Programa Interdisciplinar de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Roberto de Quadros Gomes

MigBSP++:

BALANCEAMENTO DE CARGA EFICIENTE PARA APLICAÇÕES PARALELAS EM FASES

São Leopoldo, 2014

Roberto de Quadros Gomes

MIGBSP++:
Balanceamento de Carga Eficiente para Aplicações Paralelas em Fases

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre pelo Programa Interdisciplinar de Pós-Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos — UNISINOS

Orientador:
Prof. Dr. Rodrigo da Rosa Righi

São Leopoldo
2014

Ficha catalográfica

G633m Gomes, Roberto de Quadros

MigBSP++: balanceamento de carga eficiente para aplicações paralelas em fases / por Roberto de Quadros Gomes. – 2014.

125 f.: il.,: 30cm.

Dissertação (mestrado) — Universidade do Vale do Rio dos Sinos, Programa Interdisciplinar de Pós-Graduação em Computação Aplicada, 2014.

"Orientador: Prof. Dr. Rodrigo da Rosa Righi".

1. Algoritmo de predição BSP. 2. AMPI. 3. Balanceamento de carga. 4. *Bulk-Synchronous Parallel*. 5. Estratégia. 6. MigBSP. 7. MigBSP++. 8. MigBSPLB. 9. Migração de processos. I. Título.

CDU 004.42

Catalogação na Fonte:
Bibliotecária Vanessa Borges Nunes — CRB 10/1556

(Esta folha serve somente para guardar o lugar da verdadeira folha de aprovação, que é obtida após a defesa do trabalho. Este item é obrigatório, exceto no caso de TCCs.)

AGRADECIMENTOS

Não posso deixar de registrar minha gratidão ao apoio, incentivo, paciência, colaboração e orientação que o Professor Dr. Rodrigo da Rosa Righi.

A minha esposa Luciane Bohrer pelo suporte e parceria nos finais de semana, noites em claro para as correções deste trabalho.

Um agradecimento especial aos colegas e amigos da Vieira Filho Tecnologia que permitiram minha ausência e me deram o suporte necessário para o desenvolvimento desta dissertação.

Aos amigos Vladimir Guerreiro e Márcia Elis Abech que compartilharam horas de estudos na biblioteca e foram parceiros em diversos grupos de trabalhos.

À equipe de desenvolvimento da Universidade de Ilinóis, que fizeram as adaptações necessárias no *Charm++* para que esta dissertação pudesse ser concluída.

Aos amigos que compreenderam minha ausência em eventos e momentos de diversão.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) que deu o suporte financeiro para a realização deste mestrado.

Obrigado!

RESUMO

A migração de processos é uma técnica utilizada no remapeamento de um processo para um processador mais rápido ou para aproximá-lo de outros processos com os quais se comunica frequentemente. Esta dissertação descreve o **MigBSP++**, um modelo de reescalonamento de processos que utiliza a técnica de migração para realizar o balanceamento de carga em sistemas paralelos. Direcionado às aplicações do tipo *Bulk-Synchronous Parallel* (BSP), o modelo apresentado redistribui os processos com o intuito de reduzir o tempo de cada super-passo. De modo similar ao MigBSP, o MigBSP++ combina múltiplas métricas a fim de decidir as migrações necessárias para que o sistema entre em equilíbrio sem a intervenção do usuário. As métricas utilizadas são: computação, comunicação e sobrecusto de migração. Através de sua função de decisão, chamada *Potencial de Migração* (PM), essas métricas são utilizadas para eleger os processos mais propícios a trazer o equilíbrio ao sistema. O MigBSP++ responde as questões necessárias para a política de migração de processos: *quando* realizar a migração de processos; *quais* processos são candidatos à migração e; para *onde* migrar os processos selecionados. Como contribuição científica, o MigBSP++ introduz as soluções para duas questões que estão em aberto no MigBSP: (a) a detecção de desbalanceamento de carga quando há mais processos do que processadores e; (b) a definição de quantos processos irão migrar de fato. Para a questão (a), propõe-se alteração do modo de detecção de desbalanceamento utilizada, observando o tempo total de computação de cada processador. Para a questão (b) é apresentado um algoritmo chamado de *Algoritmo de Predição BSP* (APBSP). Os dados de entrada do APBSP são os processos eleitos pela técnica de PM e a saída é uma lista de processos que irão, de fato, migrar proporcionando a redução do tempo do próximo super-passo. Para demonstrar os resultados da aplicação deste modelo, foram desenvolvidas duas aplicações BSP com o auxílio da biblioteca *Adaptive Message Passing Interface* (AMPI). Essa ferramenta oferece um arcabouço uniforme que, através da migração de processos, permite o balanceamento de carga de forma transparente ao usuário. Foram desenvolvidas as estratégias de balanceamento de carga, baseadas no MigBSP e no MigBSP++, para a realização da comparação entre elas e com as estratégias já existentes no sistema. Os resultados apontam que, nos casos onde a granularidade da tarefa é maior, os ganhos em tempo de execução são mais evidentes, podendo ser de até 46% em relação à aplicação sem balanceamento e de até 37% em relação às estratégias nativas do AMPI. Esses números sugerem que o modelo MigBSP++ tem aplicação prática e pode produzir resultados satisfatórios.

Palavras-chave: Algoritmo de predição BSP. AMPI. Balanceamento de carga. *Bulk-Synchronous Parallel*. Estratégia. MigBSP. MigBSP++. MigBSPLB. Migração de processos.

ABSTRACT

Process migration is a technique used in the remapping of a process to a faster processor or in the approaching from the processes which already have some communication among themselves. This essay describes the MigBSP++, a rescheduling process model that uses the technique of migration to perform load balancing in parallel systems. Directed to the Bulk-Synchronous Parallel (BSP) applications, the model redistributes the processes with the purpose of reducing the time of each super-step. Similar to MigBSP way, MigBSP++ combines multiple metrics to decide which migrations should be chosen in order to balance the entire system without the user intervention. The metrics used by the model are: computing, communication and extra costs of migration. Through its decision function, called Potential Migration (PM), these metrics are used to choose the most appropriate processes that will balance the system. MigBSP++ answers the questions about the policy process migration issues: when to perform the migration process, which processes are candidates for migration and where to migrate the selected processes. As scientific contribution, MigBSP++ introduces the solutions to two issues that were missing at MigBSP: (a) the detection of imbalance load when there are more processes than processors, and (b) the definition of how many processes will migrate indeed. On the question (a), a change of the mode of detection of imbalance is proposed, noting the total computation time for each processor. On the second question (b) an algorithm called the Prediction Algorithm BSP (PABSP) is presented. The input data of PABSP are elected process by the PM technique and the output is a list of processes that will, indeed, migrate providing a time reduction of the next super-step. To demonstrate the results of applying this model, two BSP applications have been developed with the assistance of Adaptive Message Passing Interface (AMPI) library. This tool provides a uniform framework that, through the migration process, allows a transparent load balancing to the user. Based on MigBSP and MigBSP++, load balancing strategies have been developed for the performance and comparison among new strategies and among the ones which were already in the system.The results indicate that, in cases where the granularity of the task, the gains in runtime are more evident, reaching up to 46% compared to the application without balancing, and 37% when compared to native strategies AMPI. These numbers suggest that the model MigBSP++ has practical application and can produce satisfactory results.

Keywords: Algorithm of prediction BSP. AMPI. Bulk-Synchronous Parallel. Load balancer. MigBSP. MigBSP++. MigBSPLB. Strategy. Process migration.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AMPI	*Adaptive Message Passing Interface*
APBSP	Algoritmo de Predição BSP
BC	Balanceamento de Carga
BSP	*Bulk-Synchronous Parallel*
DAG	*Directed Acyclic Graph*
FIC	*Fractal Image Compression*
JVM	*Java Virtual Machine*
MPI	*Message Passing Interface*
NUMA	*Non-Uniforme Memmory Access*
MIMD	*Multiple Instrution Stream - Multiple Data Stream*
MSE	*Mean Square Error*
PIPCA	Programa de Interdisciplinar de Pós-Graduação de Computação Aplicada
PUB	*Paderborn University BSP Library*
RTS	*Runtime System*
SO	Sistema Operacional
SPMD	*Single Program Multiple Data*
SSH	*Secure Shell*
UMA	*Uniforme Memmory Access*
VP	*Virtual Processor*
VPN	*Virtual Private Network*

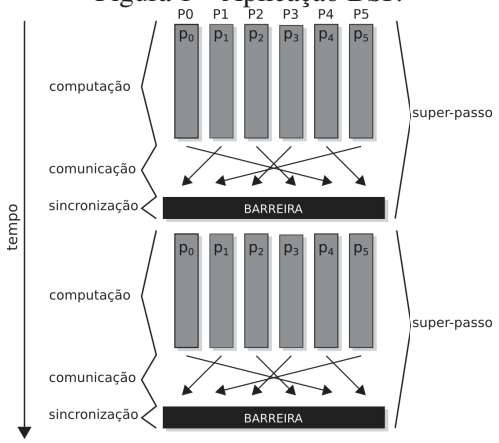
SUMÁRIO

1 INTRODUÇÃO

O avanço tecnológico permite que recursos computacionais, tais como os de rede, armazenamento e capacidade de processamento forneçam grandes oportunidades para a exploração de aplicações e pesquisas em sistemas paralelos e distribuídos (PARASHAR; LI, 2009). Estas pesquisas proporcionam condições para um aprimoramento de *softwares* utilizados em diversas áreas científicas tais como a Biologia (LIU; SCHMIDT, 2006; TANG *et al.*, 2012), Engenharia (FAN *et al.*, 2011; FUNG; CHOW; WONG, 2000) e Astronomia (HARTUNG *et al.*, 2012; DROST *et al.*, 2012).

Em problemas nos quais o alto desempenho é fundamental, a grande maioria das aplicações que os resolvem são elaboradas para serem executadas em aglomerados (*clusters*) ou em grades (*grids*) computacionais. Tais aplicações utilizam o poder computacional de duas ou mais máquinas interconectadas que trabalham em paralelo com o objetivo de resolver um único problema. Neste contexto, as aplicações podem tornar-se muito específicas para um único ambiente computacional e, muitas vezes, dependentes de arquitetura e infraestrutura de comunicação. Diante desta dificuldade de portabilidade, o modelo *Bulk-Synchronous Parallel* (BSP) foi introduzido por Valiant (1990) com o intuito de padronizar o desenvolvimento de sistemas paralelos, tanto de arquiteturas, quanto de aplicações (KRIZANC; SAARIMAKI, 1996). Além disso, o BSP tem sido utilizado como modelo para pesquisas recentes (GOMES; RIGHI, 2013; GUERREIRO; RIGHI, 2013; DIAMOS *et al.*, 2013; KAJDANOWICZ *et al.*, 2012; GRAEBIN; RIGHI, 2012; HUAN; QI-LONG; RUI, 2011) e é amplamente utilizado no meio científico.

Figura 1 – Aplicação BSP.



Fonte: Elaborado pelo próprio autor.

Observar-se na Figura 1 as fases que compõem uma aplicação do tipo BSP. Cada coluna p_n indica a execução de um processo em um processador P_j que faz parte do sistema paralelo. Estas aplicações são caracterizadas por possuírem divisões temporais definidas como *super-passos*. Cada um dos super-passos é composto pela sequência de etapas *computação*, *comu-*

nicação entre os processos e a *sincronização* entre eles. Na figura anteriormente mencionada, os processos são apresentados de forma simétrica no tempo de computação e no de comunicação. Considerando a taxonomia apresentada por Parashar e Li (2009), as aplicações paralelas possuem classificações baseadas em comportamento e divisibilidade. A Tabela 1 apresenta a classificação de acordo com a execução, atividade, granularidade e dependência.

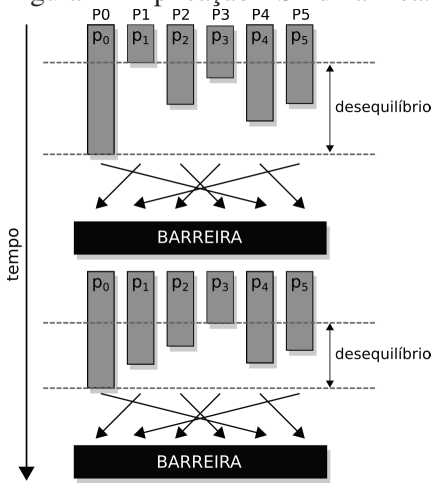
Tabela 1 – Classificação das características das aplicações

Características	Categorias
Execução	Computação Intensiva, Comunicação Intensiva, E/S Intensiva
Atividade	Dinâmica, Estática
Granularidade	Fina, Grossa, Indivisível
Dependência	Independente, Fluxo, Híbrido

Fonte: Traduzido livremente de Parashar e Li (2009).

Na linha que indica a característica **atividade** de uma aplicação, pode-se observar que as aplicações paralelas podem ser classificadas como *estáticas* ou *dinâmicas*. Programas definidos como estáticos possuem um comportamento similar em suas tarefas distribuídas e as operações realizadas pelos processadores são as mesmas. A aplicação que realiza uma multiplicação entre matrizes e a simulação de Monte Carlo (WILKINSON; MICHAEL, 2005) são exemplos desta categoria. Já as aplicações dinâmicas, possuem um comportamento imprevisível que pode gerar um desequilíbrio na distribuição das tarefas em execução. Um exemplo deste tipo de aplicação é o compressor de imagens que utiliza a técnica de Fractais (QUADROS GOMES *et al.*, 2013).

Figura 2 – Aplicação BSP dinâmica.



Fonte: Elaborado pelo próprio autor.

Aplicações BSP com o comportamento dinâmico podem ser retratadas como a Figura 2. Nesta ilustração, observa-se que os processos terminam em instantes diferentes em um superpasso. O desequilíbrio indicado na imagem pode significar uma redução de desempenho devido a processadores que se tornam ociosos enquanto outros ficam sobrecarregados, causando

um efeito de gargalo no fluxo da aplicação. Isso acontece porque o processo que tem mais instruções (na figura representado por p_0) é o principal responsável por definir o tempo de um super-passo. Além disso, um ambiente computacional heterogêneo, ou seja, composto por computadores com capacidades diferentes, também pode ser responsável por este efeito causador do desequilíbrio das tarefas. Ou ainda, uma situação onde não é possível prever a atividade da aplicação ou o ambiente de execução é desconhecido. Com objetivo de reduzir esse problema, o *balanceamento de carga* surge como uma solução viável. A principal função desta operação é evitar que a utilização dos recursos fique desproporcional, garantindo que o potencial de *hardware* disponível seja melhor explorado (WILKINSON; MICHAEL, 2005). O balanceamento é realizado com a ponderação das tarefas através dos processadores de um sistema paralelo (tanto em grade, como em aglomerado) com o intuito de equilibrar as cargas de tarefas e, consequentemente, reduzir o tempo total de execução. Quando é realizada a distribuição das tarefas de forma a aproveitar melhor os recursos somente no início da aplicação fica caracterizado o *balanceamento de carga estático*. Outra abordagem possível é a coleta de informações durante a execução de um programa e a redistribuição das tarefas quando necessário. Esta forma de atuação é denominada *balanceamento de carga dinâmico*.

Diversas ferramentas auxiliam na realização do balanceamento de carga dinâmico. Durante a execução da aplicação, elas efetuam uma nova distribuição dos processos mais carregados através de processadores menos carregados (BONORDEN, 2007; RODRIGUES, 2011). Esta realocação de tarefas é denominada *reescalonamento de tarefas* e é realizada através da *migração de processos*. Além de auxiliar no balanceamento de carga, a migração de processos proporciona a tolerância à falha e permite operações administrativas como, por exemplo, o deslocamento de um processo para o desligamento de um computador previamente em uso. A migração de um processo pode ser efetuada realizando uma *interrupção* na tarefa, *movimentação* e a *reexecução* do ponto de onde estava quando foi interrompida (TANENBAUM; STEEN, 2006).

Uma iniciativa de migração com o modelo BSP que vem sendo objeto de estudos na linha de pesquisa de Redes de Computadores e Sistemas Distribuídos do Programa Interdisciplinar de Pós-Graduação de Computação Aplicada (PIPCA) da UNISINOS é denominada **MigBSP**. Este modelo é uma estratégia que utiliza um conjunto de métricas para a decisão de migração de processos que responde as questões necessárias para o balanceamento de carga (ROSA RIGHI *et al.*, 2009; RIGHI *et al.*, 2009; GRAEBIN; da Rosa Righi, 2011). Os resultados de sua utilização são apresentados na literatura tanto no âmbito da simulação (RIGHI, 2009), como em implementação utilizando a linguagem Java (GRAEBIN, 2012).

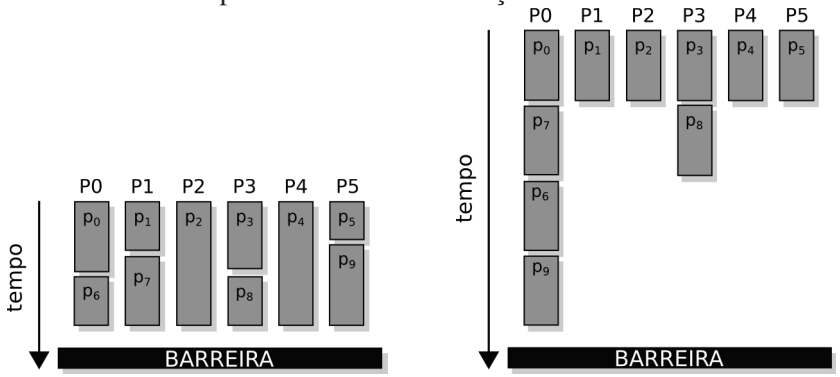
Entretanto, após a uma série de estudos e ensaios com o MigBSP, duas características demonstraram a necessidade de aperfeiçoamento. A primeira refere-se a detecção de desbalanceamento de carga. Apesar da ideia principal do modelo basear-se nos tempos de cada processo para definir um desequilíbrio, existem situações onde o modo proposto pode cometer alguns equívocos. A segunda refere-se a decisão de quantos processos serão migrados. O modelo

apresenta duas técnicas de decisão. Nesta dissertação é apresentada uma opção que considera as características das aplicações BSP para definir de maneira mais assertiva e assim mais rapidamente alcançar o equilíbrio no sistema paralelo.

1.1 Motivação

Na Figura 3, pode-se observar exemplos de cenários onde o modelo MigBSP irá gerar um falso positivo (a) e um falso negativo (b) na análise de desbalanceamento dos processos. Esta situação ocorre devido ao método que se baseia no tempo de execução das tarefas. O ambiente representado pelas figuras consideram que um processador pode executar um ou mais processos. Neste ambiente, figura 3(a) mostra a situação na qual as tarefas são sincronizadas após o término de todos os super-passos existentes em um processador. Embora os processos apresentem tempos de execução diferentes, a carga dos processadores está igualmente distribuída. Porém, o MigBSP irá indicar o desequilíbrio nesse cenário. Se avaliarmos a situação indicada por 3(b), embora os processos possuam tempos de execução semelhantes, o primeiro processador está com o tempo total de execução maior que os outros. Neste caso, o MigBSP equivocadamente irá indicar que o sistema está em equilíbrio.

Figura 3 – Cenários de possíveis erros na detecção do desbalanceamento de carga.



(a) Sistema em equilíbrio detectado como desbalanceado.

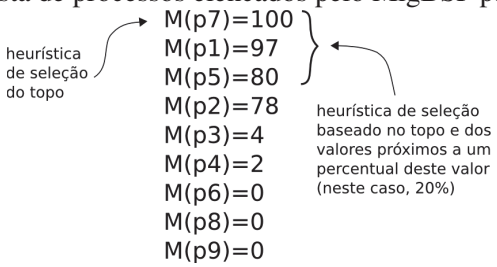
(b) Sistema desbalanceado e não detectado.

Outra técnica alvo de estudo nesta dissertação é o modo de escolha da quantidade de processos a serem migrados. Atualmente, o MigBSP utiliza uma métrica M, Figura 4, para listar de forma decrescente os processos mais propícios a serem migrados. Baseado nesta lista, ou se migra o processo do topo, ou se seleciona os processos que estão dentro de um intervalo percentual definido pela implementação realizada.

