j,k}(\vec{N}-\vec{1_i})$), ou seja, $A_{i,k}(\vec{N}) = \sum_{j=1}^{C} Q_{j,k}(\vec{N}-\vec{1_i})$.

Para capturar as restrições de precedência entre as tarefas, os autores propõe inflacionarem o tamanho da fila no instante de chegada, $A_{i,k}(\vec{N})$, pelos fatores de sobreposição (que serão descritos com mais detalhes na próxima Seção), como se pode ver na equação 2.5, onde o primeiro termo ($alpha_{ij}$) captura o atraso de fila experienciada por uma tarefa i devido a contenção com outra tarefa do mesmo *job* (*intra-job*), enquanto que o segundo termo ($beta_{ij}$) captura o atraso devido a tarefas de *jobs* distintos (*inter-job*). Outra premissa associada é que o tempo entre chegadas de tarefas ao sistema segue um processo Poisson (*Poisson Arrivals See Time Averages* - PASTA).

$$A_{i,k}(\vec{N}) \approx \frac{1}{N} \sum_{j=1, j\neq i}^{C} \alpha_{ij} Q_{j,k}(\vec{N}-\vec{1_i}) + \frac{N-1}{N} \sum_{j=1}^{C} \beta_{ij} Q_{j,k}(\vec{N}-\vec{1_i}) \tag{2.5}$$

Aplicando a Lei de Little (Little, 1961) e estendendo a aproximação de Zahorjan (1980) [4] para um sistema multi-classe, $Q_{j,k}(\vec{N}-\vec{1_i})$, pode se escrito em função de $R_{j,k}(\vec{N}-\vec{1_i})$:

$$Q_{j,k}(\vec{N}-\vec{1_i}) = \begin{cases} \frac{N_j}{\sum_{k=1}^{K} R_{j,k}(\vec{N}-\vec{1_i})} \cdot R_{j,k}(\vec{N}-\vec{1_i}), & j \neq i \\ \frac{(N_j-1)}{\sum_{k=1}^{K} R_{j,k}(\vec{N}-\vec{1_i})} \cdot R_{j,k}(\vec{N}-\vec{1_i}), & j = i \end{cases} \tag{2.6}$$

Então dado as estimativas de $R_{j,k}(\vec{N})$, $\alpha_{i,j}$ e $\beta_{i,j}$, o algoritmo computa $R_{j,k}(\vec{N}-\vec{1_i})$ (Equação 2.7), $Q_{j,k}(\vec{N}-\vec{1_i})$ (pela Lei de Little (Little, 1961)), $A_{i,k}(\vec{N})$ (Equação 2.5) e finalmente, novos valores $R_{i,k}(\vec{N})$.

$$R_{j,k}(\vec{N}-\vec{1_i}) \approx \begin{cases} R_{j,k}(\vec{N}) - (\frac{1}{N}\alpha_{j,i} + \frac{N-1}{N}\beta_{ji}) \frac{D_{j,k} R_{i,k}(\vec{N})}{\sum_{k=1}^{K} R_{i,k}(\vec{N})}, & j \neq i \\ R_{j,k}(\vec{N}) - \beta_{ji} \frac{D_{j,k} R_{i,k}(\vec{N})}{\sum_{k=1}^{K} R_{i,k}(\vec{N})}, & j = i \end{cases} \tag{2.7}$$

2.2.2 Cálculo dos Fatores de Sobreposição entre as Tarefas

Nesta Seção apresentamos a derivação dos fatores de sobreposição entre tarefas de um mesmo *job* (*intra-job*) e de *jobs* diferentes (*inter-job*). Neste processo, consideram-se conhecidas além da árvore de precedência, a média dos tempos de resposta de cada tarefa (R_i), dado pela soma das demandas (caso seja a 1ª iteração) ou pela estimativa do tempo de resposta das tarefas na iteração anterior.

[4] Para um sistema de uma única classe, Zahorjan propos estimar $R_k(N-1) \approx R_k(N) - \frac{D_k R_k(N)}{\sum_{k=1}^{K} R_k(N)}$

2.2.2.1 Fator de sobreposição inter-job

Após a árvore de precedência ser percorrida, passo 2 da solução apresentada na Figura 2.6, o algoritmo calcula o fator de sobreposição *inter-job* (β) entre as tarefas T_i e T_j, pertencentes a *jobs* diferentes. O fator de sobreposição do *job* entre T_i e T_j é dado pelo tempo de sobreposição as duas tarefas, $L_I(T_i, T_j)$, em função do tempo de resposta de T_i (R_i):

$$\beta_{i,j} = \frac{L_I(T_i, T_j)}{R_i} \tag{2.8}$$

Para estimar $L_I(T_i, T_j)$, assume-se que a tarefa T_i pode chegar a qualquer ponto do tempo de resposta do *job* ($\Re_0$) a qual T_j pertence (referido como J_0), com igual probabilidade, o tempo da sobreposição *inter-jobs* entre duas tarefas T_i e T_j, $L_I(T_i, T_j)$ é dado por:

$$L_I(T_i, T_j) = Prob(T_i \; encontre \; T_j) \cdot L_I(T_i, J_0) \approx \frac{R_j}{\Re_0} \cdot R_i \tag{2.9}$$

Substituindo a equação 2.9 na equação 2.11 e simplificando a equação (cortando R_i) obtém-se o fator de sobreposição *inter-job*:

$$\beta_{i,j} = \frac{R_j}{\Re_0}, \tag{2.10}$$

2.2.2.2 Fator de sobreposição intra-job

Similarmente, o fator de sobreposição entre duas tarefas T_i e T_j de um mesmo *job* é dado pelo tempo de sobreposição *intra-job* destas tarefas ($L_X(T_i, T_j)$) em função do tempo de resposta de T_i:

$$\alpha_{i,j} = \frac{L_X(T_i, T_j)}{R_i} \tag{2.11}$$

Enquanto a árvore é percorrida para composição do tempo médio de resposta do *job* (passo 2), o algoritmo também estima o tempo de sobreposição entre cada par de tarefas i e j do mesmo *job*($L_X(T_i, T_j)$). Quando um nó interno é visitado, o tipo associado do operador determina a sobreposição entre as tarefas na sub-árvore a esquerda e à direita. No caso de um operador serial, para qualquer par de tarefas T_i e T_j , em diferentes sub-árvore, $\alpha_{i,j} = \alpha_{j,i} = 0$, pois ambas as tarefas nunca irão ser executada concorrentemente. Vamos representar o nó interno por J e suas sub-árvore por J_1 e J_2. J_1 e J_2 podem ser expressos como uma série onde cada J_{1i} representa um

nó interno (de qualquer classe de *job*, exceto serial) ou uma tarefa:

$$\begin{aligned} J_1 &= J_{11} + J_{12} + \cdots + J_{1n} \\ J_2 &= J_{21} + J_{22} + \cdots + J_{2m} \end{aligned} \tag{2.12}$$

Cada componente desta computação é estimado baseado na média da distribuição do tempo de resposta de folhas individuais ou nós internos, os quais já foram visitados na travessia da árvore. Seja $L_X(T_i, T_j)$ o tempo de sobreposição *intra-job* entre as tarefas, para $T_i \in J_{1i}$ $(1 \leq i \leq n)$ e $t_j \in J_{2j}$ $(1 \leq j \leq m)$, assumindo que J_{1i} e J_{2j} são independentes e igualmente distribuídos (i.i.d), L_X é então definido como:

$$\begin{aligned} L_X(T_i, T_j) &= Prob(T_i \text{ e } T_j \textit{ estarem no sistema} | J_{1i} \textit{ e } J_{2j} \textit{ estao no sistema}) \cdot L_X(J_{1i}, J_{2j}) \\ &\approx Prob(T_i|J_{1i}) Prob(T_j|J_{2j}) \cdot L_X(J_{1i}, J_{2j}) \\ &= \frac{R_i}{\Re_{1i}} \frac{R_j}{\Re_{2j}} \cdot L_X(J_{1i} J_{2j}) \end{aligned} \tag{2.13}$$

onde R_i, $\Re_{1i}$, R_j e $\Re_{2j}$ são tempo de resposta médios da tarefa e nó interno correspondente, estimados a medida que a árvore é percorrida (veja Seção 2.2.3). Em outras palavras, a sobreposição é dada pela probabilidade de ambas as tarefas, T_i e T_j, estarem no sistema dado que J_{1i} e J_2 estão no sistema, vêzes o tempo de sobreposição dos *jobs* J_{1i} e J_{2j}. Uma vez que J_{1i} e J_{2j} rodam independentemente, assume-se que $Prob(T_i|J_{1i})$ e $Prob(T_j|J_{2j})$ são independentes. $L_X(J_{1i}, J_{2j})$ é computado para todo T_i e T_j como:

$$L_X(J_{1i}, J_{2j}) = L_X(J_{1i}, J^*_{2j}) - L_X(J_{1i}, J^*_{2j-1}) \tag{2.14}$$

onde $J^*_{1i} = J_{11} + \cdots\cdots\cdots + J_{1i} = J^*_{1i-1} + J_{1i}$ e $J^*_{2j} = J_{21} + \cdots\cdots\cdots + J_{2j} = J^*_{2j-1} + J_{2j}$.

Tomando Y_1, Y_2, Y_3 e Y_4 como variáveis aleatóriaa que representam os tempos de resposta de J^*_{1i-1}, J_{1i}, J^*_{2j-1} e J_{2j}, então $L_X(J_{1i}, J_{2j})$ pode ser escrito como:

$$\begin{aligned} L_X(J_{1i}, J_{2j}) = E[min(Y_1 + Y_2, Y_3 + Y_4)] \\ -E[min(Y_1, Y_3 + Y_4)] \\ -E[min(Y_1 + Y_2, Y_3)] \\ +E[min(Y_1, Y_3)] \end{aligned} \tag{2.15}$$

A Figura 2.9 apresenta um exemplo (adaptado de Liang e Tripathi (2000)) do cálculo de LX entre duas tarefas T_2 e T_4, denotadas por suas variáveis aleatórias, Y_2 e Y_4.

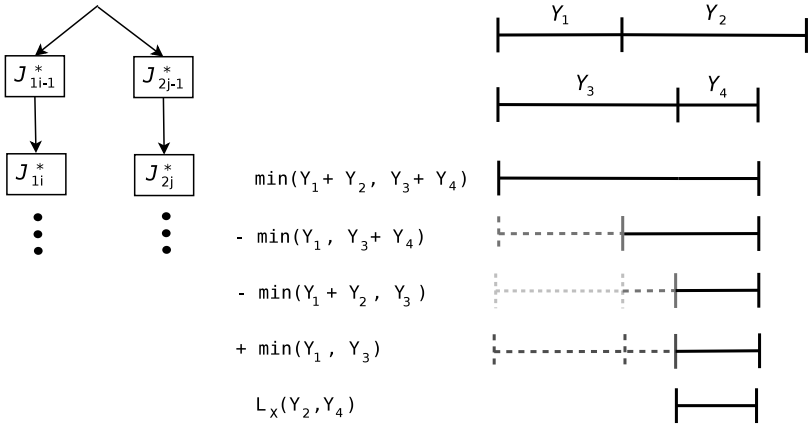


Figura 2.9. Ilustração do cálculo do tempo de sobreposição *intra-job* (LX)

Primeiro toma-se a sobreposição das duas séries J_{1i} e J_{2j}, até atingir, respectivamente, T_2 e T_4, dado pelo $\min(Y_1+Y_2, Y_3+Y_4)$. Em seguida, remove o que tem para tŕas de T_2 e T_4, calculados por $\min(Y_1, Y_3+Y_4)$ e $min(Y_1+Y_2, Y_3)$, respectivamente. Entretanto, a sobreposição entre o que tem para trás de T_2 e T_4 é removido duas vezes, então adiciona-se de volta esta sobreposição, dado por $min(Y_1, Y_3)$.

2.2.3 Composição da Distribuição do Tempo de Resposta do Job

A distribuição do tempo de resposta do *job* é estimada dado a árvore de precedência e (estimativas) da distribuição do tempo de resposta da cada tarefa. Uma vez que a distribuição do tempo de resposta de uma rede de filas fechada é difícil de derivar, o modelo se baseia na premissa de que as tarefas são *coarse grained* [5] e, então seus tempos de resposta são exponencialmente distribuídos (Salza e Lavenberg, 1981).

A base da recursão consiste em assumir que os *jobs*-folha do tipo L (tarefa) possuem tempo de serviço seguindo uma distribuição exponencial com média $R_i, i = 1, ..., C$. Após isso, basta percorrer os nós da árvore em profundidade, ou seja, da esquerda para a direta e de baixo para cima, subindo das folhas (tarefas) até a raiz (*job*), computando a média e a variância do tempo de resposta de cada nó a partir da média e variância dos tempos de resposta de seus filhos de acordo com a classe do *job* do nó interno.

Uma vez que a solução exata para calcular a distribuição do *job* J é computacionalmente intensa (Balakrishnan, 1994), requer calcular a convolução da distribuição de J_1 por J_2, os autores assumem que J_1 é uma Erlang se seu $CV \leq 1$, ou, caso contrário,

[5] *Coarse-grained*: considera que as tarefas têm um tempo de resposta considerável e circulam várias vezes pelo centro de serviço (assumindo que o mesmo tem disciplina *processor sharing*).

uma hiper-exponencial (Trivedi, 1982). Então, em cada nó interno, o algoritmo estima a média e variância (e CV) do tempo de resposta, usando a distribuição do tempo de resposta (ou seja suas médias e variâncias) dos filhos a esquerda e a direita, fazendo uso das equações apresentadas no Apêndice A.

Para cada nó interno da árvore de composição, é possível obter os parâmetros das distribuições Erlang ou Hiper-exponencial a partir de sua média (m) e variância (σ^2) do tempo de resposta. No caso de uma Erlang temos: $r = \frac{m^2}{\sigma^2}$, $\lambda = \frac{m}{\sigma^2}$. No caso de uma hiper-exponencial tem-se: $\lambda_1 = \frac{1}{m} + \frac{1}{m}\sqrt{\frac{\sigma^2-m^2}{\sigma^2+m^2}}$, $\lambda_2 = \frac{1}{m} - \frac{1}{m}\sqrt{\frac{\sigma^2-m^2}{\sigma^2+m^2}}$ e $p = \frac{\lambda_1(\lambda 2m-1)}{\lambda_2-\lambda_1}, \sigma^2 \geq m^2$.

Liang e Tripathi (2000) focaram principalmente em como introduzir as restrições de precedência expressas em cinco tipos de grafos, no algoritmo iterativo AMVA. Portanto, eles assumem que o modelo de árvore de precedência é dado como entrada. Entretanto, nenhum dos tipos de operadores (nós internos da árvore de precedência) capturam explicitamente a dinâmica das sincronizações entre tarefas de um *job* com paralelismo *pipeline*.

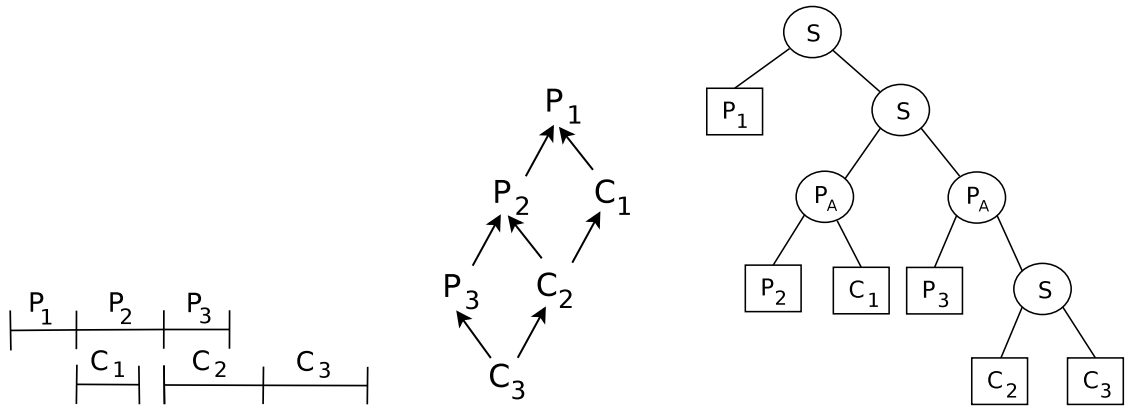
Capítulo 3

Estratégia de Modelagem

Este capítulo apresenta a metodologia de modelagem de desempenho de sistemas que possuam um paralelismo *pipeline* proposta nesta dissertação. O meu papel principal neste projeto foi no desenvolvimento deste modelo, com o auxílio do grupo.

A estratégia do modelo referência consiste em representar as relações de precedência de uma aplicação paralela através de um grafo de tarefas. O grafo de tarefas é uma das formas mais naturais de representar a aplicação paralela. Os nós são as tarefas e as arestas indicam suas dependências. Este grafo é construído a partir de um conjunto de operadores que modelam diferentes tipos de precedências (descritas na Seção 2.2). Como o grafo não é uma estrutura muito prática, devido a presença de *loops*, o modelo referência mapeia o grafo de tarefas para uma árvore binária de precedência, onde as folhas são as tarefas e os nós internos as relações de precedências entre elas. Entretanto, nenhum dos operadores propostos no modelo referência capturam explícitamente as relações de precedência de um paralelismo *pipeline*.

Inspirado no modelo referência Liang e Tripathi (2000), a abordagem utilizada para modelar o paralelismo *pipeline* consiste em construir uma árvore de precedência serial-paralela para capturar o paralelismo médio e as sincronizações entre as tarefas produtoras e a consumidoras. Dados alguns parâmetros que caracterizam a carga de trabalho, o tempo de resposta das tarefas (estimado pela rede de filas) e um conjunto de regras de precedência é possível representar o fluxo de execução do *job* em uma linha de tempo (*timeline*). O método para construção da árvore percorre esta linha de tempo buscando identificar eventos de término de tarefas produtoras e consumidoras e mapeá-los para a árvore de precedência. A árvore de precedência foi construída utilizando os operadores propostos no modelo referência, mais específicamente os operadores serial (S), *paralell-and* (P_A).



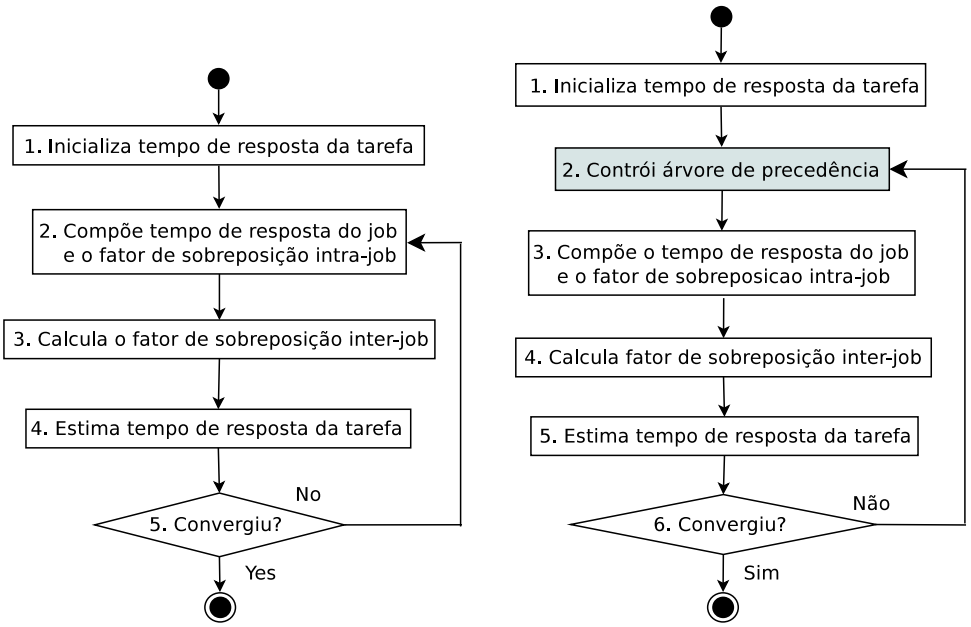
(a) Linha de tempo (b) Grafo tarefa (c) Arvore de precedência

Figura 3.1. Exemplo de modelagem do paralelismo *pipeline*.

A seguir será explicado um exemplo ilustrativo. Seja, um *job* J composto por uma tarefa produtora P e uma consumidora C que se comunicam em um paralelismo *pipeline*. A Figura 3.1(a) apresenta a linha de tempo de um cenário possível do fluxo de execução do job J. Considere que estas tarefas foram divididas em três sub-tarefas para capturar as inter-dependências entre estas tarefas. Típicamente em um paralelismo pipeline cada sub-tarefa i, produtora ou consumidora, deve aguardar a sub-tarefa anterior $i-1$ terminar para poder iniciar (relação serial) e cada sub-tarefa consumidora P_i deve aguardar pela a sub-tarefa produtora corresondente, C_i, para poder iniciar (seriais). Entretanto sub-tarefas produtoras P_i e consumidoras C_j, onde $j > i$, podem executar concorrentemente (relação paralela).

O grafo de tarefas, mostrado na Figura 3.1(b), representa as dependências entre as tarefas. Pode se ver, por exemplo que o produtor P_2 depende de P_1 e que o consumidor C_2 depende de P_1 e de C_1. Assim, como no modelo referência o grafo de tarefas foi mapeado para uma árvore binária de precedência, apresentada na Figura 3.1(c). Considere as sub-tarefas individuais com as unidades de cargas do modelo.

A estratégia da solução adotada consiste em desenvolver um algoritmo, que dado alguns parâmetros da carga de trabalho e as estimativas do tempo médio de resposta das tarefas, constrói automáticamente a árvore de precedências serial-paralela, de modo a capturar as sincronizações *intra-query* inerentes ao paralelismo do *pipeline*. Se não houver nenhum bloqueio, a árvore é dado por um nó paralelo (*parallel-and*) com dois galhos, um com todas as sub-tarefas produtoras (*disk*) em série e outro com todas as sub-tarefas consumidoras (*root*) em série.



(a) Solução anterior para *jobs* paralelos (Liang e Tripathi, 2000)

(b) Nossa extensão para o paralelismo pipeline

Figura 3.2. Principais passos do modelo de desempenho de aplicações paralelas

Entretanto, como podemos observar neste exemplo ilustrativo, a ordem de execução das tarefas de um paralelismo *pipeline* não são estáticas. Isto implica que a forma exata da árvore, determinada quando cada tarefa bloqueia, pode variar de acrodo com a (estimativa) média do tempo de resposta de cada sub-tarefa. Como o algoritmo é iterativo, a cada iteração é feito uma nova estimativa do tempo médio de resposta das tarefas, até convergir. Sendo assim, não foi possível passar a árvore de precedência como um parâmetro de entrada, como é feito no modelo referência, apresentado na Figura 3.2(a). Para capturar esta dinâmica das sincronizações do *pipeline*, estendemos o modelo refereência, como mostrado na Figura 3.2(b), inserindo o algoritmo para construção da árvore como um passo extra em sua solução analítica (passo 3), que reconstrói a árvore de precedência a cada iteração, utilizando o tempo médio de resposta das sub-tarefas estimados na iteração anterior.

Sendo assim, a metodologia pode ser dividida em três passos: (1) primeiramente é feito uma modelagem descritiva da arquitetura do sistema, onde é feito um mapeamento dos recursos físicos da arquitetura do sistema (CPU, disco, etc.) para um modelo de rede de filas, onde cada recurso é representado por um centro de serviço com uma fila associada; (2) em seguida é feito a decomposição da carga de trabalho, onde é feito um mapeamento das porções do código de uma aplicação paralela para a unidade de carga, que corresponde às tarefas do modelo de filas, define-se quais destas tarefas são produtoras e consumidoras e identifica as regras de precedências entre estas tarefas; (3) por fim, deve ser desenvolvivo um algoritmo para gerar automática a árvore de precedência a cada iteração do algoritmo.

Capítulo 4

Modelagem do Paralelismo Pipeline entre Duas Tarefas

Este Capítulo apresenta a modelagem de desempenho realizada para a carga de trabalho composta por consultas simples de *Business Intelligence* (BI), caracterizadas pela presença de um paralelismo *pipeline* entre dois operadores (tarefas). Primeiramente serão apresentadas, na Seção 4.1, as características da arquitetura do banco de dados distribuído *HP Neoview* (nível físico) e o fluxo de execução de uma consulta simples de BI (nível lógico), destacando os aspectos a serem considerados durante a modelagem. A explicação da estratégia da modelagem de desempenho adotada é feita na Seção 4.2. As ferramentas utilizadas na validação do modelo analítico são apresentadas na Seção 4.3. A Seção 4.4 apresenta a sumarização dos resultados encontrados na validação do modelo analítico, assim como uma análise da influência dos principais parâmetros do modelo.

4.1 Descrição do Sistema

A arquitetura do sistema de banco de dados avaliada é baseada em um sistema real chamado *HP-Neoview* (Winter, 2009). O *HP-Neoview* é um sistema de banco de dados distribuído compreendendo um grande número de *hosts* em processamento, que se comunicam através de um barramento de rede de alta velocidade. Cada nó possui dois disco anexados. Uma tabela de um banco de dados pode ser armazenada ao longo de vários *hosts* ou armazenada inteiramente em um único disco.

Uma consulta de banco de dados pode ser representada por uma árvore de operadores onde os operadores folha lêem os dados dos discos, fazem algum processamento (como selecionar sub-conjuntos, filtros) e passa as tuplas resultantes para o operador pai. O fluxo de dados percorre a árvore até o operador raiz, o qual envia os resultados finais para a aplicação do cliente. Em um sistema real, o ciclo de vida de uma consulta inclui várias fases além de sua execução (tais como compilação, otimização). Para o propósito de validação, focamos apenas na fase de execução e ignoramos o impacto das outras fases na *performance* do sistema. Em particular, nós deixamos a rede que conecta o banco de dados a aplicação do cliente fora do escopo atual.

Foram modeladas dois tipos de consultas, chamadas OLTP e BI simples, que compreendem duas tarefas, chamadas *disk* e *root*. A tarefa *disk* (produtora) faz uma leitura de disco, enquanto que a tarefa *root* (consumidora) não acessa o disco. Cada tarefa divide-se em F sub-tarefas, que se comunicam através de um *buffer*, com disciplina FCFS (*first come first served*), de tamanho m e capacidade M (parâmetro fixo de configuração). O operador *root* só pode começar depois que o *disk* concluir a leitura dos dados no disco físico (demanda de disco). Sendo assim, a primeira sub-tarefa do *disk* consome toda demanda de I/O de disco da tarefa, o mesmo vale para a demanda da CPU da tarefa *root*. Quando o produtor termina de processar uma sub-tarefa, verifica, se o *buffer* não estiver cheio ($m < M$), então insere na fila, senão bloqueia. O bloqueio do produtor é causado pelo consumidor, que está processando mais lentamente que o produtor, gerando uma retenção no processamento do produtor (*back-pressure*). Quando o consumidor termina de processar uma sub-tarefa, verifica, se a fila do consumidor não estiver vazia ($m > 0$) retira da fila e inicia processamento, senão, bloqueia.

4.2 Modelagem Analítica

Esta seção apresenta a aplicação da metodologia proposta no Capítulo **??** na carga de trabalho do banco de dados HP *Neoview*.

A metodologia de modelagem proposta nesta dissertação para análise de desempenho de sistemas que possuam um paralelismo *pipeline* entre duas tarefas (um produtor e um consumidor) é apresentado nesta Seção, onde foi feito um estudo de caso da modelagem de consultas simples de *Business Intelligence* (BI). O meu papel principal no primeiro estudo de caso foi no desenvolvimento deste modelo. A Seção **??** apresenta a estratégia de modelagem adotada, que consiste em inserir um passo na solução analítica proposta no modelo referência (descrito na Seção 2.2), para construção da

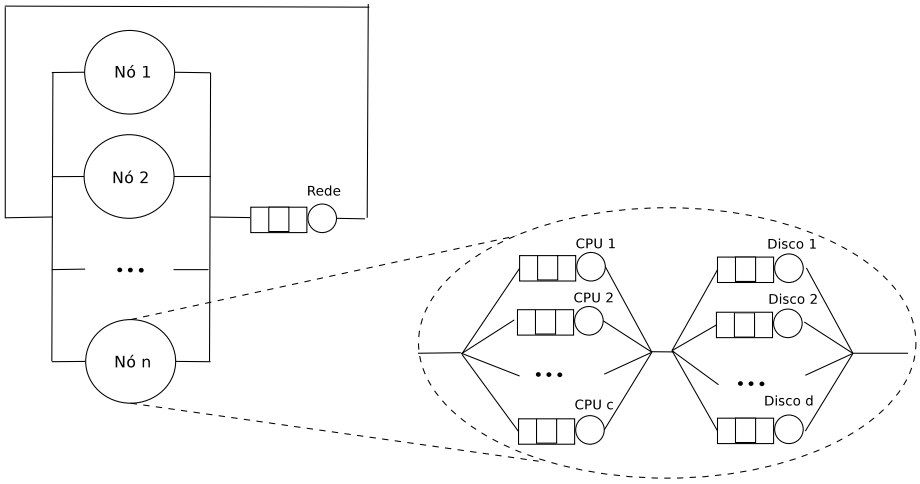


Figura 4.1. Diagrama da rede de filas do sistema alvo.

árvore de precedências, de modo a capturar as sincronizações *intra-query* inerentes ao paralelismo do *pipeline*. Em seguida, apresentamos, na Seção 4.2.3, um algoritmo para construir automaticamente, esta árvore de precedência.

