

Algoritmos para Problemas de Escalonamento em Grades

Robson Roberto Souza Peixoto

Este exemplar corresponde à redação final da
Dissertação devidamente corrigida e defendida
por Robson Roberto Souza Peixoto e aprovada
pela Banca Examinadora.

Campinas, 15 de abril de 2011.



Eduardo Candido Xavier (Orientador)

Dissertação apresentada ao Instituto de Com-
putação, UNICAMP, como requisito parcial
para a obtenção do título de Mestre em Ciência
da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Peixoto, Robson Roberto Souza
P359a Algoritmos para problemas de escalonamento em
grades/Robson Roberto Souza Peixoto-- Campinas, [S.P. :
s.n.], 2011.

Orientador : Eduardo Candido Xavier.
Dissertação (mestrado) - Universidade Estadual de
Campinas, Instituto de Computação.

1.Computação em grade (Sistema de computador).
2.Escalonamento de produção. 3.Algoritmos aproximados.
4.Algoritmos online. I.Xavier, Eduardo Candido.
II. Universidade Estadual de Campinas. Instituto de
Computação. III. Título.

Título em inglês: Algorithms for scheduling problems in grid

Palavras-chave em inglês (Keywords): 1.Computational grids
(Computer systems). 2.Scheduling. 3.Approximation algorithms.
4.Online algorithms.

Área de concentração: Teoria da Computação

Titulação: Mestre em Ciência da Computação

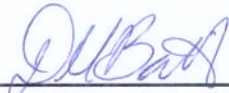
Banca examinadora:
Prof. Dr. Eduardo Candido Xavier
Prof. Dr. Edmundo Roberto Mauro Medeira
Prof. Dr. Daniel Macêdo Batista

Data da defesa: 15/04/2011

Programa de Pós-Graduação: Mestrado em Ciência da
Computação

TERMO DE APROVAÇÃO

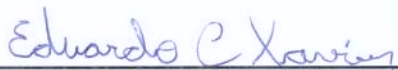
Dissertação Defendida e Aprovada em 15 de abril de 2011, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Daniel Macêdo Batista
IME / USP



Prof. Dr. Edmundo Roberto Mauro Madeira
IC / UNICAMP



Prof. Dr. Eduardo Candido Xavier
IC / UNICAMP

Algoritmos para Problemas de Escalonamento em Grades

Robson Roberto Souza Peixoto¹

Abril de 2011

Banca Examinadora:

- Eduardo Candido Xavier (Orientador)
- Edmundo Roberto Mauro Madeira
Instituto de Computação - Universidade Estadual de Campinas
- Daniel Macêdo Batista
Instituto de Matemática e Estatística - Universidade de São Paulo
- Carlile Campos Lavor (Suplente)
Instituto de Matemática, Estatística e Computação Científica - Universidade Estadual de Campinas
- Cid Carvalho de Souza (Suplente)
Instituto de Computação - Universidade Estadual de Campinas

¹Suporte financeiro de: Bolsa CAPES (processo 01 P-08503-2008) 2008–2009, Bolsa da FAPESP (processo 2008/07589-1) 2009–2010

Resumo

Nesta dissertação estudamos algoritmos para resolver problemas de escalonamento de tarefas em grades computacionais. Dado um conjunto de tarefas submetidas a uma grade computacional, deve-se definir em quais recursos essas tarefas serão executadas. Algoritmos de escalonamento são empregados com o objetivo de minimizar o tempo necessário para executar todas as tarefas (*makespan*) que foram submetidas. Nosso foco é estudar os atuais algoritmos de escalonamento usados em grades computacionais e comparar estes algoritmos. Nesta dissertação apresentamos algoritmos *onlines*, aproximados e heurísticas para o problema. Como resultados novos, provamos fatores de aproximação para o algoritmo RR quando utilizado para resolver os problemas $R; s_{it}|T_j|C_{max}$, $R; s_{it}|T_j|TPCC$, $R; s_{it}|T_j = L|C_{max}$ e $R; s_{it}|T_j = L|TPCC$ é justo. Por fim, definimos uma interface que adiciona replicação de tarefas a qualquer algoritmo de escalonamento, onde nós mostramos a aproximação desta interface, e apresentamos uma comparação via simulação dos algoritmos sem e com replicação. Nossas simulações mostram que, com a utilização de replicação, houve a redução no *makespan* de até 80% para o algoritmo Min-min. Nas nossas análises também fazemos uso da métrica RTPCC que calcula exatamente a quantidade de instruções que foram usadas para executar todas as tarefas.

Abstract

In this dissertation, we studied algorithms to solve task scheduling problems in computational grids. Given a task set that was submitted to a computational grid, the problem is to define in which resources these tasks will be executed and the order they will be executed. Scheduling algorithms are used in order to minimize the time required to execute all tasks (makespan). We studied the most recent scheduling algorithms proposed to be used in computational grids, and then compare them using simulations. In this dissertation we also present approximate algorithms and new heuristics for the problem. As new results, we proved approximation factors to the RR algorithm when applied to solve the problems $R; s_{it}|T_j|C_{max}$, $R; s_{it}|T_j|TPCC$, $R; s_{it}|T_j = L|C_{max}$ and $R; s_{it}|T_j = L|TPCC$. Finally, we defined an interface that adds task replication capability to any scheduling algorithm. We then show approximation results for algorithms using this interface, and present a comparison of well know algorithms with and without replication. This comparison is done via simulation. Our simulations show that, with replication, there was up to 80% of reduction in the makespan to some algorithms like the Min-min.

Agradecimentos

Agradeço a painho, Robson, a minha irmã, Andréia, e a Thais, por todo o carinho, ajuda e estímulos durante todo o mestrado. Também agradeço a mainha, Nádia, por todo o carinho que ela sempre me deu, do qual sinto muita falta. Agradeço ao novo xodó da família, Pedrinho, por trazer ainda mais alegria para a nossa vida. Também agradeço a minha namorada, Sumaia, por todo apoio dado no final do mestrado.

Um agradecimento especial ao meu orientador Prof. Eduardo Xavier, por sua paciência, puxões de orelha, incentivo, amizade e bate papo durante toda a orientação. O incentivo dele foi o grande responsável pela conclusão desta dissertação.

Também agradeço a todos os amigos estavam do lado nos mais variados momentos. A excelente recepção de Lechuga, Vitão, Fernandinha e Depende. As cervejas, sambas, forró, bate papo e almoços com Mariazinha, Nego Preto (vulgo Marcos), Elsão. Aos sagrados almoços aos finais de semana regrado a imagem e ação e baralho com Maria Angélica, Daniel, Willian, Guilherme, Isaura, Bunitinho (vulgo Trípodi), Nine, Priscila, Aninha Rézio, Gordinho (Fábio). As várias horas de estudo com Lehilton (me ajudou muito), Ceará (Thiago), Projeto de gente (Aline), André, Pará e Jefferson.

Por fim, agradeço à FAPESP e a CAPES pelo suporte financeiro.

Sumário

Resumo	v
Abstract	vi
Agradecimentos	vii
1 Introdução	1
1.1 Organização da Dissertação	3
1.2 Visão geral de problemas de escalonamento	3
1.3 Visão geral de grades e problemas de escalonamento em grade	7
1.3.1 Tipos de algoritmos	9
1.4 Definição geral de algoritmos de aproximação	10
2 Exemplos de Algoritmos Aproximados para Grades Computacionais	11
2.1 Algoritmo RR	11
2.2 Algoritmo LR	17
2.3 Grid Concurrent-Submission	19
3 Heurísticas para Problemas de Escalonamento em Grades	27
3.1 Algoritmo WorkQueue(WQ)	27
3.2 Min-min	28
3.3 Max-min	29
3.4 Fastest Processor to Largest Task First(FPLTF)	30
3.5 Sufferage	31
4 Novos resultados de aproximação dos algoritmos RR	34
4.1 Tarefas com tamanhos iguais	35
4.2 Tarefas com tamanhos diferentes	43
4.3 Replicação de tarefa	45

5	Análise de desempenho	47
5.1	Modelo do simulador	47
5.2	Resultados dos experimentos	49
6	Conclusão	60
6.1	Trabalhos futuros	61
	Bibliografia	62

Lista de Tabelas

2.1	Lista de algoritmos e sua aproximação	11
4.1	Lista de algoritmos atualizada e sua aproximação	34
5.1	<i>Makespan</i> (MSpan), TPCC e fator de aproximação(App) dos algoritmos replicados	58
5.2	O melhor algoritmo para cada ambiente de execução	59

Lista de Figuras

2.1	Exemplo de escalonamento, onde tarefas são elipses, e cada posição da matriz corresponde ao número máximo de instruções executadas num intervalo de tempo. O TPCC é a soma do número de instruções desde o tempo 0 até 7.5	12
2.2	Exemplo de escalonamento gerado pelo RR.	15
2.3	Grafo de tarefas G onde as tarefas têm tamanho 20 e nível l	18
4.1	É impossível deterministicamente calcular um escalonamento ótimo se o desempenho das máquinas for imprevisível.	38
4.2	Quando a última tarefa é alocada existem m vagas disponíveis.	42
4.3	O escalonamento gerado pelo algoritmo RR, e as 9 últimas tarefas escalonadas.	42
4.4	Um exemplo de instância do pior caso.	45
5.1	<i>Makespan</i> (Variabilidade das máquinas $\in U[1, 1]$)	51
5.2	<i>Makespan</i> (Variabilidade das máquinas $\in U[1, 4]$)	51
5.3	<i>Makespan</i> (Variabilidade das máquinas $\in U[1, 8]$)	52
5.4	RTPCC (Variabilidade das máquinas $\in U[1, 1]$)	52
5.5	RTPCC (Variabilidade das máquinas $\in U[1, 4]$)	53
5.6	RTPCC (Variabilidade das máquinas $\in U[1, 8]$)	53
5.7	Desperdício (Desperdício em % X Quantidade de réplicas)	55
5.8	Desperdício (Ganho em % X Quantidade de réplicas)	56

Capítulo 1

Introdução

Nesta dissertação estudamos algoritmos voltados para problemas de escalonamento de tarefas. Muitas das variações de problemas de escalonamento são problemas de otimização que pertencem à classe NP-difícil. Problemas de otimização, na sua forma geral, têm como objetivo maximizar ou minimizar uma função definida sobre um certo domínio. A teoria clássica de otimização trata do caso em que o domínio é infinito. Já no caso dos chamados problemas de otimização combinatória, o domínio é tipicamente finito; além disso, em geral é fácil listar os seus elementos e também testar se um dado elemento pertence a esse domínio. Ainda assim, a ideia ingênua de testar todos os elementos desse domínio na busca pelo melhor mostra-se inviável na prática, mesmo para instâncias de tamanho moderado.

Nesta dissertação, assumimos a hipótese de que $P \neq NP$. Dessa forma, tais problemas de escalonamento e muitos outros problemas de otimização que são NP-difíceis, não possuem algoritmos eficientes para resolvê-los de forma exata. Muitos desses problemas aparecem em aplicações práticas e há um forte apelo econômico para resolvê-los. A alocação de tarefas é uma área crítica, por exemplo, na obtenção de bom desempenho em sistemas computacionais paralelos. À medida que a tecnologia VLSI (*Very-large-scale integration*) avança, são desenvolvidos computadores com um número cada vez maior de processadores e é crescente o desenvolvimento de estratégias de escalonamento de tarefas para tais tipos de computadores. São problemas que necessitam de soluções rápidas, muitas vezes de forma *on-line*, ou seja, não existe o conhecimento prévio de todas as tarefas que serão escalonadas. Como na maioria das vezes os sistemas requerem respostas rápidas, a busca de soluções ótimas é inaceitável, devido à grande quantidade de tempo de execução desta abordagem. Assim, é necessário obter algoritmos rápidos mas que possuam uma boa garantia de desempenho em relação a solução ótima. Nesse caso, os algoritmos de aproximação *on-line* são bastante adequados, pois, além de dar garantias de desempenho, são polinomiais, sendo na sua grande maioria muito rápidos.

Apesar de estarmos focados na alocação de tarefas para computação paralela [4], muitas das variações dos problemas aparecem também em outros casos práticos como compiladores, sistemas operacionais [7] e indústria. São problemas que necessitam de soluções rápidas. Assim, é necessário obter algoritmos rápidos mas que possuam uma boa garantia de desempenho em relação à solução ótima. Consideramos problemas de alocação de tarefas, que podem estar sujeitas à várias restrições, em máquinas de tal forma a minimizar alguma função de custo associada ao problema. Exemplos de casos envolvidos nesse tipo de problema são a obtenção de escalonamentos de tarefas em computadores onde a média de atendimento de uma tarefa seja minimizada, ou que tarefas importantes tenham maior prioridade para serem finalizadas, ou mesmo a obtenção de um escalonamento que gaste tempo total mínimo.

Como não conseguimos resolver tais problemas de forma exata e eficiente, buscamos alternativas que possam ser úteis. Existem vários métodos que são utilizados na prática, como o uso de heurísticas, programação inteira, programação por restrições, métodos híbridos, algoritmos de aproximação e algoritmos probabilísticos.

Heurísticas englobam várias técnicas de resolução de problemas, com as quais obtêm-se soluções que não necessariamente são ótimas. Dentre as várias técnicas citamos a busca tabu, algoritmos genéticos e GRASP[31]. Em geral, não podemos afirmar quão boa será uma solução gerada por uma heurística, ao contrário de soluções geradas por algoritmos aproximados. Ainda assim, na prática, o uso de heurísticas tem sido promissor para a obtenção de soluções de boa qualidade.

Em programação inteira, formulamos o problema considerado como um programa linear onde algumas ou todas as suas variáveis devem assumir valores inteiros. A resolução de um programa linear inteiro (PLI), geralmente se dá através da resolução relaxada do PLI, e em seguida aplica-se um algoritmo de *branch-and-bound* para obtenção de uma solução inteira.

O desenvolvimento de algoritmos de aproximação surgiu em resposta à impossibilidade de se resolver satisfatoriamente diversos problemas de otimização NP-difíceis. Tais algoritmos são eficientes (têm complexidade de tempo polinomial) e produzem soluções que distam de no máximo um fator da solução ótima. Nesse caso, o algoritmo sacrifica a otimalidade em troca da garantia de uma solução aproximada computável eficientemente. Certamente, o interesse é, apesar de sacrificar a otimalidade, fazê-lo de forma que ainda possamos dar boas garantias sobre o valor da solução obtida, procurando ganhar o máximo em termos de eficiência computacional.

Em linhas gerais, algoritmos de aproximação são aqueles que não necessariamente produzem uma solução ótima, mas soluções que estão dentro de um certo fator da solução ótima. Esta garantia deve ser satisfeita para todas as instâncias do problema. Desta forma devemos dar uma demonstração formal deste fato.

Nesta dissertação nós estudamos um conjunto de algoritmos aproximados para diversos problemas de escalonamento. Para o algoritmo RR, nós apresentamos uma aproximação justa para os problemas $R; s_{it}|T_j|C_{max}$, $R; s_{it}|T_j|TPCC$, $R; s_{it}|T_j = L|C_{max}$ e $R; s_{it}|T_j = L|TPCC$. Também apresentamos algumas heurística onde modificamos os algoritmos Max-min e Min-min para torná-los dinâmicos. Por fim, definimos uma interface que adiciona replicação de tarefas a qualquer algoritmo de escalonamento, onde nós mostramos a aproximação desta interface, e apresentamos uma comparação via simulação dos algoritmos sem e com replicação.

1.1 Organização da Dissertação

Na seção 1.2 apresentamos brevemente uma introdução sobre problemas de escalonamento e na seção 1.3 discutimos as singularidades de escalonamento em grades computacionais. Na seção 1.4 apresentamos uma definição sobre algoritmos aproximados.

No capítulo 2 apresentamos algoritmos aproximados para o problema de escalonamento de tarefas em grades computacionais, onde cada algoritmo foi projetado para resolver o problema com condições específicas.

No capítulo 3 apresentamos heurísticas usadas em problemas de escalonamento em sistemas paralelos, destacando alguns que tiveram que ser adaptados para grades computacionais.

No capítulo 4 mostramos uma aproximação justa para o algoritmo RR. Também definimos uma interface que adiciona replicação a qualquer algoritmo de escalonamento e também mostramos qual fator de aproximação os algoritmos possuem ao usar tal interface.

No capítulo 5 mostramos os resultados de várias simulações usando os algoritmos apresentados no capítulo 3 aplicando a interface apresentada no capítulo 4.

Por fim, no capítulo 6 fazemos as considerações finais e sugerimos alguns trabalhos futuros.

1.2 Visão geral de problemas de escalonamento

Problemas de escalonamento têm sido uma das principais áreas de pesquisa no desenvolvimento de algoritmos de aproximação. Vale dizer que é atribuído a um problema de escalonamento de tarefas em computadores paralelos o primeiro algoritmo de aproximação (Graham [19]).

Nesta dissertação, usamos a seguinte notação. As tarefas a serem escalonadas (*Jobs*) são denotadas por J_j , $j \in \{1, \dots, n\}$ onde n denota o número de tarefas. As máquinas (processadores) são denotadas por M_i , e denotamos por m o número de máquinas. Cada

tarefa J_j tem um tempo de processamento p_{ij} que depende da máquina M_i onde a tarefa J_j será processada. No caso especial de termos apenas uma máquina, ou todas as máquinas idênticas, denotamos a requisição de processamento apenas por p_j . A tarefa J_j deve ser processada por uma respectiva quantidade de tempo em uma das m máquinas. Pode ocorrer que $p_{ij} = \infty$, indicando que a tarefa J_j não poderá ser executada na máquina M_i . Cada máquina pode processar no máximo uma tarefa de cada vez. Além disso, atribuímos um peso $w_j > 0$ que indica a prioridade que uma tarefa tem para ser terminada.

Cada tarefa J_j tem um tempo de liberação $r_j \geq 0$ antes do qual não pode ser processada. Denotamos por C_j o tempo em que a tarefa J_j é completada. No caso de escalonamentos preemptivos (*preemptive*), uma tarefa pode ser repetidamente interrompida e continuada depois na mesma ou em outra máquina. Em escalonamentos não preemptivos (*nonpreemptive*), uma tarefa deve ser processada de maneira ininterrupta.

Outra condição comum em escalonamentos é a precedência entre tarefas. Denotamos por $J_j \prec J_k$ se a tarefa J_j deve ser completada antes da tarefa J_k começar.

O ambiente pode ser previsível, ou seja, conhece o tamanho de todas as tarefas e o desempenho das máquinas. Ou imprevisível, onde não conhece o tamanho das tarefas e o desempenho das máquinas.

Como nem todas as condições acima precisam estar presentes em um problema de escalonamento, Graham, Lawler, Lenstra e Rinnooy Kan [20] (veja também [25]) apresentaram um esquema de classificação para estes problemas denotado pela tripla $\alpha|\beta|\gamma$. A seguir apresentamos os valores de α , β e γ para os problemas de nosso interesse:

- O termo α é a característica da máquina que pode ser 1, P ou R , onde $\alpha = 1$ denota o escalonamento em uma máquina. Quando $\alpha = P$ temos várias máquinas paralelas idênticas. O caso geral onde as máquinas não tem nenhuma relação entre si, é denotada com $\alpha = R$. Junto com as letras P e R pode-se colocar o número de máquinas no ambiente de escalonamento. Assim, $P2$ indica que temos duas máquinas idênticas em paralelo. O ; s_{it} no α significa que o poder de processamento da máquina varia com o tempo.
- O termo β pode ser vazio ou conter as características das tarefas: r_j (ou r_{ij}), $prec$, $pmtn$ e T_j . A inclusão do tipo r_j indica que as tarefas tem tempo de liberação, $prec$ indica que as tarefas podem ter precedência entre elas, $pmtn$ indica que o escalonamento pode ser preemptivo e T_j indica o tamanho das tarefas, onde podem ser todas diferentes (T_j) ou todas iguais ($T_j = L$) onde L é o número de instruções necessárias para executar uma tarefa.
- O termo γ refere-se à função objetivo. Quando temos $\gamma = \sum w_j C_j$ estamos minimizando o tempo de finalização ponderado das tarefas. Quando temos $\gamma = C_{\max}$ o objetivo é minimizar o tempo máximo para completar todas as tarefas, ou seja, o

makespan. Note que este último pode ser reduzido para o caso onde temos precedência e pesos. Para isso, basta colocar todos os pesos iguais a 0 e inserir uma nova tarefa com peso unitário e que sucede todas as demais tarefas.

A função objetivo mais usada nos problemas de escalonamento é o *makespan*, mas existem outras funções objetivos. Por exemplo, um escalonamento que reduza o tráfego de rede. Nessa dissertação só analisamos os problemas que querem otimizar o *makespan* ou o clico total de processamento (TPCC).

Outra distinção feita em relação a um algoritmo para escalonamento é se ele é *on-line* ou *off-line*. Um algoritmo *off-line* tem como entrada todos os dados (p_j, r_j, etc) relativos ao conjunto de tarefas. Neste caso o algoritmo tem previamente estes dados e constrói o escalonamento a partir deles. Já um algoritmo *on-line* constrói o escalonamento à medida que o tempo passa e que novas tarefas são liberadas. Se um processo J_j é liberado no tempo t então o algoritmo só toma conhecimento dos dados relativos a J_j no tempo t . No tempo t o algoritmo só possui dados das tarefas cujo $r_j \leq t$. O algoritmo tem que decidir o escalonamento sem conhecimento das tarefas que virão no futuro.