A primeira observação é realizada considerando a heurística baseada no topo da lista. Apesar de funcional, ela pode ser considerada conservadora e, dependendo do número de tarefas, uma determinada quantidade elevada de super-passos pode ser necessária até que o equilíbrio no sistema seja alcançado. Já no modo em que consideramos uma porcentagem do valor definido pelo topo da lista, não há nenhuma garantia da conservação do equilíbrio quando mais de

um processo efetuar a migração, visto que pode ocorrer uma sobrecarga nos processadores. Isso acontece devido a métrica M informar um valor que considera um único processo em relação a um processador, sem relacionar os demais processos existentes.

Figura 4 – Lista de processos elencados pelo MigBSP para migração.



Fonte: Elaborado pelo próprio autor.

1.2 Objetivo

Frente as observações realizadas na seção anterior, esta dissertação apresenta um modelo de reescalonamento de processos que solucione as duas questões apontadas dedicado às aplicações do tipo BSP. Como principal contribuição este modelo aperfeiçoa a seleção de processos através de um algoritmo que realiza a predição do super-passo seguinte garantindo que o ele tenha um tempo de execução menor do que o super-passo atual.

1.3 Abordagem do Problema

O novo modelo apresentado é denominado **MigBSP++**. Ele possui a maioria das características existentes no MigBSP. Porém, diferentemente do modelo MigBSP que é desenvolvido para solucionar questões de desbalanceamento em grades computacionais, ele é orientado à aplicações BSP executadas em aglomerados. O MigBSP++ analisa os processadores do sistema paralelo e os processos em execução e realiza a redistribuição de tarefas de forma mais eficiente.

As principais características exploradas pelo modelo são:

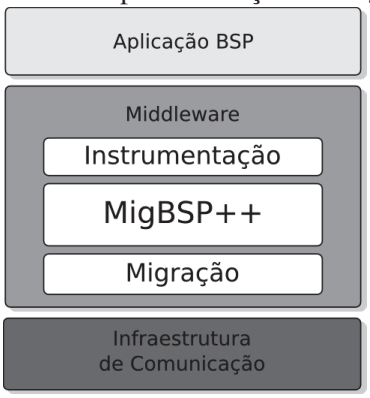
- Detecção de desbalanceamento de carga;
- Seleção automática dos processos a serem migrados através de um algoritmo específico;
- Observar as características do sistema paralelo na decisão de migração;
- Responder as questões *quem*, *quando* e *onde*.

Para solucionar o problema da detecção do desequilíbrio no sistema paralelo o método proposto não considera apenas o tempo das tarefas. Ele utiliza o tempo total de todas as tarefas que estão em um processador.

O MigBSP++ possui o mesmo princípio empregado pelo MigBSP para elencar os processos a serem migrados. Entretanto, a partir das características das aplicações BSP, é utilizado um algoritmo de predição que avalia o sistema como um todo e realiza a simulação de cada migração elencada. Cada migração que resulte em um super-passo subsequente com tempo menor, será adicionada em uma lista de processos a serem migrados.

De modo geral, o MigBSP++ é dedicado a uma arquitetura computacional representada pela Figura 5.

Figura 5 – Sistema para utilização do MigBSP++.



Fonte: Elaborado pelo próprio autor.

A aplicação BSP não sofrerá nenhuma alteração em seu desenvolvimento, sendo o modelo de reescalonamento proposto de uso transparente por parte do desenvolvedor. Como parte central da arquitetura é demonstrado a utilização de um *middleware* no qual está a implementação do MigBSP++. Através de sua **Instrumentação**, ele é responsável por realizar a coleta de informações pertinentes ao comportamento dos processos, dados sobre o padrão de comunicação, ocupação dos processadores e outras questões necessárias para avaliação do sistema. De posse destas informações, o MigBSP++ retorna uma lista de processos a serem migrados. Em seguida, esta lista é utilizada pelo bloco de **Migração** que realiza as ações necessárias para o transporte dos processos.

1.4 Organização do Trabalho

Este trabalho está organizado de forma a oferecer as informações necessárias para o acompanhamento do desenvolvimento do modelo de reescalonamento para aplicações paralelas BSP, denominado MigBSP++. No capítulo 2, a fundamentação teórica dos elementos que fazem parte do desenvolvimento desta dissertação é apresentada. Encontram-se expostas às informa-

ções sobre escalonamento, migração de processos/tarefas/objetos, o modelo BSP, informações sobre a biblioteca utilizada nos ensaios, *Charm++* e suas peculiaridades. Neste mesmo capítulo, algumas características da biblioteca AMPI e suas vantagens são apresentadas.

O capítulo seguinte, 3, possui o objetivo de apresentar o estado-da-arte sobre as bibliotecas BSP e balanceamento de carga. Alguns trabalhos que utilizam a migração de processos, migração de máquinas virtuais e comparativos de desempenho entre estratégias de balanceadores são relacionados. Dado que o presente trabalho utiliza como ferramenta os recursos que *Runtime System* (RTS) *Charm++* fornece, são apresentados trabalhos contendo as informações sobre os balanceadores de carga nativos do pacote e suas características. Pesquisas utilizando o modelo MigBSP também são analisados, e assim, justificando a sua escolha para esta dissertação.

O capítulo 4 expõe o modelo proposto MigBSP++. Também são demonstrados como as métricas fornecidas pelo *middleware* são utilizadas para elencar os processos a serem migrados. Ainda, são apresentados os modelos matemáticos para as questões *quando*, *quem* e *onde* e o algoritmo de predição para aplicações BSP. Este algoritmo soluciona a questão quantidade de processos a serem migrados.

O capítulo 5 traz as observações pertinentes ao processo de implementação, tais como as condições de contorno e possíveis tratamentos para aproximar as características da estratégia implementada sobre o RTS *Charm++*, o *MigBSPLB* ao modelo MigBSP++. Também encontra-se a metodologia que foi utilizada para a avaliação do modelo proposto.

No penúltimo capítulo, 6, são expostos os resultados alcançados e as observações pertinentes a eles em cada um algoritmos utilizados.

O último capítulo expõe as considerações finais, realizando um resumo geral dos resultados, demonstrando os pontos fracos e fortes do modelo *MigBSP++* baseado nos resultados dos ensaios com o MigBSPLB. Também são elencadas as principais contribuições desta dissertação e as análises futuras que podem ser propostas em trabalhos posteriores a este.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo introduz os tópicos necessários para o desenvolvimento desta dissertação. As seções seguintes expõem as principais características da programação paralela pertinentes ao assunto abordado. Também, são apresentadas algumas justificativas de um sistema paralelo e a importância de estudá-los.

Adotado como alvo chave para a utilização do MigBSP++, o modelo BSP também é descrito e tem suas vantagens detalhadas na utilização para a escrita de programas paralelos. Ainda estão demonstrados alguns conceitos de migração e escalonamento de processos e quais as maneiras que estas técnicas podem se apresentar.

2.1 Computação Paralela

Quando era necessário aumentar o poder de um sistema computacional, a solução direta era elevar a frequência do relógio (*clock*) global do processador. Atualmente, os processadores estão alcançando velocidades próximas ao limite fundamental. Como apresentado por Tanenbaum (2011), baseando-se na teoria da relatividade de Einstein, um sinal elétrico, no cobre, atingirá a velocidade máxima de $20\frac{cm}{ns}$. Se considerarmos um sistema que alcance 10GHz, os sinais elétricos não poderão trafegar mais do que 2cm. Caso a frequência continue aumentando, este limite de percurso ficará cada vez menor, forçando cada vez mais a miniaturização das conexões. Com isso, a energia dissipada terá áreas menores para a troca de calor.

Segundo Pacheco (2011), o desempenho de um processador está relacionado diretamente com a densidade de transistores que o compõe. Com a redução de tamanho destes semicondutores, a velocidade máxima deles se eleva, fazendo com que o tempo de resposta do circuito integrado diminua como um todo. Entretanto, a quantidade de energia consumida cresce devido ao aumento da frequência de operação deste. A maior parte desse consumo de potência, devido ao efeito joule, acaba se convertendo em energia dissipada. Este aquecimento pode danificar os próprios transistores. Diante deste cenário, e considerando a tecnologia atual de construção de processadores, o aumento da densidade de transistores atingirá um limite físico no qual será impossível aumentar a velocidade sem que a miniaturização e o aquecimento gerado danifiquem o circuito integrado. Para evitar esta limitação futura, ao invés de desenvolver um processador mais rápido e mais complexo a indústria encontrou uma opção viável: o *paralelismo*.

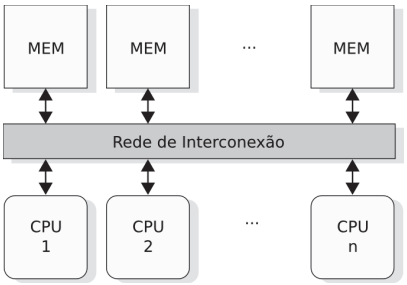
Wilkinson e Michael (2005) definem como **computador paralelo** um sistema computacional desenvolvido para conter múltiplos processadores ou muitos computadores independentes interconectados de alguma forma. Esta abordagem traria um acréscimo significativo de desempenho. Considerando uma situação ideal, a ideia principal era fazer com que *n* computadores possam prover o poder computacional de *n* vezes a capacidade de processamento de um único computador. Na maioria dos programas reais não é possível dividí-los em partes exatamente iguais e a interação entre estas partes costumam existir na sincronização e na troca de dados.

Dependendo do quanto for possível dividir o problema, é possível obter melhoras significativas nos tempos de execução com a paralelização.

2.1.1 Multiprocessadores de memória compartilhada

Um sistema computacional no qual dois ou mais processadores acessam uma mesma memória RAM (*Random Access Memmory*), de forma que todos tenham acesso irrestrito aos dados, é definido como **sistema de multiprocessadores de memória compartilhada**. Neste modo de operação computacional, um processo pode escrever em uma determinada posição de memória e, ao ler, obter um valor diferente, pois um outro processador pode ter acesso a mesma área e sobrescrevê-la. Quando organizada corretamente, esta configuração permite que ocorra a troca de informação entre processadores de forma eficiente (TANENBAUM, 2011).

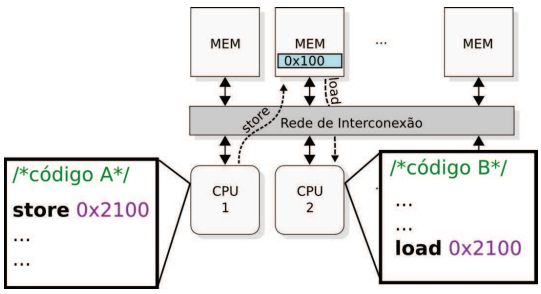
Figura 6 – Sistema com multiprocessadores e memória compartilhada.



Fonte: Traduzido livremente de Wilkinson e Michael (2005).

A Figura 6 apresenta este modelo. Nela, os processadores estão conectados aos módulos de memória através de algum tipo de rede de interconexão. Este sistema garante que os endereços de memória sejam únicos e cada processador tenha a possibilidade de acessá-los utilizando o mesmo endereço.

Figura 7 – Endereçamento virtual.



Fonte: Elaborado pelo próprio autor.

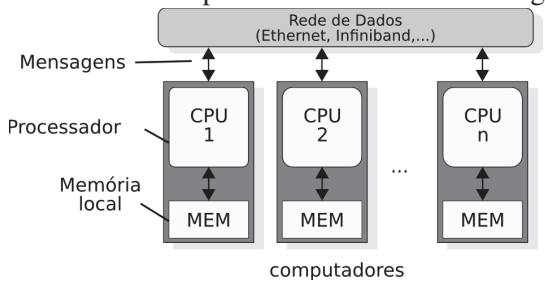
A Figura 7 demonstra uma situação na qual dois processos distintos utilizam uma mesma posição de memória para trocar informações. No processo em execução na CPU 1, o código "A" realiza a operação **store** (armazenar) que faz uma escrita de valor no endereço 0x2100 e, em seguida, o processo em execução na CPU 2, código "B", realiza a operação **load** (carregar) que faz a leitura do valor armazenado em 0x2100. Apesar de ambos os códigos indicarem a posição 0x2100, a posição de memória é 0x100 do segundo módulo. Esta tradução de endereços é conhecida como *endereçamento virtual* (WILKINSON; MICHAEL, 2005). O termo usado para definir este modo de acesso é *Acesso Uniforme à Memória* (UMA).

A maioria dos sistemas operacionais tratam o sistema multiprocessador de memória compartilhada de modo regular, tendo reservado áreas destinadas para a troca de informações, sincronização dos processos, gerenciamento de recursos e escalonamento. Uma característica interessante que Tanenbaum (2011) ressalta neste contexto é a troca de informação entre processos de forma invisível ao programador. Porém, a implementação em *hardware* de tal arquitetura é custosa devido à dificuldade de alcançar acessos rápidos em todos os módulos de memória (WILKINSON; MICHAEL, 2005).

2.1.2 Multicomputador com troca de mensagens

Um sistema com multiprocessadores pode ser criado conectando-se computadores através de uma rede de dados. Cada processador pode ter acesso a sua memória local de forma mais rápida do que o acesso em arquiteturas UMA. Como apresentado na Figura 8, esta arquitetura é conhecida por ter *Acesso Não-Uniforme à Memória* (NUMA).

Figura 8 – Multicomputador com troca de mensagens.

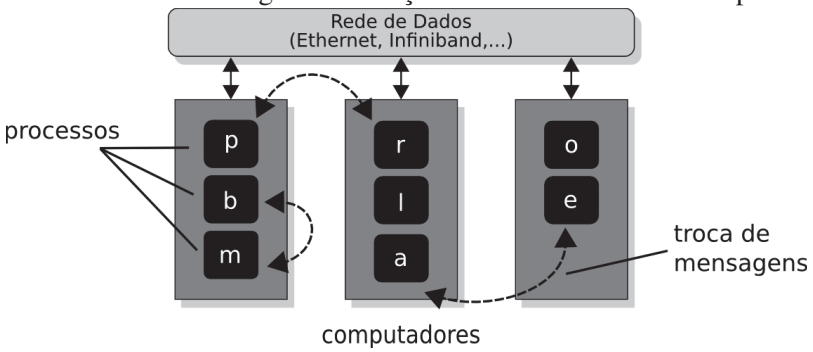


Fonte: Elaborado pelo próprio autor.

Neste sistema, os processadores não possuem acesso direto a memória de outros processadores, sendo necessária a troca de mensagens entre eles. Esta arquitetura de multiprocessadores é denominada *multiprocessador com troca de mensagens* (WILKINSON; MICHAEL, 2005). Apesar deste modelo apresentar um atraso maior ao acesso de memórias remotas, ele apresenta uma melhor escalabilidade do sistema (TANENBAUM, 2011). Diferentemente do modelo UMA, no qual a troca de dados ocorre de forma invisível ao programador, a troca de mensagem precisa ter declarações explícitas em código usando as funções **send** (enviar) e **recv**

(receber).

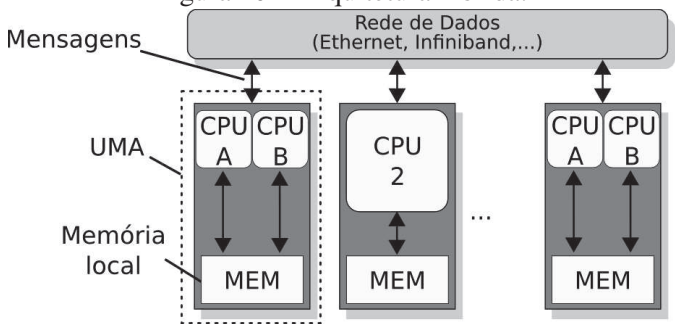
Figura 9 – Toca de mensagens na solução de sub-instâncias de um problema.



Fonte: Elaborado pelo próprio autor.

O problema a ser computado é dividido em n instâncias de um problema maior e cada parte é executada em um processador. Se o número de processadores disponíveis for igual ao número de instâncias, cada processador realiza a execução de cada uma delas. Porém, se o número de instâncias for maior do que a quantidade de computadores, mais de um processo será executado por processador, como mostrado na Figura 9. Os processos se comunicam através da troca de mensagens e esta é a única forma de atualização dos dados e resultados entre eles. Com as arquiteturas atuais dos processadores também é possível utilizar o modo de operação híbrida, no qual, máquinas UMA fazem parte de um sistema NUMA, como mostrado na Figura 10.

Figura 10 – Arquitetura híbrida.



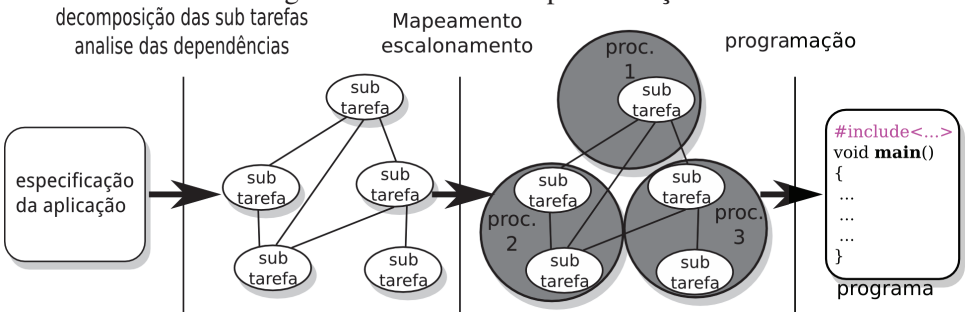
Fonte: Elaborado pelo próprio autor.

2.2 Escalonamento em Sistemas Paralelos

De acordo com Sinnen (2007), a paralelização de uma aplicação consiste em dividí-la em subtarefas que, em geral, não são independentes. Para a correta execução, as subtarefas que são dependentes de outras, ou que necessitam de algum tipo de sincronização, devem possuir uma ordenação para a execução. Esta ordenação consiste em uma alocação das tarefas para

processadores e na definição da ordem de suas execuções. Este mapeamento, tanto espacial quanto temporal, é definido por Sinnen (2007) como *escalonamento*. Na Figura 11, da esquerda para direita, o processo inicia com a definição da especificação da aplicação. Em seguida, ocorre o processo de decomposição das subtarefas, análise das dependências, o mapeamento e escalonamento. O processo de programação define a etapa de escrita de código como *Programa Único Múltiplos Dados* (Single Programa Multiple Data - SPMD).

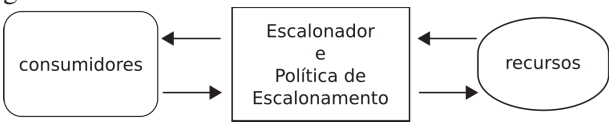
Figura 11 – Processo de paralelização.



Fonte: Traduzido livremente de Sinnen (2007)

Outras formas de apresentar o conceito de escalonamento são demonstradas nos trabalhos de Yamin (2001) e Righi (2009). Segundo eles, o escalonamento é a política de acesso de um conjunto de consumidores a um conjunto de recursos como demonstrado na Figura 12.

Figura 12 – Escalonamento de consumidores e recursos.



Fonte: Traduzido livremente de Righi (2009)

O escalonador pode ser usado com dois propósitos: alcançar um melhor desempenho ou melhorar a eficiência (YAMIN, 2001). Uma outra forma definida pelo próprio autor, Yamin (2001), como semi-formal é a seguinte: *"Escalonamento é uma função σ que associa cada tarefa (processo) tanto com um instante t_s na qual sua execução irá começar, como com um processador específico da arquitetura"*.

Em Sinnen (2007) e Semar Shahul e Sinnen (2010), os autores utilizam o recurso de grafos acíclicos dirigidos (DAG) para representar o processo de maneira mais formal. Um DAG é definido como $G = (V, E, w, c)$ representando um programa Pi. Os nós, no conjunto V, representam as tarefas do programa Pi e as arestas, pertencentes ao conjunto E, representam a comunicação entre as tarefas. Uma aresta $e_{ij} \in E$ que conecta o nó n_i até n_j, com n_i e $n_j \in V$, representa a comunicação do nó n_i até o nó n_j. O custo positivo $w(n)$ associado com

o nó $n \in V$ representa o custo computacional da tarefa n e $c(e_{ij})$ associado a aresta $e_{ij} \in E$ representa o custo de e_{ij}.

