Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 25, Num. 2 (2018) 56-72

RESEARCH ARTICLE

Gang Scheduling Algorithms with Migration Strategies in an Environment MCMCA

Algoritmos de Escalonamento Gangue com Estratégias de Migração em um Ambiente MCMCA

Francisca Aparecida P. Pinto[1]*, Henrique Jorge A. Holanda[2], Giovanni Cordeiro Barroso[3], Carla K. de M. Marques[2]

Abstract: Over the past decade, the fast advance of network technologies, hardware and middleware, as well as software resource sophistication has contributed to the emergence of new computational models. Consequently, there was a capacity increasing for efficient and effective use of resources distributed aiming to integrate them, in order to provide a widely distributed environment, which computational capacity could be used to solve complex computer problems. The two most challenging aspects of distributed systems are resource management and task scheduling. This work contributes to minimize such problems by (i) the use of migration techniques; (ii) implementing a multicore multicluster simulation environment with mechanisms for load balancing with the purpose of analyzing the system in different contexts; (iii) plus, the gang scheduling implementation algorithms will be analyzed through the use of metrics, in order to measure the schedulers performance in different situations. Thus, the results showed a better use of resources, implying operating costs reduction.

Keywords: Distributed systems — Parallel jobs — Techniques migration — Gang scheduling

Resumo: Ao longo da última década, o rápido avanço das tecnologias de rede, hardware e middleware, bem como a sofisticação dos recursos de software contribuíram para o surgimento de novos modelos computacionais. Consequentemente, houve uma capacidade crescente para o uso eficiente e efetivo de recursos distribuídos visando integrá-los, de modo a fornecer um ambiente amplamente distribuído, cuja capacidade computacional podendo ser utilizada para resolver problemas complexos. Os dois aspectos mais desafiadores dos sistemas distribuídos, são o gerenciamento de recursos e o escalonamento de tarefas. Este trabalho contribui para minimizar tais problemáticas: (i) através do uso de técnicas de migração; (ii) a implementação de um ambiente de simulação multicore multicluster com mecanismo de balanceamento de carga, a fim de analisar o sistema em diversos contextos; (iii) implementação e análise dos escalonadores de gangues através de métricas com intuito de medir o desempenho em diferentes situações. Assim, os resultados mostraram um melhor uso dos recursos, implicando na redução de custos operacionais.

Palavras-Chave: Sistemas distribuídos — Jobs paralelos — Técnicas de migração — Escalonamento de gangue

[1] *Departamento de Informática - Universidade Estadual do Rio Grande do Norte (UERN) e Universidade Federal Rural do Semi-Árido (UFERSA)*
[2] *Departamento de Informática - Universidade Estadual do Rio Grande do Norte, Brazil - RN - Mossoró*
[3] *Programa de Pós-Graduação de Engenharia de Teleinformática - Universidade Federal do Ceará (UFC), Brazil - CE - Fortaleza*
***Corresponding author**: aparecidapradop@gmail.com
DOI: http://dx.doi.org/10.22456/2175-2745.79994 • **Received:** 10/04/2018 • **Accepted:** 16/05/2018

1. Introdução

Na última década, as plataformas de computação em Cluster, Grade e computação em Nuvem emergiram como importantes fontes de poder computacional [1] [2] [3]. Todas essas plataformas podem consistir em computadores distribuídos e conectados através de uma rede de alto desempenho [3]. Tradicionalmente, o foco principal da indústria tem sido a melhoria do desempenho de sistemas computacionais, através de projetos mais eficientes, ampliando a densidade dos seus

componentes. Associado ao crescimento exponencial do tamanho dos dados em simulação/instrumentação científica, armazenamento e publicações na Internet, o aumento do poder computacional de tais sistemas impulsionou o investimento por parte tanto dos grandes provedores quanto do governo e laboratórios de pesquisa em ambientes computacionais, cada vez mais potentes, a fim de hospedar aplicações que vão desde redes sociais até workflows científicos [1].

Nesse contexto, os sistemas distribuídos surgem como uma solução interessante para prover recursos físicos sob demanda, pois permitem adicionar poder computacional de diversos nós interligados através de uma rede de computadores, a fim de executar tarefas. Sistemas de computação distribuídos têm sido utilizados devido a seus atributos importantes, tais como custo eficiente, escalabilidade, desempenho e confiabilidade. Em grade computacional existem três aspectos importantes que devem ser tratados: o gerenciamento de tarefas, o escalonamento de tarefas e a gestão de recursos [4]. Em particular, o *Grid Task Scheduling* (GTS) desempenha um papel importante em todo o sistema, sendo que seus algoritmos têm um efeito direto sobre o sistema de grade. O escalonamento de tarefa no ambiente de computação heterogêneo provou ser um problema NP-completo (NP-Complete) [1] [5] [6] [7] e ainda tem atraído a atenção dos pesquisadores.

A fim de resolver esse problema, vários tipos de algoritmos de escalonamento foram propostos para ambientes distribuídos, sendo classificados em diversas formas. Por exemplo, os autores [8], propõem uma classificação em forma de árvore, que divide, na hierarquia mais alta, os algoritmos em locais e globais. Os autores [9] definem uma taxonomia para problemas de escalonamento em plataformas de computação de grade. Smanchat e Viriyapant [10] estenderam a taxonomia de grade para definir uma taxonomia de problemas de escalonamento em computação em nuvem. Em [11] é definida uma taxonomia geral, oferecendo modelos conceituais para problemas e soluções de escalonamento. Além disso, eles apresentaram uma análise do impacto desse assunto ainda na pesquisa. Além do conjunto de características apresentadas acerca da taxonomia, existem várias heurísticas para escalonamento de tarefas em ambientes distribuídos.

Na literatura, encontram-se diversos algoritmos de escalonamento [12] [13] [14] [15] [16] [17] [18] que se adaptam a diferentes tipos de problemas e sistemas. Dentre eles, destacamos os algoritmos de escalonamento de tarefas mais tradicionais, como *First Come First Served* (FCFS), *Shortest Job First* (SJF), *Opportunistic Load Balancing* (OLB), *Minimum Execution Time* (MET), *Minimum Completion Time* (MCT), MinMin e MaxMin.

Entre as técnicas de escalonamento existentes, destacamos os escalonamentos de *gang* (gangue) ou grupo ou coescalonadores [19] [20], os quais são considerados algoritmos eficientes para escalonamento de *jobs* paralelos, que consistem em tarefas que devem ser alocadas e executadas simultaneamente em diferentes processadores. Esses tipos de algoritmos de escalonamento proveem tempo de resposta interativo para tarefas com baixo tempo de execução, por meio de preempção, e como desvantagem ocasionam a fragmentação, reduzindo o desempenho do sistema [21] [22] [23] [24] [25] [26] [16] [27]. De maneira análoga à fragmentação externa em memória, a fragmentação de recursos ocorre em uma grade computacional constituída por um conjunto de *clusters*, quando não se consegue localizar um *cluster* que possa executar as tarefas de um *job* simultaneamente, sendo o número total de recursos computacionais ociosos em toda a grade maior do que esse número de tarefas. A fragmentação ocorre no sistema quando este apresenta processadores livres, mas os requisitos computacionais dos *jobs* não podem ser cumpridos, permanecendo, assim, recursos inativos [28]. A fragmentação dos recursos tem sido um tema de pesquisa comum nas duas últimas décadas. Várias abordagens para a fragmentação de recursos foram desenvolvidas; melhor ajuste (*Best-fit*) e migração de tarefas (*task migration*) são as duas abordagens mais comuns.

Baseado no exposto acima, este trabalho tem como objetivo investir na redução da fragmentação ocasionada pelos escalonamentos de gangue, bem como no tempo de resposta. Dentre as principais contribuições deste trabalho, pode-se destacar:

1. A implementação de heurísticas, *Adapted First Come First Served* (AFCFS), *Largest Slowdown First* (LXF) e *Largest Job First Served* (LJFS) utilizando mecanismo gangue. Partindo do pressuposto de que o escalonamento de gangue ocasiona fragmentação no ambiente, busca-se, o uso de mecanismos de migração, como, por exemplo, verificar os *clusters* que apresentam processadores disponíveis; analisar qual dos jobs tem suas tarefas no começo da fila destes últimos; e verificar o job que tem o menor número de tarefas para migrar. Com o uso de estes reduzir a fragmentação e, consequentemente, o tempo de resposta dos jobs. Além dessas estratégias de migração, utilizamos mecanismos para evitar migrações desnecessárias, bem como a sobrecarga no sistema;

2. A implementação de técnicas e algoritmos, *Join The Shortest Execution Queue* (JSEQ) e *Opportunistic Load Balancing* (OLB), para balanceamento de carga nos gerenciadores dos despachos (*dispatchers*), grade e local, visando à distribuição dos *jobs* para os *clusters*, a fim de reduzir o tempo de espera das tarefas, e, consequentemente, melhorar a eficiência do sistema;

3. Baseado em um estudo de simuladores para ambientes de grade, a composição de um ambiente de simulação utilizando uma Arquitetura Multicore Multicluster, com o objetivo de analisar o desempenho dos escalonadores em situações diversas, bem como o comportamento do sistema em diferentes contextos. Esse ambiente proposto é intitulado de *Grid Schedule Management Simulator* (GSMSim);

4. Ainda como contribuição, os algoritmos de escalonamento implementados serão analisados em contextos diferentes por métricas, *Average Wait Time* (Tempo Médio de Espera), *Average Response Time* (Tempo Médio de Resposta), *Loss of Capacity* (Perda de Capacidade) e *Utilization* (Utilização), com o intuito de medir tanto a utilização do sistema quanto a fragmentação.

O artigo está organizado da seguinte forma: na Seção 2, apresentam-se trabalhos relacionados; na Seção 3, apresenta-se o modelo do sistema proposto; na Seção 4, descreve-se o funcionamento do sistema; nas Seções 5 e 6, mostram-se os escalonadores gangue e os mecanismos de migração; na Seção 7, apresentam-se as métricas de desempenho, que são utilizadas para analisar o desempenho dos escalonadores em situações diversas; na Seção 8, apresentam-se os resultados das simulações; na Seção 9, encontra-se a conclusão do trabalho, juntamente com as perspectivas de desenvolvimento futuro.

2. Trabalhos relacionados

Este trabalho tem como objetivo investir na redução da fragmentação ocasionada pelos escalonamentos de gangue, bem como na redução do tempo de resposta dos *jobs*. Pesquisadores buscam mecanismos eficientes para reduzir o tempo de execução, bem como melhorar a utilização dos recursos e, consequentemente, minimizar a fragmentação. A seguir, são apresentados alguns trabalhos nesta área.

Através do estudo das técnicas de escalonamentos de gangue, [29] [23] [24] [25] [30] [26] [31] [16] [18] [27], pode-se observar que tais técnicas são eficazes para alocação de *jobs* paralelos em ambientes distribuídos, causando como desvantagem a fragmentação. Esta última acontece no sistema quando existem tarefas em espera na fila para executar e há processadores ociosos, mas que ainda não podem executar as tarefas em espera.

