UNIVERSIDADE DO ESTADO DA BAHIA

DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

JOÃO VICTOR ALVES BARRETO

UTILIZAÇÃO DE SERVIDORES MESTRE-ESCRAVOS EM ESTRATÉGIA SEMI-PARTICIONADA DE ESCALONAMENTO

SALVADOR

2021

JOÃO VICTOR ALVES BARRETO

UTILIZAÇÃO DE SERVIDORES MESTRE-ESCRAVOS EM ESTRATÉGIA SEMI-PARTICIONADA DE ESCALONAMENTO

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação. Área de Concentração: Ciência da Computação

Orientador: Ernesto de Souza Massa Neto

SALVADOR

2021

FICHA CATALOGRÁFICA
Sistema de Bibliotecas da UNEB

B273u Barreto, João Victor Alves

Utilização de servidores mestre-escravos em estratégia semi-particionada de escalonamento / João Victor Alves Barreto. - Salvador, 2021.
77 fls : il.

Orientador(a): Ernesto de Souza Massa Neto.
Inclui Referências
TCC (Graduação - Sistemas de Informação) - Universidade do Estado da Bahia. Departamento de Ciências Exatas e da Terra. Campus I. 2021.

1.Sistemas de Tempo Real. 2.Escalonamento. 3.Multiprocessadores.

CDD: 003

JOÃO VICTOR ALVES BARRETO

UTILIZAÇÃO DE SERVIDORES MESTRE-ESCRAVOS EM ESTRATÉGIA SEMI-PARTICIONADA DE ESCALONAMENTO

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação. Área de Concentração: Ciência da Computação

Aprovada em:

BANCA EXAMINADORA

Ernesto de Souza Massa Neto (Orientador)
Universidade do Estado da Bahia – UNEB

Profa. Dra. Flávia Maristela Santos Nascimento
Instituto Federal da Bahia

Prof. Dr. Cláudio Alves de Amorim
Universidade do Estado da Bahia – UNEB

RESUMO

O escalonamento de tarefas de tempo real é uma área com soluções bem consolidadas em um ambiente com um processador. Contudo, a adição de mais processadores ao sistema traz outros desafios. Os algoritmos que lidam com multiprocessadores podem usar diversas abordagens, com o objetivo de escalonar as tarefas de forma eficiente, para que elas cumpram com seus prazos. A abordagem semi-particionada foi criada com o intuito de combinar os benefícios das estratégias global e particionada, porém, ela tem como ponto negativo a perda da capacidade computacional devido ao processo de sincronização das tarefas. Os servidores mestres-escravo apresentados no *quasi-partitioning scheduling*, solucionam esse problema de sincronização sem esse tipo de perda. Neste trabalho, os servidores mestres-escravo foram adicionados ao algoritmo semi-particionado *Notional Processors*, com a finalidade de viabilizar a adoção de estratégias semi-particionadas sem perdas de capacidade computacional. Os resultados obtidos através de simulação mostraram que essa adição trouxe melhorias à implementação original. Entre os benefícios estão a transformação do *Notional Processors* em um escalonador ótimo, com relação à utilização dos processadores, e uma melhor performance que o algoritmo original, avaliada através do número de preempções e migrações, principalmente quando toda capacidade do sistema é exigida.

Palavras-chave: tempo real. escalonamento. multiprocessadores. semi-particionado

ABSTRACT

Real-time task scheduling is an area with well-known solutions for uniprocessor environment. However, by adding more processors to the system brings other challenges. Algorithms that deal with multiprocessors can use several approaches, in order to efficiently schedule tasks so that they meet their deadlines. The semi-partitioned approach was created to combine the benefits of global and partitioned strategies, however, it has as negative point the loss of computational capacity due to the task synchronization. Master-slave servers presented in quasi-partitioning scheduling, solve this synchronization issue without this kind of loss. In this work, the master-slave servers were used in the semi-partitioned algorithm Notional Processors, to make possible the adoption of semi-partitioned strategies without loss of computational capacity. The results obtained through simulation showed that this addition brought improvements to the original implementation. Among the benefits are the transformation of Notional Processors into an optimal scheduler, in terms of processor utilization, and better performance than the original algorithm, evaluated through the number of preemptions and migrations, especially when all system capacity is required.

Keywords: real-time. scheduling. multiprocessor. semi-partitioned

LISTA DE FIGURAS

Figura 1 – Sistema de tempo real. 16
Figura 2 – Ilustração de um *notional processor* 27
Figura 3 – Ilustração do escalonamento usando o *Quasi-Partitioned Scheduling* (QPS) . 31
Figura 4 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em quatro processadores 32
Figura 5 – Primeira rodada de atribuições com o *First-Fit Heavy-First* (FFHF) 34
Figura 6 – Criação dos servidores mestre-escravos após atribuições 34
Figura 7 – Criação dos processadores lógicos a partir dos servidores mestres 35
Figura 8 – Atribuição das tarefas aos processadores lógicos 36
Figura 9 – Criação de servidores mestre-escravos para o processador lógico de 2º nível 36
Figura 10 – Criação do processador lógico de 2º nível 37
Figura 11 – Atribuição ao processador lógico de 2º nível 38
Figura 12 – Variáveis de customização do gerador 46
Figura 13 – Migrações por job (média) em quinze processadores 48
Figura 14 – Migrações e preempções por job (média) em quinze processadores 48
Figura 15 – Migrações por job (média) em quatro processadores 49
Figura 16 – Migrações e preempções por job (média) em quatro processadores 50
Figura 17 – Migrações e preempções por job (média) em cinco processadores 50
Figura 18 – Migrações e preempções por job (média) em oito processadores 51
Figura 19 – Migrações e preempções por job (média) em dez processadores 51
Figura 20 – Migrações por job (média) em oito processadores, para utilizações maiores . 52
Figura 21 – Migrações por job (média) em nove processadores, para utilizações maiores 53
Figura 22 – Migrações e preempções por job (média) em dez processadores, para utilizações maiores 53
Figura 23 – Migrações por *job* (média) em quatro processadores 59
Figura 24 – Migrações e preempções por *job* (média) em quatro processadores 60
Figura 25 – Migrações por *job* (média) em quatro processadores e utilizações maiores . 60
Figura 26 – Migrações e preempções por *job* (média) em quatro processadores e utilizações maiores 61

Figura 27 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em quatro processadores 61
Figura 28 – Migrações por *job* (média) em cinco processadores 62
Figura 29 – Migrações e preempções por *job* (média) em cinco processadores 62
Figura 30 – Migrações por *job* (média) em cinco processadores e utilizações maiores . . 63
Figura 31 – Migrações e preempções por *job* (média) em cinco processadores e utilizações maiores 63
Figura 32 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em cinco processadores 64
Figura 33 – Migrações por *job* (média) em seis processadores 64
Figura 34 – Migrações e preempções por *job* (média) em seis processadores 65
Figura 35 – Migrações por *job* (média) em seis processadores e utilizações maiores . . . 65
Figura 36 – Migrações e preempções por *job* (média) em seis processadores e utilizações maiores 66
Figura 37 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em seis processadores 66
Figura 38 – Migrações por *job* (média) em sete processadores 67
Figura 39 – Migrações e preempções por *job* (média) em sete processadores 67
Figura 40 – Migrações por *job* (média) em sete processadores e utilizações maiores . . . 68
Figura 41 – Migrações e preempções por *job* (média) em sete processadores e utilizações maiores 68
Figura 42 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em sete processadores 69
Figura 43 – Migrações por *job* (média) em oito processadores 69
Figura 44 – Migrações e preempções por *job* (média) em oito processadores 70
Figura 45 – Migrações por *job* (média) em oito processadores e utilizações maiores . . . 70
Figura 46 – Migrações e preempções por *job* (média) em oito processadores e utilizações maiores 71
Figura 47 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em oito processadores 71
Figura 48 – Migrações por *job* (média) em nove processadores 72
Figura 49 – Migrações e preempções por *job* (média) em nove processadores 72

Figura 50 – Migrações por *job* (média) em nove processadores e utilizações maiores . . 73
Figura 51 – Migrações e preempções por *job* (média) em nove processadores e utilizações maiores 73
Figura 52 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em nove processadores 74
Figura 53 – Migrações por *job* (média) em dez processadores 74
Figura 54 – Migrações e preempções por *job* (média) em dez processadores 75
Figura 55 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em dez processadores 76
Figura 56 – Migrações por job (média) em quinze processadores 77
Figura 57 – Migrações e preempções por job (média) em quinze processadores 77

LISTA DE ABREVIATURAS E SIGLAS

BF	*Best-Fit*
EDDP	*Earliest Deadline Deferrable Portion*
EDF	*Earliest Deadline First*
EKG	*EDF with task splitting and k processors in a group*
FF	*First-Fit*
FFHF	*First-Fit Heavy-First*
NF	*Next-Fit*
NPwMS	*Notional Processors with Master and Slave*
QPS	*Quasi-Partitioned Scheduling*
RM	*Rate Monotonic*
S-EKG	*Sporadic EKG*
WCET	*worst-case execution time*
WF	*Worst-Fit*

LISTA DE SÍMBOLOS

τ	Tarefa
D	Deadline
C	Tempo estimado de execução no pior caso (*Worst-case execution time*)
T	Intervalo mínima entre chegadas dos trabalhos da tarefa
U	Utilização do sistema
m	Quantidade de processadores
S	Duração da reserva de tempo
σ	Servidor

SUMÁRIO

1 INTRODUÇÃO 13
2 SISTEMAS DE TEMPO REAL 16
2.1 Tarefas 17
2.2 Criticidade do sistema 18
2.3 Escalonamento 19
3 ALGORITMO NOTIONAL PROCESSORS 23
3.1 Estratégias em um Processador 23
3.2 Estratégias Semi-Particionadas de Escalonamento 24
3.3 Notional Processors 26
3.4 Ociosidade Típica das Estratégias Semi-Particionadas 28
4 SERVIDORES MESTRE-ESCRAVOS 29
4.1 Sincronização de Tarefas com Migração 29
4.2 QPS 29
4.3 Servidores de Taxa-Fixa 30
4.4 Mecanismo dos Servidores Mestre-Escravo 31
5 SERVIDORES MESTRE-ESCRAVOS NO NOTIONAL PROCESSORS 32
5.1 Criação dos processadores lógicos 33
5.2 Atribuição dos servidores mestre-escravos 38
5.3 Escalonamento 39
6 SIMULAÇÃO 40
6.1 Simulação de eventos discretos 40
6.2 Simulação do escalonamento 41
7 TESTES E RESULTADOS 44
7.1 Métricas 44
7.2 Conjuntos de tarefas 45
7.3 Resultados 46
8 CONSIDERAÇÕES FINAIS 54
REFERÊNCIAS 56

APÊNDICES **58**
APÊNDICE A – Gráficos dos resultados **59**

1 INTRODUÇÃO

Os sistemas de tempo real estão presentes em diversas áreas do nosso cotidiano, como por exemplo: automotiva, aviação, telecomunicações, sistemas espaciais, imagiologia médica e produtos eletrônicos de consumo. Diferente dos sistemas computacionais convencionais, os de tempo real não dependem apenas dos resultados lógicos, mas também do instante físico em que os resultados são produzidos (KOPETZ, 1997).

A produção do resultado de um sistema computacional de Tempo Real está atrelada a um conjunto de tarefas o qual produz possíveis infinitos trabalhos (*jobs*), que precisam ser executados de forma recorrente dentro de um certo prazo. Portanto, é necessário escaloná-los de forma que eles possam cumprir seu requisitos temporais. O escalonador leva em conta as especificidades de cada trabalho para estabelecer a prioridade dele.

Davis e Burns (2011) defendem que a teoria de escalonamento em tempo real em um único processador pode ser vista como razoavelmente madura, com um grande número de resultados-chave documentados em livros didáticos e transferidos com sucesso para a prática industrial. Porém, para um ambiente contendo múltiplos processadores, este é um problema com mais desafios, sendo objeto de estudo até hoje. Os algoritmos apresentados nas pesquisas com a finalidade de resolver o problema em um ambiente com vários processadores, podem utilizar abordagens distintas. Andersson (2019) diz que a literatura de pesquisa, tradicionalmente, oferece duas categorias de escalonadores: o escalonamento particionado e o escalonamento global. No escalonamento particionado cada tarefa é atribuída a um processador, enquanto no escalonamento global as tarefas que estiverem prontas para execução são armazenadas numa fila compartilhada entre os processadores. Sendo assim, a migração das tarefas é permitida no esquema global, ao contrário do particionado que as tarefas não podem migrar.

As duas estratégias possuem prós e contras (os quais serão detalhados mais adiante neste trabalho), e com o intuito de combinar o melhor das duas, surgiu a estratégia semi-particionada. Essa estratégia atribui as tarefas aos processadores, como na particionada, porém, permite que algumas tarefas sejam divididas/migrem entre os processadores (ANDERSSON,

2019). Ainda assim, as implementações semi-particionadas trazem ociosidade aos processadores, devido ao cuidado existente de sincronização para que as partes da tarefa não sejam executadas simultaneamente como demonstrado por Andersson e Tovar (2006).

A estratégia de escalonamento denominada *Quasi-Partitioned Scheduling* (QPS) (MASSA et al., 2014), apresentou uma solução para este problema de sincronização através da utilização de pares de servidores, denominados mestre-escravos, sem ocasionar perda da capacidade computacional dos processadores. A premissa deste trabalho, está pautada na adição desses servidores a um algoritmo semi-particionado, a fim de otimizar a sincronização desse escalonador.

O algoritmo escolhido foi o *Notional Processors* (BLETSAS; ANDERSSON, 2009) por utilizar a estratégia semi-particionada, ter relevância na literatura e ser de fácil compreensão. Além disso, assim como os outros escalonadores dessa categoria, a forma como ele lida com a sincronização das tarefas traz a desvantagem de gerar ociosidade aos processadores. Considerando que a utilização desse mecanismo resulta em ociosidade, e que os servidores mestre-escravos fazem o trabalho de sincronização sem gerar perdas de capacidade computacional, a substituição do mecanismo original pelos servidores mestre-escravos é uma alternativa para viabilizar a adoção de estratégias semi-particionadas (como o *Notional Processors*) sem a existência de perdas de capacidade computacional.

Portanto, este trabalho avalia a possibilidade de utilização de servidores mestre-escravo para resolver o problema de sincronização de tarefas em estratégias semi-particionadas, reduzindo a ociosidade característica dos algoritmos que utilizam esta estratégia. Para tal, foi necessário construir um simulador, que baseia-se na simulação de eventos discretos, com o objetivo de observar o comportamento do *Notional Processors* modificado. Também é feita uma avaliação dos resultados obtidos através do escalonamento simulado, com uma análise da sua performance a partir das métricas estabelecida. Bem como comparações com os algoritmos *Notional Processors* original e o QPS, o qual é a implementação de onde surgiram os servidores mestre-escravos.

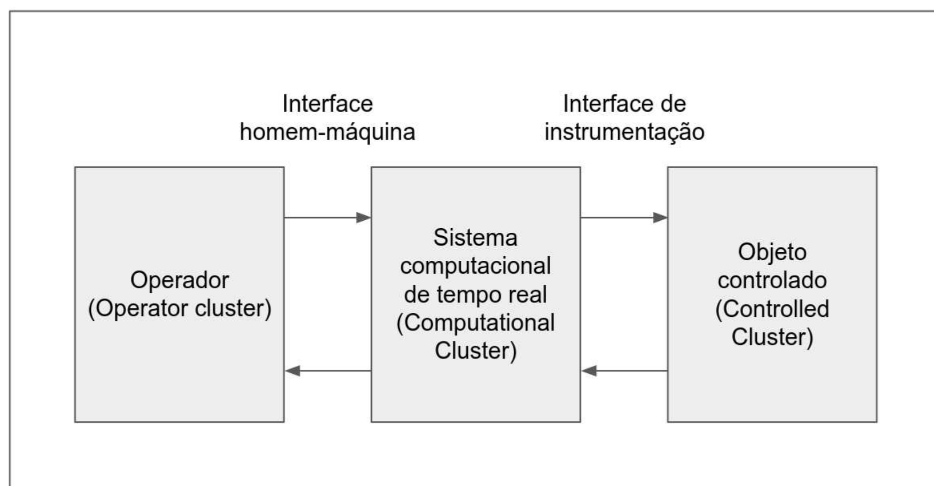
O capítulo 2 contém a definição de um sistema de tempo real, abordando conceitos essenciais para a compreensão dele e o escalonamento de tarefas de tempo real. Nos capítulos 3 e 4 são apresentados os trabalhos que serviram de motivação para este, trazendo algoritmos como

o *Notional Processors* e outros importantes como o QPS, que apresentou os servidores mestre-escravos. A solução proposta por este trabalho está descrita no capítulo 5, a qual é resultante da adição dos servidores mestre-escravos ao *Notional Processors*. O percurso metodológico do trabalho pode ser visto no capítulo 6, com os testes e resultados no capítulo 7. Por fim, as considerações finais sobre o trabalho no capítulo 8, que também expõe as possibilidades de trabalhos futuros.

2 SISTEMAS DE TEMPO REAL

Segundo Kopetz (1997), um sistema computacional de tempo real é sempre parte de um sistema maior. Um sistema de tempo real muda seu estado em função do tempo físico. Por exemplo, uma reação química continua a mudar de estado mesmo depois de seu sistema computacional de controle tenha parado. Kopetz (1997) também aborda a decomposição do sistema de tempo real em um conjunto de subsistemas chamado de *clusters*, sendo eles: o objeto controlado (*the controlled cluster*), o sistema computacional de tempo real (*the computational cluster*) e o operador humano (*the operator cluster*). O objeto controlado e o operador são as entidades que compõem o ambiente no qual o sistema computacional opera.

Figura 1 – Sistema de tempo real.



Fonte – Retirada de Kopetz (1997)

Dentro desso modelo, Kopetz (1997) denomina a interface entre o operador humano e o sistema computacional como interface homem-máquina, a qual é composta por dispositivos de entrada e saída. Já a interface entre o objeto controlado e o sistema computacional é denominada de interface de instrumentação, sendo composta por sensores e atuadores os quais traduzem os sinais físicos para uma forma digital e *vice versa*. A Figura 1 representa esses *clusters*, bem como as relações que existem entre eles.

2.1 TAREFAS

Tarefas ou processos formam as unidades de processamento sequencial que concorrem sobre um ou mais recursos computacionais de um sistema (FARINES; FRAGA; OLIVEIRA, 2000). Um sistema de tempo real é constituído por um conjunto de tarefas que, além de apresentar uma correção lógica devem apresentar também uma correção temporal. Davis e Burns (2011) dizem que a grande maioria das pesquisas em escalonamento multiprocessador em tempo real concentra-se em dois modelos de tarefa simples: o modelo de tarefa periódica e o modelo de tarefa esporádica. Em ambos os modelos, as tarefas dão origem a uma sequência potencialmente infinita de trabalhos, ou *jobs*. Para um modelo de tarefa periódica, os *jobs* da tarefa, chegam periodicamente, separados por um intervalo fixo. No modelo de tarefa esporádica, os tempos de chegada dos trabalhos são desconhecidos, porém existe um intervalo de tempo mínimo entre eles. Se os trabalhos chegam em tempos indeterminados e não existe o intervalo de tempo mínimo, é um modelo de tarefa aperiódica.

Cada tarefa possui um conjunto de características que são importantes para as tomadas de decisões do escalonador. Para Davis e Burns (2011), em um modelo de tarefa periódica, uma tarefa τ_i é caracterizada por:

- D_i o *deadline* relativo, ou seja, o prazo que cada *job* de τ_i tem para terminar sua execução a partir da sua chegada, ou liberação.
- C_i o *worst-case execution time* (WCET). O tempo estimado, no pior caso, que o *job* de τ_i será executado.
- T_i o intervalo mínimo entre chegadas dos *jobs* de τ_i.

O *job* de uma tarefa é representado por Massa et al. (2014) através da notação $J : (c, r, d)$, sendo definido como uma instância de τ, que consome até c unidades do tempo de processamento dentro do intervalo $[r, d)$. Onde r é o *release time* (o tempo que o *job* é liberado e passa a disputar o processador com os outros trabalhos) e d é o *deadline* do trabalho. O tempo de chegada (*arrival*) do primeiro *job* de uma tarefa é também uma informação importante para o escalonamento.

A utilização da capacidade computacional por uma tarefa (u_i) é dada por C_i/T_i. Já a utilização do sistema é dada pelo somatório da utilização individual das tarefas. É importante que o somatório das utilizações dessas tarefas não ultrapasse a capacidade do sistema, pois algum *job*

ficaria com a execução comprometida, ocasionando a perda do seu *deadline*. Além disso, alguns algoritmos possuem limites de utilização específicos, como mencionado por Andersson (2019) sobre os algoritmos particionados. No artigo, é abordada a limitação do algoritmo particionado, mostrando que existem conjuntos de tarefas, os quais utilizam pouco mais de 50% de toda a capacidade de processamento que não cumprem o prazo.

Quanto ao prazo das tarefas, existem três níveis de restrição nos seus *deadlines* que são estudados na literatura: *deadlines* implícitos (*implicit deadlines*), *deadlines* limitados (*constrained deadlines*) e *deadlines* arbitrários (*arbitrary deadlines*) (DAVIS; BURNS, 2011). Para *deadlines* implícitos, todas as tarefas possuem *deadline* igual ao período ($D_i = T_i$). *Deadlines* limitados consideram que os *deadlines* são menores ou iguais ao período ($D_i \leq T_i$). Já os *deadlines* arbitrários, consideram que os *deadlines* podem ser menores, iguais ou maiores do que o período da tarefa. O *deadline* também pode ser expressado como relativo ou absoluto. O relativo foi mencionado no modelo de tarefa periódica descrito acima, o qual considera estritamente o intervalo que tem início a partir da chegada ou liberação do *job*. Já o *deadline* absoluto leva em consideração além do tempo de chegada ou liberação, o tempo global que se encontra a aplicação. Isso quer dizer que se uma tarefa τ_i possui *deadline* relativo D_i, seu *deadline* absoluto é dado por $D_i + t$, onde t é o tempo marcado pelo relógio no momento de chegada ou liberação do *job* dessa tarefa.

2.2 CRITICIDADE DO SISTEMA

Os sistemas de tempo real podem estar presentes em diversas aplicações, desde as simples até as mais complexas. A depender de onde o sistema é empregado ele pode exigir um nível maior de criticidade, sendo altamente crítico, ou menor sendo mais tolerante. Se uma tarefa é crítica (tarefas *hard*), ao ser completada depois do seu *deadline*, ela pode causar falhas catastróficas no sistema e em seu ambiente (FARINES; FRAGA; OLIVEIRA, 2000). Se o sistema possui em seu conjunto pelo menos uma tarefa crítica, ele é considerado crítico (*hard real-time*). Tarefas brandas ou não críticas, quando se completam depois de seus *deadlines*, no máximo implicam numa diminuição de desempenho do sistema (FARINES; FRAGA; OLIVEIRA, 2000). O conjunto de tarefas dos sistemas não críticos (*soft real-time*) são compostos por tarefas não críticas.

Em uma aplicação multimídia de áudio e de vídeo, caso ocorra a perda do prazo de

alguma tarefa (por exemplo, um travamento), a produção do resultado ainda é válida. Isso quer dizer que o sistema é não crítico (*soft real-time*), ou seja, a perda do prazo é tolerada, apesar de não ser desejada.

Em um programa de previsão para o mercado financeiro, se o resultado não for produzido no tempo correto, a informação perde o seu valor. Isso quer dizer que a perda do *deadline* de uma tarefa torna a resposta inútil. Nesse caso, o sistema é considerado firme (*firm real-time*).

Quando a perda do prazo acarreta em um resultado catastrófico, o sistema é classificado como crítico (*hard real-time*). Por exemplo, um sistema de controle de semáforos ou um sistema de controle de uma aeronave. Se durante a execução do sistema alguma tarefa não cumprir seu requisito temporal, é a iminência de um acidente com danos elevados.

2.3 ESCALONAMENTO

O escalonamento das tarefas de tempo real é parte importante dentro do sistema computacional e é o foco deste trabalho. O escalonador tem como objetivo organizar a ordem de execução das tarefas de modo que elas possam cumprir o seu prazo. Alcançando esse objetivo principal, eles buscam usufruir o máximo da capacidade do(s) processador(es), com eficiência no procedimento. Para entender melhor sobre o escalonador, é necessário conhecer alguns conceitos que serão apresentados nas subseções a seguir.

2.3.1 Processadores

Quando o sistema computacional possui mais de uma unidade de processamento, esses processadores podem ser homogêneos, heterogêneos ou uniformes. Se os processadores forem heterogêneos, eles são diferentes, o que implica numa taxa de execução dependente do processador e da tarefa. Os processadores homogêneos são idênticos e, consequentemente, a taxa de execução de todas as tarefas são as mesmas em todos os processadores. Para os processadores uniformes, a taxa de execução de uma tarefa depende apenas da velocidade do processador, ou seja, um processador de velocidade 2 executará todas as tarefas exatamente com o dobro da taxa de um processador de velocidade 1 (DAVIS; BURNS, 2011).

2.3.2 Escalonamento preemptivo

Durante a execução de uma tarefa, pode chegar outra com maior prioridade. Nessa situação, deve ser feita uma escolha. Ou a tarefa de maior prioridade espera o término da execução da atual, mesmo sendo menos prioritária, ou a execução da tarefa atual é interrompida para que a mais prioritária execute (DAVIS; BURNS, 2011). Essa interrupção é conhecida como preempção, e os projetistas do algoritmo devem escolher se o escalonador será preemptivo ou não preemptivo.