A definição é: *Um escalonamento S de um DAG $G = (V, E, w, c)$ em um conjunto finito de processadores P, é um par de funções $(t_s, proc)$, onde*

- t_s:$V \rightarrow Q_0$ *é a função de início dos nós (tarefas) de G.*
- $proc$:$V \rightarrow P$ *é a função de alocação dos processadores dos nós (tarefas) de G.*

As definições apresentadas por Yamin (2001) e Sinnen (2007) convergem em:

$$\sigma = (t_s, proc) \tag{2.1}$$

As funções em t_s e $proc$ descrevem a associação temporal e espacial das tarefas, representadas pelos nós de um DAG sobre processadores de um sistema paralelo, e estes processadores definidos pelo conjunto P. A tarefa representada pelo nó $n \in V$ é agendada para iniciar a execução no instante $t_s(n)$ no processador $proc(n) = p$, $p \in P$ (SINNEN, 2007). Consequentemente,

$$t_s(n, p) \Leftrightarrow t_s(n), proc(n) = p, p \in P. \tag{2.2}$$

De acordo com Semar Shahul e Sinnen (2010), o modelo clássico considera que o sistema paralelo P consiste em um conjunto de processadores iguais interconectados por uma rede de comunicação. Tais sistemas possuem as seguintes características:

- Sistema dedicado - O sistema paralelo é dedicado a execução do DAG escalonado. Nenhum outro programa ou tarefa é executada enquanto a tarefa DAG é executada;
- Processador dedicado - Cada processador $p(p \in P)$ pode executar apenas uma tarefa por vez e a execução não é preemptiva;
- Comunicação local sem custo - O custo de comunicação entre tarefas num mesmo processador é negligenciado e definido como 0. Isto ocorre pois o custo de comunicação entre processadores é muito mais elevado do que as comunicações locais;
- Subsistema de comunicação - Os processadores não se envolvem na comunicação entre eles. Um subsistema dedicado realiza esta função;
- Concorrência de comunicação - A comunicação entre os processadores ocorre sem nenhuma contenção;
- Totalmente conectado - A rede de comunicação é totalmente conectada. Todos os processadores podem comunicar com todos os outros livremente;
- Homogeneidade - Os processadores em P são idênticos.

A caracterização do gasto elevado em comunicação é tipicamente para arquiteturas NUMA ou de troca de mensagens, nas quais o uso de memória é distribuído pelos processadores do sistema paralelo utilizado (SINNEN, 2007).

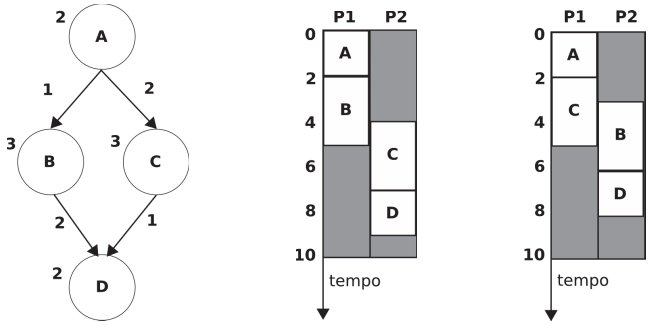
O tempo de execução t_f de qualquer nó $n \in V$, pode ser calculado a partir do valor de início da tarefa adicionado ao seu custo de computação (SEMAR SHAHUL; SINNEN, 2010), dado por

$$t_f(n) = t_s(n) + w(n). \tag{2.3}$$

Além disso, o escalonamento considera dois conjuntos de restrições relacionados aos processadores e precedência das tarefas. O sistema paralelo garante que um processador pode executar uma tarefa por vez. A precedência impõe a ordem das execuções devido às dependências $e \in E$ entre os nós $n \in V$ em G. Cada nó n_j só poderá iniciar sua execução se os seus predecessores, **pred**$(n_j) = \{n_i \in V : e_{ij} \in E\}$, estiverem finalizados. Porém, se os predecessores estiverem em processadores diferentes é necessário adicionar o tempo de comunicação (SEMAR SHAHUL; SINNEN, 2010). Logo, para n_i,$n_j \in V$,$e_{ij} \in E$ e $i \neq j$, temos

$$t_s(n_j) \geq t_f(n_i) + \begin{cases} 0 & \text{se } proc(n_i) = proc(n_j) \\ c(e_{ij}) & \text{caso contrário} \end{cases} \tag{2.4}$$

Figura 13 – Escalonamento de um grafo de tarefas.



(a) (b) (c)

Fonte: Traduzido livremente de Semar Shahul e Sinnen (2010)

A Figura 13 apresenta um exemplo de um escalonamento viável 13(b) e outro ótimo 13(c) do DAG em 13(a). Considerando que o momento inicial seja 0, nos gráficos de Gantt apresentados em 13(b) e 13(c), podemos expressar o tempo total de escalonamento sl como:

$$sl(\sigma) = max_{n \in V}\{t_f(n)\}. \tag{2.5}$$

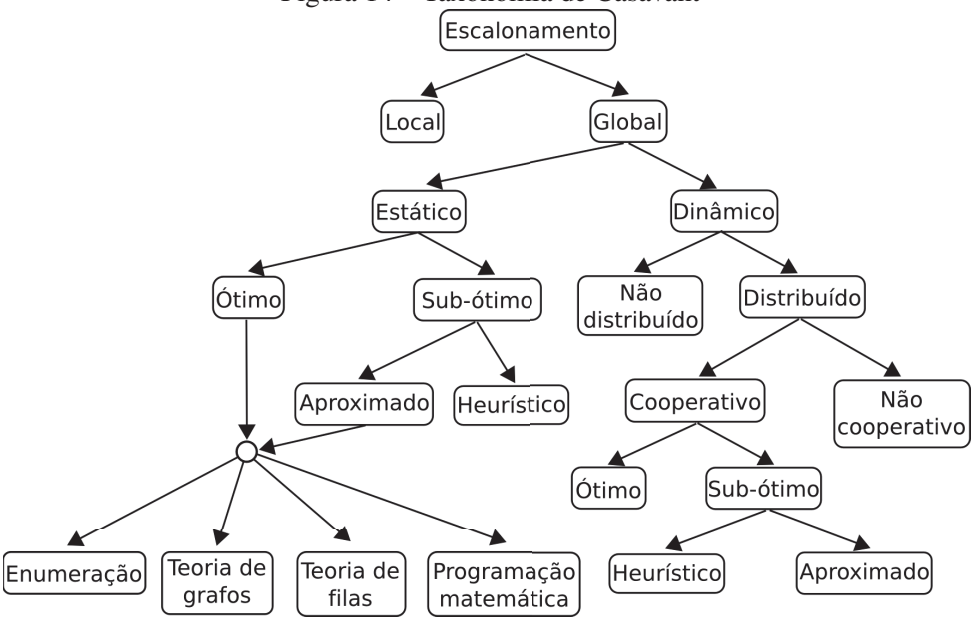
O principal objetivo de um escalonador é encontrar um escalonamento válido que minimize sl (YAMIN, 2001). Este problema de encontrar tal escalonamento é conhecido por ser NP-difícil (SEMAR SHAHUL; SINNEN, 2010; CATALYUREK *et al.*, 2009).

2.3 Taxonomia de Casavant

Uma classificação dos algoritmos de escalonamento foi proposta por Casavant e Kuhl (1988) e continua sendo adotada por trabalhos recentes como referência para indicar os tipos de algoritmos utilizados na literatura (GRAEBIN, 2012; LOWE; OROU, 2012; TANAKA; TATEBE, 2012; RODRIGUES, 2011). Esta taxonomia tem uma abordagem hierárquica e outra horizontal (YAMIN, 2001).

2.3.1 Classificação Hierárquica

Figura 14 – Taxonomia de Casavant



Fonte: Traduzido livremente de Casavant e Kuhl (1988)

Observando a taxonomia de Casavant, Figura 14, do ponto de vista hierárquico (de cima para baixo), o primeiro nível apresenta a grande divisão *Local* e *Global*. Casavant e Kuhl (1988) se referem ao fato do escalonador do sistema ter atuação sobre um único processador ou uma estrutura multiprocessada. Escalonadores inclusos na classe *Local* se referem a um sistema de processador único e o acesso aos recursos é realizado utilizando a técnica de *time-slicing*, na qual o escalonador reserva um intervalo de tempo para cada consumidor acessar o recurso disponível. A classe *Global* refere-se a sistemas com multiprocessadores, na qual o escalonador, que trata do problema de decisão para definir onde (qual processador) as tarefas devem ser executadas, deixa que o sistema operacional (SO) defina a política de escalonamento local. Abaixo da classe *Global*, há outra camada hierárquica, com duas grandes abordagens: *Estático* e *Dinâmico*.

A abordagem de escalonamento *Estático* considera o conhecimento prévio do sistema para-

lelo e o comportamento da aplicação, os tipos de processadores e suas características, o padrão de comunicação entre tarefas, o tempo de execução em cada nó do DAG e a quantidade de dados a serem processados. As definições de escalonamento são determinados em tempo de compilação em função destes parâmetros (DROST *et al.*, 2012). As distribuições de tarefas são realizadas ao início da aplicação e se mantêm até o fim da execução (YAMIN, 2001). Escalonamento estático é utilizado para problemas denominados **problemas estáticos**. Nesta classe de problemas estão a FFT, eliminação Gaussiana, etc. A escalabilidade de um escalonamento estático é restrito pois uma grande quantidade de espaço de memória é necessária para armazenar o DAG. Além disso, não é possível realizar uma redistribuição das tarefas tendo um desconhecimento da estrutura futura (DROST *et al.*, 2012).

Com o conhecimento das necessidades da aplicação e os recursos disponíveis, é possível propor uma solução estática *ótima* para o problema, realizando as combinações possíveis através de métodos computacionais adequados para isto. Porém, estes métodos podem ser computacionalmente caros devido à quantidade de testes necessários para avaliar todas as opções de arranjos possíveis, o que torna essa uma solução inviável (RIGHI, 2009; YAMIN, 2001). Deste modo, uma solução de escalonamento *sub-ótimo* pode ser proposta utilizando-se de dois tipos de tratamentos: *aproximado* ou *heurístico*. Uma solução aproximada utiliza os mesmos métodos para encontrar a solução ótima, porém, a condição de parada da busca é encontrar uma solução considerada "boa", evitando assim uma busca exaustiva. Nas soluções em que é possível definir as métricas "boas", a aproximação é uma alternativa viável (YAMIN, 2001). Já a solução de escalonamento heurístico faz uso de parâmetros genéricos que, de alguma forma, alteram o desempenho do sistema paralelo. Segundo Yamin (2001) tais parâmetros devem ser encontrados de forma simples. Os algoritmos heurísticos, possuem uma representação mais realística, por serem baseados no conhecimento prévio dos processos e da ocupação dos recursos disponíveis (DIAZ; MUOZ-CARO; NIO, 2009).

O escalonamento *Dinâmico*, também conhecido na literatura como *on-line* (GHOSH *et al.*, 2012), refere-se a necessidade de realizar o remapeamento das tarefas durante a execução da aplicação. Diferentemente do modo estático, na execução do modo dinâmico há pouco ou nenhum conhecimento sobre a aplicação executada ou sobre os recursos disponíveis (RIGHI, 2009; GHOSH *et al.*, 2012). Ou seja, a execução de novas tarefas são imprevisíveis e o remapeamento de consumidores e recursos é efetuado durante a execução da aplicação.

Descendo mais uma camada hierárquica na Figura 14, observamos que a decisão de escalonamento pode ser tratada em um único processador (fisicamente *não-distribuído*) ou depender dos demais processadores existentes no sistema paralelo (fisicamente *distribuído*). Neste último caso, existe a preocupação em atentar para a autorização da decisão global de escalonamento (CASAVANT; KUHL, 1988).

A classe fisicamente *distribuído* pode ser ainda dividida em *cooperativo* e *não-cooperativo*. O sentido de cooperação está relacionado a cada processador ser capaz, ou não, de contribuir para a tarefa de escalonamento global lidando com o seu escalonamento local (CASAVANT;

KUHL, 1988). Os pertecentes a classe *não-cooperativo* realizam o escalonamento local independentemente das ações dos outros processadores. Segundo Casavant e Kuhl (1988), o grau de autonomia que os processadores do sistema possuem para utilizar seus próprios recursos é a chave para esta classe. Cada processador atua como uma entidade autônoma e decide sem considerar os efeitos sobre o sistema paralelo. Nos sistemas pertencentes a classe *cooperativo* cada processador tem a responsabilidade de realizar o escalonamento local, mas todos os processadores tem o objetivo de cooperar para um resultado global. Como no caso do escalonamento estático, abaixo da classificação *cooperativo* estão as soluções ótimas, sub-ótimas e suas derivações.

2.3.2 Classificação Horizontal

Outro aspecto da taxonomia apresentada na Figura 14 é a classificação horizontal. Esta análise permite que as mesmas classificações apresentadas anteriormente possam ser divididas em:

- Adaptativo ou não-adaptativo
- Balanceamento de carga
- Licitação
- Probabilístico
- Atribuição única ou re-atribuição

O comportamento *adaptativo* se caracteriza pela política de escalonamento que muda seus parâmetros de acordo com a execução da aplicação. Em contraste, o *não-adaptativo* desconsidera qualquer alteração dos parâmetros. De acordo com Yamin (2001), a política de *balanceamento de carga* tem a função de fazer uma progressão dos processos através dos nós de modo uniforme. Isto é realizado compartilhando a informação dos processadores através da interconexão com uma determinada periodicidade, ou sob demanda, permitindo que todos os nós tenham uma visão global do sistema. Desta forma, é possível a cooperação entre eles, removendo as tarefas dos processadores mais carregados para outros com disponibilidade de recursos. O balanceamento de carga será detalhado na próxima seção.

No escalonamento por *licitação* os nós podem assumir o papel tanto de gerente como de contratante (YAMIN, 2001). As tarefas a serem compartilhadas são anunciadas pelos nós gerentes, e os contratantes anunciam a disponibilidade computacional para executar novos processos. Os gerentes têm autonomia para decidir para qual nó enviarão a tarefa a ser executada (os critérios para decisão podem ser ótimos, sub-ótimos ou heurísticos), porém, os nós contratantes podem rejeitar as tarefas destinadas a ele (YAMIN, 2001).

A classe de escalonamento *probabilístico* existe para resolver o problema no qual o espaço de soluções necessários é muito grande, tornando proibitivo uma abordagem analítica (YAMIN,

2001). A ideia principal é gerar aleatoriamente (de acordo com uma distribuição conhecida) um conjunto de diferentes escalonamentos. Então, este conjunto é analisado e um arranjo é escolhido utilizando algum parâmetro que permita se determinar uma definição de melhor dentre outros arranjos aleatoriamente gerados (CASAVANT; KUHL, 1988).

O escalonamento de *atribuição única* refere-se a aquele em que ocorre no momento em que o escalonador recebe as informações sobre a aplicação. As informações são utilizadas somente neste instante sem que ocorram alterações até o fim da aplicação. Já na *reatribuição dinâmica*, após o decorrer de uma execução parcial, os parâmetros são reavaliados e uma nova redistribuição das tarefas pode ser efetuada. A avaliação dos parâmetros de escalonamento pode ocorrer diversas vezes durante a execução do programa paralelo (YAMIN, 2001). Segundo Casavant e Kuhl (1988), a *reatribuição dinâmica* utiliza a migração de processos/tarefas/objetos para transferência dos estados atuais dos processos.

2.4 Balanceamento de Carga

O *Balanceamento de Carga* (BC) é a distribuição de processamento e comunicação em um sistema paralelo de modo que nenhum processador fique mais carregado do que outros. O objetivo do BC pode ser definido como:

"Dado uma coloção de tarefas que involvem computação, comunicação e um conjunto de computadores conectados em uma certa topologia, encontrar um mapeamento destas tarefas nos computadores tal que cada computador tem uma aproximadamente a mesma quantidade de computação e a quantidade de comunicação minimizada"(ZHENG, 2005).

Como já apresentado anteriormente em 2.3.1, o BC pode ser tratado como um subconjunto de escalonamento (ZHENG, 2005; ALAM; RAZA, 2012). O BC e o escalonamento possuem um papel importante em grades computacionais (MCHEICK; MOHAMMED; LAKISS, 2011). Este processo faz com que seja maximizado o desempenho da aplicação mantendo o tempo de ociosidade e a comunicação entre processos tão baixos quanto possível (DEVINE *et al.*, 2005). De acordo com Alam e Raza (2012) o BC resulta na alocação dos recursos do sistema para tarefas ou processos individuais durante um período que otimiza uma ou mais funções objetivas. O BC é especialmente importante para algumas aplicações paralelas que necessitam estarem sincronizadas para iniciar uma nova tarefa (RIGHI, 2009).

O balanceador de carga deve possuir algumas características: criar uma sobrecarga (*overhead*) reduzida no tráfico de dados; o algoritmo de BC deve gerar baixa sobrecarga na execução da aplicação; deve ser "justo" de modo que o nó altamente carregado seja equilibrado com o nó levemente carregado e deve utilizar o mínimo de tempo possível da CPU.

Como apresentado por Righi (2009), um importante tópico sobre o BC é a análise de como avaliar a carga no sistema paralelo. Existem diversas formas de se medir isto. Alguns exemplos são: a carga média dos processadores; a utilização dos processadores no momento atual; a utilização de E/S; a quantidade de CPU livre; a quantidade de memória livre e a quantidade de

comunicação entre processos. Ainda é possível considerar uma combinação desses parâmetros para a obtenção de uma solução com múltiplas métricas e de uma avaliação mais adequada da ocupação nos processadores do sistema.

Para o BC ser eficiente e não apresentar resultados indesejados é necessário obter a informação do desbalanceamento do sistema. Esta tarefa ocorre basicamente em duas etapas. A primeira é detecção do desbalanceamento e, a segunda, é o custo para realizar o balanceamento deve ser menor do que as vantagens proporcionadas à aplicação.

Segundo Righi (2009), o desbalanceamento pode ser detectado de forma *síncrona* ou *assíncrona*. O modo de detecção síncrona se aplica de forma natural a maioria das aplicações científicas devido a existência de barreiras criando pontos de sincronização entre as tarefas. Estes momentos podem ser utilizados para o lançamento do algoritmo de BC permitindo que as condições de todos os processadores do sistema paralelo sejam analisadas(RIGHI, 2009). A abordagem assíncrona baseia-se na existência de uma tarefa ou processo dedicado a analisar o histórico da carga no sistema. Caso esta análise constate que o desempenho da aplicação foi reduzido, ele pode fazer a solicitação para a execução do BC.

De forma ampla, o BC pode ser classificado como *centralizado* ou *descentralizado*, *estático* ou *dinâmico*, *periódico* ou *aperiódico*, com *threshold* ou sem *threshold* (ALAM; RAZA, 2012). O BC pode ser tanto uma ou a combinação das características anteriores podendo mudar o comportamento de acordo com a aplicação (ALAM; RAZA, 2012). O algoritmo pode possuir pontos de decisão (*thresholds*) de modo que se alguma métrica ultrapassar o valor predeterminado, o sistema reage redistribuindo a carga. Quando o valor da métrica reduz novamente, o sistema interrompe o processo de balanceamento (ALAM; RAZA, 2012). Ainda, um *threshold* adaptativo pode ser adotado permitindo que os recursos sejam utilizados de forma mais otimizada.

A periodicidade está relacionada à frequência de troca de informações. A estratégia periódica configura-se pela ação dos processadores que informam sua carga de trabalho para os outros em um intervalo de tempo pré-definido. Segundo Graebin (2012), um dos aspectos mais difíceis para implementar esta estratégia é definir o intervalo adequado, devido a possibilidade da criação de uma sobrecarga desnecessária no sistema. Já a estratégia aperiódica caracteriza-se pela transmissão de informações de carga, feitas pelo processador, sempre que ocorrer alguma variação significativa de algum parâmetro. Uma visão global do estado do sistema pode ser melhor observada na utilização desta estratégia. Contudo, pode ocorrer uma sobrecarga de comunicação caso a variação seja muito frequente. Devido a isso, em grandes ambientes esta abordagem pode ser impraticável.