Em [24] [32] [26], são propostos mecanismos de migração, a fim de minimizar a fragmentação ocasionada pelos escalonadores de gangue nesses ambientes. A partir desses estudos de migração em sistemas heterogêneos, implementamos algumas estratégias com o objetivo de minimizar tal problemática, como, por exemplo, verificar os *clusters* que apresentam processadores disponíveis, analisar qual dos *jobs* tem suas tarefas no começo da fila destes últimos, e verificar o *job* que tem o menor número de tarefas para migrar. O estudo deste trabalho foi apresentado em [33] [34] aplicando os mecanismos de migração nos escalonadores *Adapted First Come First Served* (AFCFS) e *Largest Gang First Served* (LGFS), em um ambiente de simulação *multicluster* utilizando uma estrutura hierárquica de duas camadas, composto por gerenciadores, *Grid Dispatcher* (GD) e *Local Dispatcher* (LD) para a alocação dos jobs nos recursos. Estes autores, visando a um balanceamento de carga mais eficiente, consideraram um conjunto de estratégias para balanceamento de carga: primeira estratégia, introduziu-se no GD, antes do envio dos *jobs* para os *clusters*, um *feedback* de informações a respeito das cargas dos *clusters*, para um balanceamento de carga mais eficiente; e, segunda estratégia, utilizou-se um algoritmo no LD, *Join The Shortest Queue* (JSQ), que aplica técnica na distribuição das tarefas para a fila dos processadores. Essa distribuição é feita de acordo com o número de tarefas na fila do processador mais a tarefa em execução, ou seja, não levando em consideração o tempo de execução das tarefas. Diferente de [33] em [34] foram analisados os algoritmos AFCFS e LGFS em um sistema de simulação multicluster heterogêneo, este último em relação à quantidade de recursos por clusters. Além disso, foram aplicados em ambientes diferentes tamanhos de carga de trabalho.

Nesta proposta, a partir do estudo das taxonomias de escalonamentos em ambientes distribuídos, [35] [9] [36] [10] [18], utiliza-se no modelo de simulação uma Arquitetura *Multicore* Multicluster (MCMCA)[37], a fim de atender a uma demanda maior de conjunto de dados. Neste ambiente, a heterogeneidade acontece em relação ao número de recursos por *cluster*, à taxa de *clock* dos recursos e às características dos recursos em cada *cluster*. Tais dados são compostos por dois tipos diferentes de *jobs*, sequencial e paralelo. Este último consiste em várias tarefas que são independentes e executadas simultaneamente, já o *job* sequencial é uma tarefa de prioridade que requer apenas um processador para sua execução e o menor tempo de processamento estimado em relação aos outros jobs. Portanto, ao chegar ao ambiente, o job é enviado para o melhor processador disponível. Com isso, pretende-se reduzir o tempo de execução dos *jobs*. Além disso, são utilizados dois algoritmos no LD: *Join The Shortest Execution Queue* (JSEQ) e *Opportunistic Load Balancing* (OLB) [38], os quais aplicam técnicas na distribuição das tarefas para a fila dos processadores. Tais algoritmos têm como intuito reduzir o tempo da tarefa na fila, bem como o tempo de resposta de um *job* e, por conseguinte, a fragmentação. O JSEQ é uma adaptação do JSQ, o qual foi aplicado nos trabalhos dos autores [33] [34]. Com esta adaptação pretende reduzir ainda mais o tempo de espera das tarefas nas filas. Ademais, são aplicadas as seguintes políticas para o escalonamento das filas: AFCFS, LJFS e LXF. Esses escalonadores são adaptados para o mecanismo gangue, a fim de escalonar as tarefas dos *jobs* alocados a filas, e implementados no ambiente de simulação. Tais políticas são avaliadas separadamente no sistema em situações diversas, utilizando métricas para mensurar tanto a utilização do sistema quanto a fragmentação.

3. Proposta Simulador - Grid Schedule Management Simulator

Neste trabalho é proposto um modelo de simulador multicore multicluster baseado em filas. Uma metodologia de simulação é aplicada a fim de validar o modelo e quantificar o desempenho em condições realistas (ver Figura 1). O sistema de simulação *Grid Schedule Management Simulator* (GSMSim) consiste de um ambiente multicore *multicluster*, utilizando

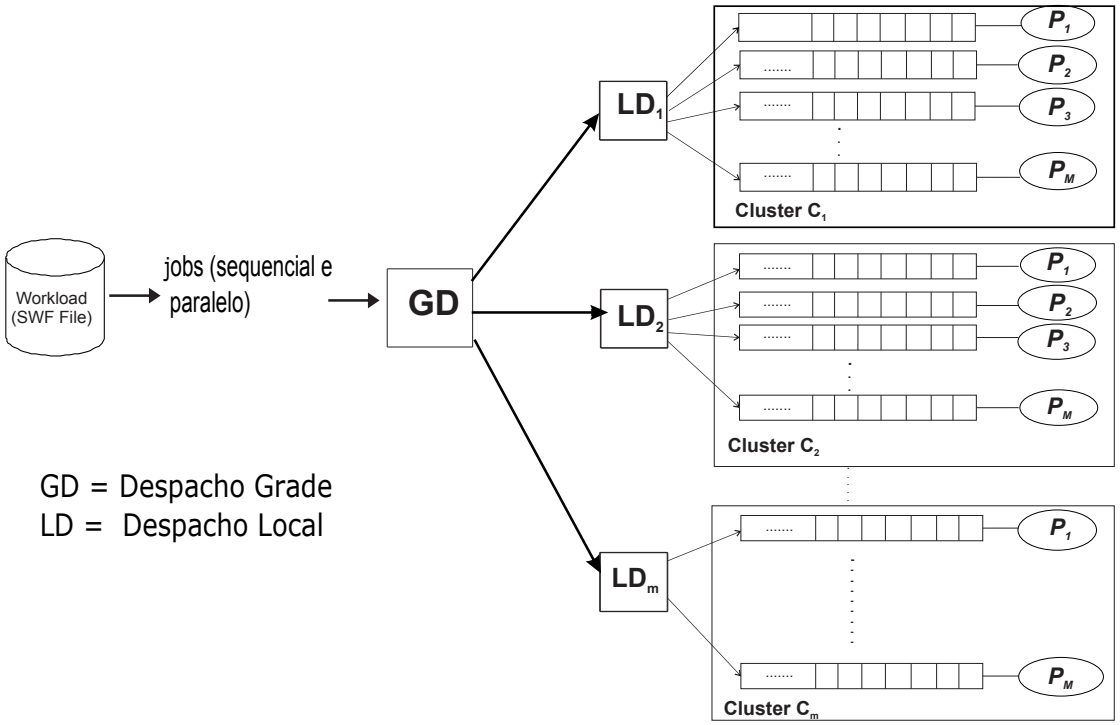

Figura 1. Modelo do GSMSim *multicore* multicluster baseado em filas

uma estrutura hierárquica de duas camadas. O GSMSim foi implementado no intuito de analisar o desempenho dos escalonadores em situações diversas, bem como o comportamento do ambiente em diferentes contextos. Este sistema foi implementado no laboratório do Grupo de Pesquisa em Modelagem Computacional Aplicada (GrPeC) da Universidade Federal do Ceará (UFC).

O modelo do GSMSim é baseado em teoria de filas (Figura 1), que são úteis para a análise de sistemas, nos quais ocorrem conflitos quando diversas entidades tentam acessar simultaneamente o mesmo recurso [39] [40]. O GSMSim é composto por gerenciadores, despacho da grade (GD), e despacho local (LD), administradores dos *clusters*.

O GD é responsável pelo envio dos *jobs*, sequenciais e paralelos, para os *clusters*, e o LD é responsável por enviar a(s) tarefa(s) pertencentes aos *jobs* para as filas dos processadores. Cada LD é composto por um *cluster* (C_i) (i variando de *1* a m) constituído por um conjunto de processadores *multicore* (P_l) (l variando de *1* a M), sendo $\{M, i, l, m \in N\}$. Além disso, cada P_l tem a sua própria fila no sistema.

No sistema, utilizamos diferentes cenários em relação ao número de processadores, características das máquinas e quantidade dos *clusters*, a fim de simular *workloads* (cargas de trabalho), que apresentem *jobs* com vários níveis de paralelismo. Neste trabalho, consideramos que um sistema é homogêneo quando a taxa do *clock* das máquinas é igual e cada C_i possuindo uma quantidade de processadores diferente; da mesma forma, um sistema é heterogêneo quando a taxa do *clock* das máquinas é diferente, podendo variar de $\{1500, 1600, 1700, 1800, 1900, 2000, 2500, 3000, 3500\}$ (megahertz), gerado aleatoriamente no momento da criação dos recursos no ambiente de simulação. Portanto, existe a heterogeneidade nos recursos do mesmo *cluster* e, consequentemente, entre os *clusters*. Dessa forma, o ambiente proposto pode ser utilizado em diferentes cenários.

No ambiente desenvolvido, os *clusters* pertencem ao domínio administrativo, de modo que eles são capazes de comunicar-se com o GD. Além disso, a comunicação entre os processadores é de contenção livre. Desta forma, a latência de comunicação é calculada da seguinte forma [41] [42] [43]:

$$T(z) = \alpha + \frac{z}{\rho} \tag{1}$$

em que α é uma constante, z representa o tamanho de um *job* (megabytes) e ρ é a largura de banda (megabytes).

3.1 *Workload*

As cargas de trabalho foram extraídas de um ambiente distribuído real e apresentam características do *Standard Workload Format* (SWF) [44] [45] [46]. Elas são compostas por dois tipos diferentes de *jobs*, que estão competindo pelos mesmos recursos: *job* sequencial; e *job* paralelo. Uma carga de trabalho $W = \{J_1, J_2, ..., J_j\}$ $(j = 1, 2, 3, \ldots)$ é composta por vários *jobs*, onde um *job* J_j é representado por uma tupla (id_j, at_j, s_j, pt_j). Veja a descrição dos parâmetros na Tabela 2.

Um *job* J_j é constituído por uma ou mais tarefas, ou seja, $J_j = \{v_{1,j}, \ldots, v_{i,j}\}$. Se $J_j = \{v_{1,j}\}$, então, o $J_j = 1$ é um *job* sequencial composto por uma única tarefa, que requer apenas um processador para sua execução. Portanto, ela é uma tarefa

Tabela 1. Os parâmetros do *workload*.

Parâmetros	Descrição
id_j	identificação do *job*, ($id_j = 1,2,3\ldots$).
at_j	tempo de chegada, ($at_j \geq 0$).
s_j	número de tarefas de um *job*, ($s_j \geq 1$).
pt_j	estimativa do tempo de processamento de um *job*, ($pt_j > 0$).

de alta prioridade, uma vez que, chegando no ambiente, é enviada para o processador disponível de maior velocidade. Isso somente acontecerá, quando essa tarefa apresentar menor tempo de processamento estimado pt_j em relação aos outros *jobs*. No caso de todos os processadores estarem ocupados, essa tarefa é enviada para a fila do processador que apresenta menor tempo de execução. Já um *job* paralelo J_j consiste em $v_{j,|J_j|}$ tarefas em que $|J_j| > 1$. O mapeamento entre as tarefas e os processadores deve ser de um para um. Portanto, as tarefas de um *job* não podem ser atribuídas para a mesma fila do processador. Além disso, as tarefas pertencentes a um *job* paralelo serão escalonadas para execução, conforme a técnica de escalonamento de fila no sistema.