2.3.3 Prioridades

Quando o *job* chega e está pronto para ser executado, devem ser estabelecidas prioridades para que o processador escolha um, caso existam mais *jobs* nesse estado. O escalonador pode determinar essa hierarquia de execução utilizando uma abordagem de prioridade fixa ou dinâmica. Liu e Layland (1973) apresentam dois algoritmos, *Rate Monotonic* (RM) e *Earliest Deadline First* (EDF), que são escalonadores preemptivos. O RM é um exemplo de escalonamento com prioridade fixa, o qual atribui prioridade mais alta às tarefas que possuem um menor período. Já o EDF faz uso de prioridade dinâmica, escolhendo as tarefas que possuem *deadline* mais próximo. O EDF obtém um resultado ótimo em sistemas com um único processador compostos por tarefas periódicas, preemptivas e com *deadline* implícito. Isso significa que, nessas condições, ele consegue utilizar a capacidade total do processador. Por esse motivo, diversas implementações, até mesmo do estado da arte, utilizam-o como base.

2.3.4 Classificações para o escalonador

Os algoritmos concebidos para solucionar o problema do escalonamento de tempo real em múltiplos processadores são classificados tradicionalmente como globais ou particionados. Entretanto, há uma terceira classificação, conhecida como semi-particionados, que visa combinar o melhor das abordagens. Ser a junção dos pontos vantajosos de ambas as estratégias, foi o motivo primordial para a escolha de um algoritmo semi-particionado como base deste trabalho.

Na abordagem global, todas as tarefas são armazenadas numa única fila ordenada por prioridade, a qual é compartilhada entre os processadores, e uma tarefa pode migrar entre essas unidades de processamento durante a execução. O *Pfair* apresentado por Baruah et al. (1996) é

um exemplo de algoritmo global que consegue atingir o ótimo. Na particionada, o conjunto de tarefas é particionado levando a atribuição de cada tarefa a um processador dedicado, no qual todos os seus *jobs* serão executados. Os processadores são escalonados de forma independente e uma tarefa não pode migrar de uma unidade de processamento para outra. Essa estratégia é uma forma de reduzir o problema, tratando o ambiente com multiprocessadores como um conjunto de processadores únicos. Davis e Burns (2011) mencionam os algoritmos particionados, sendo eles combinações dos algoritmos EDF e RM com heurísticas utilizadas para resolver o problema do particionamento (*bin-packing*) como: *First-Fit* (FF), *Next-Fit* (NF), *Best-Fit* (BF) e *Worst-Fit* (WF). E, por fim, a estratégia semi-particionada (ou divisão de tarefas), é considerada como intermediária entre as duas anteriores. Ela atribui as tarefas aos processadores, como na particionada, porém, permite que algumas tarefas sejam divididas/migrem entre os processadores (ANDERSSON, 2019). Os algoritmos abordados por Bletsas e Andersson (2009), Sousa et al. (2013) e Brandenburg e Gul (2016) usam essa estratégia.

Andersson (2019) demonstra as vantagens e desvantagens da estratégia global e particionada. Os escalonadores particionados geram poucas preempções e nenhuma migração, em decorrência da atribuição que é feita das tarefas para cada processador. Contudo, existem conjuntos de tarefas que utilizam pouco mais de 50% de toda a capacidade de processamento, mas falham em cumprir seu prazo. Isso quer dizer que os escalonadores particionados possui essa grande limitação no uso do processador. Já os escalonadores globais conseguem escalonar um conjunto de tarefas que utiliza mais de 50% da capacidade do processador, mas, eles tendem a gerar um grande número de preempções.

Então, os pesquisadores buscaram uma abordagem alternativa, chamada de escalonamento semi-particionado (também chamado de divisão de Tarefas) com o objetivo de combinar o melhor do escalonamento global e particionado. Isto é, obter um limite de utilização maior do que o particionado sem ter um alto número de preempções característico do escalonamento global. Visto que ela consegue unir as vantagens de ambas estratégias, a escolha por um escalonador semi-particionado acaba sendo a melhor opção e, por isso, este trabalho utiliza o *Notional Procesors* como base que é um algoritmo semi-particionado. A desvantagem dessa estratégia fica por conta da ociosidade dos processadores, a qual ocasiona perda na capacidade de processamento. Isso acontece devido ao cuidado existente de sincronização, para que as partes da tarefa não sejam executadas simultaneamente em mais de um processador como demonstrado

por Andersson e Tovar (2006) e os algoritmos semi-particionados, para lidar com isso, recorrem a mecanismos que reservam mais tempo do que a tarefa realmente precisa, impondo ociosidade ao processador. A solução desse problema é o objetivo desse trabalho, levando em conta que o escalonador proposto por Massa et al. (2014) obteve sucesso ao resolvê-lo com o QPS, através dos servidores mestre-escravos. É importante ressaltar que o QPS utiliza o *quasi-partitioning* e não o semi-particionamento, o que deixa em aberto a possibilidade de utilizar a solução dada no contexto do *quasi-partitioning* em uma abordagem semi-particionada.

2.3.5 Modelo de tarefas

Neste trabalho o escalonador semi-particionado proposto é preemptivo e o modelo de tarefas utilizado considera tarefas periódicas com *deadline* implícito e processadores homogêneos. As tarefas são críticas, ou seja, não serão toleradas perdas de *deadline*, e é assumido que não há atraso entre a liberação do *job* e sua execução.

3 ALGORITMO NOTIONAL PROCESSORS

O *Notional Processors* foi a solução semi-particionada escolhida para o desenvolvimento deste trabalho por ser relevante dentre os que utilizam essa estratégia, presente em diversos trabalhos como (DAVIS; BURNS, 2011) e (MASSA et al., 2014). Além disso, ele é um algoritmo semi-particionado fácil de compreender, pela utilização das estruturas que representam os processadores lógicos, auxiliando o entendimento de mais pessoas, até as que desejam iniciar nessa área. Neste capítulo, ele será explicado, além de abordar outros escalonadores que servem de suporte para sua compreensão.

3.1 ESTRATÉGIAS EM UM PROCESSADOR

O *Earliest Deadline First* (EDF) foi apresentado por Liu e Layland (1973) como o algoritmo dinâmico de escalonamento baseado no *deadline*. Ele tem a premissa de atribuir maior prioridade às tarefas que possuem *deadline* absoluto mais próximo. Dessa forma, a prioridade da tarefa pode mudar com o tempo, ao contrário dos algoritmos que atribuem essa prioridade estaticamente.

No artigo de Liu e Layland (1973), ainda é proposto outro algoritmo, o *Rate Monotonic* (RM). O RM é uma solução de prioridade fixa, atribuindo prioridade mais alta às tarefas que possuem um menor período. Enquanto o limiar de utilização no RM, quando o número de tarefas tende ao infinito, é $\ln(2)$, o EDF consegue escalonar conjuntos de tarefas, utilizando a capacidade máxima do processador (100%), desde que atenda a um modelo de tarefas periódicas com *deadline* implícito. É importante salientar que o artigo realiza essas provas em um sistema com um processador, ou seja, o EDF comporta-se diferente em múltiplos processadores dependendo de como for implementado.

Embora essas soluções considerem um ambiente com um único processador, elas são aproveitadas em muitos algoritmos que propõem o escalonamento em multiprocessadores que utilizam a abordagem particionada, vide o apresentado por Andersson, Baruah e Jonsson (2001) o qual utiliza o RM como base e Andersson e Tovar (2006) que usa o EDF. Porém, o EDF tem um maior destaque e mais utilizações em implementações atuais por ser ótimo em um único

processador.

3.2 ESTRATÉGIAS SEMI-PARTICIONADAS DE ESCALONAMENTO

O termo escalonamento semi-particionado foi cunhado no artigo de Kato e Yamasaki (2009). Logo, algoritmos anteriores à 2009 não o utilizam. Posteriormente, o termo escalonamento semi-particionado passou a ter uma interpretação mais ampla; qualquer algoritmo em que uma tarefa pode ser dividida antes do tempo de execução passou a ser referido como escalonamento semi-particionado (ANDERSSON, 2019).

O algoritmo *EDF with task splitting and k processors in a group* (EKG) apresentado por Andersson e Tovar (2006) foi um dos primeiros a utilizar a abordagem semi-particionada, fazendo a comunidade de pesquisa começar a trabalhar seriamente na divisão de tarefas (ANDERSSON, 2019). O objetivo principal é ter um número menor de preempções do que uma implementação global e um limite maior de utilização que uma estratégia particionada. O modelo de tarefas da solução é baseado em tarefas periódicas com *deadlines* implícitos, em processadores idênticos.

A parte de divisão de tarefas é um dos pilares da solução, alocando-as aos processadores de forma que a utilização não exceda 100%. O parâmetro k é um valor escolhido pelo projetista tal que $1 \leq k \leq m$, sendo m o número de processadores. A partir do k, é calculado o valor de SEP, como mostrado na equação 3.1, o qual servirá para separar as tarefas em leves e pesadas. Uma tarefa τ_i é pesada se $C_i/T_i > SEP$, caso contrário, é leve. As tarefas pesadas são atribuídas aos seus próprios processadores dedicados, os restantes se dedicam a lidar com as tarefas leves.

$$SEP = \begin{cases} \dfrac{k}{k+1} & k < m \\ 1 & k = m \end{cases} \tag{3.1}$$

Na parte do escalonamento, as tarefas pesadas são executadas no seu processador assim que chega. As leves são segmentadas e, após o cálculo dos dois instantes os quais as tarefas devem sofrer preempção, uma das tarefas divididas é executada antes do primeiro instante e a outra tarefa dividida é executada depois do segundo instante. Durante esse intervalo, as

tarefas não divididas são escalonadas utilizando o EDF. Outro ponto importante do algoritmo é a técnica de espelhar o escalonamento, demonstrada no artigo como uma solução para diminuir a ocorrência de preempções

Ao selecionar $k = 2$, o limite de utilização do algoritmo é de 66%. Isso é mostrado por Andersson e Tovar (2006) através do teorema demonstrado que diz: se as tarefas possuírem uma utilização total do sistema de $U_s \leq SEP$, e são escalonados pelo EKG, então todos os *deadlines* serão cumpridos. Selecionando k = m, o limite de utilização é 100%.

Além do seu limite de utilização maior, o limite de preempções provado no artigo de até $2k$ por *job* é apontado como uma grande vantagem dele em relação a outras implementações como o Pfair.

O EKG teve uma versão adaptada para as tarefas esporádicas, chamada de *Sporadic EKG* (S-EKG) (ANDERSSON; BLETSAS, 2008). Nessa versão, são utilizados intervalos de tempo (*timeslots*) de duração definida S. O parâmetro δ, cujo valor é definido pelo projetista, é utilizado no cálculo de S, sendo $S = min(T)/\delta$, onde $min(T)$ é o menor tempo entre chegadas das tarefas. Quanto maior for o valor de δ, maior será o seu limite de utilização, chegando a 1 se S for o máximo denominador comum do tempo mínimo de chegada das tarefas. Entretanto isso também leva a um aumento no número de preempções.

O *Earliest Deadline Deferrable Portion* (EDDP) (KATO; YAMASAKI, 2008) apresenta a abordagem de escalonamento em porções, a qual não utiliza reservas de tempo. Se uma tarefa possui utilização que não é suficiente para ultrapassar o limite do processador, ele é considerada fixa. Caso a utilização ultrapasse esse limite, ela é dividida em duas porções e a tarefa é denominada migrante. Essas tarefas são escalonadas com um escalonador normal para uma unidade de processamento. Em sua estratégia de sincronização, quando a a parte principal da tarefa é escalonada para execução, a execução de sua outra parte é suspensa. A utilização dessa técnica garante que as partes de uma tarefa dividida não serão executadas simultaneamente. O limite de utilização desse algoritmo é de 65%.

Andersson (2019) defende que antes do desenvolvimento da divisão de tarefas/semi-particionamento, era necessário escolher entre escalonadores particionados, os quais não permitem migração e devido a isso possuem utilização limitada a no máximo 50%, ou escalonadores globais que permitem um limite de utilização alto, mas geram um grande número de preempções.

Com o desenvolvimento do semi-particionamento, tornou-se possível obter o melhor de ambas as estratégias.

3.3 NOTIONAL PROCESSORS

O *Notional Processors* (BLETSAS; ANDERSSON, 2009) apresenta um método de escalonamento através da criação de processadores lógicos. O algoritmo utiliza um modelo de tarefas esporádicas com *deadlines* implícitos.

A implementação utiliza janelas de tempo periódicas, que possuem duração fixa, chamadas de reservas, onde as tarefas são escalonadas. Essas reservas são criadas para buscar uma previsibilidade na ocorrência de tempos ociosos nos processadores, utilizando-os para escalonar tarefas adicionais (BLETSAS; ANDERSSON, 2009).

Os intervalos de tempo S são criados com duração igual ao menor tempo entre chegadas das tarefas do sistema. Com o intervalo definido são criadas as reservas de tempo para execução, as quais correspondem a uma fração do intervalo S que seja suficiente para as tarefas cumprirem os seus *deadlines* quando escalonadas pelo EDF. As reservas são intercaladas com "*gaps*", que são períodos ociosos no processador. Sempre que o *"gap"* do processador P_p termina, começa o do processador P_{p+1}. A junção das capacidades de processamento dos períodos ociosos é o que dá a origem aos processadores lógicos. A Figura 2 mostra 5 processadores físicos e o processador lógico P_6 resultante do mapeamento das áreas ociosas dos processadores físicos.

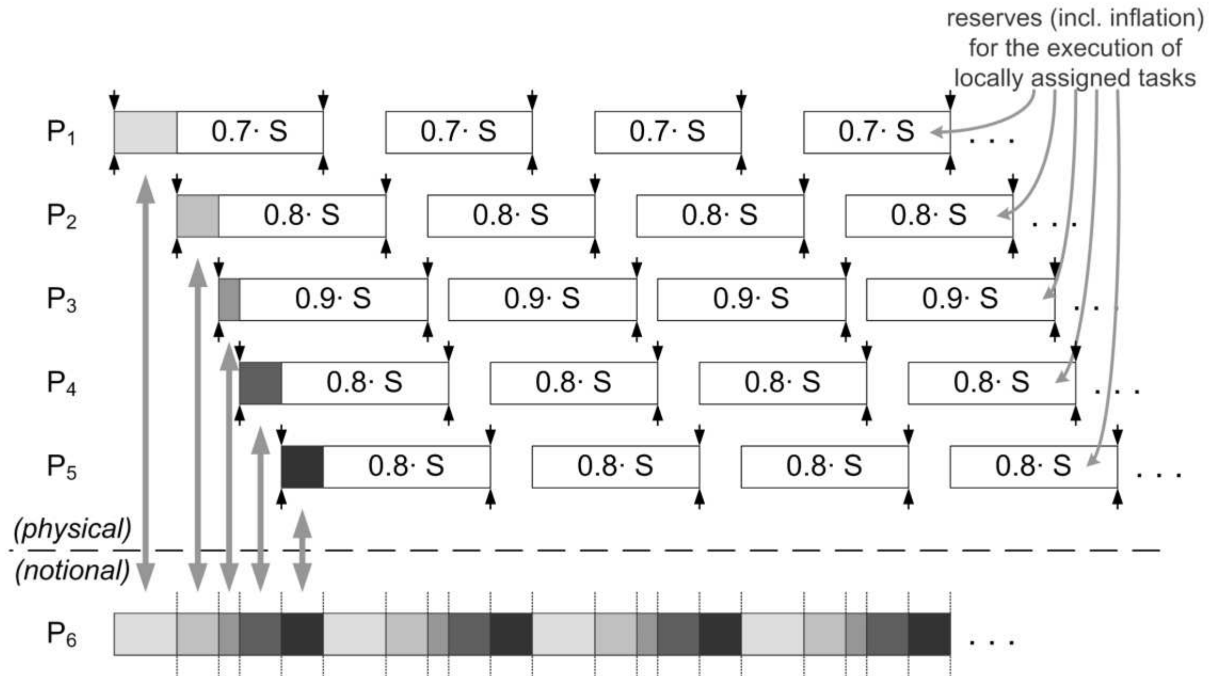
A grande vantagem do algoritmo é superar o limite de utilização da versão do S-EKG, o qual é de 65,7% (O limite de utilização do *Notional Processors* é de 66%), tendo também menos preempções do que o melhor caso possível do S-EKG (BLETSAS; ANDERSSON, 2009).

3.3.1 Distribuição das Tarefas pelos Processadores

Bletsas e Andersson (2009) definem a estrutura do algoritmo em três etapas:

- Primeiro, as tarefas são atribuídas aos processadores físicos, até que uma tarefa não possa ser atribuída em nenhum lugar;
- Então, a carga de trabalho é restrita em cada processador para execução dentro de reservas periódicas (de tamanho apropriado) e são organizados os intervalos de tempo entre as reservas nos processadores físicos em processadores lógicos;

Figura 2 – Ilustração de um *notional processor*



Fonte – Retirada de BLETSAS; ANDERSSON

- As tarefas restantes são atribuídas aos processadores lógicos.

Para fazer essa atribuição aos processadores, é utilizado o empacotamento (*bin-packing*) *First-Fit Heavy-First* (FFHF). A técnica *First-Fit* (FF) atribui as tarefas ao primeiro processador que ela cabe, considerando as atribuições anteriores. Se a primeira tentativa de alocação falhar, o algoritmo tenta alocar a tarefa a outro processador até que seja possível atribuí-la. Normalmente, quando uma tarefa não consegue ser alocada em nenhum processador, a execução é interrompida e a falha é declarada. Porém, o *Notional Processors* utiliza uma estratégia diferente quando isso ocorre. O FFHF utilizado pelo *Notional Processors* é uma variante que garante que todas as tarefas serão atribuídas (assumindo que as utilizações das tarefas não excedam 100%), a não ser que todos os processadores sejam utilizados mais que 50%. Um conjunto de tarefas está ordenado seguindo a ordem *heavy-first* se e somente se toda tarefa com utilização maior que 50% preceder toda tarefa com utilização igual ou menor que 50% (BLETSAS; ANDERSSON, 2009). Essa estratégia de empacotamento é utilizada para as atribuições tanto no processador físico quanto nos processadores lógicos criados posteriormente.

O FF é utilizado duas vezes, executando primeiro para alocar as tarefas aos processadores físicos, sendo interrompido quando uma tarefa não consegue ser atribuída a nenhum deles. Essa falha na alocação é onde começa a criação dos processadores lógicos. Depois de

criados, o FF será requisitado novamente, para alocar as tarefas aos processadores lógicos. Se nessa segunda vez, uma tarefa não conseguir ser atribuída aos processadores lógicos, o algoritmo é parado.

3.3.2 Escalonamento

Os processadores lógicos são estruturas que fazem o mapeamento para cada instante, indicando algum processador físico (provavelmente ocioso). Quando a tarefa está executando no processador lógico, na verdade ela está executando no processador físico que o processador lógico indicou estar livre naquele momento. O algoritmo utiliza o EDF para a escalonar as tarefas tanto nos processadores físicos quanto nos processadores lógicos. Quando o instante t pertence ao intervalo de uma reserva de tempo do processador físico, as tarefas são escalonadas normalmente com o EDF. Caso contrário, é preciso fazer uma verificação se nesse tempo específico algum processador lógico aponta para um processador físico através do mapeamento feito estaticamente. Então, o processador físico irá executar uma tarefa se o processador lógico estiver indicando a execução naquele instante ou estiver dentro de sua reserva. Não ocorrendo uma dessas duas condições, o processador físico fica ocioso.

3.4 OCIOSIDADE TÍPICA DAS ESTRATÉGIAS SEMI-PARTICIONADAS

A estratégia semi-particionada leva vantagem sobre a particionada, no que diz respeito ao limite de utilização, pois utilizam o conceito de divisão de tarefas. Através dos algoritmos mencionados anteriormente, é possível perceber essa vantagem no limite de utilização sobre a estratégia particionada, com uma quantidade menor de preempções do que a estratégia global. Porém, a delimitação de áreas para execução de tarefas, como as utilizadas no *Notional Processors* e S-EKG, traz a necessidade de criar folgas no processador. Nas estratégias semi-particionadas, é possível dividir as tarefas entre os processadores e isso cria a preocupação do escalonador escolher a mesma tarefa em dois processadores distintos ao mesmo tempo. Caso isso ocorra, uma das partes da tarefa tem que ser bloqueada, interrompendo o escalonamento natural. Para compensar esses bloqueios, as estratégias semi-particionadas alocam reservas de processamento maiores do que as tarefas realmente precisam. Mas, recursos desse tipo são necessários nessas estratégias porque é preciso impedir que ocorra essa execução simultânea das tarefas migratórias.

4 SERVIDORES MESTRE-ESCRAVOS

Os servidores mestre-escravos tem um papel muito importante na sincronização das tarefas no escalonador do *Quasi-Partitioned Scheduling* (QPS). Por esse motivo, esse recurso é fundamental para a construção do algoritmo proposto neste trabalho. O diferencial desse recurso é sincronizar as tarefas sem as perdas de capacidade computacional que os mecanismos dos escalonadores semi-particionados impõem. Neste capítulo, será explicado como esses servidores conseguem fazer essa sincronização sem o custo da perda de capacidade computacional.

4.1 SINCRONIZAÇÃO DE TAREFAS COM MIGRAÇÃO

Quando é permitida a migração de uma tarefa, para que uma tarefa dividida não seja executada simultaneamente em processadores distintos, é necessário utilizar recursos de sincronização. Portanto, quando uma tarefa migrante está escalonada em um processador, é preciso impedir que ela seja escalonada em outro. No caso do algoritmo *Notional Processors*, as reservas de execução tem essa finalidade. Porém, como visto anteriormente, essa estratégia resulta em ociosidade por inserir folgas nos processadores. O mecanismo dos servidores mestre-escravos sincroniza as tarefas sem a necessidade de usar essas folgas.

4.2 QPS

Massa et al. (2014) define o QPS como o primeiro algorimo de escalonamento capaz de adaptar a estratégia de escalonamento em função da carga do sistema. O QPS monitora a carga do sistema em tempo de execução e alterna entre o modo QPS e EDF, conforme a necessidade do conjunto de execução. A adaptação dinâmica é apontada como responsável por reduzir drasticamente o número de migrações que o escalonador necessita.

De forma resumida, o funcionamento se baseia no particionamento das tarefas em subconjuntos de dois tipos: conjuntos de execução menores e maiores. Os subconjuntos menores requerem somente um processador, enquanto os maiores necessitam de múltiplos processadores. Caso todos os subconjuntos sejam menores, o QPS é se comporta como o EDF particionado. Já os conjuntos de execução maiores, são escalonados ou por um conjunto de servidores QPS

em múltiplos processadores (modo QPS), ou por EDF em um único processador (modo EDF), dependendo dos requisitos de execução. Através do monitoramento dos conjuntos de execução maiores em tempo de execução, o QPS é capaz de permutar entre os dois modos, permitindo a adaptação dinâmica à carga do sistema (MASSA et al., 2014).

4.3 SERVIDORES DE TAXA-FIXA

O servidor de taxa-fixa é um conceito utilizado no algoritmo e é descrito por Massa et al. (2014) como um mecanismo de escalonamento usado para reservar tempo de processamento a um conjunto de tarefas ou outros servidores, conhecidos como seus clientes. O servidor se apresenta como uma tarefa para mecanismos de escalonamento externos, como o EDF, e libera uma série de *jobs* virtuais (MASSA et al., 2014). A taxa de um servidor σ, representada por $R(\sigma)$ sendo $R(\sigma) \leq 1$, corresponde à fração do processador necessária para execução de seus clientes. As seguintes regras são mencionadas por Massa et al. (2014) para definir os atributos e comportamentos de um servidor:

- ***Deadline***. O próximo *deadline* de um servidor σ após o tempo t é denotado por $D(\sigma,t)$. Esses *deadlines* incluem, mas não são limitados por, os *deadlines* dos clientes de σ.
- **Liberação de *job***. Um *job* é denotado por J:(c,r,d), onde c,r e d representam, respectivamente, WCET, tempo de liberação e o *deadline* absoluto . Um *job* liberado pelo servidor σ no tempo r satisfaz $c = R(\sigma)(d-r)$ e $d = D(\sigma,r)$
- **Ordem de execução**. Sempre que um *job* J de um servidor σ executa, σ escalona os *jobs* de seus clientes para execução na ordem do EDF.

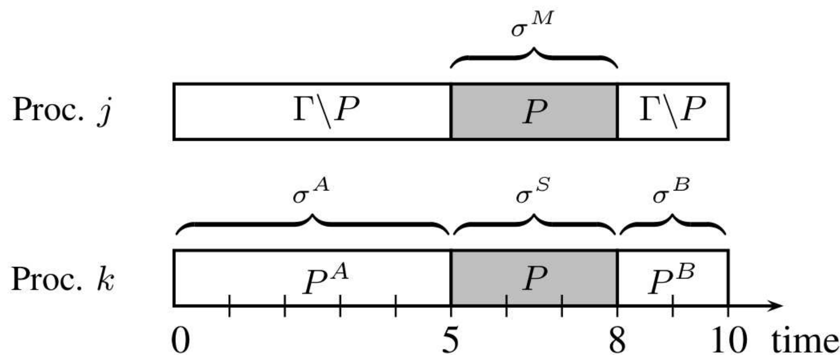
Esses servidores de taxa-fixa são utlizados pelo QPS para realizar o escalonamento. Considerando P um conjunto de servidores com a taxa $R(P) \leq 1$, P é um conjunto de execução menor e o escalonamento consegue ser realizado com sucesso utilizando o EDF. O conjunto de execução maior possui taxa $R(P) > 1$ (MASSA et al., 2014). Os servidores QPS lidam com esses conjuntos maiores, cujo criação e funcionamento serão explicados a seguir.