Algoritmos de BC estáticos direcionam as tarefas de maneira probabilística ou determinística, sem considerar os eventos durante a execução da aplicação. Entretanto, apresentam-se eficientes com aplicações que não possuem muitas variações de carga (MCHEICK; MOHAMMED; LAKISS, 2011). O principal objetivo dos algoritmos estáticos é minimizar o tempo de execução reduzindo os atrasos de comunicação. Esta classe possui maior simplicidade de im-

plementação e uma sobrecarga reduzida sobre o programa principal. Além disso, o algoritmo de BC não necessita monitorar constantemente a execução da aplicação.

Algoritmos de BC dinâmicos realizam mudanças na distribuição das tarefas através dos processadores disponíveis utilizando informações mais recentes da aplicação. Estes algoritmos adicionam naturalmente uma sobrecarga maior ao programa principal devido ao custo de coleta dos dados relevantes e a manutenção destas informações. Por este motivo, é muito importante manter os limites dessas influências em valores que permitam a sua usabilidade (MCHEICK; MOHAMMED; LAKISS, 2011). Para a realização do BC dinâmico é necessário obter as repostas para as seguintes questões:

- Quem decide quando o processo de BC acontece?
- Quais informações são utilizadas para iniciar o processo de BC?
- Onde as decisões de BC são realizadas?

As respostas destas questões definem o tipo do BC. A questão *quem* define se a estratégia adotada é do tipo *Sender-Initiated*, *Receiver-Initiated* ou *Symmetric-initiated*. O modo *sender-initiated* se caracteriza por nós altamente carregados solicitarem a nós menos carregados o recebimento parte de suas tarefas. No modo *receiver-initiated* os nós menos carregados procuram por nós carregados e solicitam o recebimento de tarefas. *Symmetric-initiated* combina as duas estratégias anteriores (RODRIGUES, 2011). A informação que a resposta *quais* revela é se o algoritmo de BC é *Global* ou *Local*. Quando o BC é global considera-se todos os dados relevantes de todos os nós participantes do sistema paralelo. No modo local, os processadores podem ser agrupados por afinidades ou categorias e separadamente realizarem o BC. Embora o BC local produza um custo de comunicação menor, o modo de estratégia global resulta em uma melhor eficiência além de proporcionar uma convergência mais rápida (RIGHI, 2009). *Centralizado* ou *Distribuído* são respostas para a questão *onde*. Um BC centralizado apresenta um único nó mestre que realiza a decisão baseada nas informações coletadas. A estratégia distribuída se caracteriza pela cópia do balanceador de carga em todos os nós disponíveis. Em Rodrigues (2011) a classificação *híbrida* é definida por apresentar uma solução mais adequada para uma grande quantidade de processadores.

A aplicação pode oferecer o BC incorporado (*embedded*) (RODRIGUES, 2011). Porém, este tipo de solução geralmente se apresenta de forma especializada no código original impossibilitando a sua reutilização em outras aplicações. Em oposição a isto, o uso de arcabouços de BC permite isolar a aplicação da estratégia utilizada, garantindo a usabilidade posterior do método implementado.

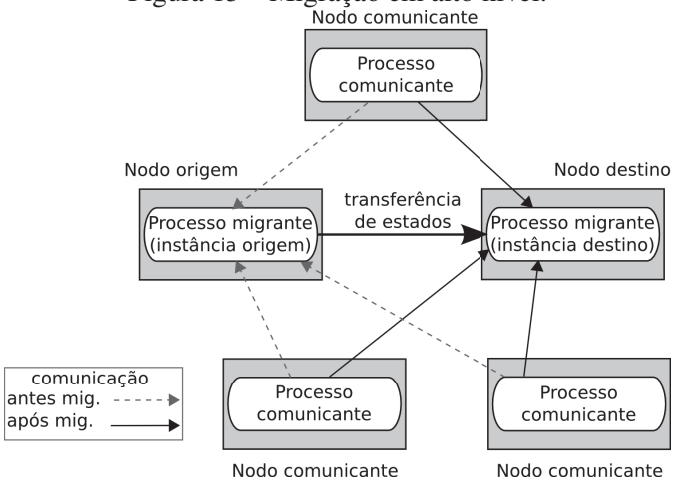
2.5 Migração de Processos

A forma mais flexível de se tratar o balanceamento de carga é com a utilização da técnica de migração (RODRIGUES, 2011; RIGHI, 2009). *Processo* é uma abstração do SO que repre-

senta uma instância de um programa em execução (MILOJICIC *et al.*, 2000; TANENBAUM, 2011). Um processo é composto por diversas entidades (PACHECO, 2011): um programa em linguagem de máquina; um bloco de memória, que contém o código executável, uma pilha de chamadas das funções ativas, um *heap* e algumas outras localizações de memórias; descritores de recursos alocados ao processo pelo SO; informações de segurança tais como os acessos permitidos de *hardware* e *software*; informações sobre os estados dos processos como, por exemplo, se ele está pronto para executar ou aguardando algum recurso e informações sobre a sua memória.

Um processo pode possuir uma ou mais *threads* de controle. A *thread*, ainda conhecida como processo leve, consiste em uma pilha de chamadas e conteúdos próprios mas compartilham do mesmo espaço de endereço e alguns estados específicos do SO, tais como os *signals*. O conceito de **tarefa** foi introduzido como uma generalização do conceito de processo. Em seguida, um processo é desacoplado em tarefa e um número de *threads*. Um processo tradicional é tratado como uma tarefa que possui uma *thread* (MILOJICIC *et al.*, 2000).

Figura 15 – Migração em alto nível.



Fonte: Traduzido livremente de Milojicic *et al.* (2000).

A **migração de processos** é a ação de transferir um processo que está em execução em um processador (origem) para outro (destino). A Figura 15 mostra uma visão em alto nível da migração de processos. A migração consiste em extrair o estado do processo no nó origem, transferí-lo ao nó destino e atualizar as conexões com os nós comunicantes. O estado transferido inclui o espaço de endereços do processo, ponto de execução, estado da comunicação e informações relevantes ao SO. Quando a migração ocorre, duas instâncias do processo existem: uma no nó origem e outro no nó destino. Esta segunda instância, será a que permanecerá com processo migrado. Segundo Milojicic *et al.* (2000), a migração de processos pode ser utilizada para algumas funções:

- **Balanceamento de carga dinâmico** – Através da migração de processos de um nó mais carregado para outro menos carregado;

- **Tolerância à falha** – Pela migração de processos de nós que possam ter ocorrido falha parcial;

- **Administração do sistema** – Realização de migração de processos de computadores que necessitem serem desligados por motivos administrativos;

- **Acesso a dados** – Migrar um processo para mais próximo da origem dos dados que estão sendo acessados.

A migração é dita transparente quando os efeitos dela são ocultadas do programador, sendo oferecida por implementações em *middleware* (GRAEBIN; RIGHI, 2012; MATEOS; ZUNINO; CAMPO, 2010; HUET; CAROMEL; BAL, 2004). Algumas delas são *Runtime Systems* (RTS) tais como MOXIS (KAUL *et al.*, 2010; MILOJICIC *et al.*, 2000), *Charm++* (ZHENG, 2005; KALE; KRISHNAN, 1993), *ProActive* (BAUDE *et al.*, 2002; GRAEBIN; RIGHI, 2012) e *Kerrighed* (SANDHYA; RAJU, 2011; RODRIGUES, 2011).

O tempo da operação de migração pode ser consideravelmente alto devido à quantidade de informação a ser transferida. Para a aplicação em balanceamento de carga, a migração de processos traz algumas limitações como o aumento da complexidade do RTS. Ainda, dado que a imagem de um processo é grande, um significante custo de comunicação é necessário, o que acarreta em um atraso no processo de migração. Além disso, mover um novo processo pode causar um desbalanceamento no nó destino (ZHENG, 2005). A alternativa que tem demostrado resultados interessantes em alguns trabalhos é a *migração de objetos*. Isso ocorre porque, neste caso a quantidade de informação é menor do que a quantidade de informação necessária para a transferência de um processo (GRAEBIN; RIGHI, 2012; KALE; ZHENG, 2009; ZHENG, 2005; HUET; CAROMEL; BAL, 2004; KALE; KRISHNAN, 1993). Para a migração com o fim de balanceamento de carga (independente de ser processo, tarefa ou objeto), é necessário atentar para os 3 componentes apresentados na Tabela 2.

Tabela 2 – Componentes para a Migração

Componentes	Descrição
Política de BC	Indica "Quais", para "Onde" e "Quando" os processos/tarefas/objetos devem ser migrados
Mecanismo de migração	A partir da aplicação (Proativa ou subjetiva) ou ação externa (Reativa, objetiva ou forçada)
Execução remota	Responsável por iniciar a execução no novo destino

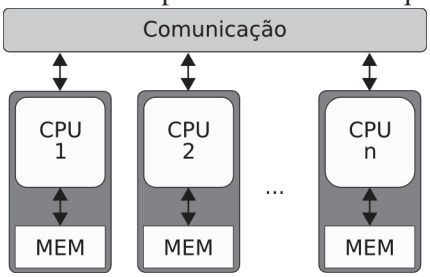
Fonte: Traduzido livremente de Righi (2009)

2.6 O Modelo Bulk-Synchronous Parallel

O modelo *Bulk-Synchronous Parallel* (BSP) foi introduzido por Valiant (1990) com o objetivo de definir uma forma geral de desenvolvimento de sistemas paralelos, descrevendo tanto a

estrutura da aplicação como a de recursos computacionais. O BSP é apresentado com o intuito de ser independente de arquitetura. Assim, não é necessário alterar o código para a mudança de plataforma (RIGHI, 2009). O modelo de uma máquina BSP é mostrado na Figura 16 de forma simplificada. Segundo Krizanc e Saarimaki (1999), a arquitetura paralela deve possuir alguns elementos: (i) processador e/ou memória, (ii) comunicação ponto-a-ponto entre os elementos e (iii) uma estrutura que permita a sincronização de todos ou do grupo. Na Figura 16, estes componentes são representados por processadores e cada um com sua memória local. Uma tecnologia de rede, que permita a comunicação global, atende os itens (ii) e (iii). É importante enfatizar que o modelo proposto não faz restrição nem de proximidade dos processadores, nem em relação ao tipo de rede adotada (RIGHI, 2009).

Figura 16 – Visão simplificada de uma máquina BSP.



Fonte: Traduzido livremente de Righi (2009).

De acordo com Skillicorn, Hill e McColl (1997), as vantagens fundamentais deste modelo são:

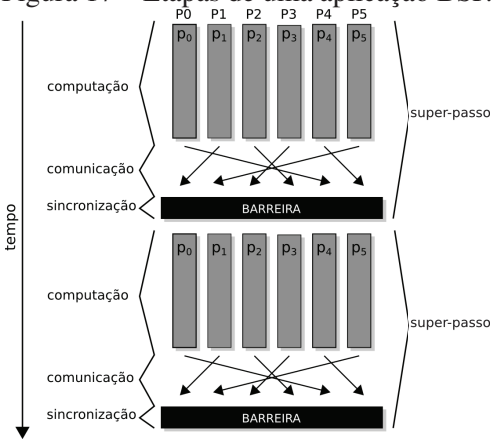
- Escrita simples;
- Independência da arquitetura;
- O desempenho do programa em uma dada arquitetura é previsível.

Segundo Righi (2009), a escrita de programas BSP se assemelha com a de programas sequenciais, onde pouca informação é necessária para indicar a paralelização. O modelo BSP consiste em uma sequência de iterações denominadas *super-passos* (*supersteps*). Cada super-passo é separado por uma etapa global e uma sequência deles ocorre até o algoritmo terminar (SRIVATSA; KAWADIA; YANG, 2012). Os super-passos são compostos por 3 fases: computação local, comunicação global e barreira de sincronização.

As fases de um super-passo estão ilustradas na Figura 17 relacionadas com o tempo. A etapa de computação local representa a execução em cada processador utilizando as informações de sua memória local. A movimentação dos dados entre os processadores é realizada somente durante a etapa de comunicação global e a barreira de sincronização garante que todas as transferências sejam realizadas antes de dar início a um novo super-passo.

Conforme Gerbessiotis e Siniolakis (2001), o sistema paralelo é caracterizado por:

Figura 17 – Etapas de uma aplicação BSP.



Fonte: Elaborado pelo próprio autor.

- O número de processadores n;
- Um fator de conversão g que transforma o custo de comunicação para um custo computacional;
- O mínimo tempo necessário para sincronizações l.

Cada processador possui um número de mensagens recebidas, h_{in}, e uma quantidade de mensagens enviadas, h_{out}. Se denotarmos $h = \max\{h_{in}, h_{out}\}$, então *relação-h* define o padrão de comunicação no processador. O parâmetro g refere-se a capacidade de um processador enviar, de forma uniformemente distribuída, dados para os demais componentes da rede de comunicação. Este parâmetro está relacionado a fatores como: protocolos de comunicação, gerenciamento de *buffer* de ambos processadores envolvidos na transação e a estratégia de roteamento utilizado pela rede de comunicação. Se definirmos um super-passo S qualquer e w como o tempo necessário para cada processador P realizar as operações locais durante o intervalo de S, podemos determinar o tempo total do super-passo S como:

$$t_S = \max\{w\} + g \times h + l, \tag{2.6}$$

e o tempo total da aplicação é dado por:

$$t = \sum_{n=1}^{N} \max\{w_s sn\} + g \times \sum_{n=1}^{N} h_n + N \times l, \tag{2.7}$$

onde N é o número total de super-passos realizados. Dadas as Equações 2.7 e 2.6, é possível prever o tempo total de execução estimando os valores de w e h (provenientes do código) e os valores g e l (dependentes do sistema paralelo).

Em um primeiro momento, a barreira de sincronização pode parecer um processo muito

custoso, mas ela traz algumas vantagens. As barreiras fazem com que um programa BSP não possua interdependência entre os dados (RIGHI, 2009) e, além disso, estes intervalos podem ser utilizados para introduzir recursos de tolerância a falhas. A cada super-passo pode-se avaliar os estados das tarefas e, utilizando a técnica de *checkpoints*, retornar a etapa anterior no caso de algum problema ser detectado (BONORDEN, 2007). O desempenho de uma aplicação BSP pode ser alterado se atuarmos em qualquer uma das três características apontadas na Tabela 3.

Tabela 3 – Mecanismos de desempenho de aplicações BSP.

Mecanismo	Descrição
Equilíbrio das tarefas	Dado que o tempo w faz parte do custo em cada super-passo, distribuir de maneira uniforme os processos de forma a realizar um equilíbro de carga entre os processadores para reduzir as diferenças entre o término da tarefa mais rápida e a mais lenta.
Reduzir o custo	Reduzir a distância (custo) entre os processos que das comunicações possuem comunicação intensiva, dado que gh produz o tempo que relaciona as condições de enlaces, quantidades de saltos e congestionamentos.
Redução a quantidade	Sendo l um valor dependente da barreira, a redução de super-passos diminui o custo da parcela Nl.

Fonte: Traduzido livremente de Skillicorn, Hill e McColl (1997).

A maioria dos programas BSP é tratada como *Single Program Mutilple Data* (SPMD), predominantemente com utilização de linguagem C ou Fortran seguindo as recomendações MPI (HUAN; QI-LONG; RUI, 2011; SKILLICORN; HILL; MCCOLL, 1997; RIGHI, 2009). O BSP é um arcabouço geral e diversos algoritmos distribuídos têm adotado este modelo (SRIVATSA; KAWADIA; YANG, 2012).

2.7 MigBSP

O modelo de reescalonamento de processos **MigBSP** foi introduzido por Righi (2009) como uma ferramenta para auxiliar nas decisões de balanceamento de carga em aplicações BSP com atividade dinâmica e ambiente heterogêneo. Esta ferramenta responde as questões necessárias para a ativação da migração de processos (Tabela 2): *quais*, *onde* e *quando*. O modelo utiliza as etapas de barreiras de sincronização das aplicações BSP para ajustar a localização dos processos e diminuir o custo de comunicação entre eles com o objetivo de reduzir o tempo dos super-passos (DA ROSA RIGHI *et al.*, 2010). As informações são coletadas durante a execução da aplicação e utilizadas para compor o valor do *potencial de migração* (PM).

Atuando sobre dois dos mecanismos apresentados na Tabela 3, equilíbrio na execução das tarefas e redução de custo de comunicação, o MigBSP oferece um balanceamento de carga automático em camada de *middleware*. Ele utiliza uma abordagem reativa e é executado sem o conhecimento da aplicação.

Com o objetivo de ser menos intrusivo possível, o MigBSP traz dois fatores adaptativos: baseado no balanceamento dos processos e no controle do intervalo de reescalonamento baseado

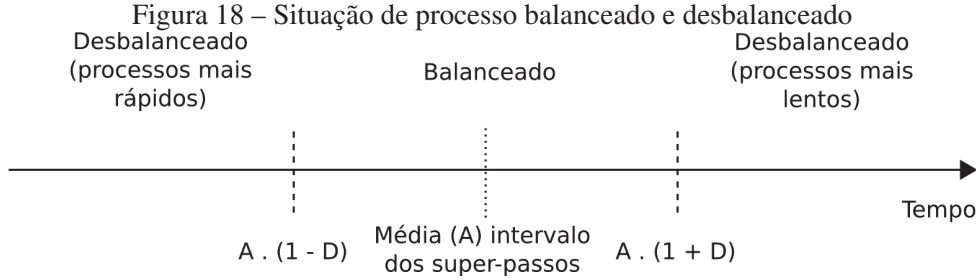
no número de chamadas sem a necessidade de migração.

O desbalanceamento é detectado se uma das Inequações 2.8 ou 2.9 for falsa. O valor de D indica a tolerância percentual para a detecção de processos instáveis e é definido no início da aplicação. A Figura 18 apresenta graficamente estas situações.

$$\textit{tempo do processo mais lento} < \textit{tempo médio dos processos} \times (1 + D) \tag{2.8}$$

$$\textit{tempo do processo mais rápido} > \textit{tempo médio dos processos} \times (1 - D) \tag{2.9}$$

Figura 18 – Situação de processo balanceado e desbalanceado



Fonte: Adaptado de Da Rosa Righi *et al.* (2010)

O fator de ajuste (i) acontece na variação do parâmetro D. O princípio é aumentar o valor de D a cada ω consecutivas barreiras e nenhuma migração acontecer. O incremento de D permite uma variação maior dos tempos de término dos processos em cada super-passo. Entretanto, enquanto ocorrer alguma migração, o valor de D é reduzido apresentando uma rigidez maior na detecção do desbalanceamento dos processos. Segundo Da Rosa Righi *et al.* (2010), este ajuste é importante principalmente nas situações onde ocorre custos altos de migração. O Algoritmo 1 apresenta como é variado o valor de D.

Algoritmo 1: Cálculo de D

```
1 λ ← reescalonamento sem migração
2 Se λ ≥ ω Então
3 |   Se D + D/2 ≤ 1 Então
4 |   |   D ← D + D/2;
5 |   Fim Se
6 Fim Se
```

O fator (ii) considera contabilizar a quantidade de super-passos $\alpha(\alpha \in N^*)$ entre um reescalonamento e outro e, ao passo que não existe intervenção, aumenta-se o valor de α de modo a evitar intrusões desnecessárias. Após a um desbalanceamento ser detectado, o valor de α é reduzido para que as decisões de migração ocorram mais rapidamente e, assim, permitam uma estabilização mais rápida do sistema. O Algoritmo 2 apresenta como é calculado o valor de α.