4. Funcionamento do GSMSim

Nesta seção, é descrito com detalhes o funcionamento dos gerenciadores do sistema, ou seja, o GD e LD, como é ilustrado na Figura. 1.

4.1 Despacho Grade

O GD é responsável pelo envio dos *jobs* para os *clusters*. Esse envio é feito baseado em um *feedback* de informações a respeito da carga total de cada *cluster*, ou seja, o número total de tarefas nas filas mais o número de tarefas em execução nos processadores (Algoritmos 1 e 2). Essas informações a respeito das cargas dos *clusters* somente serão enviadas mediante solicitação do GD, pois *feedbacks* excessivos podem ocasionar sobrecarga no sistema. É muito importante saber o valor da carga de cada *cluster* para um balanceamento de carga mais eficiente. No caso de os *clusters* estarem balanceados, acontece o despacho randômico.

Algoritmo 1: Despacho grade(job)

Entrada: jobs
Saída: V (job pode ser executado pelo sistema) ou F (sistema não pode executar o job)

```
1  início
2      conjunto S ← vazio
3      clusters_selecionaveis ← 0
4      para i ← 1 até num_clusters faça
5          se (cluster[i].num_processadores) <
            (job.num_tarefas) então
6              S ← S U (cluster[i])
7              clusters_selecionaveis ←
                clusters_selecionaveis + 1
8          fim
9      fim
10     se clusters_selecionaveis > 0 então
11         se job.num_tasks > 1 então
12             Cluster ← menor_carga(S)
13             Cluster.LocalDispatcher(job)
14         senão
15             Cluster ← aleatorio(S)
16             Cluster.LocalDispatcher(job)
17         fim
18         retorna (V)
19     senão
20         retorna (F)
21     fim
22 fim
```

4.2 Despacho Local

Depois de um *job* paralelo J_j ter sido enviado para o *cluster* C_i, de acordo com a menor carga de LC_i, o LD atribui as tarefas do *job* para as filas disponíveis, com base no algoritmo *Opportunistic Load Balancing*, ou no algoritmo *Join The Shortest Execution Queue*, os quais foram adaptados e implementados nos LDs.

4.2.1 Opportunistic Load Balancing

O algoritmo *Opportunistic Load Balancing* (OLB) (Algoritmo 3) envia as tarefas pertencentes a um *job* para os processadores disponíveis ou para as filas dos mesmos, independentemente do tempo de execução esperado das tarefas nos processadores [47]. Esse algoritmo tem como vantagem manter as máquinas ocupadas e, como desvantagem, não se preocupar em minimizar o tempo de espera da tarefa na fila, consequentemente, o tempo de resposta de um *job*.

4.2.2 Join The Shortest Execution Queue

O algoritmo *Join The Shortest Execution Queue* (JSEQ) é uma adaptação proposta neste trabalho, tomando como base o *Join The Shortest Queue* (JSQ)[48] [33] [34]. O JSEQ (Algoritmo 4) é responsável pelo envio da(s) tarefa(s) que pertence(m) a um *job* para as filas dos processadores, tal que as tarefas já enfileiradas possuem o menor tempo de execução. Importante ressaltar que o valor do tempo de execução enviado para o LD é a soma do tempo de execução das tarefas na fila mais o tempo de execução da tarefa que está no processador, diferente do JSQ [33] [34], em que o envio das tarefas para processadores acontece através da quantidade de tarefas nas filas dos processadores. Isso pode acarretar um aumento do

Algoritmo 2: menor_carga(cluster[n])

Entrada: conjunto n clusters
Saída: (cluster que possui a menor carga)

```
1  início
2      para (i ← 1 até n) faça
3          se i ≐ 1 então
4              menor ← i
5              tamanho ← cluster[i].num_tarefas() / cluster[i].num_maquinas()
6          senão
7              se ( cluster[i].num_tarefas() / cluster[i].num_maquinas() < tamanho) então
8                  menor ← i
9                  tamanho ← cluster[i].num_tarefas() / cluster[i].num_maquinas()
10             fim
11         fim
12     fim
13     retorna (cluster[menor])
14 fim
```

Algoritmo 3: algortimo OLB

Entrada: uma gangue g

```
1  início
2      Lista S ← vazio
3      Lista T ← vazio
4      para i ← 1 até cluster.num_proc faça
5          se cluster.proc[i].tarefa_executando = null
             então
6              S.incluir(cluster.proc[i])
7          senão
8              T.incluir(cluster.proc[i])
9          fim
10     fim
11     ordenar_aleatorio(S)
12     ordenar_aleatorio(T)
13     cluster.proc ← vazio
14     cluster.proc ← S.concatena(T)
15     para i ← 1 até g.num_tarefas faça
16         cluster.proc[i].incluir(g.tarefas[i])
17     fim
18 fim
```

Algoritmo 4: Algoritmo JSEQ

Entrada: uma gangue g

```
1  início
2      para i ← 1 até cluster.num_proc faça
3          eleito ← cluster.proc[i]
4          j ← i - 1
5          enquanto (j ≥ 0 ) e (eleito.tempo_estimado <
             cluster.proc[j].tempo_estimado) faça
6              cluster.proc[j + 1] := cluster.proc[j]
7              j := j - 1
8          fim
9          cluster.proc[j+1] ← eleito
10     fim
11     para i ← 1 até g.num_tarefas faça
12         cluster.proc[i].incluir(g.tarefas[i])
13     fim
14 fim
```

tempo de espera das tarefas nas filas.

Desta forma, quando um *job* sequencial chega no ambiente, o mesmo tem prioridade, conforme a explicação na Seção 3.1, independentemente do algoritmo que esteja atuando no LD. Além disso, o *feedback* das informações a respeito do comportamento das filas dos processadores somente ocorre quando o LD solicita, evitando, assim, sobrecarga no sistema.

Após a distribuição das tarefas nas filas por um dos algoritmos de máquina (OLB ou JSEQ), usa-se um dos escalonadores de fila *Adapted First Come First Served* (AFCFS); *Largest Job First Served* (LJFS) [49] [23] [26] [25] [24] ou *Largest Slowdown First* (LXF) [50] para escalonar as tarefas nas filas.

Na próxima seção, são apresentados tais escalonadores utilizando a técnica de gangue e adaptados para o escalonamento das tarefas nas filas dos processadores.

5. Escalonamento de Gangue

O escalonamento de gangue é muito aplicado em escalonamento de *job*, onde cada *job* é composto por um conjunto de tarefas que devem ser executadas simultaneamente em processadores distintos [51] [52]. Este tipo de escalonamento de gangue é considerado eficiente para escalonamento de *jobs* paralelos em ambientes distribuídos e, como desvantagem, ocasionam a fragmentação, reduzindo o desempenho do sistema [53] [54] [52] [23] [24] [25] [26].

No sistema de simulação, foram aplicadas as seguintes políticas para o escalonamento das filas: AFCFS, LJFS e LXF. Esses escalonadores foram adaptados para o mecanismo gangue, a fim de escalonar as tarefas dos *jobs* alocadas a filas, e implementados no ambiente de simulação.

5.1 AFCFS

O algoritmo AFCFS (Algoritmo 5) tende a favorecer *jobs* com um número de tarefas menor, e, consequentemente, exige menor número de processadores. Por outro lado, isso ocasiona aumento no tempo de resposta dos *jobs* maiores. No Algoritmo 5, na linha 2, inicializa-se o procedimento de busca nas filas dos processadores pelos *jobs* que apresentam número de tarefas menor e, em seguida, na linha 7 começa a ordenação de troca das tarefas.

Na Figura 2, é ilustrado um cenário, onde as tarefas pertencentes aos *jobs* $J_1 = v_{1,1}, \ldots, v_{4,1}$; $J_2 = v_{1,2}, \ldots, v_{3,2}$;

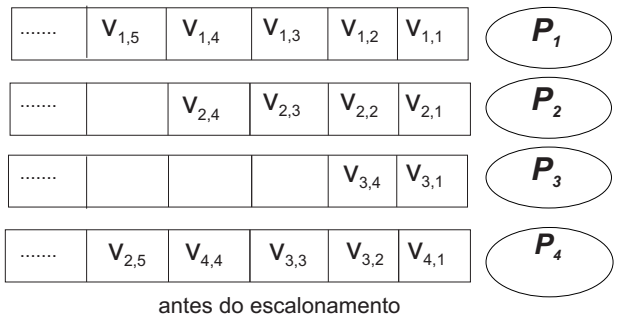

Figura 2. *Jobs* nas filas - antes do escalonamento

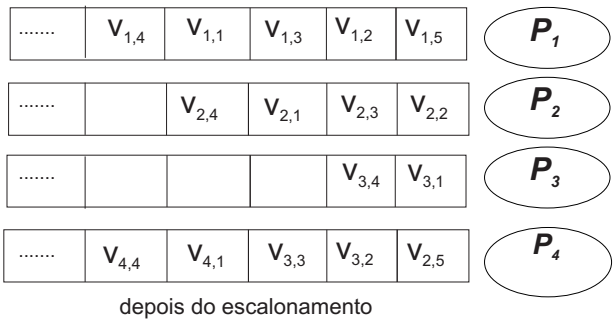

Figura 3. Depois do escalonamento AFCFS

Algoritmo 5: AFCFS (Processador.Fila *p.f*)

Entrada: Fila f de tarefas de um processador p

```
1  início
2    se p.f.começo ≠ null então
3      se iniciar(p.f.começo)≠ null então
4        se p.f.começo.prox ≠ null então
5          menor ← p.f.começo
6          aux ← p.f.começo.prox
7          enquanto aux ≠ null faça
8            se (aux.num_tarefas_pertence_job)
               <
             (menor.num_tarefas_pertence_job)
             então
9              menor ← aux
10           fim
11           aux ← aux.prox
12         fim
13         aux ← p.f.começo
14         p.f.começo ← menor
15         p.f.começo.prox ← aux
16         eliminar_duplicata(p.f.começo)
17       fim
18     fim
19   fim
20 fim
```

$J_3 = v_{1,3},\ldots,v_{3,3}$; $J_4 = v_{1,4},\ldots,v_{4,4}$; $J_5 = v_{1,5},v_{2,5}$ foram distribuidas para as filas dos processadores, conforme o tempo de chegada no sistema. Como podemos observar, os *jobs* requerem quantidades de processadores diferentes, $J_1 = 4$; $J_2 = 3$; $J_3 = 3$; $J_4 = 4$; $J_5 = 2$, respectivamente. Considerando o tamanho dos *jobs*, estes serão escalonados de acordo com o Algoritmo 5: $J_5 = v_{1,5},v_{2,5}$; $J_2 = v_{1,2},\ldots,v_{3,2}$; $J_3 = v_{1,3},\ldots,v_{3,3}$; $J_1 = v_{1,1},\ldots,v_{4,1}$; $J_4 = v_{1,4},\ldots,v_{4,4}$, como é mostrado na Figura 3. Essas tarefas foram distribuidas nas filas, conforme o algoritmo do LD (OLB ou JSEQ) e, depois, foram escalonadas de acordo com a política AFCFS.