4.2.1 Modelo Descritivo da Arquitetura do Sistema

O sistema alvo é modelado por uma rede de filas fechada, como descrita na Figura 4.1. Cada nó é modelado por uma coleção de centros de serviço com filas infinitas. O barramento de rede é modelado também como um centro com fila infinita que transmite os dados produzidos pelas sub-tarefas vindo de todos os nós. A entrada do modelo analítico é o número total de centros de serviço K, o nível de multiprogramação N (em número de consultas concorrentes) e a matriz de demanda D_{ik} especificando o tempo médio de serviço de cada sub-tarefa i em cada centro de serviço k. K é derivado do número de nós no sistema (n), e do número de CPUs (c) e discos (d) por nó. $D_{i,k}$ é computado dado as demandas de cada fragmento (produtor e consumidor) para recursos do sistema (CPU, disco e rede) e a especificação das probabilidades de roteamento pelos recursos do sistema (nó, CPU e disco).

4.2.2 Decomposição da Carga de Trabalho

No intuito de capturar o paralelismo e os atrasos de sincronização entre produtor e consumidor, nós consideramos que o tempo total de processamento de cada tarefa é dividido em F estágios, aqui referido como *sub-tarefas*, para capturar as interdependências do *pipeline*. Cada sub-tarefa captura uma porção do trabalho total realizado por cada tarefa. Por exemplo, cada sub-tarefa produtora representa o tempo de

processamento correspondente de recuperar, processar e enviar um certo montante de tuplas resultantes para o consumidor. O processamento destas tuplas pelo consumidor é representado pela sub-tarefa do consumidor correspondente. Em outras palavras, uma vez que o produtor termine de processar a i-ésima sub-tarefa, p_i, ele envia as tuplas resultantes para a consumidor, que será processado pelo consumidor correspondente, c_i. Concorrentemente, o produtor continua processando o próximo (i+1)-ésimo sub-tarefa. F pode também ser visto como o número de mensagens de saída enviadas pelo produtor para o consumidor. Note que o tempo de processamento de uma tarefa pode não ser uniformemente dividido através das sub-tarefas, dado que uma diferente quantidade de trabalho pode ser realizada em cada estágio. Por exemplo, o produtor pode ter de recuperar todos os dados do disco para a memória primeiro antes de processá-los e enviar as tuplas resultantes para o consumidor. Então a primeira sub-tarefa do produtor pode requerer mais tempo de processamento, uma vez que ela inclui todas as operações de disco (juntamente com seu tempo de CPU e rede). Além disso, cada sub-tarefa poder experienciar atrasos de filas (contenções) diferentes.

Nós também modelamos o *buffer* de comunicação entre o produtor e consumidor como uma fila lógica finita. Cada posição da fila contém um *token* correspondente a uma sub-tarefa. Quando o produtor terminar de processar a sub-tarefa i, ele insere o *token* t_i na fila[1]. Assim que o consumidor estiver pronto para processar, ele então remove t_i da fila. Então o tamanho máximo da fila, M, especifica o máximo número de sub-tarefas esperando para ser processados pelo consumidor. Se o produtor estiver executando mais rápido que o consumidor, a fila lógica irá aumentar. Quando ela atingir o tamanho máximo M, o produtor para de processar e fica bloqueado até que um *slot* seja liberado na fila (efeito do *back-pressure*). Similarmente, o consumidor pode bloquear, se, quando ele terminar de processar a sub-tarefa, o *token* correspondente à próxima sub-tarefa não estiver na fila (ou seja, se a fila estiver vazia). Então, de certo modo, M captura a pressão que um consumidor impõe no produtor assim como o grau de serialização/paralelismo entre eles.

Consideramos que a unidade de carga são as sub-tarefas. Em outras palavras, substituímos no algoritmo iterativo original, as tarefas por sub-tarefas e *jobs* por *queries*. Na inicialização do modelo, a demanda média de serviço gasto em cada centro por cada tarefa foi dividido em F componentes, um para cada sub-tarefa. A demanda de disco do produtor é toda consumida pela primeira sub-tarefa.

[1] O *token* $t_i - th$ representa a mensagem de saída enviada pela sub-tarefa produtora esperando ser processada pelo consumidor

4.2.3 Construção da Árvore de Precedências

A seguir, apresentamos o algoritmo para gerar automaticamente a árvore de precedência dado as estimativas (atuais) do tempo médio dos sub-tarefas. Este algoritmo correspondente ao passo introduzido na nova solução.

A árvore de precedência é construída pela criação de múltiplas sub-árvores com um nó paralelo (P_A) na raiz que são interconectadas através de nós internos seriais (S). Cada sub-árvore com um nó P_A na raiz corresponde a uma fase paralela da execução em *pipeline*. Um nó S é adicionado para representar o final de uma fase causada pelo bloqueio de uma tarefa. Nós S são também introduzidos para capturar a serialização no processamento da primeira a última sub-tarefa de uma tarefa. A Figura 4.3 mostra um exemplo da árvore para um cenário com $F = 8$ e $M = 2$, que foi construída a partir da linha de tempo da execução da *query* apresentada na Figura 4.2. Os intervalos em branco representam bloqueios. Como em Liang e Tripathi (2000), utilizou-se uma árvore binária, entretanto, para simplificar a representação, mostramos apenas os nós S conectando à sub-árvore com nó P na raiz, omitindo aqueles que conectam sub-tarefas da mesmo tarefa executando em série, que são representados por uma cadeia.

Dado F e M, o *pipeline* entre produtor e consumidor é modelado como uma

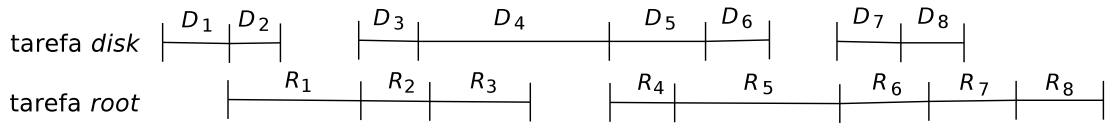


Figura 4.2. Exemplo da linha de tempo de uma *query* simples de BI.

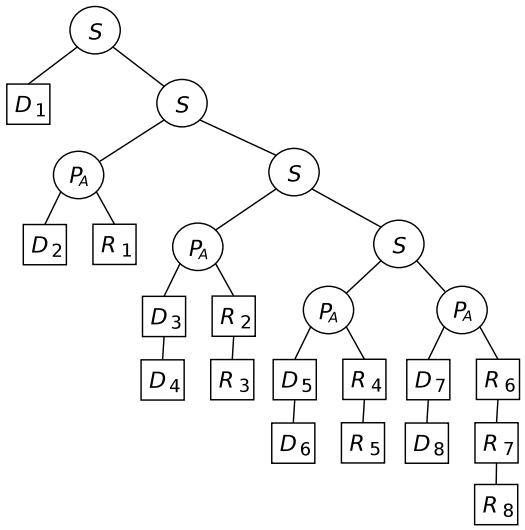


Figura 4.3. Árvore de precedência referente a linha de tempo da Figura 4.2.

Tabela 4.1. Parâmetros do modelo e variáveis da construção da árvore.

Notação	Descrição das variáveis utilizadas no algoritmo
F	Número de sub-tarefas por tarefa
M	Tamanho máximo da fila lógica entre produtor e consumidor
R_{p_i}	Tempo médio de resposta do i-ésimo tarefa produtor
R_{c_j}	Tempo médio de resposta do j-ésimo tarefa consumidor
R_p^{acc}	Tempo médio de resposta acumulado da sub-árvore do produtor
R_c^{acc}	Tempo médio de resposta acumulado da sub-árvore do consumidor

árvore de precedência, usando operadores serial (S) e *parallel-and* (P_A) introduzidos em Liang e Tripathi (2000). Ao invés de tarefas cada folha aqui representa uma sub-tarefa produtora(p_i) ou consumidora (c_i) e a raiz representa um *job*. A ideia principal é que sub-tarefas da mesmo tarefa devem executar em série (S), enquanto que sub-tarefas de diferentes tarefas podem rodar em paralelo (P_A) desde que elas não causem bloqueio. Se uma tarefa bloquear, ela deve esperar que a outra tarefa termine de processar uma sub-tarefa atual. Quando isso ocorrer, ambas as tarefas podem continuar a execução, em paralelo, até o próximo bloqueio ocorrer ou o processamento do produtor terminar. Então, podemos ver a execução do *pipeline* como uma série de fases paralelas, cada fase contendo múltiplas sub-tarefas produtoras e consumidoras executando em paralelo.

A construção da árvore de precedência é feita pelo procedimento Constroi-Arvore(), descrito no Algoritmo 27. Este algoritmo toma como entrada os valores de F, M e o tempo médio de resposta de cada sub-tarefa produtora e consumidora (estimados pelo modelo de rede filas, na última iteração), sumarizados na Tabela 3.1 (juntamente com outras variáveis utilizadas pelo algoritmo). O algoritmo constrói a árvore da raiz para as folhas. As variáveis i e j indicam, respectivamente, o índice da próxima sub-tarefa produtora e consumidora, a ser adicionada na árvore.

O primeiro passo é inicializar a árvore com um nó S na raiz e uma folha à esquerda referente a primeira sub-tarefa produtor p_1 (linha 1). Cada iteração do *loop* "equanto"(linhas 5-23) processa a execução de uma fase do *pipeline*, representada por uma sub-árvore com um nó P_A na raiz, contendo a série de sub-tarefas produtoras e consumidoras em paralelo nos galhos à esquerda e direita, respectivamente (linhas 11-23). A identificação de qual a próxima sub-tarefa a ser adicionada na sub-árvore da fase é dado por qual ramo da sub-árvore termina primeiro, ou seja: se $R_p^{acc} \leq R_c^{acc}$, a próxima sub-tarefa produtora é adicionada na cadeia a esquerda da sub-árvore, desde que ele não tenha sido bloqueado ($m < M$) e que haja ao menos mais sub-tarefa produtora ($i < F$) (linhas 12-16); se $R_c^{acc} \leq R_p^{acc}$, a próxima sub-tarefa consumidora é adicionado a cadeia à direita da sub-árvore enquanto o sub-tarefa do produtor correspondente não

Algoritmo 2: Constroi-Arvore(F, M, R_{p_i}, R_{c_j} $(1 \leq i \leq F, 1 \leq j \leq F)$

```
1  Inicie a árvore com um nó "S"na raiz e uma sub-tarefa p_1 à esquerda
2  i = 2                    /* índice da próxima sub-tarefa produtora a ser inserida */
3  j = 1                    /* índice da próxima sub-tarefa consumidora a ser inserida */
4  m = 1                    /* tamanho atual da fila lógica */
5  enquanto i ≤ F faça
6  |  R_p^acc = R_{p_i}     /* inicializa tempo cumulativo do produtor */
7  |  R_c^acc = R_{c_j}     /* inicializa tempo cumulativo do consumidor */
8  |  Adicione um nó "S"à direita do nó "S"mais a direita
9  |  Adicione um nó "P_A"à esquerda do novo nó "S"
10 |  Adicione p_i e c_j como filhos à esquerda e direita do novo nó "P_A"
11 |  repita                 /* repita até a fila encher ou ocorrer um bloqueio */
12 |  |  se R_p^acc ≤ R_c^acc e m < M e i < F então        Adiciona produtor
13 |  |  |  i++
14 |  |  |  Adicione nó p_i no final da cadeia do produtor à esquerda
15 |  |  |  R_p^acc += R_{p_i}     /* acumula tempo de resposta do produtor */
16 |  |  |  m++
17 |  |  se R_c^acc <= R_p^acc e i > j + 1 então          Adiciona consumidor
18 |  |  |  j++
19 |  |  |  Anexe o nó c_j no final da cadeia do consumidor a direita
20 |  |  |  R_c^acc += R_{c_j}     /* acumula tempo de resposta do consumidor */
21 |  |  |  m−−
22 |  até m == M e (i == j + 1 e R_p^acc > R_c^acc) ou i == F;
23 |  i++, j++
24 enquanto j ≤ F faça
25 |  Adiciona nó c_j no final da cadeia do consumidor à direita
26 |  j++
27 retorna tree
```

terminar $(i > j + 1)$ (linhas 17-21).

Uma fase termina quando uma das tarefas bloqueia, ou seja, ou $m = M$ (ou seja, a fila está cheia), o consumidor está pronto para processar o próxima sub-tarefa ($R_p^{acc} > R_c^{acc}$), mas ele ainda não foi processado pelo produtor ($i = j + 1$) (ou seja, a fila está vazia), ou não há mais sub-tarefas produtores para serem processados ($i = F$) (linhas 22). Ao final da fase adiciona-se um novo nó S como filho à direita do nó S mais a direita da árvore geral, marcando o fim de uma fase (linha 8) e anexa a nova sub-árvore como filho a esquerda do novo nó S (linha 10). No final, o restante das sub-tarefas consumidoras são adicionadas serialmente como os nós mais a direita do nós S mais a direita (linhas 24-26).

4.3 Ferramentas para Validação do Modelo

O modelo foi validado por um simulador estocástico da rede de filas e por um emulador do sistema. Uma descrição do emulador do sistema é apresentada na Seção 4.3.1. As características do simulador da rede de filas são descritas na Seção 4.3.2.

4.3.1 Emulador do Sistema

O emulador do sistema *HP-Neoview* foi desenvolvido pelo grupo de pesquisadores da *HP-Labs* e foi utilizado no projeto como ferramenta de validação do modelo analítico, visto que ele está em um nível de abstração mais próximo do sistema real do que simulador da rede de filas e que o modelo analítico (desenvolvido pelo nosso grupo de pesquisa). O emulador é um modelo de simulação detalhado do mecanismo de execução do sistema de banco de dados distribuído *HP-Neoview*. Ele é implementado usando um pacote de simulação orientado a processo, CSIM (Schwetman, 2001) que provê uma abstração do processo para modelagem de sistemas de *softwares*. Processos disputam por recursos e o tempo de simulação passa quando os processos usam os recursos.

Um grafo direcionado de operadores, representando a consulta do banco de dados, é aceito como entrada pelo emulador. Ele mapeia os operadores (tarefas) para os processos do CSIM, estabelece um *pipeline* de mensagem entre os processos, inicia os processos e monitora suas execuções. Cada operador *disk* se torna um processo *disk* separado e é atribuído a um *host* específico cujo disco anexado contém os dados requeridos. Cada operador *root* se torna um processo separado e é aleatoriamente atribuído a um *host*. Outro recurso, separado dos *hosts* é usado para modelar a interconexão dos *hosts*. O fluxo de mensagens entre os processos é modelado de modo que o processo pai irá bloquear esperando pelo filho enviar mais dados e então um *back-pressure* irá forçar o processo filho a esperar até que seu pai precise de mais dados.

O emulador suporta vários tipos de disciplinas de escalonamento para gerência dos recursos: *first come first served* (FCFS) para o recurso disco, escalonamento *processor sharing* (PS) para o barramento de rede e escalonamento *priority based preemptive resume* (PB-PR) para o recurso processador. Todas as *queries* rodam com a mesma prioridade. Entretanto, o processo do operador *disk* têm maior prioridade que os processos *root*. O emulador modela algum *overhead* do sistema, como troca de contexto entre as mensagens, mas ele não modela o *buffer* da *cache* nem o *overhead* de gerenciamento de transações, tais como contenção de *locks*, escrita de *logs*, etc.

4.3.2 Simulador da Rede de Filas

Assim como o emulador do sistema (desenvolvido pelo grupo de pesquisa da *HP-Labs*), o simulador da rede de filas (desenvolvido pelo nosso grupo de pesquisa, mais especificamente pela Tatiana e pela Marisa) foi utilizado como ferramenta de validação do modelo analítico. O simulador se posiciona entre o modelo e o emulador em termos de grau de abstração do sistema alvo. Ele possui a vantagem de capturar os aspectos dinâmicos da execução das *queries* (como veremos a seguir), ao contrário do modelo analítico que representa apenas o caso médio e tenta capturar aspectos dinâmicos através de distribuições de probabilidades.

O simulador modela a execução de múltiplas *queries* em paralelo em uma rede de filas (não no sistema real), considerando um modelo fechado, ou seja, assim que um *job* termina sua execução outro *job* é iniciado imediatamente. O simulador reproduz o ambiente analisado pelo modelo analítico, de forma mais dinâmica, executando um conjunto de cargas sintéticas e medindo o desempenho médio delas no sistema. Sendo assim o simulador foi utilizado para validação do modelo, uma vez que ele é capaz de capturar diversos aspectos dinâmicos (contenção, sincronizações, bloqueios).

Para capturar os aspectos dinâmicos foi considerado que a demanda das tarefas (D) são exponencialmente distribuídas. A geração de tempo de serviços exponenciais é baseada na função de distribuição cumulativa (CDF) de uma variável aleatória exponencial (com esperança $D = \frac{1}{\lambda}$), dada por $F(x) = P(X \leq x) = 1 - e^{-\lambda \cdot x}$ (Knuth, 1981; Devroye, 1986). Gerou-se números aleatórios uniformemente distribuídos, utilizando o procedimento Drand48() (biblioteca *stdlib.h*), para $F(x)$ (eixo Y) e desejamos encontrar a média correspondente (x, no eixo X). Para isso é necessário inverter a equação, substituindo λ pela esperança (D), utilizando logaritmo para extrair x e colocando a equação em função de x, obtendo: $x = -D \cdot log_e(1 - F(x))$. Como $1 - F(x)$ é uniformemente distribuído, $F(x)$ também é, simplificando a equação: $x = -D \cdot lnF(x)$.

Primeiramente há um período de *warm-up* (aquecimento) onde as *queries* são dis-

paradas, uma de cada vez, espaçadas por um intervalo de tempo, até atingir o número de *queries* do nível de multi-programação (MPL *multi-programming level*) desejado (parâmetro de entrada). Inicia-se o período de simulação, utilizando uma abordagem orientado a eventos, onde o instante tempo é atualizado a cada ocorrência de um novo evento, caracterizado pelo início ou término de uma visita a um centro de serviço (podendo ocorrer múltiplos eventos em um instante, de acordo com o *quantum* e o tempo de serviço gerado). Assumimos que todos os centros de serviço tem disciplina *processor sharing*, onde o *quantum* foi calculado pela menor demanda de cada centro. O roteamento das tarefas segue a seguinte ordem: primeiramente é visitado a CPU, em seguido vai para o disco e para a rede. Em cada centro o número de visitas é dado pela razão entre a demanda e o quantum do centro de serviço ($\frac{D_k}{quantum_k}$). Sendo assim, quanto menor o *quantum*, maior o número de visitas, levando mais tempo para o simulador executar (significativamente maior que o do modelo, que é quase imediato). O critério de parada do simulador é terminar a execução de $NQueries$ (parâmetro).

$$E[R_0] = \frac{((J-1) \cdot E[R_0] + R_0)}{J} \tag{4.1}$$

Ao final de cada *query* é utilizado o método para cálculo dinâmico da esperança das métricas de desempenho. Por exemplo, o cálculo da esperança do tempo de resposta da *query* (equação 4.1) é dado pelo somatório de todos R_0 até a iteração anterior (inferido pelo produto do número de *jobs* completados menos um com a esperança do tempo de resposta da *query*, $E[R_0]$) somado com o tempo de resposta da *query* atual, R_0 (medido pela diferença entre o instante de tempo que a *query* terminou para o instante que ela começou), sobre o número de *queries* completados (J). As métricas do simulador são medidas, enquanto que no modelo elas são estimadas. Os resultados das simulações são médias de $NQueries$ execuções.

4.4 Análise Experimental

Os resultados da validação da modelagem de desempenho do paralelismo *pipeline* entre duas tarefas (consultas simples de BI) estão sumarizados nesta Seção. O modelo analítico foi validado através de um simulador da rede de filas e de um emulador do banco de dados distribuído *HP-Neoview*. Primeiramente foram descritos a configuração do sistema alvo e os cenários da carga de trabalho avaliados nos experimentos (Seção 4.4.1). A Seção 4.4.2 discute os resultados encontrados na validação. Uma avaliação do impacto dos parâmetros chave do modelo (parâmetros F e M) é feito na Seção 4.4.3

4.4.1 Configuração dos Experimentos

A arquitetura utilizada nos experimentos, como mostrado na Figura 4.1, consiste de quatro nós ($n = 4$). Cada nó contém quatro CPU's ($p = 4$) e dois discos ($d = 2$). O barramento de dados possui uma largura de banda de 1 Gbps e o tamanho máximo do *buffer* de mensagens da rede é de 64 KB. Todos os resultados do simulador e do emulador são médias de 10 execuções, as quais foram mostradas juntamente com o intervalo de confiança de 95%. Cada execução do simulador utiliza uma semente para geração de número aleatórios uniformemente distribuídos diferente e representa o processamento de 10.000 *queries* ($NQueries = 10.000$). Os intervalos de confiança foram muito justos, uma vez que o desvio padrão foi muito baixo (menor que 3% da média).

Foram avaliadas seis cargas de trabalho variando a especificação da consulta ao longo de duas dimensões, demanda de CPU, *bytes* transmitidos pela rede (tamanho da mensagem). O número de sub-tarefas por tarefa (F) pode ser visto como o número de mensagens trocadas entre as tarefas e pode ser derivado pela razão entre o total de *bytes* trafegados (tamanho da mensagem) pelo tamanho máximo do *buffer* de mensagens (arredondando para cima se necessário). Sendo assim, em uma consulta OLTP, a tarefa *disk* envia uma única mensagem para a tarefa *root*, enquanto que em uma consulta de BI simples (longa), o número de mensagens é 7. Consideramos que a fila lógica do consumidor tem capacidade para armazenar os dados produzidos de duas sub-tarefas (duas mensagens) produtoras ($M = 2$). Em uma consulta balanceada, as tarefas *disk* e *root* têm demanda de CPU igual. Em uma consulta do tipo *disk*-maior, a tarefa *disk* usa 39 vezes mais CPU que o *root*. Em uma consulta *root*-maior, a taxa é o inverso. As demandas de serviço foram coletadas em um ambiente paralelo real, no laboratório da *HP Labs*, rodando o sistema HP *Neoview* e estão sumarizadas na Tabela 4.2.

Tabela 4.2. Cenários da carga de trabalho

Tipo de consulta	Tarefa *disk*			Tarefa *root*
	Demanda de CPU (s)	Demanda disco (s)	Total de bytes transmitidos	Demanda de CPU (s)
OLTP balanceada	0,0020	0,007	25.600	0,0020
OLTP *root*-maior	0,0001	0,007	25.600	0,0039
OLTP *disk*-maio	0,0039	0,007	25.600	0,0001
BI simples balanceada	0,0020	0,007	409.600	0,0020
BI simples *root*-maior	0,0001	0,007	409.600	0,0039
BI simples *disk*-maior	0,0039	0,007	409.600	0,0001

Em um sistema de banco de dados real (e no emulador descrito abaixo), uma consulta de BI simples (longa) deve exibir mais paralelismo que uma OLTP (curta). De fato, uma consulta OLTP não deve ter paralelismo de *pipeline* pois só haverá uma única mensagem de saída do *disk* para o *root*, uma vez que a mensagem de saída (25 KB) é menor que o tamanho máximo da mensagem (64 KB). Entretanto, em uma consulta de BI simples, o *disk* irá sempre ter de mandar pelo menos 7 mensagens para o *root*. Uma consulta balanceada deve ter mais paralelismo que uma requisição com uma das tarefas maior, pois a demanda de CPU são balanceadas (assumindo que as tarefas *disk* e *root* são atribuídas a nós diferentes). Note que a demanda de CPU total é a mesma para consultas balanceadas e desbalanceadas. Então, em um único processador sem contenção, eles devem executar em uma mesma duração. As consultas de OLTP foram aproximadas para um modelo serial e as consultas de BI simples foram modeladas pelo modelo do paralelismo *pipeline* entre duas tarefas (produtor e consumidor). Modelamos as consultas OLTP e BI separadamente, pois o modelo analítico não foi derivado para uma carga de trabalho mista (composta por múltiplas classes de *job* concorrentes).

Desta forma, foram criados seis cenários de carga de trabalho diferentes: OLTP balanceado, OLTP *disk*-maior, OLTP *root*-maior, BI simples balanceado, BI simples *root*-maior e BI simples *disk*-maior. Para cada um deles, o emulador foi configurado para rodar uma amostra uniforme de 8 tipos de consultas (8 discos), cada uma com demanda de I/O de um disco diferente do sistema. Nós também parametrizamos nosso modelo analítico e simulador como a seguir. Consideramos os operadores *disk* e *root* como tarefas produtora e consumidora, respectivamente, assumindo $F = 1$ para consultas OLTP e $F = 7$ para consultas de BI simples. Então, ao contrário do emulador, as consultas OLTP não exibem paralelismo de *pipeline* no modelo analítico e no simulador. Nós consideramos a demanda média de I/O de disco da primeira sub-tarefas de *disk* igual a demanda total de disco da tarefa, e distribuímos a demanda de CPU das tarefas *disk* e *root* uniformemente por todas suas sub-tarefas. Nós fizemos

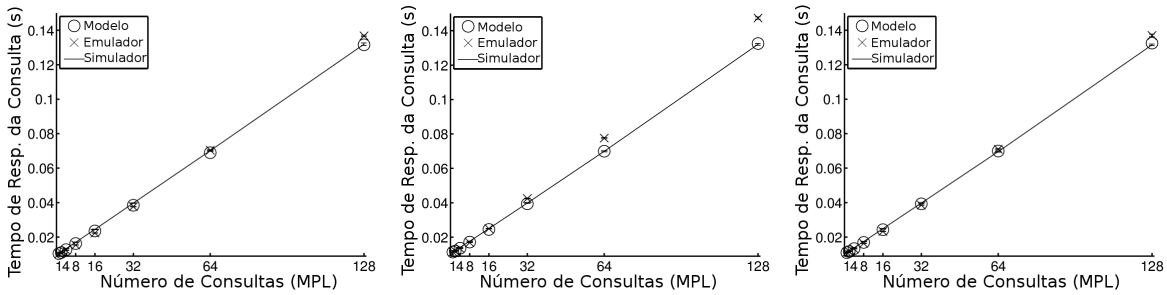
o mesmo para as demandas de rede do *disk*, as quais foram computados com base no tamanho da mensagem e na largura de banda da rede. Finalmente, nós assumimos que as tarefas *disk* e *root* podem ser atribuídas a qualquer nó, e, dentro do nó, para qualquer disco, com probabilidades iguais, o qual replica o que é feito na carga de trabalho submetido no emulador. No simulador, nós roteamos todos as sub-tarefas da mesma tarefa para o mesmo recurso, como esperado em um sistema real. No modelo analítico, nós assumimos probabilidade iguais de roteamento também no nível os sub-tarefas (aproximação).

4.4.2 Validação do Modelo Analítico

A Figura 4.4 mostra o tempo médio de resposta da consulta em função do número de consultas concorrente no sistema (MPL) para cada cenário, fixando $M = 2$. Lembrando que o parâmetro M não tem efeito para consultas OLTP, uma vez que $F = 1$ (serial). Os resultados para outros valores de M são qualitativamente similares e então foram omitidos. Cada gráfico mostra uma curva para o modelo analítico, emulador e simulador. Em todos os seis cenários, as curvas das métricas do modelo analítico ficaram próximas das curvas do simulador e do emulador. Apesar de alguma variações na diferença relativa entre os resultados do modelo analítico e do emulador (cenários BI simples com demanda desbalanceada), tais diferenças estão abaixo de 11.5% em todos os casos, o qual é uma boa aproximação para estimativas de tempo de resposta. Resultados qualitativamente similares foram também obtidos para o *throughput* de consultas e utilização de recursos, como mostrado em 4.5 e 4.6 para os cenários escolhidos (os outros apresentam resultados similares e foram omitidos). A maior diferença relativa foi abaixo de 11% tanto para *throughput* quanto para utilização de recursos.

O modelo analítico provê estimativas de tempo médio de resposta um pouco mais conservadoras que o emulador em vários casos. Em alguns outros, particularmente para MPLs altos, os resultados do emulador são por algum motivo maiores, possivelmente devido a *overheads*, como tempo para troca de contexto. Pontos notáveis de divergência ocorrem na carga *disk*-maior, particularmente para MPL alto. Nestes casos, o modelo analítico subestima o tempo médio em relação ao emulador. Interessante que, à medida que aumenta o tempo médio de resposta da consulta (estimado pelo emulador), o mesmo não é acompanhado pelo *throughput* de consultas e utilização de recursos, como mostrado nas Figuras 4.5. O *throughput* de consultas e a utilização de recursos reportadas pelo emulador são mais baixas que as estimativas correspondentes no modelo. Isto ocorre pois o emulador implementa um escalonamento baseado em prioridade para a CPU o qual favorece tarefas *disk*. Nos cenários *disk*-maior, no qual as tarefas

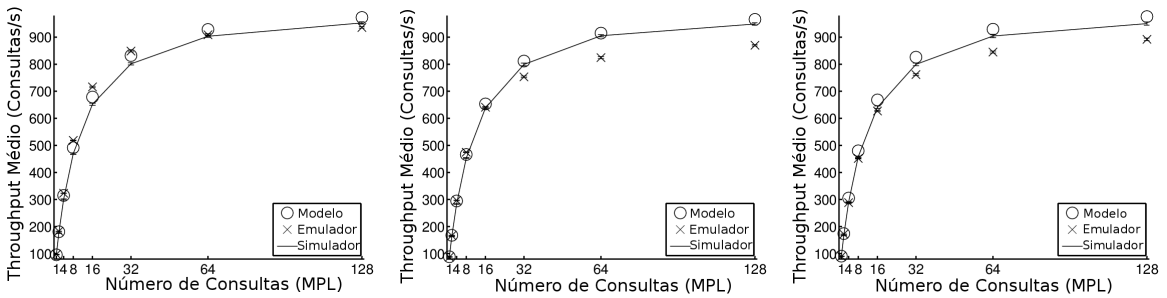
disk têm demanda média de CPU 39 vêzes maior que as tarefas *root*, a priorização do tarefa de maior demanda leva a maior períodos de bloqueio (devido às sincronizações) experienciadas pela tarefa *root*, e consequentemente a maiores tempos de resposta das consultas, como mostrado em 4.4. Observamos que, se a prioridade do disco é desativada no emulador, o modelo se ajusta melhor para os cenários *disk*-maior. Sendo assim, a diferença relativa máxima observada nas estimativas do tempo médio de resposta do modelo, em relação ao simulador e ao emulador, apresentou um bom acordo. A diferença relativa está apenas por volta de 11% para MPL=128 e, como mostrado em 4.6, onde o recurso gargalo, ou seja, a CPU, já está com 90% de utilização.