Dado um conjunto de tarefas periódicas Γ e considerando $P = \{\tau_1 : (6,15), \tau_2 : (12,30), \tau_3 : (5,10)\}$ um subconjunto de Γ, onde $\tau_n : (c,d)$. A taxa desse subconjunto é igual a 1,3 ($U_{\tau_1} = 0,4$, $U_{\tau_2} = 0,4$ e $U_{\tau_3} = 0,5$), podendo ser representada por $R(P) = 1+0,3$. Como a taxa foi superior a 1, é criada uma bi-partição sendo elas $P^A = \{\tau_1, \tau_2\}$ e $P^B = \{\tau_3\}$ e os servidores QPS σ^A, σ^B, σ^M e σ^S são criados. Sendo $x = 0,3$, seguindo a definição dos servidores

$\sigma^A : (R(P^A)-x, P^A)$, $\sigma^B : (R(P^B)-x, P^B)$, $\sigma^S : (x, P)$ e $\sigma^M : (x, P)$, nesse exemplo os servidores são $\sigma^A : (0.5, P^A)$, $\sigma^B : (0.2, P^B)$, $\sigma^S : (0.3, P)$ e $\sigma^M : (0.3, P)$. A qualquer momento t, todos os servidores QPS associados à P compartilham do mesmo *deadline* $D(P,t)$. σ^A e σ^B são servidores dedicados associados com P^A e P^B, respectivamente. σ^M e σ^S são os servidores mestre (*master*) e escravo (*slave*), respectivamente. A Figura 3 mostra como esses servidores seriam escalonados dentro de $[0, 10)$ (MASSA et al., 2014).

4.4 MECANISMO DOS SERVIDORES MESTRE-ESCRAVO

Figura 3 – Ilustração do escalonamento usando o QPS



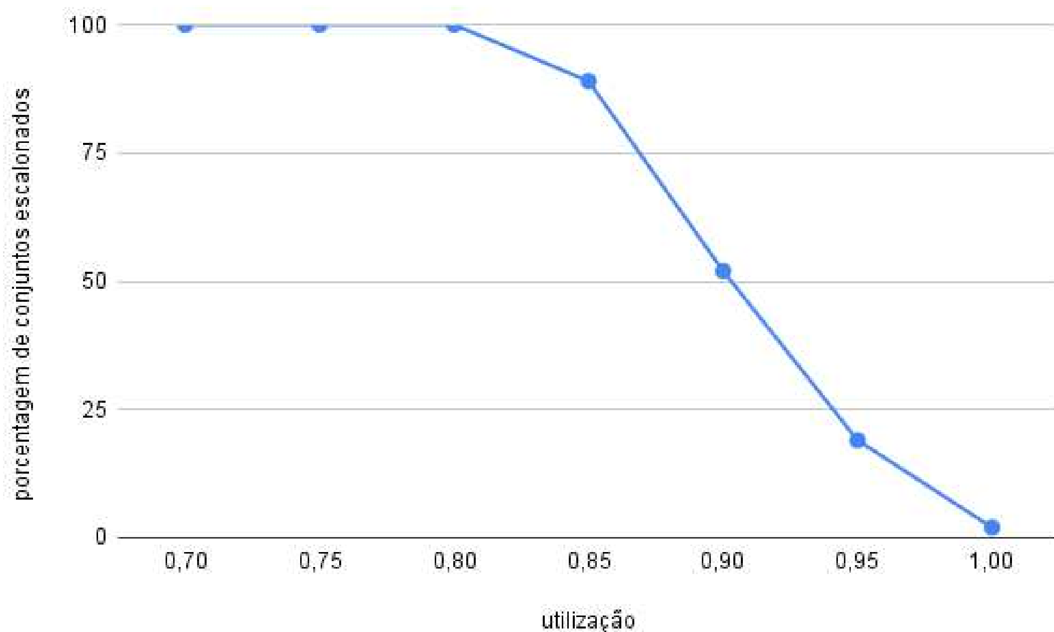
Fonte – Retirada de MASSA et al.

Enquanto os servidores QPS σ^A e σ^B lidam com a execução não paralela de P^A e P^B, respectivamente, os servidores σ^M e σ^S lidam com a execução paralela deles. Como a soma das taxas $\sigma^A + \sigma^B + \sigma^S = 1$, os servidores σ^A, σ^B e σ^S podem executar em um único processador. O servidor σ^M, precisa ser executado em outro processador diferente. Sempre que σ^M é escalonado para executar, σ^S também é, em outro processador, o que justifica o nome dado a eles. Outra característica desse par de servidores é a não execução do servidor escravo quando o servidor mestre não está executando. O comportamento hierárquico deles resulta numa execução paralela de uma tarefa de P^A e outra de P^B (MASSA et al., 2014). A Figura 3 exibe esse comportamento, com os servidores mestre e escravo atuando no intervalo 5-8 realizando a execução paralela. O mestre e escravo podem potencialmente servir qualquer cliente do conjunto de execução. Então, como forma de prevenir que dois clientes executem simultaneamente em dois processadores, sempre são escolhidos clientes diferentes para mestre e escravo (MASSA et al., 2014). Esse mecanismo é a forma encontrada pelo QPS para realizar a sincronização das tarefas migratórias.

5 SERVIDORES MESTRE-ESCRAVOS NO NOTIONAL PROCESSORS

A Figura 4 mostra um gráfico com as porcentagens de conjuntos escalonados por utilização, em 4 processadores e utilizações de 70% a 100%. Esses valores obtidos através da simulação do escalonamento mostram que conforme a utilização do conjunto de tarefas vai aumentando, o *Notional Processors* perde a capacidade de escaloná-los, chegando a praticamente 0 nos grupos de 100%. Isso reforça a constatação que a ociosidade imposta pelo mecanismo de sincronização implica na incapacidade do escalonador garantir o escalonamento de grupo de tarefas com utilizações maiores. Portanto, a substituição do mecanismo de sincronização original do *Notional Processors* pelo servidores mestre-escravos é uma decisão lógica, pois os servidores não precisam deixar as unidades de processamento ociosas. Nas seções abaixo, será descrito como foi feita a inclusão desses servidores no algoritmo semi-particionado

Figura 4 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em quatro processadores



Fonte – O autor

5.1 CRIAÇÃO DOS PROCESSADORES LÓGICOS

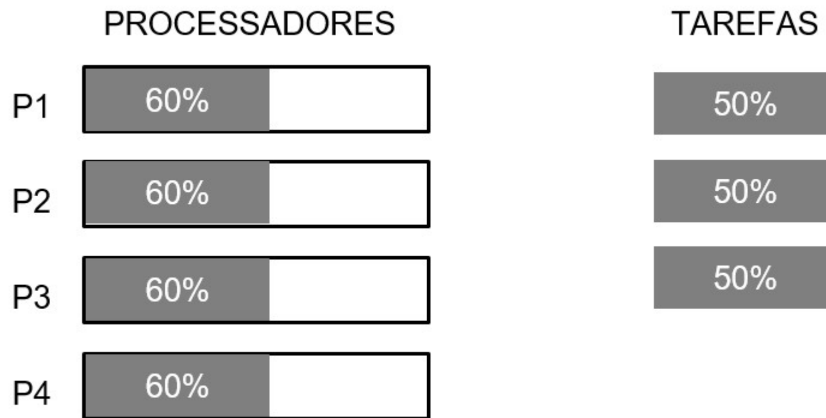
A adição dos servidores mestre-escravos no algoritmo *Notional Processors* altera a forma como os processadores lógicos são criados. No algoritmo original, os processadores lógicos precisam que as reservas para execução das tarefas que não migram estejam criadas. E então, os processadores lógicos fazem o mapeamento das áreas que não estão reservadas para a execução das tarefas que migram. Dessa forma, a distribuição dessas áreas de execução entre os processadores físicos é a forma do *Notional Processors* não permitir a execução simultânea de uma mesma tarefa migratória. Assim, com a adição dos servidores mestre-escravos, não há necessidade de estabelecer reservas para execução, já que o trabalho de sincronização será feito por eles.

No algoritmo original, aplicando o FFHF, quando uma tarefa não consegue ser alocada em nenhum dos processadores físicos, é o momento que as reservas são criadas e os processadores lógicos fazem o mapeamento. Na modificação proposta nesse trabalho, o que difere nessa parte da criação é que não será necessário a utilização das reservas de execução e janelas de tempo. Isso quer dizer que o algoritmo modificado não precisa que o processador lógico seja um mapeamento estático indicando que em um instante específico um processador físico deve executar. Agora a única informação relevante para a criação dos processadores lógicos será as capacidades sobressalentes em cada processador físico. A indicação de qual processador agora se dá de forma dinâmica, sendo responsabilidade dos servidores mestre-escravos.

Considerando um sistema com 4 processadores e um conjunto de tarefas com 7 tarefas onde 4 tarefas possuem utilização de 60% e 3 possuem utilização de 50%, utilizando o FFHF, ao final do processo, as tarefas são atribuídas aos processadores, ficando no estado que mostra a Figura 5. As 4 tarefas de 60% de utilização são atribuídas aos processadores P1 até P4, restando as 3 tarefas de 50% que não podem ser alocadas a nenhum deles.

Se todas as tarefas fossem atribuídas aos processadores físicos, não teria a necessidade de criar os *notional processors*, uma vez que as tarefas poderiam ser escalonadas com o EDF particionado. Porém, nesse exemplo são necessários os processadores lógicos. No algoritmo original, esse é o momento de criar as reservas de execução para posterior mapeamento das áreas ociosas (criação do processador lógico). Mas, nessa modificação, esse passo é substituído pela criação dos servidores mestre-escravos os quais terão as capacidades iguais às capacidades

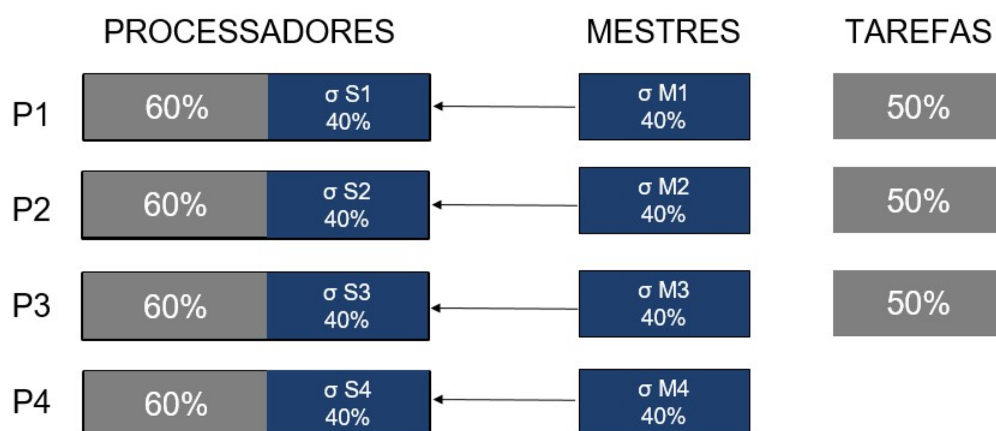
Figura 5 – Primeira rodada de atribuições com o FFHF



Fonte – O autor

restantes de cada processador. Nesse exemplo, cada processador físico terá um servidor escravo de capacidade igual a 40%, criando também seus respectivos servidores mestres de mesma capacidade, como mostra a Figura 6.

Figura 6 – Criação dos servidores mestre-escravos após atribuições

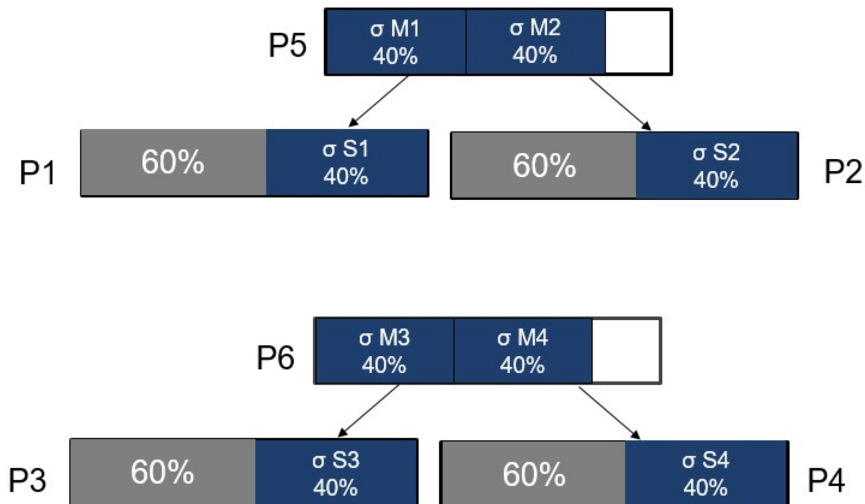


Fonte – O autor

Os *notional processors* serão criados a partir da junção desses servidores mestres.

Como o processador está restrito a 100% de utilização, nesse exemplo, não será possível juntar 3 processadores mestres para criar um processador lógico, pois resultaria em 120%. Ao invés disso, serão criados dois processadores lógicos P5 e P6 com capacidade de 80%, como pode ser visto na Figura 7.

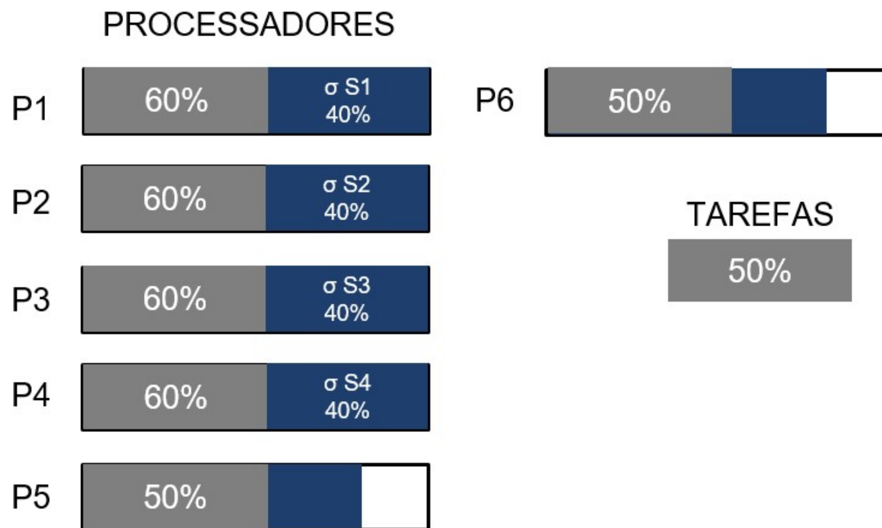
Figura 7 – Criação dos processadores lógicos a partir dos servidores mestres



Fonte – O autor

Agora com os processadores lógicos criados, é possível rodar o FF novamente para alocar as tarefas que estavam sem processadores. Após as atribuições, os processadores ficam no estado mostrado na Figura 8. As duas tarefas de 50% são alocadas aos processadores P5 e P6.

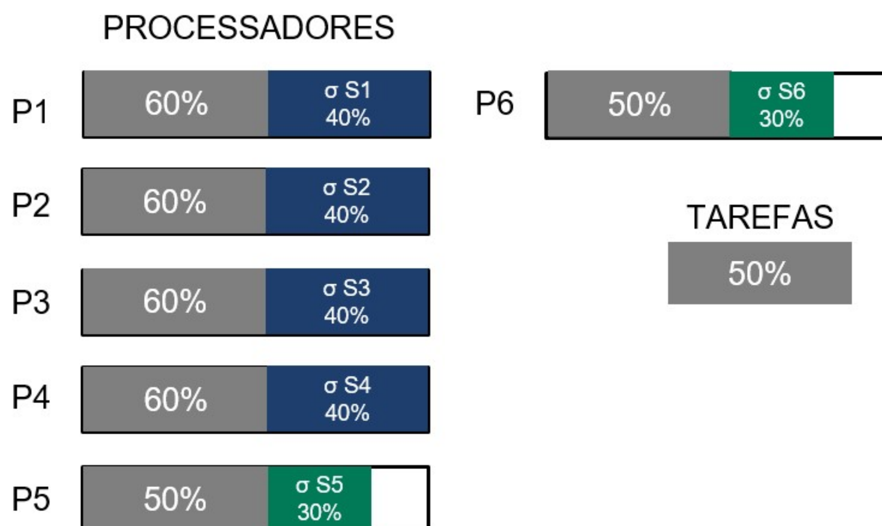
Figura 8 – Atribuição das tarefas aos processadores lógicos



Fonte – O autor

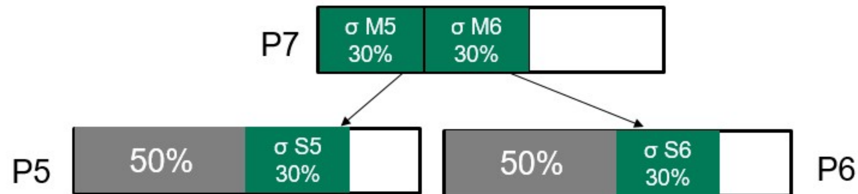
Ainda tem uma tarefa de 50% pendente de alocação, contudo, restaram 30% da capacidade de cada processador lógico. Como pode ser visto na Figura 9 e Figura 10, o processo de criação de servidores mestre-escravos se repete, agora para criar um *notional processor* de 2º nível, gerando o processador P7 com capacidade de 60%.

Figura 9 – Criação de servidores mestre-escravos para o processador lógico de 2º nível



Fonte – O autor

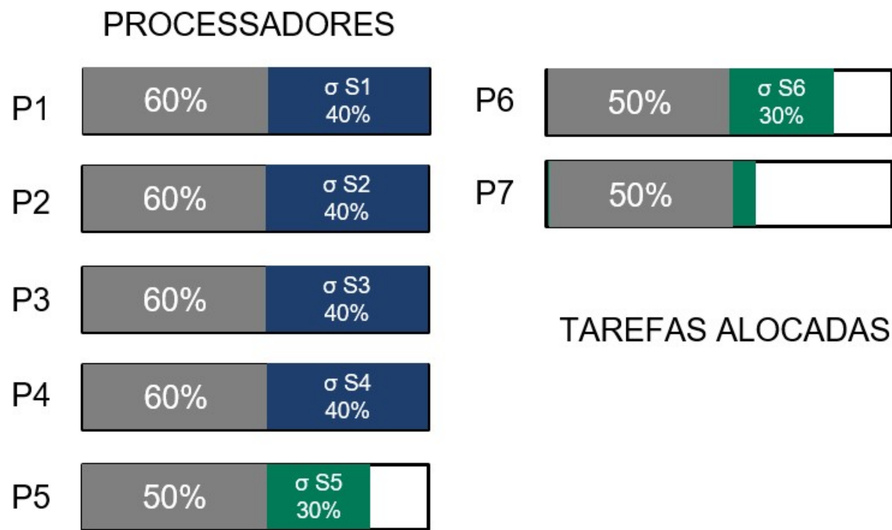
Figura 10 – Criação do processador lógico de 2º nível



Fonte – O autor

Por fim, a Figura 11 mostra o resultado após utilizar, mais uma vez, o FF. Ele atribui a última tarefa de 50% ao processador lógico de 2º nível P7. Através desse exemplo percebe-se que enquanto tiver capacidades sobressalentes nos processadores, é possível criar servidores mestre-escravos e gerar processadores lógicos a partir deles. Vale lembrar que se não tiver a capacidade de gerar os *notional processors* e ainda restar tarefas pendentes de atribuição, então, não é possível escalonar esse conjunto de tarefas com o algoritmo.

Figura 11 – Atribuição ao processador lógico de 2º nível



Fonte – O autor

5.2 ATRIBUIÇÃO DOS SERVIDORES MESTRE-ESCRAVOS

Os servidores mestre-escravos fazem a função importante de sincronizar as tarefas no QPS. Nessa modificação do *Notional Processors*, eles serão utilizados também para esse fim. Dois comportamentos dos servidores mestre-escravos são fundamentais para a escolha deles no papel de sincronizador das tarefas. O primeiro comportamento é a ativação, sempre que o servidor mestre executa, o escravo também. O segundo é o escalonamento de tarefas distintas, ou seja, quando uma tarefa é escalonada para o servidor mestre, uma tarefa distinta é escalonada para o servidor escravo. Com base nessas propriedades, as atribuições dos servidores serão feitas aos processadores físicos e lógicos.

Como visto nas Figuras 7 e 10, os servidores mestres serão atribuídos aos processadores lógicos e os servidores escravos aos físicos. Os servidores escravos são encarregados por escalonar uma tarefa nula. Isso quer dizer que, enquanto o processador lógico estiver executando uma tarefa real através do seu mestre, o servidor escravo correspondente no processador físico não estará executando, pois para eles são atribuídas tarefas nulas. Portanto, os servidores escravos estão sendo utilizados para ativar um servidor mestre que está no processador lógico. Através desse mecanismo, nota-se que a função dos processadores lógicos é indicar a execução das tarefas nos processadores físicos correspondentes.

5.3 ESCALONAMENTO

Pode-se entender o escalonador apresentado como a junção da estratégia de criar processadores virtuais do algoritmo *Notional Processors* (BLETSAS; ANDERSSON, 2009), com as decisões de escalonamento do QPS (MASSA et al., 2014). Foram criados os processadores virtuais, mas sem a utilização das janelas de tempo (*timeslots*), as quais são normalmente empregadas pelos algoritmos semi-particionados e que trazem ociosidade aos processadores. As decisões de escalonamento do *Notional Processors* original, se baseia justamente no mapeamento feito das áreas ociosas dos processadores físicos. Se tiver dentro da reserva daquele processador físico, é feito o escalonamento com o EDF normalmente. Caso contrário, é feita uma verificação se naquele instante tem um *notional processor* que aponta para o processador em questão. Se não tiver um processador lógico para o processador físico no instante da verificação, ele permanece ocioso. Como na modificação proposta neste trabalho, já foi mostrado que as reservas e *timeslots* não serão utilizadas, é necessário mudar essas decisões de escalonamento.

Tendo isso em vista, essas decisões serão aproveitadas do QPS, pois são as mais adequadas para lidar com os servidores mestre-escravos. Servidores esses que foram utilizados para substituir a estratégia com *timeslots* do *Notional Processors*, pelos comportamentos já descritos como a hierarquia e ativação simultânea. Como as estratégias com *timeslots* são conhecidas por trazer ociosidade aos processadores, logo, a implementação proposta traz uma melhora na escalonabilidade do algoritmo *Notional Processors*.

6 SIMULAÇÃO

Com o objetivo de avaliar o comportamento do algoritmo semi-particionado após a adição dos servidores mestres-escravo do QPS, a metodologia de pesquisa escolhida para este trabalho tem como pilar a simulação. A simulação tem como base a utilização de um simulador de eventos discretos, escrito na linguagem de programação *python*, que se encarregará de simular o escalonamento em multiprocessadores. O simulador foi construído originalmente para as avaliações do QPS (MASSA et al., 2014) tendo a parte do escalonamento customizável, o que possibilitou utilizá-lo neste trabalho. Desta maneira, foi possível implementar nele os escalonadores *Notional Processors* original e o *Notional Processors* com os servidores mestre-escravos.

6.1 SIMULAÇÃO DE EVENTOS DISCRETOS

Jain (1990) aborda três técnicas para avaliar a performance de um sistema computacional: a modelagem analítica, a simulação e a medição. No livro, Jain (1990) defende que a modelagem analítica requer muitas simplificações e suposições, apesar de ser a técnica com menor custo. As simulações podem possuir mais detalhes e requerem menos suposições, sendo mais próximo da realidade. Medições, embora estejam mais associadas à realidade, podem não fornecer resultados precisos porque muitos parâmetros ambientais, como a configuração do sistema, podem ser exclusivo de cada experimento. No caso do escalonamento de tempo real, o comportamento do processador, a utilização da memória cache e outras variáveis são exemplos do que pode provocar interferência no experimento. Além disso, as medições possuem um custo maior porque necessitam de equipamentos e instrumentos reais. Para fazer uma comparação do *Notional Processors* original com a modificação proposta neste trabalho, foi necessário simular considerando uma grande quantidade de processadores (nesse teste específico, foram 15 processadores). Seria mais difícil ter a disposição 15 unidades de processamento para realizar esse teste. Ademais, a simulação proporciona uma flexibilidade e agilidade maior na realização de testes onde são alteradas diversas variáveis como a quantidade de processadores e conjuntos de tarefas específicos. A maior desvantagem por optar pela simulação, é fazer algumas suposições para eventos que acontecem durante um experimento real, como é o caso dos custos atrelados à

troca de contexto. Contudo, ainda que esses custos não sejam aferidos diretamente, a escolha da quantidade de preempções e migrações como métricas de avaliação ajuda a ter uma noção do desempenho do algoritmo em um ambiente real. Diante dessas considerações, este trabalho opta por utilizar a simulação, para obter resultados próximos do real, mas sem a interferência de muitas variáveis ambientais e sem os custos associados a medição, principalmente com equipamentos.

A simulação de eventos discretos é adequada para o escalonamento de tempo real, pois lida com uma sequência de eventos que ocorrem em pontos distintos do tempo. A forma da simulação usada neste trabalho é a dirigida a eventos, ou seja, eles são simulados cronologicamente e após a simulação de um evento, o relógio é avançado para o tempo do próximo (MISRA, 1986). Isso evita que o simulador tenha que fazer as operações a cada mudança do relógio, assim, otimizando a quantidade de iterações necessárias no processo.

Para exemplificar a simulação de eventos discretos, é possível usar a rotina de uma pessoa. Ela pode de acordar às 6h da manhã, tomar banho às 6h15, tomar café após o banho e etc. Cada tarefa dessa rotina pode ser um evento discreto que compõe a simulação. Nessa abstração computacional, o tempo que acontece cada evento é uma informação indispensável, porque eles são armazenados em uma lista, denominada agenda, que ordena-os por tempo.