Construir um escalonamento de tarefas para o problema $P || \sum C_j$ é um problema polinomial [2, 23, 42]. Por outro lado, a adição de tempos de liberação ou precedência ou pesos para os tempos de finalização, torna este problema NP-difícil [5]. Até recentemente, pouco se sabia sobre algoritmos de aproximação para este problema. Com uso de novas técnicas, foi possível obter algoritmos com bons limites de desempenho em relação ao valor de uma solução ótima. Algumas destas técnicas são o uso de relaxações lineares de programas inteiros, desenvolvimento de algoritmos probabilísticos e sua desaleatorização e programação semidefinida.

O problema $1 || \sum w_j C_j$ pode ser resolvido em tempo polinomial de forma ótima usando a regra de Smith (Weighted Shortest Processing Time (WSPT)), que ordena as tarefas em ordem decrescente considerando as razões peso-tempo de processamento [42].

Para o caso *on-line* do problema $1 | r_j, pmtn | \sum w_j C_j$, Schulz e Skutella [34] obtiveram um fator de aproximação igual a $4/3$ (ver também [17, 16]) e em [6], Chekuri, Motwani, Natarajan e Stein consideram a versão *on-line* para máquinas simples obedecendo diversos tipos de grafos de precedência.

Para o caso de máquinas não relacionadas, Hall, Shmoys e Wein apresentaram um algoritmo $16/3$ -aproximado para $R | r_j | \sum w_j C_j$. Posteriormente, fazendo uso de relaxações lineares e a interpretação de sua solução como probabilidades, Schulz e Skutella [36] obtiveram um algoritmo $(2+\epsilon)$ -aproximado para o problema $R | r_{ij} | \sum w_j C_j$ e um algoritmo $(3/2 + \epsilon)$ -aproximado para o caso sem r_{ij} , onde ϵ pode ser tomado tão pequeno quanto se queira. Esse problema foi provado ser fortemente NP-difícil por Lenstra, Rinnoy Kan e Brucker [26]. Para o problema com máquinas idênticas, $P | r_j | \sum w_j C_j$, o algoritmo apresentado por Schulz e Skutella tem fator de aproximação igual a 2.

Phillips, Stein e Wein [32], obtiveram um algoritmo 2-aproximado elegante para o caso de escalonamento de tarefas com precedência em dois processadores. Além disso, é possível obter extensões deste algoritmo para obter uma 3-aproximação para os problemas $1|r_j, prec|\sum w_j C_j$ e $P|r_j, prec, p_j = 1|\sum w_j C_j$, uma 4-aproximação para $P|r_j|\sum w_j C_j$ e uma 7-aproximação para $P|r_j, prec|\sum w_j C_j$. Esses resultados podem ser generalizados para modelos nos quais as máquinas tem velocidades diferentes (veja [22, 35, 21]).

Em [33], Sahni obteve uma $1/2(1+\sqrt{2})$ aproximação para o problema com número fixo de processadores. Em [41] Skutella e Woeginger apresentam um esquema de aproximação para $P||\sum w_j C_j$. Em [1] Chekuri *et al* apresentaram esquemas de aproximação polinomial para os problemas $1|r_j|\sum C_j$, $P|r_j|\sum w_j C_j$, $P|r_j, pmtn|\sum w_j C_j$, $Rm|r_j|\sum w_j C_j$ e $Rm|r_j, pmtn|\sum w_j C_j$.

Megow e Schulz [29] estudaram a versão *on-line* dos problemas $P|r_j, pmtn|\sum w_j C_j$ e $P|r_j|\sum w_j C_j$. Eles apresentaram um algoritmo baseado na regra WSPT com aproximação igual a 2 para o caso preemptivo e também apresentaram um outro algoritmo com fator de aproximação 3.28 para o caso não preemptivo. Este último resultado foi melhorado posteriormente por Correa e Wagner [9] que apresentaram um algoritmo 2.62 aproximado. Além disso, Correa e Wagner apresentaram algoritmos probabilísticos com aproximação esperada menor do que 2.

Skutella e Uetz [40] desenvolveram os primeiros resultados para o problema de escalonamento com pesos quando o tempo das tarefas é governado por distribuições de probabilidade independente. Esses problemas são conhecidos como escalonamento estocástico.

Recentemente a versão estocástica e *on-line* dos problemas $P||\sum w_j C_j$ e $P|r_j|\sum w_j C_j$ foi estudada por Megow et al [30]. Eles analisaram várias políticas combinatórias do problema e mostraram fatores esperados de aproximação constante.

Outra técnica bastante interessante é a programação semidefinida, que foi usada com bastante sucesso para os problemas do corte máximo [18], máxima satisfatibilidade e alguns problemas de coloração [24]. Tal técnica permitiu obter limitantes melhores que os existentes para os respectivos problemas. Skutella [39] apresentou um algoritmo baseado em programação semidefinida para o escalonamento de duas máquinas não relacionadas para o escalonamento com pesos, ou seja, $R2||\sum w_j C_j$. Enquanto que com o melhor algoritmo de aproximação baseado em relaxação de programas lineares foi obtido um limitante de $3/2$, este método proporcionou uma melhora para 1,2752 usando a técnica de arredondamento de Goemans e Williamson. Essa foi a primeira vez que o método de programação semidefinida foi usado para um problema de escalonamento de tarefas.

Recentemente, Fujimoto e Hagihara [13, 15] projetaram um algoritmo $(1 + \frac{m(\ln(m-1)+1)}{n})$ aproximado para o problema de escalonamento de tarefas em grades computacionais. A função objetivo usada foi $\gamma = \sum s_{jk}$ que representa o **TPCC** (consumo total de ciclo do

processador. Do inglês *total processor cycle consumption*). Fujimoto e Hagihara criaram essa nova função objetivo por causa da inviabilidade de se usar $\gamma = C_{\max}$ (tempo máximo para executar todas as tarefas). Isso ocorre porque qualquer solução que não seja a ótima pode estar muito distante da ótima, pois o poder de processamento dos nós na grade varia durante o tempo. Por exemplo, suponha que temos uma solução ótima de tempo OPT . Se as máquinas, por um motivo qualquer, no instante OPT diminuíssem o poder de processamento por um longo período de tempo, qualquer solução que não seja a ótima estará bem distante da solução ótima. Por fim, o problema estudado por Fujimoto e Hagihara foi $R|r_j = L|\sum s_{jk} R; s_{it}|p_{ij} = L|TPCC$, onde $\sum s_{jk}$ é o TPCC e L é uma constante do modelo. Mais detalhes serão vistos na seção 2.1.

Fujimoto e Hagihara [14] também projetaram um algoritmo $(1 + \frac{L_{cp}(n) \cdot m(\ln(m-1)+1)}{n})$ aproximado para o problema $R; s_{it}|prec; p_{ij} = L|TPCC$ onde as tarefas têm precedência. Mais detalhes serão vistos na seção 2.2

Schwiegelshohn, Tchernykh e Yahyapour [37] fizeram um estudo sobre algoritmos de escalonamento online para grades computacionais onde mostraram que o algoritmo *List Scheduling* tem um desempenho ruim para grades computacionais. Eles propuseram uma melhoria para o algoritmo que garante um taxa de competitividade 5 usando a técnica *job stealing*. A ideia dessa técnica é montar um escalonamento estático e se um processador ficar ocioso, ele rouba as tarefas que foram alocadas para outro processador. Isso torna o algoritmo mais adaptado para o ambiente de Grades.

1.3 Visão geral de grades e problemas de escalonamento em grade

Nesta seção apresentamos alguns conceitos básicos de Grade Computacional e sobre Escalonamento em Grade.

A grade computacional (grade) foi criada graças ao avanço das tecnologias de rede e o baixo custo dos componentes computacionais. Grade é um sistema que, através da utilização de *interfaces* e protocolos padronizados, abertos e de propósito geral, coordena recursos que não estão sujeitos a um controle centralizado com o objetivo de fornecer qualidades de serviços não triviais [12]. A agregação cria um serviço viabilizando o processamento de aplicações de larga escala. Para melhor uso desses recursos, ou para suprir algum requisito específico, é necessário que a grade faça um bom escalonamento das tarefas. Os requisitos podem ser bem variados, como minimizar o tempo máximo de computação, diminuir consumo de processamento ou diminuir o tráfego na rede.

Alguns termos que serão usados neste trabalho são descritos a seguir.

- Uma **tarefa** é uma unidade atômica a ser escalonada por um escalonador e atribuída

a um recurso.

- As **propriedades** de uma tarefa são os parâmetros como requisito de CPU, requisito de memória, prazo final, prioridade, etc.
- Um **trabalho** é um conjunto de tarefas atômicas que serão executadas num conjunto de recursos. Os trabalhos podem ser estruturados recursivamente, mostrando que o trabalho é composto de outros trabalhos ou tarefas.
- Um **recurso** é qualquer objeto, físico ou lógico, necessário para que o processo seja executado. Pode ser CPU, memória, enlace de rede ou dispositivo de armazenamento.
- Um **nó** é uma entidade autônoma composta por um ou mais recursos.
- Um **escalonamento das tarefas** é um mapeamento das tarefas para um selecionado grupo de recursos que podem estar distribuídos em múltiplos domínios administrativos.

Os algoritmos de escalonamento que foram desenvolvidos anteriormente para sistemas paralelos e distribuídos não são os mais adequados para Grade já que esses algoritmos assumem a existência de um controle dos recursos e os sistemas são homogêneos. Nas Grades todo recurso é doado pelo dono deste. O dono tem controle total do recurso, como o controle de acesso, quantidade de memória disponibilizada, máximo de consumo de CPU, permissão de acesso e várias outras regras que podem ser configuradas. Problemas maiores, como o caso do usuário desligar seu recurso ou sobrecarregá-lo a ponto de não sobrar recurso para a Grade, podem ocorrer. Por isso o escalonamento em Grades necessita de um estudo mais focado nas características específicas dessa classe de sistema. Dong e Akl [11] listaram características que influenciam o projeto de algoritmos de escalonamento em grades:

- **Heterogeneidade e Autonomia**

Mesmo a heterogeneidade sendo um tópico já estudado em outros problemas de sistemas distribuídos, ele ainda é um problema que precisa ser explorado considerando as características de grade. Em Grades existem recursos que estão distribuídos em múltiplos domínios na internet, e tanto os nós quanto as sub-redes internas são heterogêneas.

Em sistemas paralelos e distribuídos tradicionais, os recursos são controlados pelo sistema, então é mais fácil prever possíveis eventos que influenciem nas decisões. Em Grades cada recurso é de controle total do dono, e não há como controlar ou tentar prever os eventos, já que existem outros processos concorrentes de que a Grade não

tem conhecimento e controle. Além disso, existem políticas locais que o dono do recurso configura que a Grade tem que respeitar.

- **Variação do desempenho**

As Grades estão sob um ambiente dinâmico onde o desempenho do recursos varia durante o tempo. Além disso, não há controle central desses recursos. Por exemplo, no caso de um *cluster*, o sistema tem controle de todos os recursos, e caso uma tarefa nova precise ser escalonada, ele a escala com sua visão de todo o ambiente. Já na Grade, se uma tarefa concorrente local rodar no sistema ele não terá controle sobre ela. Em relação aos sistemas tradicionais, a flutuação do desempenho dos recursos é a principal característica que influencia o projeto do algoritmo de escalonamento de tarefas.

- **Escolha dos recursos e Separação dos dados**

A heterogeneidade da Grade pode fazer com que os nós de armazenamento estejam em domínios diferentes, e isso tem que ser levado em conta no escalonamento das tarefas, já que o tráfego dos dados irá influenciar no tempo final do escalonamento.

Essas características fazem com que a Grade tenha características únicas e que devem ser levadas em consideração no projeto de algoritmos de escalonamento.

1.3.1 Tipos de algoritmos

Os algoritmos de escalonamento têm sempre duas das características citadas abaixo. São estáticos ou dinâmicos com ou sem predição. Essas características são definidas abaixo:

Estáticos São algoritmos que definem todo o escalonamento baseados nas informações iniciais [10] da grade computacional, ou seja, eles não acompanham as mudanças que ocorrem na grade.

Dinâmicos São algoritmos que precisam tomar decisões várias vezes durante todo o tempo que o sistema está em execução. Cada vez que o algoritmo precisa tomar alguma decisão, ele usa as informações da grade naquele instante. Este tipo de algoritmo é o mais adequado para grades computacionais, já que as grades são sistemas bem dinâmicos e heterogêneos.

Sem predição Algoritmos sem predição, significa que não precisam conhecer as características da grade. Só precisam conhecer a quantidade de máquinas que estão disponíveis e em alguns casos a quantidade de tarefas.

Com predição Algoritmos com predição precisam conhecer características mais específicas sobre a grade, como o poder de processamento atual de uma máquina da grade, o tamanho da tarefas, a velocidade da rede, ou qualquer outra informação que seja relevante para o algoritmo. Com essas informações o algoritmo consegue tomar decisões mais interessantes, mas essas informações são custosas, complicadas e as vezes pouco confiáveis [10, 15].

Nós estudamos e implementamos algoritmos com essas características, principalmente os algoritmos dinâmicos.

1.4 Definição geral de algoritmos de aproximação

Nesta seção discutimos brevemente as principais abordagens algorítmicas que utilizamos nesta dissertação, que são algoritmos aproximados e os *on-lines*.

Dado um algoritmo \mathcal{A} , para um problema de minimização, se I for uma instância para este problema, vamos denotar por $\mathcal{A}(I)$ o valor da solução devolvida pelo algoritmo \mathcal{A} aplicado à instância I e vamos denotar por $\text{OPT}(I)$ o correspondente valor para uma solução ótima de I . Diremos que um algoritmo tem um fator de aproximação α , ou é α -aproximado, se $\mathcal{A}(I)/\text{OPT}(I) \leq \alpha$, para toda instância I . No caso dos algoritmos probabilísticos, consideramos a desigualdade $E[\mathcal{A}(I)]/\text{OPT}(I) \leq \alpha$, onde a esperança $E[\mathcal{A}(I)]$ é tomada sobre todas as escolhas aleatórias feitas pelo algoritmo. É importante ressaltar que algoritmos de aproximação têm complexidade de tempo polinomial.

Ao elaborar um algoritmo aproximado, o primeiro passo é buscar uma prova de seu fator de aproximação. Outro aspecto interessante é verificar se o fator de aproximação α demonstrado é o melhor possível. Para isto, devemos encontrar uma instância cuja razão entre a solução obtida pelo algoritmo e sua solução ótima é igual, ou tão próxima quanto se queira, de α . Neste caso, dizemos que o fator de aproximação do algoritmo é justo, ou seja, seu fator de aproximação não pode ser melhorado.

Os algoritmos on-line são aqueles que recebem suas entradas de dados ao passar do tempo. As decisões são tomadas com base nas informações que o algoritmo já possui. É importante grifar que toda decisão já tomada não pode ser anulada.

Capítulo 2

Exemplos de Algoritmos Aproximados para Grades Computacionais

Nesta seção apresentamos alguns exemplos de algoritmos aproximados para grades computacionais que foram estudados. Na seção 2.1 apresentamos o algoritmo RR $(1 + \frac{m(\ln(m-1)+1)}{n})$ aproximado para o problema $R; s_{it} | T_j = L | TPCC$. Na seção 2.2 apresentamos o algoritmo LR $(1 + \frac{L_{cp}(n) \cdot m(\ln(m-1)+1)}{n})$ para o problema $R; s_{it} | prec; T_j = L | TPCC$. Na seção 2.3 apresentamos o algoritmo Grid Concurrent-Submission para o problema $GP_m | size_j; r_j | C_{max}$ com fator 3 de aproximação para o problema off-line e fator 5 para o problema on-line.

A tabela abaixo resume os algoritmos que foram estudados:

α	β	γ	observação	algoritmo	aproximação
$R; s_{it}$	$T_j = L$	$TPCC$	não preditivo	RR	$(1 + \frac{m(\ln(m-1)+1)}{n})$
$R; s_{it}$	$prec; T_j = L$	$TPCC$	não preditivo	RR	$(1 + \frac{L_{cp}(n) \cdot m(\ln(m-1)+1)}{n})$
GP_m	$size_j; r_j$	C_{max}	on-line	RR	3
GP_m	$size_j; r_j$	C_{max}	off-line	RR	5

Tabela 2.1: Lista de algoritmos e sua aproximação

2.1 Algoritmo RR

Nesta seção apresentamos detalhadamente o algoritmo RR para o problema $R; s_{it} | T_j = L | TPCC$ proposto por Fujimoto e Hagihara [13] que obteve uma $(1 + \frac{m(\ln(m-1)+1)}{n})$ aproximação. Neste problema temos n tarefas de tamanho L a serem escalonadas em m máquinas, e cada máquina varia o poder de processamento em função do tempo. O algoritmo tem o objetivo de minimizar a função TPCC que calcula o total de instruções gastas

para executar todas as tarefas do início até o instante que a última tarefa é concluída, instante C_{max} .

O TPCC é formalmente definido como:

$$\sum_{i=1}^m \sum_{t=0}^{\lfloor C_{max} \rfloor - 1} s_{i,t} + \sum_{i=1}^m (C_{max} - \lfloor C_{max} \rfloor) s_{i, \lfloor C_{max} \rfloor}.$$

Onde $s_{i,t}$ é a velocidade da máquina M_i durante o intervalo de tempo $[t, t + 1)$, medidos em quantidade de instruções que podem ser executadas, onde t é um inteiro não negativo, e C_{max} é o tempo máximo de execução de todas as tarefas para um dado escalonamento, ou seja, *makespan*. Então o TPCC é a quantidade de instruções executadas por todos os processadores durante o instante $[0, C_{max}]$. Na figura 2.1 damos um exemplo de cálculo do TPCC para um escalonamento com 5 tarefas e 3 máquinas. Neste exemplo o TPCC é $\underbrace{(15 + \frac{2}{2})}_{M_1} + \underbrace{(14 + \frac{4}{2})}_{M_2} + \underbrace{(8 + \frac{6}{3})}_{M_3} = 43$.

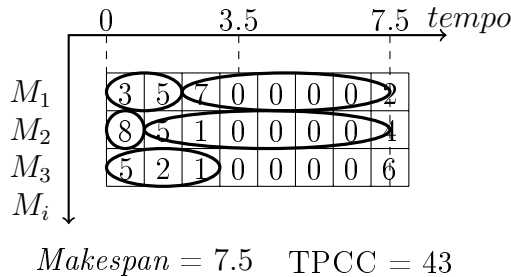


Figura 2.1: Exemplo de escalonamento, onde tarefas são elipses, e cada posição da matriz corresponde ao número máximo de instruções executadas num intervalo de tempo. O TPCC é a soma do número de instruções desde o tempo 0 até 7.5

O algoritmo proposto por Fujimoto e Hagihara considera que:

- As tarefas têm **tamanhos** iguais, e representamos pelo número de instruções necessárias para executar a tarefa.
- A **velocidade** do processador é o número de instruções que podem ser computadas por unidade de tempo.
- A velocidade da máquina M_i não varia durante o tempo $[t, t + 1)$ e tem velocidade igual a $s_{i,t}$. É assumido que a velocidade é zero quando o processador estiver muito sobrecarregado ou desligado.

- Não está previsto adição ou remoção de máquinas e também não está previsto ocorrência de falhas de qualquer natureza.
- Seja T um conjunto de n tarefas independentes de mesmo tamanho L . Seja m o número de máquinas numa grade computacional. Um escalonamento S numa grade com m máquinas e n tarefas é definido como um conjunto finito de triplas $\langle j, i, t \rangle$, que satisfazem as regras **R1** e **R2** (abaixo), onde $1 \leq j \leq n$ é o índice da tarefa, $1 \leq i \leq m$ é o índice do máquina e t é o instante de tempo em que a tarefa T_j iniciou.
 - R1)** Para cada $1 \leq j \leq n$, existe uma tripla $\langle j, i, t \rangle \in S$. Ou seja, todas as tarefas $T_j \in T$ serão executadas.
 - R2)** Não existem duas triplas $\langle j, i, t \rangle, \langle j', i, t' \rangle \in S$ com $t \leq t' < t + d$ onde $t + d$ é o tempo de conclusão da tarefa T_j . Ou seja, um processador não pode executar mais de uma tarefa ao mesmo tempo.
- A **replicação de tarefas** ocorre quando uma tarefa T_j é executada em mais de um processador.

No Algoritmo 1 apresentamos o algoritmo RR proposto por Fujimoto e Hagihara. Neste algoritmo é utilizado um **anel** que é uma estrutura de dados circular onde os elementos tem relação de ordem total, ou seja, não existem dois ou mais elementos na mesma ordem. O anel tem uma **cabeça** que identifica qual é o atual elemento. Essa cabeça pode mudar de posição para o **próximo** elemento do anel. Quando a cabeça se move para o próximo elemento, ela irá para o próximo elemento de maior ordem, mas se o elemento atual for o de ordem máxima, então ela irá para o de ordem mínima. Os elementos do anel podem ser removidos e se o elemento for a cabeça do anel, a cabeça do anel terá que apontar para o próximo elemento. O Algoritmo 1 usa um anel de tarefas com essas propriedades.

O algoritmo RR (Algoritmo 1) faz uso da estrutura de dados **anel** R que gerencia as tarefas que ainda não foram concluídas. O anel tem uma **cabeça** que aponta para a **tarefa atual**. As tarefas que são gerenciadas pelo anel seguem a lógica do Round-Robin, ou seja, assim que a tarefa atual T_r é atribuída a um processador livre, a cabeça do anel R muda para a próxima tarefa no anel.