(a) BI simples balanceada (b) BI simples *disk*-maior (c) BI simples *root*-maior

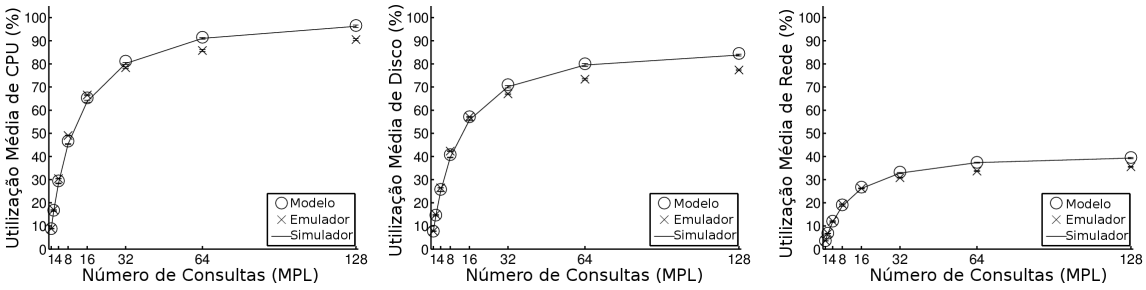
(d) OLTP balanceada (e) OLTP *disk*-maior (f) OLTP *root*maior

Figura 4.4. Validação do tempo médio de resposta das consultas.



(a) BI simples balanceada (b) BI simples *disk*-maior (c) OLTP *disk*-maior

Figura 4.5. Validação do *throughput* médio de consultas de alguns cenários selecionados.

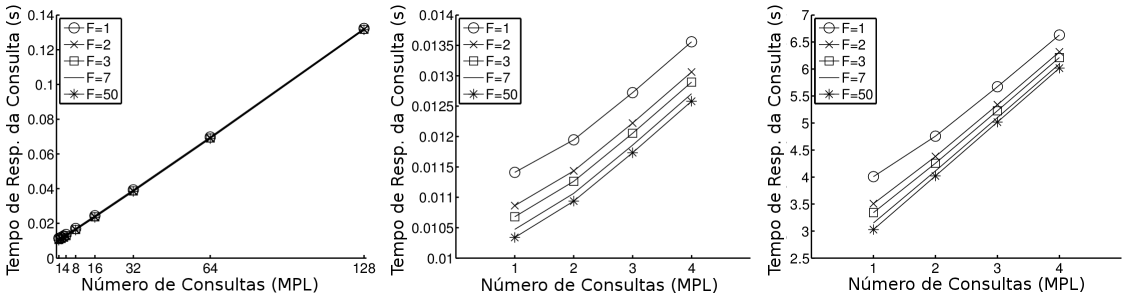


(a) Utilização de CPU (b) Utilização de disco (c) Utilização de rede

Figura 4.6. Validação da utilização média dos recursos (cenário *disk* maior).

4.4.3 Impacto dos Parâmetros do Modelo

Nós iniciamos pela avaliação do impacto do parâmetro número de sub-tarefas por tarefa (F) na estimativa de *performance* produzidas pelo modelo analítico. Nós variamos F, enquanto fixamos $M = 2$ e utilizamos a demanda de serviço das tarefas dos cenários BI simples. Nós esperávamos que maiores valores de F iriam levar a um paralelismo de *pipeline intra-query* maior e, consequentemente reduziria o tempo médio de resposta da consulta. Entretanto, para valores maiores de MPL, o aumento do grau de paralelismo leva também a uma maior contenção de recursos e por fim a maiores tempos de fila. Este aumento da latência acaba compensando o aumento do paralelismo. Sendo assim, o F teve um impacto muito pequeno no tempo médio de resposta da consulta para valores de MPL alto (carga pesada). Isto é ilustrado na Figura 4.7(a), a qual mostra o tempo médio de resposta para o cenário balanceado (BI simples) e vários MPLs. Resultados similares são obtidos para *throughput* de consultas e para utilização de recursos.



(a) Carga pesada (b) Carga leve (c) CPU inflacionada

Figura 4.7. Impacto do parâmetro F

Alguma diferença é observada apenas em cargas de trabalho leves e balanceada, como mostra a Figura 4.7(b). Ainda, o *gap* é mais notável apenas entre $F = 1$ e $F = 2$ e retorna a um valor mais baixo à medida que aumentamos o F. De fato, para MPL igual a 1, o tempo médio de resposta da consulta reduz 5% quando o F vai de 1 para 2. Para MPL igual a 4, a redução já é de 3.7%. Para carga de trabalho *disk*-maior, maiores valores de F levam a maiores atrasos de sincronização experienciada pela tarefa *root*. Entretanto, o tempo médio de resposta da consulta acaba sendo dominado pela tarefa *disk*, o qual é na maior parte inafetado por F. A Figura 4.8 ilustra este efeito para a consulta *disk*-maior, considerando $F = 3$ e $F = 8$, e qualquer valor de M. Embora a tarefa *root* experiencie um tempo de resposta maior (devido a maiores atrasos de sincronização total), o tempo de resposta da consulta é na maior parte dominado pelo tempo de resposta do *disk*. Em outras palavras, a última sub-tarefa do *root* representa uma fração muito pequena do tempo de reposta total da consulta, para ambos os valores de F.

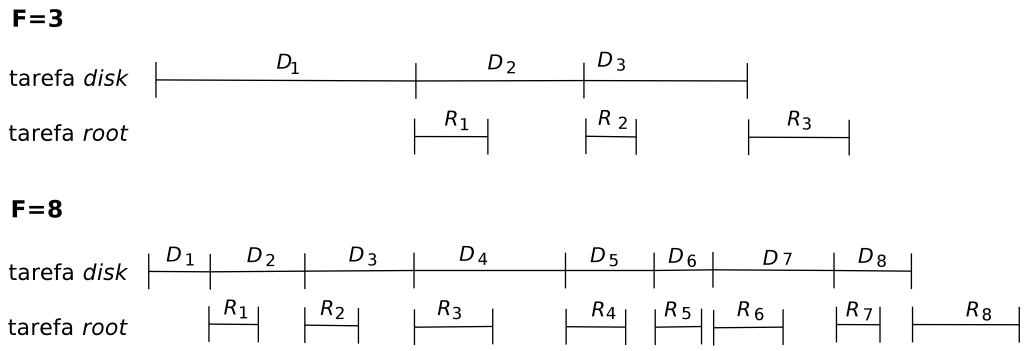


Figura 4.8. Impacto de F no tempo de resposta da query (*disk*-maior, $M = \infty$).

Lembre que toda a demanda de I/O da tarefa *disk* é atribuída a primeira sub-tarefa, o qual deve terminar de processar antes que ocorra qualquer paralelismo. Então, o tempo de resposta da primeira sub-tarefa *disk* corresponde a uma fração significativa do processamento da consulta, particularmente considerando a demanda de CPU mostrada na Tabela 4.2. No intuito de permitir maior paralelismo, nós inflacionamos as demandas de CPU para ambas as tarefas por um fator de 1.000. Como mostrado na Figura 4.7(c), foi observado um impacto maior na variação do parâmetro F, embora um efeito decrescente, à medida que aumentamos o parâmetro F, também é notável. Em particular, para $MPL = 1$, o tempo médio de resposta da consulta reduz 12.5% a medida que F vai de 1 para 2. Para MPL igual a 4, a redução é de 4.7%.

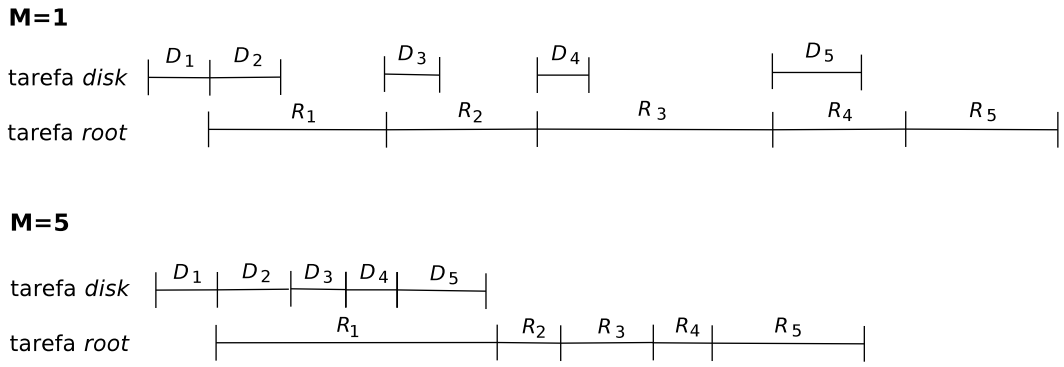


Figura 4.9. Impacto de M no tempo de resposta da query (*root*-maior, F=5).

Nós também avaliamos a sensitividade do modelo em relação ao parâmetro tamanho máximo da fila lógica M. Novamente, não encontramos nenhum impacto para cenários BI simples (longos), para os quais a contenção de recurso é um fator dominante. Focando nas cargas de cenários leve, para as cargas *disk*-maior, onde haverá raramente um bloqueio no *disk*, independente de M, uma vez que o *root* roda mais rápido. Então, M tem um pequeno impacto no tempo de resposta total da consulta, o qual é dominado principalmente pelo tempo de resposta do *root*. Isto é ilustrado na Figura 4.9, com $M = 1$ e $M = 5$, fixando $F = 5$.

Para cargas balanceadas, as tarefas experienciam eventualmente algum bloqueio dependendo das dinâmicas de execução das tarefas. Entretanto, isto não é claramente capturado por nosso modelo analítico, o qual constrói a árvore de precedência baseado apenas no tempo médio de resposta das sub-tarefas. Em contrapartida, nós observamos diferença pouco significativa nos resultados do simulador para MPL igual a 1. Por exemplo, fixando $F = 100$, e variando M de 1 a 10, nós notamos uma diferença de até 20%. Contudo, para maiores MPLs, a diferença cai nitidamente abaixo de 10%.

Capítulo 5

Modelagem do Paralelismo Pipeline de $M \times N$ Tarefas

Este Capítulo apresenta a estratégia utilizada para modelar a carga de trabalho do sistema *Haddop Online Prototype* (HOP) (Condie et al., 2010), uma extensão do *Hadoop*, caracterizado pela presença do paralelismo *pipeline* entre múltiplas tarefas. A Seção 5.1 apresenta uma descrição do processo de execução dos *jobs* na plataforma HOP, destacando o funcionamento do *pipeline* e ressaltando pontos relevantes para a modelagem. A Seção 5.2 explica as extensões que foram feitas na nossa estratégia de modelagem para atender os requisitos do *pipeline* de $m \times n$ tarefas. As ferramentas utilizadas na validação do modelo analítico são apresentadas na Seção 5.3. Os resultados dos experimentos de validação do modelo, em relação ao simulador e a experimentos reais, estão sumarizados na Seção 5.4.

5.1 Descrição do Sistema

Em geral, a infra-estrutura de um *cluster* HOP consiste em n *hosts* (nós), onde há um nó *master* e vários nós trabalhadores. O nó *master* executa o processo *JobTracker* e é responsável por aceitar novos *jobs* e atribuí-los aos nós trabalhadores. Cada nó trabalhador executa um processo *TaskTracker* e gerencia a execução das tarefas atribuídas a seu nó. Cada nó trabalhador tem um ou mais discos associados com um sistema de arquivo local e um compartilhado (*Hadoop Distributed File System* - HDFS).

Um *job* HOP, como mostrado na Figura 5.1, lê um ou mais arquivos de entrada e produz um arquivo de saída. Cada arquivo é uma sequência de registros contendo dois campos: uma chave e um valor. Cada *job* possui duas fases: *map* e *reduce*. Estes arquivos são logicamente particionados, definindo a faixa de chaves a ser processada por

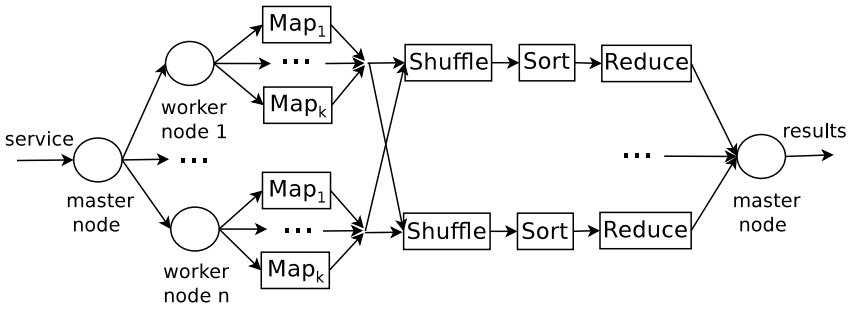


Figura 5.1. Fluxo de execução de um *job* do *HOP*

cada nó trabalhador. Cada partição é dividida em sub-partições de tamanho igual, chamados *splits*. Cada fase *map* e *reduce* são divididas em m e r tarefas, respectivamente. Além disso, o número de *threads* disponível em cada nó para executar (em paralelo) tarefas *map* e *reduce* são fixas, dado pelos parâmetros *pm* e *pr*, respectivamente. O nó *master* usa um protocolo *heartbeat* para checar o *status* dos nós trabalhadores. Quando um nó *master* identifica que um nó trabalhador possui uma *thread* ociosa (ou seja, ele terminou de processar a tarefa *map reduce* atribuída a ele anteriormente), ele atribui uma nova tarefa do mesmo tipo a esta *thread*.

Cada tarefa *map* lê os registros de um *split* dos arquivos de entrada. Para cada registro do *split*, ela executa a função MAP() e produz um ou mais registros de saída. Os registros de saída são então ordenados e escritos em um arquivo temporário no sistema de arquivo local. Neste ponto a tarefa *map* termina sua execução e a *thread* correspondente se torna ociosa.

Cada tarefa *reduce* é responsável pelo processamento de uma faixa de chave produzido pelas tarefas *map*. Cada tarefa *reduce* possui três sub-fases: *shuffle*, *sort* and *reduce*. Uma sub-fase *shuffle* requisita a cada nó que percorra o arquivo temporário produzido pelos *maps*, procurando por registros na faixa de chaves do *reduce* a qual ela pertença. Além do nível de paralelismo da tarefa, o HOP permite também que cada *reduce* execute um número *ps* de instâncias *shuffle* em paralelo. A sub-fase *shuffle* é então dividida em $\frac{m}{ps}$ sub-tarefas, onde cada sub-tarefa processa os resultados de *ps* tarefas *map* (em paralelo). Então, cada *shuffle* está associado com *ps* tarefas *map*. Após cada *shuffle*, uma ordenação parcial é realizada. Os registros são escritos em um arquivo de saída no sistema de arquivo local do nó do *reduce*, o qual pode ser diferente do nó que o *shuffle* esteja executando. Quando todos os *sorts* parciais terminarem, um *sort* final, que faz um *merge* de todos os arquivos temporários produzidos pelos *shuffles*, é realizado. Por fim, a sub-fase *reduce* lê este arquivo ordenado, aplica a função REDUCE() e escreve os resultados no sistema de arquivo compartilhado.

Note que os resultados produzidos por cada tarefa *map* deve ser processado por todas as tarefas *reduce*. Ou seja, quando uma tarefa *map*, executando em um nó n_i, termina sua execução, cada tarefa *reduce*, executando no nó n_j, envia uma sub-tarefa para n_i para transferir os dados de n_i para n_j. Além disso, após cada *map* rodando no nó n_i terminar, uma nova tarefa *map* (se ainda houve alguma para ser executada) é dinamicamente alocada para n_i. Note que, uma tarefa *reduce* pode ser bloqueada se, quando todas as instâncias de uma sub-tarefa *shuffle* terminarem de processar os resultados de um sub-conjunto de tarefas *map*, ainda não houver *maps* finalizados. Então, pode-se ver uma comunicação entre uma coleção de tarefas *map* e um conjunto de sub-tarefas *shuffle* de cada *reduce* como um *pipeline*, onde cada *shuffle* (consumidor) precisa processar os resultados de todos os *maps* (produtores).

5.2 Modelagem Analítica

Esta Seção apresenta a aplicação da metodologia proposta nesta dissertação para modelagem da carga de trabalho composta por *jobs* do *Hadoop Online Prototype* (HOP), caracterizada pela presença de um paralelismo *pipeline* entre múltiplos produtores e consumidores. O mapeamento da arquitetura para o modelo de rede de filas é apresentado na Seção 5.2.1. A decomposição da carga de trabalho do HOP para modelar o paralelismo *pipeline* é descrita na Seção 5.2.2. A Seção 5.2.3 apresenta o método utilizado para construção da árvore de precedência do fluxo de execução de um *job* do HOP.

5.2.1 Modelo Descritivo da Arquitetura do Sistema

O sistema alvo, ilustrado na Figura 5.2, representa a arquitetura com n nós, cada um com c CPU's e d discos, onde os discos são conectados às CPU's através de um canal de fibra e os nós são interconectados por uma rede de alta velocidade. Esta arquitetura foi modelada através de um modelo de filas fechado, com centros de serviço representando cada CPU, disco, fibra e rede. As restrições de memória não foram explicitamente modeladas. O nível mais alto é dado por uma árvore de precedência que captura as sincronizações e restrições precedências entre as tarefas.

5.2.2 Decomposição da Carga de Trabalho

A carga de trabalho é composta por um número MPL de *jobs* executando concorrentemente no sistema. Cada *job* possui m tarefas *map* e r tarefas *reduce*. Lembre

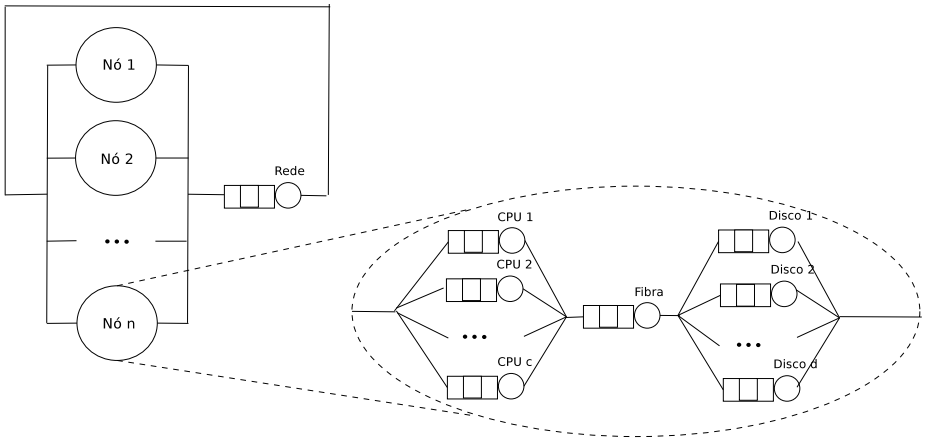


Figura 5.2. Diagrama da rede de filas do sistema alvo.

que, uma ordenação parcial é realizada após cada *shuffle*. Como estas sub-fases são realizadas em sequência, optou-se, para propósito de modelagem, em agrupar cada par de *shuffle* e *sort* em uma única sub-tarefa chamada *shuffle-sort* (ss). Similarmente, após todos os *sorts* terminarem, é executado um *sort* final seguido pela sub-fase que aplica a função *Reduce()*. Optou-se também em agrupar essas duas sub-fases em uma única sub-tarefa chamada de *merge*. Então, no modelo adotado, cada *reduce* é composto por $\frac{m}{ps}$ *shuffle-sort* (executando ps instâncias) e uma sub-tarefa *merge*. Nós consideramos sub-tarefas individuais como a unidade de carga. Para este propósito, uma tarefa *map* pode também ser vista como uma sub-tarefa.

A tabela 5.1 sumariza os parâmetros de entrada do modelo analítico, que podem ser divididos em dois grupos: (1) parâmetros da arquitetura do HOP, onde temos n nós, cada nó com c CPU's e d discos. (2) parâmetros da carga de trabalho dos *jobs* do HOP, que descrevem o número de tarefas de cada tipo (m e r), o número de *threads* para processar cada tipo de tarefa ($pmepr$), o número de *shuffle* em paralelo (ps) e a matriz com a demanda de cada sub-tarefa i em cada centro de serviço k.

O principal desafio em desenvolver este modelo foi em como capturar os atrasos de sincronização introduzidos pelo paralelismo *pipeline* que ocorre entre *maps* e *shuffle-sorts*. A solução adotada foi construída sobre um modelo referência, descrito na Seção 2.2. Lembre que o modelo referência considera que a árvore de precedência é dado como um parâmetro de entrada. Entretanto as precedências dos *jobs* HOP não são estáticas, impedindo de passar a árvore como entrada. Por exemplo, uma tarefa *shuffle-sort* inicia sua execução quando o primeiro *map* terminar. Este momento depende da contenção nos recursos experienciada por cada *map*, ou seja, um *map* pode levar mais tempo para terminar se ele executar concorrentemente com várias outras tarefas, que compartilham os mesmos recursos. Além disso, um *shuffle-sort* pode bloquear

Tabela 5.1. Notação dos parâmetros de entrada do modelo de precedência.

Notação	Parâmetros de entrada
n	Número de nós
c	Número de CPU por nó
d	Número de discos por nó
$D_{i,k}$	Demanda média da tarefa i no centro k
m	Número total de tarefas *map*
r	Número total de tarefas *reduce*
pm	Número máximo de tarefas *map* em paralelo por nó
pr	Número máximo de tarefas *reduce* em paralelo por nó
ps	Número de *maps* agregados por um *shuffle-sort*

dependendo da dinâmica da execução dos outros *maps*. *Shuffle-sorts*, executando no mesmo nó, por sua vez, determinam o tempo que o *merge* deve iniciar. Em outras palavras, há uma complexa relação de dependências entre as tarefas. A árvore de precedência depende do tempo de resposta das tarefas individuais, as quais dependem da contenção experienciadas por elas. A contenção do recurso depende em quais tarefas executam concorrentemente, as quais, por sua vez, dependem da árvore de precedência.

Como os *clusters* HOP são usados para rodar *jobs* com paralelismo massivo, em geral, não haverá utilização de recursos disponíveis para executar mais de um *job* em paralelo. Então foi assumido que $MPL = 1$, ou seja, considera-se que há um único *job* em execução no sistema, ciclando pela rede de filas (modelo fechado). Como consequência, a sobreposição entre *jobs* distintos, computada no passo 3 do algoritmo original (veja Figura 3.2(a) não foi utilizado na solução adotada, mostrada na Figura 3.2(b)).

5.2.3 Construção da Árvore de Precedência

Há também dois desafios para dinamicamente construir esta árvore de precedência: (1) como estimar quando cada tarefa inicia e termina, e (2) como estimar o tempo de resposta médio dos nós internos da árvore. A Seção **??** descreve como o modelo referência (Liang e Tripathi, 2000) foi estendido para capturar as sincronizações do *pipeline* e representá-las através de uma árvore de precedência. A Seção 5.2.3.1 apresenta como o paralelismo *pipeline* dos *jobs* HOP foi modelado. A identificação das regras de precedência do fluxo de execução do *job* foi apresentado na Seção 5.2.3.2. Foi desenvolvido um algoritmo para dinamicamente construir a árvore de precedência de um *job* HOP e adicionamos este método como um passo extra no algoritmo proposto por Liang e Tripathi (2000), como mostrado na solução proposta na Figura 3.2(b).

5.2.3.1 Modelagem do Paralelismo Pipeline

Uma vez que o modelo analítico provê estimativas médias de desempenho de *jobs* HOP, a árvore de precedência deve capturar os atrasos de sincronização e o paralelismo experienciado, em média, pelas tarefas do *job*. A estratégia utilizada para construir esta árvore é baseada na linha de tempo da execução das tarefas individuais do *job*, a qual é construída a partir da estimativa do tempo médio de resposta das tarefas pelo modelo de rede de filas e por um conjunto de regras definindo as restrições de precedência entre diferentes tarefas. Nesta Seção será discutido como a árvore de precedência pode ser construída a partir da linha do tempo do *job*. Uma discussão sobre como identificar as precedências para construir esta linha do tempo é apresentada na Seção 5.2.3.2.

A Figura 5.3 mostra um exemplo simples de uma linha do tempo representando a execução de um *job HOP* composto por dois *maps* ($m = 2$) e um *reduce* ($r = 1$). Cada nó possui duas *threads* para processar *maps* em paralelo ($pm = 2$) e uma *thread* para processar *reduce* ($pr = 1$). O *reduce* é composto por quatros *shuffle-sorts* ($R_1 - R_4$) e uma tarefa *merge* (R_M), as quais devem ser executadas em sequência. Para facilitar o entendimento, foi utilizando uma linha do tempo simples para ilustrar a estratégia de construção da árvore de precedência. Note que a estratégia pode ser aplicada em *jobs* muito mais complexos, sendo executados em uma larga infra-estrutura.

Como mostrado na Figura 5.3, as duas *threads map* iniciam a execução imediatamente no início da execução do *job*, enquanto que a *thread reduce* fica bloqueada. Assim que o primeiro *map* termina, a primeira sub-tarefa *shuffle-sort* (R_1) inicia sua execução e outra tarefa *map* M_3 é atribuída a *thread* que esta executando M_1. Neste ponto do tempo, mostrado na Figura 5.3 por uma linha vertical tracejada, marca um ponto de sincronização, onde o conjunto de tarefas em execução em paralelo mudam: M_3, M_2 e R_1 estão agora em execução. Note que, como M_3 é executada apenas após M_1 terminar, há uma precedência serial entre elas. O mesmo se aplica entre R_1 e M_1. Após R_1 terminar, a *thread reduce* bloqueia, esperando pelo próximo *map* terminar. Quando M_2 termina, há outro ponto de sincronização, marcado pelo inicio da execução de R_2. A próxima sincronização ocorre apenas quando M_4 termina e R_4 inicia sua execução. Note que R_3 pode iniciar sua execução imediatamente após R_2 terminar, pois M_3 já havia terminado neste momento.

Então, a execução de um *job* HOP pode ser visto como um série de pontos de sincronização, cada um dos quais delimita a execução paralela de um conjunto diferente de tarefas. Durante cada "fase paralela", um conjunto de tarefas *maps* e *reduces* são executados em paralelo. Como a mesma tarefa *map* pode executar durante múltiplas fases paralelas, por exemplo, M_2 na Figura 5.3, nós utilizamos sobrescrito

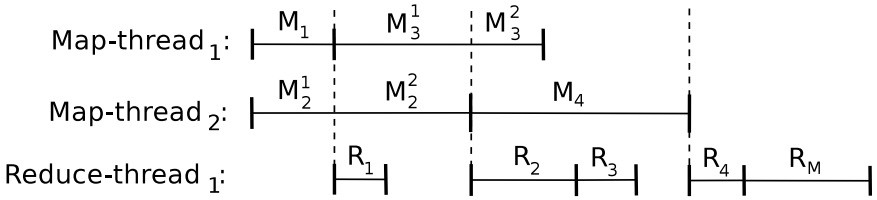


Figura 5.3. Linha do tempo de um *job* HOP.

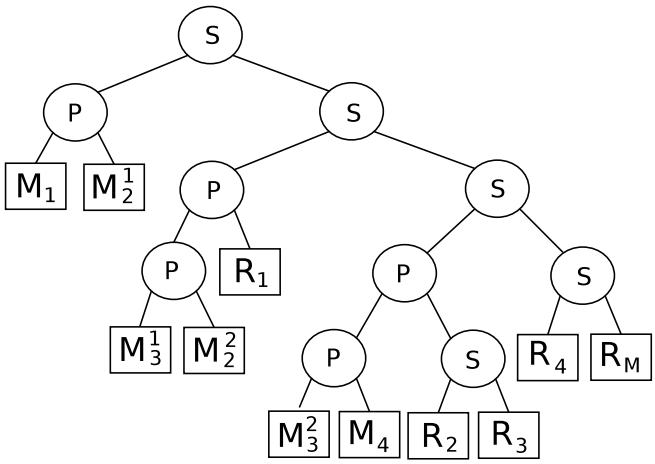


Figura 5.4. Árvore de precedência referente a linha do tempo da Figura 5.3.

para distinguir entre múltiplas fases da execução da mesma tarefa, por exemplo M_2^1 e M_2^2 e tratamos cada fase como sub-tarefas independentes. Claramente, estas sub-tarefas devem executar sequencialmente.