Supondo que o relógio do sistema está marcando o tempo 0, após a leitura de um evento da agenda que acontece no tempo 10, o simulador faz essa passagem do tempo através das mudanças de variáveis do sistema. Como a simulação é dirigida a evento, não é necessário simular cada tique do relógio. Então, agora o relógio indica o tempo 10 e é feita a simulação do que ocorreu nesse intervalo de 0 a 10 unidades de tempo. Considerando o escalonamento de tarefas de tempo real, nesse intervalo pode ter acontecido algum evento de chegada, *deadline*, término de execução de um *job* e *etc*.

6.2 SIMULAÇÃO DO ESCALONAMENTO

Para iniciar a simulação do escalonamento, é necessário fornecer as informações sobre as tarefas que serão escalonadas e a quantidade de processadores. Nesse caso, a opção será por tarefas fictícias, contendo as suas características mencionadas no capítulo 2, sendo elas: tempo de execução no pior caso C, *deadline* relativo D e o intervalo mínimo entre chegadas dos trabalhos T. Esses dados são obtidos através da leitura de um arquivo, o que é um facilitador

para realização de testes. Vale relembrar que algumas suposições são feitas nessa simulação. Os processadores são idênticos, ou seja, as tarefas executam da mesma forma em todos. As tarefas são periódicas, e seus *jobs* já ficam aptos a ficar prontos assim que chegam. E o *deadline* é implícito, portanto, o *deadline* da tarefa é igual ao seu período.

De posse das informações sobre os processadores e as tarefas, o simulador executa o procedimento *offline*, o qual tem que ser executado antes de começar de fato o escalonamento. Nesse procedimento, os processadores serão associados aos conjuntos de execução. Se for utilizada uma estratégia global, os conjuntos de execução serão associados a todas unidades de processamento disponíveis. Caso seja uma estratégia que utiliza particionamento, os conjuntos serão atribuídos aos seus respectivos processadores. Esse particionamento pode ser feito através de algoritmos de empacotamento (*bin-packing*) como o FF. Depois de feitas as atribuições aos processadores, os primeiros eventos já podem ser instanciados, indicando a chegada dos *jobs* das tarefas.

No simulador, os eventos possuem códigos únicos para identificar seu tipo, assim como propriedades particulares de cada um. Geralmente os eventos carregam qual *job* ou servidor eles fazem referência, isto é, se o evento indica a liberação de um *job*, esse *job* faz parte das propriedades do evento. Além das propriedades particulares de cada evento, a mais importante é o tempo que cada evento ocorre. Essa informação é imprescindível, pois trata-se de uma simulação de eventos discretos e é a partir dos tempos indicados que o sistema envelhecerá.

Diversos eventos são criados durante a simulação, e o simulador lida com cada um deles de maneira específica. O processamento de um evento, geralmente, resulta na criação de um evento distinto. Por exemplo, o que indica a chegada do período de uma tarefa cria outro de liberação do *job* dessa tarefa. O evento de liberação do *job* vai sinalizar que ele está apto a entrar na fila de *jobs* prontos para serem executados. São criados também eventos que marcam a chegada do *deadline* de um *job*, assim como os que indicam a finalização da execução do *job*. Os eventos de chegada do *deadline* são importantes para verificação se houve a perda do *deadline* de um *job*. Essa apuração deve ser feita pois, conforme a especificação do escalonamento feito neste trabalho, ele deve contemplar o tempo real crítico, não tolerando qualquer perda de *deadline*. Como também trata-se de um escalonamento preemptivo, o evento de finalização do *job* nem sempre quer dizer que ele foi de fato finalizado. Isso irá depender se algum outro *job* com prioridade maior não interrompeu sua execução. Caso tenha interrompido, esse evento de

conclusão tem que ser cancelado, e o *job* deve voltar a lista de *jobs* prontos.

O simulador processa todos os eventos na agenda que possuem o mesmo tempo marcado no relógio até que não tenha mais nenhum. Sempre que o próximo evento da agenda for de um tempo superior ao do relógio, é feita uma chamada ao escalonador, que é o responsável por fazer a execução dos *jobs* prontos com prioridade mais alta. Visto que o escalonador só é chamado após o processamento de todos os eventos do tempo atual, depois do término de sua execução é feito o processo de envelhecimento do sistema, que muda o tempo marcado pelo relógio para o tempo do próximo evento da agenda. Tendo como estrutura base a agenda e os tratamentos dados a cada um deles, o escalonador é a parte customizável do sistema. Cada escalonador possui os métodos específicos para decidir qual *job* executar durante um certo período. A utilização de servidores modifica também a maneira que os *jobs* são retornados para serem executados, existindo uma camada extra onde há uma busca dos *jobs* que são clientes de um determinado servidor.

Se não tiver problemas na atribuição das tarefas ou no atendimento dos *deadlines* dos *jobs*, a simulação ocorre até o tempo determinado para finalização, sendo ele informado pelo usuário. Ao final da simulação onde todas as tarefas foram escalonadas com sucesso, são retornadas as informações de migrações e preempções que ocorreram durante o processo. Para isso, o vetor montado com instâncias das tarefas é percorrido, e o método de contagem de migrações e preempções é chamado para cada uma delas.

Com as informações geradas por essa simulação do escalonamento através dos eventos discretos, é possível avaliar a modificação do algoritmo semi-particionado que é a proposta deste trabalho. O sucesso da modificação, depende da obtenção de resultados satisfatórios ao final do processo. Para um algoritmo de escalonamento, baixo número de preempções, migrações e um alto uso da capacidade computacional do sistema são os resultados almejados.

7 TESTES E RESULTADOS

Os resultados obtidos através da simulação de eventos discretos, possibilitam avaliar a performance do algoritmo proposto neste trabalho, utilizando as métricas escolhidas. As métricas são o número de preempções e migrações, que fazem parte da maioria dos trabalhos onde são apresentados algoritmos escalonadores. Elas estimam a performance da solução diante de vários cenários como diferentes quantidades de processadores e conjunto de tarefas. Sendo assim, esses testes e posteriores análises estão fundamentados por essas métricas que dão as informações necessárias para a avaliação da implementação.

7.1 MÉTRICAS

A execução do código de escalonamento possui diversos custos associados. Quando ocorre a preempção de uma tarefa, por exemplo, acontece uma troca de contexto. Isso quer dizer que a tarefa que será interrompida precisa ter seu contexto salvo antes de parar sua execução, e a tarefa que passará a executar naquele momento precisa carregar seu contexto. No caso da migração, além da troca de contexto, uma tarefa que migra perde todos os dados contidos na memória *cache* que estavam no processador, sendo necessário carregar novamente os dados em outro processador. Além disso, interrupções do sistema operacionais e variações temporais relacionadas às tarefas (conhecidas como *jitters*) são exemplos de custos do escalonamento real (OLIVEIRA, 2020). Esses são exemplos de custos associados ao escalonamento que através da forma como está sendo simulado o escalonamento, não são possíveis de medir.

Vale ressaltar que diante da incapacidade de avaliar diretamente esses custos, o mais comum é utilizar como métricas de performance a quantidade de preempções e migrações. A escolha por esses métricas está relacionado com o que foi dito no parágrafo anterior. As preempções estão relacionadas às trocas de contexto, as quais possuem um alto custo e as migrações, além dessas trocas, inutilizam a memória cache de um processador quando uma tarefa migra para outro.

7.2 CONJUNTOS DE TAREFAS

Para realizar a simulação do escalonamento, é necessário gerar conjuntos de tarefas e fornecê-los como entrada para o simulador. Essa entrada é feita através de arquivo com um formato específico, contendo todos os dados da tarefa. Como o escalonador leva em consideração tarefas periódicas e *deadline* implícito, o *deadline* e o intervalo mínimo entre chegadas possuem o mesmo valor. Além desses dois dados, os conjuntos devem informar a quantidade de processadores do sistema, o número de tarefas e seus respectivos WCET. Esses arquivos de texto que representam as coleções de tarefas foram gerados a partir de um programa construído em *Python*.

O gerador recebe como parâmetros, o número de processadores, a utilização do sistema pelo conjunto e a precisão dos valores que serão gerados. O número de tarefas da geração é determinado aleatoriamente, pertencendo a um intervalo que vai do número de processadores + 1 até 2 vezes o número de processadores. Dessa forma, os conjuntos terão uma variação na quantidade de tarefas para uma mesma utilização. O período tem o mesmo valor do *deadline*, e é gerado também aleatoriamente sendo um valor inteiro dentro do intervalo 1 a 100. A aleatoriedade na produção dos grupos de tarefas ajuda a fazer uma avaliação não enviesada.

Inicialmente, um número aleatório entre 0 e 1 é associado a cada tarefa para representar sua porcentagem de utilização do sistema. Se o desejado é a criação de um conjunto que possui 70% de utilização do sistema, ele não pode ultrapassar 0,7 multiplicado pela quantidade de processadores. Por exemplo, um grupo que tem utilização de 70% e 4 processadores, a soma da utilização das tarefas não deve ultrapassar 2,8. Porém, como os valores são gerados aleatoriamente, essa soma dos valores aleatórios pode ultrapassar o limite. Então, os valores são ajustados de maneira proporcional, para que a soma das utilizações das tarefas resulte exatamente na capacidade desejada. É importante destacar que nessa etapa de ajuste, o resultado pode ser superior a 1. Nesse caso, como uma tarefa não pode usar mais de 100% do sistema, o conjunto é descartado e o algoritmo dá origem a outro, até que chegue na utilização desejada.

Como o WCET pode ser um valor decimal e no gerador ele é obtido pela multiplicação da utilização da tarefa com o período, a precisão do gerador é um fator importante para não gerar discrepâncias. As aproximações para diminuir o número de casas decimais, podem gerar inconsistências no que diz respeito à utilização, podendo ultrapassar o limite do sistema.

Para lidar com esses conjuntos que tem utilização superior ao limite, é feita uma verificação. O conjunto que exceder a capacidade do sistema ou não estiver próximo da utilização desejada dentro da precisão são descartados, e o gerador tenta novamente criar um válido. Para a geração dos conjuntos de testes utilizados neste trabalho, foi utilizada a precisão de 10^{-2}, porque ela foi capaz de gerar os conjuntos com a utilização desejada em um tempo menor. Em precisões maiores, a geração dos conjuntos com uma alta utilização (95% em diante) demoram mais para ser criados, uma vez que aqueles grupos de tarefas que não estiverem dentro dessa precisão são descartados.

Ao final do processo, o programa gera os arquivos que representam os conjuntos de tarefas. Para facilitar, as variáveis presentes na Figura 12 foram definidas para customizar a geração. Antes de gerar os conjuntos, o usuário deve informar: a quantidade de processadores (*PROCESSORS_NUMBER*), a quantidade de coleções por utilização (*ROUNDS*), a utilização inicial (*INITAL_UTILIZATION*), a utilização final (*FINAL_UTILIZATION*), o valor incrementado à utilização inicial até chegar a 1 (*STEP*), a quantidade de casas decimais para as informações das tarefas (*DECIMAL_PLACES*) e o valor inicial para o identificador da geração (*GENERATION_INITIAL_NUMBER*).

Figura 12 – Variáveis de customização do gerador

```
1  class Enviroment:
2      PROCESSORS_NUMBER = 10
3      DECIMAL_PLACES = 2
4      GENERATION_INITIAL_NUMBER = 1
5      ROUNDS = 100
6      INITAL_UTILIZATION = .7
7      FINAL_UTILIZATION = 1
8      STEP = .05
```

Fonte – O autor

7.3 RESULTADOS

Após definidas as métricas, o desempenho do algoritmo pode ser avaliado através do simulador que já fora detalhado. Os algoritmos escalonadores foram executados em sistemas de 4 até 10 processadores. Realizando o teste de escalonamento com grupos de tarefas que possuem utilizações de 40% até 65%, os algoritmos não apresentaram quantidade relevante de migrações, chegando até 0 de migrações por *job* (em média) em 4 a 10 processadores. Levando isso em consideração não foram construídos gráficos para essas utilizações menores. Portanto, foram gerados conjuntos com utilização de 0,7 até 1, com o incremento de 0,05. Cada utilização dá

origem a 100 conjuntos, totalizando 700 coleções escalonadas por sistema. Com essa quantidade de conjuntos foi possível obter uma média dos dados resultantes que cada escalonador retornou, variando as utilizações e processadores. Para identificação nos resultados, a implementação proposta nesse trabalho está denominada como *Notional Processors with Master and Slave* (NPwMS). Todos os gráficos estão disponíveis nos apêndices deste trabalho.

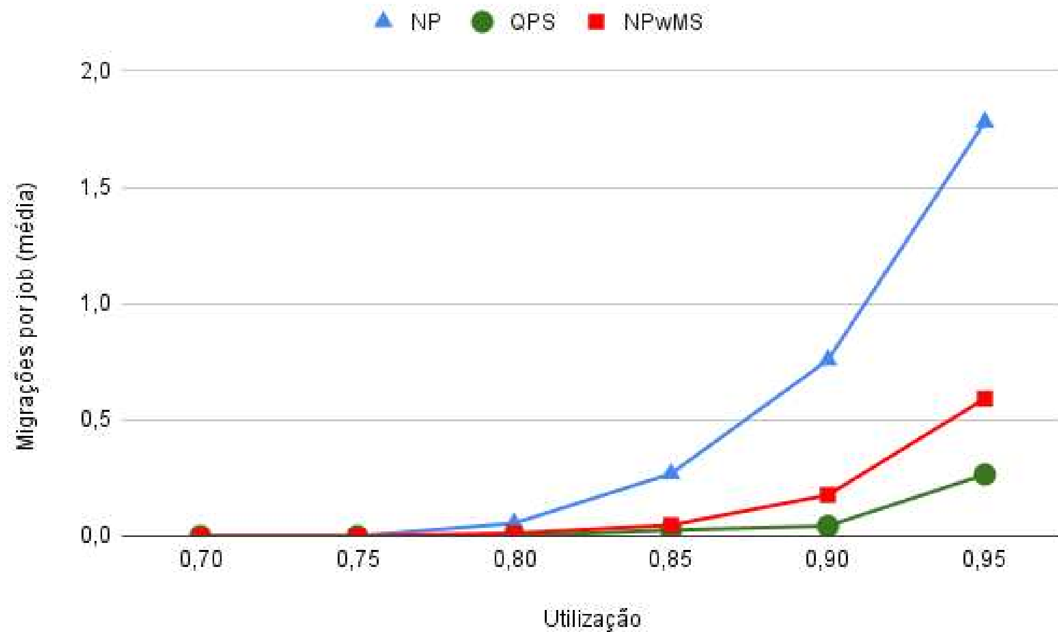
Esses grupos de tarefas foram testados no algoritmo *Notional Processors* original, resultando no gráfico presente na Figura 4 que mostra a porcentagem dos conjuntos escalonados por utilização. Como visto anteriormente, enquanto o algoritmo original vai perdendo a capacidade de escalonar os conjuntos a medida que aumenta a utilização do conjunto, o NPwMS consegue ser bem sucedido no escalonamento de todos os grupos de tarefas, demonstrando ser ótimo em termos de utilização.

Para fazer uma comparação da quantidade de preempções por *job* com o *Notional Processors* original, foi escolhido um sistema com 15 processadores, pois esse alto número de unidades de processamento leva o algoritmo trabalhar com mais intensidade, gerando uma quantidade maior de preempções e migrações. O grupo de tarefas segue as utilizações mencionadas anteriormente, mas, vai até 0,95 já que o *Notional Processors* original não garante escalonamento em quase nenhum conjunto com utilização igual a 1. Pelos resultados presentes no gráfico da Figura 13, percebe-se que o número de migrações por *job* do *Notional Processors* é muito superior aos valores apresentados para o NPwMS e QPS. De todos os algoritmos comparados no gráfico, o QPS se saiu melhor. Porém, o NPwMS teve desempenho praticamente igual até 0,85 de utilização, com uma diferença maior em 0,95 mas não tão distante.

Levando em consideração também as preempções, a Figura 14 mostrou um início melhor do *Notional Processors* original, mas em utilizações mais altas as médias ainda são as piores em comparação com os outros. Nesse gráfico, o QPS é superior ao NPwMS em todas as utilizações, apesar de apresentarem valores próximos.

Na Figura 15 é possível observar que o número de migrações por *job* NPwMS nas utilizações mais baixas está muito próximo de 0. Por esse motivo, foram escolhidos sistemas com utilizações maiores, onde os valores são mais significativos. Isso também justifica a escolha por comparar o NPwMS com o QPS periódico (que é um algoritmo ótimo) já que, em utilizações altas, o *Notional Processors* original não garante escalonamento.

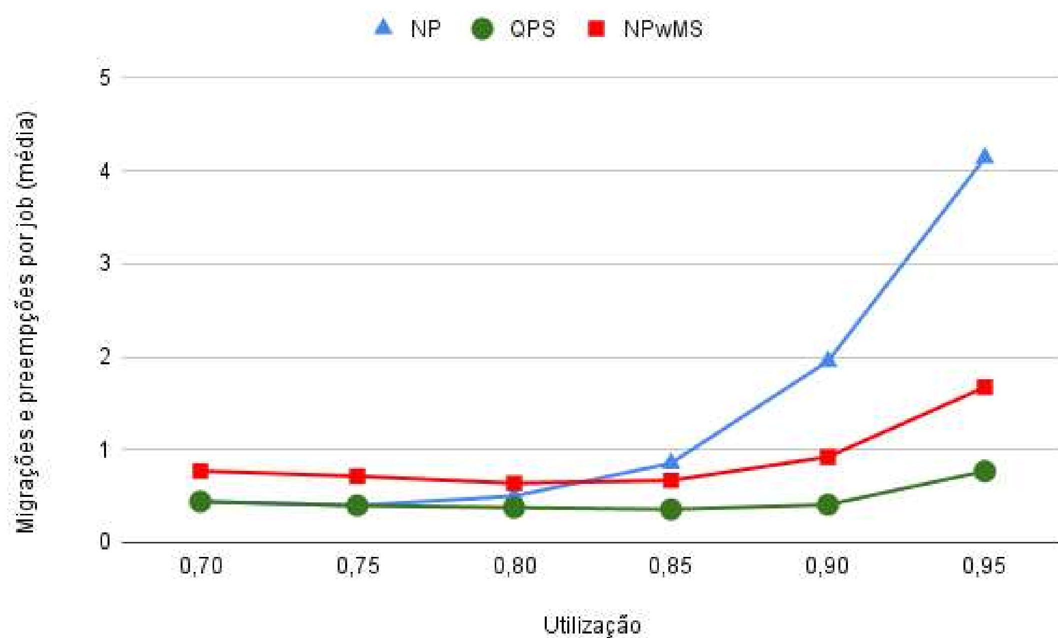
Figura 13 – Migrações por job (média) em quinze processadores



Fonte – O autor

Os resultados apontam que mesmo diferenciando a quantidade de processadores do sistema, o valor médio de migrações por *job* tem o mesmo comportamento. Da utilização 0,70 até a 0,85 as médias dos escalonadores QPS e NPwMS são muito próximas. O NPwMS possui médias maiores que o QPS entre 0,85 e 0,95. Contudo, após esse intervalo, o NPwMS

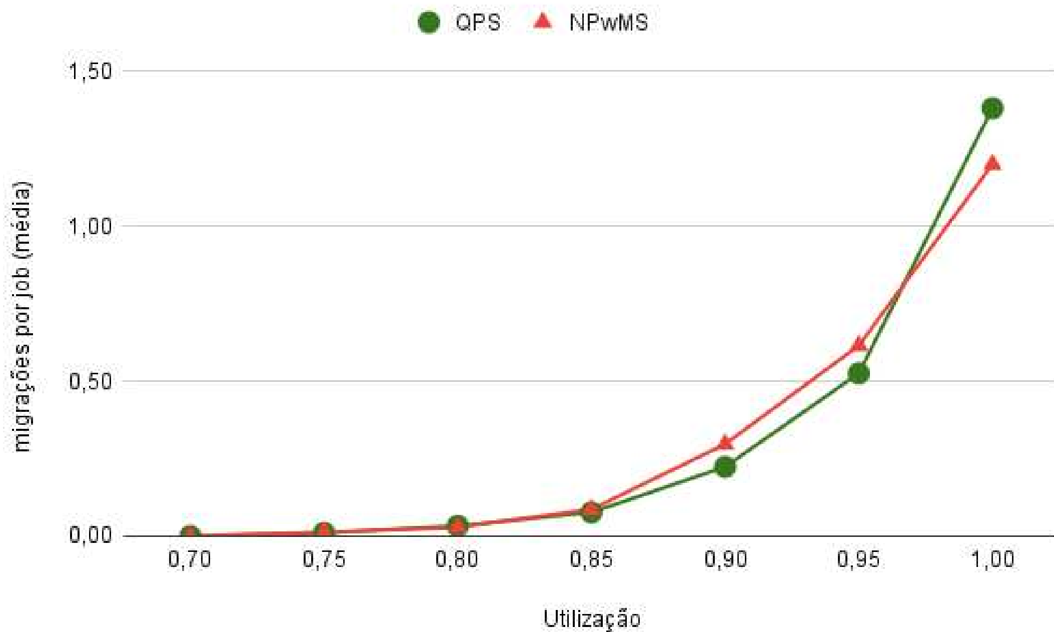
Figura 14 – Migrações e preempções por job (média) em quinze processadores



Fonte – O autor

não tem um crescimento tão acentuado das médias quanto o QPS, que chega a ultrapassá-lo em utilizações maiores. Com base nesses valores, denota-se uma performance melhor do NPwMS em relação ao QPS, em utilizações próximas a 1.

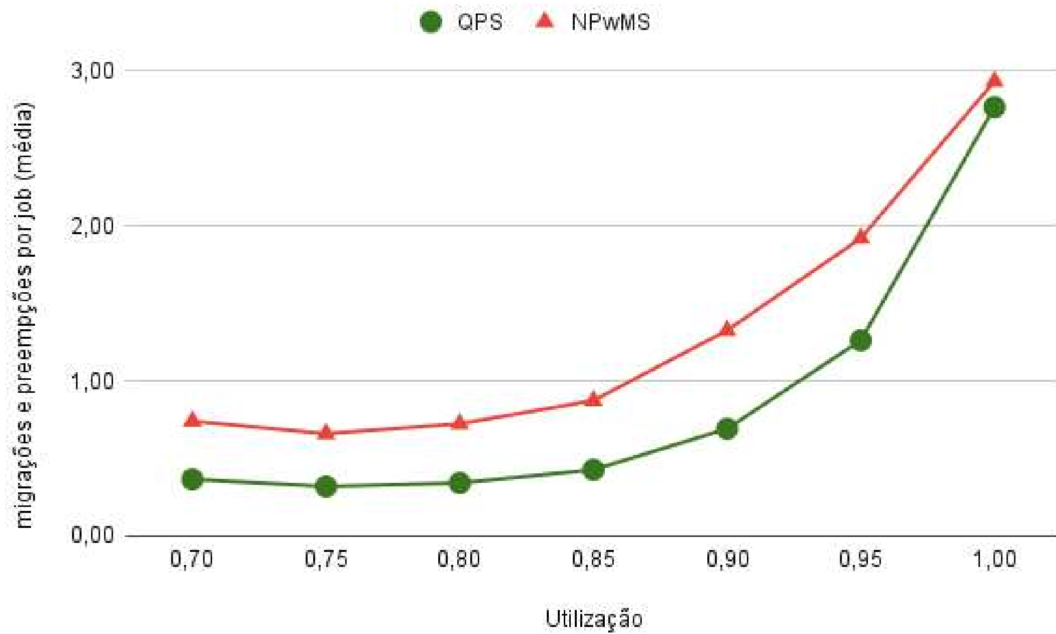
Figura 15 – Migrações por job (média) em quatro processadores



Fonte – O autor

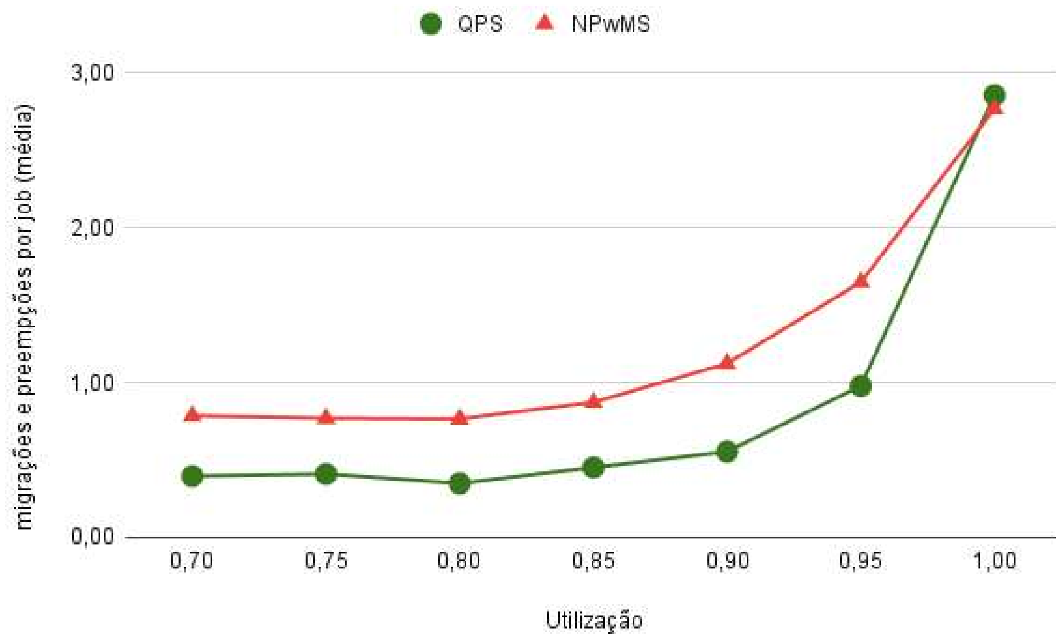
Já os gráficos que levam em consideração a soma das médias de preempções e migrações por *job* (média), mostram uma superioridade do QPS em relação ao NPwMS, tendo uma constância de médias superiores do NPwMS na maior parte do intervalo de utilizações. Mas, em utilizações maiores há uma tendência de igualdade como demonstrado na Figura 17. Entretanto, nos demais sistemas, em utilizações mais próximas a 1, o NPwMS leva vantagem sobre o QPS, como indica a Figura 19. Alguns gráficos de preempções e migrações por *job* exibem, ainda que próximos, valores maiores para utilização 0,70 do que 0,75. Nas figuras 16 e 18, é possível visualizar esse comportamento inesperado.