O algoritmo AFCFS possui complexidade $O(n)$, sendo n é o número de tarefas na fila as quais serão escalonadas. É apenas $O(n)$ porque o escalonador passa pela fila uma vez para verificar qual é o *job* que possui menor número de tarefas, e, depois, encaminha para o começo da fila as tarefas apresentam menos tarefas irmãs. Essa situação pode ser realizada em tempo constante que seria $O(1)$. Sendo assim, complexidade $O(n)+O(1) \doteq O(n)$.

5.2 LJFS

O algoritmo LJFS (Algoritmo 6) tende a favorecer o desempenho dos *jobs* maiores à custa dos menores, ou seja, os *jobs* maiores têm suas tarefas alocadas nas filas dos processadores antes de qualquer outra que pertença a um *job* com tamanho menor, acarretando aumento no tempo de resposta dos *jobs* menores. Na Figura 5, é apresentado um

Algoritmo 6: LJFS (Processador.Fila p.f)

Entrada: Fila f de tarefas de um processador p

```
1  início
2    para i ← 1 até p.f.tamanho faça
3      eleito_proc ← p.f[i]
4      i ← i - 1
5      enquanto (j≥0) e (eleito.num_irmas >
         p.f[j].num_irmas) faça
6        p.f[j + 1] := p.f[j]
7        j:= j - 1
8      fim
9      p.f[j+1] ← eleito_proc
10   fim
11 fim
```

novo cenário, utilizando-se o LJFS para os mesmos *jobs* do exemplo anterior (Figura 2). Considerando o tamanho dos *jobs* paralelos (Figura 4), serão escalonados na seguinte ordem: $J_1 = v_{1,1},\ldots,v_{4,1}$; $J_4 = v_{1,4}\ldots v_{4,4}$; $J_2 = v_{1,2},\ldots,v_{3,2}$; $J_3 = v_{1,3},\ldots,v_{3,3}$; $J_5 = v_{1,5},v_{2,5}$, como é ilustrado na Figura 5.

O algoritmo LJFS possui complexidade $O(n*log(n))$, sendo n é o número de tarefas na fila que serão escalonadas. Como o escalonador se resume a reordenar a fila em ordem decrescente de acordo com o número de tarefas pertencentes ao *job*, a complexidade é a mesma que a de um método de ordenação comum, portanto, tomamos como base o método de ordenação *merge sort* [55].

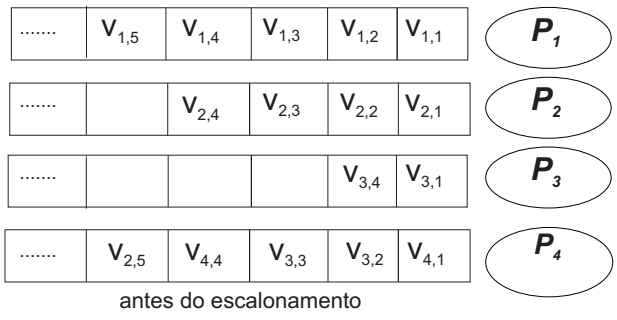

Figura 4. *Jobs* nas filas - antes do escalonamento

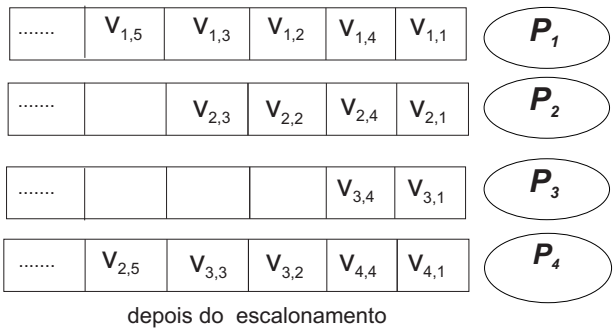

Figura 5. Depois do escalonamento LJFS

5.3 LXF

O algoritmo LXF (Algoritmo 7) tende a beneficiar *jobs* que apresentam maior fator de expansão (XF), que é frequentemente utilizado quando se comparam algoritmos de escalonamento. O LXF está relacionado à métrica XF, que é dada pela função objetivo do Algoritmo 7 (linha 5), onde o *eleito.processamento* é a estimativa do tempo de processamento de um *job* e o *eleito.tempo_espera* é o tempo de espera do *job* no sistema.

Algoritmo 7: LXF (Processador.Fila $p.f$)

Entrada: Fila f de tarefas de um processador p

```
1  início
2      para i ← 1 até p.f.tamanho faça
3          eleito ← p.f[i]
4          i ← i - 1
5          t1 ← (eleito.processamento+eleito.tempo_espera) / eleito.processamento
6          enquanto j>0 e t1 < (p.f[j].processamento+p.f[j].tempo_espera) / p.f[j].processamento faça
7              p.f[j + 1] := p.f[j]
8              j:= j - 1
9          fim
10         p.f[j+1] ← eleito
11     fim
12 fim
```

Baseado no exemplo da Seção 5.1, na Figura 7, é ilustrado um cenário usando-se o LXF. Os *jobs* paralelos J_1, J_2, J_3, J_4 e J_5 apresentam os seguintes XF, respectivamente: 1,6; 2,5; 1,8; 1,7; 1,3. Tais resultados foram calculados, utilizando a equação da linha 5 (Algoritmo 7), onde os valores do *eleito.processamento* (tempo de processamento estimado pt_j) e o *eleito.tempo_espera* (tempo de espera) são coletados a partir das informações dos *jobs*, J_1, J_2, J_3, J_4 e J_5, que serão executados. Considerando os *jobs* que apresentam maior XF, serão escalonados na seguinte ordem $J_2 = v_{1,2}, \ldots, v_{3,2}$; $J_3 = v_{1,3}, \ldots, v_{3,3}$; $J_4 = v_{1,4}, \ldots, v_{4,4}$; $J_1 = v_{1,1}, \ldots, v_{4,1}$; $J_5 = v_{1,5}, v_{2,5}$, como é ilustrado na Figura 7. O LXF tende a favorecer os jobs que mostram maiores tempos de execução no sistema.

O algoritmo LXF possui complexidade $O(n*log(n))$, sendo n o número de tarefas na fila do processador. Esse algoritmo consiste em ordenar as tarefas em ordem decrescente, conforme o resultado do XF (Seção 5.3). O XF de cada *job* é calculado em tempo constante. Com isso, consideramos o método *merge sort* como o algoritmo de ordenação do LXF.

6. Migração

Partindo do pressuposto de que o escalonamento de gangue ocasiona fragmentação no ambiente, busca-se reduzir a fragmentação, através do uso de migração de tarefas. Neste trabalho, foram estudados diferentes esquemas de migração para sistema heterogêneo, com o intuito de minimizar tal problemática. Portanto, assumimos dois tipos de migração: local m_l (algoritmo migração local) e externa m_e (algoritmo migração externa). Estes algoritmos foram detalhados no trabalho dos autores, [34] (Seção VI).

A m_l envolve a transferência de tarefas no mesmo cluster, já a m_e acontece a transferência de tarefas de um *cluster* para outro. A seguir, pontuamos algumas estratégias para evitar migrações desnecessárias e, consequentemente, sobrecarga no sistema. São elas:

1. verificam-se todos os *clusters* que apresentam processadores disponíveis;
2. analisa-se qual dos *jobs* tem suas tarefas no começo da fila dos processadores ociosos;
3. baseada na análise acima, verifica-se o *job* que tem o menor número de tarefas e que seja menor ou igual à quantidade de processadores ociosos;
4. por último, migram-se as tarefas do *job* que tem o menor número de tarefas.

Na Figura 8, é ilustrado um cenário de migração. Os processadores P_1, P_2 e P_3 estão disponíveis, as tarefas $v_{1,1}$ e $v_{2,1}$ se encontram, respectivamente, no começo da fila dos processadores P_1 e P_2, e a tarefa $v_{3,1}$ se encontra na fila do P_6, este último está ocupado e apresentando outras tarefas na fila à frente da tarefa $v_{3,1}$. Portanto, para que as tarefas $v_{1,1}$, $v_{2,1}$

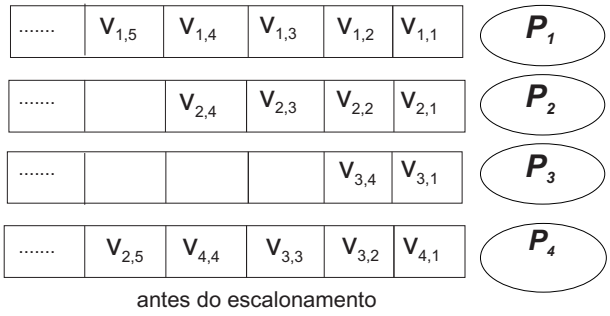

Figura 6. *Jobs* nas filas - antes do escalonamento

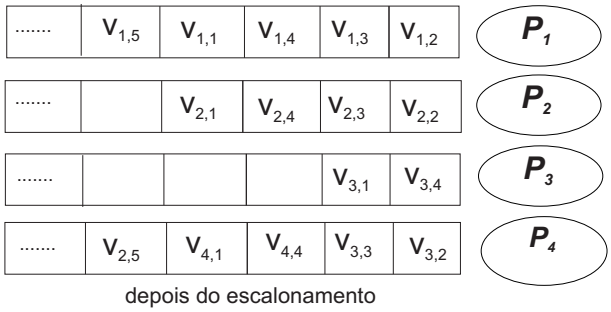

Figura 7. Depois do escalonamento LXF

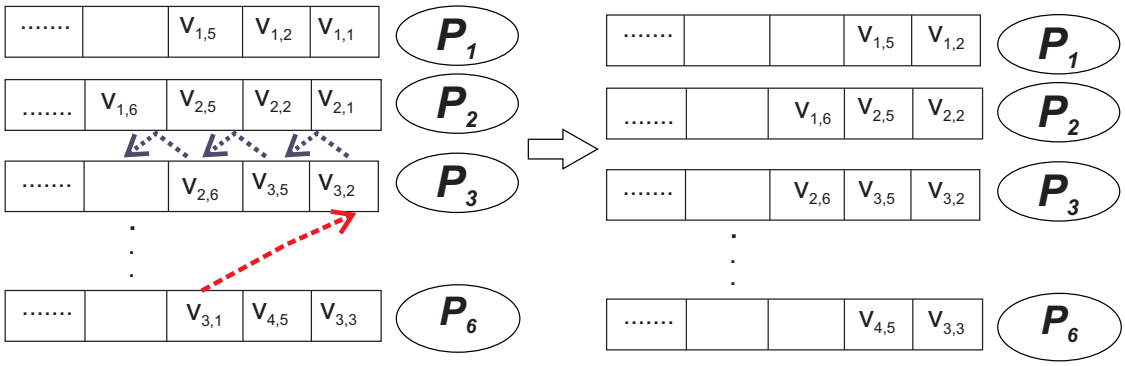

Figura 8. Exemplo de um cenário de migração

e $v_{3,1} \in J_1$ sejam executadas imediatamente, a $v_{3,1}$ é migrada para o P_3.