A árvore de precedência é construída criando múltiplas sub-árvore com um operador P na raiz, uma para cada fase paralela, e as interconectando com operadores S. Então cada sub-árvore com raiz P correspondem a uma fase paralela. Um nó S é adicionado para representar o fim de uma fase, assim como a serialização de *maps* dinamicamente alocados para a mesma *thread* e a execução serial das sub-tarefas *shuffle-sorts* e *merge* de um dado *reduce*.

A Figura 5.4 mostra a árvore de precedência correspondente a linha do tempo da Figura 5.3. Cada nó folha representa uma sub-tarefa *map*, *shuffle-sort* ou *merge* (M_i e R_i), os nós internos podem ser operadores S ou P, e a raiz representa o *job*. Sub-tarefas representando fases de execução de uma mesma tarefa (como o *shuffle-sort* e o *merge*), assim como sub-tarefas executadas pela mesma *thread* devem ser executadas em série (S), enquanto que sub-tarefas executadas por diferentes *threads* durante uma mesma fase devem rodar em paralelo (P).

Em termos gerais, a árvore é construída mapeando os pontos de sincronização em uma sub-árvore com um operador S na raiz que possui dois filhos. O filho a esquerda

é a sub-árvore representando a fase paralela anterior e o filho a direita, se não for a última fase, será um operador S, ligando a sub-árvore da fase anterior à sub-árvore da próxima fase paralela, senão será a sub-árvore da última fase paralela. Cada sub-árvore com um operador P na raiz de uma fase paralela possui dois galhos (filhos): um para as *threads map* em paralelo à esquerda e outro para as *threads reduce* em paralelo do lado direito. Cada galho é então uma sub-árvore com um operador P na raiz representando a execução paralela de múltiplas *threads* de um dado tipo. Um nó S é adicionado para representar a execução de múltiplas tarefas do mesmo tipo (*map* ou *shuffle-sort*) de uma mesma *thread* durante a mesma fase paralela.

Note que a forma exata da árvore de precedência depende do número de pontos de sincronização, os quais, por sua vez, dependem do tempo médio de resposta de cada tarefa. Então, a árvore deve ser reconstruída a cada iteração do algoritmo (Figura 3.2(b)), usando o tempo médio de resposta das tarefas estimadas na iteração anterior. O algoritmo para construção da árvore de precedência é apresentado com mais detalhes no Apêndice B. Uma vez que a árvore foi construída, ela pode ser utilizada ao longo da solução de Liang e Tripathi (2000) para produzir novas estimativas de desempenho das tarefas e do *job*. Isto foi feito considerando as tarefas individuais e, neste caso também as tarefas que abrangem múltiplas fases, como a unidade de carga. A seguir, será discutida como a linha de tempo do fluxo de execução de um *job* HOP é construída dado as estimativas do tempo de resposta médio das sub-tarefas. Será discutido também como que o tempo médio de resposta correspondente às fases paralelas da execução do *job* são estimados.

5.2.3.2 Identificação das Regras de Precedência

Para definir a execução da linha de tempo de um *job* HOP, deve-se identificar os eventos de término de sub-tarefas. Tais eventos podem disparar pontos de sincronização e o início da execução de outras sub-tarefas. Ou seja, a cada evento, um conjunto de regras de precedência, definidas baseado no *pipeline* entre *maps* e *shuffle-sorts* e a relação serial entre sub-tarefas, como por exemplo *maps* alocados dinamicamente para uma mesma *thread*, são aplicadas para determinar se novas tarefas podem ser executadas ou se haverá bloqueio. Para capturar a sincronização do *pipeline*, assumiu-se que cada *thread reduce*, as quais executam todos os *shuffle-sorts* e o *merge* de um único *reduce*, possui uma fila lógica associada a ela. Como todos os *maps* devem ser processados por todos os *reduces*, cada *map*, após o término de sua execução, coloca um "token"na fila lógica de cada *reduce* para indicar que seus resultados estão prontos para serem processados por eles. Um *shuffle-sort* inicia sua execução se há pelo menos

ps "tokens"para serem consumidos na fila lógica correspondente na *thread* do *reduce*. Os eventos a seguir foram considerados:

Map M_i é o próximo a terminar: este evento ocorre se o tempo médio de resposta acumulado da *thread* que está executando M_i, definido pela soma do tempo médio de resposta de todas as tarefas *map* já processadas pela *thread* de M_i, é o menor entre o tempo de resposta médio acumulado de todas as *threads map* e *reduce* em execução. Deve-se em seguida verificar se há *threads reduces* bloqueadas. Se houver e se a fila associada ao *reduce* tiver pelo menos *ps* "tokens"*map* processados, então estes *reduces* devem ser desbloqueado e um novo *shuffle-sort* é executado pela *thread* de cada um destes *reduces* previamente bloqueados. O *reduce* remove os *ps* "tokens" de uma só vez quando uma sub-tarefa *shuffle-sort* (com *ps* instâncias em paralelo) inicia. Além disso, se ainda há *maps* esperando para serem executados, um deles será dinamicamente alocado para a *thread* de M_i, que inicia sua execução. Note que, se houver *reduces* bloqueados e se a fila tiver tamanho maior ou igual a *ps*, este evento implica em um fim de uma fase paralela, ou seja, em um ponto de sincronização.

O *shuffle-sort* S_i é o próximo a terminar: se S_i é o último *shuffle-sort* a ser executado pela *thread*, então ele inicia a execução da tarefa *merge*. Caso contrário, ele verifica a fila lógica. Se a fila tiver pelo menos *ps* "tokens", então a próxima sub-tarefa (S_{i+1}) começa a executar e remove *ps* "tokens"de uma só vez da fila lógica do *reduce*. Caso contrário, se a fila lógica tiver menos que *ps* "tokens", a *thread* do *reduce* deve esperar pelo término de *ps maps*.

Um *merge* R_{M_i} for o próximo a terminar: neste caso a *thread* do *reduce* que está processando R_{M_i}, termina sua execução.

A ideia básica deste algoritmo é caminhar dinamicamente sobre a linha de tempo da execução de um *job* HOP, comparando o tempo de resposta acumulado de todas as *threads map* e *reduce* em paralelo e identificando os eventos de término de sub-tarefas. A cada evento, são aplicadas as regras descritas acima para mover em frente na linha do tempo. Ao mesmo tempo, adiciona-se novos nós e sub-árvores à árvore de precedência que correspondem a novas sub-tarefas e fases paralelas adicionadas a linha de tempo, como descrito na Seção 5.2.3.1.

Este algoritmo toma como entrada o tempo médio de resposta das sub-tarefas (M_i, R_i e R_M)i, o número de *thread* de cada tipo de tarefas (pm e pr), o número de *threads* para processar *shuffles* em paralelo (ps) e o número total de tarefas. O último pode ser calculado pelo número de sub-tarefas *map* e *reduce*.

Como uma nota final, lembre que em sistemas reais o *shuffle-sort* inicia quando o primeiro *map* termina. O mesmo se aplica ao desbloqueio da *thread* do *reduce*. Então, o menor tempo de resposta de todos os *maps* define quando os *shuffle-sorts* iniciam.

Como a linha do tempo e a árvore de precedência são baseadas no tempo médio de resposta, no intuito de estimar o tempo de início de um *shuffle-sort*, deve-se tomar a média do mínimo do tempo de resposta de todos os *maps* em execução. Isto corresponde a calcular a média do mínimo de diferentes distribuições exponenciais. Mais uma vez, isto não é a mesma coisa que tomar o mínimo das médias. O algoritmo detalhado para identificação das precedências é apresentado no Apêndice C.

5.3 Ferramentas para Validação do Modelo

O modelo analítico foi validado por um simulador estocástico da rede de filas e através de experimentos reais (descrito na Seção 5.1). O simulador da rede de filas é utilizado como ferramenta de validação do modelo analítico, uma vez que ele se posiciona entre o sistema real e o modelo analítico quanto ao nível de abstração considerado. As modificações feitas no simulador para contemplar as particularidades dos *jobs* do HOP estão descritos na Seção 5.3.1.

5.3.1 Simulador da Rede de Filas

Esta Seção apresenta as modificações realizadas no simulador (desenvolvido inicialmente para a modelagem do paralelismo *pipeline* entre duas tarefas, Seção 3), para avaliar os *jobs* do arcabouço de computação paralela *Hadoop Online Prototype* (HOP), que possui relações de precedências mais complexas na comunicação entre múltiplos produtores (tarefas *map*) e múltiplos consumidores (tarefas *reduces*). Estas extensões foram feitas inicialmente pela aluna Tatiana Pontes e depois continuadas por mim.

A primeira modificação feita foi adicionar os novos parâmetros da carga de trabalho do *jobs* do HOP: número de tarefas *map* (m), número de *reduces* (r), número de *threads* para processar tarefas de cada tipo (pm, pr), número de *threads* para processar *shuffles* em cada *reduce* (ps). Na arquitetura foi adicionado o recurso canal de fibra interligando os nós ao *array* de discos, que não havia na arquitetura do banco de dados distribuído. O roteamento de cada tarefa inicia sua execução na CPU, depois vai para a fibra, em seguida vai para o disco e para rede (se houver). Assumimos que as filas de todos os centros têm disciplina *processor sharing* (PS) com *quantum* dado pela menor demanda encontrada para cada tipo de recurso. Sendo assim as tarefas circulam várias vezes por cada centro.

Na carga de trabalho, generalizamos o número de tarefas (que antes só consideravam duas tarefas) para permitir o *pipeline* de múltiplos produtores e consumidores. Em seguida passamos a unidade de carga para sub-tarefas, modificando as filas. Como

o simulador é orientado a eventos, modificamos as regras que identificam o início/fim dos eventos, que caracterizam o fluxo de execução das tarefas dos *jobs* HOP (descritos com mais detalhes na Seção 5.2.3.2). Assim como era feito antes, o tempo de serviço das tarefas nos centros são exponencialmente distribuídos (estocástico).

5.4 Análise Experimental

Esta Seção apresenta os resultados da aplicação do método de modelagem de desempenho desenvolvido na carga de trabalho do arcabouço de computação paralela *Hadoop Online Prototype* (HOP). O maior desafio consiste em capturar as sincronizações do paralelismo *pipeline* entre múltiplas tarefas *map* e *reduce*. A Seção 5.4.1 apresenta a configuração utilizada nos experimentos. Os resultados da validação de vários cenários, em relação ao simulador da rede de filas, foi apresentado na Seção 5.4.2. Entretanto alguns destes cenários foram super-estimados pelo modelo, nos levando a introduzir, na Seção 5.4.3, um novo operador primitivo que utiliza um método diferente para estimar o tempo de resposta de um sub-conjunto de tarefas em paralelo. Foi feito uma avaliação das limitações do modelo analítico, na Seção 5.4.4, através de simulações simplificadas, buscando identificar qual a premissa, adotada no modelo, que levou a previsões de desempenho super-estimadas, assim como, avaliar se a utilização do novo operador obtém uma melhor aproximação. Os resultados da validação por simulação, utilizando o novo operador, foram apresentados na Seção 5.4.5, enquanto que os resultados da validação por experimentos reais foi mostrada na Seção 5.4.6. Foi feito também, na Seção 5.4.7, uma avaliação do impacto dos principais parâmetros do modelo.

5.4.1 Configuração dos Experimentos

A aplicação utilizada nos experimentos foi a ordenação de um arquivo no arcabouço de computação paralela *Hadoop Online Prototype* (HOP), implementação do modelo de programação *MapReduce*, que provê um paralelismo *pipeline* as tarefas *map* e *reduce*. A ordenação é uma aplicação bastante estudada para o *MapReduce*, pois ela permite avaliar sua escalabilidade, ou seja, quanto que o tempo de resposta do *job* varia quando o tamanho do arquivo é fixo e o número de nós varia ou, o oposto, quando o número de nós é fixo e o tamanho do arquivo varia. *MapReduce* é um *framework* muito conveniente para estes experimentos. O fluxo de execução das tarefas na ordenação pelo HOP se dá da seguinte forma: a tarefa *map* extrai a chave do registro de entrada e grava um registro de saída com a chave ordenada e o restante dos dados do registro em

Tabela 5.2. Demandas de serviço reais, coletada para cada tarefa em cada recurso.

pm	ps	Tarefa	CPU (s)			Fibra	Disco	Rede
			Node 1	Node 2	Node 3	(s)	(s)	(s)
1	1	Map	5,0764			0,0677	3,1844	0,0
		SS	0,5293	0,5842	0,5217	0,0085	0,4003	0,1815
		Merge	40,3753	41,9946	37.4983	1,2583	59,2157	0,0
1	5	Map	5,1360			0,0677	3,1847	0,0
		SS	3,0615	2,9475	2,9440	0,0425	2,0	0,9067
		Merge	21,215	20,425	20,400	1,258	59,2157	0,0
4	1	Map	5,1057			0,0677	3,1844	0,0
		SS	0,5425	0,6887	0,6608	0,0085	0,4007	0,1817
		Merge	41,956	48,243	47,258	1,258	59,2157	0,0
4	5	Map	5,0764			0,0677	3,1844	0,0
		SS	2,7524	2,8628	2,5563	0,0425	2,0	0,9067
		Merge	41,348	43,006	38,402	1,258	59,216	0,0

um arquivo temporário. Cada tarefa *reduce* é responsável por gravar todos os valores associados com sua faixa de chaves, que estão distribuídos pelos *maps*.

A arquitetura utilizada foi representada por um modelo de rede de filas fechado, composto de n nós e uma rede, onde cada nó possui c CPUs interligados a um *array* de d discos (dedicados a este nó) por um canal de fibra. O ambiente de experimentação foi configurado no laboratório do grupo de pesquisa da *HP Labs*, onde prepararam três máquinas ($n = 3$), para medição das demandas e avaliação dos experimentos. Cada máquina contém quatro CPU's ($c = 4$) e um disco dedicado no *array de discos* ($d = 1$). Foi instalado em cada máquina o arcabouço *open source* HOP e dividido os discos do *array de discos* em duas partições: uma partição local e outra para o *Hadoop File System* (HDFS). Como a nossa modelagem não considerou restrições de memória (memória infinita), configuramos as máquinas virtuais *Java* do HOP para utilizar o máximo de memória disponível em cada nó.

A carga de trabalho considerada consiste na ordenação de um arquivo com 100.000.000 registros, de 100 *bytes* cada. Para processar esta carga foram utilizados 150 tarefas *map* e 3 *reduces* (um por nó). Foram coletados, para cada tarefa: (1) o tempo de execução efetivo da CPU, em nanosegundos; (2) o total de *bytes* lidos e escritos pelo sistema de arquivo local; (3) o total de *bytes* lidos e escritos pelo sistema de arquivo compartilhado (HDFS); (4) o total de *bytes* transmitidos pelo canal de fibra; (5) e o total de *bytes* transmitidos pelo barramento de rede. Consideramos que o disco lê e grava à 85MB/s e que o canal de fibra, assim como o barramento de rede, transmitem a uma velocidade de 1 Gbit/s. A partir da velocidade dos dispositivos foi possível calcular a demanda em cada recurso.

A demanda de disco depende do sistema de arquivo utilizado por cada tarefa. Consideramos que as tarefas *map* consomem toda a demanda de leitura do HDFS, enquanto que os *reduces* consomem toda a demanda de escrita do HDFS. A demanda do sistema de arquivo local se deve à materialização em disco dos resultados parciais de cada sub-tarefa. Não foi possível coletar estes dados para as sub-tarefas *shuffe-sort* e *merge* separadamente, pois as demandas foram agregadas por tarefa e compreendem as duas sub-tarefas do *reduce*. Para distribuir a demanda pelas sub-tarefas do *reduce*, primeiramente consideramos que o *shuffle-sort* consome metade da demanda de leitura do sistema de arquivo local e o *merge* a outra metade. Como o *shuffle-sort* grava os resultados em um arquivo temporário e o *merge* no HDFS, cada sub-tarefa, consome inteiramente as respectivas demandas (de escrita). As demais demandas (CPU, fibra e rede) foram estimadas com base na porcentagem do tempo, de cada sub-tarefa, sobre o tempo total do *reduce*, dado pelo *timestamp* de cada sub-tarefa (em nanosegundos), assumindo assim um consumo uniforme por estas sub-tarefas. A Tabela 5.2 sumariza as demandas coletadas para cada sub-tarefa em cada recurso. Note que, a CPU apresentou velocidades diferentes em cada nó não sendo possível agrupá-las em um único valor.

5.4.2 Validação do Modelo por Simulação

O simulador busca replicar o que é feito no modelo analítico, com o intuito de avaliar as premissas adotadas. No simulador as demandas por recurso, de cada tarefa, são exponencialmente distribuídas, diferentemente do modelo, que assume a média das demandas, sendo assim, o simulador consegue capturar aspectos dinâmicos da execução dos *jobs*, tais como, diversas tarefas (alocadas dinamicamente) irem para um mesmo nó (*burst*) e, dentro deste nó, várias tarefas irem para o mesmo recurso ou um recurso ficar sub-utilizado (inanição). Em termos de grau de abstração, o simulador está em uma posição intermediária, entre o sistema real e o modelo analítico. O simulador permite analisar mais cenários que os experimentos, pois, apesar de ser executado milhares de vezes, leva menos tempo para processar, permitindo avaliar as premissas do modelo em uma combinação maior de parâmetros. Note que o simulador representa a execução do *job* na rede de filas e não o sistema, buscando capturar suas principais características. Nesta validação foi comparado as métricas entre o simulador da rede de filas e o modelo analítico. O erro relativo foi calculado em função do simulador, como mostrado na equação 5.1, ou seja, calcula-se o módulo da diferença entre o tempo de resposta do modelo para o simulador, sobre tempo de resposta do simulador.

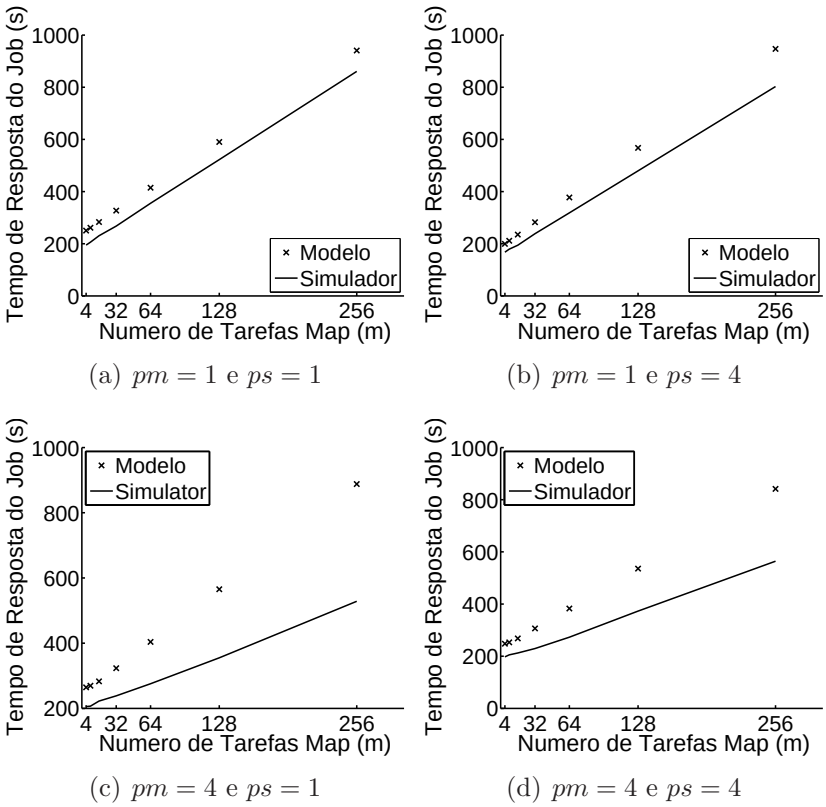
$$R_t = \frac{|R_{t,sim} - R_{t,\triangle}|}{R_{t,sim}} \tag{5.1}$$

A Tabela 5.3 apresenta os parâmetros utilizados na validação por simulação. Montamos experimentos balanceados com quatro nós ($n = 4$), cada nó contendo quatro CPU's ($c = 4$) e um disco ($d = 1$). Note que para poder aumentar o número de tarefas *map* na potência de dois, consideramos quatro nós ao invés de três (ambiente utilizado na coleta das demandas). Foram analisadas as mesma combinações do parâmetro pm, no entanto, o parâmetro ps foi variado em $ps = 1$ e $ps = 4$ (múltiplo de dois), ao invés de $ps = 5$ (utilizado no experimento). Utilizamos as demandas coletadas para cada configuração, no entanto, para avaliar cenários balanceados, consideramos a média das demandas de CPU dos três nós coletados, para as tarefas cujas demandas de CPU variam. Como as demandas coletadas foram medidas por tarefa, variamos o número de tarefas *map* utilizando as mesmas demandas. Isto equivale a aumentar o tamanho do arquivo e consequentemente o número de *splits*. Variamos o número da tarefas *map* (dobrando) de 4 à 256 para ver a acurácia do modelo desde cenários mais simples a cenários mais complexos. Os resultados do simulador são médias de 5.000 execuções com intervalo de confiança de 95%. Os intervalos de confiança mostraram que os resultados de replicações individuais desviam da média no máximo 1.04% com 95% de confiança. As métricas de desempenho avaliadas foram tempo de resposta do *job*, das tarefas, utilização dos recursos. Como consideramos que a carga possui apenas um *job* (MPL 1), o *throughput* de *jobs* não é uma métrica tão interessante e seu decaimento pode ser visto através da utilização do centro de serviço gargalo, no caso o disco.

Tabela 5.3. Demandas medidas no experimento real

Notação	**Parâmetro**	**Valores utilizados**
pm	Número máximo de *maps* paralelos por nó	1, 4
pr	Número de *maps* em paralelo no nó	1
ps	Número de *shuffles* em paralelo	1, 4
m	Número de *maps*	4, 8, 16, 32, 64, 128, 256
r	Número de *reduces*	4
n	Número de nós	4
c	Número de CPU/nó	4
d	Número de discos/nó	1

A Figura 5.5 apresenta a validação do tempo de resposta médio do *job*, principal métrica avaliada. No primeiro experimento há apenas um *map* em paralelo por nó ($pm = 1$) e onde cada *shuffle* agrega apenas um *map*, havendo portanto m *maps*, onde m é o número de *maps*. Este é um cenário que exibe pouco paralelismo, podendo haver um pouco de contenção entre os *maps* e os *shuffles*. A curva do modelo analítico ficou super-estimada quando há poucas tarefas *map*, erro relativo máximo de 29%, mas



(a) $pm = 1$ e $ps = 1$ (b) $pm = 1$ e $ps = 4$

(c) $pm = 4$ e $ps = 1$ (d) $pm = 4$ e $ps = 4$

Figura 5.5. Validação do tempo médio de resposta do *job*.

apresentou um bom acordo à medida que aumentamos o número de tarefas (erro abaixo de 15%). Pode-se observar que o modelo capturou bem a contenção da CPU e do disco, no entanto não capturou bem a utilização de fibra e rede. Verificamos que a estimativa do tempo de resposta médio das tarefas *map* e *merge* (que não possui contenção, dado pela demanda) pelo modelo de rede de filas foram bem estimados. Entretanto o tempo de resposta dos *shuffles* em cenários com poucos *maps* foi sub-estimado em até 41%.

No segundo experimento realizado aumentamos o número de *maps* agregados pelo *shuffle* mantendo todos os outros parâmetros iguais ao experimento anterior. O aumento do número de *maps* agregados significa que o *shuffle* inicia após *ps maps* terminarem de processar e consome os *ps maps* de uma só vez, utilizando *ps threads*. Observamos que o modelo apresentou um bom acordo em relação ao simulador na estimativa do tempo médio de resposta do *job*, erro máximo de 19% (m=4). Assim, como no experimento anterior a utilização de CPU e disco foram bem estimadas. Apesar do modelo não ter produzido uma boa estimativa para a utilização de rede, a rede não é utilizada pelos *maps* nem pelos *merge*, sendo utilizada apenas pelo *shuffle*, levando a uma estimativa do *shuffle* sub-estimada. Entretanto o tempo de resposta do *shuffle* não influência muito o tempo de resposta do *job* neste cenários

No terceiro experimento aumentamos o número de *maps* em paralelo por nó, ou

seja o número de *threads* disponíveis para processar tarefas *map*. Como cada nó possui quatro CPU's haverá pouca contenção na CPU, entretanto poderá haver formação de filas em outros recursos não só entre *maps* e *shuffles*, como nos cenários anteriores, mas também entre *maps*. Pode-se observar que à medida que aumentamos o número de tarefas, o modelo super-estima muito o tempo de resposta do *job*. O modelo capturou uma contenção excessiva na utilização do disco (gargalo).

Para o quarto experimento aumentamos o número de *maps* agregados pelo *shuffle*, mantendo os parâmetros anteriores. Pode-se observar que, assim como no experimento anterior, o modelo super-estimou o tempo de resposta médio do *job*, consequentemente sub-estimando as utilizações.

5.4.3 Introdução do Operador Fork/Join

Observamos na validação por simulação, que o tempo de resposta médio das tarefas foi bem estimado pelo modelo de rede de filas, com exceção do *shuffle*, que não afeta muito o tempo médio de resposta do *job*, entretanto os tempos de sincronizações entre estas tarefas, calculado pelo nós internos (operadores) da árvore de precedência não foram bem capturados, levando a previsões de desempenho super-estimadas. Sendo assim, esta Seção introduz um novo operador primitivo, usado na composição do tempo de resposta do *job* pela árvore que utiliza o método de redes de filas *fork/join* para estimar o tempo de sincronização entre um sub-conjunto de tarefas paralelas.

O método de redes de filas *fork/join* foi inicialmente proposto por Varki (1999) e aplicado para modelar o tempo de execução de uma requisições a um *array* de discos, como representado na Figura 5.6. Uma requisição ao controlador dos discos faz uma sub-requisição a cada um dos k discos e completa quando todos os discos terminarem de executar. O tempo de sincronização corresponde ao tempo que cada sub-requisição deve esperar nos pontos de *join*.

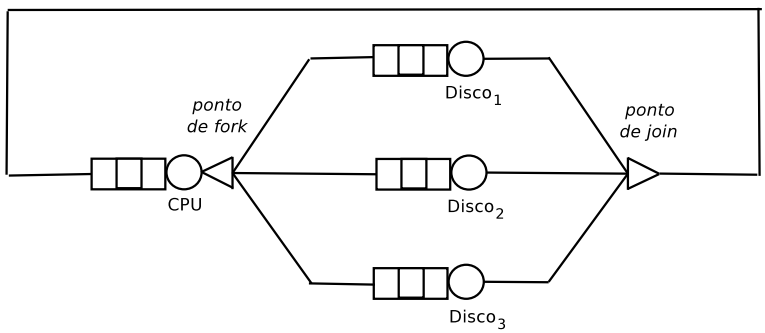


Figura 5.6. Modelo de uma rede de fila *fork/join* (Menascé et al., 2004)

Sendo assim, o tempo de execução de uma requisição a um controlador de um *array* de discos é dado pela média do máximo dos tempos de residência nos discos. O tempo de execução em cada disco são variáveis aleatórias (v.a.) com uma determinada distribuição, logo, o tempo de uma requisição ao *array* de discos é dado pelo máximo da média de k v.a.s. Note que a média do máximo de k v.a.s. é diferente do máximo da média de k v.a.s., pois no primeiro o cálculo leva em consideração a variâncias das v.a. no cálculo dos máximos, produzindo estimativas muito maiores.

A solução exata para calcular a média do máximo do tempo de residência de k tarefas é computacionalmente intensiva (Balakrishnan, 1994), pois requer calcular a convolução entre as distribuições das k tarefas. Varki (1999) derivou as equações para estimar a média do máximo de k requisições a uma rede de filas balanceada, como a um *array* de discos, onde cada disco possui o mesmo tempo de acesso (mesma demanda) e o tempo de residência das requisições é dado por sua demanda somada a seu tempo de sincronização.

A estimativa do tempo de sincronização de k requisições é dado pelo produto do k-ésimo número harmônico (H_k), apresentado na equação 5.2 com a demanda média dos k discos. A aproximação vêm da esperança de uma variável aleatória definida como o máximo de k v.a. exponencialmente distribuídas. Os autores (Varki, 1999) mostraram que esta aproximação obteve um erro relativo menor que 5% para maioria dos casos analisados.

$$H_k = \sum_{i=1}^{k_i} \frac{1}{i} \tag{5.2}$$

A solução adotada nesta dissertação consiste em utilizar o método proposto por Varki (1999) para estimar o tempo de sincronização de um sub-conjunto de tarefas em paralelo, ou seja, a média do máximo do tempo de resposta de k tarefas em paralelo. A estimativa é dada pelo produto do k-ésimo número harmônico com o máximo das médias dos tempos de resposta das k tarefas. Note que as tarefas podem ter diferentes tempos de resposta, mas assumimos (aproximação) que todas as tarefas têm um tempo de resposta igual ao maior tempo de resposta das tarefas, o que provê um limite superior para esta estimativa.