Figura 16 – Migrações e preempções por job (média) em quatro processadores



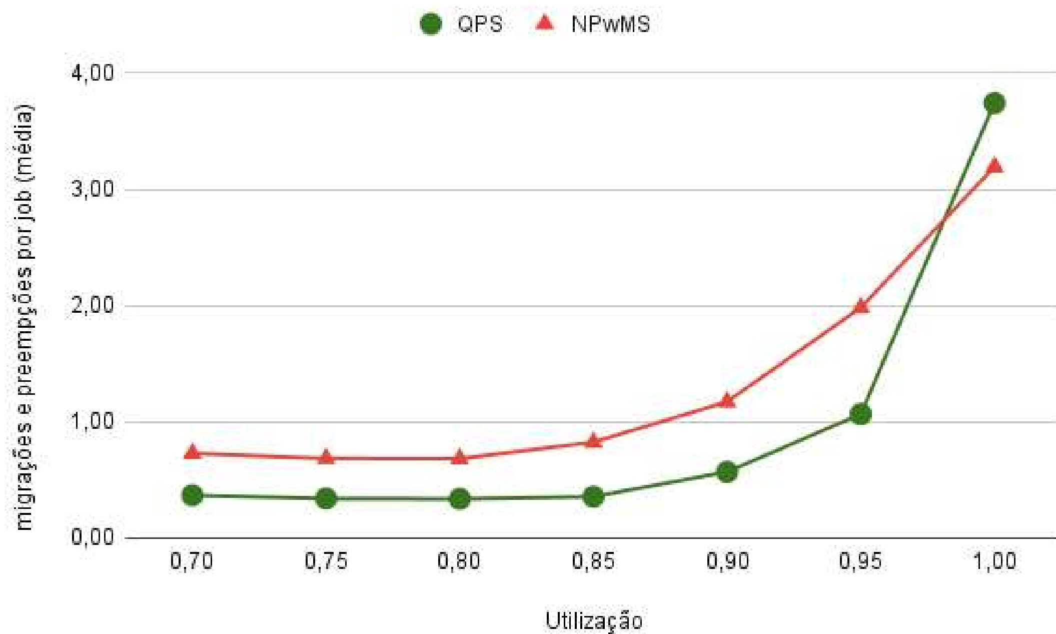
Fonte – O autor

Figura 17 – Migrações e preempções por job (média) em cinco processadores



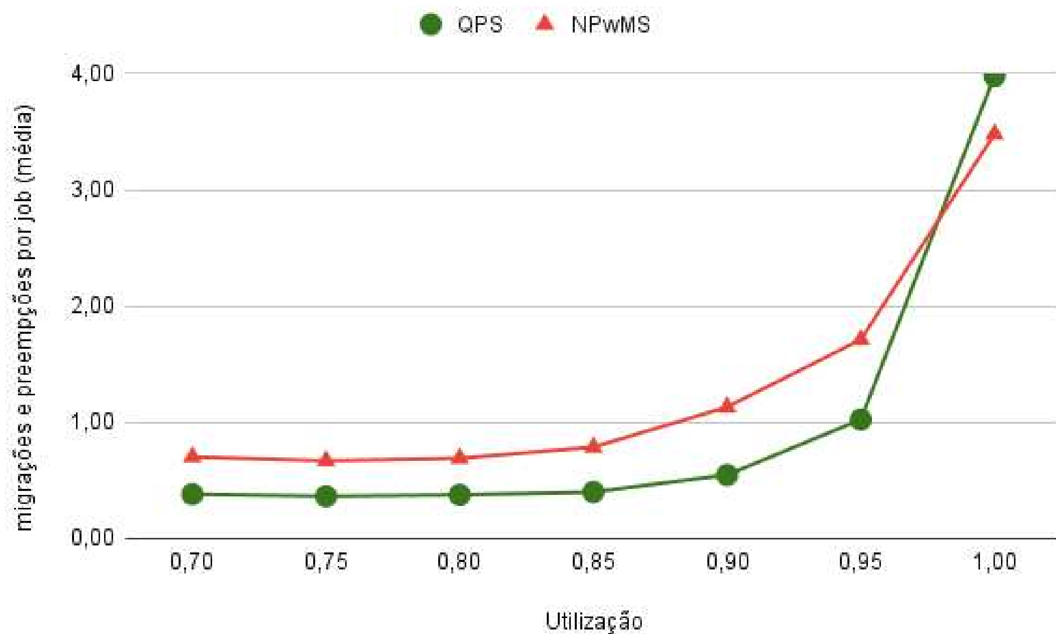
Fonte – O autor

Figura 18 – Migrações e preempções por job (média) em oito processadores



Fonte – O autor

Figura 19 – Migrações e preempções por job (média) em dez processadores



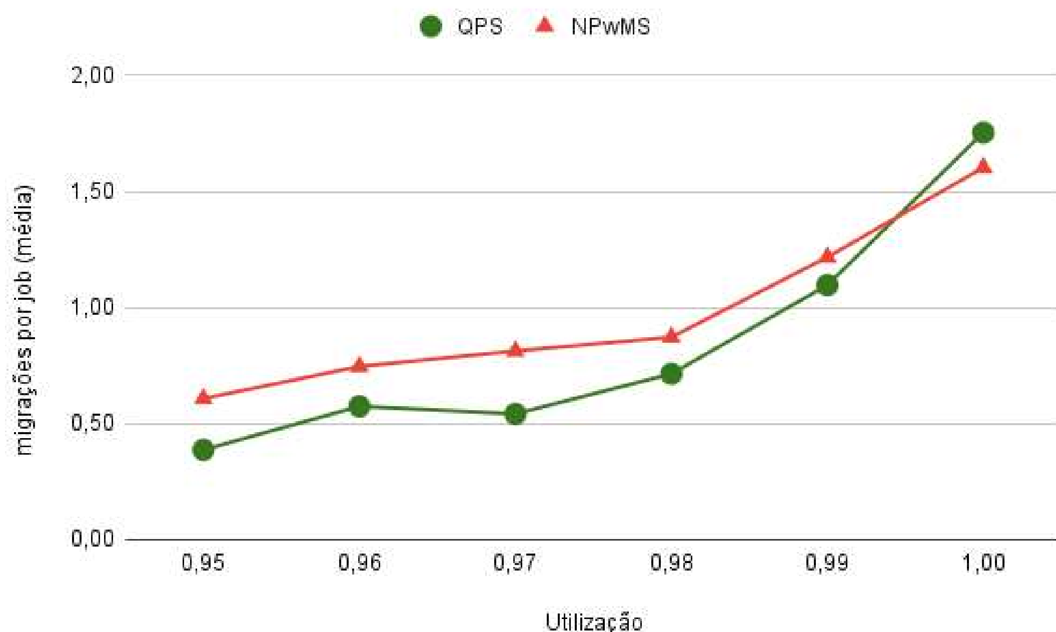
Fonte – O autor

Devido ao comportamento vantajoso do NPwMS em relação ao QPS em utilizações mais altas, foi produzida outra rodada de testes a fim de confirmar essa análise. Foram gerados conjuntos de tarefas com utilizações de 0,95 até 1, incrementando 0,01. Para cada utilização, foi

produzido 100 grupos de tarefas, totalizando 600. O intervalo da quantidade de processadores se repete. Os gráficos de migrações por *job* resultantes desses testes mostram que o NPwMS possui médias maiores, mas bem próximas do QPS na maioria das utilizações. O NPwMS leva vantagem nas coleções de utilização 1. É possível notar também que alguns sistemas possuem diferenças maiores em 1 como aponta as Figura 21 enquanto outros tem valores mais próximos como apresentados nas Figura 20.

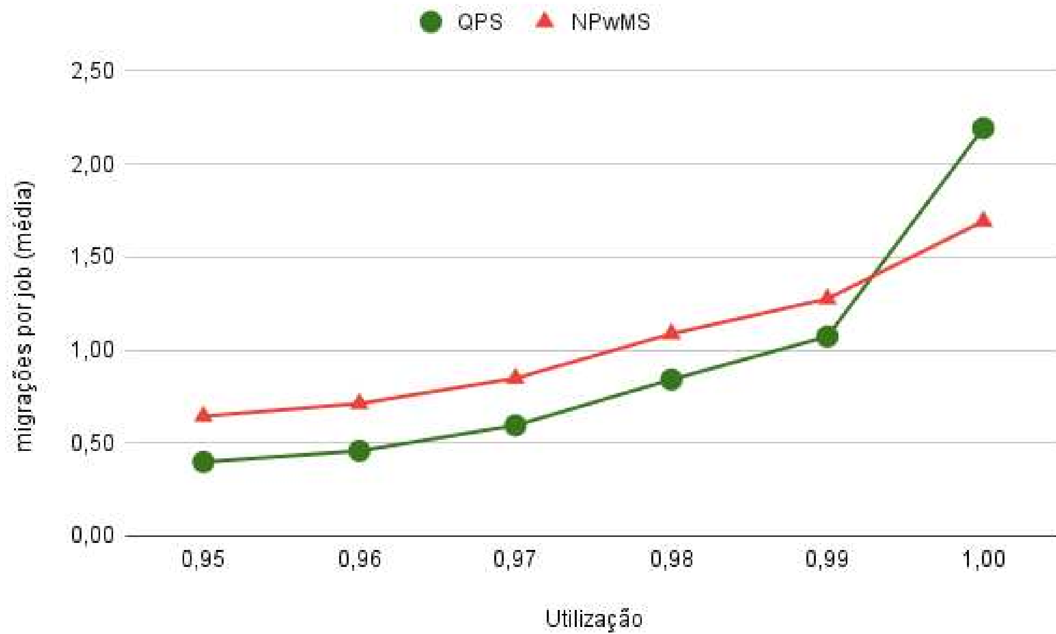
Os números que representam a soma das preempções e migrações, considerando esse cenário de alta utilização, aponta para uma proximidade maior das médias em utilização igual a 1. Ao longo das utilizações, o NPwMS possui valores maiores, o que denota uma vantagem do QPS nessa avaliação. Em alguns gráficos como o da Figura 22 exibe médias maiores do QPS em 1, mas a maioria dos testes mostram médias com pouca diferença.

Figura 20 – Migrações por job (média) em oito processadores, para utilizações maiores



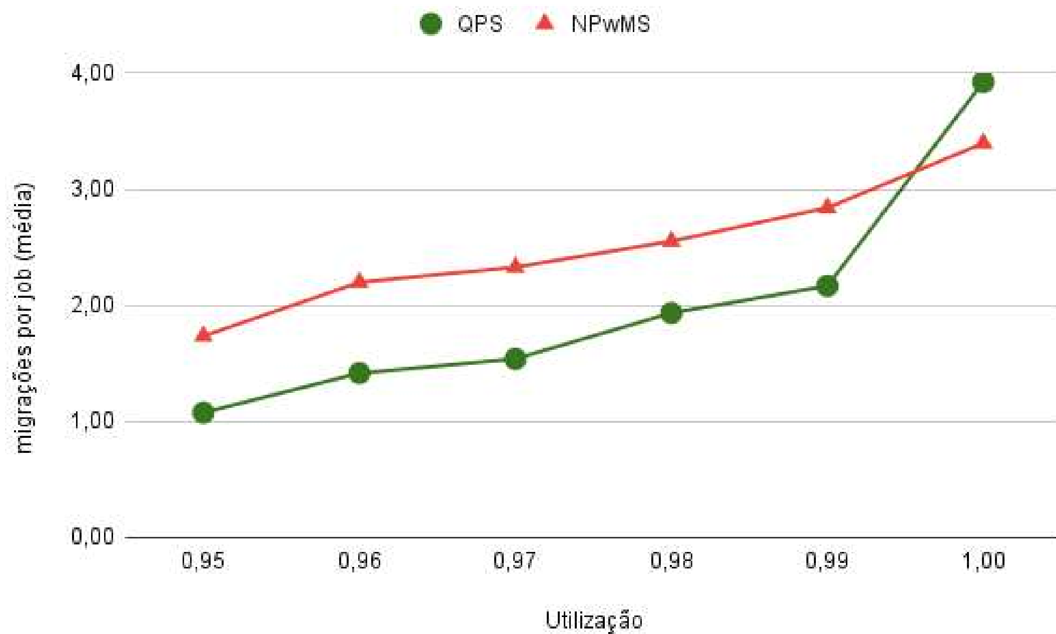
Fonte – O autor

Figura 21 – Migrações por job (média) em nove processadores, para utilizações maiores



Fonte – O autor

Figura 22 – Migrações e preempções por job (média) em dez processadores, para utilizações maiores



Fonte – O autor

8 CONSIDERAÇÕES FINAIS

A utilização de servidores mestre-escravos no algoritmo semi-particionado *Notional Processors* trouxe melhorias, principalmente no limite de utilização. A ociosidade típica da estratégia semi-particionada, que é imposta pelos mecanismos de sincronização dos algoritmos, foi removida com o auxílio desses servidores. Isso é perceptível através dos resultados mostrados no Capítulo 7, que foram obtidos a partir da simulação do escalonamento. Outro aperfeiçoamento foi o dinamismo dado ao funcionamento dos processadores lógicos. Ao invés de fazer o mapeamento das áreas ociosas dos processadores físicos de forma estática, os servidores possibilitaram a indicação dessas áreas de forma dinâmica.

Enquanto o *Notional Processors* original não garante o escalonamento de conjuntos com utilizações maiores, o algoritmo com os servidores mestre-escravos escalonou todos. Sendo assim, a adição dos servidores tornou o *Notional Processors* um algoritmo ótimo, no que diz respeito à utilização do sistema. Além disso, notou-se também uma melhora significativa na quantidade de migrações por *job*, como demonstrado pela Figura 13.

Em comparação com o QPS periódico a implementação proposta nesse trabalho, denominada então como NPwMS, teve performance bem próxima, porém, um pouco pior na maioria das utilizações testadas. Entretanto, quando o grupo de tarefas possui uma alta utilização do sistema, num intervalo que inicia a partir de um pouco mais que 99% até 100%, o NPwMS teve melhores resultados.

Desta maneira, é possível concluir a adição dos servidores mestre-escravos numa estratégia semi-particionada foi positiva, trazendo ganhos na escalonabilidade da implementação e, em termos de performance, o algoritmo também mostrou-se eficiente tendo uma grande melhora na quantidade de migrações em relação ao *Notional Processors* original. A obtenção de resultados superiores ao QPS em conjuntos próximos do 100% de utilização é um indicativo disso. Tendo isso em vista, a análise dos resultados permitiu validar o NPwMS como uma implementação que trouxe melhorias a um algoritmo semi-particionado, através da inclusão dos servidores mestre-escravos, sendo essa a principal contribuição deste trabalho. Nota-se também que os servidores mestre-escravos mesmo sendo um mecanismo de sincronização proveniente

do QPS, comportou-se muito bem ao ser adicionado a outro algoritmo. Para os escalonadores que a sincronização recorrem à inserção de ociosidade aos processadores, os servidores passam a ser uma alternativa viável, pois não geram perdas de capacidade computacional.

Durante a execução deste trabalho, foram identificadas algumas possibilidades de trabalhos futuros como: (i) A investigação do que leva a ocorrência de valores maiores para preempções e migrações, mesmo numa utilização menor; (ii) Utilização de outro modelo de tarefas, por exemplo, tarefas esporádicas.

REFERÊNCIAS

ANDERSSON, B. Semi-partitioned Multiprocessor Scheduling. In: TIAN, Y.-C.; LEVY, D. C. (Ed.). **Handbook of Real-Time Computing**. Singapore: Springer Singapore, 2019. p. 1–17. ISBN 978-981-4585-87-3. Disponível em: <http://link.springer.com/10.1007/978-981-4585-87-3_2-1>.

ANDERSSON, B.; BARUAH, S.; JONSSON, J. Static-priority scheduling on multiprocessors. In: **Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)**. London, UK: IEEE Comput. Soc, 2001. p. 193–202. ISBN 978-0-7695-1420-8. Disponível em: <http://ieeexplore.ieee.org/document/990610/>.

ANDERSSON, B.; BLETSAS, K. Sporadic multiprocessor scheduling with few preemptions. In: **2008 Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2008. p. 243–252.

ANDERSSON, B.; TOVAR, E. Multiprocessor Scheduling with Few Preemptions. In: **12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)**. Sydney, Australia: IEEE, 2006. p. 322–334. ISBN 978-0-7695-2676-8. Disponível em: <http://ieeexplore.ieee.org/document/1691331/>.

BARUAH, S. K.; COHEN, N. K.; PLAXTON, C. G.; VARVEL, D. A. Proportionate progress: A notion of fairness in resource allocation. **Algorithmica**, v. 15, n. 6, p. 600–625, jun. 1996. ISSN 1432-0541. Disponível em: <https://doi.org/10.1007/BF01940883>.

BLETSAS, K.; ANDERSSON, B. Notional Processors: An Approach for Multiprocessor Scheduling. In: **2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium**. San Francisco, CA, USA: IEEE, 2009. p. 3–12. ISBN 978-0-7695-3636-1. Disponível em: <http://ieeexplore.ieee.org/document/4840562/>.

BRANDENBURG, B. B.; GUL, M. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In: **2016 IEEE Real-Time Systems Symposium (RTSS)**. Porto, Portugal: IEEE, 2016. p. 99–110. ISBN 978-1-5090-5303-2. Disponível em: <http://ieeexplore.ieee.org/document/7809847/>.

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 43, n. 4, out. 2011. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/1978802.1978814>.

FARINES, J.-M.; FRAGA, J. d. S.; OLIVEIRA, R. d. Sistemas de tempo real. **Escola de Computação**, v. 2000, p. 201, 2000.

JAIN, R. **The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling**. [S.l.]: John Wiley & Sons, 1990.

KATO, S.; YAMASAKI, N. Portioned EDF-based scheduling on multiprocessors. In: **Proceedings of the 7th ACM international conference on Embedded software - EMSOFT '08**. Atlanta, GA, USA: ACM Press, 2008. p. 139. ISBN 978-1-60558-468-3. Disponível em: <http://portal.acm.org/citation.cfm?doid=1450058.1450078>.

KATO, S.; YAMASAKI, N. Semi-partitioned fixed-priority scheduling on multiprocessors. In: **2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium**. [S.l.: s.n.], 2009. p. 23–32.

KOPETZ, H. Real-time scheduling. In: ____. **Real-Time Systems: Design Principles for Distributed Embedded Applications**. Boston, MA: Springer US, 1997. p. 227–243. ISBN 978-0-306-47055-4. Disponível em: <https://doi.org/10.1007/0-306-47055-1_11>.

LIU, C. L.; LAYLAND, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. **Journal of the ACM**, v. 20, n. 1, p. 46–61, jan. 1973. ISSN 0004-5411, 1557-735X. Disponível em: <https://dl.acm.org/doi/10.1145/321738.321743>.

MASSA, E.; LIMA, G.; REGNIER, P.; LEVIN, G.; BRANDT, S. OUTSTANDING PAPER: Optimal and Adaptive Multiprocessor Real-Time Scheduling: The Quasi-Partitioning Approach. In: **2014 26th Euromicro Conference on Real-Time Systems**. Madrid, Spain: IEEE, 2014. p. 291–300. Disponível em: <http://ieeexplore.ieee.org/document/6932610/>.

MISRA, J. Distributed discrete-event simulation. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 1, p. 39–65, mar. 1986. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/6462.6485>.

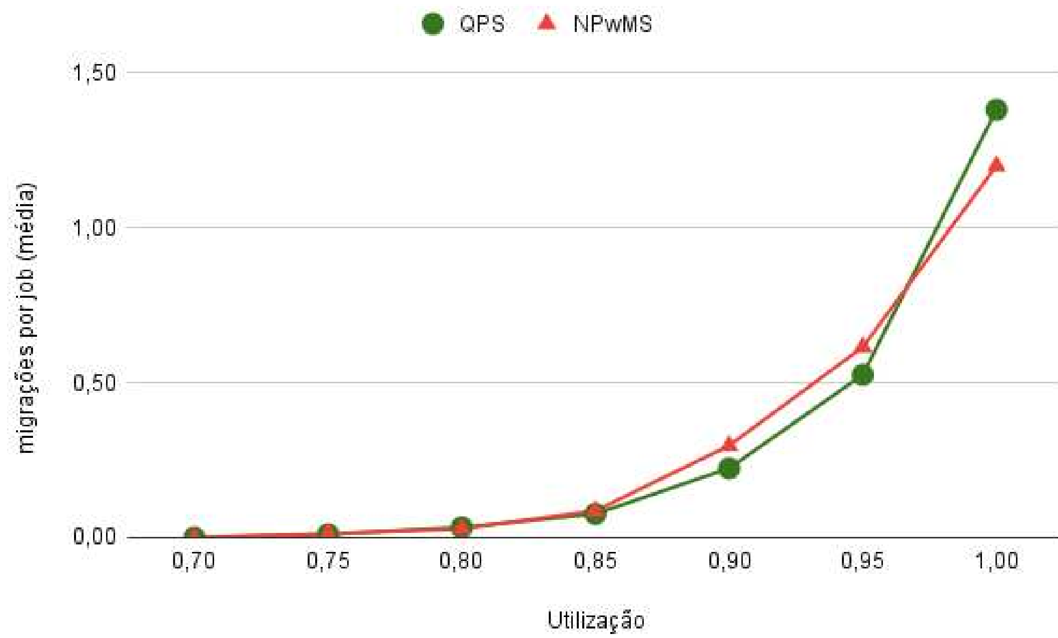
OLIVEIRA, R. de. **Fundamentos dos Sistemas de Tempo Real: Segunda Edição**. Amazon Digital Services LLC - KDP Print US, 2020. ISBN 9798681424635. Disponível em: <https://books.google.com.br/books?id=7SjZzQEACAAJ>.

SOUSA, P. B.; SOUTO, P.; TOVAR, E.; BLETSAS, K. The Carousel-EDF scheduling algorithm for multiprocessor systems. In: **2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications**. Taipei, Taiwan: IEEE, 2013. p. 12–21. ISBN 978-1-4799-0850-9. Disponível em: <http://ieeexplore.ieee.org/document/6732199/>.