Durante a migração de tarefas, os processadores destinos são reservados, a fim de se evitar que outras tarefas possam utilizá-los. Ao reservar o processador de destino, garante-se que as tarefas migradas iniciem as suas execuções imediatamente. Importante ressaltar que a m_e só é aplicada quando a m_l não resolve o problema.

As estratégias de migração foram aplicadas nos algoritmos de escalonamentos AFCFS, LJFS e LXF (Seção 5), os quais foram utilizados como escalonador de fila no ambiente de simulação. Portanto, esses algoritmos com migração serão definidos como AFCFSm, LJFSm e LXFm.

A hierarquia de escalonamento exige que primeiro sejam executados os algoritmos de escalonamento (AFCFS, LXF ou LJFS) e, depois, a migração m_l tenta escalonar os *jobs* não alocados pelo algoritmo de escalonamento. A migração m_e somente será utilizado numa tentativa de se fazer uso de mais recursos.

Na próxima seção, são descritas as métricas de desempenho que foram aplicadas para analisar o comportamento do modelo do sistema em diferentes situações.

7. Métricas de Desempenho

Neste trabalho, foram aplicadas as seguintes métricas de desempenho: Tempo Médio de Espera (AWT), Tempo Médio de Resposta (ART), Perda de Capacidade (LoC) e Taxa de Utilização (U) [56] [57] [58] [59] [18], tudo no intuito de analisar o desempenho dos escalonadores em situações diversas, bem como o comportamento do sistema em diferentes contextos.

7.1 Tempo Médio de Espera

O AWT mede o tempo entre a chegada do *job* no sistema e o início de sua execução.

$$AWT = \frac{1}{w} \times \sum_{j=1}^{w} wt(j) \tag{2}$$

em que $wt(j)$ mede o tempo entre a chegada do *job* no sistema e o início de sua execução e w é o número total de *jobs* executados.

7.2 Tempo Médio de Resposta

A métrica tempo de resposta (em segundos) mede o intervalo de tempo entre a chegada do *job* no sistema até o final da sua execução. Assim, o ART é dado pela Equação 3.

$$ART = \frac{1}{w} \times \sum_{j=1}^{w} rt(j) \tag{3}$$

em que $rt(j)$ representa o tempo de resposta de um *job* e w é o número total de *jobs* executados.

7.3 Perda de Capacidade

Essa métrica é relevante para mensurar tanto a utilização do sistema quanto a fragmentação. Em um sistema acontece a fragmentação quando: (i) existem tarefas em espera na fila

para executar; e ii) há nós ociosos, mas que ainda não podem executar as tarefas em espera. A métrica perda de capacidade (LoC) tem sido utilizada em alguns trabalhos, como [56] [57] [58] [59]. Neste trabalho, a métrica LoC é calculada da seguinte forma:

$$LoC = \frac{\sum_{j=1}^{q-1} n_j(t_{j+1} - t_j)\delta_j}{N(t_q - t_1)} \times 100 \quad (4)$$

em que n_j representa o número de processadores ociosos durante o tempo $(t_{j+1} - t_j)$, N é o total de processadores no sistema, $t_q - t_1$ representa o tempo de chegada do primeiro *job* no sistema e a saída do último, e δ_j é o estado em que se encontra os processadores e *jobs* no sistema. Se $\delta_j = 1$ indica a existência de processadores disponíveis para executar pelo menos um *job* na fila, no instante em que um novo *job* é despachado; e $\delta_j = 0$ indica que as filas estão vazias ou não existem nas filas *jobs* de tamanho menor ou igual ao número de processadores ociosos.

A seguir, é apresentado um exemplo do cálculo de LoC, com um total de $N = 96$ processadores, veja a Tabela 2.

Tabela 2. Exemplo do cálculo LoC.

t_j	δ_j and n_j
$t_1 = 0; t_2 = 10$	$\delta_1 = 1$ and $n_1 = 5$
$t_2 = 10; t_3 = 13$	$\delta_2 = 0$ and $n_2 = 3$
$t_3 = 13; t_4 = 17$	$\delta_3 = 1$ and $n_3 = 6$
$t_4 = 17; t_5 = 30$	$\delta_4 = 0$ and $n_4 = 4$
$t_5 = 30; t_q = 100$	$\delta_5 = 1$ and $n_5 = 8$

$$LoC = \frac{\begin{array}{c}5(10-0)1+3(13-10)0+6(17-13)1\\+4(30-17)0+8(100-30)1\end{array}}{96(100-0)} \times 100 \doteq 6,6\% \quad (5)$$

O resultado acima corresponde à fragmentação ocorrida no sistema no intervalo de tempo $t_q - t_1 = 100$ (6,6% de fragmentação no sistema).

7.4 Utilização

Em estudos de simulação, a taxa de utilização (U) dos *clusters* é simplesmente uma medida indireta do *makespan* [57] [60] e seu cálculo dado pela Equação 6:

$$U = \frac{\sum_{j=1}^{w} s_j \times rt(j)}{Makespan \times N} \quad (6)$$

em que U é a taxa de utilização dos *clusters*, N é o total de processadores no sistema, s_j representa o número de tarefas do *job* J_j e, consequentemente, como cada tarefa do *job* deve ser executada em um processador distinto simultaneamente, s_j também expressa o número de processadores necessários para executá-lo e o *Makespan* é a diferença entre o tempo inicial de execução do primeiro *job* e o tempo final do último *job*.

8. Simulação e Análise dos Resultados

8.1 Parâmetros de entrada

As simulações foram realizadas no sistema GSMSim, implementado em Java, o qual foi desenvolvido no laboratório do GrPeC da UFC, permitindo a simulação de entidades em sistemas de computação paralela e distribuída, como usuários, aplicações, gerenciadores de recursos e escalonadores.

O ambiente de simulação utilizado para o experimento consiste de *clusters* heterogêneos com 128 e 256 processadores, respectivamente, pertencendo ao mesmo domínio administrativo. O GD recebe as informações necessárias de cada *cluster*. A comunicação entre os processadores é de contenção livre. Além disso, a latência (Seção 1) de comunicação está inclusa no tempo de serviço do *job*.

O simulador recebe um carga de trabalho como entrada e, conforme a política de escalonamento atual, toma uma decisão para atender às demandas dos usuários. Para análise do ambiente, utilizamos diversos *traces*, extraídos de um ambiente distribuído real [45]. Além disso, foram realizados vários testes com ambientes heterogêneo e homogêneo, utilizando diferentes cargas de trabalho. Entretanto, a carga de trabalho utilizada para o experimento aqui apresentado é composta de 3000 *jobs*, num total de 140441 tarefas, as quais são descritas pela tupla (id_j, at_j, s_j, pt_j) (Seção 3.1).

Na carga de trabalho utilizada na simulação, os *jobs* apresentam características bem diferentes, como, por exemplo, 1.860 *jobs* requerem 32 processadores; 30, 90, 60, 60, 570 e 330 *jobs* requerem 1, 2, 4, 8, 64 e 128 processadores, respectivamente. Portanto, em média, 366 tarefas são atendidas por cada processador.

Para a simulação, foram propostos dois cenários:

- no primeiro cenário, S1, utilizamos no LD o algoritmo OLB, a fim de atribuir as tarefas para as filas de maneira aleatória para os processadores disponíveis;
- no segundo cenário, S2, utilizamos no LD o algoritmo JSEQ, com intuito de atribuir as tarefas para as filas, conforme o tempo de execução das tarefas nos processadores;

Vale ressaltar que os escalonadores de fila AFCFS, LJFS e LXF (sem migração) e AFCFSm, LJFSm e LXFm (com migração) foram aplicados nos dois cenários S1 e S2.

Para cada cenário, foram executadas dez simulações, a partir das quais se calcularam os valores médios dos tempos de espera, tempos de resposta, percentual de utilização dos *clusters* e LoC. Em cada algoritmo de escalonamento, citado anteriormente, usou-se um intervalo de confiança de 95% para o tempo médio de resposta.

Na próxima seção, apresentam-se os resultados das simulações executadas, usando-se as métricas descritas na Seção 7.Os resultados a seguir, descrevem o impacto sobre o desempenho do sistema acima mencionado, em relação à migração aplicada nos escalonadores de gangue AFCFS, LJFS e LXF. Além

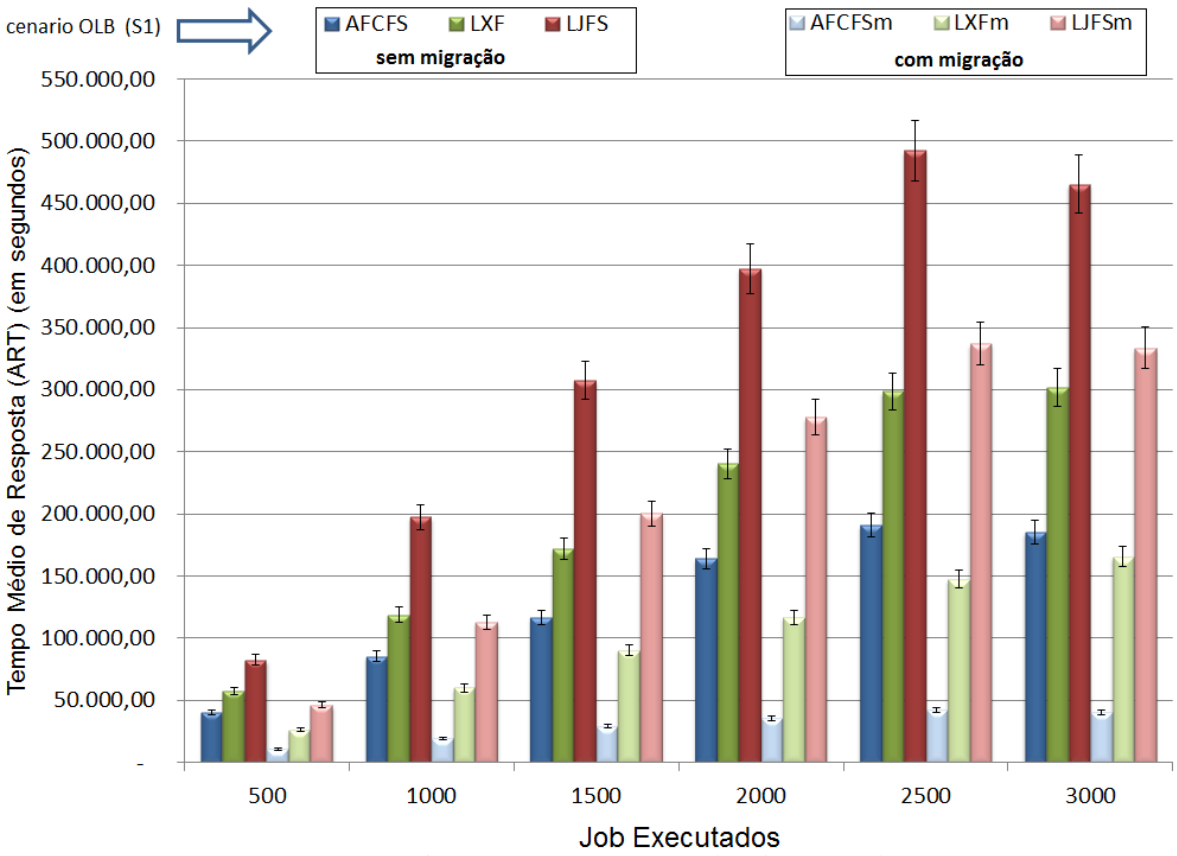

Figura 9. Cenário OLB - ART *versus* No. de *jobs* executados

disso, o impacto do OLB (Cenário S1) e do JSEQ (Cenário S2) no LD são examinados.