A seguir será descrito o funcionamento do algoritmo RR. No início do escalonamento, todas as máquinas têm uma tarefa atribuída. Se alguma tarefa finalizar, RR recebe o resultado da tarefa e atribuirá uma nova tarefa, que ainda não foi atribuída a nenhuma máquina, para a máquina que ficou livre. O RR repete esses passos até que todas as tarefas tenham sido atribuídas a alguma máquina. Logo, nesse momento exatamente m tarefas estão sendo executadas, ou seja, ainda não concluíram o processamento. Essas m tarefas são gerenciadas no anel R . O algoritmo executará os seguintes passos até que

Algoritmo 1: Algoritmo RR

Entrada: Um conjunto T de tarefas, Um número m de máquinas**Saída:** O escalonamento dinâmico (S) de T em m máquinas**início**Seja Q uma fila de tarefas;Enfileire todas as tarefas de T em Q ;Desenfileire as primeiras m tarefas que estão em Q e atribua cada uma a uma máquina distinta;**repita**

Espere a finalização de uma tarefa em alguma máquina;

 Desenfileire a primeira tarefa que está em Q e atribua a máquina que está livre;**até** $Q = \emptyset$;Seja U o conjunto de todas as tarefas que ainda não concluíram;▷ Note que $|U| = m$;Seja R um anel com todas as tarefas de U ;

▷ A cabeça do anel é inicializada arbitrariamente;

repita

Espere a finalização de uma tarefa em alguma máquina;

Mate todos as instâncias da tarefa que concluiu;

 Remova a tarefa que completou de R ; **se** $R \neq \emptyset$ **então** **enquanto** *uma máquina livre existir* **faça** Atribua a máquina livre a atual tarefa de R ; Mova a cabeça de R ; **até** $R = \emptyset$;

todas as m tarefas sejam concluídas. Se qualquer instância de uma tarefa T_j concluir na máquina M_i , ele irá receber o resultado, abortará todas as instâncias de T_j na grade e removerá T_j do anel R . Enquanto existir processador livre, RR irá pegar a tarefa atual do anel R e criará uma réplica da tarefa no processador livre, movendo a cabeça do anel para a próxima tarefa.

Na Figura 2.2 apresentamos um exemplo de escalonamento gerado pelo algoritmo RR para um conjunto de cinco tarefas $\{T_1, T_2, \dots, T_5\}$ e três máquinas $\{M_1, M_2, M_3\}$. Nesse caso a fila Q é inicializada com $\langle T_1, T_2, \dots, T_5 \rangle$ e o anel R é inicializado com $\{T_1, T_4, T_5\}$.

O algoritmo RR tem as seguintes propriedades:

Propriedade 1. *Durante todo o tempo de execução das tarefas, o algoritmo RR não permite que nenhuma máquina fique ociosa.*

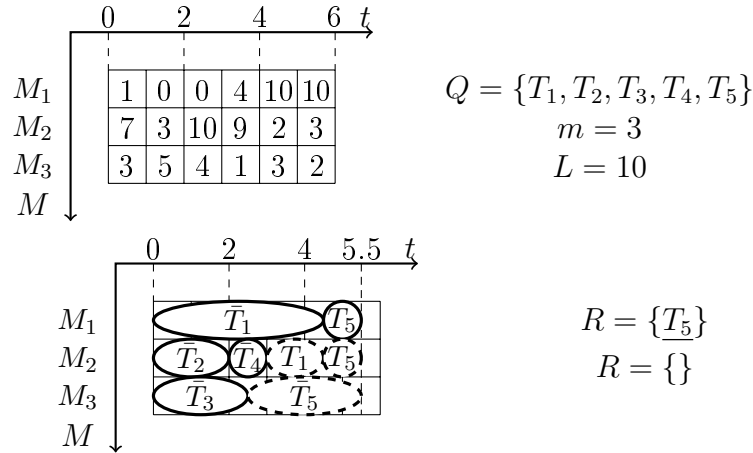


Figura 2.2: Exemplo de escalonamento gerado pelo RR.

Enquanto o $|Q| > 0$ todas as máquinas estão ocupados. E quando $|Q| = 0$ a replicação de tarefas irá garantir que não exista máquina ociosa.

Definindo a i -ésima última tarefa como a tarefa que concluiu seguindo a ordem de trás para a frente, temos a seguinte propriedade.

Propriedade 2. *Para toda tarefa i ($m \leq i \leq n$), RR nunca replica a i th-última tarefa.*

Na figura 2.2, o 1-ésima é a T_5 , o 2-ésima é a T_1 e o 3-ésima é a T_4 . Isso ocorre porque se existem m processadores então somente m tarefas entram no anel R . Já que a replicação de tarefas só ocorre quando um processador fica livre e isso só acontecerá quando concluir uma das m tarefas, então somente $m - 1$ tarefas podem ser replicadas.

Definimos como **instância de grupo** de uma tarefa T_j no instante t o conjunto de todas as instâncias de T_j no instante t . Por exemplo, o grupo de réplicas de T_1 no instante $t = 4$ é $\{\bar{T}_1$ em T_1 e T_1 em $T_2\}$ e de T_5 é $\{\bar{T}_5$ em $T_3\}$.

Propriedade 3. *A diferença entre quaisquer dois pares de conjuntos de instância no instante t é de no máximo um.*

Por exemplo, para T_1 e T_5 no instante tempo $t = 4$ temos uma diferença de 1, já que T_1 está executando em duas máquinas diferentes e T_5 só está executando em uma máquina. Já no instante de tempo $t = 1$ só existe uma instância das tarefas T_1 , T_2 e T_3 . Essa propriedade ocorre porque o algoritmo RR replicará a tarefa que a cabeça do anel está apontando. Após a replicação a cabeça do anel apontará para o próximo elemento do anel. Já que o anel tem relação de ordem total, então a tarefa que acabou de ser replicada só poderá ser replicada após todas as outras tarefas do anel forem replicadas. Então a diferença entre a quantidade de réplicas de cada tarefa será de no máximo um.

Usando as definições e propriedades acima iremos mostrar alguns lemas para provar a garantia de desempenho do algoritmo RR.

Lema 1. *Seja $f(x)$ o número de réplicas da x th-última tarefa para as $m - 1$ tarefas replicáveis. Então $f(x)$ satisfaz a inequação abaixo.*

$$\lfloor \frac{m-x}{x} \rfloor \leq f(x) \leq \lceil \frac{m-x}{x} \rceil$$

Demonstração. No instante que a x -ésima tarefa concluir, temos x tarefas originais e $m-x$ réplicas em execução. Usando a propriedade 3 sabemos que a diferença da quantidade de réplicas é de no máximo 1, então o número de réplicas por tarefa varia entre $\lfloor \frac{m-x}{x} \rfloor$ e $\lceil \frac{m-x}{x} \rceil$

Por exemplo, na figura 2.2 temos um ambiente com $m = 3$ e a 1ª tarefa tem 2 réplicas. Já a 2ª tarefa tem 1 réplica. \square

Lema 2. *O número total de réplicas em um escalonamento gerado pelo RR é de até no máximo $m \ln(m - 1) + m$.*

Demonstração. É conhecido que

$$\lim_{n \rightarrow \infty} \sum_{k=1}^n \frac{1}{k} = \ln(n) + \gamma$$

onde a constante de Euler $\gamma \approx 0.5772156$. Então

$$\lim_{n \rightarrow \infty} \sum_{k=1}^n \frac{1}{k} \leq \ln(n) + 1$$

Para cada x th tarefa, temos no máximo $\lceil \frac{m-x}{x} \rceil$ réplicas. Então o total de réplicas (r) é:

$$\begin{aligned} r &= \sum_{x=1}^{\min(n, m-1)} \lceil \frac{m-x}{x} \rceil \\ &\leq \sum_{x=1}^{m-1} \lceil \frac{m-x}{x} \rceil \\ &\leq \sum_{x=1}^{m-1} \left(\frac{m-x}{x} + 1 \right) \\ &\leq \sum_{x=1}^{m-1} \left(\frac{m}{x} \right) \\ r &\leq m \ln(m - 1) + m \end{aligned}$$

\square

Lema 3. *O TPCC ótimo é pelo menos nL .*

Demonstração. Temos n tarefas de tamanho L . Todas devem ser executadas e portanto o TPCC será pelo menos nL . \square

Teorema 1. *O TPCC do RR é no máximo $(1 + \frac{m \ln(m-1)+m}{n})$ vezes o TPCC ótimo.*

Demonstração. Pelo lema 3 temos que o ótimo é pelo menos nL e pelo lema 2 temos um limite máximo para o número de réplicas. Já que cada tarefa tem o tamanho L e temos n tarefas originais, então no máximo serão executados $(n + m \ln(m-1) + m)L$ instruções. Logo a aproximação é limitada por

$$\frac{(n + m \ln(m-1) + m)L}{nL}$$

\square

2.2 Algoritmo LR

Nesta seção apresentaremos com detalhes o algoritmo LR para o problema $R; s_{it}|prec; T_j = L|TPCC$ apresentado por Fujimoto e Hagihara [14] que obteve uma $(1 + \frac{L_{cp}(n) \cdot m(\ln(m-1)+1)}{n})$ aproximação. Neste problema temos n tarefas de tamanho L a serem escalonadas em m máquinas com poder de processamento não previsível e variável, e respeitando o grafo de tarefas. O grafo de tarefas é um grafo acíclico direcionado $G = (T, E)$ que define a precedência das tarefas. O algoritmo tem o objetivo de otimizar a função TPCC.

Como foi dito acima, o **grafo de tarefas** é um grafo acíclico direcionado $G = (T, E)$ onde cada nó do conjunto de nós T tem peso (tamanho) L e as precedências são representadas por um conjunto de arestas direcionadas E . Cada aresta é representada pela notação $(T_j, T_{j'})$ e significa que o nó T_j é o **predecessor imediato** de $T_{j'}$. A tarefa que não tem predecessor imediato é chamado de **tarefa de entrada**. Dado um caminho no grafo de tarefas, definimos o **tamanho** de um caminho como o número de tarefas neste. O **nível** da tarefa T_j , denotado por $level(T_j)$, é o tamanho do maior caminho partindo de uma tarefa de entrada até a tarefa T_j . É importante ressaltar que o nível de uma tarefa é imutável, mesmo se suas precedências já estiverem concluídas. O maior caminho de um grafo de tarefas é chamado de *caminho crítico*, denotado por $L_{cp}(n)$. A figura 2.3 mostra um grafo de tarefas, composto por 5 tarefas e todas com tamanho igual a 20. Nesse grafo de tarefas a tarefa de entrada é a T_1 , o conjunto $E = \{(T_1, T_2), (T_1, T_3), (T_1, T_4), (T_3, T_5), (T_4, T_5)\}$, e o caminho crítico tem tamanho 3. O nível das tarefas $\{T_1, T_2, T_3, T_4, T_5\}$ é $\{1, 2, 2, 2, 3\}$. Cada nível l tem n_l tarefas, então $\sum_{l=1}^{L_{cp}(n)} n_l = n$.

O **escalonamento** S de G numa grade com m máquinas é um conjunto finito de triplas $\langle j, i, t \rangle$ que satisfazem as regras **R1** e **R2** visto na seção 2.1 e a regra **R3** abaixo,

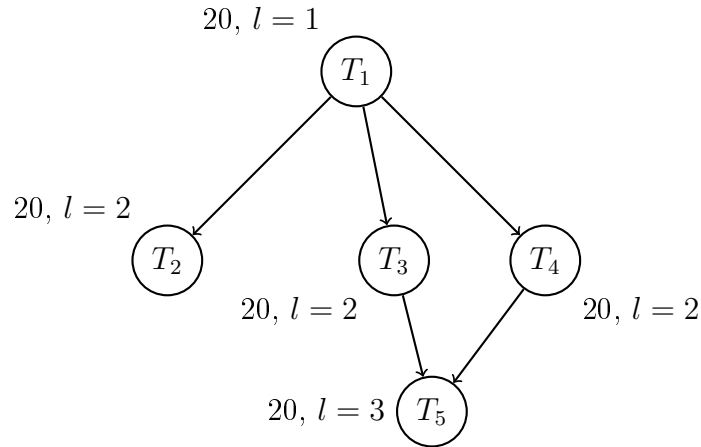


Figura 2.3: Grafo de tarefas G onde as tarefas têm tamanho 20 e nível l .

onde $j \in T$, $i \in (1 \leq i \leq m)$ é o índice da máquina e t é o instante de tempo em que a tarefa T_j iniciou. A tripla $\langle j, i, t \rangle$ significa que a tarefa T_j iniciou a execução na máquina M_i no instante t .

R3) Se $(j, j') \in E$ e $\langle j, i, t \rangle \in S$, então existe uma tripla $\langle j', i', t' \rangle \in S$ com $t + d \leq t'$ onde $t + d$ é o tempo de conclusão da tarefa j . Ou seja, a tarefa j' só irá executar depois do término da tarefa j .

O algoritmo LR [2] separa as tarefas em grupos V_l , onde $l \in (1 \leq l \leq L_{cp}(n))$ é o nível da tarefa no grafo de tarefa, calculado usando a função $level(T_j)$. Por exemplo, na Figura 2.3 temos os grupos $V_1 = \{T_1\}$, $V_2 = \{T_2, T_3, T_4\}$ e $V_3 = \{T_5\}$. Cada grupo V_l é escalonado usando o algoritmo RR [1]. A separação em grupos garante que a precedência das tarefas sejam respeitadas e o uso do algoritmo RR garante a aproximação apresentada na seção 2.1 em cada nível l .

Agora vamos mostrar a garantia de desempenho do algoritmo LR. Usando a propriedade 1 e o lema 2 da seção 2.1 podemos provar o teorema abaixo.

Teorema 2. O algoritmo LR é $(1 + \frac{L_{cp}(n)(m(\ln(m-1)+1))}{n})$ -aproximado.

Demonstração. Considere uma instância I onde temos n tarefas de tamanho L a serem escalonadas em m máquinas. Já que n tarefas de tamanho L precisam ser executadas, então o *makespan* ótimo $OPT \geq nL$. Seja $L_{cp}(n)$ o tamanho do caminho crítico do grafo de tarefas. Seja n_l a quantidade de tarefas no nível l ($1 \leq l \leq L_{cp}(n)$). Usando a propriedade 1 e o lema 2, sabemos que são computados no máximo

$$(n_l + m(\ln(m-1) + 1))L$$

instruções usando o algoritmo **RR** para cada nível l . Já que o **LR** executa o algoritmo **RR** $L_{cp}(n)$ vezes, temos o fator de aproximação:

Algoritmo 2: Algoritmo LR

Entrada: Um grafo de tarefas G e cada tarefa com mesmo tamanho L , número m de máquinas

Saída: O escalonamento dinâmico (S) de n tarefas em m máquinas

início

▷ Calcule o nível de todas as tarefas em G ;

para $j = 0$ **até** n **faça**

$l = level(T_j)$ ▷ Nível da tarefa;

 Adicionar T_j no conjunto V_l ;

▷ Seja $L_{cp}(n)$ o caminho crítico do grafo;

para $l = 1$ **até** $L_{cp}(n)$ **faça**

 Seja V_l o conjunto de tarefas no nível l de G ;

 Escalonar V nas m máquinas usando o **Algoritmo RR**;

$$\frac{LR(I)}{OPT(I)} = \frac{\sum_{l=1}^{L_{cp}(n)} (n_l + m(\ln(m-1) + 1))L}{nL} \quad (2.1)$$

$$= \frac{\sum_{l=1}^{L_{cp}(n)} (n_l L) + \sum_{l=1}^{L_{cp}(n)} (m(\ln(m-1) + 1))L}{nL} \quad (2.2)$$

$$= \frac{nL + L_{cp}(n)(m(\ln(m-1) + 1))L}{nL} \quad (2.3)$$

$$= 1 + \frac{L_{cp}(n)(m(\ln(m-1) + 1))}{n} \quad (2.4)$$

□

2.3 Grid Concurrent-Submission

Nessa seção iremos apresentar o algoritmo *Grid Concurrent-Submission* para o problema $GP_m | size_j; r_j | C_{max}$ apresentado por Schwiegelshohn, Tchernykh e Yahyapour [37]. Eles obtiveram uma aproximação com fator 3 para o problema *off-line* e um fator 5 para o problema *on-line*. Neste problema temos uma Grade, GP_m , composta por m máquinas, onde cada máquina M_i tem m_i processadores. As máquinas são organizadas em ordem crescente pela quantidade de processadores, ou seja, $m_{i-1} \leq m_i$. Por definição $m_0 = 0$.

As tarefas são independentes e submetidas em diferentes instantes r_j . Cada tarefa tem o nível de paralelismo, $size_j$, necessário para executar a tarefa, ou seja, para que uma máquina M_i possa executar a tarefa J_j é necessário que $size_j \leq m_i$. A execução da tarefa J_j na máquina M_i é denotada por $i = a(j)$. O tempo de submissão da tarefa é

$r_j \geq 0$ e o tempo de execução da tarefa é $p_j > 0$. O tempo p_j só é conhecido depois que a tarefa termina, ou seja, não existe um mecanismo de predição no ambiente. O instante de conclusão de uma tarefa J_j em um escalonamento S é denotado por $C_j(S)$, ou para simplificar C_j .

Para um escalonamento ser considerado viável tem que obedecer as duas restrições abaixo:

- $r_j + p_j \leq C_j$ — A tarefa J_j tem que terminar antes do tempo de execução somado ao tempo de submissão.
- $m_i \geq \sum_{J_j | C_j - p_j \leq t < C_j \wedge i = a(j)} size_j$ — Para todo instante t e cada máquinas M_i , a quantidade de processadores que estão alocados para executar tarefas não pode ser superior ao número máximo m_i de processadores da máquina M_i . Ou seja, a quantidade de tarefas que estão executando em paralelo em uma máquina M_i não pode usar mais capacidade de processamento do que a disponível.

O *makespan* de um escalonamento S é definido como $C_{max}(S) = \max_j \{C_j(S)\}$, ou seja, é o instante de tempo que a última tarefa terminou. O objetivo do algoritmo é reduzir o *makespan*, onde o *makespan* ótimo é denotado por

$$C_{max}^* = \min_{\text{escalonamento legal } S} C_{max}(S).$$

O algoritmo e as análises assumem inicialmente que o $r_j = 0$. É importante mostrar que nesse ambiente temos uma forte restrição que é o nível de paralelismo mínimo que uma tarefa exige para ser executada. Isso faz com que certas tarefas só possam ser escalonadas num subconjunto de máquinas.

Os autores apresentaram o limitante inferior do *makespan* ótimo no caso de submissão concorrente (uma máquina pode executar mais de uma tarefa ao mesmo tempo) para o problema de escalonamento em grades computacionais:

$$C_{max}^* \geq \max \left\{ \max_j p_j, \max_{1 \leq i \leq m} \frac{\sum_{j | size_j > m_{i-1}} p_j \cdot size_j}{\sum_{v=i}^m m_v} \right\}.$$

O limitante é o máximo entre os dois limitantes abaixo:

A O maior tempo de execução de uma tarefa.

B Dado um índice de máquina i , considera-se todas as tarefas que necessitam de mais processadores do que m_{i-1} , ou seja, $size_j > m_{i-1}$. Todas estas tarefas devem ser executadas necessariamente nas máquinas M_i, \dots, M_m . Portanto usa-se um

limitante conhecido, que é o total de processamento necessário para estas tarefas dividido pelo total de processadores disponíveis. Isto é feito para cada i e toma-se o maior de todos os valores.

Agora vamos começar a definir os tipos de dados usados pelo algoritmo.

Definição 1. Para cada máquina M_i existem três diferentes categorias de tarefas:

1. $A_i = \{J_j | \max\{\frac{m_i}{2}, m_{i-1}\} < size_j \leq m_i\}$
2. $B_i = \{J_j | m_{i-1} < size_j \leq \frac{m_i}{2}\}$
3. $H_i = \{J_j | \frac{m_i}{2} < size_j \leq m_{i-1}\}$

A categoria A_i é um conjunto com todas as tarefas que não podem ser executadas na máquina M_{i-1} e que precisam de mais que 50% dos processadores da máquina M_i . O conjunto B_i tem todas as tarefas que não podem ser executadas na máquina M_{i-1} , mas que precisam de menos que 50% de processadores da máquina M_i . E o conjunto H_i contém as tarefas que podem ser executadas na máquina M_{i-1} , mas que precisam de mais que 50% de processadores da máquina M_i .

Toda tarefa J_j está no conjunto A_i ou B_i para algum índice i , já que $A_i \cup B_i = \{J_j | m_{i-1} < size_j \leq m_i\}$. Se $m_{i-1} \geq \frac{m_i}{2}$, então J_j faz parte de algum conjunto A_i , caso contrário J_j faz parte do conjunto B_i ou H_i . Se $m_{i-1} = \frac{m_i}{2}$, então $B_i = \emptyset$ e $H_i = \emptyset$. Nunca podemos ter $B_i \neq \emptyset$ e $H_i \neq \emptyset$ pois os dois conjuntos são excludentes: o primeiro considera $m_{i-1} < \frac{m_i}{2}$ enquanto o segundo $\frac{m_i}{2} < m_{i-1}$. Ou seja, se um conjunto não for vazio, necessariamente o outro conjunto é vazio. Todas as tarefas que estão em B_i não estarão contidas em nenhum outro conjunto, mas as tarefas que estão em H_i necessariamente estão em A_{i-1} ou em H_{i-1} . Note que se $H_i \cap H_{i-1} \neq \emptyset$ então $A_{i-1} \subseteq H_i$. Para ver isso note que se J_j pertence a H_i então $\frac{m_i}{2} < size_j$ e se J_j também pertence a H_{i-1} então $size_j \leq m_{i-2}$. Logo $\frac{m_{i-1}}{2} < \frac{m_i}{2} < m_{i-2}$. Logo qualquer $J_{j'} \in A_{i-1}$ tem $size_{j'} > m_{i-2} > \frac{m_i}{2}$ e portanto pertence a H_i .