APÊNDICES

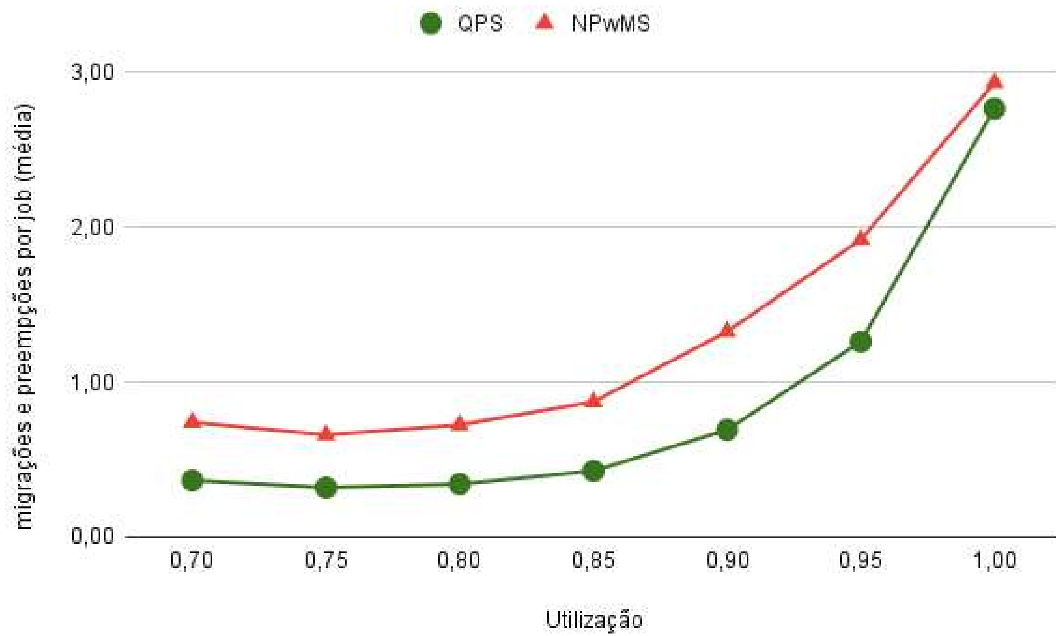
APÊNDICE A – Gráficos dos resultados

Figura 23 – Migrações por *job* (média) em quatro processadores



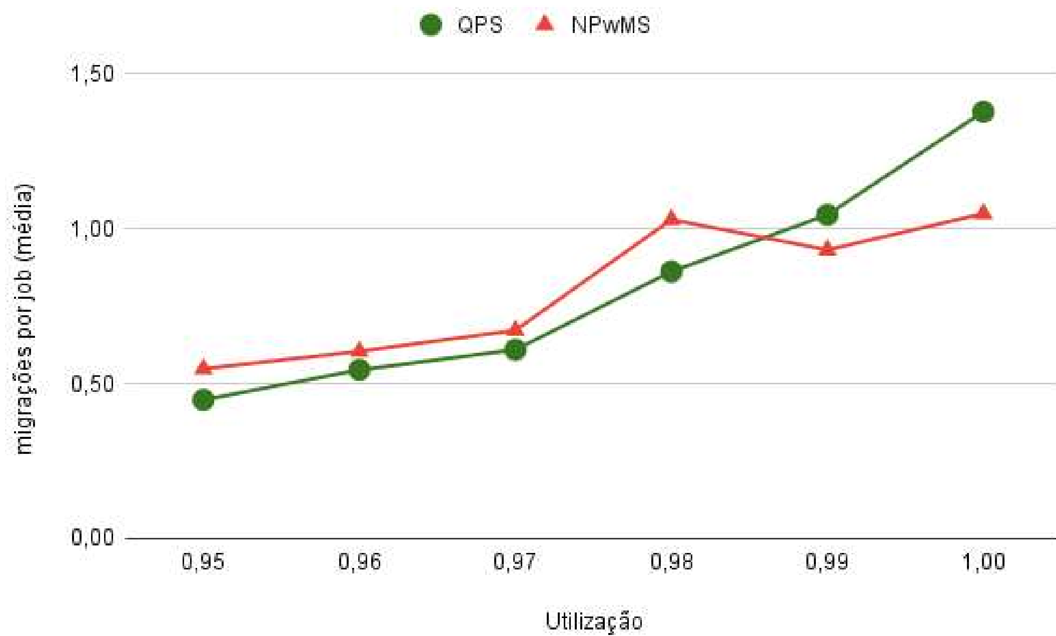
Fonte – O autor

Figura 24 – Migrações e preempções por *job* (média) em quatro processadores



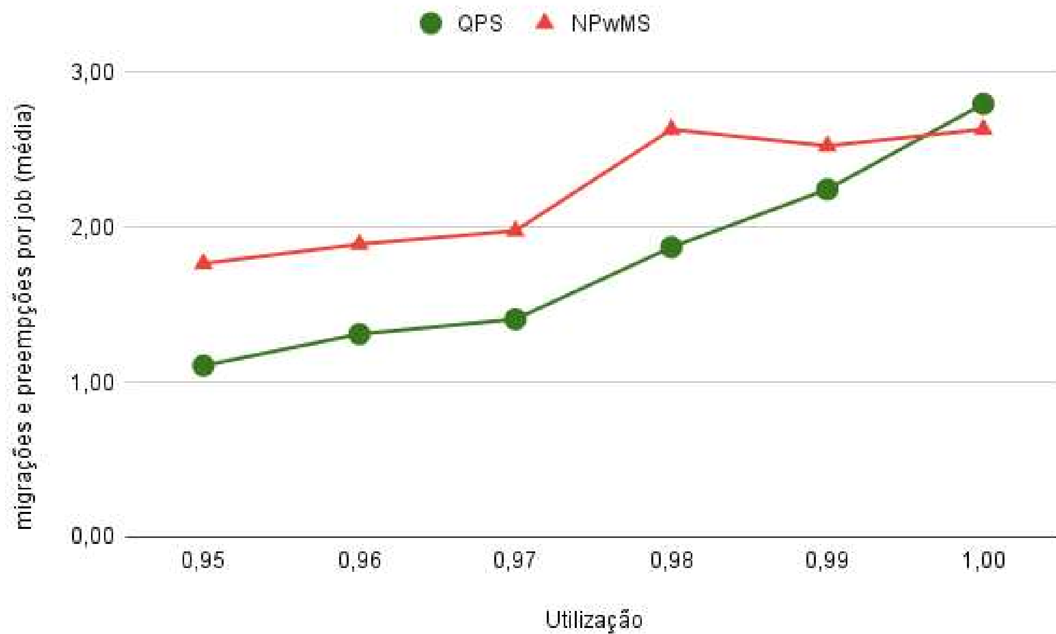
Fonte – O autor

Figura 25 – Migrações por *job* (média) em quatro processadores e utilizações maiores



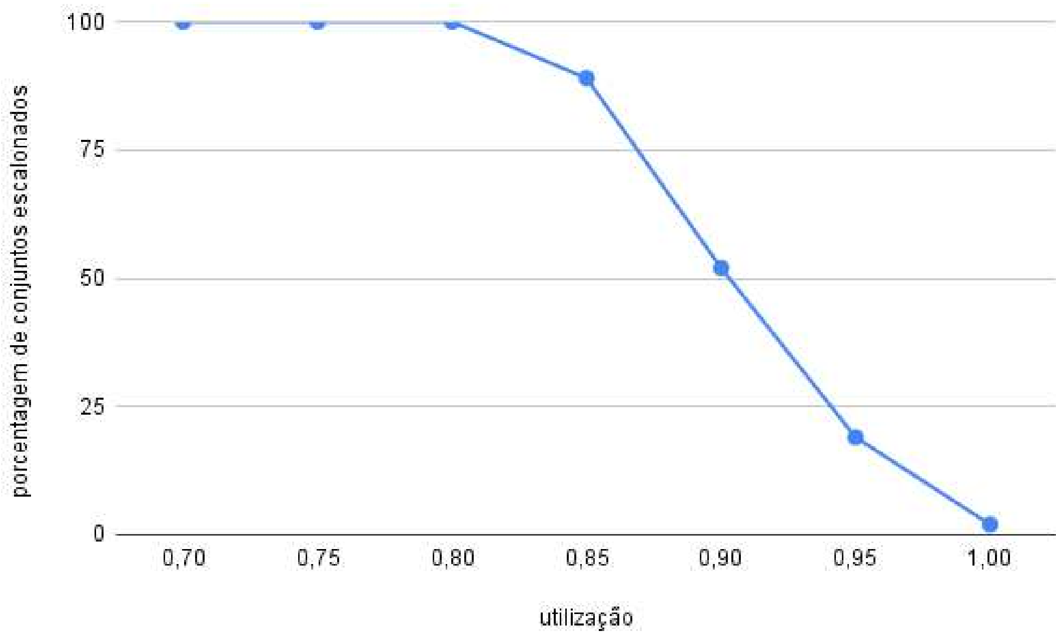
Fonte – O autor

Figura 26 – Migrações e preempções por *job* (média) em quatro processadores e utilizações maiores



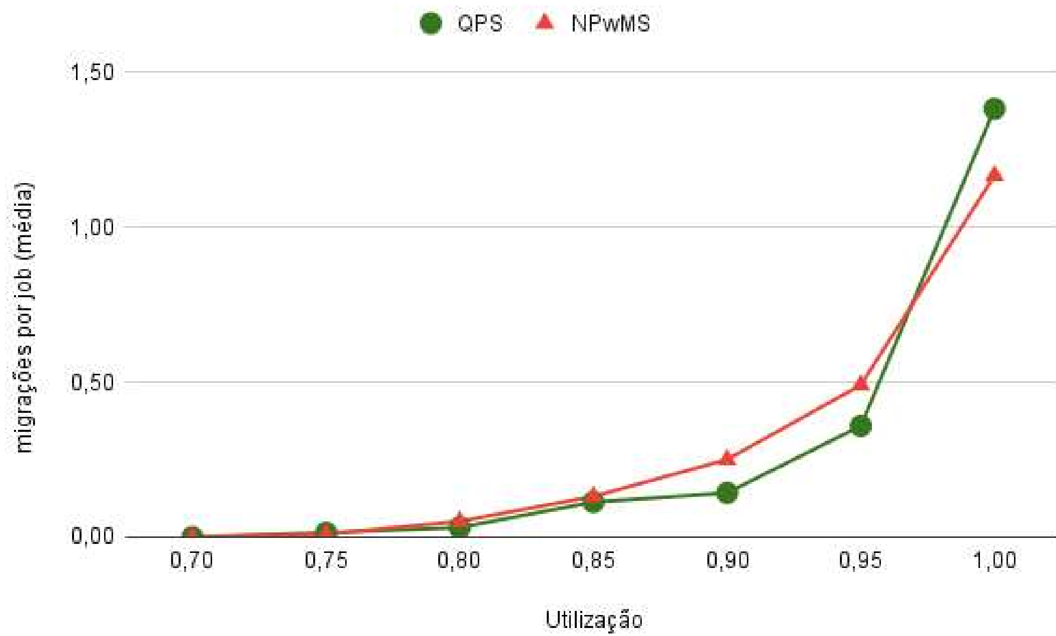
Fonte – O autor

Figura 27 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em quatro processadores



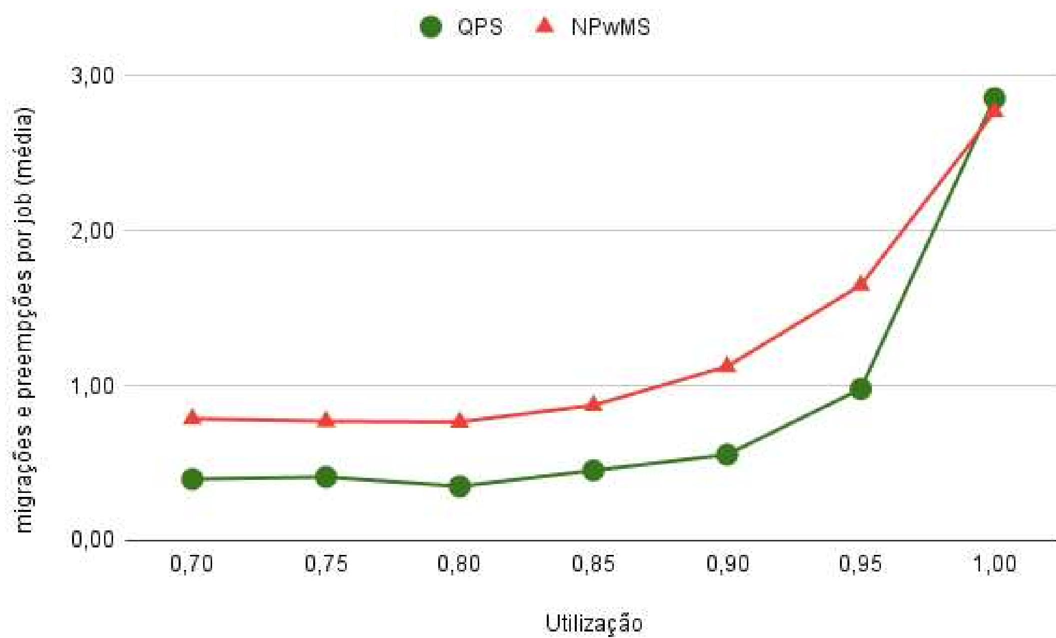
Fonte – O autor

Figura 28 – Migrações por *job* (média) em cinco processadores



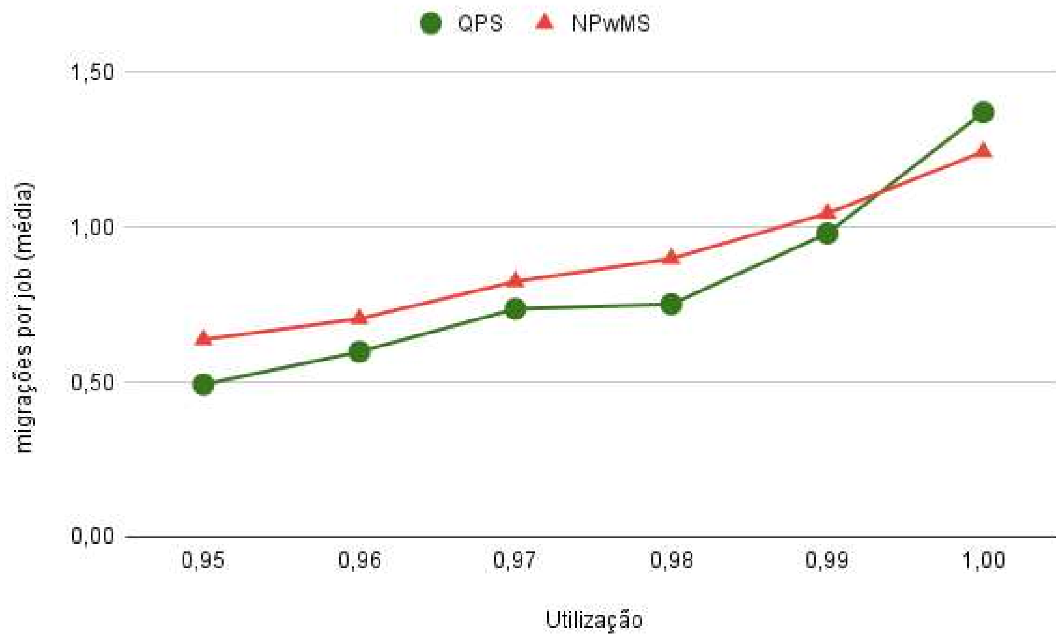
Fonte – O autor

Figura 29 – Migrações e preempções por *job* (média) em cinco processadores



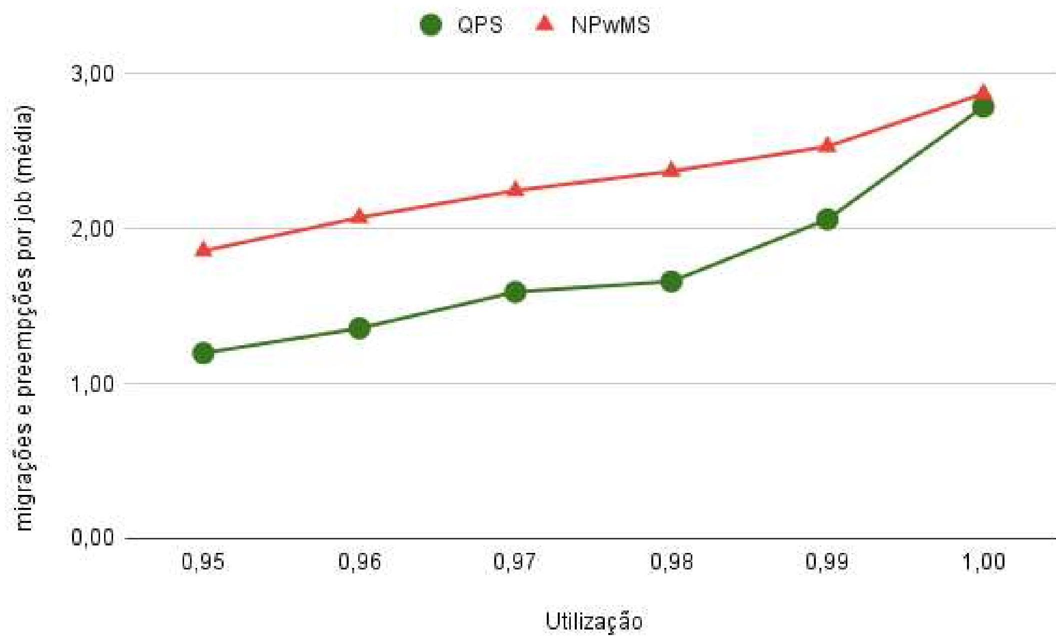
Fonte – O autor

Figura 30 – Migrações por *job* (média) em cinco processadores e utilizações maiores



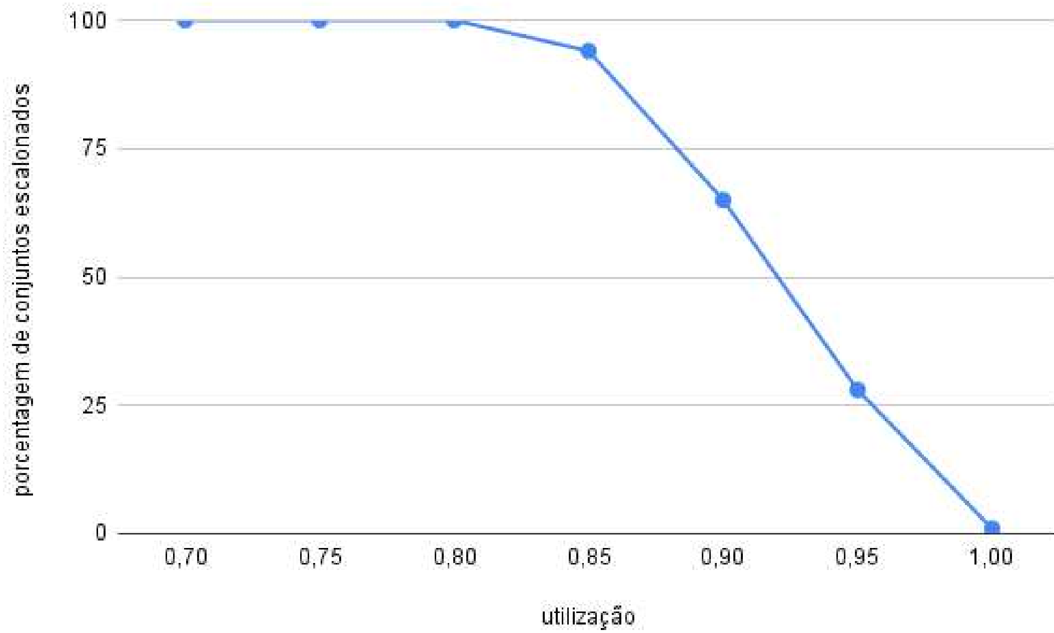
Fonte – O autor

Figura 31 – Migrações e preempções por *job* (média) em cinco processadores e utilizações maiores



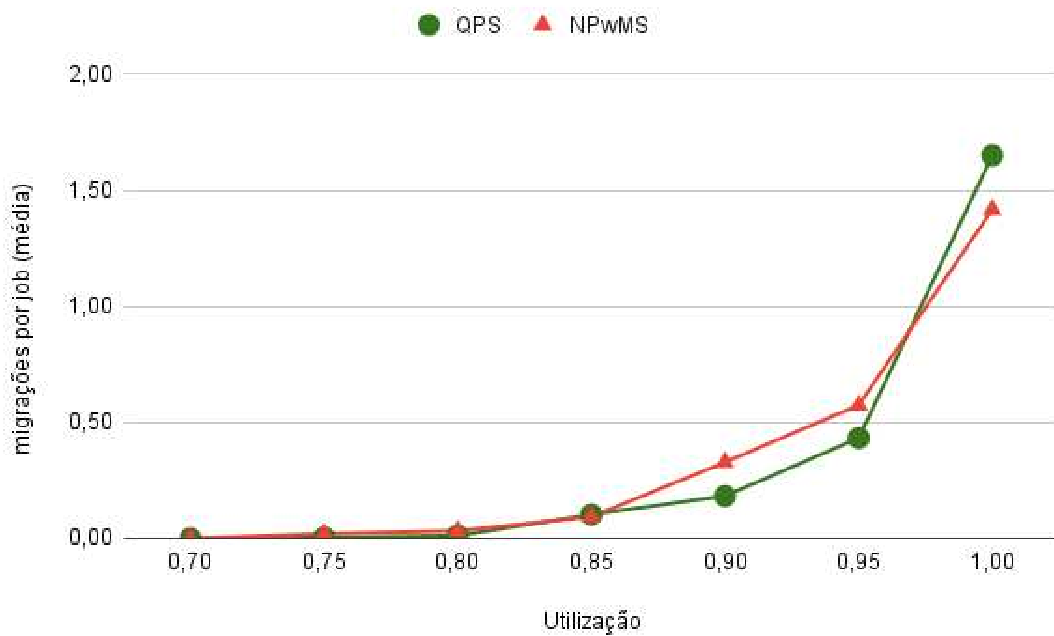
Fonte – O autor

Figura 32 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em cinco processadores



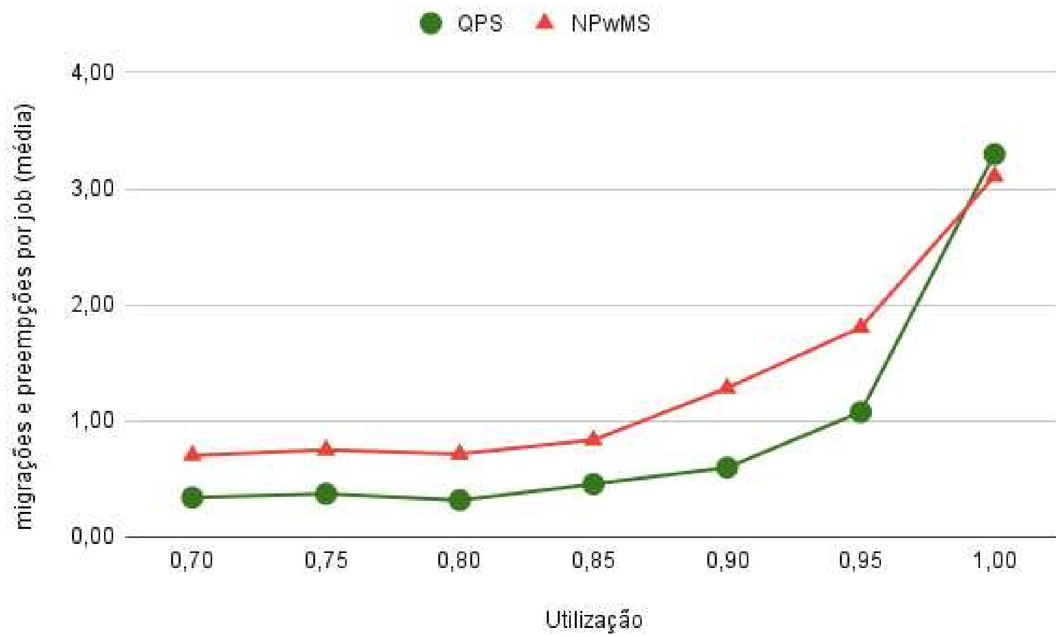
Fonte – O autor

Figura 33 – Migrações por *job* (média) em seis processadores



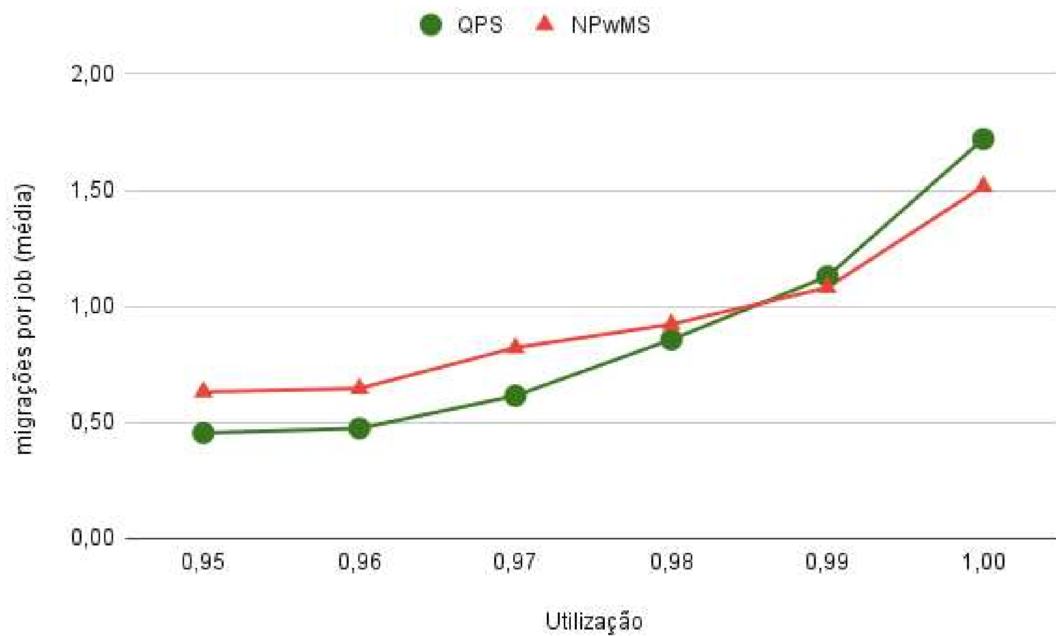
Fonte – O autor

Figura 34 – Migrações e preempções por *job* (média) em seis processadores



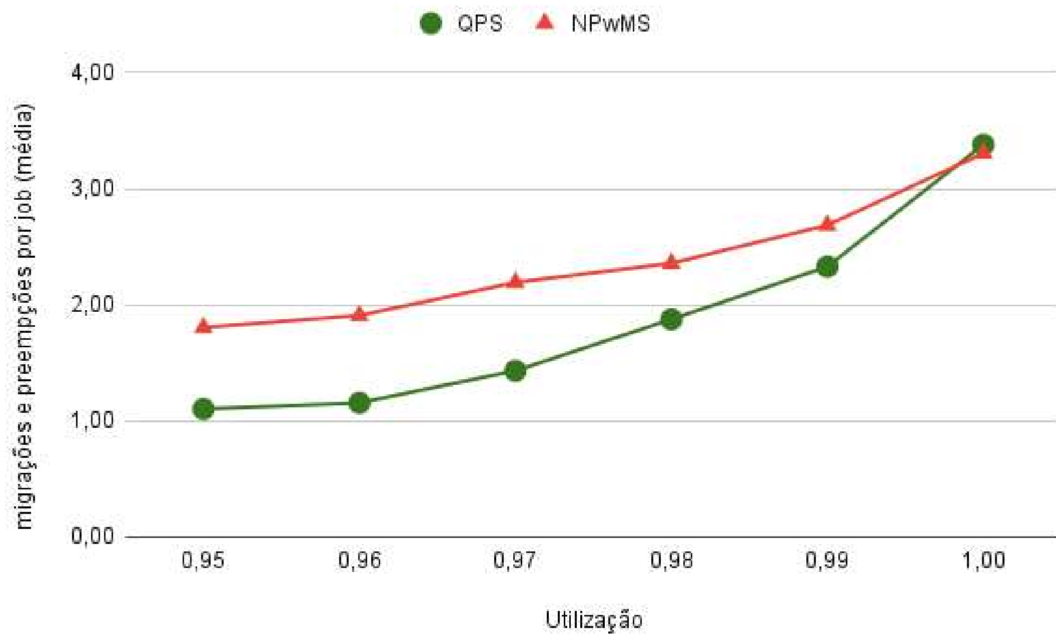
Fonte – O autor

Figura 35 – Migrações por *job* (média) em seis processadores e utilizações maiores