8.2 Tempo médio de resposta *versus* número de *jobs* executados

8.2.1 Cenário S1 - Uso do algoritmo OLB e Escalonadores de fila

Na Figura 9 (Cenário S1), é ilustrado o Tempo Médio de Resposta (ART), com o uso do algoritmo OLB e os escalonadores AFCFS, LXF e LJFS e AFCFSm, LXFm e LJFSm, respectivamente, onde o eixo *x* representa a quantidade de *jobs* executados. Neste cenário, o algoritmo AFCFS apresentou o menor *ART* em todas as faixas de *jobs* executadas em relação ao LXF e LJFS. A política LXF mostrou melhores resultados em relação à LJFS. Isso se justifica, pois a política LXF tende a favorecer *jobs* que apresentam maior fator de expansão XF (Seção 5.3), diferente do LJFS, que visa a beneficiar os *jobs* maiores, pois nem sempre o sistema disponibiliza processadores para atender os *jobs* menores, causando aumento no tempo de resposta destes.

Como podemos observar, Figura 9 e Tabela 3, os algoritmos AFCFSm, LXFm e LJFSm mostram uma redução bem significativa no *ART*, em relação aos mesmos sem migração. Isso mostra que o uso da migração causa um grande impacto no tempo de resposta. Portanto, o método sugerido conseguiu utilizar os processadores disponíveis de forma mais eficiente, reduzindo o tempo de resposta dos *jobs*. A política AFCFSm apresentou visivelmente o melhor resultado.

8.2.2 Cenário S2 - Uso do algoritmo JSEQ e Escalonadores de fila

Na Figura 10 (Cenário S2), é ilustrado o ART, o uso do algoritmo JSEQ e os escalonadores AFCFS, LXF e LJFS e AFCFSm, LXFm e LJFSm, respectivamente, onde o eixo *x* representa a quantidade de *jobs* executados. Neste cenário, o algoritmo AFCFS continuou apresentando o menor *ART* em todas as quantidades de *jobs* executadas, em relação ao LXF e LJFS. O LXF também mostrou resultados melhores em relação ao LJFS. Conforme o que foi dito na Seção 8.2.1, o LXF tende a favorecer *jobs* com maior fator de expansão.

Os algoritmos AFCFSm, LXFm e LJFSm no S2 (Tabela 4) confirmaram que a técnica de migração reduz o tempo de resposta, pois utiliza os processadores disponíveis de forma mais eficiente.

Comparando os resultados dos algoritmos AFCFS, LXF e LJFS nos cenários S1 e S2 (Tabelas 3 e 4), constata-se que no S2 diminui-se consideralmente o ART independente do escalonador de fila utilizado. Isso mostra que o algoritmo JSEQ distribui as tarefas nas filas de forma mais justa. A informação do valor total do tempo de processamento das tarefas, ou seja, o tempo de processamento das tarefas na fila mais a existência ou não da tarefa que está em execução no processador implicam a redução do tempo de espera das

Tabela 3. Cenário S1 - Uso do algoritmo OLB e escalonadores de fila

No. de *jobs* executados	AFCFS	AFCFSm	LXF	LXFm	LJFS	LJFSm
500	40.133,44	10.719,64	57.826,43	26.209,00	82.580,55	46.161,78
1000	85.287,65	19.498,69	118.982,20	59.897,42	197.405,05	113.076,48
1500	116.247,61	29.175,47	171.619,96	90.376,76	307.561,28	200.460,79
2000	164.252,48	35.410,80	240.306,39	116.669,94	397.360,80	277.986,15
2500	190.661,94	42.315,17	298.628,58	147.463,46	492.325,32	337.005,26
3000	185.395,03	40.170,85	301.549,44	165.735,51	465.193,56	333.426,95

Tabela 4. Cenário S2 - Uso do algoritmo JSEQ e escalonadores de fila

No. de *jobs* executados	AFCFS	AFCFSm	LXF	LXFm	LJFS	LJFSm
500	22.432,98	11.940,57	43.339,01	27.840,66	76.982,39	50.044,96
1000	61.298,36	20.463,54	92.728,93	62.541,50	170.497,66	119.391,59
1500	87.077,07	29.937,91	138.066,91	93.214,22	265.064,16	215.580,81
2000	120.630,94	36.003,33	193.861,07	117.629,26	355.929,28	284.934,41
2500	149.538,01	41.998,61	242.923,46	147.423,36	445.306,99	349.568,18
3000	143.846,61	40.030,39	245.386,95	162.278,49	429.514,51	340.029,61

tarefas e, consequentemente, o tempo de resposta. Por outro lado, os algoritmos com migração no cenário S1 (Figura 9) apresentam ARTs semelhantes aos do S2(Figura 10).

Baseado no exposto acima, os resultados dos algoritmos de escalonamento AFCFSm, LXFm e LJFSm nos dois cenários são satisfatórios, pois, apresentam diminuição do ART, através do uso dos mecanismos de migração propostos.

8.3 Perda de capacidade no sistema (LoC)

8.3.1 Cenário S1 - Uso do algoritmo OLB e escalonadores de fila e o Cenário S2 - Uso do algoritmo JSEQ e escalonadores de fila

Nas Figuras, 11 (Cenário S1) e 12 (Cenário S2), são apresentadas as perdas de capacidade (em percentual) no sistema. Comparando os gráficos dos dois cenários, os resultados das políticas de escalonamento AFCFS, LXF e LJFS e AFCFSm, LXFm e LJFSm mostram percentuais de LoC equivalentes. O uso do OLB ou JSEQ implementados no LD não influenciam nos resultados da métrica LoC.

Nos cenários S1 e S2, os algoritmos AFCFS, LXF e LJFS ocasionam aproximadamente em média LoC=50% no sistema. Analisando AFCFSm, LXFm e LJFSm, respectivamente: o AFCFSm apresenta percentuais de LoC=3,48% e LoC=3,39%, resultados inferiores em relação ao LXFm (LoC=15,63% e LoC=16,20%), e ao LJFSm (LoC=9,63% e LoC=6,89%). Isso implica que o AFCFSm escalona os *jobs* de forma mais eficaz, minimizando a fragmentação no sistema. O LXFm tende apresentar maior fragmentação no sistema em relação ao LJFSm. Os resultados confirmam que os algoritmos de escalonamento com migração minimizam a fragmentação no sistema.

8.4 Utilização dos *Clusters*

8.4.1 Cenário S1 - Uso do algoritmo OLB e escalonadores de fila e o Cenário S2 - Uso do algoritmo JSEQ e escalonadores de fila

Na Figura 13 (Cenário S1) e Figura 14 (Cenário S2), é ilustrada a porcentagem de utilização dos *clusters*, em relação ao número de interações. É considerada uma interação a chegada de um *job* ao GD e a finalização da execução de um *job*. Comparando o Cenário S1 com o Cenário S2, os algoritmos AFCFS, LXF e LJFS e AFCFSm, LXFm e LJFSm apresentam médias de utilização dos recursos semelhantes.

Nos algoritmos de escalonamento AFCFS, LXF e LJFS, Figura 13 e Figura 14, as médias de utilização dos *clusters* se mantêm constantes nos intervalos de $1000 - 4900$. O AFCFS apresenta os percentuais inferiores em relação aos escalonadores LXF e LJFS. Isso acontece devido a esse algoritmo favorecer o atendimento aos *jobs* menores, gerando um aumento de processadores ociosos. Já as políticas LXF e LJFS apresentam melhores resultados na utilização dos *clusters*.

Analisando AFCFSm, LXFm e LJFSm, Figura 13 e Figura 14, confirma-se que os algoritmos com a estratégia de migração são mais eficientes na utilização dos recursos.

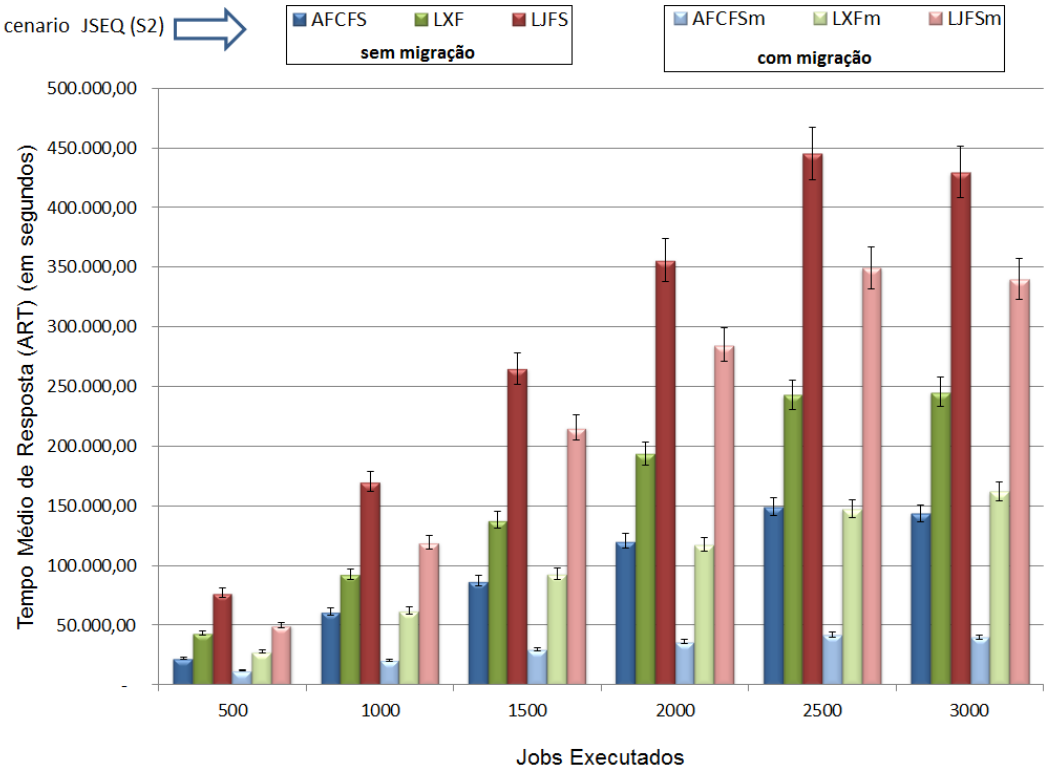

Figura 10. Cenário JSEQ - ART *versus* No. de *jobs* executados

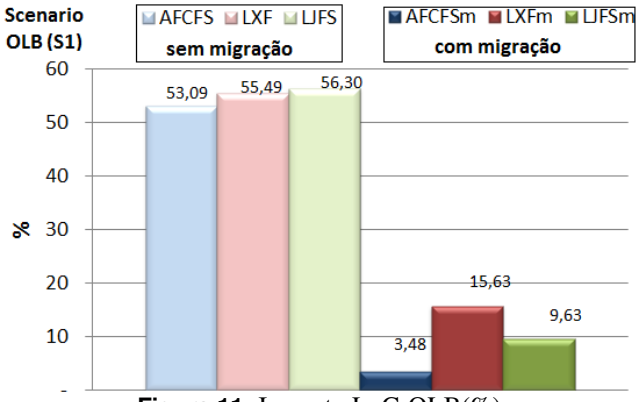

Figura 11. Impacto LoC-OLB(%)