5.4.4 Avaliação das Limitações do Modelo

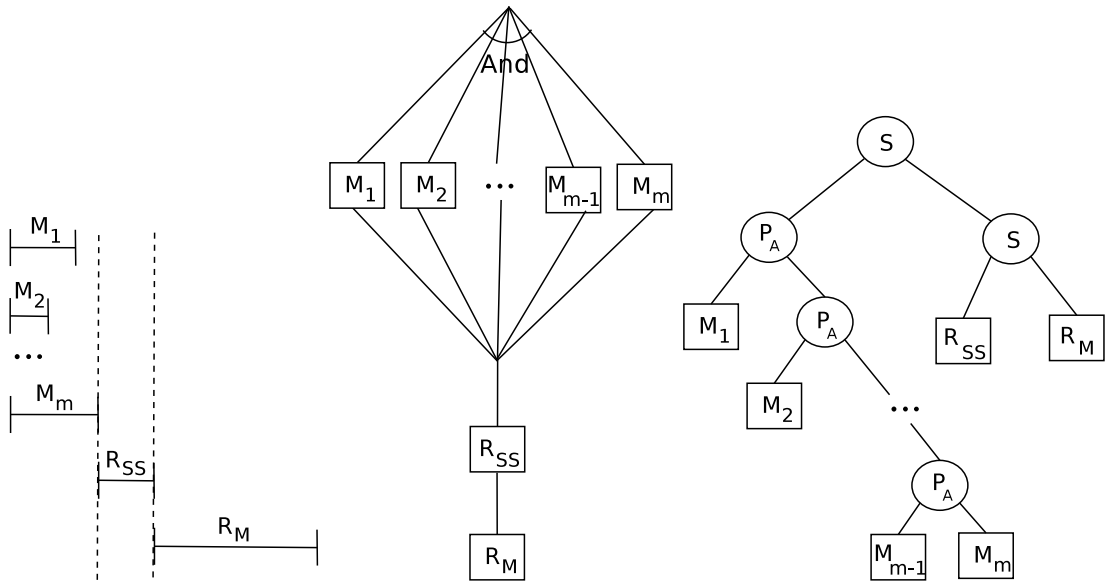
O maior desafio encontrado na modelagem do paralelismo *pipeline* foi na estimativa do operador paralelo (P), que estima a média do máximo de um sub-conjunto de tarefas (variáveis aleatórias) em paralelo, cada estágio do *pipeline*. Lembrando que a

média (esperança) dos máximos é maior que o máximo das médias, pois no cálculo da média do máximo leva-se em conta a variância da distribuição de probabilidade, o que torna esta previsão difícil de ser feita. Sendo assim, foi feito uma avaliação específica da modelagem da previsão do máximo buscando avaliar a premissa do modelo referência que considera que o tempo de resposta das tarefas é exponencialmente distribuídos. Esta premissa possui várias implicações como veremos nesta Seção.

Foi avaliado um cenário *naive* do *Hadoop* original, onde as fases *map* e *reduce* são seriais. Para fazer com que as fases *map* e *reduce* sejam sequenciais consideramos que o número de *maps* agrupados (ps) pelo *shuffle-sort* (ss) é igual ao número total de *maps* ($ps = m$), ou seja, um único ss consome todos os *maps* processados. Sendo assim, o *reduce* só inicia após os m *maps* terminarem de processar. Dessa forma haverá apenas um *shuffle-sort* (ss) e um *merge*. O tempo médio de resposta do *job* é dado por:

$$E[\Re_{job}] = E[max(\Re_{map_i})] + E[\Re_{ss}] + E[\Re_{merge}] \tag{5.3}$$

Os cenários do *Hadoop* original possui relações de precedência bem mais simples que o *HOP*. Na modelagem do HOP, como as relações de precedências podem mudar de acordo com o tempo de resposta das tarefas, foi preciso construir uma árvore de composição a cada iteração do algoritmo. Para os *jobs* do *Hadoop* original é possível utilizar uma árvore de composição fixa, como ilustrado na Figura 5.7, onde é mostrado o mapeamento do grafo de tarefas do fluxo de execução do *Hadoop* para a árvore de composição. A utilização de uma árvore fixa ajuda na convergência do algoritmo iterativo do modelo. A média do máximo do tempo de resposta das tarefas é utilizada para capturar o tempo que cada tarefa fica esperando no ponto de sincronização (barreira) pelo término das outras tarefas.



(a) Linha do Tempo (b) Grafo de Tarefas (c) Árvore de Precedência

Figura 5.7. Fluxo de execução de *jobs* do *Hadoop*

Para avaliar a aproximação da estimativa do máximo das médias das tarefas foram realizados três experimentos, todos partindo da mesma configuração: (1) o primeiro é um teste de sanidade onde consideramos que o tempo de resposta é dado pela demanda da tarefa; (2) em seguida foi feito um teste com o tempo de resposta dado pelo tempo de residência em todos os centros (com fila); e (3) por fim foi feito uma avaliação em um cenário onde o tempo de resposta é dado pelo tempo de residência de um recurso.

5.4.4.1 Configuração dos Experimentos

As demandas utilizadas foram demandas medidas por experimentos reais no cenário $pm = 1, ps = 1$ da Seção 5.4.1. Na comparação dos resultados com o modelo analítico foi utilizado dois métodos para estimar a média dos máximos: (1) o método do modelo referência, que aproxima a distribuição dos *jobs* para *Erlang* ou hiper-exponencial de acordo com seus coeficientes de variação; (2) e o método do *fork/join*, solução exata para tarefas balanceadas, que utiliza o i-ésimo número harmônico para estimar o tempo de sincronização.

A Tabela 5.4 apresenta os parâmetros utilizados nesta avaliação. Na configuração da arquitetura utilizou-se apenas um nó para que não haja múltiplas instâncias do *shuffle-sort*. Consideramos o número de CPU's igual ao número de *maps*, para que os *maps* não peguem contenção na CPU. Fixamos o número de tarefas *reduces* em um e variamos o número de tarefas *maps* (m), conjuntamente com os parâmetros número de *maps* paralelos (pm) e número de *maps* agrupados pelo *shuffle-sort* (ps).

Tabela 5.4. Parâmetros utilizados no experimento.

Notação	Parâmetro	Valores utilizados
n	Número de nós	1
m	Número de *maps*	{ 1, 4, 8, 16, 32, 64 }
T	Número de tipos de tarefas	3 (*map*, *ss* e *merge*)
R	Número de tipos de recurso	4 (CPU, fibra, disco e rede)
C_t	Número de tarefas de cada tipo	{ m, 1, 1 }
K_r	Número de recursos de cada tipo	{ m, 1, 1, 1}
pm	# de *maps* em paralelo por nó	m
ps	# de *maps* agrupados pelo *ss*	m
Rt	Tempo de resposta da tarefa t	Cenário $pm1ps1$ (Cap. 4)
Rtk	Tempos de residência da tarefa t	Cenário $pm1ps1$ (Cap. 4)
Dt	Demanda da tarefa t	Cenário $pm1ps1$ (Cap. 4)
E	# de execuções da simul. simplificada	1.000.000
E'	# de execuções da simul. detalhada	5.000

5.4.4.2 Tempo de Resposta Sem Fila

O primeiro teste realizado foi um teste de sanidade onde consideramos que as tarefas *map* não possuem fila, ou seja, que o tempo de resposta é dado pela soma de suas demandas (exponenciais). Sendo assim, consideramos apenas a demanda de CPU, pois neste cenário cada *map* possui uma CPU exclusiva ($c = m$). Foi desenvolvido um simulador bem simples para avaliar a aproximação da média do máximo do tempo de resposta dos *maps*, descrito no Algoritmo 13. Este simulador simplificado toma como entrada a média do tempo de resposta das tarefas (*map*, *ss* e *merge*), obtidos do simulador detalhado e gera números aleatórios exponencialmente distribuído para cada variável aleatória $\Re_{map_i}$, $\Re_{ss}$ e $\Re_{merge}$. A cada iteração é calculado a média do máximo dos *maps* dinamicamente, assim como a média do tempo de resposta do *job*.

Algoritmo 3: Simulador-Ri(R_t, C_t, E, T)

```
1  mj = −1.0
2  M_t = [ ]_{1×T}
3  para e = 1 até E faça /* para cada execução e */
4  |  j = 0.0
5  |  para t = 0 até T faça /* para cada tipo de tarefa t */
6  |  |  max = −1.0
7  |  |  para i = 0 até C_t[t] faça /* para cada tarefa i do tipo t */
8  |  |  |  ri = (−R_t[t]) · Log( Rand() ) /* gera nº aleatório exponencial */
9  |  |  |  se ri > max então /* procura o máx dos tempos de resposta*/
10 |  |  |  |  max = ri
11 |  |  j+ = max /* acumula o tempo do job */
12 |  |  M_t[t] = ((e − 1) · M_t[t] + max) / e /* calcula a média do máx das tarefas */
13 |  mj = ((e − 1) · mj + j) / e /* calcula a média do máximo do job */
```

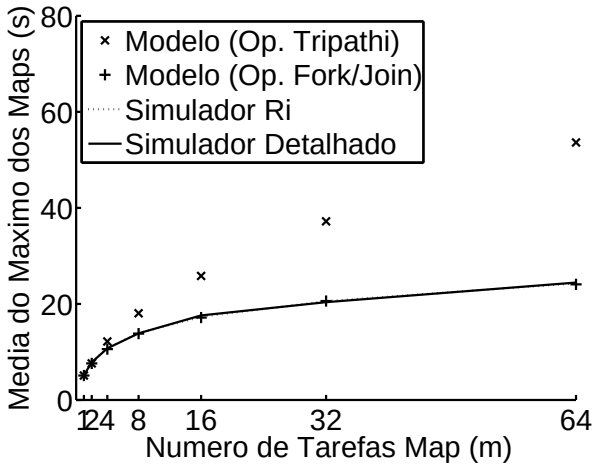


Figura 5.8. Gráfico de validação da média dos máximos (sem fila).

A Figura 5.8 apresenta as curvas do simulador simplificados, dos dois métodos do modelo e do simulador detalhado (*baseline*). Pode-se observar que a curva da estimativa do máximo das médias dos *maps* do simulador simplificado e do modelo com o método do *fork/join* tiveram um bom acordo com o simulador detalhado. Note que, como o modelo referência assume que as tarefas são exponencialmente distribuídas, era esperado o modelo utilizando apenas os operadores do modelo referência ficasse próximo do simulador do tempo de resposta. Entretanto, mesmo em experimentos sem fila, este modelo tende a super-estimar o tempo de resposta, onde o erro relativo máximo foi de 119%, com $m = 64$. Como neste experimento não há fila, o modelo convergiu em apenas uma iteração para todos os cenários. Este experimento mostrou que a aproximação do simulador simplificado em relação ao simulador detalhado indica que o erro da aproximação do tempo de resposta exponencialmente distribuído está na estimativa das filas, pois as demandas estão sendo bem estimadas.

Observamos que não é possível fazer um simulador da demanda e do tempo de fila como duas variáveis aleatórias exponenciais independentes pois elas são correlacionadas. Se o recurso for *processor sharing* (PS) o número de vezes que as tarefas vão circular pela filas é dado pelo tamanho da fila e pelo número de visitas (dado pelo *quantum* e por sua demanda). Nestes experimentos todos os recursos têm disciplina PS. E mesmo se for um recurso FCFS, o aumento na demanda, aumenta o tamanho de fila das outras tarefa. Então mantivemos nossa avaliação em cima do tempo de resposta.

5.4.4.3 Tempo de Resposta Com Fila

O segundo experimento utilizou os mesmos parâmetros do experimento anterior, no entanto, sem zerar as demandas de fibra e disco para as tarefas *map*. Desta forma, a medida que aumentarmos o número de *maps* haverá formação de fila no barramento de dados para o disco (canal de fibra), cuja demanda é muito pequena, e principalmente no único disco deste nó, sendo portanto o gargalo do sistema. Foi desenvolvido um segundo simulador simplificado, apresentado no Algoritmo 16, similar ao simulador anterior, no entanto levando em consideração a variância dos tempos de residência, considerando que a esperança de uma variável aleatória (v.a.) definida como o tempo de resposta de uma tarefa, $E[\Re]$, é dado pela soma das v.a.s exponenciais (com médias diferentes) referentes ao tempo de residência em cada recurso (assumindo que uma tarefa não executa em mais de um recurso ao mesmo tempo), formando assim uma distribuição hiper-exponencial. Essa simulação foi realizada para verificar se utilizar uma única variável aleatória exponencial é uma boa aproximação para uma série de

exponenciais, ou seja: $E[R_i] \approx E[\sum_{k=1}^{K} R_{ik}]$.

Algoritmo 4: Simulador-Rik(R_{tk}, C_t, E, T, K)

```
1  mj = −1.0
2  M_t = [ ]_{1×T}
3  para e = 1 até E faça /* para cada execução e */
4  |  j = 0.0
5  |  para t = 0 até T faça /* para cada tipo de tarefa t */
6  |  |  max = −1.0
7  |  |  para i = 0 até C_t[t] faça /* para cada tarefa i */
8  |  |  |  ri = 0.0
9  |  |  |  for k = 0 até K do /* para cada centro k */
10 |  |  |  |  rik = (−R_tk[t][k]) · Log( Rand() ) /* gera nº aleatório exp. */
11 |  |  |  |  ri+= rik
12 |  |  |  se ri > max então /* procura o máx dos tempos de resposta*/
13 |  |  |  |  max = ri
14 |  |  j+ = max /* acumula o tempo do job */
15 |  |  M_t[t] = ((e − 1) · M_t[t] + max) / e /* calcula a média do máx das tarefas */
16 |  mj = ((e − 1) · mj + j) / e /* calcula a média do máximo do job */
```

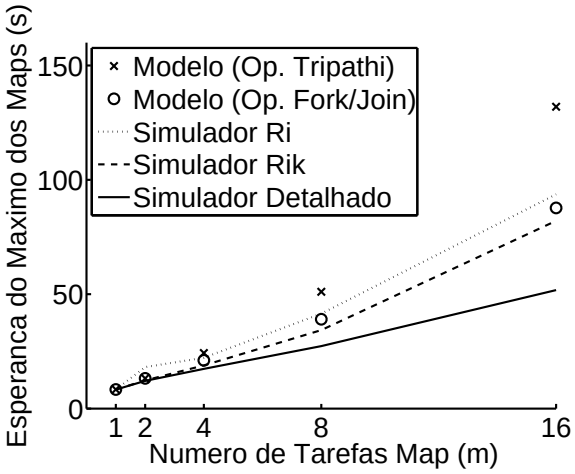


Figura 5.9. Gráfico de validação da média dos máximos (com fila).

As curvas dos dois modelos simplificados (tempo de resposta e residência), do simulador (*baseline*) e dos dois métodos para estimar o máximo das médias para este cenário foram apresentados na Figura 5.9. Pode-se observar que os dois métodos de estimativa do máximo das médias do modelo e os dois simuladores super-estimaram o máximo do simulador detalhado. Em todos os cenários, o modelo convergiu em menos de 6 iterações. O método *fork/join* ficou próximo do simulador do tempo de resposta. Verificamos que a esperança da soma de m v.a. exponenciais (tempos de residência)

com médias diferentes (hiper-exponencial), ficou menor que a esperança de uma única v.a. exponencial (tempo de resposta) com média igual a soma das médias das m v.a.s.

Foi observado que a utilização do disco (gargalo) já estava 100%, ou seja, as filas deveriam supostamente crescer exponencialmente. Vimos que houve um crescimento exponencial na utilização do simulador detalhado, aumentando o tamanho das filas, mas não com a mesma variância que sua demanda (exponencial), para poder correlacionar Di+Wi em uma única variável aleatória exponencial, Ri, ou correlacionar soma(Dik+Wik) em uma série de exponenciais, soma(Rik). Aqui percebemos uma primeira limitação da premissa de assumir que o tempo de resposta das tarefas são exponenciais associada com a utilização: para o tempo de fila crescer exponencialmente é necessário que a utilização esteja próximo de 100%, o que não ocorre em cenários de carga leve, levando a estimativas super-estimadas das variâncias da v.a. aleatória tempo de resposta com média igual a soma da demandas com a média do tempo de fila. Esta relação não se aplica a este cenário, pois o mesmo possui utilização 100%, mas se aplica a vários experimentos anteriores. Sendo assim, concentração na avaliação da variância do tempo de fila.

5.4.4.4 Tempo de Resposta Considerando Apenas o Disco

Foi realizado um terceiro experimento onde primeiramente zeramos as demandas de CPU e fibra, para garantir que todas as tarefas cheguem ao disco ao mesmo tempo (*burst*), evitando as variâncias do tempo de residência nestes centros antes de chegar no disco. Focamos nossa avaliação no tempo de residência do disco (gargalo). Sendo assim, o tempo de resposta das tarefas *map* é dado pelo tempo de residência do disco, $R_i = R_{i,disco} = D_{i,disco} + W_{i,disco}$, logo não utilizamos o simulador do tempo de residência.

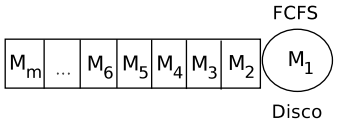


Figura 5.10. Representação da fila do disco com disciplina FCFS.

A Figura 5.10 representa o centro de serviço disco. Considerando que todas as tarefas *map* chegam ao disco ao mesmo tempo, o máximo tempo de residência é dado pelo tempo de residência da última tarefa a chegar no centro (M_m). Isto ocorre independente se a disciplina do centro é FCFS ou PS. FCFS é mais intuitivo, no caso de PS pode-se compreender melhor considerando que a demanda de cada tarefa se divide em várias visitas em função do *quantum* e que a última tarefa deverá esperar todas as visitas das tarefas (alternando entre as tarefas), incluindo suas próprias, equivalendo

ao FCFS. Para simplificar as análises passamos a disciplina do disco de PS para FCFS (setando o *quantum* para infinito). Então, a média de uma v.a. definida como o máximo do tempo de residência no disco de todas as tarefas é dado pela média da soma do tempo que a última tarefa espera pelas $m-1$ tarefas (v.a.s exponenciais com média diferentes) mais o tempo de sua demanda (v.a. exponencial), ou seja:

$$E[X_m] = E[X_1 + X_2 + ... + X_m] = E[\sum_{i=1}^{m} X_i] \tag{5.4}$$

Foi desenvolvido um terceiro simulador bem simples, apresentado no Algoritmo 12, que toma como entrada a demanda de cada tipo de tarefa (*map*, *ss* e *merge*), o número de tarefas de cada tipo, o número de execuções do experimento e o número de tipos de tarefa. A cada iteração do simulador são gerados m demandas exponencialmente distribuídas, calcula-se o máximo do tempo de resposta dado pela soma das m demandas, seguido do cálculo da média do máximo das tarefas e do *job*.

Algoritmo 5: Simulador-Fcfs(D_t, C_t, E, T)

```
1  mj = −1.0
2  M_t = [ ]_{1×T}
3  para e = 1 até E faça /* para cada execução e */
4  |  j = 0.0
5  |  para t = 0 até T faça /* para cada tipo de tarefa t */
6  |  |  max = −1.0
7  |  |  para i = 0 até C_t[t] faça /* para cada tarefa i */
8  |  |  |  di = (−D_t[t]) · Log( Rand() ) gera nº aleatório exp. */
9  |  |  |  max+ = di
10 |  |  j+ = max /* acumula o tempo do job */
11 |  |  M_t[t] = ((e − 1) · M_t[t] + max) / e /* calcula a média do máx das tarefas */
12 |  mj = ((e − 1) · mj + j) / e /* calcula a média do máximo do job */
```

Antes de analisar a média do máximo das tarefas verificamos que a média do tempo de resposta das tarefas *map* (sem ser o máximo) foram bem estimados pelo modelo analítico, erro relativo médio de 7.7% para os dois métodos de estimativa do máximo (*fork/join* e modelo referência). Estas estimativas são utilizadas como parâmetros de entrada para a composição da média do tempo de resposta do *job* no modelo. Verificamos que o modelo de rede de filas capturou bem a contenção do disco.

A Figura 5.11 apresenta as curvas dos dois métodos de estimativa do máximo utilizados no modelo, dos simuladores simplificados do tempo de resposta e da fila FCFS e do simulador detalhado (*baseline*). Primeiramente, observamos que o simulador da fila FCFS teve um bom acordo com a curva do simulador detalhado. As curvas do modelo simplificado do tempo de resposta exponencial e dos dois métodos de estimativa

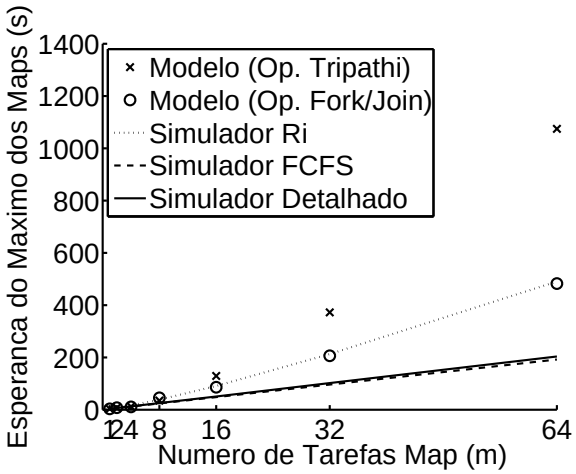


Figura 5.11. Gráfico da validação da média dos máximos, apenas com o disco.

do máximo super-estimaram a média do máximo do simulador detalhado. O método do modelo referência super-estimou muito a média do máximo (erro relativo médio de 427%), enquanto que o método do *fork/join* se ajustou a curva do simulador do tempo de resposta. Sendo assim concluímos que, por mais que a demanda seja exponencial o tempo de fila é dado por uma série de m exponenciais com médias diferentes, ou seja, hipo-exponencial, e a média (esperança) de uma série de exponenciais, como vimos na análise do simulador de R_{ik}, na Seção anterior, é muito menor que uma v.a. exponencial com média dada pela soma das médias da série.

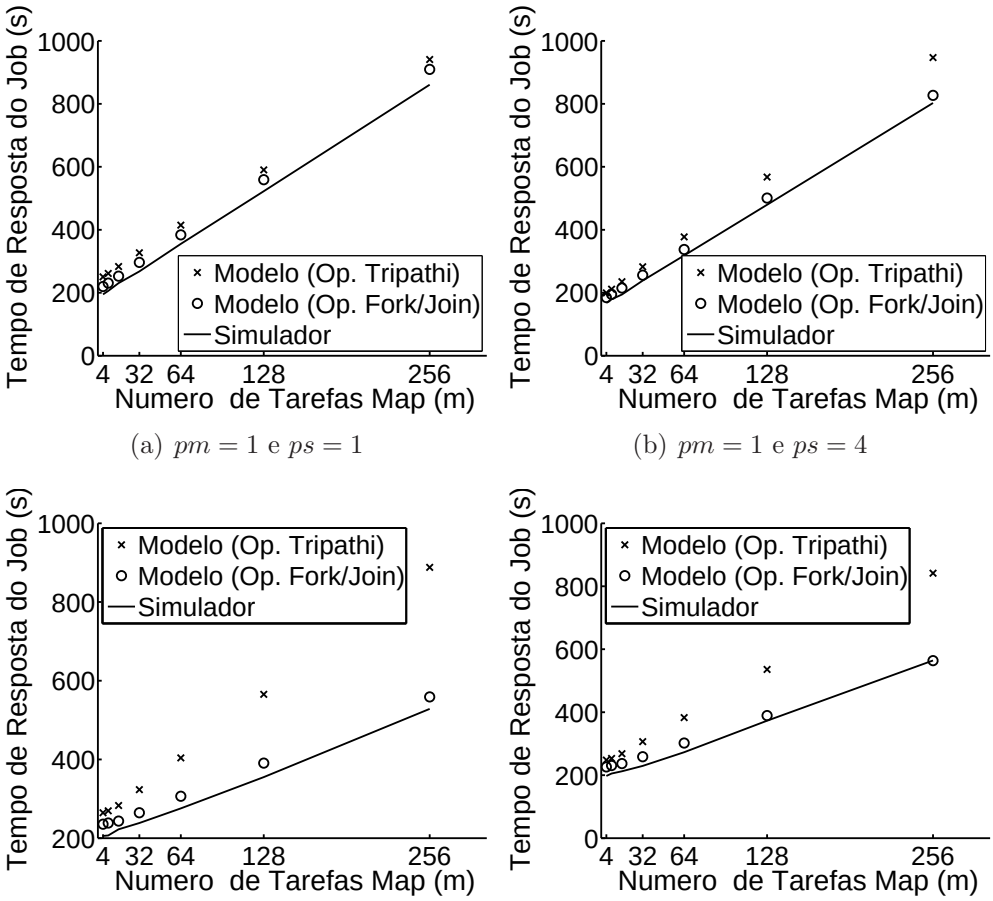
A maioria dos modelos analíticos não levam em consideração as distribuições de probabilidade na estimativa da média (esperança) do tempo de resposta das tarefas (Raatikainen, 1989). A maior parte se baseia na premissa *overtaken-free condition*, que diz que qualquer tarefa que chegue após outra tarefa sempre estará atras dela na fila, o que garante que os tempos de residência de centros de serviço diferentes (com disciplina FCFS ou *delay centers*) sejam independentes. No caso de *delay centers* a esperança do tempo de residência é dado pela esperança do tempo de serviço (em geral, aproximados para uma exponencial). A estratégia utilizada por alguns métodos aproximados (Salza e Lavenberg, 1981; Lazowska e C., 1979; Rege e Sengupta, 1988), que consideram as distribuições, para estimar a distribuição do tamanho das filas em centros com disciplina FCFS e PS é substituir um sub-conjunto de centros de serviço por um centro exponencial dependente da carga (*load dependent*), que é analisado para estimar a esperança do tempo de residência. (Raatikainen, 1989) propôs um método baseado na transformada de Laplace, que estima a esperança do tempo de resposta pela esperança da soma de seus tempos de residência. Para estimar a esperança do número de visitas utilizam uma função de probabilidade disjunta (*z-transform*).

Quantis e percentis do tempo de resposta são estimados através de inversões numéricas da transformada de Laplace. Estes trabalhos nos atentaram para o fato de que a esperança do tamanho da fila de um centro de serviço com utilização 100% não cresce exponencialmente, mas sim seguindo uma distribuição dada pelo tempo de fila da última tarefa a chegar na fila, que é dado por uma série das distribuições das demandas das tarefas encontradas em sua frente na fila.

Assim como foi verificado nesta dissertação, (Jonkers, 1994a; Adve e Vernon, 1993) avaliaram as limitação da premissa de assumir que os tempos de resposta das tarefas são exponencialmente distribuídos. Essa premissa foi proposta inicialmente em (Salza e Lavenberg, 1981) e reavaliada em (Heidelberger e Trivedi, 1983) e se baseia em que: se o tempo de resposta de uma tarefa é dado pelo período que a tarefa cicla pela rede de fila com probabilidade p até que saia com probabilidade $1 - p$, e se p é próximo de 1, então o tempo de reposta da tarefa pode ser aproximado para uma exponencial. Embora tempos de resposta de tarefas com pouca variância possam ser modelados desta forma, esta aproximação, avaliada em um estudo de caso (Jonkers, 1993), implica em assumir que o número de passadas é determinado pela probabilidade de circular pela rede de filas (*loop bounds*). Verificou-se em (Jonkers, 1993), através de experimentos utilizando redes de filas separáveis, redes de petri e técnicas híbridas que, em seções paralelas (como um *fork/join*) o tempo de resposta das tarefas é determinado pela média do máximo de passadas ao invés da média de passadas, o que leva a severas super-estimativas do tempo de resposta, pois na realidade o número de passadas é constante, ou seja, a média e o máximo são iguais.

5.4.5 Validação por Simulação com o Operador Fork/Join

Nesta Seção repetimos os mesmos experimentos realizados na Seção 5.4.2 utilizando o operador *fork/join*, introduzido na Seção 5.4.3, que na avaliação da limitação do modelo, Seção 5.4.4, apresentou melhores estimativas para as métricas de desempenho. Lembrando que o operador *fork/join* é utilizado para estimar o tempo de sincronização de um sub-conjunto de tarefas na composição do tempo médio de resposta do *job* pela árvore de precedência. Para permitir a comparação também mostramos os resultados da validação anterior, onde a árvore era construída utilizado apenas os operadores propostos pelo modelo referência. A Figura 5.12 apresenta os resultados da validação por simulação do tempo médio de resposta do *job* estimado pelo modelo analítico, utilizando o operador *fork/join*, para os quatro cenários avaliados.