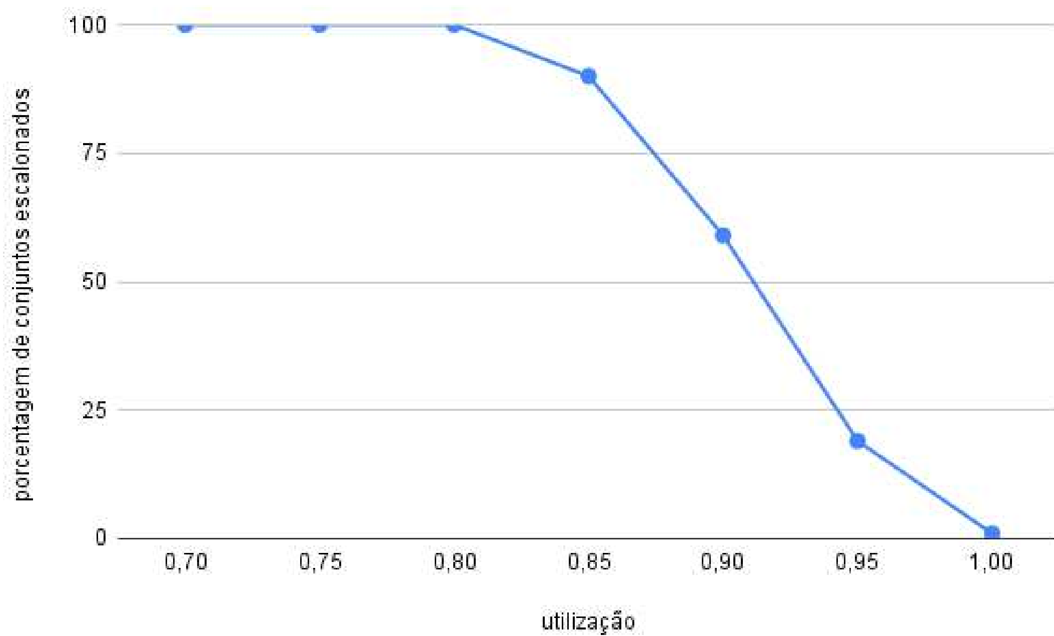
Fonte – O autor

Figura 36 – Migrações e preempções por *job* (média) em seis processadores e utilizações maiores



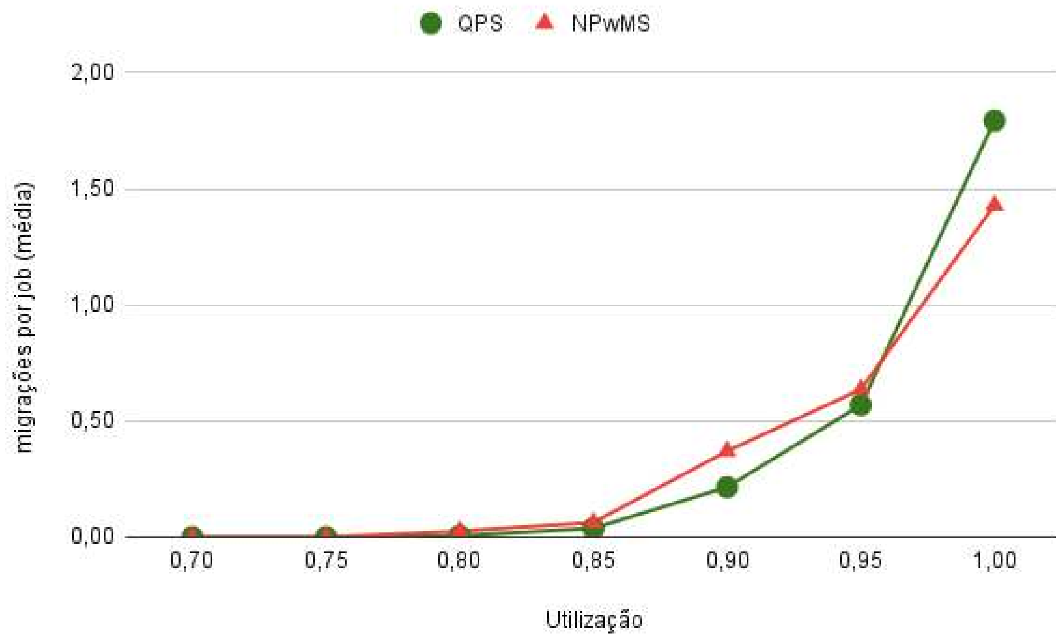
Fonte – O autor

Figura 37 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em seis processadores



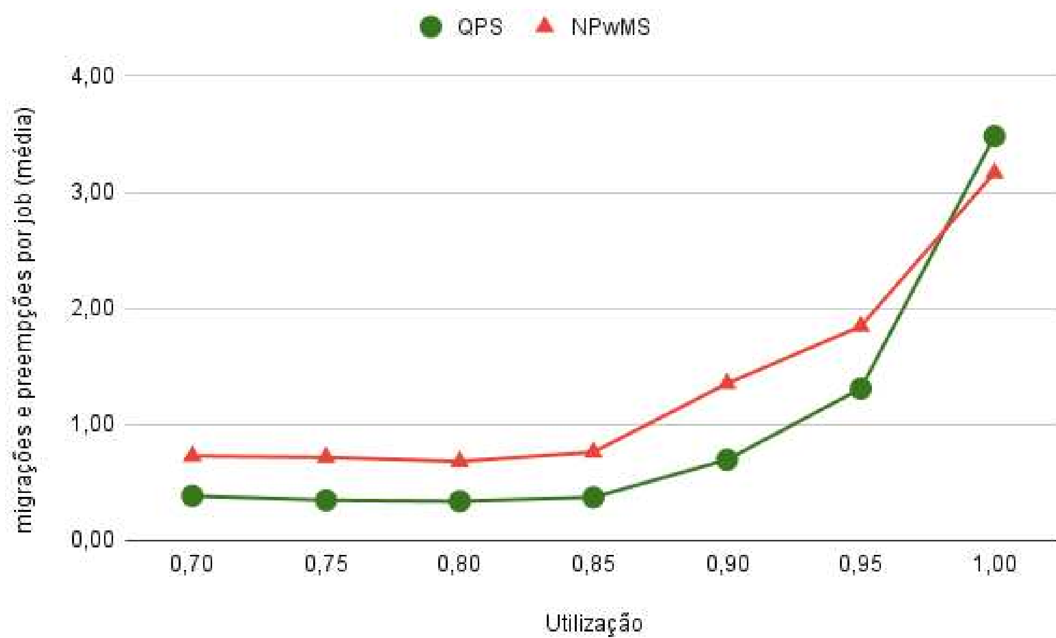
Fonte – O autor

Figura 38 – Migrações por *job* (média) em sete processadores



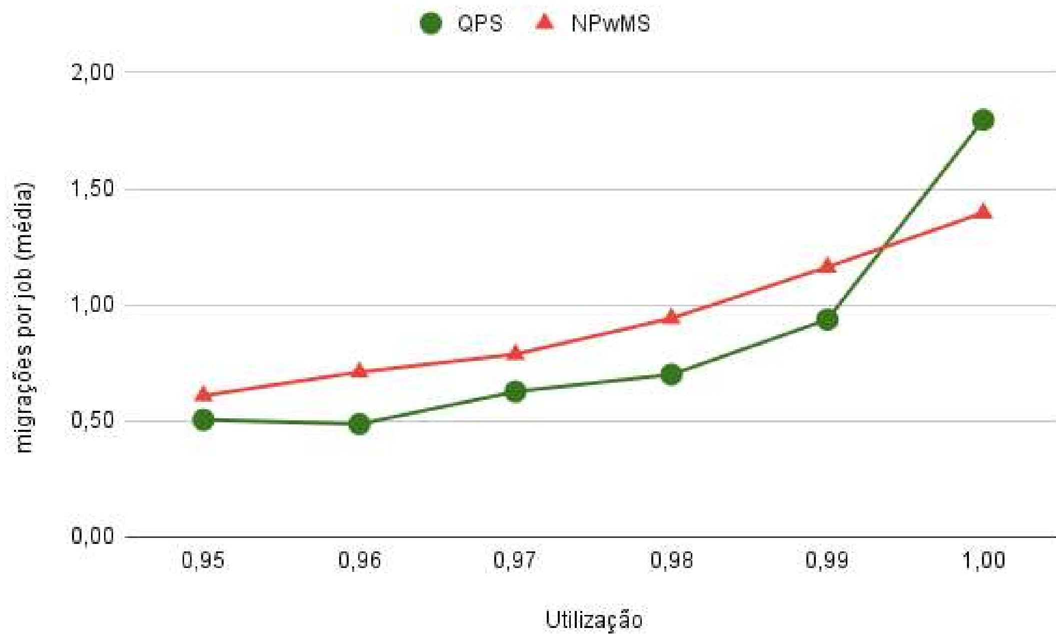
Fonte – O autor

Figura 39 – Migrações e preempções por *job* (média) em sete processadores



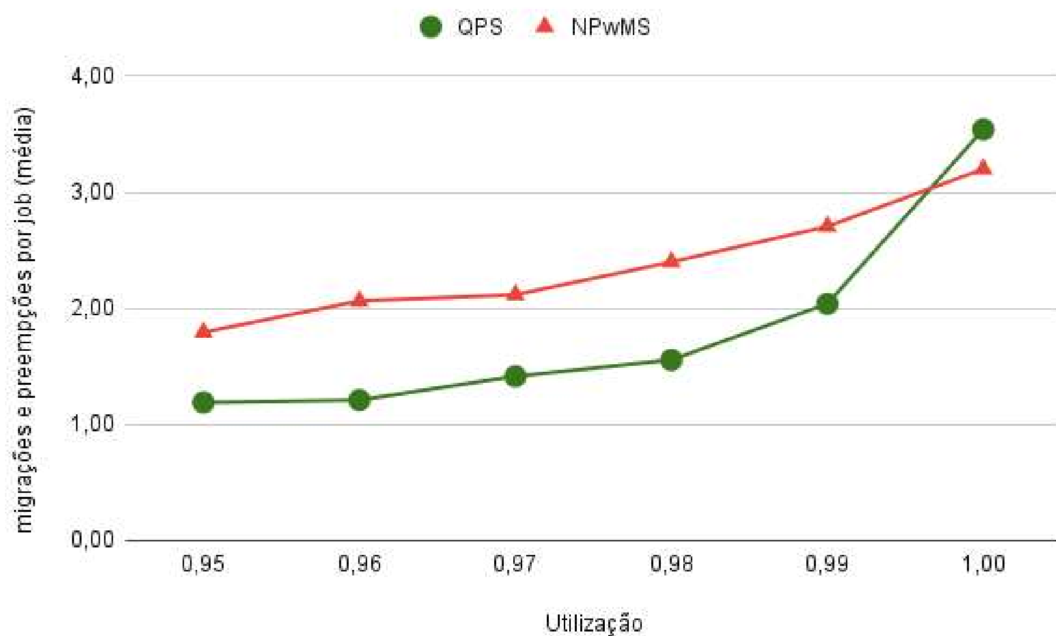
Fonte – O autor

Figura 40 – Migrações por *job* (média) em sete processadores e utilizações maiores



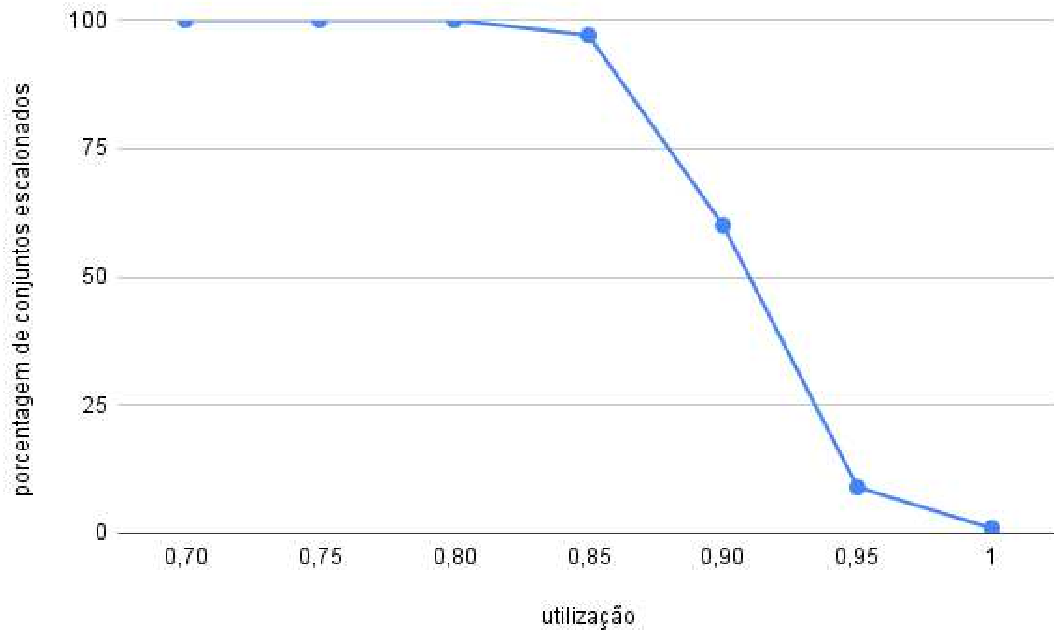
Fonte – O autor

Figura 41 – Migrações e preempções por *job* (média) em sete processadores e utilizações maiores



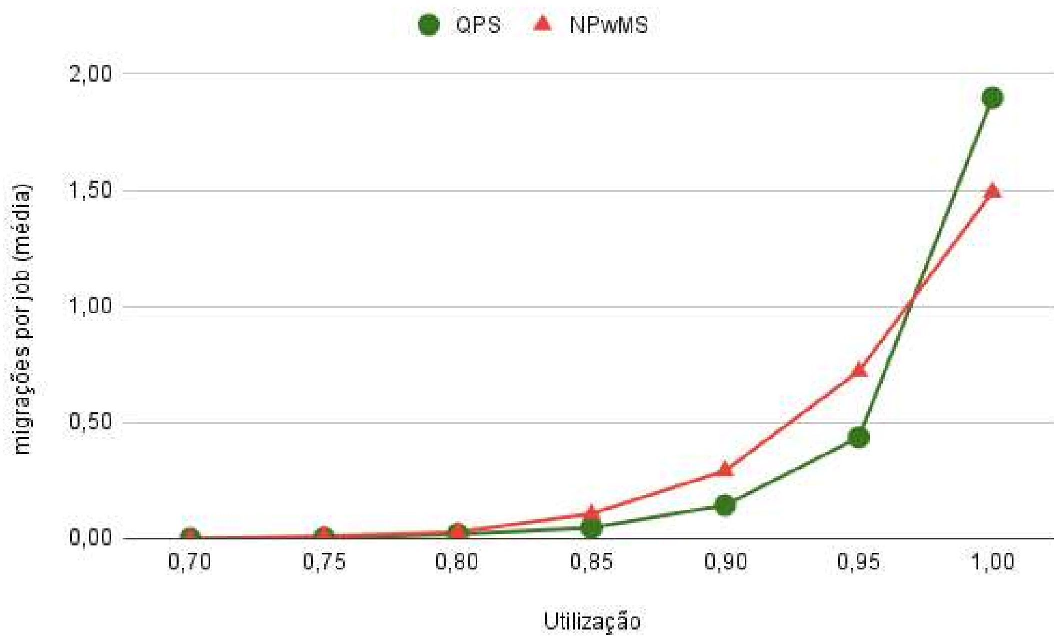
Fonte – O autor

Figura 42 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em sete processadores



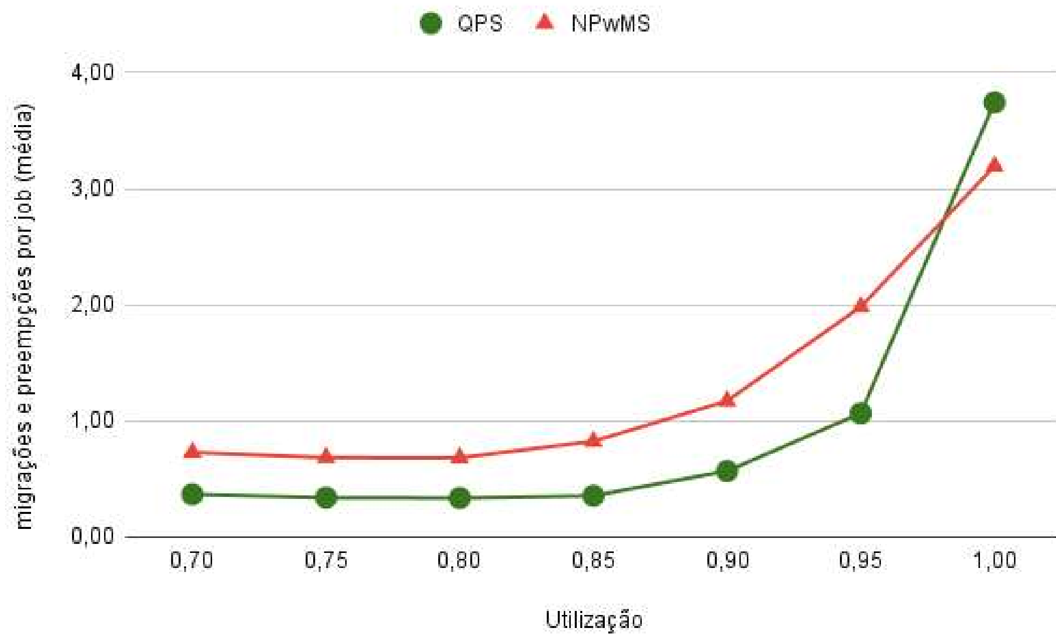
Fonte – O autor

Figura 43 – Migrações por *job* (média) em oito processadores



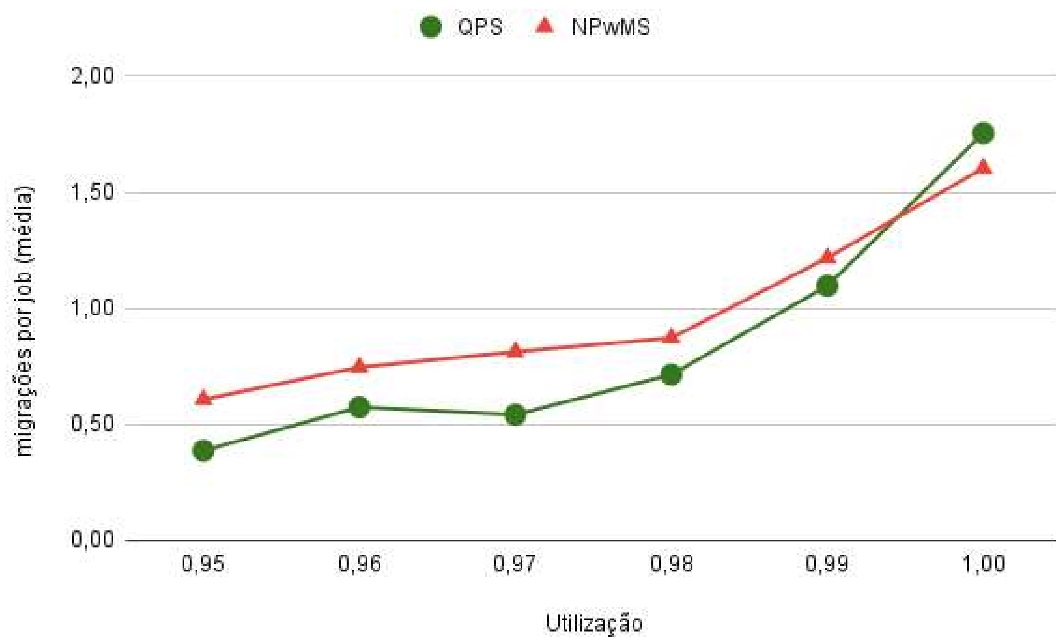
Fonte – O autor

Figura 44 – Migrações e preempções por *job* (média) em oito processadores



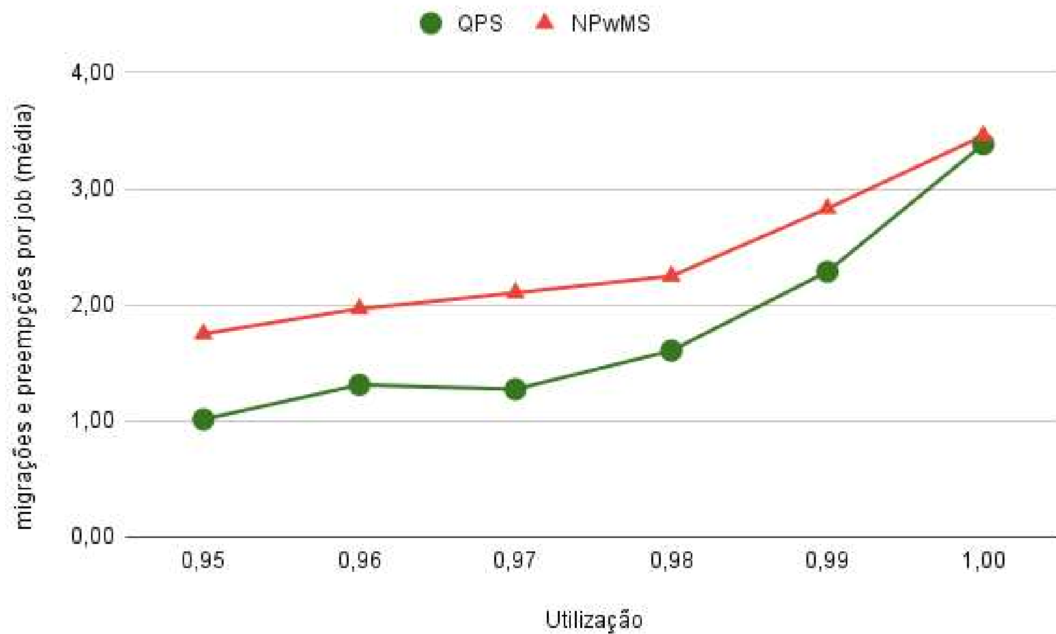
Fonte – O autor

Figura 45 – Migrações por *job* (média) em oito processadores e utilizações maiores



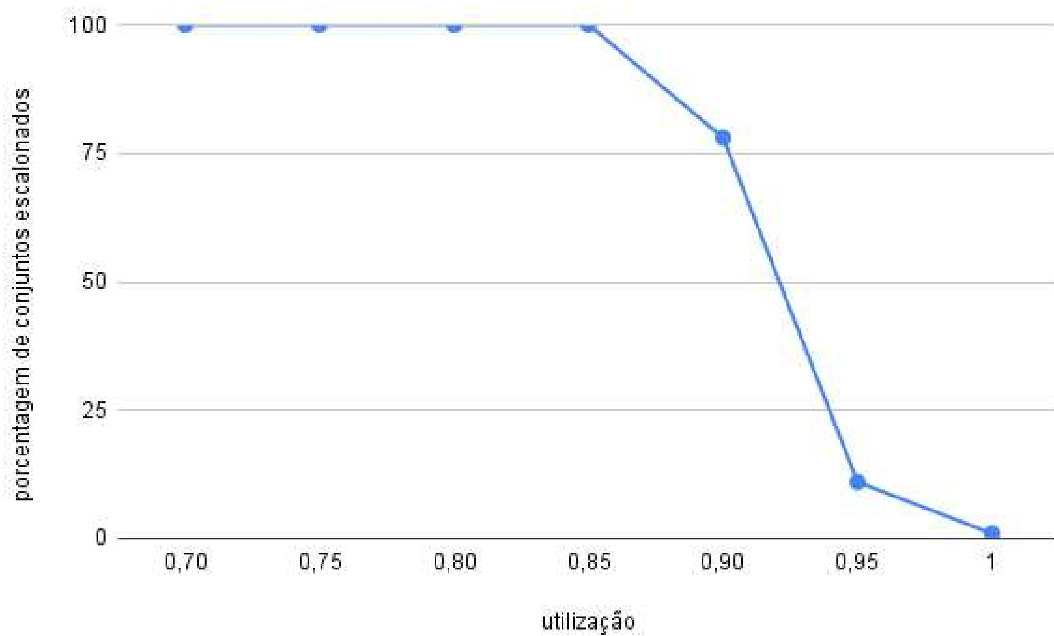
Fonte – O autor

Figura 46 – Migrações e preempções por *job* (média) em oito processadores e utilizações maiores