Por exemplo, dado um conjunto de $m = 4$ máquinas, $M = \{m_1 = 2, m_2 = 4, m_3 = 5, m_4 = 7\}$, e um conjunto com 7 tarefas, $J = \{size_1 = 1, size_2 = 2, size_3 = 3, size_4 = 4, size_5 = 5, size_6 = 6, J_7 = 7\}$ as categorias ficam assim:

- $A_1 = [1 < size_j \leq 2] = \{J_2\}$
 $B_1 = [0 < size_j \leq 1] = \{J_1\}$
 $H_1 = [1 < size_j \leq 0] = \emptyset$
- $A_2 = [2 < size_j \leq 4] = \{J_3, J_4\}$
 $B_2 = [2 < size_j \leq 2] = \emptyset$
 $H_2 = [2 < size_j \leq 2] = \emptyset$

- $A_3 = [4 < size_j \leq 5] = \{J_5\}$
 $B_3 = [4 < size_j \leq 2.5] = \emptyset$
 $H_3 = [2.5 < size_j \leq 4] = \{J_3, J_4\}$
- $A_4 = [5 < size_j \leq 7] = \{J_6, J_7\}$
 $B_4 = [5 < size_j \leq 3.5] = \emptyset$
 $H_4 = [3.5 < size_j \leq 5] = \{J_4, J_5\}$

Algoritmo 3: Grid Concurrent-Submission

Entrada: Um conjunto n de tarefas, Um número m de máquinas

Saída: O escalonamento dinâmico (S) de n em m máquinas

início

para $i \leftarrow 1$ **até** m **faça**

$L_i \leftarrow A_i$;

$S_i \leftarrow H_i$;

 Update;

repita

para $i \leftarrow 1$ **até** m **faça**

enquanto existir uma quantidade suficiente de processadores ociosos na máquina i **faça**

 Escalone uma tarefa de L_i na máquina i ;

 Remova a tarefa escalonada de todas as listas L_k ;

se alguma lista $L_k = \emptyset$ **então**

 Update;

até todas as tarefas estiverem escalonadas;

Agora vamos apresentar o algoritmo *Grid Concurrent-Submission* [3] e o procedimento Update. No Algoritmo 3, para cada máquina M_i existe uma lista principal L_i e uma lista auxiliar S_i . A lista L_i contém todas as tarefas que estão prontas para serem escalonadas na máquina M_i e é inicializada com as tarefas da categoria A_i . A lista S_i contém todas as tarefas do conjunto H_i que ainda não foram transferidas para L_i . As duas listas contêm tarefas que exigem mais do que 50% dos processadores da máquina M_i .

O Procedimento Update é responsável pela manutenção das listas L_i e S_i de todas as máquinas. Ela também garante que a lista L_i nunca ficará vazia caso exista alguma tarefa em qualquer lista L_k para $k < i$. Sempre que alguma lista L_i está vazia, ela é atualizada seguindo a ordem abaixo:

1. Caso exista alguma tarefa na lista auxiliar S_i , o algoritmo colocará as tarefas que estão na lista S_i e que também já estão na lista L_{i-1} na lista L_i . Depois o procedi-

Procedimento Update

```

para  $i \leftarrow 1$  até  $m$  faça
  se  $L_i = \emptyset$  então
    se  $i \neq 1$  e  $S_i \neq \emptyset$  então
       $L_i \leftarrow L_{i-1} \cap S_i;$ 
       $S_i \leftarrow S_i \setminus L_i;$ 
    senão se  $B_i \neq \emptyset$  então
       $L_i \leftarrow B_i;$ 
    se  $i \neq 1$  e  $L_{i-1} \neq \emptyset$  e  $S_i = \emptyset$  então
       $L_i \leftarrow L_{i-1};$ 

```

mento remove estas tarefas da lista S_i . Essas tarefas precisam de mais da metade dos processadores da máquina M_i .

2. Caso existam tarefas na categoria B_i , essas tarefas são colocadas na lista L_i . Já que todas as tarefas contidas nessa categoria precisam de menos da metade dos processadores da máquina, a máquina consegue executar tarefas em paralelo. Caso exista mais de uma tarefa na categoria B_i , a máquina M_i terá mais do que a metade dos processadores em uso.
3. Caso uma tarefa J_j faça parte das categorias $H_{i+1} \cap A_{i-1}$, essa tarefa fará parte da lista L_{i+1} assim que todas as tarefas de A_{i+1} e $A_i \cap H_{i+1}$ já estiverem sido escalonadas. Mas para isso acontecer é necessário que o procedimento Update coloque na lista L_i todas as tarefas que estão em L_{i-1} e que ainda não foram escalonadas.

Seja a tarefa $J_{j'}$ a última tarefa a ser escalonada na máquina $M_{i'}$. Nós iremos mostrar como o algoritmo *Grid Concurrent-Submission* evita que a maioria dos processadores da máquina M_i fiquem ociosos antes que a tarefa $J_{j'}$ seja escalonada, para cada máquina i' onde $1 \leq i' \leq m$.

Lema 4. *O algoritmo Grid Concurrent-Submission garante que para todas as máquinas da Grade mais que 50% dos seus processadores estarão ocupados executando tarefas antes do instante de submissão da última tarefa para a máquina.*

Demonstração. Lembrando que primeiro todas as tarefas das classes A_i e H_i são escalonadas na máquina M_i , temos que durante a execução dessas tarefas todas as máquinas terão mais de 50% dos processadores ocupados. Após o escalonamento dessas tarefas, todas as tarefas da classe B_i serão executadas, mas como elas precisam de menos que 50% dos processadores, elas podem ser executadas paralelamente. Caso as tarefas que estão

em B_i acabem ou não sejam o suficiente para ocupar pelo menos $\frac{m_i}{2}$ processadores, então o algoritmo irá escalonar as tarefas que fazem parte das listas $L_{i'}$ para $i' < i$, lembrando que essas tarefas podem ser executadas em paralelo com as tarefas da categoria B_i e/ou entre elas, já que essas tarefas têm grau de paralelismo menor do que $\frac{m_i}{2}$. Se mesmo assim a máquina não foi ocupada em pelo menos 50%, significa que a última tarefa da máquina M_i foi escalonada. \square

Para provarmos o fator de aproximação, será mostrado no lema abaixo que para um escalonamento não balanceado as maiores máquinas não executam tarefas de máquinas menores.

Lema 5. *Seja a tarefa $J_{j'}$ a última tarefa a ser escalonada ($C_{j'}(S) = C_{max}$) num escalonamento S gerado pelo Grid Concurrent-Submission. Se uma máquina M_i com $i < a(j')$ tiver pelo menos $\frac{m_i}{2}$ processadores ociosos em um instante $t < C_{j'} - p_{j'}$ então o algoritmo Concurrent-Submission irá produzir um escalonamento S' com o mesmo makespan ($C_{max}(S') = C_{max}(S)$) se todas as tarefas nos conjuntos $A_{i'}$ e $B_{i'}$ com $i' \leq i$ forem removidas.*

Demonstração. A afirmação está correta se nenhuma máquina M_k para $k > i$ executar tarefas dos conjuntos $A_{i'}$ e $B_{i'}$ com $i' \leq i$.

Vamos assumir que exista uma tarefa J_j no conjunto $A_{i'}$ ou $B_{i'}$ com $i' \leq i$ e que essa tarefa é executada na máquina M_k para algum $k > i$. Imposto pelo procedimento Update, a tarefa J_j só pode entrar na lista L_z para $i < z \leq k$ se todas as tarefas de A_z e B_z já estiverem escalonadas. Por esse motivo, a remoção de todas as tarefas que estão nas categorias $A_{i'}$ e $B_{i'}$ não irá influenciar o instante de conclusão das tarefas A_z e B_z , para $i < z \leq k$.

Por fim, temos $J_{j'} \in A_v$ ou $J_{j'} \in B_v$ para alguma máquina M_v onde $v > i$ visto que a tarefa $J_{j'}$ não iniciou na máquina M_i no instante t ou antes do instante t . Por consequência, a remoção de todas as tarefas nos conjuntos $A_{i'}$ e $B_{i'}$ para $i' \leq i$ não irá reduzir o $C_{max}(S)$. \square

Agora vamos mostrar o fator competitivo do algoritmo *Grid Concurrent-Submission*.

Teorema 3. *O algoritmo Grid Concurrent-Submission garante que $\frac{C_{max}}{C_{max}^*} < 3$ para qualquer entrada e qualquer configuração de Grade para o caso de submissão concorrente.*

Demonstração. Seja $J_{j'}$ uma tarefa com $C_{j'}(S) = C_{max}$ em um escalonamento S produzido pelo algoritmo *Grid Concurrent-Submission*. Se existe uma máquina M_i para $i < a(j')$ e pelo menos $\frac{m_i}{2}$ processadores estão ociosos antes do instante $C_{j'} - p_{j'}$ então nós removemos todas as tarefas dos conjuntos $A_{i'}$ e $B_{i'}$ para $i' \leq i$. Devido ao lema 5, o fator $\frac{C_{max}}{C_{max}^*}$ continua

inalterado, pois os valores C_{max} e o C_{max}^* não se alteram. Se necessário esse procedimento pode ser executado repetidamente.

Devido ao lema 5, nós podemos assumir que existe uma máquina M_k do qual cada máquina M_z para $k \leq z \leq a(j')$, mais que 50% dos processadores estão sempre ocupados antes do instante $C_{j'} - p_{j'}$ e todas as máquinas $M_{i'}$ para $i' < k$ podem ser ignorados já que não podem executar as tarefas que estão nos conjuntos A_z e B_z para $z > k$.

Agora assumamos que exista algum instante $t < C_{j'} - p_{j'}$ tal que no máximo 50% dos processadores de alguma máquina $M_{i'}$ para $i' > a(j')$ estão executando tarefas no instante t em S . Então nós temos $L_{i'} = \emptyset$ e $L_{a(j)} \neq \emptyset$ no instante t . Novamente isso não é possível devido a execução do procedimento Update. Ou seja, todas as máquinas que são maiores do que a máquina que executou a última tarefa estão com mais que 50% dos processadores ocupados até o instante $C_{j'} - p_{j'}$, senão o procedimento Update haveria alocado a tarefa $J_{j'}$ em alguma máquina $M_{i'}$.

Então temos

$$\begin{aligned} C_{max}(S) &= p_{j'} + C_{j'} - p_{j'} \\ &< p_{j'} + 2 \cdot \frac{\sum_{J_j \in A_v \cup B_v | v \geq k} p_j \cdot size_j}{\sum_{v \geq k} m_v} \\ &\leq C_{max}^* + 2C_{max}^*. \end{aligned}$$

□

Depois de apresentar a aproximação da versão *off-line* iremos apresentar uma versão modificada do algoritmo apresentado por Schwiegelshohn e Tchernykh [37] usando o resultado apresentado por Shmoys, Wein e Williamson [38]. Essa modificação irá aumentar, no máximo, o fator competitivo em um fator de 2, resultando em $\frac{C_{max}(S)}{C_{max}^*} < 6$.

Os conjuntos A , B e H deixaram de ser estáticos e passaram a ser dinâmicos. Ou seja, uma vez que uma tarefa é escalonada, elas são removidas de todos os conjuntos e das listas L e S .

Agora todas as listas L_i e S_i são removidas sempre que uma nova tarefa é submetida e o procedimento Update é chamado para inicializar as listas. Esse algoritmo é chamado de *Grid Over-Time-Submission*

A análise de desempenho do *Grid Over-Time-Submission* é baseada na análise do algoritmo *Grid Concurrent-Submission*.

Teorema 4. *O algoritmo Grid Over-Time-Submission garante $\frac{C_{max}}{C_{max}^*} < 5$ para todos os dados de entrada no caso de submissão ao longo do tempo.*

Demonstração. Vamos supor que a ultima tarefa do escalonamento S tem tempo de submissão r . Após o tempo r o algoritmo *Grid Concurrent-Submission* é usado. Mas antes

de usar o algoritmo, é importante lembrar que o algoritmo já foi usado anteriormente e por isso os processadores podem estar ocupados executando tarefas que foram submetidas anteriormente. As tarefas que estão no conjunto A e H precisam esperar até que tenham processadores o suficiente para executá-las. Durante esse tempo, $\frac{m_i}{2}$ processadores ou mais podem estar ociosos em uma máquina M_i . Mas como todas as listas L_i foram esvaziadas esse tempo é limitado pelo tempo máximo de execução de alguma tarefa. Portanto, as condições dos lemas 4 e 5 são válidas após o instante $r + \max_j p_j$ e teremos o fator de aproximação do algoritmo. Usando o limitante apresentado no início da seção e a inequação $r < C_{max}^*$ também aplicado ao caso de submissão ao longo do tempo, temos

$$C_{max}(S) < r + \max_j p_j + 3C_{max}^* < 5C_{max}^*$$

□

Capítulo 3

Heurísticas para Problemas de Escalonamento em Grades

Neste capítulo vamos apresentar as heurísticas que foram usados nos experimentos do capítulo 5. Todas as heurísticas que serão apresentadas neste capítulo usam as definições abaixo.

J_j — Tarefa j .

M_i — Máquina i .

p_{ij} — Tempo de processamento da tarefa J_j na máquina M_i .

c_{ij} — Instante de conclusão da tarefa J_j na máquina M_i .

r_i — Instante quando a máquina M_i estará livre para executar outra tarefa.

Nas seções 3.1, 3.2, 3.3, 3.4 e 3.5 apresentaremos respectivamente os algoritmos WQ, DMin-min, DMax-min, DFPLTF e Sufferage. Apresentamos os algoritmos DMin-min e o DMax-min são versões dinâmicas dos algoritmos Min-min e Max-min para se adequar ao ambiente de grades computacionais.

3.1 Algoritmo WorkQueue(WQ)

O algoritmo WorkQueue(WQ) é um clássico algoritmo de escalonamento [15] que foi desenvolvido para máquinas paralelas homogêneas. O WQ é um algoritmo muito simples que atribui uma tarefa a uma máquina arbitrariamente.

Como pode ser visto no algoritmo 4, o algoritmo toma decisão de forma aleatória, simplesmente alocando a tarefa que tem em mãos no primeiro processador que ficar livre.

Algoritmo 4: Algoritmo WorkQueue

Entrada: Um conjunto J de tarefas, um número m de processadores**Saída:** O escalonamento S dinâmico de J sobre m máquinas**início** **enquanto** *Existir processador livre* **faça** └ Remova uma tarefa que está em J e atribua o processador que está livre; **repita**

└ Espere por retorno de resultado de algum processador;

 └ Remova uma tarefa que está em J e atribua o processador que está livre; **até** $J = \emptyset$;

Espera-se com este algoritmo que a máquina mais rápida execute mais tarefas. É uma ideia simples, fácil de implementar, não precisa ter o conhecimento prévio do tamanho das tarefas e da velocidade dos processadores, mas que não obtém bons resultados. Existe um problema quando o processador mais lento é sorteado com as tarefas grandes. Por causa disso normalmente o *makespan* usando este algoritmo é maior do que os outros algoritmos.

3.2 Min-min

O algoritmo Min-min (Algoritmo 5) é uma heurística apresentada em [28] e tem como ideia básica executar as tarefas que irão terminar primeiro. Esse algoritmo é estático e precisa conhecer o poder de processamento de todas as máquinas em todos os instantes de tempo e também precisa saber o tamanho de todas as tarefas. A ideia é montar um escalonamento onde uma tarefa J_j sempre irá executar na máquina que a tarefa tiver o menor tempo de conclusão. O comportamento comum desse algoritmo é executar paralelamente pequenas tarefas deixando por ultimo as tarefas grandes.

Em um ambiente de grade computacional não é possível conhecer o poder de processamento das máquinas em todos os instantes de tempo t , então fizemos uma adaptação do algoritmo. Sempre que uma ou mais máquinas ficarem ociosas o algoritmo vai escalonar as tarefas que ainda não foram escalonadas no conjunto de máquinas que estão ociosas. Nós chamamos esse novo algoritmo de DMin-min.

Esse algoritmo precisa conhecer o tamanho das tarefas e o poder de processamento das máquinas do instante que vai fazer o escalonamento. Isso é uma operação custosa e que dificulta a implementação [15].

Algoritmo 5: Algoritmo Min-min

Entrada: Um conjunto J de tarefas, um conjunto M de máquinas**Saída:** Escalonamento de J sobre m máquinas**início**

```

para todo  $J_j \in J$  faça
  para todo  $M_i \in M$  faça
     $c_{ij} = p_{ij} + r_i$ ;
  enquanto  $J \neq \emptyset$  faça
    para cada  $J_j \in J$  faça
      Achar o menor  $c_{ij}$  e a respectiva máquina;
      Encontre a tarefa  $J_j$  com o menor  $c_{ij}$ ;
      Atribua a tarefa  $J_j$  para a máquina  $M_i$ ;
      Remova  $J_j$  de  $J$ ;
    para todo  $M_i \in M$  faça
      Atualize  $r_i$ ;
    para todo  $J_j \in J$  faça
      Atualize  $c_{ij}$ ;

```

3.3 Max-min

O Max-min é uma heurística apresentada em [28] e tem como ideia básica executar as tarefas grandes primeiro. Ou seja, para cada máquina ele irá achar a tarefa com o maior tempo de conclusão, e destas irá selecionar o que tiver o menor tempo de conclusão. A ideia do algoritmo é escalonar primeiro as tarefas grandes, deixando por ultimo as pequenas tarefas.

Assim como o Min-min, o Max-min (Algoritmo 6) também foi adaptado para o ambiente dinâmico das grades computacionais. A única alteração é que a escolha da tarefa ocorre quando uma ou mais máquinas ficarem ociosas.

Nos testes que realizamos o algoritmo Max-min obteve resultados melhores do que o Min-min principalmente se tivermos poucas tarefas grandes e muitas tarefas pequenas. Nesse cenário as tarefas grandes são executadas paralelamente com as tarefas pequenas. Assim como o Min-min, o algoritmo precisa ter conhecimento sobre o tamanho das tarefas e o poder de processamento das máquinas.

Algoritmo 6: Algoritmo Max-min**Entrada:** Um conjunto J de tarefas, um conjunto M de máquinas**Saída:** Escalonamento de J sobre m máquinas**início**

```

para todo  $J_j \in J$  faça
  para todo  $M_i \in M$  faça
     $p_{ij} = p_{ij} + r_j$ ;
  enquanto  $J \neq \emptyset$  faça
    para cada  $J_j \in J$  faça
      Achar o menor  $c_{ij}$  e a respectiva máquina;
      Encontre a tarefa  $J_j$  com o maior  $c_{ij}$ ;
      Atribua a tarefa  $J_j$  para a máquina  $M_i$ ;
      Remova  $J_j$  de  $J$ ;
    para todo  $M_i \in M$  faça
      Atualize  $r_i$ ;
    para todo  $J_j \in J$  faça
      Atualize  $c_{ij}$ ;

```

3.4 Fastest Processor to Largest Task First(FPLTF)

O algoritmo *Fastest Processor to Largest Task First* (FPLTF) apresentado em [10, 8, 15, 13] é um algoritmo de escalonamento estático que tem como principal ideia escalonar as maiores tarefas nos processadores mais rápidos. Mas as grades computacionais são sistemas heterogêneos e dinâmicos, ou seja, o recurso que está disponível para a grade varia com o tempo, então escalonamentos estáticos não percebem essas alterações.

Para resolver esse problema foi desenvolvido o *Dynamic Fastest Processor to Largest Task First* (DFPLTF). O DFPLTF executa o FPLTF que monta todo escalonamento. Quando uma tarefa é concluída, ele coloca todas as tarefas que não foram concluídas ou que ainda não foram escalonadas novamente na sacola de tarefas. O DFPLTF executa novamente o FPLTF, remontando o escalonamento de todas as tarefas que estão na sacola, assim o DFPLTF tomará as decisões com base nas informações atuais da grade computacional.

O DFPLTF precisa conhecer qual é o tamanho da tarefa (*Task Size*), poder de processamento da máquina (*Host Speed*) e quanto poder de processamento está disponível para a grade (*Host Load*). Com essas informações ele calcula o custo da tarefa

$$p_{ij} = \frac{\frac{TaskSize}{HostSpeed}}{1 - HostLoad}.$$

Temos o pseudo-código do DFPLTF no Algoritmo 7.