Como dito anteriormente, a definição de qual processo será migrado ocorre através do valor PM. A cada barreira de sincronização, n funções $PM(i, j)$ são computadas para o processo i, onde n é a quantidade de processadores de destinos possíveis e j representa um destino

Algoritmo 2: Cálculo de α

```
1  Para i ← superpasso k Até superpasso k + α − 1 Faça
2      Se Processos balanceado Então
3          α' = α + 1;
4      Senão
5          Se α' > α inicial Então
6              α' = α − 1;
7          Fim Se
8      Fim Se
9  Fim Para
10 RunRescheduling() ;
11 α = α' ;
```

específico. A utilização das métricas de computação $Comp(i,j)$, comunicação $Comm(i,j)$ e memória $Mem(i,j)$ permitem uma análise de $PM(i,j)$ como um somatório no qual $Comp$ e $Comm$ possuem uma influência favorável a migração enquanto Mem representa um fator contrário. Assim, a Equação 2.10 mostra como PM é encontrado (DA ROSA RIGHI *et al.*, 2010).

$$PM(i,j) = Comp(i,j) + Comm(i,j) - Mem(i,j) \tag{2.10}$$

A métrica $Comp$ tem o objetivo de simular o desempenho do processo i no processador j. Este valor é calculado coletando as informações de tempo de execução (CT) e instruções (I_t) realizadas em cada super-passo $t(k \leq t \leq k + \alpha - 1)$. O valor de I_t é utilizado para definir o *padrão de computação* de i ($P_{comp}(i)$). Este valor de $P_{comp}(i)$ pode variar de 0 até 1, sendo 0 a indicação de uma grande variância da quantidade de instruções e 1 representado um valor contínuo. Baseado no conceito *Aging* (TANENBAUM, 2003), $PI_t(i)$ é definido como o valor que prevê o comportamento do valor de I_t em um super-passo. Ele é calculado considerando os valores I_t dos super-passos anteriores a partir da Equação 2.11.

$$PI_t(i) = \begin{cases} I_t(i) & \text{se } t = k \\ \frac{1}{2}PI_{t-1}(i) + \frac{1}{2}I_t(i) & \text{se } k < t \leq k + \alpha - 1 \end{cases} \tag{2.11}$$

Este método economiza memória e tempo de cálculo por considerar apenas as informações dos dois últimos valores de tempo, dado que estes são os mais significativos (RIGHI; PILLA; CARISSIMI, 2008). Porém, o valor de $P_{comp}(i)$ se mantém independente da quantidade de reassociações de processos realizadas. O Algoritmo 3 mostra como sua atualização ocorre.

Algoritmo 3: Cálculo do padrão do processo $P_{comp}(i)$

```
1 Para t from superpasso k Até superpasso k − α − 1 Faça
2     Se PI_t(i) ≥ I_t(i) × (1 − σ) E PI_t(i) ≤ I_t(i) × (1 + σ) Então
3         Incrementa P_comp(i) em 1/α até 1;
4     Senão
5         Decrementa P_comp(i) em 1/α até 0;
6     Fim Se
7 Fim Para
```

A computação do processo i é considerada estável se a previsão estiver dentro de uma mar-

gem σ. Para determinar o valor de $Comp(i,j)$ é necessário ainda definir a previsão do tempo de execução dado por $CTP_{k+\alpha-1}(i)$. Semelhante ao cálculo de PI, o valor é dado considerando o conceito de Aging, representado na Equation 2.12.

$$CTP_t(i) = \begin{cases} CT_t(i) & \text{se } t = k \\ \frac{1}{2}CTP_{t-1}(i) + \frac{1}{2}CT_t(i) & \text{se } k < t \leq k+\alpha-1 \end{cases} \tag{2.12}$$

O valor que define o desempenho do processador destino j é representado por $ISet_{k+\alpha-1}(i)$. Este valor é normalizado de acordo com o valor teórico de cada processador. Utilizando os valores de P_{comp},CTP e $ISet$ definimos $Comp(i,j)$ como:

$$Comp(i,j) = P_{comp}(i) \times CTP_{k+\alpha-1}(i) \times ISet_{k+\alpha-1}(j) \tag{2.13}$$

A métrica que considera a comunicação entre os processos, $Comm$ é dada por:

$$Comm(i,j) = P_{comm}(i,j) \times BTP_{k+\alpha-1} \tag{2.14}$$

Para ponderar a comunicação entre os processos, o MigBSP considera apenas as recepções envolvidas provenientes do processador j para o processo i (RIGHI, 2009). $Comm(i,j)$ é um valor entre 0 e 1. Como o padrão de computação, o padrão de comunicação $P_{comm}(i,j)$ utiliza o cálculo para previsão de quantidade de dados a serem ainda transportados de j para i, $PB(i,j)$.

$$PB_t(i) = \begin{cases} B_t(i) & \text{se } t = k \\ \frac{1}{2}PB_{t-1}(i) + \frac{1}{2}B_t(i) & \text{se } k < t \leq k+\alpha-1 \end{cases} \tag{2.15}$$

Neste contexto, $B_t(i,j)$ indica a quantidade de *bytes* recebidos por i do processador j. A previsão dos tempos de comunicação é representada por $BTP_{k+\alpha+1}$ que é demonstrado na Equação 2.16. Análogo a σ, β é o valor de aceitação de variação no padrão de comunicação. O Algoritmo 4 representa como os valores de $P_{comm}(i,j)$ são obtidos.

$$BTP_t(i) = \begin{cases} BT_t(i) & \text{se } t = k \\ \frac{1}{2}BTP_{t-1}(i) + \frac{1}{2}BT_t(i) & \text{se } k < t \leq k+\alpha-1 \end{cases} \tag{2.16}$$

Algoritmo 4: Cálculo do padrão do processo $P_{comp}(i)$

```
1 Para t from superpasso k Até superpasso k − α − 1 Faça
2     Se (1 − β) × B_k(i,j) ≤ PB_k(i,j) E (1 + β) × B_k(i,j) ≥ PB_k(i,j) Então
3         Incrementa P_comm(i) em 1/α até 1;
4     Senão
5         Decrementa P_comm(i) em 1/α até 0;
6     Fim Se
7 Fim Para
```

A parcela $Mem(i,j)$ considera o sobrecusto de migração. Avalia-se a quantidade total de memória utilizada pelo processo, $M(i)$, no momento da ativação do balanceamento de carga, o

tempo de transferência de um *byte*, $T(i,j)$, e é adicionado o custo de migração em si $Mig(i,j)$.

$$Mem(i,j) = M(i) \times T(i,j) + Mig(i,j) \tag{2.17}$$

Quanto menor o valor de Mem, mais vantagioso será migrar o processo i para j devido a um sobrecusto menor. Em relação aos valores de $Comp$ e $Comm$, esta relação é inversa. Valores mais altos melhoram o valor de PM.

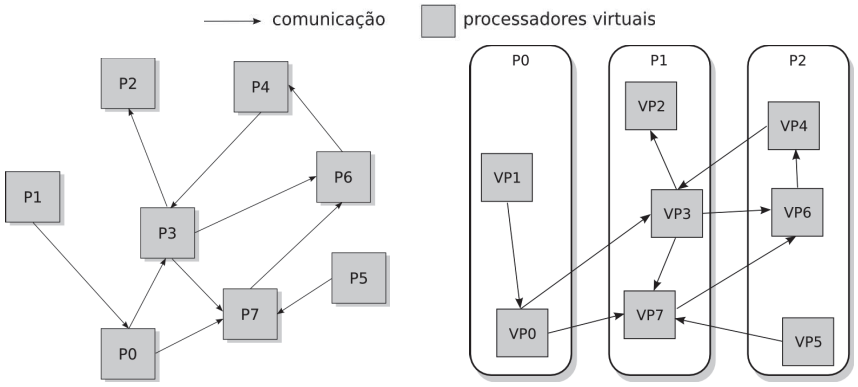
Com a informação do PM de cada processo, a escolha dos candidatos a migração é efetuada através de uma entre duas heurísticas: (i) escolha do processo com maior valor de PM ou (ii) escolha dos processos que possuem PM maior do que $Max(PM).p$, onde p é uma porcentagem definida na inicialização da aplicação.

2.8 Charm++ e AMPI

Charm++ é um *Runtime System* (RTS) para aplicações paralelas que auxilia o desenvolvedor na criação de programas sem a preocupação com questões complexas, tais como o acesso às interfaces de comunicação, escalonamento e migração de processos (KALE; ZHENG, 2009). Programas desenvolvidos em *Charm++* podem ser executados em todo tipo de sistemas MIMD sem a necessidade de mudanças em código devido a portabilidade que o sistema possui (KALE; KRISHNAN, 1993). Segundo Kale e Krishnan (1993), *Charm++* é basicamente C++ sem variáveis globais e com algumas extensões para suportar a execução paralela.

Uma das importantes características do *Charm++* é a virtualização de processadores (ZHENG, 2005). O programador realiza o desenvolvimento sem se preocupar com a quantidade de processadores reais que o sistema possui. O desenvolvimento continua do mesmo modo em MPI, porém sem a limitação da quantidade de processos a serem lançados pela aplicação.

Figura 19 – Virtualização dos Processos



(a) visão do usuário (b) visão do sistema

Fonte: Adaptado de Zheng (2005).

Como apresentado na Figura 19, do ponto de vista do programador, o problema é dividido

em N objetos migráveis que serão executados em P processadores. N é independente de P podendo se ter $N \gg P$. Cada um dos N objetos é tratado pelo *Charm++* como um processador virtual (VP) independente da quantidade de processadores reais. Desta forma, o usuário realiza a interação dos objetos (processadores virtuais) sem precisar se preocupar com a real disposição dos processos no sistema paralelo. Como apenas um processo é executado em um VP, o conceito de processo e VP se confundem.

A Figura 19(a) demonstra a visão que o usuário tem em relação ao sistema paralelo. O programador desenvolve sua aplicação paralela realizando o tratamento de tarefas de modo convencional, sem se preocupar com a limitação real dos processadores existentes no sistema. Na Figura 19(b), o RTS se encarrega de realizar a distribuição adequada dos processos virtuais utilizados e de fazer o encaminhamento correto das mensagens entre os processos, abstraindo do programador a localização real do processo.

O objeto base para execução paralela é um *chare* que possui métodos que podem ser invocados assincronamente por outros *chares*. O tempo médio de execução de um método fica na ordem das dezenas de microsegundos (KALE; ZHENG, 2009). A execução remota dos métodos em um *chare* é acionada através da recepção de uma mensagem proveniente de outro objeto. Estas invocações são naturalmente assíncronas evitando que os processos fiquem interrompidos.

O RTS *Charm++* possui um escalonador próprio para controlar o encaminhamento das mensagens. Como resultado disso, a movimentação de um objeto entre processadores é desconhecida dos outros *chares*.

Adaptive MPI (AMPI) é uma implementação aprimorada das definições MPI (MPI STANDARD, 2013) desenvolvida para utilizar os recursos do RTS *Charm++*. A AMPI implementa processos MPI virtualizados utilizando *threads* migráveis de nível-usuário. Diferente de um processo MPI tradicional, os VPs são produzidos com menos memória e podem ser iniciados mais rapidamente (ZHENG; KALE; LAWLOR, 2006).

As vantagens da virtualização são listadas abaixo (HUANG *et al.*, 2006):

- Sobreposição da comunicação e computação - Se um VP estiver aguardando uma recepção, outro VP que esteja no mesmo processador físico pode continuar executando. Isto elimina a necessidade do programador realizar esta tarefa, normalmente necessária em MPI;

- Balanceamento de carga automático - Caso algum processador físico fique sobrecarregado, o RTS pode migrar alguns VPs para os processadores menos carregados automaticamente;

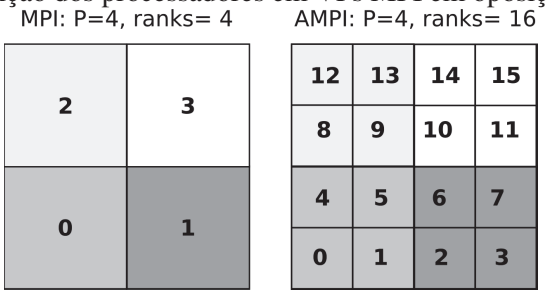
- Flexibilidade para executar em um número arbitrário de processadores - Dado que mais de um VP pode ser executado em processador físico, a AMPI permite a execução em um número arbitrário de processadores. Esta característica é útil para o processo de desenvolvimento e a fase de depuração;

- Suporte a biblioteca de comunicação otimizada - Além de fazer uso da camada de comunicação provida pelo RTS *Charm++*, a AMPI fornece comunicação nativamente assíncrona, não bloqueante e interfaces de comunicação coletiva permitindo a possibilidade de sobreposição das operações;

- Melhor desempenho de cache - Um VP lida com um conjunto de dados menor do que um processador físico. Isto garante que o VP tenha uma melhor ***memory locality***. Este efeito bloqueante é o mesmo método que muitas otimizações de *caches* utilizam, e os programas AMPI desfrutam deste benefício automaticamente.

A Figura 20 apresenta uma comparação entre a decomposição das tarefas em VPs e modelo tradicional de implementação MPI. O RTS *Charm++* divide o potencial de cada processador (com a utilização das *threads*) de modo a alocar os VPs de forma transparente. Desta forma, ocorre a sobreposição da comunicação e tarefas. Apesar da linguagem C++ proporcionar a vantagem de invocação remota assíncrona, tem-se demonstrado que a mais importante característica explorada pelos desenvolvedores é a utilização do RTS para mapeamento automático das tarefas nos processadores reais (KALE; ZHENG, 2009).

Figura 20 – Decomposição dos processadores em VPs MPI em oposição ao MPI tradicional.



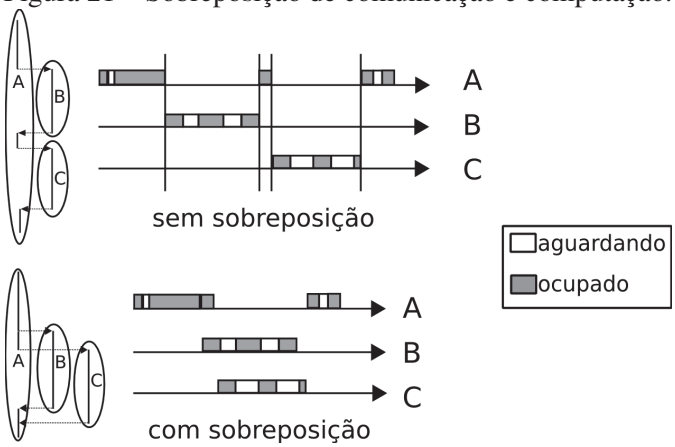
Fonte: Manual *on-line* da biblioteca AMPI (UNIVERSITY OF ILLINOIS, 2013a).

Segundo Kale e Zheng (2009) a mudança de um código MPI para AMPI pode ser realizada de forma automática na grande maioria dos casos. Em situações onde há utilização de variáveis globais ou estáticas, apenas o encapsulamento destas será necessário para que a alteração seja realizada.

2.8.1 Sobreposição da Comunicação e Computação

A sobreposição da computação com a comunicação representa a redução de custo (tempo) computacional. Na Figura 21, duas situações demonstram a diferença entre o comportamento de uma aplicação MPI e AMPI. “A”, “B” e “C” são tarefas distribuídas pelos processadores de um sistema paralelo. O processo “A” deve realizar uma solicitação ao “B” e ao “C”, e não há interdependência entre “B” e “C”. No estilo tradicional MPI “A” deve realizar a chamada de

Figura 21 – Sobreposição de comunicação e computação.



Fonte: Traduzido livremente de Kale e Zheng (2009).

"B" e aguarda sua resposta. Após a resposta de "B" para a sua requisição, "A" poderá realizar a chamada de "C". Deste modo, a computação global fica bloqueada aguardando o evento de recepção para permitir a continuidade da tarefa.

Nas aplicações AMPI, o processo "A" invoca o processo "B" e retorna imediatamente após o envio da mensagem. Em seguida, realiza uma nova invocação para "C". Como mostrado na Figura 21, "C" e "B" sobrepõem automaticamente os períodos de espera de cada um.

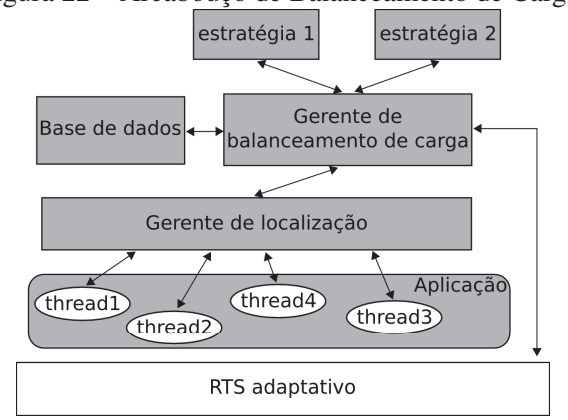
No VP de origem, executa-se um processo que deseja enviar informações a um outro processo pertencente a um VP de destino. A função *MPI_Send()* realiza a entrega da mensagem ao RTS e, assim, permite o retorno imediato desta função e a continuidade da tarefa em execução. Desta forma não é necessária a existência da função *MPI_Recv()* no VP de destino para garantir a entrega. Quando o VP destino alcançar a sua função *MPI_Recv()*, o RTS realizará a entrega da mensagem previamente enviada. Se o VP de origem ainda não tiver enviado a mensagem esperada, o VP de destino irá aguardar a sua disponibilidade para continuar a sua execução.

2.8.2 Adaptação à Variação de Carga

O comportamento dinâmico de aplicações paralelas geralmente ocasiona o desbalanceamento de carga. Neste contexto, o RTS *Charm++* explora o *princípio da persistência* para decidir quando ativar automaticamente o processo de balanceamento de carga. O princípio da persistência diz o seguinte: *"Uma vez a aplicação sendo expressa em termos de seus objetos (como VP) e suas interações, tempos de computação e padrão de comunicação (número e tamanho de mensagens trocada) tendem a persistir durante o tempo"* (KALE; ZHENG, 2009).

A Figura 22 ilustra os componentes do arcabouço de balanceamento de carga. Este arcabouço coleta estatísticas de processamento e comunicação de cada objeto e os registra na sua

Figura 22 – Arcabouço de Balanceamento de Carga.



Fonte: Adaptado de Kale e Zheng (2009).

base de dados. No topo, observa-se os blocos de estratégias. O usuário pode escolher uma delas para conectar ao arcabouço durante o lançamento da aplicação. Ao centro está o gerente de balanceamento de carga que possui uma tarefa importante. Ao informar o momento do balanceamento de carga, as estratégias em cada processador utilizam a informação recuperada da base de dados local e a situação dos processadores e dos objetos atribuídos a ele. Dependendo da estratégia, também é possível a comunicação com outros gerentes locais para se obter as suas informações coletadas. Assim, com toda informação disponível, as estratégias definem quais e para onde migrar os VP. Então, o gerente de balanceamento de carga é informado sobre migrações e supervisiona as movimentações dos objetos. Quando este processo termina, a estratégia sinaliza ao gerente de balanceamento para dar continuidade aos processos (KALE; ZHENG, 2009).

Em aplicações AMPI, o RTS poderia realizar a detecção e invocar automaticamente o balanceamento de carga assim como em aplicações desenvolvidas diretamente em *Charm++*. Entretanto, se for necessário realizar algum empacotamento especial para o envio de variáveis globais ou estáticas (necessário para manter compatibilidade com aplicações MPI nativas) seria imprudente que acontecesse alguma migração em uma etapa desconhecida. Para evitar este problema, a AMPI provê a função *MPI_Migrate()* com este propósito. Aplicações no modelo BSP possuem pontos (as barreiras de sincronização) que podem ser utilizados para a utilização desta chamada. A função *MPI_Migrate()* não diz ao RTS para realizar a migração mas indica quando a estratégia deve ser chamada para a avaliação do sistema e efetuar as medidas necessárias.

A indústria tem adotado a utilização do *Charm++*/AMPI em aplicações paralelas e com a necessidade de escalabilidade. Exemplos disso são o NAMD (PHILLIPS *et al.*, 2005), OpenAtom (WHITFIELD; MARTYNA, 2006; BHATELÉ; BOHM; KALÉ, 2009), ChaNGa (JETLEY *et al.*, 2010, 2008) e outras utilizações em programas de simulação 3D para propagação de foguetes e estilhaços (JIAO *et al.*, 2006, 2005). O NAMD em especial, é uma aplicação utilizada por milhares de cientistas (KALE; ZHENG, 2009) e ganhou os prêmios **Gordon Bell** e **Sidney**

Fernbach, nos anos 2002 e 2012 respectivamente.

3 TRABALHOS RELACIONADOS

Este capítulo mostra as frentes de pesquisas mais atuais, e algumas clássicas, relacionadas ao assunto de balanceamento de carga utilizando arcabouços e técnicas de migração de processos. O objetivo é apresentar um panorama geral de quão relevante é o assunto abordado e como o módulo proposto pode ser utilizado para pesquisas futuras. As seções a seguir apresentam alguns trabalhos recentes e outros clássicos a respeito de bibliotecas BSP, balanceamento de carga e migração de processos. Entretanto, é possível observar uma grande relação entre eles, pois não há uma separação específica entre os assuntos tratados. Desta forma, eles foram separados de acordo com a abordagem mais presente em cada um deles.