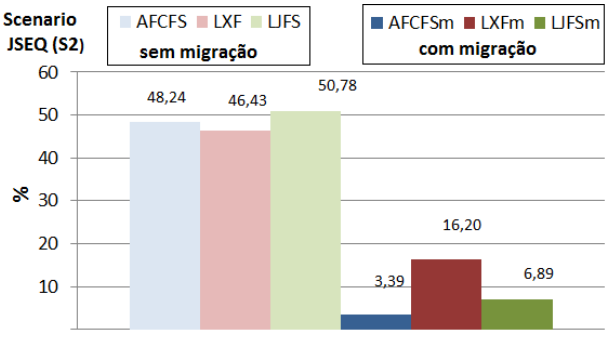

Figura 12. Impacto LoC-JSEQ(%)

9. Conclusões

Neste trabalho foi proposto e implementado um sistema de simulação GSMSim multicore *multicluster*, utilizando uma estrutura hierárquica de duas camadas, GD e LD. O GSMSim foi desenvolvido no intuito de analisar o desempenho dos escalonadores em situações diversas, bem como o comportamento do ambiente em diferentes contextos. No LD, foram implementados e adaptados dois cenários, OLB (S1) e JSEQ (S2), a fim de distribuir as tarefas de forma eficiente no sistema, que atuam antes dos escalonadores de fila. Além disso, utilizamos os escalonadores AFCFS, LXF e LJFS adaptados à técnica de gangue para o escalonamento das tarefas nas filas dos processadores. Como foi dito anteriormente, tais escalonadores ocasionam fragmentação no ambiente, portanto, implementamos mecanismos de migração para minimizar tal problemática. Para análise desses experimentos, utilizamos métricas de desempenho, com o objetivo de avaliar o comportamento dos escalonadores em situações diversas.

No Cenário S2, distribuíram-se as tarefas nas filas de forma mais eficiente, minimizando o tempo de espera das tarefas. Com isso, os escalonadores de fila AFCFS, LXF e LJFS mostraram resultados mais significativos em relação à métrica ART no cenário S1. Nesses cenários, AFCFSm, LXFm e LJFSm apresentaram ARTs satisfatórios. Isso implica que a técnica sugerida de migração conseguiu utilizar os processadores ociosos de forma mais eficiente, reduzindo, assim, a fragmentação no sistema.

Adaptada por nós para este contexto, a métrica LoC, mede o impacto que os escalonadores trazem no sistema em relação à fragmentação. Através dos resultados obtidos (Figura 11 e

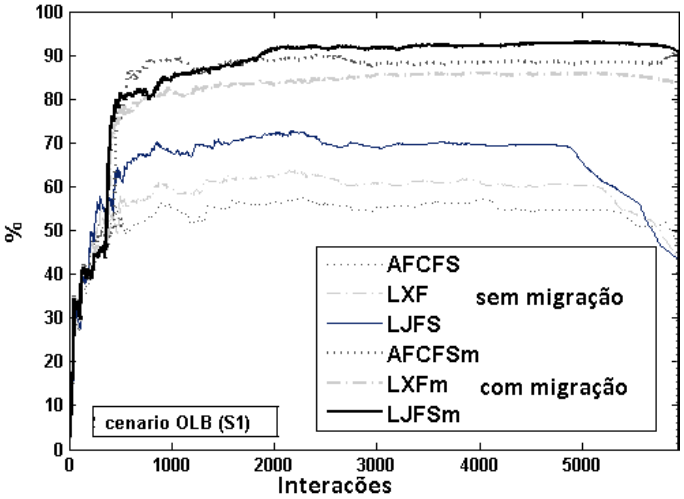

Figura 13. Cenário (S1)-Utilização(%)

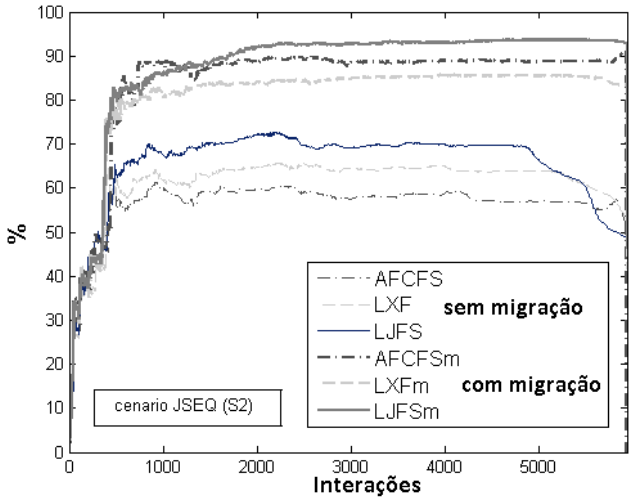

Figura 14. Cenário (S2)-Utilização(%)

Figura 12), os algoritmos AFCFS, LXF e LJFS ocasionam $\approx 50\%$ de fragmentação no sistema. Com os mecanismos de migração propostos, a fragmentação foi reduzida consideravelmente no AFCFSm (3,48% e 3,39%), LXFm (15,63% e 16,20%) e LJFSm (9,63% e 6,89%) (Figura 11 e Figura 12).

Com relação à métrica de utilização dos *clusters* (Figura 13 e Figura 14), confirmou-se que a técnica de migração diminui o número de processadores ociosos no sistema, bem como a fragmentação.

Os resultados mostraram que houve a redução da fragmentação através do uso da migração de tarefas entre as filas dos processadores num ambiente *multicluster* heterogêneo, bem como houve uma melhor utilização deles, implicando a redução dos custos operacionais, por parte dos provedores no atendimento das expectativas de Qualidade de Serviços (QoS) dos usuários. Vale salientar que o Cenário S2 apresentou, em todas as métricas, resultados satisfatórios, ao contrário do S1, que na métrica ART (AFCFS, LXF e LJFS) não foi tão eficiente. Além disso, o escalonador AFCFSm apresentou nos dois cenários os melhores resultados.

Este estudo apresenta algumas limitações, especificamente em relação as características dos jobs, pois tratamos apenas de aplicações do tipo meta-tarefa (*meta-task*), em que as tarefas são executadas simultânea e independentemente. Na literatura apresenta outros tipos aplicações, como, por exemplo, do tipo DAG (*Directed Acyclic Graph*), que existem restrições de precedência. Uma outra limitação deste estudo está relacionado a análise com outros *workloads*.

Apesar das limitações identificadas, e de outras que podem ser apontadas, considera-se que o estudo mostrou uma melhor utilização dos recursos em um ambiente de simulação *multicore* multicluster, atendendo as expectativas de QoS dos usuários.

Como trabalhos futuros, uma vasta pesquisa pode ser feita na área de escalonamento para ambientes de grades computacionais. Neste trabalho, utilizamos apenas os algoritmos OLB e JSEQ no LD, portanto, pretendemos aplicar outras heurísticas a fim de analisar o comportamento do sistema em diferentes abordagens. Além disso, poderíamos criar novas heurísticas de escalonamento para aplicações do tipo DAG. E uma outra perspectiva, são as técnicas de migração aplicadas nos escalonadores (AFCFS, LXF e LJFS) poderíamos implementar em outros escalonadores, como, por exemplo, nos algoritmos genéticos e comparando-a com os mesmos usados neste trabalho. Ademais, a validação do modelo proposto no contexto real, avaliando grande número de resultados experimentais.

10. Agradecimentos

Agradecemos a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) o apoio financeiro, o Programa de Pós-Graduação de Ciência da Computação - Universidade Estadual do Rio Grande do Norte/Universidade Federal Rural do Semi-Árido e a Secretaria da Educação do Estado do Ceará (SEDUC).

11. Contribuição dos Autores

Todos os autores participaram de forma efetiva para composição deste artigo, assumindo publicamente a responsabilidade pelo conteúdo.

References

[1] SALEHI, M. A. et al. Stochastic-based robust dynamic resource allocation for independent tasks in a heterogeneous computing system. *J Parallel Distrib Comput.*, v. 97, n. 1, p. 96–111, 2016.

[2] KANG, W.; KIM, J. Effective scheduling of grid resources using failure prediction. *Comput Inf.*, v. 35, n. 1, p. 369–390, 2016.

[3] ZAKARYA, M.; GILLAM, L. Energy efficient computing, clusters, grids and clouds: A taxonomy and survey. *SUSCOM*, v. 14, n. 1, p. 13–33, 2017.

[4] LUO, H. et al. A review of job scheduling for grid computing. *Application Res of Comput.*, v. 22, n. 5, p. 16–9, 2005.

[5] COOK, S. A. The complexity of theorem-proving procedures. In: HARRISON, M. A.; BANERJI, R. B.; ULLMAN, J. D. (Ed.). *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1971. (STOC, v. 1).

[6] ULLMAN, J. D. Np-complete scheduling problems. *J. Comput. Syst. Sci*, v. 10, n. 3, p. 384–393, 1975.

[7] TOPCUOGLU, H.; HARIRI, S.; WU, M. Performance-effective and low complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, v. 13, n. 3, p. 260–274, 2002.

[8] CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computingsystems. *IEEE Trans. Softw. Eng.*, v. 14, n. 2, p. 141–154, 1988.

[9] WIECZOREK, M.; HOHEISEL, A.; PRODAN, R. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Gener. Comput. Syst.*, v. 25, n. 3, p. 237–256, 2009.