(a) $pm = 1$ e $ps = 1$

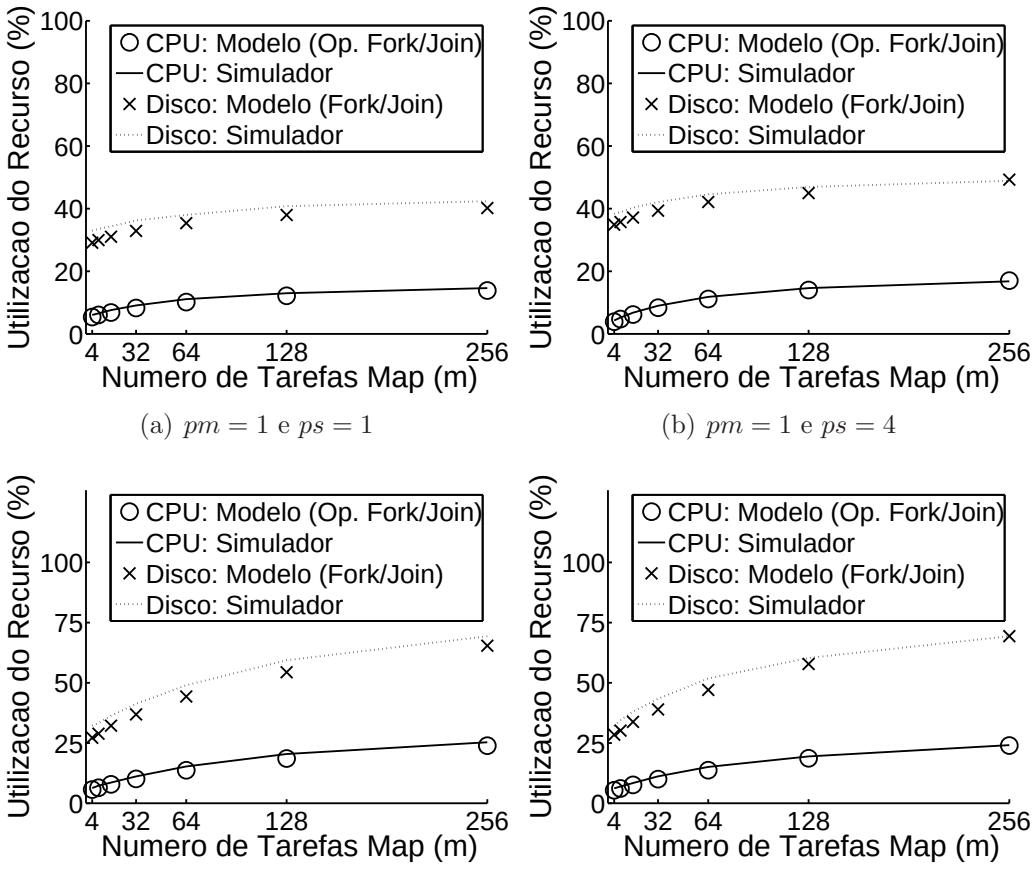
(b) $pm = 1$ e $ps = 4$

(c) $pm = 4$ e $ps = 1$

(d) $pm = 4$ e $ps = 4$

Figura 5.12. Validação do tempo médio de resposta do *job*.

Para os cenários que não há paralelismo entre os *maps* ($pm = 1$) o modelo analítico baseado no *Fork/Join*, assim como o modelo baseado no modelo referência (resultados anteriores), apresentou boas estimativas de desempenho, ficando o erro relativo máximo abaixo de 13%. O modelo analítico está sendo conservador, uma vez que, neste caso, é melhor errar para mais do que para menos. Pode-se observar que o modelo capturou bem a contenção da CPU e do disco, no entanto não capturaram bem a utilização de fibra e rede. Verificamos que a estimativa do tempo de resposta médio das tarefas *map* e *merge* pelo modelo de rede de filas foram bem estimados. Assim, a utilização de CPU e disco, apresentadas na Figura 5.13, foram bem estimadas pelo modelo utilizando o método do *fork/join*. Apesar de ambos os métodos não produziram uma boa estimativa para a utilização de rede, a rede não é utilizada pelos *maps* nem pelos *merge*, sendo utilizada apenas pelo *shuffle*, levando a uma estimativa do *shuffle* sub-estimada. Entretanto o tempo de resposta do *shuffle* não influencia muito o tempo de resposta do *job* neste cenários



(a) $pm = 1$ e $ps = 1$

(b) $pm = 1$ e $ps = 4$

(c) $pm = 4$ e $ps = 1$

(d) $pm = 4$ e $ps = 4$

Figura 5.13. Validação da utilização média dos recursos.

Os cenários que apresentam paralelismo entre as tarefas *maps* ($pm = 4$), o modelo analítico utilizando os operadores *fork/join* produziu melhores resultados, com uma diferença relativa máxima abaixo de 15%, enquanto que o erro relativo máximo do modelo utilizando apenas os operadores propostos no modelo referência havia super-estimado muito suas previsões de desempenho, chegando a 68%. Novamente o modelo analítico foi conservador ("errou para mais"). Assim como no experimento anterior estimou bem a contenção da CPU e do disco, mostrados na Figura 5.13, mas não estimou bem a utilização da fibra e da rede. Como a demanda da fibra é muito pequena esta diferença não tem grande impacto. A rede, da mesma forma, só é utilizada pelo *shuffle* e na influi muito no tempo de resposta do *job*. Note que, enquanto na Figura 5.12, do tempo médio de resposta do *job*, o modelo analítico está acima do simulador, na Figura 5.13, da utilização do recursos, ele está abaixo do simulador. Isto ocorre pois tanto no modelo quanto no simulador o tempo em que os recursos estão sendo efetivamente utilizados pelas tarefas, dado por sua demanda, são aproximadamente os mesmos, o que varia é a estimativa do tempo de fila, onde o modelo produziu estimativa

maiores, o que implica em utilização de recursos menores.

5.4.6 Validação por Experimentos Reais

Esta Seção apresenta os resultados da validação do modelo analítico através de experimentos reais. Foi utilizado a mesma configuração descrita na Seção 5.4.1. Ao contrário da validação por simulação, nos experimentos foi considerado o desbalanceamento da demanda de CPU em cada nó. As métricas avaliadas foram o tempo médio de resposta do *job* e os tempos médio de resposta de cada tarefa. Note que, no modelo, os tempos médio de resposta das tarefas são estimados pelo modelo de rede de filas (modelo físico) enquanto que o tempo médio do *job* é estimado através da árvore de composição (modelo lógico) que toma os tempos médios de respostas das tarefas como entrada. O erro relativo foi calculado em função do experimento.

A tabela 5.5 apresenta os resultados da validação do tempo médio de resposta do *job*, enquanto que os resultados do tempo médio de resposta das tarefas são mostrados no Apêndice D. Foram avaliados os seguintes cenários: (1) sem paralelismo nos *maps* e nos *shuffles*, (2) sem paralelismo nos *maps* e com paralelismo nos *shuffles*, (3) com paralelismo nos *maps* e sem paralelismo nos *shuffles* e (4) com paralelismo nos *maps* e nos *shuffles*.

Tabela 5.5. Validação do tempo médio de resposta do *job*.

pm	*ps*	**Exp. (E)**	**Simul. (S)**	**Tripathi (T)**	**Fork/Join (F)**	**S x E**	**T x E**	**F x E**
1	1	722.23	686.06	798.56	732.56	-5%	11%	1%
1	5	605.40	629.99	701.75	659.13	4%	16%	9%
4	1	321.77	405.77	620.01	365.51	26%	93%	14%
4	5	325.58	399.06	597.81	356.20	23%	84%	9%

Nos dois primeiros cenários há apenas uma *thread* para processar tarefas *map* em cada nó. Estes cenários apresentam paralelismo entre *maps* e *shuffle-sorts*, mas não há paralelismo entre as tarefas *map*. Como há quatro CPU's disponíveis ($c = 4$), em cenários que não há paralelismo no *shuffle* ($ps = 1$), haverá no máximo duas sub-tarefas em paralelo em cada nó (um *map* e um *shuffle-sort*), não havendo contenção na CPU. Entretanto, pode haver formação de filas no disco, fibra e rede. Ao aumentar o número de *threads* para processar *shuffles* para cinco ($ps = 5$), o *shuffle* passa a consumir cinco *maps* em paralelo, gerando contenção na CPU e nos outros recursos.

A estimativa do tempo médio de resposta do *job*, quando não há paralelismo nos *shuffles*, utilizando os dois modelos, utilizando os operadores do modelo de Tripathi e do *fork/join*, apresentaram um bom acordo comparado com o experimento, com erros

relativos de 11% e 1%, respectivamente. O erro relativo do tempo de resposta médio do *job* medido no simulador foi 5%.

Enquanto que para os cenários com paralelismo nos *shuffles*, as previsões de desempenho do modelo utilizando os operadores do Tripathi e do *fork/join* produziram boas estimativas, com erros relativos de 16% e 9%, respectivamente. O simulador apresentou um erro relativo de 4%.

Note que o modelo analítico foi conservador, uma vez que suas estimativas foram maiores que a do experimento, pois, neste caso, é melhor errar para mais do que para menos. Observamos que, em cenários desbalanceados, sem paralelismo nos *shuffles* (com pouca contenção), ao contrário da validação por simulação, o método do *fork/join* ficou abaixo do simulador, não sendo uma estimativa tão conservadora em relação ao simulador. Entretanto foi a que mais se aproximou dos experimento real para estimativa do tempo médio do *job*. Ao aumentar o paralelismo nos *shuffles* o modelo com o método *fork/join* voltou a ficar entre o simulador e o modelo com o método do Tripathi.

No segundo conjunto de experimentos, dois últimos cenários, aumentou-se o número de *threads* para processar tarefas *map* para quatro ($pm = 4$), podendo haver além do paralelismo entre *maps* e *shuffles*, um paralelismo entre as tarefas *map*. No último experimento o número de *shuffles* em paralelo foi aumentado para cinco, o que implica em um aumento na demanda dos *shuffles* uma vez que cada *shuffle* deve consumir cinco *maps* de uma só vez e, consequentemente, gerar mais contenção na competição por recursos com os quatro *maps* em paralelo de cada nó.

O modelo utilizando os operadores *fork/join* produziu um bom ajuste comparado com o experimento e o simulador, apresentando um erro relativo de 14% e 10% para $ps = 1$ e $ps = 4$, respectivamente. O tempo médio de resposta medido no simulador apresentou um erro relativo de 26%, o qual ainda é aceito na literatura.

A estimativa do tempo de resposta do *job* pelo modelo baseado no Tripathi, assim como na Seção 5.4.5, foi super-estimado, atingindo uma diferença relativa máxima de 93%. Observamos, assim como no experimento anterior, que o modelo com o método do *fork/join* para estimar a média do máximo das tarefas obteve a melhor estimativa. O modelo com o método do Tripathi não obteve boa aproximação na composição da árvore do paralelismo e super-estimou muito o tempo de resposta do *job*.

A contenção nos recursos foi bem capturada, tanto no modelo utilizando apenas os operadores do Tripathi, quanto no modelo utilizando os operadores *fork/join*, exceto para o *shuffle* que sub-estimou a utilização de rede e de fibra, mas, como as demandas nestes recursos é muito baixa, o impacto no tempo de resposta do *job* foi marginal. Isto indica que o modelo da rede de filas está capturando bem a contenção e árvore de precedência foram bem capturados pelo modelo que utiliza os operadores *fork/join*,

mas que a árvore do modelo utilizando os operadores do Tripathi não está capturando bem os atrasos de sincronização.

5.4.7 Impacto dos Parâmetros no Modelo

Além da aplicação de um modelo de desempenho para avaliar a eficiência de um sistema, é muito comum utilizar modelos analíticos para encontrar a configuração ótima de ajuste dos parâmetros, como por exemplo o número de *threads* para processar uma carga de trabalho (parâmetro pm) ou o número de estágios do paralelismo *pipeline* (parâmetro ps). Esta Seção analisa a influência dos principais parâmetros do modelo. A validação do modelo foi estendida para avaliar o impacto dos parâmetros: o número máximo de *maps* em paralelo de um nó (pm), o número de *maps* agrupados pelo *shuffle-sort* (ps) e o número de nós (n). Foram comparados os resultados do simulador detalhado com as métricas do modelo, utilizando os operadores do *fork/join* e do modelo referência na construção da árvore de precedência. O foco principal foi no tempo médio de resposta do *job*. Estes experimentos foram realizados com o intuito de avaliar a escalabilidade e precisão do modelo em diversas combinações de parâmetros.

5.4.7.1 Variando Número de Maps em Paralelo (pm)

Primeiramente o número de nós foi fixado em $n = 4$, o número de *threads* por *shufffle* em $ps = 4$ e variamos o número de *threads* para processar tarefas *map* (pm), de um, dobrando até, 32. Para $pm = 1$ as tarefas *map* são serialmente executadas em cada nó enquanto que para $pm = 32$ haverão 32 tarefas *map* em paralelo por nó. Nós utilizamos a mesma carga de trabalho para todos os cenários e mantivemos a mesma intensidade de carga ao aumentar o número de *threads* para processamento de tarefas *map* (pm), ou seja, nós aumentamos o paralelismo de tarefas mas cada tarefa *map* executa a mesma quantidade de trabalho. Enquanto aumentamos o número de tarefas *map*, nós mantivemos fixa a demanda total de cada recurso. Um aumento em pm causa uma redução na demanda de CPU (veja Tabela 5.2), então nós calculamos a proporção na mudança de $pm = 1$ para $pm = 4$ e mantivemos esta proporção de $pm = 4$ para $pm = 8$ e assim por diante.

A Figura 5.14 mostra que, novamente, o modelo baseado no Tripathi super-estimou o tempo médio de resposta do *job* enquanto que utilizando os operadores *fork/join* as estimativas do modelo apresentaram uma acordo bem melhor em relação ao simulador, com diferenças relativas tipicamente abaixo de 22%, exceto no $pm = 32$ que atinge 33% Como cada nó possui quatro CPU's, quando $pm = 1$ as CPU's estarão sub-utilizadas, apresentando menos contenção, como podemos ver em um decaimento

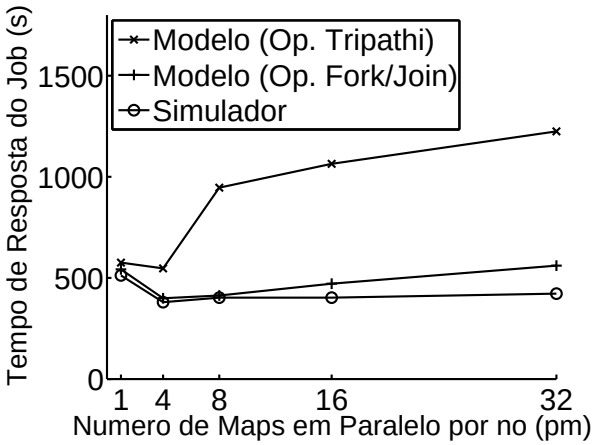


Figura 5.14. Tempo de resposta do *job* variando o parâmetro *pm*.

no início da curva na Figura 5.14. Note que após $pm = 4$ o desempenho do sistema começa a degradar, indicando que o aumento do número de *threads* para processar *maps* em paralelo ao invés de aumentar o *speedup* do *job* está gerando mais contenção.

5.4.7.2 Variando Shuffles em Paralelo (ps)

Em seguida, o número de *maps* em paralelo foi fixado em $pm = 4$, o número de nós $n = 4$ e variamos o número de *shuffles* em paralelo (ps), de um, dobrando até 128. Repare que com $ps = 1$, para cada *map* processado será disparado um *shuffle*, de cada *reduce*, havendo um grande paralelismo *pipeline*, e no outro extremos temos quando $ps = m = 128$, onde haverá um único *shuffle* por *reduce*, que só será disparado após todos os *maps* processarem e consumirá todos os *maps*. Da mesma forma que a análise anterior, verificamos que com o aumento do ps há uma redução na demanda de CPU do *map* e do *shuffle*. Calculamos a proporção entre as demandas dos cenários com $ps = 1$ para $ps = 4$ e mantivemos estas mesma proporção para $ps = 8$ em diante.

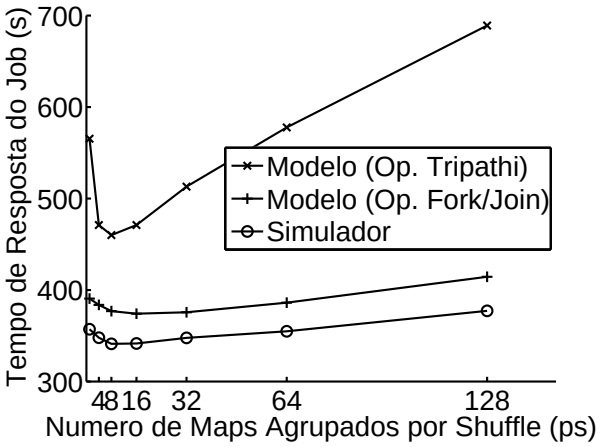


Figura 5.15. Tempo de resposta do *job* variando o parâmetro *ps*.

A comparação das métricas do modelo e simulador foram feitas na Figura 5.15. Assim como nos experimentos anteriores, o modelo utilizando o método do Tripathi super-estimou o tempo médio do *job*. Já o modelo utilizando o método do *fork/join* obteve boa aproximação em relação ao simulador (*baseline*) tendo o erro relativo máximo sido de 10% com $ps = 16$.

5.4.7.3 Variando o Número de Nós (n)

Nós agora avaliamos o impacto do aumentar o número de nós, na Figura 5.16. Para estes experimentos, nós fixamos o número de *maps* $m = 128$, o número de *maps* em paralelo $pm = 4$ e variamos o número de nós de 1, dobrando até 16. Note que, a medida que aumentamos o número de nós, nós também aumentamos o número de *reduces*, mantendo um *reduce* por nó. Como os *reduce*, principalmente o *merge* possui uma demanda considerável, assim como uma variância grande, a medida que aumentamos o número de nós também aumentamos o tempo de resposta dos *jobs*.

Os resultados da validação do número de nós foi apresentado na figura 5.16. O modelo baseado no Tripathi, assim como nos experimentos anteriores, super estimou o simulador cenários com mais de quatro nós ($n \geq 4$). O modelo utilizando o método do *fork/join* obteve boa aproximação até $n = 8$, erro relativo máximo 29% ($n = 8$), entretanto com $n = 16$ super-estimou o tempo médio do *job* em 35%.

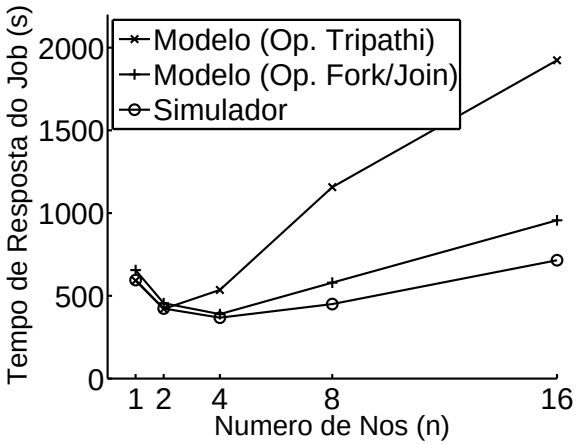


Figura 5.16. Tempo de resposta do *job* variando o número de nós (n).

Capítulo 6

Conclusão e Trabalhos Futuros

Modelos de desempenho são fundamentais para dar suporte a gerência e ao planejamento de capacidade de sistemas paralelos. Esta dissertação apresentou um modelo de desempenho hierárquico que aborda um componente chave de cargas de trabalho com computação paralela que é o paralelismo *pipeline*. O modelo adotado se baseia em uma solução anterior (Liang e Tripathi, 2000), aqui referido como modelo referência, que combina um modelo de precedência, representado por uma árvore, o qual expressa as restrições de precedência entre as tarefas de um *job* com paralelismo *pipeline* e em um algoritmo iterativo aMVA, o qual captura a contenção de recursos. O principal foco foi em mostrar como construir a árvore de precedência para uma carga de trabalho que contenha *pipeline* e desenvolver um algoritmo para gerá-la automaticamente. Este algoritmo é usado como um passo extra na solução iterativa proposta no modelo referência. Este modelo foi aplicado em duas cargas de trabalhos:

Primeiramente foi feito uma modelagem de um banco de dados distribuído, chamado HP *Neoview*, para uma carga de trabalho composta de consultas simples de *Business Intelligence* (BI), contendo paralelismo *pipeline* na troca de mensagens entre duas tarefas. Foi mostrado como construir a árvore de precedência para capturar as sincronizações e o paralelismo *pipeline* entre duas tarefas de um *query* de BI. A carga de trabalho de BI foi validada por um simulador da rede de filas e por um emulador detalhado do banco de dados HP *Neoview*.

Os experimentos mostraram um acordo muito próximo entre os três conjuntos de resultados, com diferença relativa abaixo de 11,5% para o tempo de resposta do *job*, nos seis cenários avaliados. Foi avaliado também o impacto dos parâmetros número de mensagens (F) e tamanho do *buffer* de comunicação (M) na estimativas do modelo analítico, onde encontramos algum impacto apenas para cargas balanceadas leves (MPL baixo), ou seja, antes do sistema saturar. Após a saturação, o tempo de res-

posta é dominado pelos o atrasos de filas e, um aumento no paralelismo pipeline, ao invés de aumentar o *speedup*, gera mais contenção.

Em seguida o modelo analítico foi aplicado na plataforma para computação distribuída *Hadoop Online Prototype* (HOP), que contém um paralelismo em *pipeline* entre múltiplas tarefas, apresentando relações de precedência mais complexas. A estratégia adotada consistiu em representar o paralelismo *pipeline* entre múltiplas tarefas de um *job* através de uma árvore de precedência, construída utilizando os operadores primitivos apresentados pelo modelo referência. O modelo analítico foi validado através de um simulador da rede de filas e de experimentos reais, em um ambiente configurado pelo grupo da HP-Labs.

Observou-se que o modelo analítico, utilizando apenas os operadores propostos no modelo referência, apresentou previsões de desempenho super-estimadas. As fontes de erro foram investigadas através de simuladores simplificados, onde identificamos que a premissa do modelo referência que assume que o tempo de resposta das tarefas são exponencialmente distribuídos, proposta originalmente por Salza e Lavenberg (1981), leva a estimativa super-estimadas. Verificamos que o tempo de fila segue uma distribuição hipo-exponencial, dado pela série de variáveis aleatórias exponenciais correspondentes a demanda de todas as tarefas que chegaram antes de uma determinada tarefa, tendo assim uma taxa de crescimento bem menor que a exponencial. Com isso, como foi observado em Jonkers (1994a), para cenários que não possuem uma variância baixa, a premissa do tempo de resposta exponencial leva a previsões super-estimadas.

Com o intuito de melhorar a acurácia do modelo, foi introduzido um novo operador primitivo, que adaptou o método de redes de filas *fork/join* (Varki, 1999) para estimar o tempo de resposta de um sub-conjunto de tarefas em paralelo. As previsões do modelo utilizando o operador *fork/join* obtiveram um bom acordo, em relação ao simulador e ao experimento, com diferenças relativas abaixo de 15% para o tempo de resposta do *job*. Note que o modelo *fork/join* foi conservador, uma vez que suas estimativas foram maiores que a do experimento, pois, neste caso, é melhor errar para mais do que para menos. Sendo assim, o operador primitivo *fork/join* produziu estimativas do tempo de resposta mais apertadas e mais acuradas.

Foi feito também uma análise do impacto dos parâmetros do modelo, onde foi observado que o modelo capturou bem o aumento do número de *threads* para processar cada tarefas *map* (pm), com diferença relativa máxima de 23%. Ao variar o número de *threads* para processar *shuffles* em cada *reduce* (ps), que pode ser visto também como o número de *maps* agregados pelos *shuffles*, o modelo analítico obteve uma precisão de 10%. Ao aumentar o número de nós o modelo obteve boas previsões até $n = 8$, entretanto super-estimou o tempo de resposta para $n = 16$ (35%).

Como trabalhos futuros serão abordados outros tipos de recursos, como a memória. Pretende-se também estender o modelo para carga de trabalho aberta, ou seja, onde a carga varia em função da taxa de chegada. As equações do modelo aberto foram derivadas pelo aluno Giovanni Commarela na disciplina Análise de Desempenho, ministrada pela Prof[a] Jussara, no semestre em fui monitor, onde ajudei em sua formulação. Planeja-se também realizar mais experimentos reais, avaliando uma combinação maior de parâmetros. Outro plano para trabalhos futuros é estender o modelo analítico para permitir a modelagem de uma carga de trabalho composta de múltiplas classes de *jobs*, como por exemplo, uma carga de trabalho composta de consultas OLTP e BI concorrentemente.

Apêndice A

Composição dos Nós Internos da Árvore de Precedência

O princípio para o cálculo do tempo médio de resposta do *job* (R_0) é observar que cada $J_i \in J$ é composto por dois outros *subjobs* J_{i1} e J_{i2}, de forma que $J_i = J_{i1} \diamond Ji2$, com $\diamond \in \{+, \vee, \wedge\}$. Então, conhecendo as distribuições dos tempos de resposta de J_{i1} e J_{i2} faz-se possível obter algebricamente seu o valor médio.

Sejam X, X_1 e X_2 as variáveis aleatórias que representam o tempo de resposta de J_i, J_{1i} e J_{2j} respectivamente. Denote ainda $m = E[X]$, $m_1 = E[X_1]$ e $m_2 = E[X_2]$, $\sigma^2 = Var[X]$, $\sigma_1^2 = Var[X_1]$ e $\sigma_2^2 = Var[X_2]$. Para simplificar a notação, dadas as variáveis aleatórias $X_i, i = 1, 2$, têm-se que $m_i = E[X_i]$ e $\sigma_i^2 = Var[X_i]$. Então:

- Se $\diamond = +$, $m = m_1 + m_2$ e $\sigma^2 = \sigma_1^2 + \sigma_2^2$
- Se $\diamond = \vee$, $m = E[min(X_1, X_2)]$ e $\sigma^2 = Var[min(X_1, X_2)]$
- Se $\diamond = \wedge$, $m = E[max(X_1, X_2)]$ e $\sigma^2 = Var[max(X_1, X_2)]$

Sejam X_1 e X_2 variáveis aleatórias com distribuições Erlang com parâmetros (λ_1, r1) e (λ_2, r2), respectivamente, então a média e variância de $X = min(X_1, X_2)$ são dadas por:

$$\begin{aligned} m = & \frac{\lambda_1^{r_1}}{(\lambda_1 + \lambda_2)^{r_1+1}} \sum_{k=0}^{r_2-1} (\frac{\lambda_2}{\lambda_1 + \lambda_2})^k \frac{(r1+k)!}{(r1-1)!k!} \\ & + \frac{\lambda_2^{r_2}}{(\lambda_1 + \lambda_2)^{r_2+1}} \sum_{k=0}^{r_1-1} (\frac{\lambda_1}{\lambda_1 + \lambda_2})^k \frac{(r2+k)!}{(r2-1)!k!} \end{aligned} \tag{A.1}$$

$$\sigma^2 = \frac{\lambda_1^{r_1}}{(\lambda_1+\lambda_2)^{r_1+2}} \sum_{k=0}^{r_2-1} (\frac{\lambda_2}{\lambda_1+\lambda_2})^k \frac{(r1+k+1)!}{(r1-1)!k!} + \frac{\lambda_2^{r_2}}{(\lambda_1+\lambda_2)^{r_2+2}} \sum_{k=0}^{r_1-1} (\frac{\lambda_1}{\lambda_1+\lambda_2})^k \frac{(r2+k+1)!}{(r2-1)!k!} - m^2 \tag{A.2}$$

Seja X_1 hiperexponencial com parâmetros (λ_{11}, λ_{12}, p_1) e X_2 com distribuições Erlang. Então a média e variância da variável aleatória $min(X_1, X_2)$ são dadas por:

$$m = m_1 - \frac{p_1}{\lambda_{11}} (\frac{\lambda_2}{\lambda_{11}+\lambda_2})^{r_2} - \frac{(1-p_1)}{\lambda 12} (\frac{\lambda_2}{\lambda 12+\lambda_2})^{r_2} \tag{A.3}$$

$$\sigma^2 = m_1^2 + \sigma_1^2 - m^2 - 2[\frac{p_1}{\lambda_{11}^2} (\frac{\lambda_2}{\lambda_{11}+\lambda_2})^{r_2} - \frac{(1-p_1)}{\lambda 12^2} (\frac{\lambda_2}{\lambda_{12}+\lambda_2})^{r_2}] - \frac{2r_2}{\lambda_2}[\frac{p_1}{\lambda_{11}} (\frac{\lambda_2}{\lambda_{11}+\lambda 2})^{r_2+1} + \frac{(1-p_1)}{\lambda_{12}} (\frac{\lambda_2}{\lambda_{12}+\lambda_2})^{r_2+1}] \tag{A.4}$$

Sejam X_1 e X_2 variáveis aleatórias com distribuição hiperexponencial com parâmetros (λ_{11}, λ_{12}, p_1) e (λ_{21}, λ_{22}, p_2), respectivamente, então a média e a variância de $X = min(X_1, X_2)$ são dadas por:

$$m = \frac{p_1 p_2}{(\lambda_{11}+\lambda_{21})} + \frac{(1-p_1)p_2}{(\lambda_{12}+\lambda_{21})} + \frac{p_1(1-p_2)}{(\lambda_{11}+\lambda_{22})} + \frac{(1-p_1)(1-p_2)}{(\lambda_{12}+\lambda_{22})} \tag{A.5}$$

$$\sigma^2 = \frac{2p_1 p_2}{(\lambda_{11}+\lambda_{21})^2} + \frac{2(1-p_1)p_2}{(\lambda_{12}+\lambda_{21})^2} + \frac{2p_1(1-p_2)}{(\lambda_{11}+\lambda_{22})^2} + \frac{2(1-p_1)(1-p_2)}{(\lambda_{12}+\lambda_{22})^2} - m^2 \tag{A.6}$$

Sejam X_1 e X_2 variáveis aleatórias com distribuição Erlang com parâmetros (λ_1, r_1) e (λ_2, r_2), respectivamente, então a média de $X = max(X_1, X_2)$ são dadas por:

$$m = m_1 + m_2 - \frac{\lambda_1^{r_1}}{(\lambda_1+\lambda_2)^{r_1+1}} \sum_{k=0}^{r_2-1} (\frac{\lambda_2}{\lambda_1+\lambda_2})^k \frac{(r1+k)!}{(r1-1)!k!} - \frac{\lambda_2^{r_2}}{(\lambda_1+\lambda_2)^{r_2+1}} \sum_{k=0}^{r_1-1} (\frac{\lambda_1}{\lambda_1+\lambda_2})^k \frac{(r2+k)!}{(r2-1)!k!} \tag{A.7}$$

$$\sigma^2 = m_1^2 + m_2^2 + \sigma_1^2 + \sigma_2^2 - m^2 - \frac{\lambda_1^{r_1}}{(\lambda_1+\lambda_2)^{r_1+2}} \sum_{k=0}^{r_2-1} (\frac{\lambda_2}{\lambda_1+\lambda_2})^k \frac{(r1+k+1)!}{(r1-1)!k!} + \frac{\lambda_2^{r_2}}{(\lambda_1+\lambda_2)^{r_2+2}} \sum_{k=0}^{r_1-1} (\frac{\lambda_1}{\lambda_1+\lambda_2})^k \frac{(r2+k+1)!}{(r2-1)!k!} \tag{A.8}$$

Sejam X_1 hiperexponencial com parâmetros (λ_{11}, λ_{12}, p_1) e X_2 com distribuição Erlang com parâmetros (λ_2, r_2). Então a média de $X = max(X_1, X_2)$ são dadas por:

$$m = m_2 + \frac{p_1}{\lambda_{11}} (\frac{\lambda_2}{\lambda_{11}+\lambda_2})^{r_2} - \frac{(1-p_1)}{\lambda 12} (\frac{\lambda_2}{\lambda 12+\lambda_2})^{r_2} \tag{A.9}$$

$$\sigma^2 = m_2^2 + \sigma_2^2 - m^2 + 2[\frac{p_1}{\lambda_{11}^2} (\frac{\lambda_2}{\lambda_{11}+\lambda_2})^{r_2} - \frac{(1-p_1)}{\lambda 12^2} (\frac{\lambda_2}{\lambda_{12}+\lambda_2})^{r_2}] - \frac{2r_2}{\lambda_2} [\frac{p_1}{\lambda_{11}} (\frac{\lambda_2}{\lambda_{11}+\lambda 2})^{r_2+1} + \frac{(1-p_1)}{\lambda_{12}} (\frac{\lambda_2}{\lambda_{12}+\lambda_2})^{r_2+1}] \tag{A.10}$$

Sejam X_1 e X_2 variáveis aleatórias com distribuição hiperexponencial com parâmetros (λ_{11}, λ_{12}, p_1) e (λ_{21}, λ_{22}, p_1). Então a média de $X = max(X_1, X_2)$ são dadas por:

$$m = m_1 + m_2 - \frac{p_1 p_2}{(\lambda_{11}+\lambda_{21})} - \frac{(1-p_1)p_2}{(\lambda_{12}-\lambda_{21})} - \frac{p_1(1-p_2)}{(\lambda_{11}+\lambda_{22})} - \frac{(1-p_1)(1-p_2)}{(\lambda_{12}+\lambda_{22})} \tag{A.11}$$

$$\sigma^2 = \sigma_1^2 + \sigma_2^2 + m_1^2 + m_2^2 - m^2 - \frac{2p_1p_2}{(\lambda_{11}+\lambda_{21})^2} + \frac{2(1-p_1)p_2}{(\lambda_{12}+\lambda_{21})^2} + \frac{2p_1(1-p_2)}{(\lambda_{11}+\lambda_{22})^2} + \frac{2(1-p_1)(1-p_2)}{(\lambda_{12}+\lambda_{22})^2} - m^2 \tag{A.12}$$

Apêndice B

Construção da Árvore do Paralelismo Pipeline de job HOP

O método para construção da árvore do paralelismo *pipeline*, mostrado na Seção 5.2.3.1, foi desenvolvido de forma independente, de modo que ele não não encherga o fluxo de execução das tarefas (*threads*, *buffers*, etc.). Ele consiste em um tipo abstrato de dados (TAD) que provê as seguintes primitivas: (1) cria árvore, (2) insere tarefa na árvore e (3) fim de fase. Chamamos de fase o intervalo entre dois pontos de sincronização. Os pontos de sincronização são caracterizadas por eventos de desbloqueio dos *shuffles* e é visto pela árvore como um ponto de serialização entre as tarefas (barreira). Os bloqueios ocorrem nas tarefas *shuffle*, que devem aguardar pelo término de *ps maps* para poderem iniciar. Este TAD tem como características se auto-arranjar à medida que as tarefas vão sendo inseridas, estando sempre estável, permitindo a visualização passo-a-passo de sua construção.