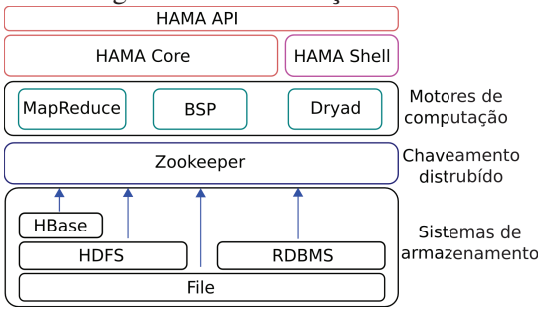
3.1 Bibliotecas BSP

Algumas ferramentas computacionais ajudam no desenvolvimento de aplicações *Bulk Synchronous Parallel* (BSP). Dentre elas, estão algumas que se comportam como APIs e outras contém um *Runtime System* (RTS) completo que auxilia o desenvolvimento de tarefas complicadas como, por exemplo, a migração de processos.

3.1.1 HAMA

Zhang e Ge (2012) apresentaram um trabalho que faz uso do *HAMA* (HAMA, 2013) para o estudo de um algoritmo paralelo que encontra comunidades em redes baseados em grafos ponderados. Nele, o algoritmo proposto apresentou um aprimoramento em relação a trabalhos anteriores. Outro estudo que utiliza a mesma tecnologia para desenvolver aplicações BSP foi apresentado por Ting, Lin e Wang (2011). Este último se refere a criação de um armazenamento de dados para redes sociais, baseado em nuvem e sua análise. Os autores realizaram uma comparação entre o HAMA BSP e o algoritmo MapReduce (MING *et al.*, 2011) e apresentaram um resultado favorável a implementação em BSP.

Figura 23 – Arcabouço HAMA



Fonte: Traduzido livremente de Seo *et al.* (2010).

HAMA é um arcabouço puramente BSP, baseado na linguagem Java, voltado para aplicações de troca de mensagens e comunicação coletiva. Como mostrado na Figura 23, o HAMA se encontra no topo do *Hadoop Distributed File System* (HDFS) (HADOOP, 2013) e faz uso de seus recursos como, por exemplo, o balanceamento de carga (CHUNG *et al.*, 2012; FAN *et al.*, 2012).

De acordo com Seo *et al.* (2010), a arquitetura do HAMA pode ser separada em três camadas: (i) *HAMA Core*; (ii) o *HAMA Shell* e; (iii) *HAMA API*. A camada HAMA Core fornece as ferramentas para computação de matrizes e grafos. Ela também determina qual o motor de computação mais apropriado para se utilizar. Os motores possíveis são o *MapReduce*, BSP e *Dryad* (LI; FOX; QIU, 2012). HAMA Shell permite a interação do usuário. As contribuições do arcabouço HAMA são:

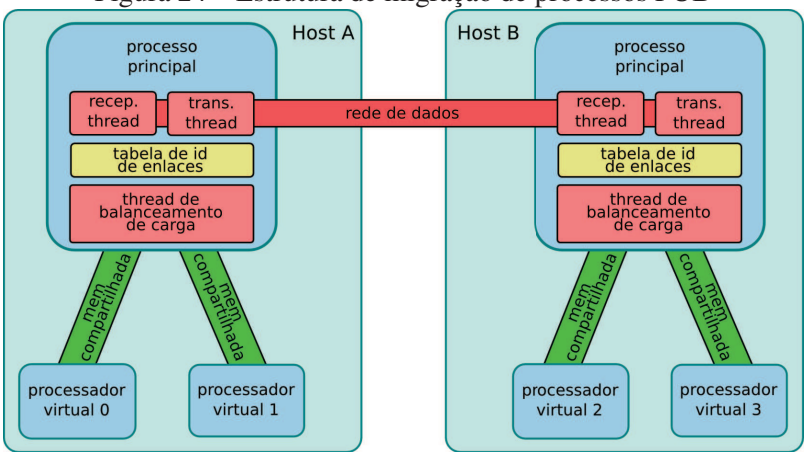
- Compatibilidade - HAMA aproveita todas as funcionalidades do Hadoop e seus pacotes dado que o HAMA mantém a compatibilidade com as interfaces Hadoop existentes;
- Escalabilidade - Pela característica de compatibilidade, o HAMA utiliza, sem nenhuma modificação, a infraestrutura da Internet e serviços como o *Amazon EC2*;
- Flexibilidade - Para aprimorar a flexibilidade necessária para suportar diferentes padrões de computação, HAMA oferece interfaces simples de motor de computação. Qualquer motor de cálculo, em conformidade com estas interfaces, pode ser conectado e utilizado. Atualmente, três motores de computação estão disponíveis para uso (MapReduce, BSP e Dryad);
- Aplicabilidade - Funções básicas oferecidas por HAMA podem ser aplicadas a vários *softwares* que requerem matriz e cálculos de grafos. De acordo com Seo *et al.* (2010), um exemplo prático é o *me2day* (ME2DAY, 2013), um serviço de rede social famosa na Coreia semelhante ao *twitter*.

3.1.2 PUB

Paderborn University BSP Library (PUB) é uma biblioteca em C considerada clássica de rotinas de comunicação para o desenvolvimento de algoritmos BSP (GAVA; FORTIN, 2009). Oferece funções para comunicação de troca de mensagens e acesso à memória remota, além de algumas funções de comunicação coletiva. Para tornar-se mais flexível, esta biblioteca tem a característica de criar objetos BSP independentes, permitindo a sincronização e migração para o balanceamento de carga.

Em Bonorden (2007) é apresentado um trabalho que faz uso desta biblioteca. O autor demostra o uso de processadores virtuais através da criação de processos. Neste, sua implementação estendia a biblioteca inicialmente desenvolvida (BONORDEN *et al.*, 2003) com a função de balanceamento de carga através da migração de processos.

Figura 24 – Estrutura de migração de processos PUB



Fonte: Traduzido livremente de Bonorden (2007).

Nesta abordagem, para cada processador virtual criado, um novo processo é lançado. A Figura 24 mostra uma visão geral dos processos e suas *threads* de comunicação e balanceamento de carga. A *thread de recepção* tem a função de receber as mensagens da rede e encaminhar para o processo correspondente colocando-as na fila de recepção de memória compartilhada. A *thread de transmissão* envia as mensagens enfileiradas para transmissão na memória compartilhada com o processador virtual para a rede. A *thread de balanceamento de carga* é responsável por coletar as informações de carga do sistema e decidir quando migrar um processador virtual. A tabela com a identificação dos processos auxilia no roteamento. Entretanto, a lista em cada nó (*host*) não sofre atualizações constantes a cada migração. Se um *host* realizou o processo de migração de um processador virtual que estava em seu domínio, ele saberá para onde rotear as mensagens que forem posteriormente recebidas, embora isto possa acarretar saltos extras.

O autor ainda apresenta algumas estratégias utilizadas para o balanceamento de carga: global centralizada, distribuída simples, distribuída conservativa e predição global. As decisões de migração são realizadas se considerando a carga atual dos processadores sem levar em conta o padrão de comunicação e o custo de migração. Nos resultados, o autor salienta que a migração de processos é uma técnica poderosa apresentando os benefícios da implementação.

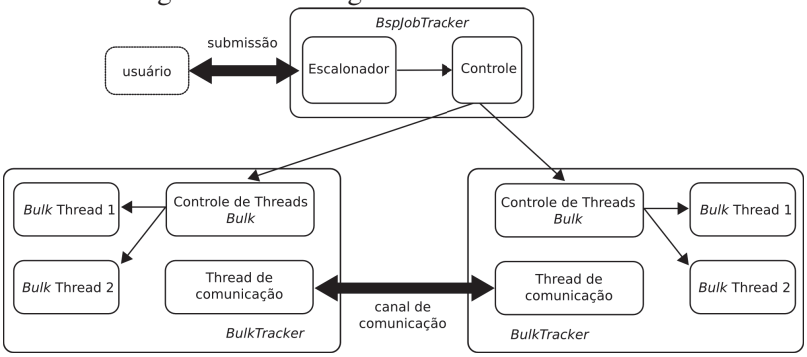
3.1.3 BSPCloud

BSPCloud é um modelo baseado no BSP com objetivo de utilizar os recursos em nuvem, tais como máquinas virtuais e migração. Com ele, o programador pode contar com um modelo simples, mas com custos realistas na concepção de um programa para computação em nuvem. Liu, Tong e Hou (2012) introduziram este modelo concebido em uma aplicação em Java. Os elementos que compõem o modelo BSPCloud são:

- *BspJob* - é a aplicação submetida;

- *Bulk* - é uma sub-tarefa de um BspJob;
- *BspJobTracker* - é utilizado para receber um BspJob e controla sua execução;
- *BulkTracker* - recebe os Bulks de um BspJobTracker.

Figura 25 – Visão geral do modelo BSPCloud.



Fonte: Traduzido livremente de Liu, Tong e Hou (2012).

Uma visão geral do modelo é mostrada na Figura 25. O exemplo é composto dois BulkTracker e um BspJobTracker. O BspJobTracker é composto por duas *threads*: a *thread Escalonador* e a *thread Controle*. Segundo os autores, ao submeter uma aplicação (BspJob), ela não é imediatamente executada. Somente após a seleção pela *thread* de escalonamento a aplicação é iniciada. A *thread* Controle particiona o BspJob em Bulks de acordo com os recursos selecionados pelo usuário. Os blocos BulkTrackers também possuem duas *threads*: *Controle de Threads Bulk* (CTB) e a *thread de Comunicação*(TC). O CTB é utilizado para receber o BspJob de um BspJobTracker e inicia a quantidade de *threads* Bulk indicadas pela *thread* de Controle. Quando um Bulk precisa comunicar com outro Bulk, ele utiliza a *thread* de Comunicação que provê o canal ponto-a-ponto entre eles. No modelo de aplicação apresentado pelos autores, o usuário precisa implementar a operação *bspOperate* e *bspDataMap*. Detalhes da implementação são encontrados em Liu, Tong e Hou (2012). Três avaliações foram realizadas sobre o modelo proposto. Na primeira análise, o trabalho verificou seu desempenho realizando a multiplicação de matrizes variando o número de máquinas virtuais com núcleo simples. A segunda avaliação considerava o uso de multi-núcleos. A terceira considerava a sua escalabilidade. Os resultados indicaram que o modelo aproveita de maneira eficiente a arquitetura multi-núcleo e possui um bom *speedup* e escalabilidade.

3.1.4 MulticoreBSP

Yzelman e Bisseling (2012) apresentam uma biblioteca orientada a objeto (Java) baseada na biblioteca BSPlib, que foi idealizada em linguagem C (HILL *et al.*, 1998). Esta biblioteca é nomeada como *MulticoreBSP*. Segundo aos autores, a implementação Java se propõe a ter um

conjunto menor de instruções e, desta forma, defendem a ideia de um aprendizado mais fácil e mantém a transparência em relação a máquina paralela. Os autores realizam uma comparação entre uma simulação utilizando o MulticoreBSP os resultados apresentados por Bisseling (2004) em seu trabalho com algoritmo BSPedupack (BISSELING, 2004).

Os autores apresentam para a criação de um programa genérico BSP com a biblioteca proposta, apenas a necessidade de estender a classe *BSP_PROGRAM* e a implementação de duas funções:

- $main\_part()$ - Executado por um único processo.
- $parallel\_part()$ - Parte do código que será executada em paralelo.

Para a troca de mensagens, a classe definida é a *BSP_COMM*, que implementa os métodos $bsp\_put()$, $bsp\_get()$ e $bsp\_send()$. Na MulticoreBSP, nenhum outro objeto realiza a comunicação entre as *threads*.

Nos resultados apresentados, a MulticoreBSP mostrou-se eficiente para uso em ambientes de memória compartilhada. Entretanto, os autores concluíram que a implementação em Java desta biblioteca não é viável para aplicações de alto desempenho, propondo trabalhos futuros com implementações em C++.

3.2 Balanceamento de Carga

O balanceamento de carga tem sido alvo de estudo e diversos trabalhos apresentam algumas bibliotecas com soluções próprias. A biblioteca *Mizan*, além de ser um sistema atual, possui um conjunto de etapas que definem o método de balanceamento que faz considerações interessantes como a distribuição normal para a definição de desbalanceamento. A *ProActive* foi a base para o desenvolvimento do *jMigBSP*. O *jMigBSP* é uma implementação do MigBSP em Java.

3.2.1 Mizan

O *Mizan* (KHAYYAT *et al.*, 2013) é um sistema, baseado em *Pregel* (MALEWICZ *et al.*, 2010), que suporta balanceamento de carga entre os super-passos.

Em uma breve descrição, o Pregel pode ser definido como um modelo de computação baseado no BSP, composto por *workers* em uma arquitetura mestre-escravo. Os *workers* são computadores ou máquinas virtuais (VM) que recebem as tarefas através de um *worker* mestre. Ele proporciona tolerância à falha, escalabilidade e eficiência. De acordo com Malewicz *et al.* (2010), o Pregel foi desenvolvido para a arquiterura de aglomerados do Google e está descrito em detalhes por Barroso, Dean e Hölzle (2003).

O Mizan monitora as características de todas as tarefas durante a execução (tempo, envio e recepção de mensagens). Baseado nestes parâmetros, ele realiza migrações das tarefas ao final

de um super-passo, de acordo com o plano montado para minimizar a variação entre os processos. Todos os componentes do Mizan possuem suporte para execução distribuída eliminando a necessidade de um controlador central. Baseado na linguagem C++, as funções BSPs de troca de mensagens são implementadas de acordo com a recomendação MPI.

Os autores apresentam a arquitetura de balanceamento de carga do Mizan baseado em 5 etapas:

- Identificar a fonte do desbalanceamento - A cada super-passo ele realiza uma comparação de todas as tarefas em relação a uma curva de distribuição normal, e seleciona aquelas que estiverem fora desta curva;
- Selecionar o objetivo da migração - Etapa que define a utilizaçãode uma política com o objetivo de equilibrar o custo de envio de mensagens, recebimento de mensagens ou tempo do super-passo;
- Parear subcarregados com sobrecarregados - Definir quais são os processos que possuem uma relação oposta em uma lista ordenada. Estes processos são organizados respeitando o critério do item anterior;
- Selecionar os processos para migração - O número de processos depende da diferença das estatísticas entre ele o e seu par equivalente. Um processo será selecionado se ele estiver fora da curva de distribuição normal;
- Migração - A migração ocorre durante a etapa denominada *barreira de migração*. O processo é codificado em um *stream* que inclui seu ID, estado e outras informações relevantes para continuação dos super-passos.

Khayyat *et al.* (2013) apresentaram uma comparação de desempenho do Mizan em relação ao *Giraph* (GIRAPH, 2013), uma implementação Java do arcabouço Pregel de código aberto. Nela, o Mizan mostrou ganhos acima 200% em relação ao Giraph na execução de algoritmos estáticos. Em algoritmos dinâmicos, o resultado teve um aprimoramento acima de 87%. O Mizan também apresentou melhorias em relação ao *overhead* inserido. Em aplicações estáticas, a redução foi de até 40%. Durante execução onde ocorrem migrações, houve uma redução de até 10%.

3.2.2 ProActive e jMigBSP

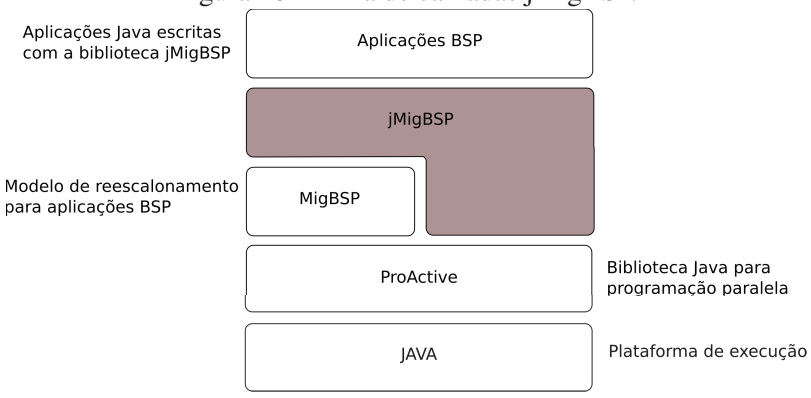
Baduel *et al.* (2006) utilizaram o *ProActive* (CAROMEL; KLAUSER; VAYSSIèRE, 1998), um *middleware* baseado em Java, direcionado para o desenvolvimento de aplicações paralelas e distribuídas. Os autores mostram que o ProActive é relevante para computação em grades devido aos protocolos de segurança e a monitoração. Também ressaltam o aspecto eficiente do tratamento de funções coletivas.

As aplicações desenvolvidas em ProActive utilizam o conceito de *objeto ativo* que possui uma *thread* de controle própria. Possuem a característica de comunicação assíncrona devido a aplicação do conceito de *objeto futuro*. Estes objetos são sincronizados com uma técnica de *espera por necessidade*. Ao invés de utilizar a troca de mensagens para comunicação, o ProActive faz uso da *invocação remota*. Além destas características, o ProActive permite o uso da técnica de migração para mover um objeto ativo de uma JVM para outra. A abstração de objetos ativos é apresentada como *nós virtuais* que compõem um DAG e são distribuídos por processadores reais. Esta técnica aliada a migração do objetos pode ser utilizada na implementação do balanceamento de carga para aplicações dinâmicas.

Amedro *et al.* (2010) realizaram um trabalho utilizando o *middleware* ProActive demonstrando a possibilidade de desenvolver aplicações paralelas para serem executadas tanto em ambiente de grade como em *cloud* e obter resultados equivalentes ou superiores. Os autores apresentam resultados e analises de aplicações paralelas executadas na *Amazon EC2* em comparação a aglomerados dedicados. Embora os resultados mostrem uma similaridade, o desempenho de execução em *cloud* se apresenta menos eficiente do que em ambientes dedicados quando uma comunicação intensiva é necessária. Um melhor resultado foi obtido ao utilizar uma estrutura heterogênea.

Em Graebin e da Rosa Righi (2011) foi introduzido o *jMigBSP*. Esta biblioteca oferece uma API para o desenvolvimento de aplicações BSP em Java. Basicamente, ela utiliza os recursos disponibilizados pelo ProActive para gerar interfaces adaptadas para aplicações BSP, como apresentado na Figura 26.

Figura 26 – Pilha de camadas jMigBSP.



Fonte: Graebin (2012).

De acordo com Graebin (2012), a escrita de aplicações paralelas com o ProActive traz algumas vantagens. Entre elas, estão a comunicação em grupo e comunicação assíncrona. Além disso, a ProActive possibilita o lançamento de aplicações em ambientes heterogêneos formados por agregados de computadores. Outro recurso interessante é a existência de técnicas incorporadas como VPN (*Virtual Private Network*) e SSH (*Secure Shell*) que auxiliam na comunicação

entre esses ambientes. As principais contribuições do Proactive são: programação orientada-a-objeto, migração de objetos, comunicação assíncrona e gerência dos recursos.

A principal contribuição do jMigBSP é fornecer o reescalonamento de tarefas através de *middleware* ou através de ativação explícita na aplicação. No trabalho Graebin e Righi (2012) foram realizados testes comparativos de desempenho da biblioteca jMigBSP em relação a biblioteca BSPlib e a análise do comportamento em aplicações desenvolvidas. As aplicações utilizadas para a os ensaios são as que seguem:

- Soma de prefixos;
- Compressão de imagens utilizando técnica de fractais (FIC);
- Transformada rápida de Fourier (FFT).

O teste com aplicação de soma de prefixos foi utilizado para validar a correta troca de mensagens entre os objetos e a sincronização das tarefas. Em particular, o algoritmo FFT foi o utilizado para a comparação com a BSPlib, e o resultado foi similar em ambas as bibliotecas, porém a BSPlib continuou apresentando melhor desempenho na maioria dos casos. Esta diferença foi atribuída a sobrecarga incluída através das camadas Java e ProActive. Entretanto, foi observado que, quanto maior a quantidade de tarefas paralelas, mais os resultados se equiparam. Na aplicação FIC, o jMigBSP obteve um ganho de desempenho acima de 90% em relação ao código equivalente tratado de forma sequencial.