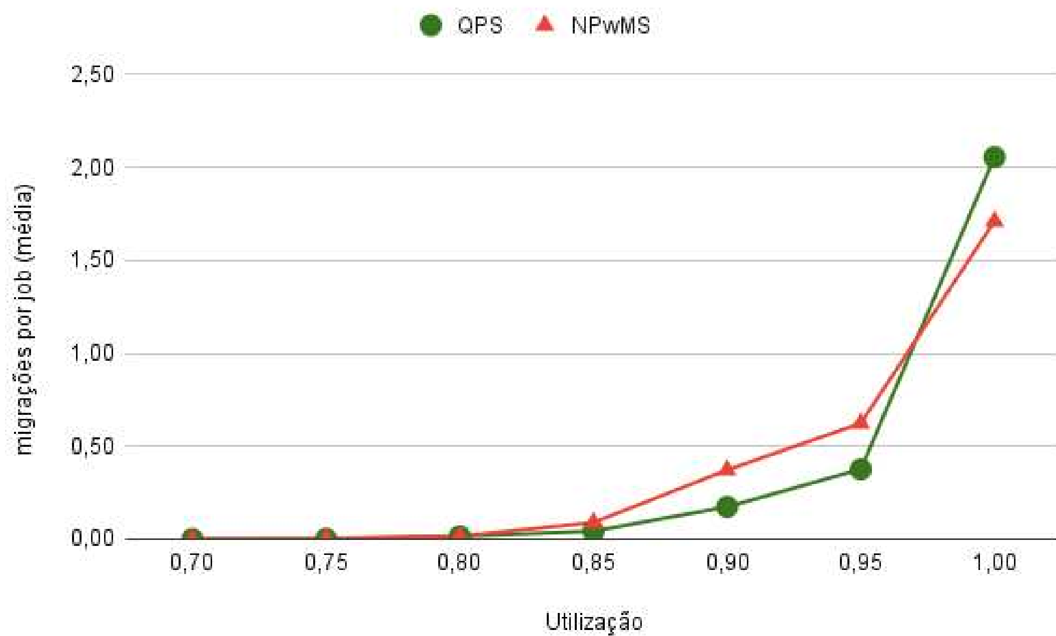
Fonte – O autor

Figura 47 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em oito processadores



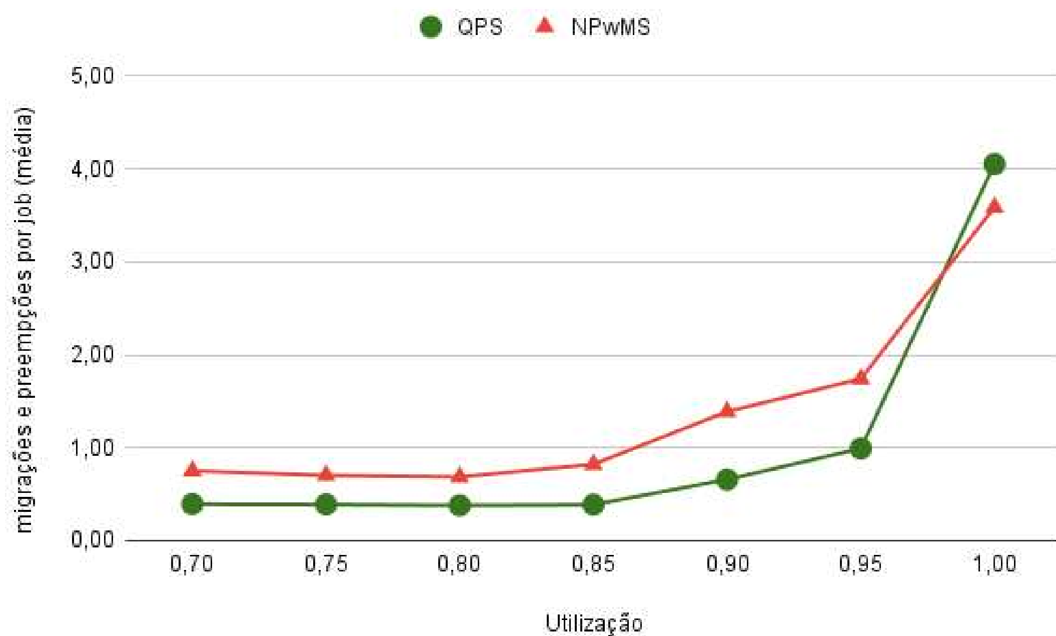
Fonte – O autor

Figura 48 – Migrações por *job* (média) em nove processadores



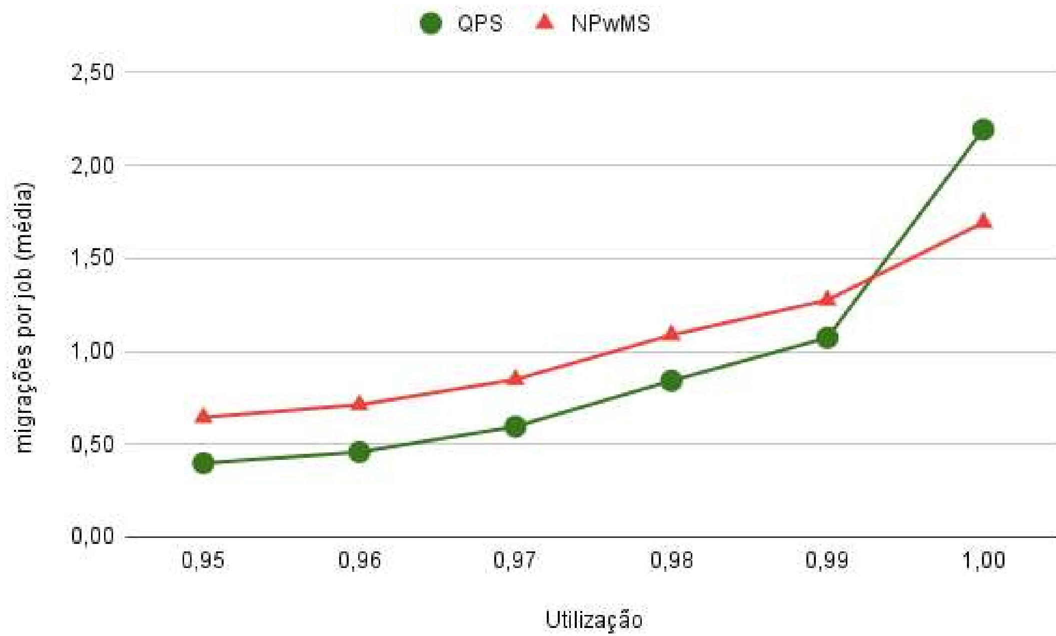
Fonte – O autor

Figura 49 – Migrações e preempções por *job* (média) em nove processadores



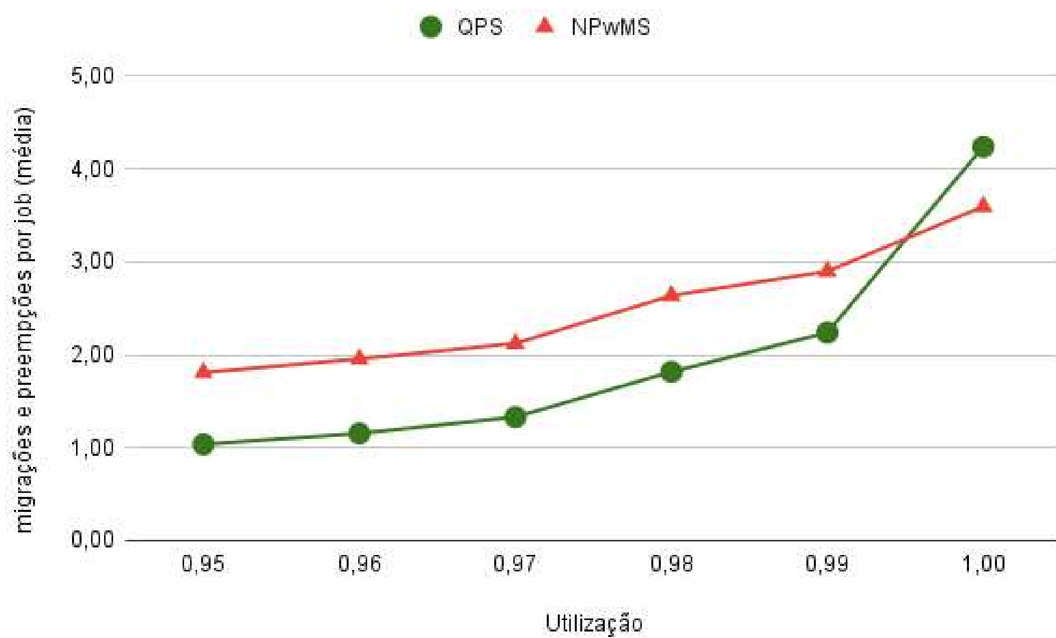
Fonte – O autor

Figura 50 – Migrações por *job* (média) em nove processadores e utilizações maiores



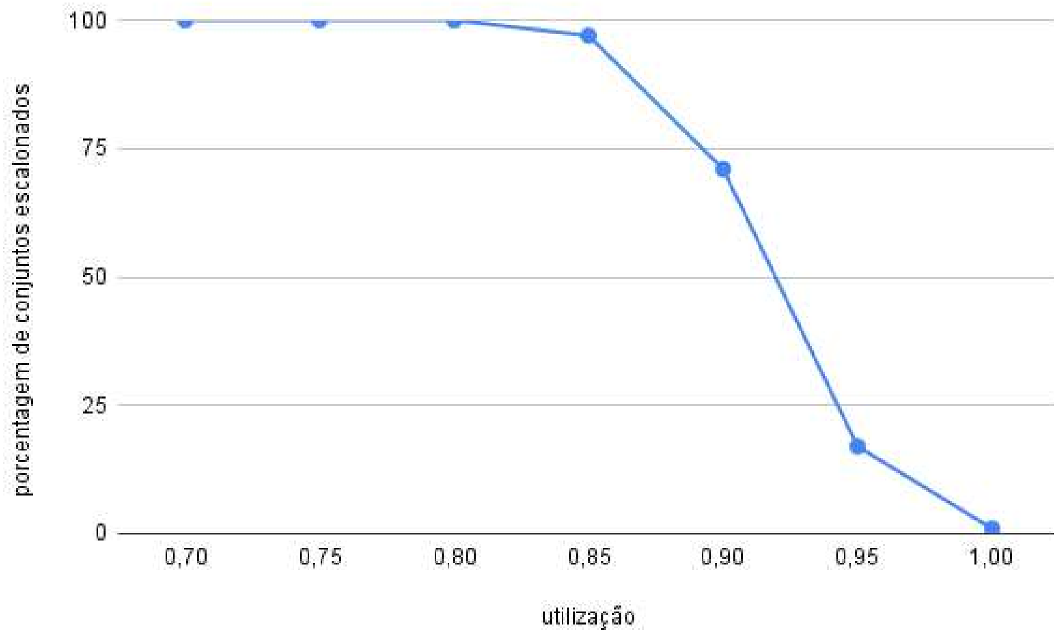
Fonte – O autor

Figura 51 – Migrações e preempções por *job* (média) em nove processadores e utilizações maiores



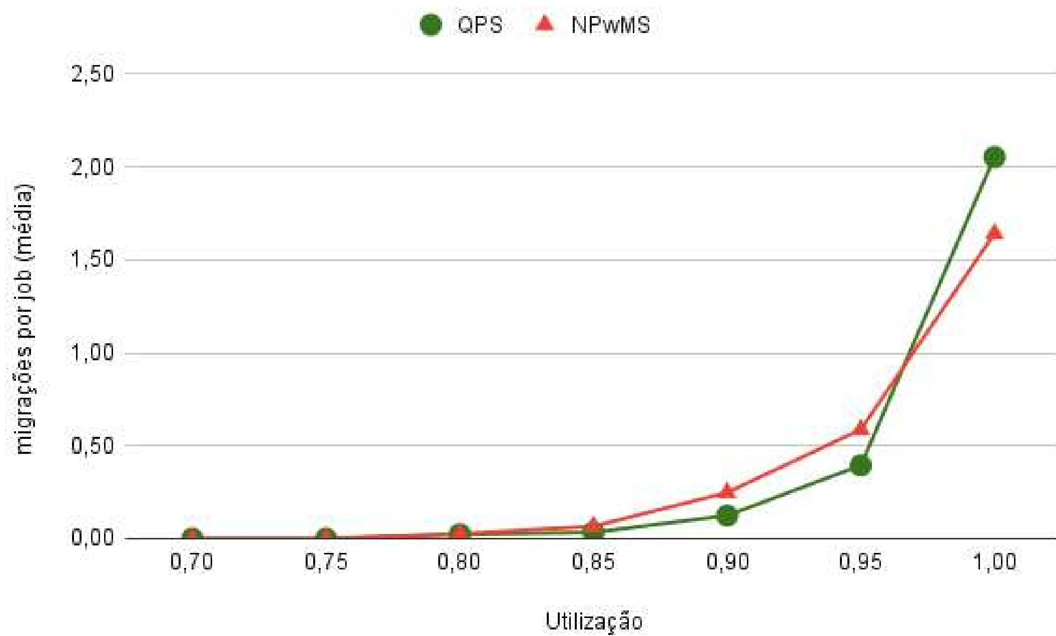
Fonte – O autor

Figura 52 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em nove processadores



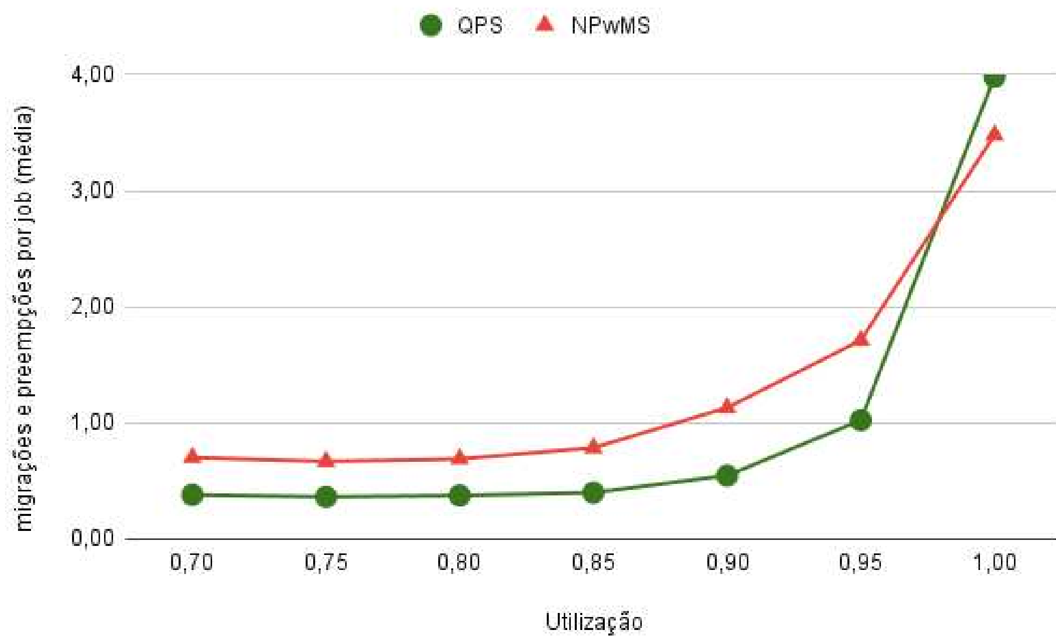
Fonte – O autor

Figura 53 – Migrações por *job* (média) em dez processadores



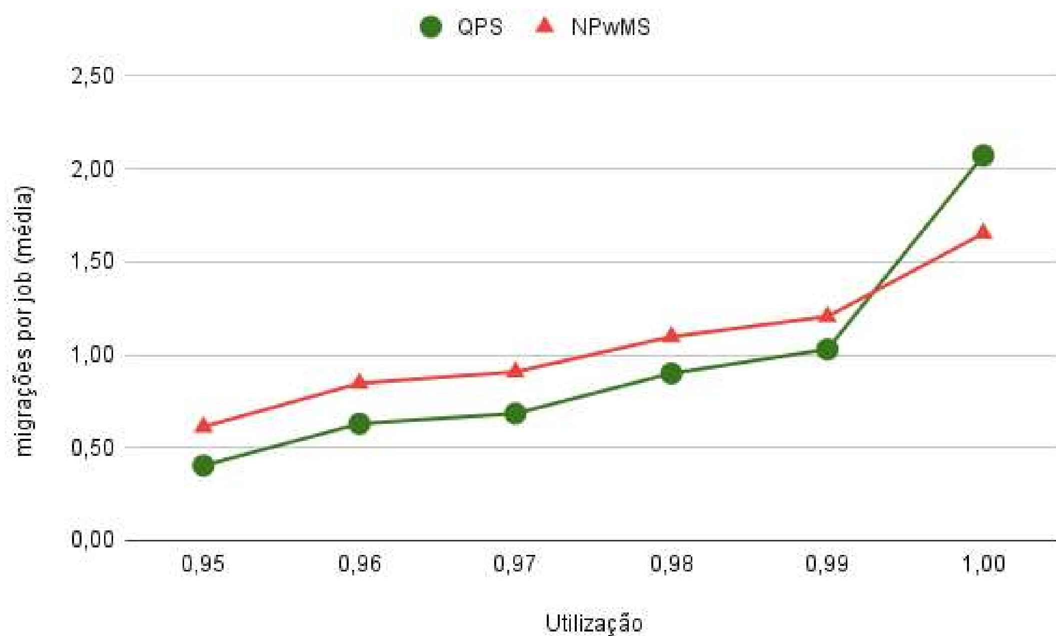
Fonte – O autor

Figura 54 – Migrações e preempções por *job* (média) em dez processadores



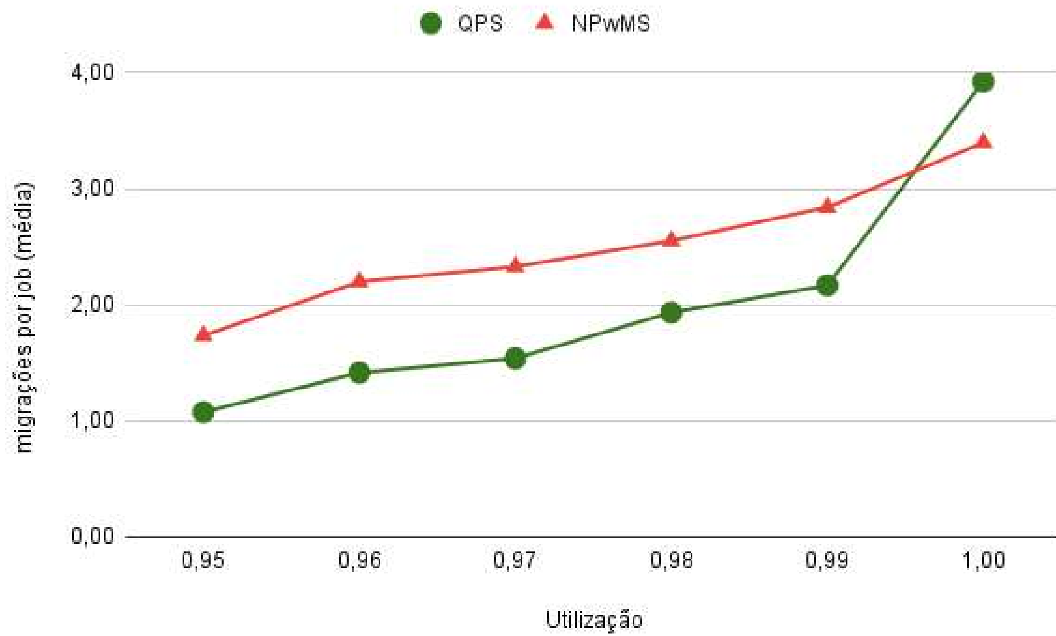
Fonte – O autor

Migrações por *job* (média) em dez processadores e utilizações maiores



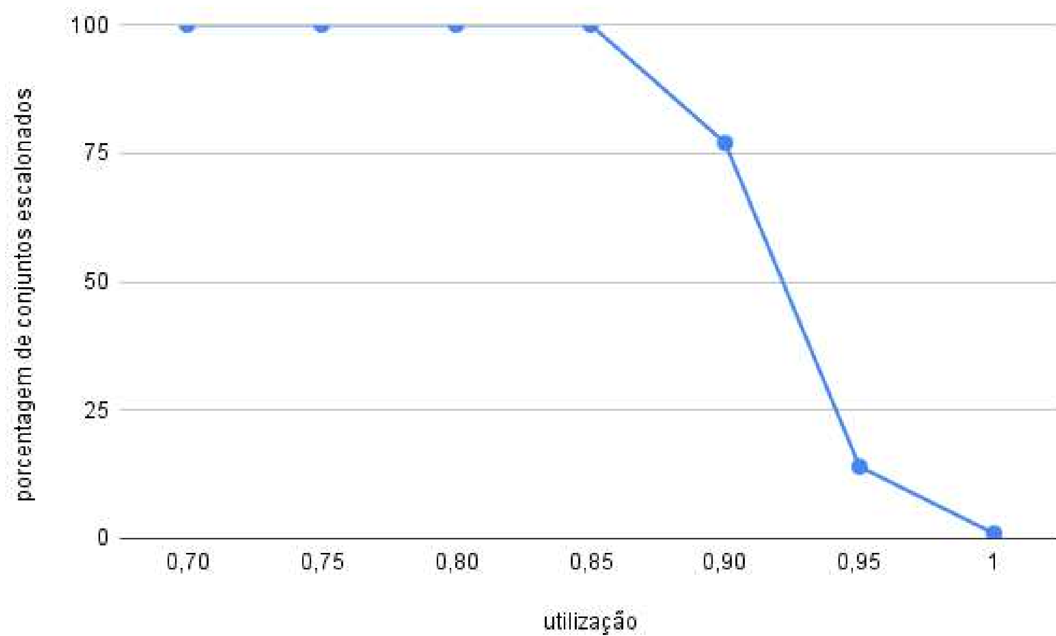
Fonte – O autor

Migrações e preempções por *job* (média) em dez processadores e utilizações maiores



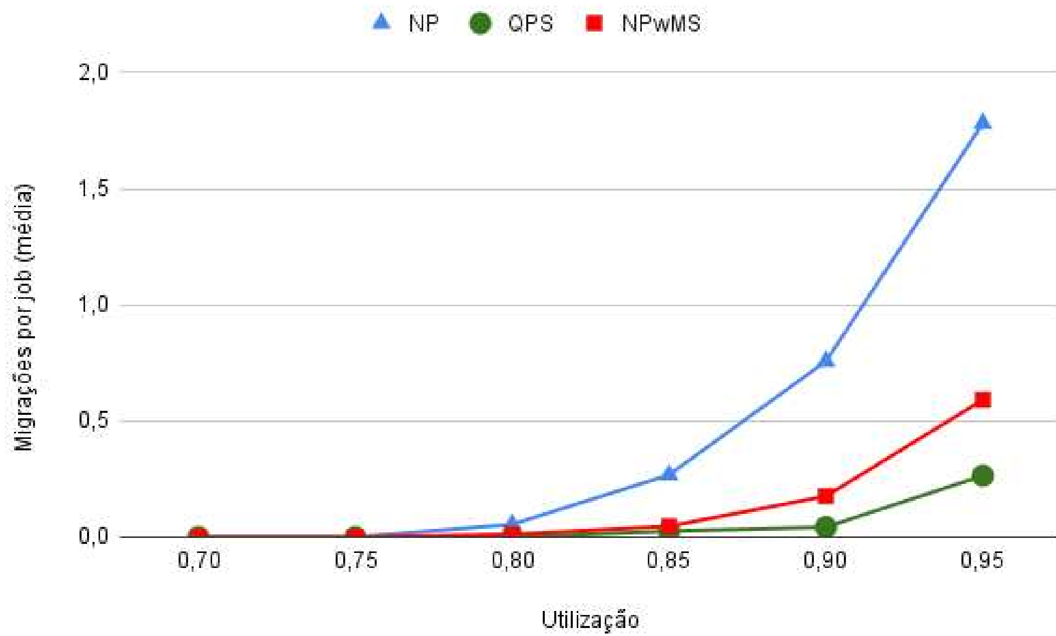
Fonte – O autor

Figura 55 – Porcentagem de conjuntos escalonados pelo *Notional Processors* original por utilização em dez processadores



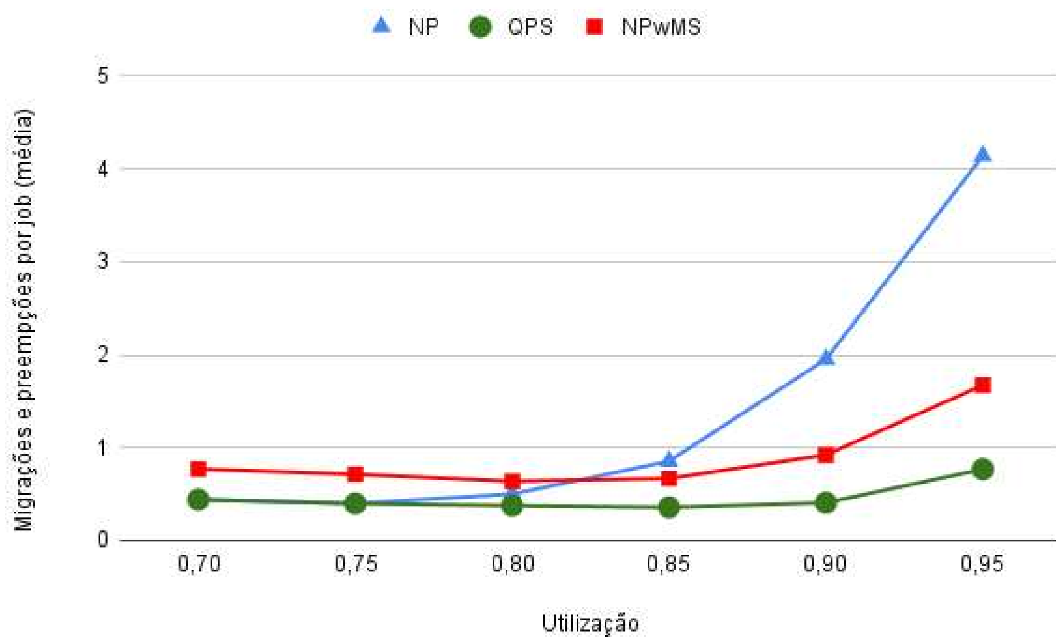
Fonte – O autor

Figura 56 – Migrações por job (média) em quinze processadores



Fonte – O autor

Figura 57 – Migrações e preempções por job (média) em quinze processadores



Fonte – O autor