Algoritmo 7: Algoritmo DFPLTF

Entrada: Um conjunto J de tarefas, um conjunto M de máquinas

Saída: Escalonamento (S) dinâmico de J sobre m máquinas

início

repita

para cada $J_j \in J$ **faça**

para cada $M_i \in M$ **faça**

$c_{ij} = p_{ij} + r_j$;

enquanto $J \neq \emptyset$ **faça**

para todo $J_j \in J$ **faça**

 Achar o menor c_{ij} e a respectiva máquina;

 Encontre a tarefa J_j com o maior c_{ij} ;

 Atribua a tarefa J_j para a máquina M_i ;

 Remova J_j de J ;

para todo $M_i \in M$ **faça**

 Atualize r_i ;

para todo $J_j \in J$ **faça**

 Atualize c_{ij} ;

 Espere por retorno de resultado de algum processador;

 Colocar todas as tarefas que ainda não foram concluídas e não estão executando em J ;

para todo $M_i \in M$ **faça**

 Atualize r_i ;

para todo $J_j \in J$ **faça**

 Atualize c_{ij} ;

até $J = \emptyset$;

3.5 Sufferage

Assim como o DFPLTF, o Sufferage [3, 28, 8] é um algoritmo dinâmico que toma decisões com base na estimativa do custo de execução da tarefa. Mas diferente do FPLTF, ele faz uso do sofrimento (*suffer*) para selecionar qual será a próxima tarefa a ser escalonada. Definindo *suffer* como

$$suffer_{ij} = \text{segundo melhor } c_{ij} - \text{melhor } c_{ij}$$

onde o *suffer* é o custo da tarefa não ser executada na máquina que traz o menor tempo de término desta. O algoritmo 8 tem como objetivo escalonar primeiro as tarefas mais sofridas e escalonar as tarefas nas máquinas em que estas tenham o menor tempo de término possível.

Algoritmo 8: Algoritmo Sufferage

Entrada: Um conjunto J de tarefas, um conjunto M de máquinas**Saída:** Escalonamento de J sobre m máquinas**início** **para todo** $J_j \in J$ **faça** **para todo** $M_i \in M$ **faça** $c_{ij} = p_{ij} + r_j$; **enquanto** $J \neq \emptyset$ **faça** **para cada** i **faça** Marcar M_i como não associado; **para cada** $J_j \in J$ **faça** Achar o menor $c[1]_{ij}$; Achar o segundo menor $c[2]_{ij}$; $suffer = c[2]_{ij} - c[1]_{ij}$; **se** M_j *não está associado a uma tarefa* **então** Associar J_j a M_i ; Remover J_j de J ; Marcar M_i como associado; **senão se** *o suffer da tarefa J_k que está atualmente associada ao processador M_i é menor do que o suffer da tarefa J_j* **então** Desassociar a tarefa J_k e devolver a J ; Associar J_i ao processador M_i ; Remover J_i do conjunto J ; **para todo** $M_i \in M$ **faça** Atualize r_i ; **para todo** $J_j \in J$ **faça** Atualize c_{ij} ;

Espere por retorno de resultado de algum processador;

 Colocar todas as tarefas que ainda não foram concluídas e não estão executando em J ; **para todo** $M_i \in M$ **faça** Atualize r_i ; **para todo** $J_j \in J$ **faça** Atualize c_{ij} ;

Capítulo 4

Novos resultados de aproximação dos algoritmos RR

Neste capítulo iremos apresentar novas análises do fator de aproximação do algoritmo RR para o caso de tarefas iguais, na seção 4.1, e para o caso de tarefas diferentes, na seção 4.2. Na seção 4.3 apresentamos uma interface que adiciona replicação de tarefas a qualquer algoritmo de escalonamento e também mostramos a aproximação de qualquer algoritmo de escalonamento que use essa interface.

Atualizando a Tabela 2.1 com as análises que são apresentadas neste capítulo, temos o conjunto de algoritmos abaixo:

α	β	γ	observação	algoritmo	aproximação
$R; s_{it}$	$T_j = L$	C_{max}	preditivo	RR	Ótimo
$R; s_{it}$	$T_j = L$	$TPCC$	preditivo	RR	Ótimo
$R; s_{it}$	T_j	C_{max}	não preditivo	RR	Não existe
$R; s_{it}$	T_j	$TPCC$	não preditivo	RR	m
$R; s_{it}$	$T_j = L$	$TPCC$	não preditivo	Interface \mathcal{A}	$\min\{(1 + \frac{(m-1)^2}{n}), m\}$
$R; s_{it}$	T_j	$TPCC$	não preditivo	Interface \mathcal{A}	m
$R; s_{it}$	$prec; T_j = L$	$TPCC$	não preditivo	RR	$(1 + \frac{L_{cp}(n) \cdot m(\ln(m-1)+1)}{n})$
GP_m	$size_j; r_j$	C_{max}	on-line	RR	3
GP_m	$size_j; r_j$	C_{max}	off-line	RR	5

Tabela 4.1: Lista de algoritmos atualizada e sua aproximação

4.1 Tarefas com tamanhos iguais

Nesta seção vamos mostrar que o algoritmo RR gera um resultado ótimo para os problemas $R; s_{it}|T_j = L|C_{max}$ e $R; s_{it}|T_j = L|TPCC$ onde o poder de processamento das máquinas é conhecido. Depois iremos mostrar que no problema $R; s_{it}|T_j = L|TPCC$ após o instante OPT gerado pelo algoritmo RR faltam no máximo $\frac{m}{2}$ tarefas a serem executadas. Isto leva a uma aproximação assintoticamente igual ao que Fujimoto e Hagihara apresentaram em [13], mas vamos mostrar que essa aproximação é justa apresentando instâncias em que $\frac{m}{2}$ tarefas serão executadas depois do instante OPT pelo algoritmo RR.

Cada instância do problema é composta por um conjunto de n tarefas de tamanho L a serem escalonadas em m máquinas sem relação entre si. Em todos os problemas o objetivo é gerar um escalonamento S que minimize o *Total Processor Cyle Consumption* (TPCC) ou o C_{max} .

Nós iremos analisar os problemas citados acima em ambiente imprevisível ou previsível. Num ambiente previsível é conhecido o poder de processamento de todas as máquinas e o tamanho de todas as tarefas. Já no ambiente imprevisível, essas informações não são conhecidas.

Pela definição apresentada na seção 2.1, o TPCC calcula a quantidade de instruções que todos os processadores podem executar do início ao fim do escalonamento. Ou seja, quanto maior o *makespan*, maior será o TPCC, e quanto menor o *makespan*, menor é o TPCC. Logo, um TPCC ótimo implica num *makespan* ótimo, e vice versa. A partir disso temos a propriedade abaixo.

Propriedade 4. *Um escalonamento S tem um *makespan* ótimo se e somente se o TPCC for ótimo.*

Vamos mostrar características de um escalonamento com *makespan* ótimo. Nós mostraremos que um escalonamento com *makespan* ótimo sem replicação pode ser transformado em um escalonamento com replicação onde todas as máquinas ficam ocupadas até o tempo do *makespan*.

Lema 6. *Para toda instância I do problema $R; s_{it}|T_j = L|C_{max}$, existe um escalonamento ótimo estendido $EOPT(I)$ onde:*

1. *Para cada máquina M_i , não existe tempo ocioso entre o término de uma tarefa e o início de outra.*
2. *Todas as máquinas executam tarefas até o tempo do *makespan* ótimo.*

Demonstração. Seja $OPT(I)$ um escalonamento com *makespan* ótimo com tempo OPT . Nós iremos construir o escalonamento ótimo $EOPT(I)$ com o mesmo instante de término.

Para provar (1) suponha que existam algumas máquinas M_i em $OPT(I)$ que ficam ociosas entre o término de uma tarefa e a atribuição de outra, ou seja, após T_j terminar existe um tempo ocioso para uma tarefa $T_{j'}$ começar a ser executada. Se movermos a tarefa $T_{j'}$ para iniciar logo após ao termino de T_j , isso não irá gerar acréscimo ao *makespan*.

Para provar (2) basta replicarmos tarefas ainda em execução em processadores ociosos até o tempo OPT . Note que serão escalonadas cópias de tarefas de tamanho L e que essas tarefas podem não terminar até o instante OPT . As cópias de tarefas que ultrapassarem o instante OPT serão abortadas no instante OPT , assim não iremos gerar um acréscimo no *makespan*.

O novo escalonamento $EOPT(I)$ tem *makespan* ótimo e também tem a propriedade de não ter máquinas ociosas desde o instante de início até o instante de término do escalonamento. \square

A partir do lema 6 conseguimos mostrar que um escalonamento ótimo para o problema $R; s_{jk}|T_j = L|C_{max}$ pode ser visto como uma escalonamento ótimo para o problema $R; s_{jk}|T_j = L|TPCC$ que permita que as tarefas sejam replicadas. A partir de agora vamos levar em consideração o problema de otimização do TPCC.

Dado um escalonamento ótimo para o problema de tarefas com tamanhos iguais, vamos considerar cada tarefa no escalonamento como uma **vaga** onde qualquer outra tarefa pode ser atribuída, já que todas as tarefas tem tamanho iguais. Agora podemos estudar o problema como um problema de alocar n vagas nas máquinas minimizando o TPCC. Nós iremos mostrar que o algoritmo RR [1] gera um escalonamento que é idêntico ao ótimo em número de vagas escalonadas.

Lema 7. *Considere um escalonamento S gerado pelo algoritmo RR onde ele escalona n vagas de mesmo tamanho L . O escalonamento gerado tem o mesmo *makespan* do escalonamento ótimo considerando n tarefas de tamanhos iguais.*

Demonstração. Seja um escalonamento ótimo para alocação de n tarefas e seu escalonamento ótimo estendido $EOPT(I)$ gerado pelo lema 6. Note que em $EOPT(I)$ uma tarefa é abortada somente quando o escalonamento acaba. Além disso, todas as máquinas estarão em execução até o instante do *makespan* ótimo, e essa é a primeira vez que n tarefas são finalizadas. Não é difícil de ver que o RR, considerando o problema de alocação de n vagas, irá gerar um escalonamento igual ao $EOPT$, uma vez que sempre que uma máquina fica ociosa é escalonada uma vaga. Quando a n -ésima vaga é escalonada, o algoritmo começa a replicar vagas nas máquinas ociosas até a primeira vez em que n vagas tenham cada uma, tido L instruções executadas. \square

Quando todas as tarefas têm tamanhos iguais, o escalonamento gerado pelo algoritmo RR, até o instante que a ultima tarefa é alocada, é similar a um $EOPT(I)$. A única

diferença entre eles é na ordem que algumas tarefas são alocadas. Depois das n vagas atribuídas, o RR inicia a fase de replicação, e a diferença acontece aqui pois o RR aborta as réplicas que já terminaram. Se o desempenho das máquinas são previsíveis, então o RR produzirá um escalonamento ótimo.

Teorema 5. *O algoritmo RR pode ser modificado para ter o makespan e o TPCC ótimo quando todas as tarefas tem tamanho L e o desempenho das máquinas são previsíveis. Além disso o escalonamento é gerado em tempo polinomial e não é necessário replicação.*

Demonstração. Seja EOPT o escalonamento estendido ótimo satisfazendo o lema 6. Considere cada tarefa atribuída em EOPT como uma vaga onde é possível atribuir qualquer tarefa. Nós podemos atribuir a qualquer vaga em EOPT as tarefas de acordo a alguma ordem definida pelo algoritmo RR. Já que todas as tarefas têm o mesmo tamanho, a ordem de atribuição das tarefas não interfere no *makespan* e nem no TPCC. Até o instante *makespan* em EOPT nós temos n tarefas diferentes executadas, isso significa que sabendo o desempenho das máquinas nós podemos construir o escalonamento EOPT antecipadamente e atribuir uma tarefa diferente para cada uma das n diferentes vagas que acabarem mais cedo. Para conseguir o escalonamento EOPT, pelo lema 7 nós só precisamos executar uma simulação do RR nos dando a velocidade das máquinas e a alocação das n vagas. Então o algoritmo só precisa atribuir as n tarefas usando o escalonamento. Também note que tendo calculado a simulação do escalonamento não é necessário o uso de replicação.

Para calcular a simulação do escalonamento, o algoritmo atribui n tarefas e inicia a fase de replicação. No máximo m tarefas estarão em execução quando a fase de replicação começa, e claramente cada uma das tarefas podem ter no máximo m réplicas. Por consequência o algoritmo executa $O(n + m^2)$ atribuições que é polinomial em n e m . \square

Quando a velocidade dos processadores é imprevisível o problema de determinar um *makespan*/TPCC ótimo não pode ser resolvido deterministicamente. Considere o simples exemplo na Figura 4.1: Nós sabemos que até o tempo do *makespan* ótimo, 3 tarefas serão concluídas. Suponha que exatamente 3 tarefas serão concluídas. Quando T_1 termina, as tarefas T_2 e T_3 ainda estão em execução e nós temos uma vaga na máquina M_1 após a tarefa T_1 . Nós sabemos com certeza que T_2 ou T_3 irá terminar no instante *OPT*, e suponha que somente uma delas irá terminar. Dessa maneira é impossível decidir quem das duas tarefas, T_2 ou T_3 , será escalonada depois de T_1 .

Então no caso em que a velocidade dos processadores é imprevisível não é possível alcançar um escalonamento com *makespan* ótimo.

A partir de agora nós iremos considerar o caso onde a capacidade dos processadores são imprevisíveis. Nesse caso podemos garantir que no instante do *makespan* ótimo no máximo $\lfloor m/2 \rfloor$ tarefas estarão em execução num escalonamento gerado pelo algoritmo RR.

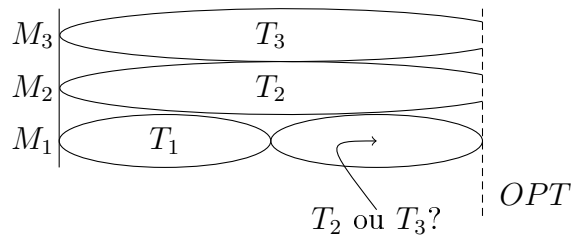


Figura 4.1: É impossível deterministicamente calcular um escalonamento ótimo se o desempenho das máquinas for imprevisível.

Lema 8. *Seja um escalonamento S gerado pelo algoritmo RR para uma instância do caso de desempenho imprevisível do problema $R; s_{jk}|T_j = L|TPCC$. Depois do instante do makespan ótimo, no máximo $\lfloor m/2 \rfloor$ tarefas estarão rodando em S .*

Demonstração. Para provar o lema teremos que analisar dois casos que dependem do número de tarefas n e o número de máquinas m .

1. ($m/2 < n < m$): Nós sabemos que até o instante ótimo todas as n tarefas podem ser executadas. Existem pelo menos n vagas até OPT onde todas as tarefas podem estar alocadas e finalizadas. Já que $n < m$, no tempo $t_0 = 0$ todas as tarefas T_1, \dots, T_n estarão alocadas e algumas outras T_1, \dots, T_x estarão duplicadas, onde $1 < x < n$. Todas as tarefas estão num anel no momento e a próxima tarefa a ser duplicada é a tarefa T_{x+1} onde a cabeça do anel está posicionada. Primeiramente note que o número de máquinas é $m = n + x$ e existem $n - x$ tarefas com somente uma réplica. Quantas tarefas precisarão terminar para que pela primeira vez todas as tarefas em execução tenham duas réplicas? Se uma tarefa que já têm duas réplicas finalizar, então você vai desperdiçar no máximo duas vagas, e duas tarefas com uma réplica serão duplicadas. Note que nesse caso, uma tarefa (a que terminou) é removida do anel e a cabeça do anel anda duas tarefas, já que duas máquinas ficam ociosas. Se uma tarefa com uma réplica terminar somente uma vaga será usada e então outra tarefa com uma réplica será duplicada. Neste caso é como se a cabeça do anel tivesse andado duas posições, já que uma tarefa terminou e é removida do anel e essa tarefa não pode estar atrás da cabeça do anel (somente tarefas com duas réplicas estão antes da cabeça). Independente da situação, para esse caso, para cada tarefa que termina nós movemos duas tarefas para frente no anel. Já que temos $n - x$ tarefas com somente uma réplica no instante t_0 , então serão necessários finalizar $\lceil \frac{m-x}{2} \rceil$ tarefas para que todas as tarefas tenham pelo menos duas réplicas pela primeira vez. O número de vagas usadas é no máximo

$$2 \lceil \frac{n-x}{2} \rceil = n-x < n$$

se $(n - x)$ é par, e no máximo

$$2\lceil \frac{n-x}{2} \rceil = 2\left(\frac{n-x+1}{2}\right) = n-x+1 \leq n$$

se $(n - x)$ é ímpar.

De qualquer forma nós usamos no máximo o número de vagas que existem até o instante ótimo. Note que o número de tarefas remanescentes é no máximo

$$n - \lceil \frac{n-x}{2} \rceil \leq n - \frac{(n-x)}{2} = \frac{n+x}{2} = \frac{m}{2}$$

2. ($n \geq m$): Seja t_1 o instante quando o algoritmo RR escalonou a n -ésima tarefa. No instante t_1 existem $m' \leq m$ tarefas diferentes em execução no escalonamento S gerado pelo RR. De forma geral $m' = m$, mas pode acontecer de mais de uma tarefa finalizar exatamente no mesmo instante t_1 quando a última tarefa é alocada. Sem perda de generalidade e para simplificar, nós assumimos que $m' = m$.

Seja t_2 o instante em que a primeira tarefa é replicada. Quando isto acontece, existem $m - 1$ tarefas diferentes sendo executadas. Seja X o conjunto com estas $m - 1$ tarefas que podem ser réplicas. Pelo Lema 7 sabemos que até antes do instante t_2 o escalonamento S é exatamente igual ao $EOPT(I)$ já que nenhuma tarefa é abortada e sabemos que existem pelo menos $m - 1$ vagas que acabam antes do instante OPT. Seja SL o conjunto das $m - 1$ vagas. Nós alegamos que cada tarefa em X possa ser atribuída a no máximo duas das vagas em SL , ou quando pela primeira vez uma tarefa ocupar três vagas, existem menos que $m/2$ tarefas em execução. Para provar isto, note que para uma tarefa usar três vagas é necessário que tenha pelo menos três réplicas. Seja T_j a primeira tarefa que conseguiu duas réplicas no instante t_2 . No instante t_2 existem $m - 1$ tarefas rodando e somente T_j tem duas réplicas. Para uma tarefa ter três réplicas será necessário passar por todas as tarefas do anel e voltar para T_j . Usando o mesmo argumento do caso $m/2 < n < m$, para que uma tarefa tenha três réplicas é necessário que $\lceil \frac{m-1}{2} \rceil$ tarefas sejam finalizadas

Desta forma se uma tarefa usa três vagas de SL , então menos que $m/2$ estão rodando após o *makespan* ótimo. Por outro lado, se qualquer tarefa usa no máximo duas vagas, existirão pelo menos $\lceil \frac{m-1}{2} \rceil$ tarefas diferentes nas vagas, de tal modo que após o instante OPT no máximo

$$m - 1 - \lceil \frac{m-1}{2} \rceil \leq \frac{m-1}{2}$$

tarefas estarão em execução.

A partir de (1) e (2) existem no máximo $m/2$ tarefas em execução após o instante OPT , e já que o número de tarefas em execução é necessariamente um inteiro, nós podemos arredondar o valor para $\lfloor m/2 \rfloor$ tarefas. \square

Considere o escalonamento após o instante OPT . Do resultado anterior existem no máximo $\lfloor m/2 \rfloor$ tarefas em execução.

Pela propriedade 3 não é difícil perceber que durante um instante t qualquer após o instante OPT , se existir um número X de tarefas distintas em execução, algumas tarefas terão $\lfloor \frac{m}{x} \rfloor$ réplicas, enquanto outras terão $\lceil m/|X| \rceil$ réplicas.

Com esses resultados nós provaremos o teorema 6. O resultado obtido em 6.2 é assintoticamente igual ao apresentado na seção 2.1, mas iremos mostrar que este resultado é justo, com instâncias onde depois do instante de tempo OPT haverá $\frac{m}{2}$ tarefas em execução.

Teorema 6. *Seja I uma instância de um caso imprevisível do problema $R; s_{it}|T_j = L|TPCC$.*

- *O TPCC do escalonamento gerado pelo algoritmo RR é no máximo $OPT + L \sum_{i=1}^{m/2} \lceil m/i \rceil$.*
- *O RR é um algoritmo $(1 + \frac{3m}{2n} + \frac{m \ln(m/2)}{n})$ -aproximado para o caso imprevisível do problema $R; s_{it}|T_j = L|TPCC$.*

Demonstração. Seja S o escalonamento gerado pelo algoritmo para a instância I e seja a i -ésima última tarefa como a tarefa que concluiu seguindo a ordem de trás para a frente. A partir da propriedade 3, no momento logo antes da i -ésima última tarefa concluir, existem no máximo $\lceil m/i \rceil$ réplicas desta tarefa, visto que existem i tarefas em execução.

Para provar (1) note que até o instante OPT o escalonamento ótimo $OPT(I)$ e o escalonamento S tem o mesmo TPCC com o valor OPT . Após o instante OPT , do Lema 8 existem no máximo $\lfloor m/2 \rfloor$ tarefas em execução em S . A partir da propriedade 3 estas tarefas irão incrementar o TPCC com no máximo $L \sum_{i=1}^{m/2} \lceil \frac{m}{i} \rceil$ instruções.