3.3 Migração de Processos e Estratégias

Os trabalhos a seguir utilizam como base a biblioteca AMPI, apresentada anteriormente na Seção 2.8. Em Rodrigues *et al.* (2010a), os autores mostraram uma análise do impacto que as estratégias de balanceamento de carga tem sobre a computação e comunicação em uma aplicação de previsão meteorológica real. Também foram apresentados os efeitos da frequência de ativação das migrações sobre o tempo total de execução. Esta avaliação foi realizada sem nenhuma alteração na aplicação original, graças a característica AMPI (Charm++) de possuir o arcabouço de balanceamento de carga em uma camada independente.

Foram investigadas as seguintes estratégias:

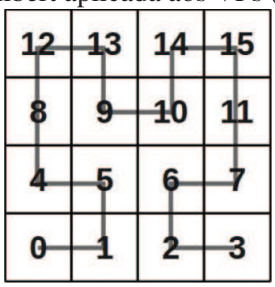
- *GreedyLB* - Baseia-se em colocar o processo mais lento no processador mais rápido. A tarefa se repete até que o equilíbrio seja alcançado. Esta estratégia não considera os custos de comunicação. Devido a sua simplicidade, ela é considerada mais rápida que as demais;
- *RefineCommLB* - Considera tanto a carga do processador quanto os custos de comunicação. Buscando atingir o valor de carga médio, a estratégia remove os objetos dos processadores mais carregados e os coloca em processadores atentando para que o custo de comunicação não se eleve. Possui um número limitado de migrações. Esta estratégia é utilizada em casos nos quais poucas migrações alcançam o equilíbrio;

- *RecBisectBfLB* - Esta estratégia realiza o balanceamento partindo o grafo de tarefas e enumerando os VPs de acordo com suas cargas até que o número de partições seja igual ao número de processadores reais. Embora sejam considerados os custos de comunicação, não há garantias de minimização deste parâmetro;

- *MetisLB* - Utiliza o método *METIS* (KARYPIS; KUMAR, 1995) para realizar a partição. Considera tanto a carga como a comunicação entre os VPs;

- *HilbertLB* - Balanceador desenvolvido pelos autores em Rodrigues *et al.* (2010b). Baseado nas curvas de preenchimento Hilbert, os Vps são ordenados de acordo com esta curva, abrangendo todo o domínio da previsão (ver Figura 27). De forma recursiva, divide-se a curva de acordo com as cargas observadas. Este processo é repetido até que o número de segmentos seja igual ao número de processadores físicos. Os segmentos resultantes terão aproximadamente a mesma carga.

Figura 27 – Curva de Hilbert aplicada aos VPs (*threads*) por HilbertLB.



Fonte: Rodrigues *et al.* (2010a).

Os autores concluíram que, apesar das estratégias MetisLB e HilbertLB apresentarem um custo de balanceamento de carga maior, é possível reduzir o tempo total de execução de forma significativa. A estratégia HilbertLB possui ainda uma vantagem em relação a MetisLB devido a sua possibilidade de uso no modo de balanceamento descentralizado, modo que MetisLB não foi desenvolvido.

Considerando um ambiente NUMA, Pilla *et al.* (2012a) apresentam a estratégia de balanceamento de carga *NucoLB* que tem o objetivo de realizar uma redistribuição hierárquica das tarefas de acordo com a topologia dos nós. Ela considera custos relacionados a quantidade de núcleos, assimetria de acesso à memória, tipo de interconexão e comunicação entre os VPs.

A estratégia NucoLB avalia se uma tarefa t deve migrar para um processador c de acordo com o cálculo do custo dado por:

$$cost(c,t) = coreLoad(c) + \alpha \times (r_{comm}(t,c) \times NUCO_F(comm(t), node(c)) - l_{comm}(t,c)) \tag{3.1}$$

A função *coreLoad(c)* representa a carga total do processador c (é a soma de todos os tempos de execução dos VPs em c). O fator α normaliza o custo comunicação em relação ao tempo de execução. A função $l_{comm}(t, c)$ retorna a quantidade de mensagens recebidas pela tarefa t de outros processadores de um mesmo nó NUMA como o processador c. Este valor representa o ganho de migração em relação a aproximação dos nós. $r_{comm}(t, c)$ refere-se a quantidade de mensagens recebidas de outros nós NUMA, de onde c não pertence. $NUCO_F(x, y)$ combina as características de não-uniformidade na topologia. Este item considera as latências de comunicação e o custo associado na comunicação entre o nó NUMA x e o nó NUMA y.

$$NUCO_F(x, y) = \frac{latência\ de\ x\ até\ y}{latência\ de\ x} \tag{3.2}$$

De acordo com Pilla *et al.* (2012a), se x e y estão em um mesmo nó NUMA, o *NUMA Factor* indica o custo de acesso de diferentes bancos de memória dentro do nó. Entretanto, o RTS *Charm++* não provê este tipo de informação necessária para a aplicação direta do NucoLB. Para solucionar esta lacuna, os autores utilizaram um conjunto de bibliotecas auxiliares para coletar estes dados dos nós: a *OpenMPI* (SQUYRES, 2012) e *LMbench* (LMBENCH BENCHMARK, 2013). Os experimentos foram baseados na comparação entre as estratégias *GreedyCommLB*, *ScotchLB*, *TreeMatchLB* e o *NucoLB*. GreedyCommLB faz a análise de acordo com a estratégia GreedyLB, mas utiliza as informações do padrão de comunicação na sua associação tarefa-processador. A ScotchLB utiliza a associação baseada na técnica de partição de grafo Schotch (PELLEGRINI; ROMAN, 1996). TreeMatchLB realiza basicamente a associação considerando o padrão de comunicação entre as tarefas. Nos resultados, NucoLB apresentou *speedup* de 1,19 sobre os demais balanceadores de carga em plataformas NUMA diferentes. Também obteve um índice de migração 11 vezes menor do que outros balanceadores.

Em Rodrigues *et al.* (2013), os autores apresentaram uma solução para a limitação que a AMPI apresenta em não permitir de forma trivial o uso de variáveis globais e estáticas. A abordagem utilizada foi a privatização das variáveis em *threads* de nível usuário. O método baseia-se em armazenamento local em *threads* (TLS), comumente utilizada em *threads* de nível *kernel*. Esta abordagem necessitou a alteração do arcabouço de balanceamento de carga do *Charm++* para adaptar as características empregadas. No protótipo, Rodrigues *et al.* (2013) utilizaram a função *isomalloc* para preservar os endereços durante a migração. Também foi realizada uma reimplementação da inicialização da TLS da *glibc*.

Os resultados obtidos mostraram uma melhora de desempenho em relação a solução atual. A troca de contexto das *threads* apresentou um desempenho significativo em relação a quantidade de variáveis globais. Em relação ao balanceamento de carga, o arcabouço modificado apresentou uma melhora de 10% em relação ao modo convencional.

3.4 Considerações Parciais

Os trabalhos apresentados neste capítulo demonstram os estudos recentes que estão relacionados com a implementação proposta. A partir destas informações, as características das bibliotecas utilizadas podem ser resumidas na apresentação da Tabela 4.

Tabela 4 – Bibliotecas e suas características

Biblioteca	Linguagem	Suporte a Migração	Estratégia Independente da Aplicação	Comunicação Bloqueante	Controle de invocação do BC
HAMA	Java	Não	Não	-	Provido por configuração em HDFS. Utiliza um valor de *threshold* passado na inicialização do serviço
PUB	C	Sim	Não	Não	Thread de BC e monitora continuamente os processadores e o lança caso detectar o desbalanceamento
BSPCloud	Java	Sim	Não	-	Não implementado nas versões atuais
MulticoreBSP	Java	Não	-	Sim	-
Mizan	C++/C	Sim	Não	Sim	Monitoração contínua e ativação a partir da comparação dos processos com a curva de distribuição normal
ProActive	Java	Sim	Não	Não	Depende da implementação por parte do programador
jMigBSP	Java	Sim	Não	Não	Através do parâmetro α (MigBSP)
AMPI	C++/C	Sim	Sim	Não	Através da função $MPI\_Migrate()$

Fonte: Elaborado pelo próprio autor.

As bibliotecas HAMA e BSPCloud não deixam explícitos se a comunicação (funções *send* e *recv*) são tarefas bloqueantes ou não. A biblioteca MulticoreBSP, apesar de ter seu trabalho atualmente baseado em Java, segundo a página do projeto (MULTICOREBSP FOR C, 2013), existe um artigo que será publicado baseado em linguagem C mas, atualmente, não há nenhuma publicação oficial com esta abordagem. Entretanto, continua sem nenhuma referência ao suporte a migração. A biblioteca Mizan, segundo o site oficial do projeto, terá seu nome alterado para Libra (MIZAN, 2013). Após a alteração, haverá abordagens diferenciadas para o tipo de grafo de tarefa a ser executado. Porém, não existe até o momento nenhum trabalho publicado com esta mudança e nem detalhes de implementação.

A AMPI tem as características semelhantes à jMigBSP, se observado apenas o modelo BSP. A principal diferença está no tratamento implícito na invocação do balanceamento de carga, ou seja, jMigBSP possui a opção automática na camada de *middleware*. Apesar do ProActive não implementar o balanceamento de carga, o jMigBSP permite a utilização do modelo MigBSP. Devido à limitação da migração de variáveis globais, na AMPI é necessário explicitar o momento da migração através da função *MPI_Migrate()*. Porém, apesar de Graebin (2012) mostrar que a linguagem Java tem uma representação crescente por parte dos desenvolvedores, Yzelman e Bisseling (2012) demonstram que sua utilização não tem apresentado um resultado

interessante para aplicações de alto desempenho.

A AMPI é a única que possui um desacoplamento entre a aplicação e a estratégia utilizada no balanceamento de carga, permitindo sua alteração através de linha de comando, sem modificação em código. Outra característica importante é a existência da instrumentação necessária para coleta de dados das aplicações e a disponibilidade para as estratégias de BC. Além disso o AMPI fornece uma infraestrutura de acesso a troca de mensagens e migração de VPs.

Em relação as estratégias disponíveis na biblioteca AMPI, a partir dos trabalhos pesquisados, podemos resumí-las através da Tabela 5.

Tabela 5 – Estratégias de balanceamento de carga Charm++/AMPI

Estratégia	Computação	Comunicação	Custo de Migração	Observações
GreedyLB	x			Mais rápido
GreedyCommLB	x	x		Considera o custo de comunicação
RefineCommLB	x	x		Limita a quantidade de migrações
RecBisectBfLB	x	x		Não garante minimização da comunicação
MetisLB	x	x		Algoritmo Metis
HilbertLB	x	x		Prioriza a comunicação de acordo com curva de Hilbert
ScotchLB	x	x		Distribui as tarefas de acordo com o algoritmo Schotch
TreeMatchLB		x		Direcionado a topologia com multicores
NucoLB	x	x	x	Direcionado a topologia com multicores

Fonte: Elaborado pelo próprio autor.

Dentre as estratégias apresentadas, a *NucoLB* é a única que considera o custo de migração de uma certa forma. O *NUMA Factor*, representado pela parcela $NUCO_F$ considera estes custos. Entretanto, para o seu uso são necessárias bibliotecas auxiliares para identificar as questões da topologia do sistema paralelo onde será empregado. Esta etapa representa uma necessidade de conhecimento prévio da arquitetura. Além disso, o sobrecusto calculado se refere a arquitetura *multi-core*, sendo pouco interessante em sistemas mais simples, ou quando não se pode prever a arquitetura dos nós. Uma das vantagens que o MigBSP++ possui em relação ao NucoLB é a ponderação do tamanho do processo a ser transferido e a falta de conhecimento do *hardware* disponível. Considerar a quantidade de memória a ser transferida na migração permite uma avaliação melhor do ganho obtido com o remapeamento do processo. Esta consideração é mais representativa quando se está trabalhando com grandes volumes de dados.

Além disso, o MigBSP++ proposto não necessitará da etapa de descoberta inicial e poderá ser aplicado em qualquer aplicação AMPI. Por não necessitar de nenhuma adaptação, o modelo permite uma escalabilidade melhor do que as estratégias anteriormente apresentadas, independente do conhecimento dos elementos do sistema paralelo.

4 MIGBSP++: MODELO DE REESCALONAMENTO DE PROCESSOS

Com o intuito de realizar um balanceamento de carga eficiente e transparente, nesta dissertação é introduzido o modelo de reescalonamento MigBSP++. Este modelo é baseado nas técnicas empregadas pelo MigBSP, introduzido por Righi (2009). Com ele é possível combinar múltiplas métricas através do potencial de migração (PM) auxiliando na definição de qual processo e para onde ele deve ser migrado. As principais diferenças estão relacionadas ao modo de detecção de desequilíbrio de carga no sistema e quantos processos serão migrados.

A motivação principal do MigBSP++ surge da observação de aplicações BSP em sistemas onde mais do que um processo é executado por processador. Esta situação causa um comportamento equivocado na detecção de desequilíbrio da carga no sistema pelo modelo MigBSP original, pois ele é idealizado em um contexto onde grades computacionais são utilizadas e a quantidade de tarefas em execução é uma por processador. Entretanto, a realização de ensaios em sistemas onde a quantidade de processadores é menor do que a de processos não é incomum. Com o intuito de suprir esta demanda, o MigBSP++ realiza uma análise que generaliza a aplicação do MigBSP, realizando uma detecção de forma mais adequada para este tipo de ambiente.

Outra questão que o MigBSP++ busca aprimorar é a decisão de quantos processos serão migrados. O MigBSP deixa a critério da implementação decidir entre duas heurísticas. Neste capítulo é introduzido o *algoritmo de predição BSP* (APSBP). Baseado nas características de aplicações BSP, ele define de forma clara quantos dos processos elencados, através de PM, irão migrar sem correr o risco de prejudicar os próximos super-passos.

4.1 Decisões de Projeto

Apesar do MigBSP++ ser uma técnica de reescalonamento que utiliza muitos dos recursos do MigBSP, ele não contempla a totalidade dos mecanismos apresentados por ele. As características que fazem parte do escopo do modelo proposto são:

- Desenvolvido para ser ativado através de *middleware*;
- Aplicado em aglomerados;
- Sistema de escalonamento centralizado;
- Utilizado em ambientes tanto homogêneo como heterogêneo;
- Independência do conhecimento da atividade da aplicação;
- Formação da matriz de PM através das métricas computação, comunicação e custo de migração;

A Figura 28 representa a arquitetura onde o modelo deve ser empregado. O desenvolvimento deve considerar que o usuário não terá atuação direta no comportamento do MigBSP++. Este reescalonamento deve ser ativado em nível de *middleware*, o qual realizará a coleta de informações do sistema paralelo e aplicação, repassando-as ao MigBSP++ para que sejam realizados as decisões de migração. Então, essas informações são retornadas para o *middleware* realizar a migração destes processos.

Para que o modelo tenha sucesso de implementação, a tabela 6 apresenta os recursos mínimos que o *middleware* deve possuir.

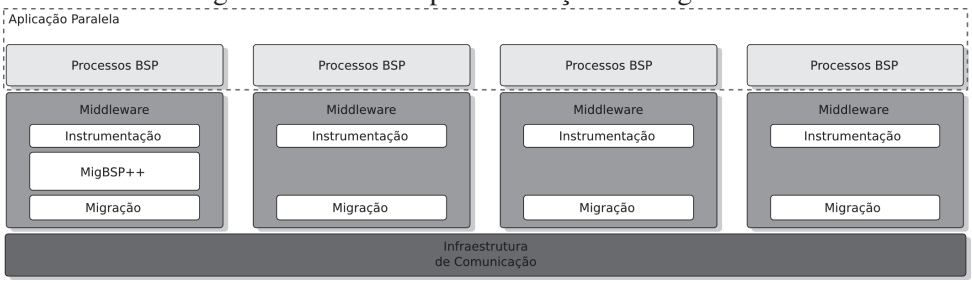
Tabela 6 – Recursos disponibilizados pelo *middleware* de implementação

Componentes	Descrição
Instrumentação	O sistema deve possuir uma instrumentação que realiza a coleta de dados sobre os estados dos processadores como a ocupação, velocidade de processamento e tempo de sincronização. Também deve fornecer ao MigBSP++ dados dos processo tais como número de instruções tempo individual de computação, tamanho do processo em *bytes* e a quantidade de dados trocados.
Acionamento de balanceamento de carga	O sistema deve oferecer algum método para o acionamento de balanceamento de carga. Ou realiza a detecção da fase de sincronia da aplicação BSP automaticamente, ou fornece interfaces ao programador para indicar o momento de ativação
Migração	Fornecer algum método para o acionamento de migração de lista de processos.

Fonte: Elaborado pelo próprio autor.

No MigBSP, a formação de $PM(i, j)$ é baseada em um ambiente de grade computacional onde nós gerentes de aglomerados, trocam as informações globais e controlam a distribuição de tarefas locais. Nesta abordagem, o parâmetro i refere-se a um processo e j a um aglomerado. Como o modelo MigBSP++ é dedicado a aglomerados, o parâmetro i continua a representar os processos, porém j indica um processador do sistema paralelo.

Figura 28 – Sistema para utilização do MigBSP++.



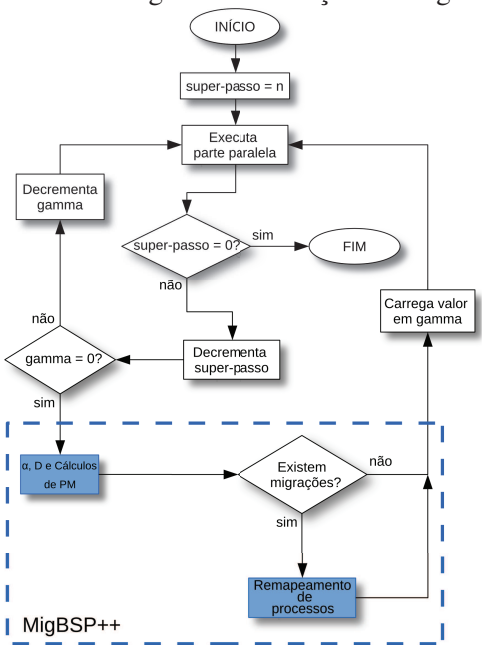
Fonte: Elaborado pelo próprio autor.

Cada nó do sistema é responsável por executar a coleta de dados sobre as informações locais e encaminhá-los ao nó central, onde o MigBSP++ fará a avaliação para o reescalonamento. Este cenário é representado na Figura 28. Em cada nó do sistema a instrumentação verifica as

informações sobre o estado do processador e dos processos em execução local. Estas informações são encaminhadas para um nó centralizador, onde o MigBSP++ será executado. Após a execução, todos os nós recebem as informações sobre a necessidade de efetuar migrações ou não.

Como as aplicações paralelas acontecem de forma independente, a avaliação de balanceamento de carga deve ocorrer em um estado controlado onde os processos estejam em estados conhecidos. Isso pode ser realizado de duas formas: (I) o *middleware* fornece uma interface de acesso ao desenvolvedor para informar quando é possível a intervenção do MigBSP++;(II) o *middleware* detecta automaticamente a etapa de sincronização. Se a opção (I) for utilizada, o fluxo de ativação pode ser realizado de acordo com a Figura 29.

Figura 29 – Fluxograma da ativação do MigBSP++.



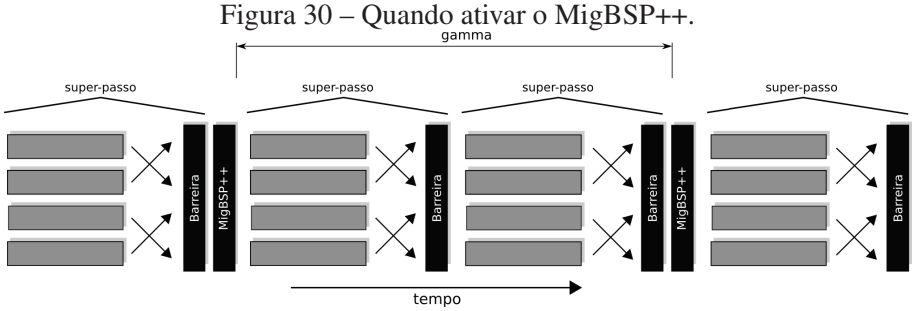
Fonte: Elaborado pelo próprio autor.