[10] SMANCHAT, S.; VIRIYAPANT, K. Taxonomies of workflow scheduling problem and techniques in the cloud. *Future Gener. Comput. Syst.*, v. 52, n. 3, p. 1–12, 2015.

[11] LOPES, R. V.; MENASCÉ, D. A taxonomy of job scheduling on distributed computing systems. *IEEE Trans. Parallel Distrib. Syst.*, v. 27, n. 12, p. 3412–3428, 2016.

[12] XHAFA, F.; ABRAHAMB, A. Computational models and heuristic methods for grid scheduling problems. *Future Gener Comput Syst.*, v. 26, n. 4, p. 608–621, 2010.

[13] MA, T. et al. Grid task scheduling: Algorithm review. *IETE Technical*, v. 28, n. 2, p. 158–167, 2011.

[14] CHATURVEDI, A. K.; SAHU, R. New heuristic for scheduling of independent tasks in computational grid. *J Grid Distr Comput.*, v. 4, n. 3, p. 25–36, 2011.

[15] MISHRA, M. K. et al. A survey on scheduling heuristics in grid computing environment. *IJMECS*, v. 6, n. 10, p. 57–83, 2014.

[16] WANG, J.; ABU-GHAZALEH, N.; PONOMAREV, D. Controlled contention: Balancing contention and reservation in multicore application scheduling. In: IEEE. *2015 IEEE International Parallel and Distributed Processing Symposium*. Hyderabad, India: IEEE, 2015. v. 1, n. 1, p. 946–955.

[17] FEITELSON, D. G. Resampling with feedback – a new paradigm of using workload data. In: *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*. New York, NY, USA: Springer-Verlag New York, Inc., 2016. v. 9833.

[18] GÓMEZ-MARTÍN, C.; VEGA-RODRÍGUEZ, M. A.; GONZÁLEZ-SÁNCHEZ, J.-L. Fattened backfilling: An improved strategy for job scheduling in parallel systems. *Journal of Parallel and Distributed Computing*, v. 97, n. 1, p. 69–77, 2016.

[19] OUSTERHOUT, J. K. et al. Scheduling techniques for concurrent systems. v. 82, n. 1, p. 22–30, 1982.

[20] LANGER, T.; OSINSKI, L.; MOTTOK, J. A survey of parallel hard-real time scheduling on task models and scheduling approaches. *ARCS 2017*, v. 1, n. 1, p. 1–8, 2017.

[21] FEITELSON, D. G.; RUDOLPH, L.; SCHWIEGELSHOHN, U. Parallel job scheduling - a status report. In: FEITELSON, D. G.; RUDOLPH, L.; SCHWIEGELSHOHN, U. (Ed.). *Job Scheduling Strategies for Parallel Processing - 10th International Workshop, JSSPP 20*. Berlin, Germany: Springer, 2004. (LNCS, 1).

[22] WANG, X. et al. Multi-cluster load balancing based on process migration. In: XU, M. et al. (Ed.). *Proceedings of the 7th International Conference on Advanced Parallel Processing Technologies*. Berlin, Heidelberg: Springer-Verlag, 2007. v. 1.

[23] KARATZA, H. D. Performance of gang scheduling strategies in a parallel system. *Simul Model Pract Theory*, v. 17, n. 2, p. 430–441, 2009.

[24] PAPAZACHOS, Z.; KARATZA, H. D. Performance evaluation of bag of gangs scheduling in a heterogeneous distributed system. *J. Syst. Softw.*, v. 83, n. 8, p. 1346–1354, 2010.

[25] PAPAZACHOS, Z. C.; KARATZA, H. D. Gang scheduling in multi-core clusters implementing migrations. *Future Gener. Comput. Syst.*, v. 27, n. 8, p. 1153–1165, 2011.

[26] MOSCHAKIS, I. A.; KARATZA, H. D. Evaluation of gang scheduling performance and cost in a cloud computing system. *J. Supercomput.*, Kluwer Academic Publishers, v. 59, n. 2, p. 975–992, 2012.

[27] HAO, Y.; WANG, L.; ZHENG, M. An adaptive algorithm for scheduling parallel jobs in meteorological cloud. *Know.-Based Syst.*, v. 98, n. 3, p. 226–240, 2016.

[28] TOMAS, L.; CAMINERO, B.; CARRION, C. Improving grid resource usage: Metrics for measuring fragmentation. In: CCGRID '12. *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computings*. Washington, DC, USA: IEEE Computer Society, 2012. v. 1, n. 1.

[29] GRUDENIC, I.; BOGUNOVIC, N. Analysis of scheduling algorithms for computer clusters. *MIPRO 2008*, v. 1, n. 1, p. 1–6, 2008.

[30] MANICKAM, V.; ARAVIND, A. A fair and efficient gang scheduling algorithm for multicore processors. In: *Wireless Networks and Computational Intelligence*. Berlin, Germany: Springer, 2012. (CCIS, 1), p. 467–476.

[31] LIU, X. et al. Scheduling parallel jobs using migration and consolidation in the cloud. *Math Probl Eng*, v. 2012, n. 1, p. 1, 2012.

[32] PAPAZACHOS, Z. C.; KARATZA, H. D. Gang scheduling in a two-cluster system implementing migrations and periodic feedback. *Simul-T Soc Mod Sim*, v. 87, n. 12, p. 1021–1031, 2011.

[33] PINTO, F. A. P. et al. Analysis of scheduling algorithms with migration strategies in distributed systems. *ICNS 2014*, v. 1, n. 1, p. 1–6, 2014.

[34] PINTO, F. A. P. et al. Algorithms scheduling with migration strategies for reducing fragmentation in distributed systems. *IEEE LATIN AMERICA TRANSACTIONS*, v. 13, n. 3, p. 762–768, 2015.

[35] BUYYA, R.; ABRAMSON, D.; NIMROD, J. G. An architecture for a resource management and scheduling system in a global computational grid. *HPCASIA '05*, v. 1, n. 1, p. 283, 2000.

[36] ABDELGADIR, A. T.; PATHAN, A.-S. K.; AHMED, M. On the performance of mpi-openmp on a 12 nodes multi-core cluster. In: XIANG, Y. et al. (Ed.). *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2011. (ICA3PP'11, v. 2), p. 225–234.

[37] HAMID, R. J. W. N.; WILLS, G. B. An architecture for measuring network performance in multi-core multi-cluster architecture (MCMCA). *IJCTE*, v. 7, n. 1, p. 57–61, 2015.

[38] MAHESWARAN, M. et al. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In: *Proceedings of the Eighth Heterogeneous Computing Workshop*. Washington, DC, USA: IEEE Computer Society, 1999. (HCW '99, 1).

[39] MARSAN, M. A.; BALBO, G.; CONTE, G. *Performance Models of Multiprocessor Systems*. 1. ed. Cambridge, MA, USA: MIT Press, 1987. v. 1.

[40] GARG, S. K. et al. Double auction-inspired meta-scheduling of parallel applications on global grids. *J. Parallel Distrib. Comput.*, v. 73, n. 4, p. 450–464, 2013.

[41] HOCKNEY, R. The communication challenge for mpp: Intel paragon and meiko cs-2. *HPCN*, v. 20, n. 3, p. 931–936, 1994.

[42] FUJIWARA, K.; CASANOVA, H. Speed and accuracy of network simulation in the simgrid framework. In: GLYNN, P. (Ed.). *Proceedings of the 2Nd International Conference on Performance Evaluation Methodologies and Tools*. Brussels, Belgium: ICST, 2007. v. 1, n. 12.

[43] CASANOVA, H. et al. Versatile, scalable, and accurate simulation of distributed applications and platforms. *J Parallel Distrib Comput.*, v. 74, n. 10, p. 2899–2917, 2014.

[44] FEITELSON, D. G. *Workload Modeling for Computer Systems Performance Evaluation*. 1. ed. New York, NY, USA: Cambridge University Press, 2015. v. 1.

[45] FEITELSON, D. G. *Job Scheduling in Multiprogrammed Parallel Systems*. Yorktow, USA, 1997.

[46] FEITELSON, D. *The Standard Workload Format*. 2017. Disponível em: ⟨http://www.cs.huji.ac.il/labs/parallel/workload/swf.html⟩.

[47] BRAUN, T. et al. A comparison of eleven static heuristics for mapping a class of independenttasks onto heterogeneous distributed computing systems. *J. Parallel Distributed Computer*, v. 61, n. 6, p. 870–837, 2001.

[48] LIN, H.-C.; RAGHAVENDRA, C. S. An analysis of the join the shortest queue (jsq) policy. *ICDCS '03*, v. 1, n. 1, p. 362–366, 1992.

[49] KARATZA, H. D. A simulation-based performance analysis of gang scheduling in a distributed system. In: *IEEE, Simulation Symposium, 1999. Proceedings. 32nd Annual*. Washington, DC, USA: IEEE Computer Society, 1999. v. 1, n. 1.

[50] VASUPONGAYYA, S.; PRASITSUPPAROTE, A. Extending goal-oriented parallel computer job scheduling policies to heterogeneous systems. *J Supercomput.*, v. 65, n. 3, p. 1223–1241, 2013.

[51] FEITELSON, D. *A Survey of Scheduling in Multiprogrammed Parallel Systems*. New York, USA: IBM T.J. Watson Research Center, 1994. v. 1. 127 p.

[52] GONZALEZ, J. C. *Coordinated Scheduling and Dynamic Performance Analysis in Multiprocessors Systems*. Tese (Doutorado) — Universitat Politecnica de Catalunya, Barcelona, Espanha, 2002.

[53] FEITELSON, D. G. Packing schemes for gang scheduling. *JSSPP*, v. 1162, n. 1, p. 89–110, 1996.

[54] ZHOU, B. B. et al. An efficient resource allocation scheme for gang scheduling. *JSSPP*, v. 1911, n. 1, p. 74–86, 2000.

[55] SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Boston, USA: Addison-Wesley Professional, 2017. v. 1.

[56] ZHANG, Y.; FRANKE, H.; MOREIRA. The impact of migration on parallel job scheduling for distributed systems. In: BODE, A.; LUDWIG, T.; WISMüLLER, W. K. andRoland (Ed.). *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2000. (Lecture Notes in Computer Science, v. 1990).

[57] LEUNG, V. J.; SABIN, G.; SADAYAPPAN, P. Parallel job scheduling policies to improve fairness: A case study. *ICPP Workshops - IEEE*, v. 1, n. 1, p. 346–353, 2010.

[58] TANG, W. et al. Reducing fragmentation on torus-connected supercomputers. In: *2011 IEEE International Parallel Distributed Processing Symposium*. Washington, USA: IEEE Computer Society, 2011. v. 1.

[59] TANG, W. et al. Adaptive metric-aware job scheduling for production supercomputers. In: FENG, W. chun;

PARASHAR, M. (Ed.). *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. Pittsburgh, USA: IEEE, 2012. v. 1.

[60] BURKIMSHER, A.; BATE, I.; INDRUSIAK, L. S. A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times. *Future Gener Comput Syst.*, v. 29, n. 8, p. 2009–2025, 2012.