Este método requer que sejam identificadas as precedências entre as tarefas. A linha do tempo (*timeline*) representa o caso médio das precedências do *job*, dado pelo tempo médio de resposta das tarefas, estimado na rede de filas. A estratégia para identificação das precedências será explicado na Seção C e consiste em identificar os eventos de início, disparo (no caso dos *maps*) e fim das sub-tarefas. Nesta Seção estamos considerando que as precedências são conhecidas. O que chega para o método de construção da árvore são apenas as três primitivas apresentadas acima. É necessário então fazer o mapeamento dos eventos para as estas primitivas.

Os eventos que são de interesse para a construção da árvore são os eventos de disparo e de fim das sub-tarefas (os de início não), que correspondem ao momento em que as tarefas são inseridas na árvore. Podem ocorrer três eventos deste tipo: (1) disparo do *map*, que pode (ou não) desencadear um evento de desbloqueio do *shuffle*.

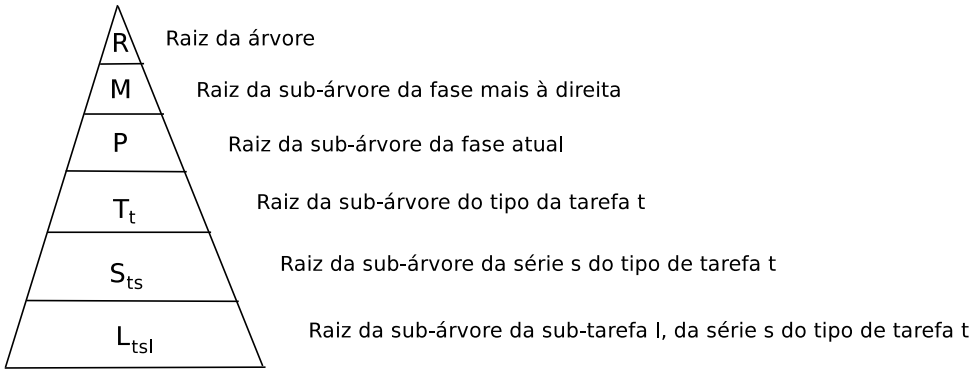


Figura B.1. Hierarquia das sub-árvores do paralelismo *pipeline*.

Caso não ocorra o desbloqueio, apenas a tarefa *map* disparada é anexada na sub-árvore correspondente (como veremos a seguir), caso contrário (ocorra desbloqueio), todas as tarefas em execução devem ser inseridas na árvore; (2) fim de uma sub-tarefa *map*, que é anexada na árvore e (3) fim de uma sub-tarefa *reduce*, que da mesma forma é anexada na árvore.

À medida que forem ocorrendo os eventos de disparo ou término das sub-tarefas, novas tarefas são adicionadas na árvore. Uma fase, como foi dito anteriormente, consiste no período entre dois pontos de sincronização. Toda árvore tem pelo menos dois pontos de sincronização: um para desbloquear o primeiro *shuffle* e outro quando termina a execução do *job*. E pode ter no máximo $\frac{m}{ps} - 1$, que é o número total de sub-tarefas *shuffle*, com exceção da última. Quando termina o último *shuffle*, ele dispara o *merge* sem possibilidade de bloqueio (sem passar pelas filas). Ao encontrar um ponto de sincronização ocorre o que chamamos de fim de fase. Todas as sub-tarefas inseridas entre o último ponto de sincronização (ou início do *job*) e o próximo ponto de sincronização, são inseridos em um mesmo galho da árvore, que chamamos de sub-árvore da fase. Inicia-se então uma nova fase. Antes de inserir a primeira tarefa da nova fase é inserido um nó serial, acima da última sub-árvore da fase, separando (serializando) as sub-árvores da fase.

A árvore da fase foi dividida em quatro sub-árvores para auxiliar sua construção: sub-árvore das sub-tarefas, sub-árvore do *thread*, sub-árvore do tipo (*map* ou *reduce*) e sub-árvore da fase. A árvore de sub-tarefa pode armazenar as sub-tarefas *map* de um *map* que foi sub-dividido por seu tempo de disparo ou as instâncias *shuffle* paralelo de uma sub-tarefa *shuffle*. A sub-árvore do *thread* contém a série de tarefas alocadas para cada *thread map* ou *reduce*. As sub-tarefas *map* e *reduce* são divididas em duas sub-árvore de acordo com seu tipo.

A hierárquia das sub-árvores é apresentada na Figura B.1. A sub-árvore da fase é composto por uma ou duas sub-árvores do tipo (se for inserido pelo menos uma sub-tarefa de cada tipo). A sub-árvore do tipo t se divide em uma ou mais sub-árvores do *thread*, podendo haver até S_t sub-árvores do *thread*, onde $St = \{pm, pr\}$ (de acordo com o número de tarefas de cada *thread* inseridas na árvore). E a sub-árvore do *thread* s (do tipo t) é composta de uma ou mais sub-árvores de sub-tarefas (dado pela sub-divisão do tempo de disparo) ou ps instâncias *shuffle*.

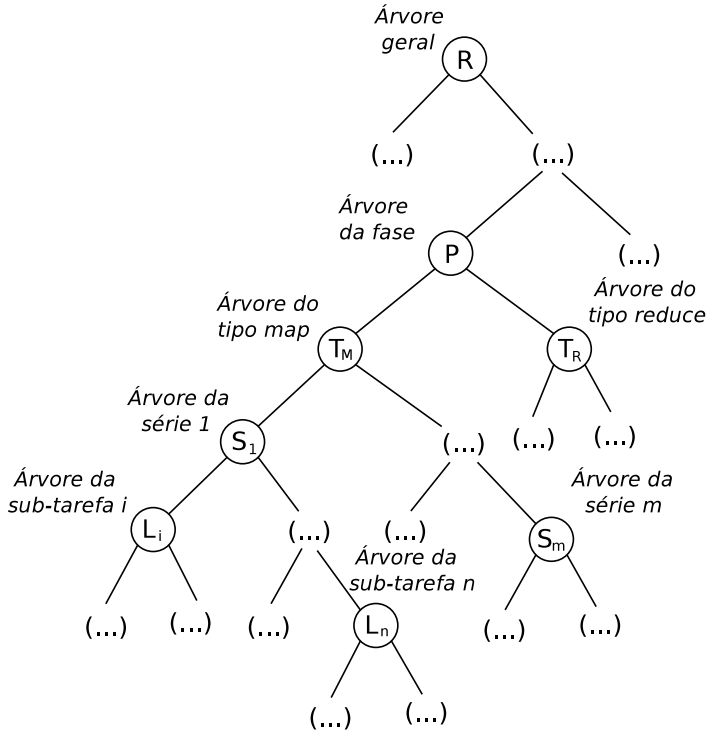


Figura B.2. Estrutura da árvore do paralelismo utilizada.

O método para construção da árvore, inicia com o procedimento CRIA-ARVORE(), descrito no Algoritmo 5, que cria um conjunto de apontadores para armazenar a referência para o nó-interno mais a direita (árvore sempre cresce para a direita) de cada tipo de sub-árvore (sub-tarefas, *thread*, tipo, fase e geral), além da raiz da aŕvore. Este apontadores são inicializados com nulo. A primeira tarefa a terminar é inserida na raiz da sub-árvore da fase (caso seja a primeira fase é também a raiz da árvore geral). As novas sub-tarefas são sempre adicionadas na árvore de sub-tarefas, do tipo e do *thread* correspondente, auto-arranjando as respectivas sub-árvores de modo a manter sempre a estrutura da árvore estável. A Figura B.2 apresenta a estrutura da árvore utilizada, destacando a localização dos ponteiros de cada tipo de sub-árvore.

Algoritmo 6: CRIA-ARVORE(T, S_t, L_t)

```
1  R = NULL                                   /* aponta raiz da árvore para nulo
2  M = NULL                                   /* aponta raiz da árvore da fase mais à direita para nulo
3  T_t = [ ]_{1×T}
4  S_ts = [ ]_{1×T}
5  L_tsl = [ ]_{1×T}
6  for t = 0 to T do                          /* para cada tipo de tarefa t */
7  |  S = S_t[t]
8  |  S_ts[t] = [ ]_{1×S}
9  |  L_tsl[t] = [ ]_{1×S}
10 |  for s = 0 to S do                       /* para cada thread s */
11 |  |  L = L_t[t]
12 |  |  L_tsl[t][s] = [ ]_{1×L}
13 FIM-DE-FASE()
14 return R, M, P, T_t, S_ts, L_tsl
```

A cada inserção de uma tarefa na árvore é chamado o procedimento INSERE-NA-ARVORE(), que toma como entrada o índice, o *thread*, o tipo e o tempo de resposta da sub-tarefa (pertencentes ao *struct* TAREFA, criado no método de identificação de precedências). Foi desenvolvido um procedimento de inserção para cada tipo de sub-árvore. Para simplificar seu entendimento agrupamos os algoritmos para inserção em cada sub-árvore em um único algoritmo recursivo (mantendo todas as propriedades do algoritmo expandido), apresentada no algoritmo 6, que possui três parâmetros adicionais: (1) um vetor com todos os ponteiros de uma sub-tarefa ($P_P = \{Ltsl[t][s][l], Sts[t][s], Tt[t], F, M, R\}$); (2) o índice do ponteiro atual p (iniciando pela sub-árvore das sub-tarefas, ou seja, pelo índice um); e (3) o número total de ponteiros P.

O procedimento INSERE-NA-ARVORE() inicia verificando se é a primeira inserção na árvore de sub-tarefas ($p = 1$), ou seja, se o ponteiro para o nó-interno (se houver) mais à direita da sub-árvore p está vazio. Se for a primeira inserção, aponta o ponteiro $P_p[p]$ para a tarefa T e propaga a inserção para a sub-árvore acima, fazendo uma chamada recursiva do procedimento INSERE-NA-ARVORE() passando o ponteiro da próxima sub-árvore $p+1$). Esses passos são repetidos até que encontre uma sub-árvore não-vazia ou chegue na árvore geral ($p = 6$). Repare que na primeira inserção da primeira fase todos os ponteiros irão apontar para a tarefa inserida. Ao encontrar uma sub-árvore não vazia é então criado o nó-interno, que irá criar um operador serial ou paralelo de acordo com a sub-árvore p e o tipo da tarefa t. Em seguida, lembrando que a sub-árvore não está vazia, é verificado se a sub-árvore p possui apenas uma tarefa ou se já foi inserido mais de uma tarefa. Para isso é verificado o tipo associado a raiz

Algoritmo 7: INSERE-NA-ARVORE(P_p, p, T)

```
1  se P_p[p] == NULL então                          /* se primeira inserção na sub-árvore p */
2  |  P_p[p] = T
3  |  INSERE-NA-ARVORE( P_p, p + 1, T )                          /* propaga inserção */
4  senão                                          /* contém uma tarefa ou um nó interno */
5  |  se p == TIPO e t == MAP então
6  |  |  I = CRIA-NO-MIN()
7  |  senão se (p == SUB e t == MAP) ou p == thread ou p == GERAL então
8  |  |  I = CRIA-NO-SERIAL()
9  |  senão
10 |  |  I = CRIA-NO-PARALELO()
11 |  I.esq = P_p[p]
12 |  I.dir = T                                  /* insere tarefa à direita do nó interno */
13 |  se P_p[p].type == NO-INTERNO então
14 |  |  I.esq = P_p[p].esq
15 |  |  I.dir = P_p[p].dir
16 |  |  P_p[p].esq = I
17 |  |  P_p[p].dir = P_p[p − 1]
18 |  senão                                                        /* contém uma tarefa */
19 |  |  para i = 0 até p faça                          /* atualiza ponteiros p/ nó interno */
20 |  |  |  se P_p[i] == P_p[p] então
21 |  |  |  |  P_p[i] = I
22 |  |  |  senão
23 |  |  |  |  P_p[i].dir = I
24 |  P_p[p] = I                    /* insere nó interno na raiz da árvore de sub-tarefas */
```

da sub-árvore p, dada pelo ponteiro P_p: (1) se na raiz houver apenas uma tarefa (que terminou antes da tarefa atual), cria-se uma sub-árvore com o nó-interno criado na raiz, com a tarefa encontrada como nó-folha à esquerda e a nova tarefa como nó-folha à direita. Insere-se esta sub-árvore na raiz da sub-árvore p. Em seguida atualiza a referência dos ponteiros das sub-árvores acima, que apontavam para a tarefa, passando a apontar para o nó interno; (2) Se na raiz houver mais de uma tarefa, o que será encontrado em P_p será o nó-interno mais à direita da sub-árvore p, contendo duas tarefas como nós-folha L_1 e L_2. Cria-se então uma sub-árvore com o nó interno criado na raiz, com a tarefa L_2 como nó-folha à esquerda e a nova tarefa como nó-folha à direita. Insere-se esta sub-árvore à direita do nó-interno mais à direita, ou seja, no lugar de L_2 e atualiza o ponteiro P_p para o novo nó-interno mais à direita. Ao encontrar um ponto de sincronização, antes de ser chamado INSERE-NA-ARVORE(), é chamado o procedimento FIM-DE-FASE, descrito no Algoritmo 7, que reinicializa todos os ponteiros da fase para nulo (sub-tarefas, *thread*, do tipo e fase).

Algoritmo 8: FIM-DE-FASE(R, M, T_t, S_{ts}, L_{tsl})

```
1 P = NULL                                   /* aponta sub-árvore da fase para nulo
2 for t = 0 to T do                          /* para cada tipo de tarefa t */
3 |   T_t[t] = NULL                          /* aponta sub-árvore do tipo para nulo
4 |   for s = 0 to S_t[t] do                 /* para cada thread s */
5 |   |   S_ts[t][s] = NULL
6 |   |   for l = 0 to L_t[t] do            /* para cada sub-tarefa l */
7 |   |   |   L_tsl[t][s][l] = NULL
```

A Figura B.3 e B.4 mostra passo-a-passo o processo de construção a árvore do paralelismo *pipeline*, entre quatro tarefas (com dois *threads*) *map* e um *reduce* com no máximo um *shuffle* em paralelo ($ps = 1$). O primeiro evento identificado é o disparo de um *map* (rótulo 1). Como número máximo de *shuffles* em paralelo é um, após o primeiro *map* disparar o *shuffle* já é desbloqueado, caracterizando um ponto de sincronização representado pela linha tracejada horizontal. Repare que o *map* do segundo *thread* ainda não terminou de processar, sendo assim ele será fragmentado no ponto de sincronização e a primeira parte será inserida na árvore juntamente com o *map* que originou o disparo. O próximo evento foi o fim de um *shuffle* (rótulo 2). Como houve um ponto de sincronização no evento anterior, antes de inserir a tarefa *shuffle* deve ser inserido um nó serial na raiz da árvore (serialização entre as tarefas). O *shuffle* será anexado à direita do nó serial da raiz. Em seguida ocorre dois eventos simultâneos: o fim do *reduce* no primeiro *thread* e o disparo/fim de um *map* no segundo *thread* (rótulo 3 e 1,3). Como houve um evento de desbloqueio de um *shuffle* (ponto de sincronizacao), o evento de fim de um *map* está contido no evento de disparo de um *map*, onde todas as tarefas que estão em execução (os dois *maps*) serão inseridos na árvore. Esses são os principais passos para a construção da árvore. Em seguida esses passos são repetidos, de acordo com as precedências identificadas na linha de tempo do *job* HOP (como será explicado na Seção 4.3.6) até que todas as tarefas tenham sido inseridas na árvore.

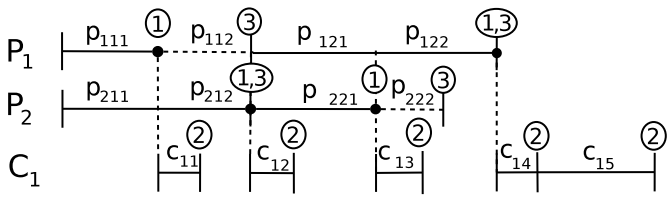


Figura B.3. Exemplo d linha do tempo de um *job*.

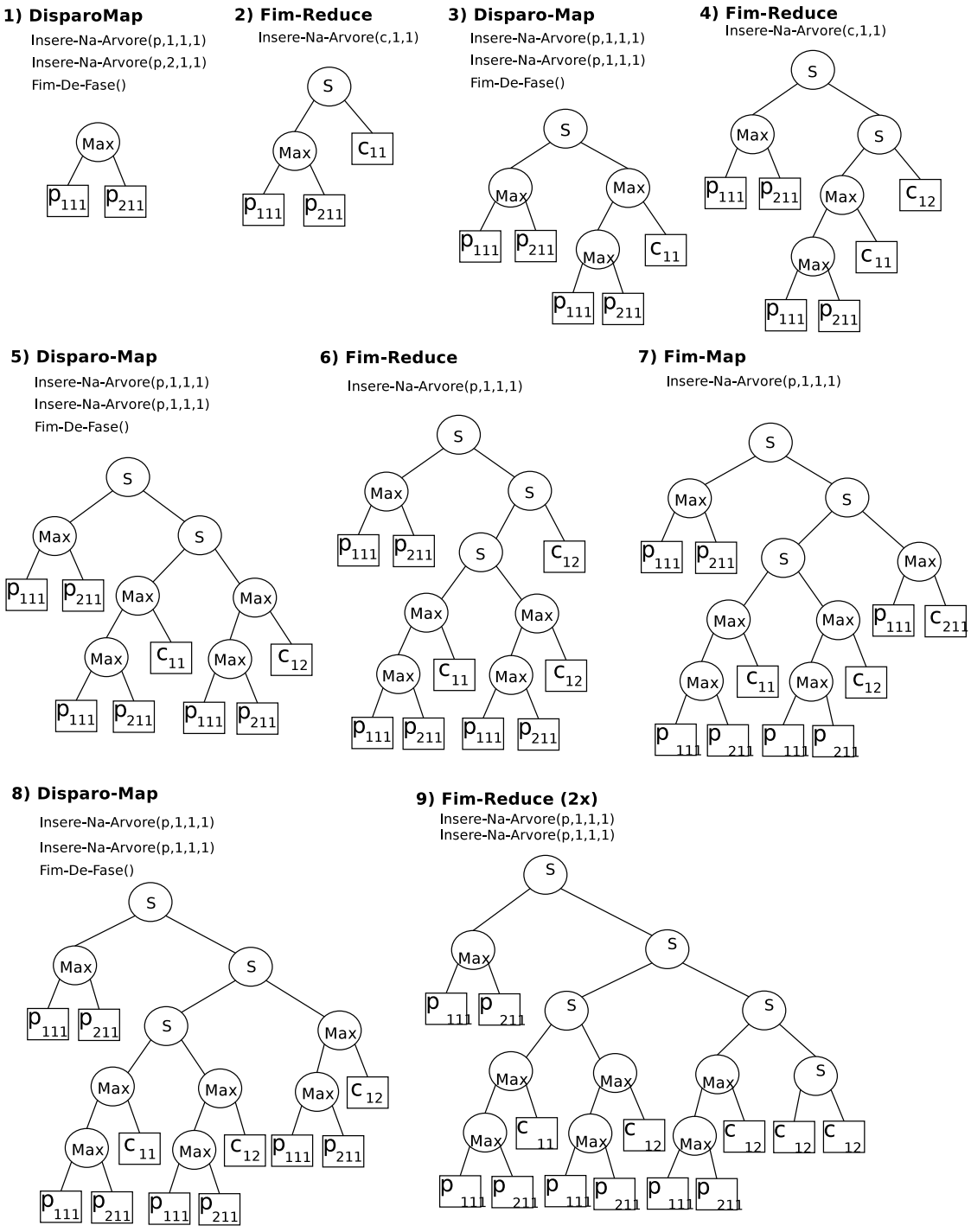


Figura B.4. Exemplo de construção da árvore a partir do *timeline*.

Apêndice C

Identificação das Precedências para Construção da Árvore

O método para construção da árvore, como foi explicado na Seção 5.2.3.2, requer que sejam identificadas as precedências entre as tarefas. A linha do tempo (*timeline*) representa o caso médio das precedências do *job*, dado pelo tempo médio de resposta das tarefas, estimado na rede de filas. Esta Seção apresenta a descrição da execução da linha de tempo do *job* HOP, realizada com base no tempo médio de resposta das tarefas (estimado pela rede de filas), buscando encontrar as precedências entre as tarefas através da identificação dos eventos (orientada a eventos) de início, disparo (no caso dos *maps*) e fim das sub-tarefas, assim como os pontos de sincronização.

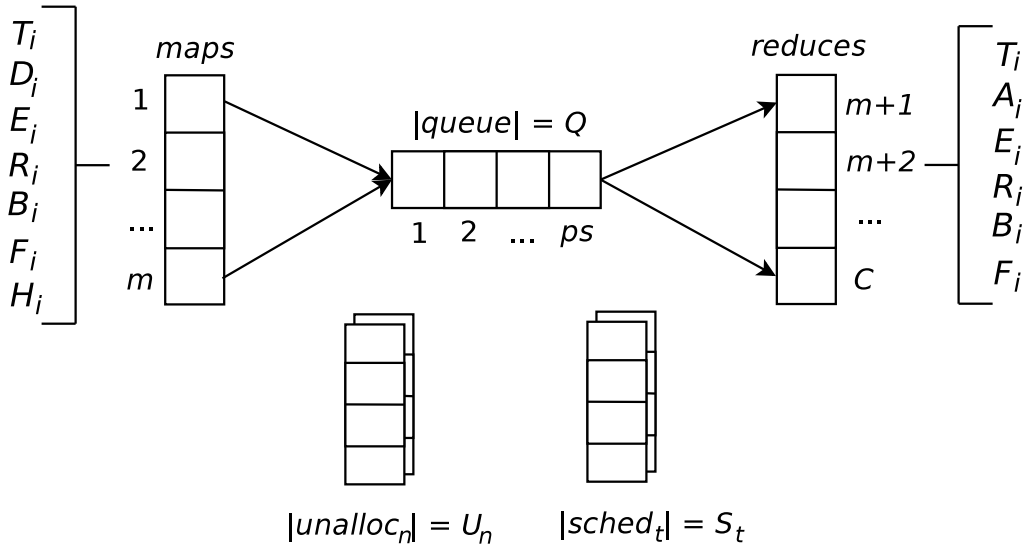


Figura C.1. Estruturas de dados utilizadas para representar a execução do *job*.

O paralelismo *pipeline* do HOP se dá entre m produtores (tarefas *map*) e $\frac{m}{ps}$ consumidores (sub-tarefas *shuffle-sort*, que se sub-dividem em ps instâncias). As sub-tarefas produzidas pelos *maps* devem ser consumidos por todos os *reduces*. A modelagem mais intuitiva para isso é utilizar uma fila para cada consumidor e, quando a tarefa *map* terminar de executar, insere o *map* processado na fila de todos os consumidores. Entretanto, esta modelagem (uma fila por consumidor) possui algumas limitações, pois se um consumidor processar mais devagar que outro, quando o produtor terminar de executar ele pode encontrar a fila deste consumidor cheia, sendo necessário armazenar os *maps* processados em outra fila provisóriamente (até que libere um *slot* na fila do consumidor mais lento). Seria preciso distinguir entre quais consumidores estavam com a fila cheia e quais não, precisando de uma fila para cada consumidor. Ao invés disso, consideramos que todos os consumidores compartilham uma única fila lógica ($buffer$), de tamanho Q, capacidade ps e disciplina FCFS. Esta fila possui também associado um contador do número total de vêzes que ela encheu (com ps sub-tarefas dos produtores) e foi consumida (de uma só vez) pelos consumidores, o qual chamamos de contador de agregações ($agregacao$). Cada consumidor também possui um contador do número de agregações realizadas pelo mesmo (A_i). Quando um consumidor termina de processar, antes de consultar a fila, ele compara os contadores. Se o número de agregações do consumidor for menor que o contador global ($A_i < A$), então significa que o consumidor ainda não processou o i-ésimo grupo de produtores, podendo disparar a próxima sub-tarefa. Se o consumidor já processou todas as agregações anteriores, então verifica: se a fila está cheia ($Q = ps$) então dispara a próximo sub-tarefa, consumindo todos os produtores da fila de uma só vez; caso contrário, se a fila ainda não encheu ($Q < ps$), o consumidor é bloqueado.

A Figura C.1 apresenta um esquema com as estruturas de dados utilizadas para representar o fluxo de execução médio de um *job* HOP. As variáveis utilizadas foram sumarizadas na Tabela C.1. O *job* é dividido em C sub-tarefas (somando *maps* e *reduce*), que se comunicam através de uma fila lógica única, *queue*, de tamanho Q, com capacidade de armazenar ps sub-tarefas *map* processadas. Para representar as sub-tarefas foi utilizado um conjunto de vetores com a respectivas propriedades. As tarefas são caracterizadas por seu tipo (T_i), pelo instante de tempo em que ela irá terminar (E_i), pelo seu tempo médio de resposta médio (R_i), pelo *flag* se a tarefa está bloqueada ou não (B_i), pelo *flag* se a tarefa terminou ou não (F_i) e pelo nó pré-alocado para a tarefa (H_i). Os *map* têm de armazenar em particular o seu tempo de disparo, enquanto que os *reduce* possuem ainda o contador do número de agregações processadas pelo mesmo. Além da fila lógica dos consumidores, há também duas outras filas: a fila de tarefas não-alocadas de cada nó, $unalloc_n$, de tamanho U_n (pré-alocadas a partir

Tabela C.1. Variáveis utilizados no algoritmo de execução do *timeline* do *job*.