Para provar (2) note que o TPCC ótimo é no mínimo nL instruções, pois cada tarefa

tem tamanho L e todas devem ser executadas. O fator de aproximação do algoritmo é:

$$r = \frac{OPT + L \sum_{i=1}^{m/2} \lceil \frac{m}{i} \rceil}{OPT} \quad (4.1)$$

$$= 1 + \frac{L \sum_{i=1}^{m/2} \lceil \frac{m}{i} \rceil}{OPT} \quad (4.2)$$

$$\leq 1 + \frac{L \sum_{i=1}^{m/2} (\frac{m}{i} + 1)}{nL} \quad (4.3)$$

$$\leq 1 + \frac{m/2 + m \sum_{i=1}^{m/2} \frac{1}{i}}{n} \quad (4.4)$$

$$\leq 1 + \frac{m/2 + m(\ln(m/2) + 1)}{n} \quad (4.5)$$

$$\leq 1 + \frac{3m/2 + m \ln(m/2)}{n} \quad (4.6)$$

$$\leq 1 + \frac{3m/2}{n} + \frac{m \ln(m/2)}{n} \quad (4.7)$$

□

Embora o fator de aproximação provado no Teorema 6 seja assintoticamente igual ao fator apresentado no Teorema 1, nós podemos provar que este resultado é justo para o algoritmo RR mostrando classes de instâncias onde $\lfloor \frac{m}{2} \rfloor$ tarefas estão executando após o tempo OPT . Considere o exemplo dado nas Figuras 4.2 e 4.3. Existem 9 máquinas e quando a última tarefa é alocada existem exatamente 9 vagas disponíveis (Figura 4.2). Para os intervalos de tempo não marcados como vagas podemos assumir que o processador tem poder de processamento igual a 0 desde o instante que a última tarefa é escalonada até o instante OPT . A atribuição das últimas 9 tarefas, feita de acordo com o algoritmo RR, ocorre como apresentado na Figura 4.3.

Note que após o instante OPT , existem $\lfloor \frac{m}{2} \rfloor = 4$ tarefas a serem finalizadas (T_4, T_6, T_8, T_9). A partir de agora, assuma que uma tarefa T_j que tenha $\lceil \frac{m}{n_j} \rceil$ réplicas termine, e todas as suas réplicas também terminem ao mesmo tempo, onde n_j é o número de tarefas executando logo antes de T_j finalizar. No exemplo, a primeira tarefa a terminar é a T_4 . Seguindo esse raciocínio nós iremos ter mais

$$L \sum_{i=1}^{\lfloor \frac{m}{2} \rfloor} \lceil \frac{m}{i} \rceil$$

instruções executadas depois do instante OPT .

Nós então podemos concluir o seguinte resultado:

Teorema 7. *A aproximação apresentada no Teorema 6 é justa.*

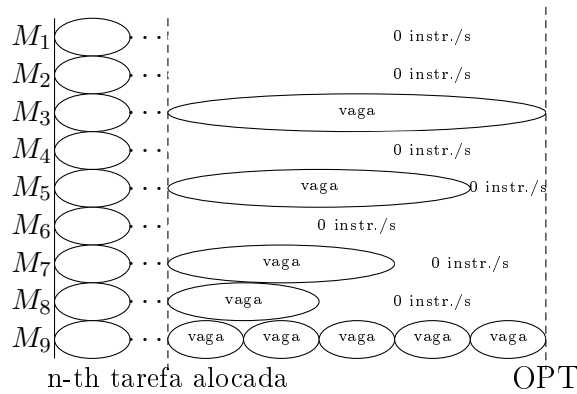


Figura 4.2: Quando a última tarefa é alocada existem m vagas disponíveis.

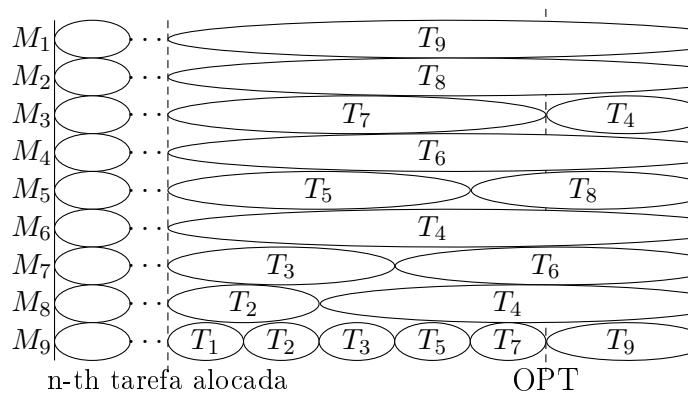


Figura 4.3: O escalonamento gerado pelo algoritmo RR, e as 9 últimas tarefas escalonadas.

Demonstração. O exemplo nas Figuras 4.2 e 4.3 pode ser estendido para qualquer número ímpar m de máquinas. Sempre existirá $\lfloor m/2 \rfloor$ tarefas executando após o instante OPT e pode-se estabelecer velocidades de processamento tal que sempre que a i -ésima última tarefa que tem $\lceil \frac{m}{i} \rceil$ réplicas termine, todas suas réplicas terminam juntas consumindo $L \lceil \frac{m}{i} \rceil$ instruções. Então o TPCC do escalonamento gerado pelo RR é

$$OPT + L \sum_{i=1}^{m/2} \lceil \frac{m}{i} \rceil.$$

□

4.2 Tarefas com tamanhos diferentes

Nesta seção iremos mostrar a aproximação justa para os problemas $R; s_{it}|T_j|C_{max}$ e $R; s_{it}|T_j|TPCC$ originalmente apresentados por Fujimoto e Hagihara [13, 14].

Cada instância do problema é composto por um conjunto de n tarefas de tamanho L_j a serem escalonados em m máquinas sem relação entre si. A precedência entre tarefas é representada por um grafo de acíclico chamado de grafo de tarefa. O objetivo é gerar um escalonamento S que minimize o *Total Processor Cycle Consumption* (TPCC) ou o C_{max} .

Na seção 4.1 nós mostramos que é possível chegar a um TPCC ou *makespan* ótimo quando o poder de processamento é previsível. Nós iremos mostrar que não é possível chegar ao resultado ótimo quando as tarefas têm tamanhos diferentes, mesmo quando o poder de processamento é previsível.

Teorema 8. *Considere uma instância de um problema $R; s_{it}|T_j|C_{max}$, e seja l a lista de n tarefas a serem escalonadas, cada tarefa T_j com tamanho L_j . Suponha que a velocidade das máquinas é previsível. Então não se pode ter que $P = NP$ é impossível chegar um algoritmo aproximado para o problema.*

Demonstração. Seja $I = (S, D)$ uma instância do Problema de Particionamento (PP) (problema NP-difícil bem conhecido), onde $S = s_1, \dots, s_n$ é um conjunto de n números naturais e $D = (\sum_{j=1}^n s_j)/2$ é um número natural. No problema PP nós temos que determinar se um conjunto S pode ser particionado em dois conjuntos, tal que a soma dos elementos do conjunto seja D .

Dado a instância I de um problema PP nós construímos uma instância I' do problema de escalonamento $R; s_{ij}|T_j|C_{max}$ com dois processadores, cada um dos processadores com poder computacional de 1 instrução por segundo durante os D primeiros segundos, e 0 instruções por segundo no restante do tempo. Nós também temos n tarefas, cada tarefa T_j com tamanho $L_j = s_j$, $j = 1, \dots, n$.

Se a instância I do problema PP admite uma partição, então a solução ótima do problema $R; s_{ij}|T_j|C_{max}$ tem *makespan* D . Por sua vez se I não admite uma partição, então a solução ótima do problema $R; s_{ij}|T_j|C_{max}$ tem *makespan* α . Então, qualquer algoritmo α -aproximado pode ser usado para decidir se o problema PP admite uma partição, já que um algoritmo α -aproximado retorna uma solução com valor finito (no máximo αD) se e somente se a instância I' admite uma partição. \square

Já que não existe uma aproximação para a função C_{max} , iremos analisar somente a função TPCC.

Teorema 9. *O algoritmo RR é um algoritmo m -aproximado para o $R; s_{it}|T_j|TPCC$ mesmo quando o poder de processamento das máquinas é imprevisível.*

Demonstração. Já que para a solução ótima todas as tarefas devem ser executadas, logo temos o seguinte limitante inferior:

$$\sum_{j=1}^n L_j \leq OPT.$$

Quando o algoritmo alocar a última tarefa inicia-se a fase de replicação. A quantidade extra de réplicas de cada tarefa é de no máximo $m - 1$ e temos no máximo $m - 1$. Seja $(T_{j_1}, \dots, T_{j_{m-1}})$ as tarefas replicadas. O número de instruções executadas pelas réplicas é limitado por

$$(m - 1) \sum_{i=1}^{m-1} L_{j_i} \leq (m - 1)OPT.$$

O fator de aproximação r pode ser limitado por

$$r \leq \frac{\sum_{j=1}^n L_j + (m - 1)OPT}{OPT} \leq \frac{mOPT}{OPT} \leq m$$

□

Teorema 10. *O fator de aproximação m do algoritmo RR é justo.*

Demonstração. Para provar este resultado nós vamos fornecer uma classe de instância onde o fator está entre o número de instruções executados pelo algoritmo e o ótimo é $\Omega(m)$.

Suponha uma lista de $n = m$ tarefas (T_1, \dots, T_m) onde as $(m - 1)$ primeiras tarefas têm tamanho L e a última tarefa tem tamanho $(m - 1)L$. Nós temos m máquinas e o escalonamento ótimo termina no instante t_1 . A partir do instante 0 até t_1 uma das máquinas executa $(m - 1)L$ instruções e as outras máquinas executam L instruções. É obvio que no escalonamento ótimo, a tarefa T_m é escalonada na máquina mais rápida e as outras tarefas nas outras máquinas. Já que o poder de processamento não é previsível, pode acontecer da tarefa T_m não ser escalonada na máquina mais rápida.

Suponha um exemplo com seis tarefas (T_1, \dots, T_6) , onde elas são atribuídas nesta ordem. A tarefa T_6 tem tamanho $5L$ instruções e as outras tem tamanho L . A máquina mais rápida tem o poder de processamento de $5L$ instruções até o instante t_1 . A Figura 4.4 mostra o escalonamento gerado pelo algoritmo da instância que acabamos de definir. Uma solução ótima seria a atribuição da tarefa T_6 à máquina mais rápida e as outras tarefas nas outras máquinas.

No caso geral, considere m um número par; assuma que um algoritmo escalone as tarefas (T_1, \dots, T_m) nesta ordem; considere T_m a maior tarefa e, sem perda de generalidade, a máquina M_1 é a máquina mais rápida. Para m par, pelo menos metade das tarefas

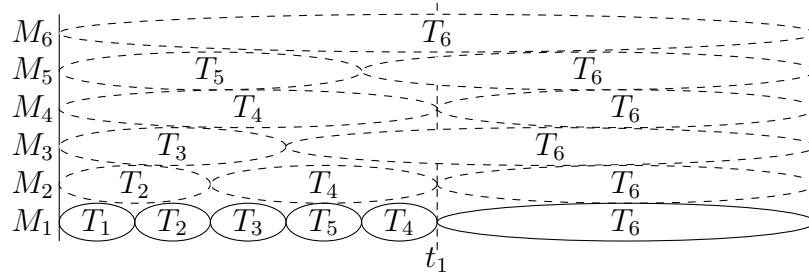


Figura 4.4: Um exemplo de instância do pior caso.

serão executadas na máquina M_1 antes que T_m seja atribuída a ela. Para enxergar isto suponha que após T_1 e T_2 terminarem sua execução em M_1 , é possível alocar um par de tarefas a serem escalonadas, uma na máquina M_1 e outra na máquina que foi liberada. Seguindo esse raciocínio, no instante t_1 , a tarefa T_m executou menos que a metade do seu tamanho na máquina M_1 . Uma vez que cada máquina pode executar L instruções até o instante t_1 , então no máximo L instruções de T_m foram executados nas outras máquinas. Já que as outras máquinas podem executar L instruções até o instante t_1 , então no máximo L instruções de T_m foram executados nas outras máquinas. Após o instante t_1 , todos os processadores estão processando uma réplica de T_m e todas terminam juntas. Desta forma, o TPCC do escalonamento gerado será $2(m-1)L$ até o instante t_1 , e após t_1 serão executados $\frac{(m-1)L}{2}$ (o que falta ser processado da tarefa T_m na máquina M_1), mais $(m-1)((m-1)L-L)$ (o que falta de T_m a ser processado nas outras máquinas) instruções.

O fator de aproximação da solução e a solução ótima satisfazem

$$r \leq \frac{2(m-1)L + \frac{(m-1)L}{2} + (m-1)^2L - (m-1)L}{2(m-1)L}$$

onde

$$\lim_{m \rightarrow \infty} \frac{2(m-1)L + \frac{(m-1)L}{2} + (m-1)^2L(m-1)L}{2(m-1)L} = \frac{m}{2}$$

□

4.3 Replicação de tarefa

Nessa seção nós vamos propor uma interface que poderá ser usada para adicionar replicação em qualquer algoritmo de escalonamento. Depois nós iremos comparar o desempenho dos algoritmos que foram estendidos usando replicação.

Seja $\mathcal{A}(T, M, C)$ um algoritmo de escalonamento que recebe por parâmetro um conjunto de tarefas T a ser escalonado num conjunto M de máquinas, com uma configuração

inicial C (esta configuração mostra as tarefas que estão executando neste exato momento e em quais máquinas elas estão executando). Sempre que existir máquina ociosa o algoritmo $\mathcal{A}(T, M, C)$ é chamado, atualizando T e C . Para gerar um escalonamento completo é só fazer chamadas sucessivas ao algoritmo (\mathcal{A}) até que todas as tarefas tenham sido escalonadas. Para adicionar suporte a replicação de tarefas é só criar uma interface que chama (\mathcal{A}) e monitora todas as tarefas que são alocadas. Depois que todas as tarefas estiverem alocadas a interface monitora todas as tarefas que estão executando numa lista R . Sempre que uma máquina ficar livre a interface executa $\mathcal{A}(R, M, C)$. Quando uma tarefa replicada termina, a interface mata todas as réplicas da tarefa.

Nós vamos mostrar a garantia de desempenho para qualquer algoritmo de escalonamento que permita replicação usando este método.

Teorema 11. *Seja \mathcal{A} um algoritmo de escalonamento para o problema $R; s_{it}|T_j = L|TPCC$ ou $R; s_{it}|T_j|TPCC$. A versão do \mathcal{A} com replicação tem fator de aproximação de no máximo $\min\{(1 + \frac{(m-1)^2}{n}), m\}$ para o problema $R; s_{it}|T_j = L|TPCC$ e no máximo m para o problema $R; s_{it}|T_j|TPCC$.*

Demonstração. Para o problema $R; s_{it}|T_j = L|TPCC$ note que quando o algoritmo escalona a última tarefa ele executou nL instruções. Na fase de replicação, cada uma das $(m - 1)$ das tarefas que ficaram executando terão no máximo m replicações. Logo o fator de aproximação é limitado por

$$r \leq \frac{nL + (m - 1)^2L}{nL} = 1 + \frac{(m - 1)^2}{n}.$$

O limitante m para a aproximação de ambos os problemas é similar a prova do teorema 9. □

Capítulo 5

Análise de desempenho

Neste capítulo iremos mostrar os resultados experimentais obtidos a partir de simulações usando os algoritmos apresentados no capítulo 3 aplicando a interface de replicação apresentado no capítulo 4. Com os resultados da simulação podemos verificar qual o ganho que cada algoritmo obteve ao aplicar a interface e podemos comparar os algoritmos em diferentes ambientes.

Na seção 5.1 vamos apresentar o modelo que usamos para desenvolver o simulador e na seção 5.2 será apresentado os resultados.

5.1 Modelo do simulador

Nesta seção iremos apresentar o modelo usado para desenvolver o nosso simulador, baseado no modelo proposto por Fujimoto e Hagihara [15].

Nossa grade computacional é formada por um conjunto de m máquinas onde cada máquina tem um processador. O desempenho de cada máquina M_i , definido por $s_{i,t}$, varia a cada instante de tempo t , mas não varia no intervalo $[t, t + 1)$. O simulador provê o tamanho de todas as tarefas que estão na grade e o desempenho das máquinas no instante que é solicitado. Para simplificar o modelo, essas funções foram implementadas a custo zero.

Na criação dos ambientes foram usadas as distribuições Uniforme e Zipf. A distribuição Uniforme ($U[a, b]$) gera um conjunto de números entre a e b , para $a < b$, de forma uniforme. Já a distribuição Zipf ($Zipf[a, b]$) gera um conjunto de números usando a lei Zipf onde a é o elemento com maior frequência, enquanto b é o elemento com menor frequência. Por exemplo para a instância $Zipf[5, 30]$ o elemento 5 terá uma frequência de 25,94% enquanto o elemento 30 terá uma frequência de 0,997%, isto é, se criarmos 100 elementos usando esta distribuição, aproximadamente 25 elementos serão o número 5 e aproximadamente 1 será o número 30.

Cada simulação é composta de quatro variáveis: quantidade de máquina (qtd_p), variabilidade das máquinas (MV), quantidade de tarefas (qtd_t) e variabilidade de tarefas (TV). Vamos mostrar abaixo como cada variável é criada no nosso simulador.

A quantidade de máquinas (qtd_p) foi fixada em somente três valores, que são $qtd_p \in \{16, 64, 256\}$.

A variabilidade das máquinas (MV) dita a heterogeneidade das máquinas no ambiente. No ambiente a máquina mais rápida pode ser no máximo MV vezes mais rápida do que a máquina mais lenta do ambiente. Essa variabilidade foi fixada em três valores, que são $MV \in \{1, 4, 8\}$, assim temos um ambiente homogêneo e dois ambientes heterogêneos. Toda máquina M_i tem um multiplicador z_i que é sorteado usando a distribuição $U[1, MV]$. O desempenho da máquina é calculada como segue: $s_{i,t} = U[0, 5] * z_i$. É importante notar que a velocidade da máquina pode ser zero, assim podemos simular a situação em que a máquina está sobrecarregada.

A quantidade de tarefas (qtd_t) é o produto da quantidade de máquinas e uma constante k , ou seja, $qtd_p * k$. Essa constante k é um número inteiro que foi fixada em seis valores diferentes, que são $k \in \{1, 2, 3, 4, 5, 6\}$. Assim podemos estudar o comportamento dos algoritmos com volume de tarefas diferentes.

A variabilidade das tarefas (TV) dita como será criado o conjunto de tarefas, criando diferentes ambientes de teste. Nas nossas simulações definimos cinco tipos de variabilidades que serão explicados abaixo:

- $U[1, 30]$ — Todos os tamanhos são sorteados usando uma distribuição uniforme.
- $Zipf[5, 30]$ — Usa a distribuição Zipf onde a maioria das tarefas são pequenas e pouquíssimas tarefas são grandes.
- $Zipf[30, 5]$ — O contrário da anterior, a maioria das tarefas são grandes e pouquíssimas são pequenas.
- $20\%U[1, 15]$ — 20% das tarefas são criadas usando $U[1, 15]$ e as outras 80% são criadas usando $U[15, 30]$.
- $80\%U[1, 15]$ — 80% das tarefas são criadas usando $U[1, 15]$ e as outras 20% são criadas usando $U[15, 30]$.

A partir da combinação das quatro variáveis que descrevem o ambiente foi possível avaliar o impacto que cada uma tem nos algoritmos. Para cada combinação das variáveis foram efetuados 20 simulações, totalizando 43200 simulações. Na próxima seção iremos apresentar os resultados relevantes.

5.2 Resultados dos experimentos

Nesta seção vamos apresentar os resultados de uma comparação prática entre quatro algoritmos de escalonamentos para o problema $R; s_{jk}|T_j|C_{max}$ onde iremos analisar várias métricas para as versões com e sem replicação.

O TPCC, uma das métricas analisadas, calcula a soma da velocidade de cada máquina até o instante do *makespan*, mas algumas máquinas não executam tarefas até esse instante. De qualquer forma o TPCC é uma boa métrica, como visto na seção 2.1, para fazer análises teóricas do desempenho de algoritmos.

Mas do ponto de vista prático é mais interessante analisar o *Real Total Processor Cycle Consumption* (RTPCC), e que nós definimos como a soma das instruções das tarefas (T_1, \dots, T_n) executadas nas máquinas, ou seja, só é calculado o que realmente foi utilizado pela execução das tarefas. Para algoritmos que não usam replicação de tarefas o RTPCC é igual a soma do tamanho de todas as tarefas, já que os algoritmos não irão gastar mais processamento do que o necessário. Mas algoritmos que fazem replicação gastam mais processamento do que é realmente necessário, e no caso de algoritmos com replicação sem limites o RTPCC é igual ao TPCC, já que sempre que uma máquina está livre, alguma tarefa é alocada/replicada para esta.

Outra métrica que iremos usar é o RTPCC perdido. Essa métrica é usada nos algoritmos com replicação para saber o quanto de processamento foi gasto em tarefas que foram abortadas em relação a quantidade total necessária para executar todas as tarefas. Esse é um ponto importante para analisarmos a relação custo-benefício dos algoritmos.

Mas para permitir a análise do custo benefício, nós também vamos analisar o quanto um algoritmo melhorou ao usar replicação. Para isso nós executamos a versão com e sem replicação de um algoritmo numa mesma instância e comparamos o *makespan* dos dois resultados, obtendo em porcentagem quanto um algoritmo é melhor do que o outro.