Notação	Variáveis
$time$	Instante de tempo atual
$queue$	Fila lógica de *maps* processados.
$unalloc_n$	Fila de tarefas não alocadas de cada nó.
$sched_t$	Fila de tarefas a serem escalonadas
Q	Tamanho da fila lógica de *maps* processados.
U_n	Tamanho da fila de tarefas não alocadas de cada nó.
S_t	Tamanho da fila de tarefas a serem escalonadas
T	Número de tipos de tarefa
K	Número total de centros de serviço
A	Número de agregações geral
P	Lista de tarefas que acabaram de terminar.
T_i	Tipo da tarefa i (*map*, *ss* ou *merge*).
D_i	Tempo de disparo da tarefa i (apenas para *map*)
R_i	Tempo de resposta da tarefa i (estimado pelo MVA)
E_i	Tempo de término da tarefa i
A_i	Número de agregações feitas pela tarefa i (apenas para *ss*)
B_i	Flag se tarefa i está bloqueada ou não.
F_i	Flag se tarefa i está terminou a ou não.
H_i	Nó pré-alocado para a tarefa i

das demandas) e a fila *sched* contendo os *maps* processados que encontraram a fila cheia ($Q = ps$) ou os *reduces* bloqueados aguardando o a fila lógica (*queue*) encher para serem disparados.

Para simplificas a explicação das regras de precedência (tipos de evento), considere que as tarefas alocadas dinâmicamente são sub-tarefas do mesma *thread*, numeradas sequencialmente a partir do índice da *thread* de cada tipo de tarefa, como mostrado na Figura C.2 e sumarizado na tabela C.2. O *job* inicia disparando $pm \cdot n$ tarefas *map*. Após o início da execução podem ocorrer seis tipos de eventos.

Se o menor dos tempos de resposta acumulados das *threads* (dado pelo $min_{ij}(T_i^{acc}$, onde T_i^{acc} pode ser tanto um P_i^{acc} quanto um C_i^{acc}) for um produtor (próximo a terminar) e a tarefa não estiver bloqueadas e não tiver terminado então verifica-se: (1) se sub-tarefa ainda não tiver sido disparada, então ocorre o evento de disparo do produtor que tentará inserir a sub-tarefa produzida na fila lógica *queue*, caso encontre a fila cheia é inserido na fila temporária *sched*; (2) se ao disparar uma sub-tarefa *map* a fila lógica se tornou cheia e há consumidores bloqueados então ocorrerá o evento de desbloqueio dos consumidores, onde o primeiro consumidor irá retirar de uma só vez todos os *maps* da fila lógica (incrementando o contador de agregações global e do consumidor) e os demais consumidores serão disparados ao comparar seus contadores; (3) se sub-tarefa

map tiver sido disparada ou for disparada no mesmo instante de seu término, então ocorre o evento de fim da sub-tarefa *map*; (4) se ocorrer o evento fim da sub-tarefa *map* e houver *map* não-alocados para o nó pré-alocada para a tarefa que acabou de terminar então ocorre o evento aloca sub-tarefa *map*.

Se o menor dos tempos de resposta acumulados das *threads* for um consumidor (próximo a terminar) e a tarefa não estiver bloqueada e não tiver terminado então ocorre o evento: (5) fim de um consumidor (*shuffle* ou *reduce*). Uma sub-tarefa *shuffle* só termina quando todas suas *ps* instancias terminam. (6) se ao terminar uma sub-tarefa consumidora o número de agregações do consumidor for menor que o número de agregações global ou se o consumidor já processou todas as agregações anteriores, mas a fila lógica está cheia, então ocorre o evento disparo da próxima sub-tarefa do consumidor. Pode ocorrer também a alocação dinâmica de um *reduce*. Como em cenários reais não é muito comum utilizar muitos *reduces* por nó (devido ao processamento pesado realizado pela função *reduce*) não consideramos este evento.

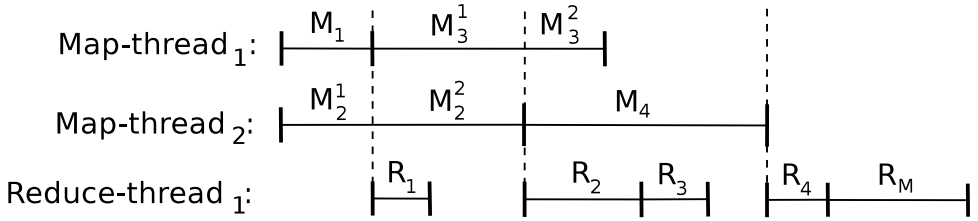


Figura C.2. Exemplo da linha de tempo de execução de um *job* com $ps = 1$

Nº	**Regras de precedência e respectivos eventos**	**Exemplo**
1	$P_i^{acc} = \min_{i,j}(T_i^{acc})$ & ! B_i & ! F_i& ! D_i → Disparo de uma sub-tarefa *map*	$p_{111} < p_{21}$ & ! D_{111} → Disparo de p_{111}
2	$P_i^{acc} = \min_{i,j}(T_i^{acc})$ & ! B_i & ! F_i& ! D_i && $Q + 1 = ps$ → Desbloqueio das sub-tarefas *shuffle*	$p_{111} < p_{21}$ & $Q + 1 = ps$ → Desbloqueio de c_{11}
3	$P_i^{acc} = \min_{i,j}(T_i^{acc})$ & ! B_i & ! F_i & $(D_i \mid R_i = time)$ → Fim de uma sub-tarefa *map*	$p_{112} < p_{21}$ & D_i → Fim de p_{11}
4	$P_i^{acc} = \min_{i,j}(T_i^{acc})$ & ! B_i & ! F_i & $(D_i \mid R_i = time)$ & $Q_{H_i} > 0$ → Alocação de um *map* dinâmico após o fim de um *map*	$p_{112} < p_{21}$ & D_i → Aloca p_{12}
5	$C_j^{acc} = \min_{i,j}(T_i^{acc})$ & ! B_j & ! F_j & $(A_j < A \mid B = ps)$ → Disparo do próximo *shuffle* após fim de um *shuffle*	$c_{12} < p_{121} < p_{22}$ & $B = ps$ → Dispara c_{13}
6	$C_j^{acc} = \min_{i,j}(T_i^{acc})$ & ! B_j & ! F_j & $B < ps$ → Bloqueio de um *shuffle* após o fim de um *shuffle*	$c_{11} < p_{112} < p_{21}$ & $B = 0$ → Bloqueia c_{11}

Tabela C.2. Tabela das principais regras de precedência e os respectivos eventos.

O procedimento EXECUTA-TIMELINE, apresentado no algoritmo 8, representa o fluxo de execução paralela das tarefas do *job* do HOP. O algoritmo toma como entrada o tempo de resposta estimado pela rede de fila na iteração anterior, assim como os parâmetros da carga de trabalho: número máximo de tarefas por nó ($N_t = pm, pr$), número de agregações por *shuffle* (ps), o número de tarefas de cada tipo (C_t) e os nós pré-alocados para cada sub-tarefa (H_i). Primeiramente inicializa-se o número de total tarefas a serem inseridas na árvore (C), o número de agregações geral (A) e aloca a fila lógica *queue*. Os vetores de ponteiros utilizados para caracterizar a árvore do paralelismo são então criados pelo procedimento CRIA-ARVORE(). As variáveis que representam a execução do *job* são inicializadas pelo procedimento INICIALIZA-JOB().

Algoritmo 9: EXECUTA-TIMELINE(R_i, N_t, C_t, ps, H_i)

```
1  C = C_t[MAP] + C_t[REDUCE]                          /* numero total de tarefas */
2  agregacao = 0                                       /* número de agregações global */
3  fila-logica = { }                                   /* fila lógica entre maps e reduce */
4  { R, P, T_t, S_ts, L_tsl } = CRIA-ARVORE( T, S_t, L_t )
5  { T_i, B_i, F_i, E_i, A_i, S_t, U_tn } = INICIALIZA-JOB( C_t, H_i, N_t, R_i )
6  CALCULA-DISPARO-MAPS( T_i, F_i, D_i, R_i, E_i, C, time )
7  enquanto C > 0 faça                                 /* enquanto não inserir todas tarefas na árvore */
8  |  {P, restantes, desbloqueio} = PROXIMOS-TERMINAR( T_i, C )
9  |  se desbloqueio > 0 ou restantes == 0 então
10 |  |  AJUSTA-TEMPO-TERMINO()
11 |  |  P = PROXIMOS-TERMINAR( T_i, C )
12 |  |  para i = 0 to C faça                           /* para cada tarefa i */
13 |  |  |  se ! F[i] então INSERE-NA-ARVORE( R, P, T_t, S_ts, L_tsl, T_i[i] )
14 |  |  |  FIM-DE-FASE()
15 |  else para cada i in P faça                        /* para cada tarefa que acabou de terminar */
16 |  |  INSERE-NA-ARVORE( R, P, T_t, S_ts, L_tsl, T_i[i] )
17 |  para cada i in P faça                             /* para cada tarefa que acabou de terminar */
18 |  |  se T_i[i].tipo == MAP então FIM-MAP(T)
19 |  |  senão FIM-REDUCE( T_i[i], agregacao, fila-logica, sched_t, unalloc_tn )
20 retorna tree
```

O procediento INICIALIZA-JOB(), apresentado no Algoritmo 9, inicializa as variáveis utilizadas na execução do *job*. Percorre-se os índices das tarefas (c) de cada tipo de tarefa (t), mapeando o tipo da tarefa ao índice global das tarefa (i). Verifica se há *threads* disponíveis. Se houver, verifica se a tarefa é *map*. Se for um *map*, dispara a tarefa, setando o tempo de témino dado pelo tempo de resposta estimado pela rede de filas. Se for um *reduce* insere na fila para ser disparado assim que *ps maps* terminarem. Se todos as *threads* estiverem ocupados insere tarefa na fila de não-alocados.

Algoritmo 10: INICIALIZA-JOB(T, C_t, H_i, N_t, R_i)

```
1  i = 0                                                    /* indice global das tarefas */
2  C = C_t[MAP] + C_t[REDUCE]                               /* numero total de tarefas */
3  T_i = [ ]_{1×C}                                          /* vetor de tarefas */
4  N_th = 0, ∀ t, h                                         /* número de threads ocupadas por nó */
5  nao-alocado_th = { }, ∀ t, h                             /* fila de tarefas não-alocadas */
6  S_t = { }, ∀ t                                           /* fila de tarefas a serem escalonadas */
7  para t = 0 até T faça                                    /* para cada tipo de tarefa t */
8      para c = 0 até C_t[t] faça                           /* para cada tarefa c */
9          T = T_i[i] = CRIA-TAREFA( t )
10         T.no = H_i[i]                                    /* obtém nó pré-alocado para a tarefa */
11         se N_tn[T.tipo][T.no] < N_t[t] então             /* se há threads disponíveis */
12             N_tn[T.tipo][T.no]++                         /* incrementa número de threads ocupados */
13             se T.tipo == MAP então
14                 T.bloqueou = falso
15                 T.termino = R_i[i]                       /* dispara map e seta tempo de término */
16             senão
17                 T.bloqueou = verdadeiro
18                 T.agregacao = 0
19                 S_t[t] ← {i}                             /* insere tarefa na fila de escalonamento */
20         senão
21             T.bloqueou = verdadeiro
22             nao-alocado_tn[T.tipo][T.no] ← { T }         /* insere T em não-alocados */
23         T.terminou = falso
24         i++
25 return T_i, B_i, F_i, E_i, A_i, S_t, U_tn
```

Em seguida é calculado o tempo de disparo das sub-tarefas *map* em execução, ou seja, que não estão bloqueadas e que ainda não terminaram. O algoritmo EXECUTA-TIMELINE() então entra em um laço (linhas 7-20) e só para quando todas as sub-tarefas forem inseridas na árvore do paralelismo, ou seja, quando o número tarefas restantes for zero ($C = 0$). A cada iteração, percorre-se todas as tarefas em execução, buscando pelas tarefas com o menor tempo de término (próximas a terminar), podendo haver mais de uma tarefa que terminem no mesmo instante. Verifica-se, em seguida, se houve um fim de fase (ponto de sincronização). Um fim de fase pode ocorrer por dois motivos: (1) por um evento de desbloqueio, ou seja, quando há *shuffles* bloqueados e um *map*, ao inserir uma sub-tarefa na fila lógica torna a fila cheia, desbloqueando todas as *reduces* bloqueados; ou (2) no fim da execução do *job*.

A execução da linha de tempo do fluxo de execução do *job* inicia procurando quais das tarefas que estao executando que é a próxima a terminar, no método PROXIMOS-TERMINAR. Para isso, percorre o indice de todas as tarefas, verificando, entre as tarefas que não estão bloqueadas (por não terem começado ou por terem sido bloqueadas durante a execução) e ainda não finalizaram quais possuem o menor tempo de término (podendo ser mais de uma).

Algoritmo 11: PROXIMOS-TERMINAR(T_i, C)

```
1  min = ∞                                              /* menor tempo de término */
2  list = { }                                  /* cria lista das próximas tarefas a terminar */
3  para i = 0 até C faça                                          /* para cada tarefa i */
4  |  T = T_i[i]                                                  /* obtem tarefa i */
5  |  se ! T.terminou e ! T.bloqueou então
6  |  |  se T.tipo == MAP e ! T.disparou então  termino = T.disparo
7  |  |  senão  termino = T.termino
8  |  |  se termino < min então          /* se encontrado término menor que min */
9  |  |  |  min = termino                              /* seta menor tempo de término */
10 |  |  |  list = { i }                  /* esvazia e cria lista com apenas tarefa i */
11 |  |  senão se termino == min então     /* se outras terminarem em min */
12 |  |  |  list ← { i }                  /* adiciona tarefa na lista de tarefas a terminar */
13 retorna list                          /* retorna lista de tarefas que acabaram de terminar */
```

O objetivo é identificar os eventos de inicio, disparo e término das tarefas e mapeá-los para as primitivas da árvore , como foi explicado na Seção 5.2.3.2, onde, para a construção da árvore, os eventos que são de interesse são os eventos de disparo/fim de uma tarefa (os de início não). Sendo assim, em seguida são identificados quais eventos ocorreram no instante de tempo atual. Os tipos de eventos foram sumarizados na Tabela C.2. Entretanto para construção da árvore são necessários apenas o evento de término das tarefas.

Ao ocorrer o fim de uma tarefa *map*, é chamado o procedimento FIM-MAP(), que trata os três possíveis fluxos de execução: (1) se houve apenas o disparo do *map*; (2) se o *map* já foi disparado e acabou de terminar; ou (3) se o *map* disparou e terminou ao mesmo tempo. Se *map* ainda não foi disparado, seta *flag* informando que foi disparada (linha 2). Verifica então se a fila lógica encheu (linha 3): se encheu, insere *map* na fila para escalonamento (*sched*) (linha 4); se não encheu, insere *map* na fila lógica (linha 6) e verifica se a fila encheu (linha 7): se fila encheu, então desbloqueia todos os *reduces* (se houver) (linhas 8-10). Em seguida verifica se o *map* terminou: se *map* terminou, seta *flag* informando que terminou e verifica se há *maps* não alocados para o nó do *map* que acabou de terminar: se houver, remove *map* da fila não-alocados e desbloqueia.

Algoritmo 12: FIM-MAP(M, $fila\text{-}logica$, B_t, ps, S_t, U_n, $tempo$)

```
1  se ! T.disparou então                                   /* se não disparou map */
2  |  T.disparou = verdadeiro
3  |  se TAMANHO( fila-logica ) == ps então               /* se fila logica está cheia */
4  |  |  sched_t[MAP] ← { T }                             /* insere map na fila de escalonamento */
5  |  senão                                                /* se fila não está cheia */
6  |  |  fila-logica ← { T }                              /* insere tarefa na fila */
7  |  |  se TAMANHO( fila-logica ) == ps então            /* se fila lógica encheu */
8  |  |  |  enquanto TAMANHO( sched_t[REDUCE] ) > 0 faça
9  |  |  |  |  R ← sched_t[REDUCE]
10 |  |  |  |  R.bloqueou = falso                         /* desbloqueia reduces */
11 se T.termino == tempo então                            /* se map acabou de terminar */
12 |  T.terminou = verdadeiro                              /* seta flag de fim da tarefa */
13 |  se TAMANHO( nao-alocado_tn[MAP][T.no] )> 0 então     /* há map p/ alocar */
14 |  |  A ← nao-alocado_tn[MAP][T.no]                    /* remove reduce da de fila não-alocados */
15 |  |  A.bloqueou = falso                               /* desbloqueia map (alocação dinâmica) */
```

Quando ocorrer o fim de um *reduce* é chamado o procedimento FIM-REDUCE, que inicia setando *flag* que informa que *reduce* terminou e em seguida verifica se é um *shuffle* sort ou *merge*. Se for um *merge*, marca o fim da tarefa *reduce*, caso contrário, verifica se é o último *shuffle*. O último *shuffle* dispara o *merge* sem possibilidade de bloqueio, ou seja, sem passar pelas filas. Caso contrário, se não for o último *shuffle*, verifica os contadores. Se *reduce* ainda não tiver consumido agregação atual, dispara a próxima sub-tarefa, caso contrário verifica se a fila lógica possui pelo menos ps sub-tarefas *map*. Se tiver, dispara a próxima tarefa, caso contrário, bloqueia.

Algoritmo 13: FIM-REDUCE(T, $agregacao$, $fila\text{-}logica$, $sched_t$, $unalloc_{tn}$)

```
1  T.terminou = verdadeiro                                 /* seta flag de fim da tarefa */
2  se T.fase == MERGE então
3  |  se TAMANHO( nao-alocado_tn[REDUCE][R.no] )> 0 então
4  |  |  A ← nao-alocado_tn[REDUCE][R.no]                 /* remove reduce da de fila
   |  |  não-alocados */
5  |  |  A.bloqueou = falso                               /* desbloqueia reduce (alocação dinâmica) */
6  senão
7  |  se T.agregacao ≤ agregacao então                     /* tarefa n consumiu agregação */
8  |  |  T.agregacao++                                    /* incrementa contador de agregações da tarefa */
9  |  senão se TAMANHO( fila-logica ) == ps então         /* se fila lógica encheu */
10 |  |  fila-logica = { }                                /* reduce consome fila toda */
11 |  |  agregacao++                                      /* incrementa contador de agregações geral */
12 |  |  T.agregacao++                                    /* incrementa contador de agregações da tarefa */
13 |  senão                                                /* não consumiu agregação atual e fila não está cheia */
14 |  |  sched_t ← {i}                                    /* insere reduce na fila de escalanamento */
15 |  |  T.bloqueou = verdadeiro                          /* bloqueia reduce */
```

Apêndice D

Tempo de Resposta das Tarefas no Experimentos Real

Tabela D.1. Experimento $pm = 1$ e $ps = 1$

Tipo	Exp. (E)	Simul. (S)	Tripathi (T)	Fork/Join (F)	S x E	T x E	F x E
Map	9.18	8.61	8.87	8.87	-6%	-3%	-3%
SS 1	538.18	520.44	549.24	549.24	-3%	2%	2%
Merge 1	100.26	100.95	100.99	100.99	1%	1%	1%
SS 2	530.05	520.44	549.21	549.21	-2%	4%	4%
Merge 2	128.74	103.38	102.63	102.63	-20%	-20%	-20%
SS 3	555.58	520.44	549.24	549.24	-6%	-1%	-1%
Merge 3	133.53	98.28	98.13	98.13	-26%	-27%	-27%
Job	722.23	686.06	798.56	732.56	-5%	11%	1%

Tabela D.2. Cenários da carga de trabalho $pm = 1$ e $ps = 5$

Tipo	**Exp. (E)**	**Simul. (S)**	**Tripathi (T)**	**Fork/Join (F)**	**S x E**	**T x E**	**F x E**
Map	8.44	8.72	9.05	9.05	3%	7%	7%
SS 1	464.28	480.57	491.84	491.84	4%	6%	6%
Merge 1	99.11	79.58	82.45	82.45	-20%	-17%	-17%
SS 2	463.94	480.60	491.89	491.89	4%	6%	6%
Merge 2	111.35	80.45	81.65	81.65	-28%	-27%	-27%
SS 3	463.76	480.62	491.89	491.89	4%	6%	6%
Merge 3	111.23	81.60	81.63	81.63	-27%	-27%	-27%
Job	605.40	629.99	701.75	659.13	4%	16%	9%

Tabela D.3. Cenários da carga de trabalho $pm = 4$ e $ps = 1$

Tipo	**Exp. (E)**	**Simul. (S)**	**Tripathi (T)**	**Fork/Join (F)**	**S x E**	**T x E**	**F x E**
Map	13.46	16.91	13.04	13.04	26%	-3%	-3%
SS 1	199.86	233.46	166.13	166.13	17%	-17%	-17%
Merge 1	88.24	103.39	102.57	102.57	17%	16%	16%
SS 2	197.77	233.39	166.11	166.11	18%	-16%	-16%
Merge 2	102.35	108.33	108.87	108.87	6%	6%	6%
SS 3	201.94	233.36	166.13	166.13	16%	-18%	-18%
Merge 3	101.51	106.35	107.88	107.88	5%	6%	6%
Job	321.77	405.77	620.01	365.51	26%	93%	14%

Tabela D.4. Cenários da carga de trabalho $pm = 4$ e $ps = 1$

Tipo	**Exp. (E)**	**Simul. (S)**	**Tripathi (T)**	**Fork/Join (F)**	**S x E**	**T x E**	**F x E**
Map	14.19	16.89	13.10	13.10	19%	-8%	-8%
SS 1	199.02	227.45	165.96	165.96	14%	-17%	-17%
Merge 1	96.18	102.47	102.48	102.48	7%	7%	7%
SS 2	203.88	227.39	165.92	165.92	12%	-19%	-19%
Merge 2	102.21	102.73	104.14	104.14	1%	2%	2%
SS 3	199.54	227.48	165.96	165.96	14%	-17%	-17%
Merge 3	103.38	99.96	99.54	99.54	-3%	-4%	-4%
Job	325.58	399.06	597.81	356.20	23%	84%	9%

Referências Bibliográficas

Adve, V. S. e Vernon, M. K. (1993). The influence of random delays on parallel execution times. *SIGMETRICS Perform. Eval. Rev.*, 21:61--73.

Adve, V. S. e Vernon, M. K. (2004). Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94--136.

Apache, F. (Jun, 2011). Hadoop,. http://wiki.apache.org/hadoop.

Balakrishnan, N. (1994). Order statistics from non-identical exponential random variables and some applications. *Comput. Stat. Data Anal.*, 18:203--253.

Bienia, C.; Kumar, S.; Singh, J. P. e Li., K. (2008). The parserc benchmark suite: Characterization and architectural implications. Em *The Seventeenth International Conference on Parallel Architectures and Compilation Techniques - PACT'08*.

Condie, T.; Conway, N.; Alvaro, P.; Hellerstein, J. M.; Elmeleegy, K. e Sears, R. (2010). Mapreduce online. Em *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pp. 21--21, Berkeley, CA, USA. USENIX Association.

Courtois, P. J. (1977). *Decomposability : queueing and computer system applications*. Academic Press.

Dean, J. e Ghemawat, S. (2004). Mapreduce: simplified data processing on large clusters. Em *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pp. 10--10, Berkeley, CA, USA. USENIX Association.

Devroye, L. (1986). *Non-Uniform Random Variate Generation(originally published with*. Springer-Verlag.

Ferscha, A. (1992). A petri net approach for performance oriented parallel program design. Number Vol. 15, No. 3 (July 1992), pp. 188 -- 206. Academic Press.

Florin, G., F. C. e Natkin, S. (1991). Stochastic petri nets: properties, applications and tools. *Microelectronics and Reliability*, 31:669–695.

Foster, I. e Kesselman, C., editores (1999). *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Heidelberger, P. e Trivedi, K. S. (1983). Analytic queueing models for programs with internal concurrency. *IEEE Trans. Comput.*, 32:73--82.

Hopcroft, J. (2010). New research directions in the information age. Em *Theory and Applications of Models of Computation*, volume 6108, pp. 1–1. Springer Berlin.

Hu, L. e Gorton, I. (1997). Performance evaluation for parallel systems: A survey. Relatório técnico, University of NSW, School of Computer Science and Engineering, Sydney, Australia.

Jonkers, H. (1993). Probabilistic performance modelling of parallel numerical applications: A case study. Em *PARCO*, pp. 707–710.

Jonkers, H. (1994a). Queueing models of parallel applications: the glamis methodology. Em *Proceedings of the 7th international conference on Computer performance evaluation : modelling techniques and tools: modelling techniques and tools*, pp. 123--138, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

Jonkers, H. (1994b). Queueing models of parallel applications: the glamis methodology. Em *Proceedings of the 7th international conference on Computer performance evaluation : modelling techniques and tools: modelling techniques and tools*, pp. 123--138, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

Kendall, D. (1948). On the generalized "birth-and-death"process. *Annals of Mathematical Statistics*, 19(1).

Kleinrock, L. (1975). *Queueing Systems: Volume 2; Computer Applications*. Wiley.

Knuth, D. E. (1981). *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley.

Kriegel, H.-P.; Borgwardt, K. M.; Kröger, P.; Pryakhin, A.; Schubert, M. e Zimek, A. (2007). Future trends in data mining. *Data Mining and Knowledge Discovery*, 15(1):87--97.

Kruskal, C. P. e Weiss, A. (1985). Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.*, 11:1001--1016.

Lavenberg, S. e Reiser, M. (1980). Stationary state probabilities at arrival instants for closed queueing networks with multiple types of customers. *Journal of Applied Probability*.

Lazowska, E. D. e C., S. K. (1979). Exploiting decomposability to approximate quantiles of response times in queuing networks. *Operating Systems: Theory and Practice*, pp. 149–166.

Lazowska, E. D.; Zahorjan, J.; Graham, G. S. e Sevcik, K. C. (1984). *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Liang, D.-R. e Tripathi, S. K. (2000). On performance prediction of parallel computations with precedent constraints. *IEEE Trans. Parallel Distrib. Syst.*, 11(5):491--508.

Liao, W.-K.; Choudhary, A.; Weiner, D. e Varshney, P. (2005). Performance evaluation of a parallel pipeline computational model for space-time adaptive processing. *J. Supercomput.*, 31(2).

Little, J. D. C. (1961). A proof for the queuing formula: $l = \lambda w$. *Operations Research*, 9(3):383--387.

Mak, V. W. e Lundstrom, S. F. (1990). Predicting performance of parallel computations. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):257--270.

Mak, V. W. K. (1987). Performance prediction of concurrent systems - technical report csl-tr-87-344. Relatório técnico CSL-TR-87-344, Computer Systems Laboratory, Standford University.

Menascé, D. A. e Bennani, M. N. (2006). Analytic performance models for single class and multiple class multithreaded software servers. Em *Computer Measurement Group Conference*, pp. 475--482.

Menascé, D. A.; Dowdy, L. W. e Almeida, V. A. F. (2004). *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Murata, T., S. B. e Shatz, S. M. (1989). Detection of ada static deadlocks using petri net invariants. *IEEE Transactions on Software Engineering*, 15:314–326.

Navarro, A.; Asenjo, R.; Tabik, S. e Cascaval, C. (2009). Analytical modeling of pipeline parallelism. Em *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*.

Papoulis, A. (1964). *Probability, random variables, and stochastic processes*. McGraw-Hill series in electrical engineering. McGraw-Hill.

Raatikainen, K. E. E. (1989). Approximating response time distributions. *SIGMETRICS Perform. Eval. Rev.*, 17:190--199.

Raman, E.; Ottoni, G.; Raman, A.; Bridges, M. e August, D. (2008). Parallel-stage decoupled software pipelining. Em *Proc. IEEE/ACM CGO*.

Rege, K. M. e Sengupta, B. (1988). Response time distribution in a multiprogrammed computer with terminal traffic. *Perform. Eval.*, 8:41--50.

Reiser, M. e Lavenberg, S. S. (1980). Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27:313--322.

Salza, S. e Lavenberg, S. (1981). Approximating response time distributions in closed queueing network models of computer performance. *Symposium on Computer Performance*, p. Pages: 12.

Schweitzer, P. (1978). Approximate Analysis of Multiclass Closed Networks of Queues. Em *Int'l Conf. Stochastic Control and Optimization*.

Schwetman, H. (2001). Csim19: A powerful tool for building system models. Em *Proc. of the Winter Simulation Conf. (WSC)*.

Thomasian, A. e Bay, P. F. (1986). Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Trans. Comput.*, 35(12):1045--1054.

Trivedi, K. S. (1982). *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Tsuei, R. F. e Vernon, M. K. (1990). Diagnosing parallel program speedup limitation using resource contention models. Em *Proceedings of the 1993 ACM SIGMETRICS Conference Parallel Processing*.

Varki, E. (1999). Mean value technique for closed fork-join networks. *SIGMETRICS Perform. Eval. Rev.*, 27:103--112.

Wang, G.; Butt, A.; Pandey, P. e Gupta, K. (2009). Using realistic simulation for performance analysis of mapreduce setups. LSAP. ACM.

Winter, A. C. (2009). Hp neoview, architecture and performance. http://www.hp.com.

Zahorjan, J. (1980). *The approximate solution of large queueing network models.* Tese de doutorado, University of Toronto, Toronto, Ont., Canada, Canada. AAI0535305.