No trabalho [8] é aplicado replicação no algoritmo *WorkQueue* sendo a versão com replicação denotada por WQRxx. O sufixo xx representa a quantidade de réplicas que cada tarefa pode ter, onde o xx significa que não existe um limite para réplicas. Mas para os casos 2x, 3x e 4x significa que cada tarefa pode ter 2, 3 e 4 réplicas respectivamente. A definição do algoritmo WQRxx é igual ao algoritmo RR, onde a única diferença é que o algoritmo RR especifica a estrutura de dados que será usada para controlar a replicação das tarefas. Nós também implementamos os algoritmos limitando a quantidade de réplicas para estudarmos a sua influência no *makespan*. Nas figuras 5.1, 5.2, 5.3, 5.4, 5.5 e 5.6 todos os algoritmos com replicação tem o sufixo -Rxx indicando quantas réplicas são permitidas. Nessas figuras nós comparamos todos os algoritmos com e sem replicação. Para cada variabilidade de máquina e tarefa nós calculamos a média do *makespan* e RTPCC de todos os testes envolvendo esta configuração.

Em trabalhos anteriores [8, 15] os algoritmos de escalonamento *WorkQueue* [8], *Sufferage* [3], *Max-min* [28], *Min-min* [27] e DFPTLF [3] foram comparados com o RR (ou WQRxx). De forma geral o *makespan* gerado pelos escalonadores que tem informações sobre a velocidade do processador são ligeiramente melhores dos que não tem, mas o trabalho também mostrou que a replicação no algoritmo RR gera bons resultados sem usar muito recurso adicional.

Nesta dissertação nós implementamos o mesmo conjunto de algoritmos, com exceção do *Max-min*. O *Max-min*, como já dito na seção 3.3, é um algoritmo estático e fizemos uma adaptação para ambientes dinâmicos, mas nessa adaptação o comportamento do *Max-min* ficou igual ao do algoritmo DFPTLF. O *Min-min* implementamos a versão dinâmica. Para todos os algoritmos foi implementada a versão com replicação usando a interface (A) definida na seção 4.3. As versões com replicação dos algoritmos são identificadas pela adição do sufixo -R nos seus nomes.

Nas figuras 5.4, 5.5 e 5.6 podemos ver que sempre que aumentamos a quantidade de replicação, o RTPCC aumenta. Esse é um comportamento natural, já que quanto mais réplicas existem menos processadores ficam ociosos. Mas a partir desses gráficos podemos ver que quando não limitamos a quantidade de réplicas o consumo de processamento aumenta muito em comparação a versão com limitação de réplicas. E como podemos ver nas figuras 5.1, 5.2 e 5.3 na maioria dos casos o ganho no *makespan* é muito pequeno em comparação a quantidade de processamento que é consumida. É interessante verificar essa relação para decidir se realmente vale a pena aumentar o número de réplicas, mas isso vai depender dos requisitos do sistema em questão.

Analisando o *makespan* da figuras 5.1, 5.2 e 5.3 vimos que aplicar a replicação trouxe uma melhoria significativa comparado com a versão sem replicação. Mas também encontramos resultados diferentes do esperado, pois achávamos que quanto mais replicação uma tarefa pudesse ter, melhor seria o seu *makespan*. Nos resultados percebemos que o algoritmo *Min-min* tem o seu melhor *makespan* na versão *Min-min-R2x*. Algo parecido acontece com o algoritmo *Sufferage*, mas que só acontece quando a variabilidade das máquinas é $U[1, 1]$ e $U[1, 4]$. Quando a variabilidade é $U[1, 8]$ o *makespan* continua caindo. Isso deve acontecer porque quando a variabilidade aumenta, o ambiente passa a ter computadores muito superiores que fazem toda a diferença no processo de replicação, mas isso não acontece com o *Min-min* porque mesmo tendo processadores muito potentes, o algoritmo tende a colocar as menores tarefas nesses processadores colocando as tarefas pesadas em processadores mais lentos. Curiosamente o algoritmo DFPLTF piora para uma variabilidade de máquina $U[1, 4]$ e de tarefas *Zipf*[30, 5] quando não limitamos a quantidade de réplicas (DFPLTF-Rxx), mas ele piora muito pouco em relação ao DFPLTF-R3x.

A figura 5.7 mostra o desperdício de ciclos de processamento dos algoritmos em função

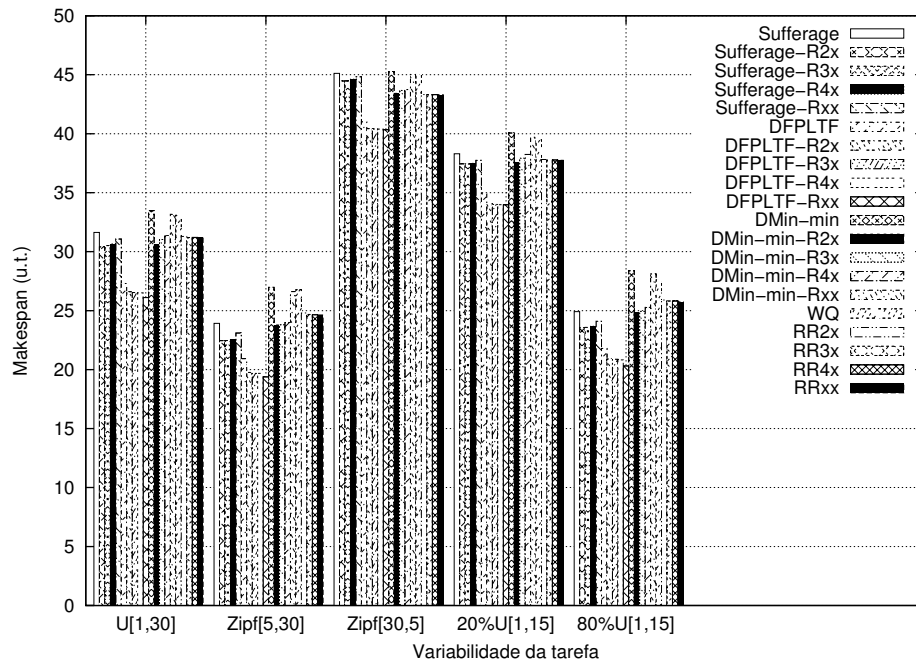


Figura 5.1: *Makespan* (Variabilidade das máquinas $\in U[1, 1]$)

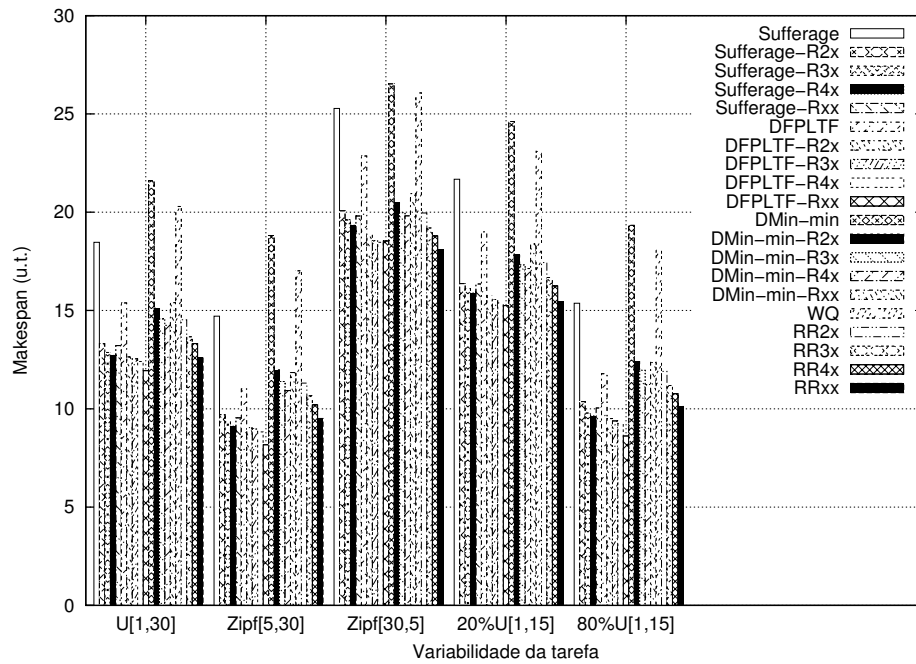


Figura 5.2: *Makespan* (Variabilidade das máquinas $\in U[1, 4]$)

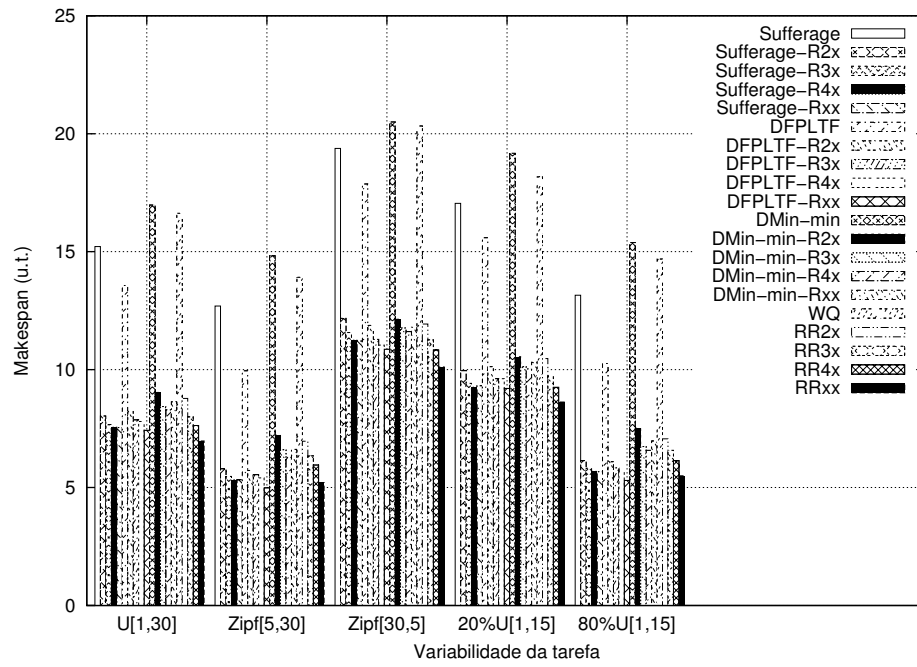


Figura 5.3: *Makespan* (Variabilidade das máquinas $\in U[1, 8]$)

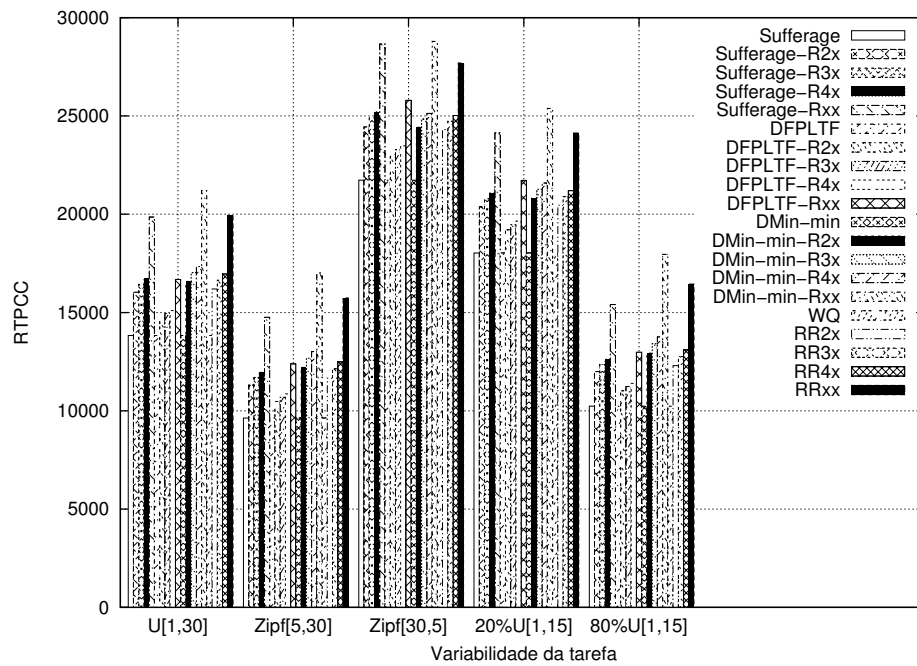


Figura 5.4: RTPCC (Variabilidade das máquinas $\in U[1, 1]$)

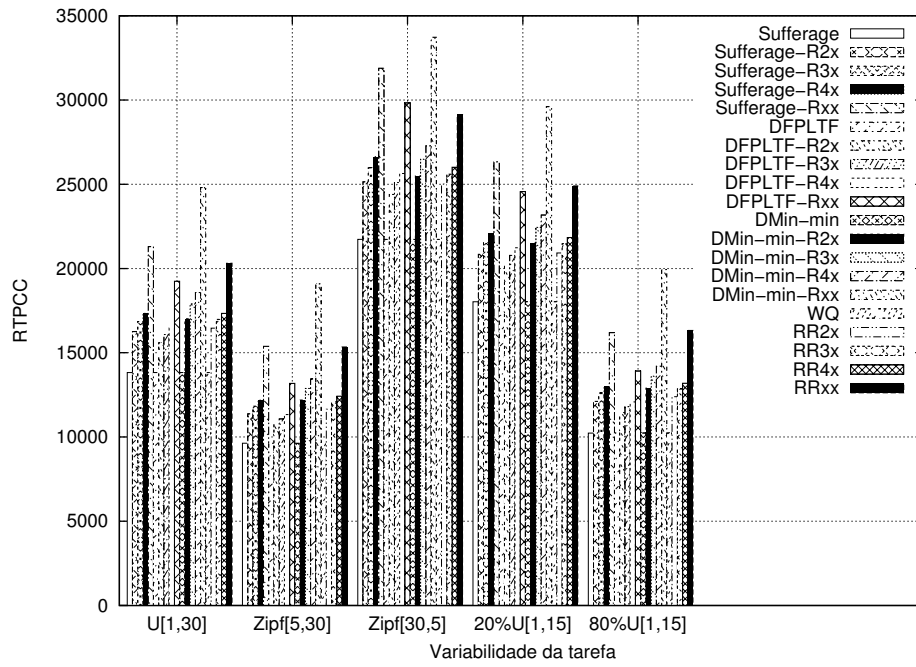


Figura 5.5: RTPCC (Variabilidade das máquinas $\in U[1,4]$)

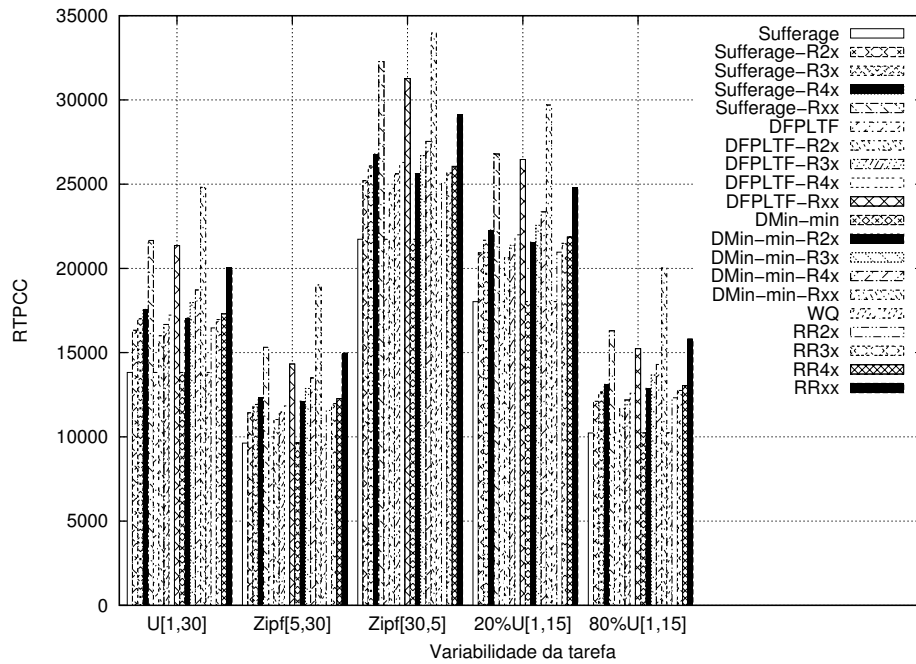


Figura 5.6: RTPCC (Variabilidade das máquinas $\in U[1,8]$)

da replicação. Cada cor representa um algoritmo. Existe um ponto para cada combinação das variáveis quantidade de máquinas, variabilidade das máquinas, quantidade de tarefas, variabilidade das tarefas e quantidade de replicação, totalizando 1350 pontos para cada algoritmo. Para facilitar a leitura do gráfico foi aplicada uma função de repulsão entre os pontos, senão todos os pontos estariam numa linha. Logo os algoritmos sem replicação tem desperdício zero, então todos os pontos estariam na coordenada $(0, 0)$. Já o gráfico 5.8 mostra o ganho do *makespan* em porcentagem dos algoritmos que aplicaram replicação comparada com a versão sem replicação. Nesse gráfico não existe os algoritmos sem replicação, totalizando 1080 pontos. Em ambos os gráficos o valor -1 no eixo Y significa que não existe limite de replicações.

Podemos ver nesses gráficos que o algoritmo DFPLTF desperdiçou pouco processamento. Os Sufferage também tem um conjunto de pontos bem concentrados e com um desperdício razoavelmente baixo, em comparação aos outros. Também podemos ver que o Min-min e o RR (WQ) são os recordistas de desperdício, mas eles também foram os recordistas em ganho.

Nas tabelas 5.1 e 5.2 limitamos a análise para os algoritmos que não restringem a quantidade de réplicas.

Na tabela 5.1 apresentamos o resultado dos experimentos dos algoritmos com replicação, executados para cada tipo de quantidade de máquinas, variabilidade de máquinas e variabilidade de tarefas. Para uma dada quantidade de máquinas, variabilidade da máquina e tarefa, o resultado é a média de todos os testes para todas as quantidades de tarefas. Nesta tabela podemos ver o *makespan* (C_{max}), TPCC e fator de aproximação (App). O fator App dessa tabela, é um fator de aproximação prático, que é o quociente entre o TPCC e a soma do tamanho de todas as tarefas. É importante notar que em todos os casos o fator de aproximação prático é muito inferior ao fator de aproximação teórico.

Na tabela 5.2 mostramos o algoritmo com o melhor *makespan* para cada tipo de ambiente de execução. Neste tabela, cada resultado é a média de todos os resultados para cada ambiente. Nela temos as métricas *makespan* do algoritmo com replicação (C_{max}), *makespan* do algoritmo sem replicação (C_{max} s/replic.), quanto o algoritmo com replicação é melhor do que o sem replicação (Melhoria em %) e quanto de processamento foi desperdiçado por causa da replicação das tarefas (Desp. em %).

Podemos ver na tabela 5.2 que o uso da replicação trouxe um ganho do *makespan* em comparação a versão sem replicação. A média de ganho que cada algoritmo teve em ordem decrescente é 27.64% para o algoritmo RR, 24.42% para o *Min-min-R*, 22.39% pro *Sufferage-R* e 19.83% para o DFPLTF-R.

Também podemos analisar qual é a influência da variação das tarefas (TV) e das máquinas (MV) no ganho do *makespan*. Considerando a média de todos os algoritmos



Figura 5.7: Desperdício em % X Quantidade de réplicas)

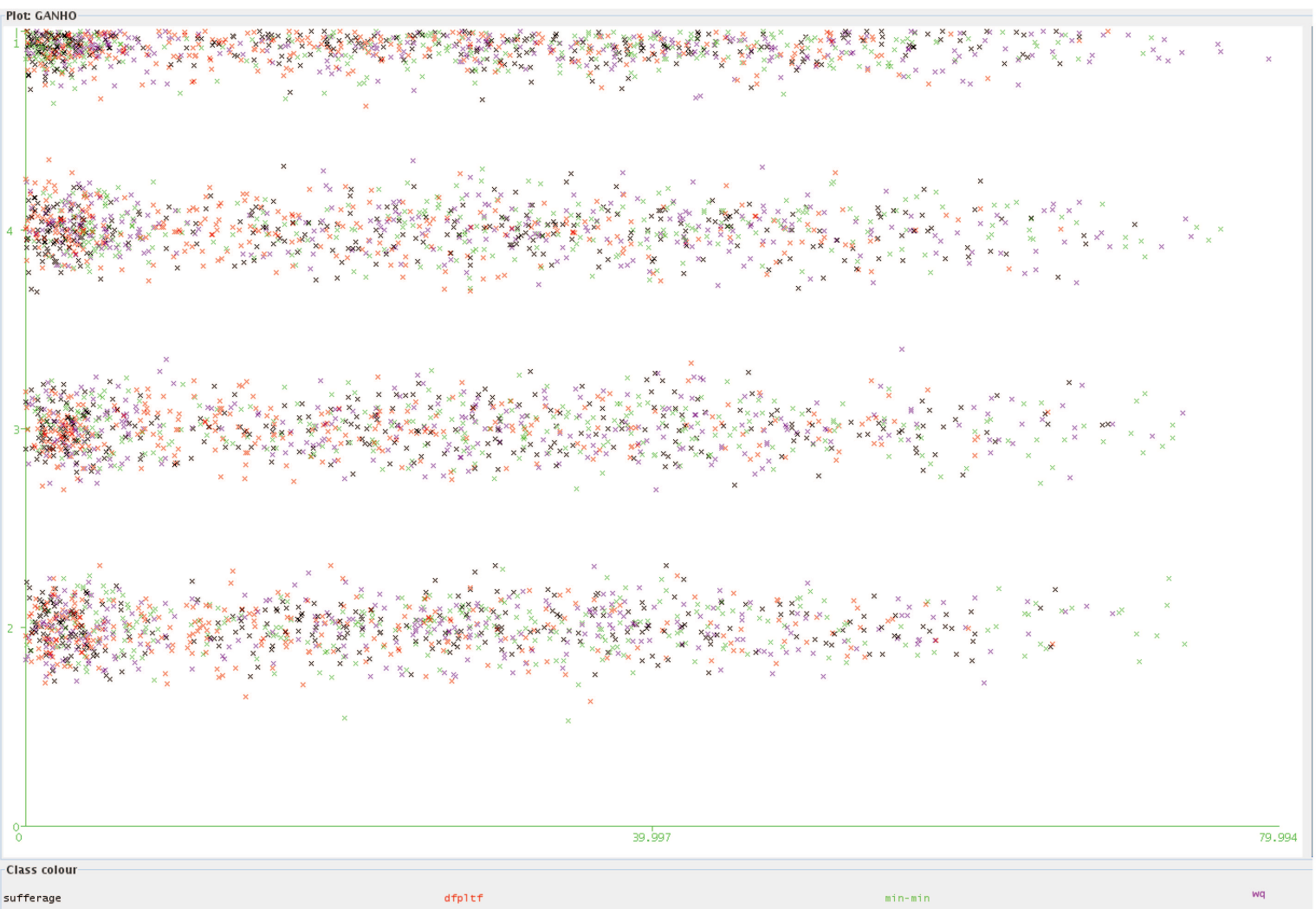


Figura 5.8: Desperdício (Ganho em % X Quantidade de réplicas)

temos que o ganho médio para cada ambiente foi: 43.05% para $MV = U[1, 8]$; 25.06% para $MV = U[1, 4]$; e 2.60% para $MV = U[1, 1]$. Quando temos instâncias onde o número de pequenas tarefas é grande e poucas tarefas são grandes o ganho normalmente foi melhor. Por exemplo o ganho para as diferentes variabilidade de tarefas foram: 27.24% para $80\%U[1, 15]$; 26.53% para $Zipf[5, 30]$; 23.61% para $20\%U[1, 15]$; 21.02% para $Zipf[30, 5]$; e 19.46% para $U[1, 30]$.

Para todos os algoritmos e em todos os casos de VT e QM , a replicação aplicada num ambiente homogêneo ($VM = U[1, 1]$) mostrou que existe um desperdício que varia de 12.57% à 47.17% para um ganho que varia de 0.13% à 8.69%. Ou seja, não existe grande vantagem em usar replicação num ambiente homogêneo, pois gasta-se muito para um ganho muito pequeno.

A partir do resultado de nossas simulações, nós pudemos perceber que os melhores algoritmos foram o DFPLTF-R e o RR. Outros algoritmos que obtiveram bons resultados com a replicação foram o Sufferage-R, mas não tão bons quanto o DFPLTF-R e o RR. Por outro lado o *Min-min* obteve uma grande melhoria no *makespan* mas sempre obteve os piores resultados. Em ambientes onde o poder de processamento das máquinas são facilmente previsíveis o DFPLTF-R é a melhor escolha, mas em ambientes onde a predição é muito custosa o RR mostrou ser a melhor alternativa.

Tabela 5.1: *Makespan*(MSpan), TPCC e fator de aproximação(App) dos algoritmos replicados

Qtd. Maq.	Var. Maq.	Var. Tarefas	DFPLTF-R			Min-min-R			Sufferage-R			RR		
			C_{max}	TPCC	App	C_{max}	TPCC	App	C_{max}	TPCC	App	C_{max}	TPCC	App
16	U[1,1]	20%U[1,15]	24.46	977.09	1.25	28.47	1137.77	1.44	27.54	1101.12	1.38	28.26	1129.37	1.42
16	U[1,1]	80%U[1,15]	16.77	669.25	1.26	20.53	820.06	1.53	18.51	738.40	1.38	20.21	806.10	1.49
16	U[1,1]	U[1,30]	37.74	1508.44	1.16	40.89	1634.37	1.25	41.08	1641.61	1.26	40.47	1617.88	1.24
16	U[1,1]	Zipf[30,5]	31.64	1265.83	1.22	35.13	1404.04	1.34	34.24	1369.05	1.31	34.69	1386.04	1.33
16	U[1,1]	Zipf[5,30]	18.44	736.56	1.32	23.14	924.25	1.63	20.69	826.87	1.46	21.86	872.83	1.56
16	U[1,4]	20%U[1,15]	11.27	1081.22	1.37	12.85	1232.70	1.58	11.98	1149.54	1.45	12.04	1154.14	1.46
16	U[1,4]	80%U[1,15]	7.49	718.89	1.32	9.17	877.43	1.63	7.81	748.80	1.38	8.29	796.06	1.46
16	U[1,4]	U[1,30]	17.23	1655.30	1.33	18.66	1791.34	1.43	18.24	1750.58	1.40	17.66	1696.56	1.33
16	U[1,4]	Zipf[30,5]	14.29	1368.99	1.34	15.80	1518.01	1.50	14.95	1433.99	1.41	14.83	1421.71	1.37
16	U[1,4]	Zipf[5,30]	8.14	780.53	1.34	10.00	958.54	1.68	8.63	827.65	1.42	9.27	888.24	1.53
16	U[1,8]	20%U[1,15]	5.86	1104.94	1.37	6.51	1229.75	1.57	6.04	1143.69	1.42	6.07	1148.67	1.43
16	U[1,8]	80%U[1,15]	3.91	733.48	1.32	4.55	857.96	1.59	4.02	756.32	1.38	4.25	801.49	1.45
16	U[1,8]	U[1,30]	8.89	1684.30	1.34	9.30	1762.43	1.41	9.26	1751.70	1.39	9.00	1703.23	1.34
16	U[1,8]	Zipf[30,5]	7.34	1391.46	1.35	7.94	1506.23	1.48	7.55	1431.38	1.40	7.52	1424.42	1.38
16	U[1,8]	Zipf[5,30]	4.22	792.57	1.32	4.95	936.95	1.62	4.35	820.23	1.39	4.60	867.83	1.50
Média para 16 máquinas			14.51	1097.92	1.31	16.53	1239.45	1.51	15.66	1166.06	1.39	15.93	1180.97	1.42
64	U[1,1]	20%U[1,15]	25.32	4049.88	1.33	31.15	4986.83	1.65	29.09	4654.41	1.52	29.81	4770.92	1.55
64	U[1,1]	80%U[1,15]	18.79	3002.77	1.47	24.84	3972.23	1.92	20.56	3286.69	1.60	23.10	3691.61	1.75
64	U[1,1]	U[1,30]	38.97	6234.79	1.25	42.79	6848.63	1.37	42.99	6877.68	1.37	41.93	6709.86	1.33
64	U[1,1]	Zipf[30,5]	32.92	5269.20	1.29	37.55	6009.12	1.47	35.98	5756.79	1.41	35.95	5754.33	1.40
64	U[1,1]	Zipf[5,30]	19.48	3112.73	1.44	25.61	4096.54	1.89	21.87	3494.68	1.60	24.25	3878.44	1.76
64	U[1,4]	20%U[1,15]	11.26	4556.39	1.49	13.64	5522.28	1.88	12.20	4937.93	1.65	12.07	4885.43	1.59
64	U[1,4]	80%U[1,15]	7.70	3115.74	1.44	10.43	4220.02	2.04	8.66	3508.44	1.68	8.94	3623.63	1.70
64	U[1,4]	U[1,30]	17.28	6996.95	1.47	19.09	7729.32	1.66	18.42	7462.43	1.57	17.51	7093.01	1.45
64	U[1,4]	Zipf[30,5]	14.42	5839.07	1.47	16.62	6732.51	1.73	15.26	6179.10	1.57	14.91	6038.51	1.49
64	U[1,4]	Zipf[5,30]	8.19	3316.22	1.46	10.92	4420.99	2.04	9.12	3693.60	1.67	9.51	3852.42	1.71
64	U[1,8]	20%U[1,15]	6.94	4866.81	1.56	7.86	5513.61	1.89	7.07	4964.63	1.63	6.92	4859.41	1.57
64	U[1,8]	80%U[1,15]	4.64	3263.45	1.44	5.95	4183.24	2.04	4.86	3423.29	1.59	5.06	3561.57	1.66
64	U[1,8]	U[1,30]	10.27	7199.92	1.53	11.01	7715.96	1.67	10.66	7471.32	1.57	10.04	7046.69	1.43
64	U[1,8]	Zipf[30,5]	8.69	6085.99	1.54	9.54	6690.89	1.75	8.82	6185.57	1.56	8.56	6003.72	1.47
64	U[1,8]	Zipf[5,30]	4.97	3496.02	1.48	6.20	4357.93	2.02	5.22	3669.40	1.62	5.31	3739.69	1.65
Média para 64 máquinas			15.32	4693.73	1.44	18.21	5533.34	1.80	16.72	5037.73	1.58	16.93	5033.95	1.57
256	U[1,1]	20%U[1,15]	26.10	16679.08	1.40	33.17	21202.61	1.77	31.06	19851.30	1.66	31.20	19943.13	1.63
256	U[1,1]	80%U[1,15]	19.37	12382.10	1.55	26.62	17010.00	2.12	23.10	14761.46	1.84	24.63	15741.85	1.91
256	U[1,1]	U[1,30]	40.34	25793.54	1.31	45.03	28791.49	1.46	44.82	28658.09	1.44	43.31	27692.47	1.40
256	U[1,1]	Zipf[30,5]	33.97	21718.29	1.35	39.70	25379.26	1.58	37.75	24133.37	1.50	37.75	24134.12	1.49
256	U[1,1]	Zipf[5,30]	20.31	12979.60	1.54	28.11	17961.53	2.10	24.10	15404.26	1.81	25.74	16449.58	1.88
256	U[1,4]	20%U[1,15]	11.94	19243.36	1.60	15.40	24796.28	2.18	13.21	21291.40	1.86	12.61	20315.83	1.67
256	U[1,4]	80%U[1,15]	8.16	13186.52	1.53	11.84	19089.92	2.39	9.53	15386.87	1.93	9.51	15342.45	1.83
256	U[1,4]	U[1,30]	18.53	29823.81	1.60	20.95	33723.16	1.85	19.81	31889.30	1.73	18.10	29139.55	1.49
256	U[1,4]	Zipf[30,5]	15.25	24565.47	1.57	18.38	29599.69	1.96	16.34	26323.23	1.73	15.46	24903.03	1.55
256	U[1,4]	Zipf[5,30]	8.62	13908.11	1.55	12.37	19935.20	2.36	10.04	16192.80	1.91	10.12	16328.59	1.84
256	U[1,8]	20%U[1,15]	7.43	21351.41	1.74	8.63	24818.47	2.22	7.54	21660.69	1.87	6.97	20042.41	1.63
256	U[1,8]	80%U[1,15]	4.98	14323.95	1.58	6.62	19017.67	2.39	5.33	15316.01	1.89	5.21	14982.57	1.77
256	U[1,8]	U[1,30]	10.86	31268.28	1.70	11.79	33963.64	1.90	11.21	32268.16	1.76	10.11	29117.31	1.48
256	U[1,8]	Zipf[30,5]	9.20	26460.63	1.72	10.32	29693.08	2.02	9.31	26789.32	1.76	8.62	24803.63	1.53
256	U[1,8]	Zipf[5,30]	5.30	15234.92	1.62	6.97	20023.79	2.39	5.67	16293.72	1.90	5.50	15817.87	1.75
Média para 256 máquinas			16.02	19927.94	1.56	19.73	24333.72	2.05	17.92	21748.00	1.77	17.66	20983.63	1.66

Tabela 5.2: O melhor algoritmo para cada ambiente de execução

Ambiente <i>MV, TV</i>	Melhor Algoritmo	C_{max}	C_{max} s/replic.	Melhoria em %	Desp. em %
$U[1, 1], 20\%U[1, 15]$	DFPLTF-R	25.29	26.12	3.15	18.97
$U[1, 1], 80\%U[1, 15]$	DFPLTF-R	18.31	19.38	5.53	22.90
$U[1, 1], U[1, 30]$	DFPLTF-R	39.01	39.40	0.98	16.88
$U[1, 1], Zipf[30, 5]$	DFPLTF-R	32.85	33.38	1.60	18.87
$U[1, 1], Zipf[5, 30]$	DFPLTF-R	19.41	20.39	4.79	22.56
$U[1, 4], 20\%U[1, 15]$	DFPLTF-R	11.49	14.02	18.05	28.63
$U[1, 4], 80\%U[1, 15]$	DFPLTF-R	7.79	9.85	21.00	27.11
$U[1, 4], U[1, 30]$	DFPLTF-R	17.68	21.00	15.82	27.03
$U[1, 4], Zipf[30, 5]$	DFPLTF-R	14.65	17.45	16.02	27.32
$U[1, 4], Zipf[5, 30]$	DFPLTF-R	8.31	10.51	20.88	27.43
$U[1, 8], 20\%U[1, 15]$	RR	6.66	13.32	50.04	32.18
$U[1, 8], 80\%U[1, 15]$	DFPLTF-R	4.51	8.11	44.38	29.38
$U[1, 8], U[1, 30]$	RR	9.72	16.84	42.31	26.91
$U[1, 8], Zipf[30, 5]$	RR	8.23	15.03	45.22	28.79
$U[1, 8], Zipf[5, 30]$	DFPLTF-R	4.83	8.37	42.27	30.17

Capítulo 6

Conclusão

A partir dos estudos feitos em relação ao problema de escalonamento de tarefas em grades computacionais, notamos que existem muitos trabalhos publicados, mas poucos analisando teoricamente o problema. Além disso, os trabalhos teóricos fazem uso de um modelo simplificado do problema real, ignorando variáveis importantes como o tamanho dos dados necessários para executar as tarefas, tempo necessário para a tarefa trafegar na rede, topologia da rede, falhas nas máquinas e restrições de hardware. Estas variáveis abrem espaço para muita pesquisa teórica na área.

Dos algoritmos estudados, RR pareceu ser interessante de ser implementado em ambientes reais, pois é um algoritmo simples e não faz uso de recursos caros como a predição. Além disso, nossos experimentos mostram que ele obtém bons resultados no objetivo de reduzir o makespan.

Nós fizemos uma nova análise do algoritmo RR para os problemas $R; s_{it}|T_j = L|TPCC$ e $R; s_{it}|T_j|TPCC$ mostrando uma aproximação e um conjunto de exemplos provando que a aproximação é justa. Sendo que este é o primeiro trabalho que mostra uma aproximação para o problema $R; s_{it}|T_j|TPCC$.

Suspeitamos que o algoritmo LR também irá obter um bom resultado, pois ele faz uso do algoritmo RR internamente para resolver os subproblemas. Mas para ambientes reais nós sugerimos uma pequena alteração no algoritmo, mudando o nível de uma tarefa sempre que todas as suas predecessoras tiverem finalizadas.

Já o algoritmo Grid Concurrent-Submission, como os próprios autores falaram, não é um algoritmo aplicável em ambientes reais, pois eles criam um algoritmo pouco eficiente com o objetivo de facilitar a análise da aproximação do algoritmo.

Usando a ideia de replicação apresentada no algoritmo RR e no algoritmo WQRxx, definimos uma interface que adiciona a característica de replicação de tarefas a qualquer algoritmo de escalonamento. Essa interface também permite que a quantidade de réplicas de cada tarefa possa ser controlada. Usando essa interface fizemos simulações onde

pudemos perceber ganhos significativos na redução do makespan, mas esses ganhos geram desperdícios de processamento. Então é interessante fazer uma análise de custo benefício entre a quantidade de réplicas e o ganho no makespan.

A simulação também mostrou que os algoritmos DFPLTF-R e o RR obtiveram os melhores resultados. Então se o ambiente prover a um baixo custo os mecanismos de predição é recomendável usar o algoritmo DFPLTF-R, caso contrário use o algoritmo RR.

6.1 Trabalhos futuros

Vamos apresentar alguns pontos que consideramos promissores:

- Estudar o problema de escalonamento de tarefas em grades computacionais levando em consideração variáveis como o atraso da rede, tamanho das tarefas e outras.
- Pesquisar por algoritmos probabilísticos para os problema de escalonamento.
- Adicionar um pouco mais de inteligência ao RR e fazer mais testes. Por exemplo, o RR priorizar o escalonamento em processadores que historicamente executaram as tarefas mais rapidamente.

Referências Bibliográficas

- [1] F. Afrati, E. Bambis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Srividenko. Approximation schemes for minimizing average weighted completion time with release dates. In *Proc. of the 40th IEEE Symposium on Foundations of Computer Science*, 1999.
- [2] J.L. Bruno, E.G. Coffman Jr, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.
- [3] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop*, pages 349–363, 2000.
- [4] S. Chakrabarti and S. Muthukrishnan. Resource scheduling for parallel database and scientific applications. In *Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 329–335, June 1996.
- [5] S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In *ICALP*, pages 646–657, 1996.
- [6] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. *SIAM J. Comput.*, 31(1):146–166, 2001.
- [7] C.S. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B.R.Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with applications to super blocks. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 58–67, December 1996.
- [8] W. Cirne, F. Brasileiro, D. Paranhos, L.F.W. Góes, and W. Voorsluys. On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Computing*, 33(3):213–234, 2007.
- [9] J. R. Correa and M. R. Wagner. Lp-based online scheduling: From single to parallel machines. In *IPCO*, pages 196–209, 2005.

- [10] D. P. da Silva, W. Cirne, and F. V. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Euro-Par*, pages 169–180, 2003.
- [11] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. *Queen's University School of Computing*, 2006.
- [12] I. Foster. What is the grid? a three point checklist. *GRID today*, 1(6):32–36, 2002.
- [13] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *ICPP*, pages 391–398, 2003.
- [14] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of precedence constrained coarse-grained tasks onto a computational grid. In *Second International Symposium on Parallel and Distributed Computing (ISPDC 2003)*, pages 80–87. Citeseer, 2003.
- [15] Noriyuki Fujimoto and Kenichi Hagihara. A comparison among grid scheduling algorithms for independent coarse-grained tasks. In *SAINT Workshops*, pages 674–680, 2004.
- [16] M. Goemans, J. Wein, and D. Williamson. A 1.47-approximation algorithm for a preemptive single-machine scheduling problem. *Operations Research Letters*, 26:149–154, 2000.
- [17] M.X. Goemans, M. Queyranne, A. Schulz, M. Skutella, and Y. Wang. Single machine scheduling with release dates. *Submitted*, 1999.
- [18] M.X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the Association for Computing Machinery*, 42:1115–1145, 1995.
- [19] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
- [20] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.
- [21] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.

- [22] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: off-line and on-line algorithms. In *Proc. of the Sixth ACM-SIAM Symposium on Discrete Algorithms*, January 1996.
- [23] W. Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21:846–847, 1973.
- [24] D. Karger, R. Motwani, and M. Sudan. Approximate graph coloring by semidefinite programming. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 2–13, 1994.
- [25] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *Handbooks in Operations Research and Management Science*, volume 4, chapter Sequencing and scheduling: algorithms and complexity, pages 445–522. North Holland, 1993.
- [26] J.K. Lenstra, A.H. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.
- [27] K. Liu, J. Chen, H. Jin, and Y. Yang. A Min-Min average algorithm for scheduling transaction-intensive grid workflows. In *Proceedings of the 7th Australasian Symposium on Grid Computing and e-Research (AusGrid 2009)*. Wellington, New Zealand: Australian Computer Society, Inc, pages 41–48, 2009.
- [28] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *8th Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, 1999.
- [29] N. Megow and A. S. Schulz. On-line scheduling to minimize average completion time revisited. *Oper. Res. Lett.*, 32(5):485–490, 2004.
- [30] N. Megow, M. Uetz, and T. Vredeveld. Models and algorithms for stochastic online scheduling. *Mathematics of Operations Research*, 41(3):513–525, 2006.
- [31] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2004.
- [32] C. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In *Proc. of the 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes on Computer Science*, pages 86–97, 1995.
- [33] S. Sahni. Algorithms for scheduling independent tasks. *Journal of the Association for Computing Machinery*, 23:116–127, 1976.

- [34] A. Schulz and M. Skutella. The power of alpha-points in preemptive single machine scheduling, 2000.
- [35] A. S. Schulz. Scheduling to minimize total weighted completion time: Performance guarantees of lp-based heuristics and lower bounds. In W.H. Cunningham, S.T. McCormick, and M. Queyranne, editors, *Integer Programming and Combinatorial Optimization*, volume 1084 of *Lecture Notes in Computer Science*, pages 301–315, 1996.
- [36] A. S. Schulz and M. Skutella. Scheduling unrelated machines by randomized rounding. *SIAM J. Discrete Math.*, 15(4):450–469, 2002.
- [37] U. Schwiegelshohn, A. Tchernykh, and R. Yahyapour. Online scheduling in grids. In *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*. IEEE Press, Los Alamitos, 2008.
- [38] D.B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:131–140, 1991.
- [39] M. Skutella. Convex quadratic and semidefinite programming relaxations in scheduling. *J. ACM*, 48(2):206–242, 2001.
- [40] M. Skutella and M. Uetz. Scheduling precedence-constrained jobs with stochastic processing times on parallel machines. In *SODA*, pages 589–590, 2001.
- [41] M. Skutella and G. Woeginger. A ptas for minimizing total weighted completion time on identical parallel machines. In *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, pages 472–481, 1998.
- [42] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.