O parâmetro *gamma* representa um valor que evita que a invocação do MigBSP++ ocorra em cada barreira de sincronização. Isto é necessário para aplicações que não demandam intervenções a cada superpasso. Sendo assim, é possível definir a quantidade mínima de super-passos para que ela ocorra. Este método é semelhante ao utilizado por Rodrigues (2011). Além disso, o modelo MigBSP++ utiliza o parâmetro interno α, adaptativo, que define o quanto intrusivo o reescalonador de processos será. Após a barreira de sincronização, ocorre a verificação do valor de *gamma*. Caso seja igual a 0, o balanceador de carga é invocado e é realizada as averiguações necessárias relativas a α, D e cálculos dos PMs. Se existirem migrações a serem efetuadas, um novo mapeamento dos processos é realizado e a execução retomada. No caso (II), este parâmetro *gamma* poderá ser passado diretamente ao *middleware* através de linha de comando.

O MigBSP++ apresenta algumas adaptações com o intuito de aprimorar as respostas dada pelo MigBSP às questões *quando*, *quais* e *onde*. A questão *como* fica a critério do *middleware* adotado na implementação.

4.2 Analise do Momento da Ativação de Reescalonamento

Figura 30 – Quando ativar o MigBSP++.



Fonte: Elaborado pelo próprio autor.

Nas aplicações BSP, a decisão de quando o MigBSP++ será invocado fica a critério do parâmetro *gamma*. Como mostrado no fluxograma (Figura 29), ao alcançar a barreira de sincronização são avaliados os parâmetros pertinentes ao reescalonamento. Na Figura 30 é apresentado onde o valor de *gamma* é 2. A importância de se utilizar a barreira de sincronização e evitar a ativação do balanceamento de carga em um estado desconhecido. De acordo com Righi (2009), o fim da barreira de sincronização representa um estado consistente, inclusive utilizado para definição de *checkpoints* em técnicas de tolerância à falha (MIAO; TONG, 2007).

4.3 Adaptatividade

Apesar do acionamento do MigBSP++ ser realizado de acordo com *gamma*, a intrusividade do modelo pode ser controlada. O objetivo é reduzir a sobrecarga que o modelo possa introduzir na aplicação. O parâmetro D define qual é a faixa aceitável de desequilíbrio tornando o MigBSP++ mais ou menos rígido na detecção. Ele representa o percentual ao redor da média que as execuções dos processadores podem estar.

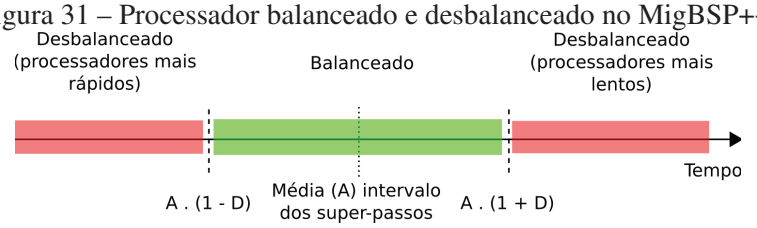
$$\textit{tempo do processador mais lento} < \textit{tempo médio dos processadores} \times (1 + D) \tag{4.1}$$

$$\textit{tempo do processador mais rápido} > \textit{tempo médio dos processadores} \times (1 - D) \tag{4.2}$$

Se alguma das inequações 4.1 ou 4.2 for falsa, é considerado pelo MigBSP++ que o sistema está desbalanceado.

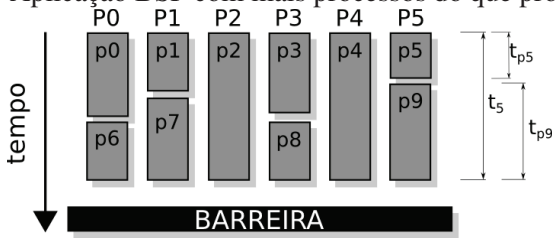
O método utilizado para detecção de desequilíbrio do sistema deve ser feito de modo a considerar o tempo total de execução de cada processador. Para isto é necessário observar

Figura 31 – Processador balanceado e desbalanceado no MigBSP++.



Fonte: Elaborado pelo próprio autor.

Figura 32 – Aplicação BSP com mais processos do que processadores.



Fonte: Elaborado pelo próprio autor.

que é possível existir mais de um processo sendo executado por processador. Nas aplicações BSPs, em um ambiente no qual a quantidade de processos lançados é maior que a quantidade de processadores, temos o cenário apresentado pela Figura 32. Nesta figura, o processo p_5 é executado em t_{p5} e o tempo de p_9 é representado por t_{p9}. O valor t_5 indica o tempo total que o processador P_5 precisa para alcançar a etapa de sincronização. Se considerarmos o tempo de troca de contexto muito baixo, teremos que $t_5 = t_{p5} + t_{p9}$. Formalizando, seja um super-passo S_n, com $n \in \{0, 1, ...\}$, em um sistema paralelo composto pelo conjunto de processadores $P = \{P_0, P_1, ..., P_{J-1}\}$, e P_j, $j \in \{0, 1, ..., J-1\}$, indicando um dos elementos do conjunto P. O conjunto de tempos de sincronização em um super-passo t_{S_n} em cada processador pode ser descrito como $t_{S_n} = \{t_0, t_1, ..., t_{J-1}\}$, e o tempo de sincronização de P_j, t_j, é dado por

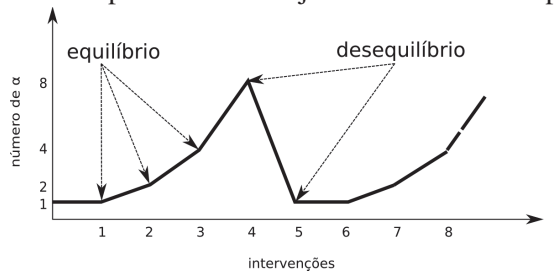
$$t_j = \sum_{n=0}^{z-1} T(n) + g \times \sum_{n=0}^{z-1} h_n + z \times l \tag{4.3}$$

O símbolo z representa a quantidade de processos p em execução em P_j. A função $T(n)$ retorna o tempo necessário de computação de p_n. Os valores de g e l são de acordo com o sistema BSP apresentado em Righi (2009). O parâmetro h é representação das transferências de dados realizadas pelos processos em P_j. Assim, tempo médio utilizado pelo MigBSP++ na detecção de desbalanceamento de carga é dado por

$$\mu_A = \frac{1}{J} \times \sum_{n=0}^{J-1} t_n. \tag{4.4}$$

Mantendo o controle de intrusividade adaptativo, herança do MigBSP, o controle de α sofreu uma alteração com o objetivo de aproveitar melhor o princípio da persistência (SAROOD; KALE, 2011). No MigBSP++, α realiza um comportamento semelhante ao controle de congestionamento utilizado pela implementação TCP (TANENBAUM, 2003). Diferentemente do modelo original, o valor dobra a cada intervenção em que o balanceamento é encontrado. Senão, o valor retorna rapidamente a 1. Este algoritmo (5) tem a vantagem de responder de forma mais eficiente em situações nas quais α já tenha alcançado valores elevados. Esta abordagem faz com que o controle seja mais intensivo em caso de desequilíbrio nos tempos das tarefas.

Figura 33 – Comportamento desejado de α durante a aplicação



Fonte: Elaborado pelo próprio autor.

Algoritmo 5: Atualização de α (MigBSP++)

Entrada: α_atual
Saída : $novo\_\alpha$
Se $desbalanceamento\ detectado$ **Então**
  $novo\_\alpha \leftarrow 1$
Senão
  $novo\_\alpha \leftarrow \alpha\_atual \times 2$
Fim Se
Retorna $novo\_\alpha$

Os parâmetros D e α devem ser passados para o MigBSP++, assim como acontece no MigBSP. A transição de um sistema para o outro é suave pois os estes parâmetros se mantém.

4.4 Analise dos Processos Candidatos e os Destinos das Migrações

Mantendo o padrão de seleção de processos, o MigBSP++ faz uso da função de potencial de migração (PM) para elencar quais processos são propícios a trazer redução de tempo na execução dos super-passos. Como a abordagem utilizada considera a utilização em aglomerados computacionais, os valores de PM já relacionam o processo i (resposta para a questão qual) e um processador j (resposta para a questão onde).

Realizando uma analogia com a ação vetorial sobre um corpo, o PM é a força resultante de um sistema que indica se um processo deve ser migrado ou não. A Figura 35 demonstra a

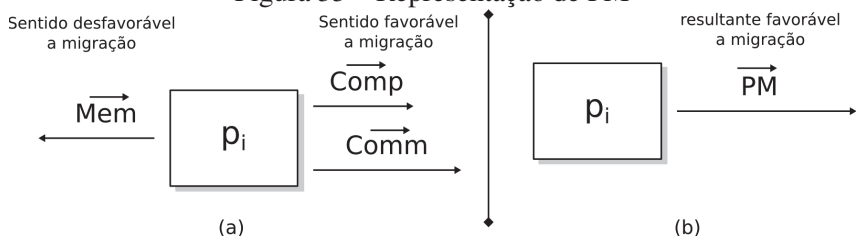
Figura 34 – Matrix M – PMs

	p_0	p_1	...	$p_{(I-1)}$
P_0	PM(0,2)	PM(1,2)	...	PM(I-1,0)
...	...	...	...	...
$P_{(J-1)}$	PM(0,J-1)	PM(1,J-1)	...	PM(I-1,J-1)

Fonte: Elaborado pelo próprio autor.

ação das forças componentes que resultam em PM. Em 35(a) estão os vetores decompostos que formam PM, e em 35(b), a força resultante da soma vetorial.

Figura 35 – Representação de PM



Fonte: Elaborado pelo próprio autor.

O cálculo de PM compõe a matriz M de dimensão $I \times J$, onde I é a quantidade de processos e J representa a quantidade de processadores disponíveis no sistema paralelo (Figura 34). Cada valor de M é uma grandeza que indica o quanto de tempo será reduzido na execução do processo no próximo super-passo (S_{n-1}) de p_i em P_j, caso a migração seja realizada. Valores iguais ou abaixo de 0 serão desconsiderados, pois não apresentam vantagens em migrar. Então,

$$PM'(i,j) = Comp(i,j) + Comm(i,j) - Mem(i,j) \tag{4.5}$$

$$PM(i,j) = \begin{cases} PM'(i,j), & \text{se } PM'(i,j) > 0 \\ 0, & \text{se } PM'(i,j) \leq 0 \end{cases} \tag{4.6}$$

Entretanto, a detecção de desbalanceamento ocorre com a comparação dos tempos dos processos. Os valores de PM são computados apenas para processos que estejam em processadores que tenham tempo de sincronização acima da média $\times(D+1)$. Os processos que estão em processadores com tempo de sincronização dentro da faixa indicada por D, possuirão o valor de $PM = 0$. Este método permite efetuar cálculos somente em processos que tenham real possibilidade de migrar. A parcela referente ao cálculo do custo de comunicação, $Comm(i,j)$, é definida pela equação 4.7

$$Comm(i,j) = BT(i,j) \times K(i,j), \tag{4.7}$$

onde $K(i,j)$ é o fator que relaciona o tempo e a unidade de *byte* transferido do processador

P_j para o processo p_i. A função $BT(i,j)$ indica quantos *bytes* foram enviados de P_j para o processo p_i. Se a expressão é descrita em função dos p_i, e representarmos P_{pi} como P_j onde p_i se encontra e $B(i,a)$ como a quantidade de *bytes* transferida do processo p_a para o p_i, temos

$$BT(i,j) = \sum_{n=0}^{I-1} \begin{cases} B(i,n), & \text{se } P_j = P_{pn} \\ 0, & \text{se } P_j \neq P_{pn}. \end{cases} \tag{4.8}$$

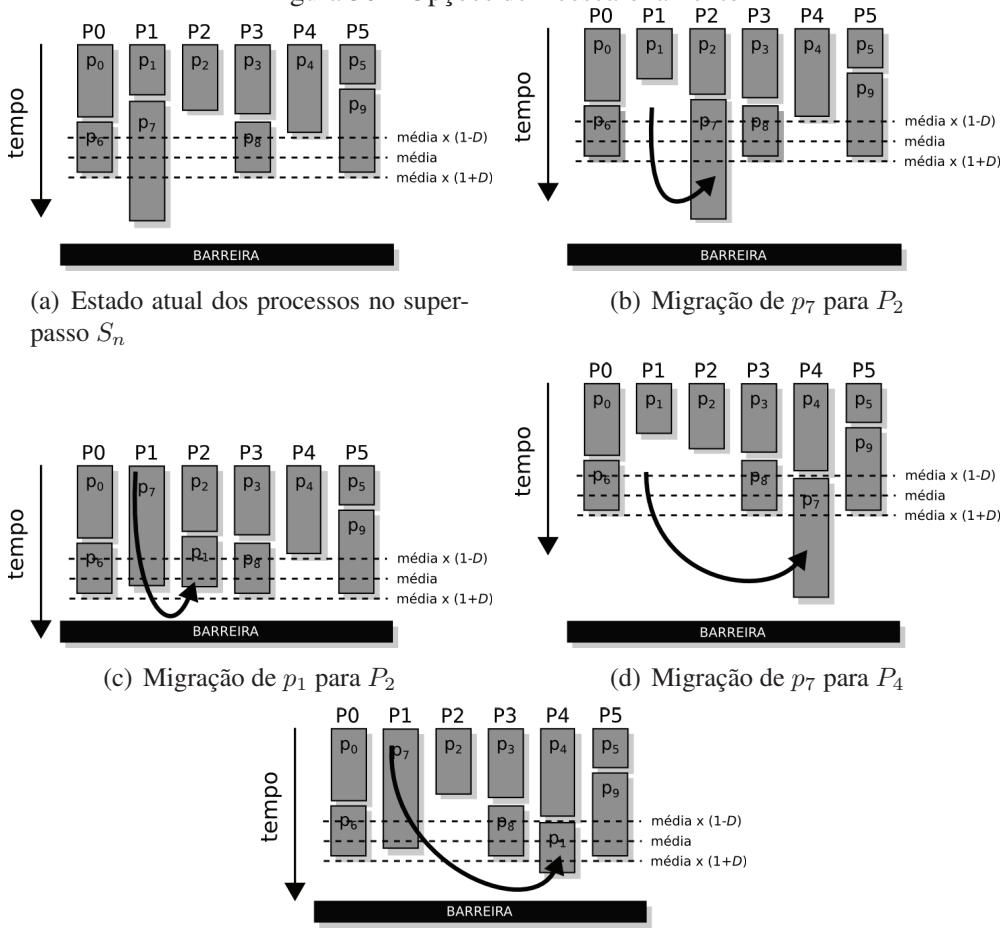
Outra parcela de PM é a função $Mem(i,j)$. Ela é responsável por mensurar o tempo necessário (custo de migração) para a transferência do p_i para P_j, e é descrita pela equação abaixo (4.9).

$$Mem(i,j) = SizeOf(i) \times K(i,j) \tag{4.9}$$

A função $SizeOf(i)$ retorna a quantidade de *bytes* que compõem todas as informações necessárias para se restaurar o processo p_i em P_j.

Para o cálculo referente à computação, considere a Figura 36.

Figura 36 – Opções de Reescalonamento



(a) Estado atual dos processos no super-passo S_n

(b) Migração de p_7 para P_2

(c) Migração de p_1 para P_2

(d) Migração de p_7 para P_4

(e) Migração de p_1 para P_4

A imagem apresentada em (a) representa um super-passo qualquer de um sistema paralelo onde são utilizados 6 processadores e executando 10 processos. Nesta ilustração todos os pro-

cessadores possuem a mesma capacidade de processamento. No super-passo S_n o MigBSP++ é lançado e se verifica que o sistema está em desequilíbrio devido ao processador P_1. Algumas observações são realizadas com o objetivo de otimizar o processo de cálculo de PM:

- Processos onde os processadores estão **abaixo** de $\mu A \times (1 + D)$, automaticamente terão $Comp(i, j) = 0$. Dado que eles já estão com tempos de execução abaixo da média, não existe a necessidade imediata de migração destes processos;

- Processos onde os processadores estão **abaixo** média μA serão os processadores que assumirão o valor de j. Os processadores que possuem alguma folga de tempo para alcançar a média são os únicos que poderão receber novos processos;

- Processos onde os processadores estão **acima** da média estipulada assumirão os valores de i. Dado que o processador já está sobrecarregado, qualquer um dos processos que está sendo executado nele, é candidato a migração.

Em 36(a) é demonstrado que o processador P_1 possui dois processos em execução, p_1 e p_7. De acordo com as diretivas acima, os valores de $Comp(i, j)$ para os processos p_0, p_2, p_3, p_4, p_5, p_6, p_8 e p_9 serão iguais a 0, por estarem de acordo com a primeira sentença. Assim fica estipulado que nenhum destes processos poderão migrar neste super-passo devido ao fator de computação. Avaliando a segunda sentença, os processadores com capacidade de receber mais algum processo são o P_2 e P_4. Logo, a função $Comp(i, j)$ será utilizada para calcular os ganhos em 4 casos possíveis. Em 36(b) há uma representação gráficafica que apresenta a migração do processo p_7 para o processador P_2. A figura resultante mostra que esta migração trará um aumento do tempo de computação de P_2 que será maior que o tempo atual de computação em P_1. Em 36(c), a migração que ocorre é de p_1 para P_2. Apesar desta migração apresentar um aumento do tempo de computação em P_2, é possível observar que este tempo ainda é menor que o atual tempo de computação em P_1. Em 36(d), a migração que ocorre é de p_7 para P_4. Esta migração obteve o mesmo resultado da migração apresentada em 36(a). E a migração apresentada em 36(e), demonstrou uma possibilidade também válida de migração, pois o tempo de computação de P_4 se manteve abaixo de tempo de P_1. Entretanto, a diferença final apresentada em 36(c) foi maior.

A parcela $Comp(i, j)$ determina qual será o ganho em redução de tempo em relação ao processador de destino P_j e o de origem P_{pi} caso p_i seja migrado para P_j. Indicamos $TP(P_n)$ como o tempo total de computação em P_n. Este modo de calcular $Comp(i, j)$ retornará um valor positivo se o novo somatório de tarefas em P_j for menor do que em P_{pi}, caso contrário, o valor será negativo. Essa função é dependente de $Inst(i)$, $T(i)$, $Speed(j)$ e $L(j)$ sendo:

- $T(i)$– o tempo de execução do processo p_i;

- $Inst(i)$– a quantidade de instruções executadas em p_i;

- $Speed(j)$– retorna a velocidade do processador P_j e;

- $L(j)$– indica a ocupação atual do processador P_j.

$$Inst(i) = T(i) \times Speed(P_{pi}) \times L(P_{pi}) \tag{4.10}$$

$$Comp(i,j) = \begin{cases} TP(P_{pi}) - (\frac{Inst(i)}{Speed(P_j)\times(1-L(P_j))} + TP(P_j)), & \text{se } TP(P_{pi}) \geq \mu_A \\ 0, & \text{se } TP(P_{pi}) < \mu_A \times (1-D) \\ 0, & \text{se } TP(P_j) > \mu_A \end{cases} \tag{4.11}$$

Com o cálculo de PM é possível preencher os valores da matriz M. Estes valores relacionam quais processos possuem condição de migrar sem trazer prejuízo ao próximo super-passo S_{n+1}. Caso a matriz M esteja com todos os valores iguais a 0, nenhuma migração ocorrerá. Mas se existir algum valor diferente, as migrações são indicadas pelo *Algoritmo de Predição BSP*.

4.5 Algoritmo de Predição BSP

O *Algoritmo de Predição BSP* (APBSP) decide quantos processos irão migrar. O tempo de sincronia do super-passo atual, S_n, é usado como referência para avaliação inicial. O APBSP ordena de forma decrescente os valores contidos na matriz M e escolhe o maior valor para realizar uma *simulação* de migração. A cada seleção de $PM(i,j)$ que produz um tempo de super-passo menor ou igual ao super-passo anterior, este é selecionado. Além disso, a indicação "p_i migra para P_j" é armazenada em uma lista de migrações válidas, a matriz M é atualizada removendo qualquer outra referência (PM) de migração deste processo p_i e o valor de referência é atualizado com o tempo retornado pela simulação. Em seguida, verifica-se o próximo $PM(i,j)$, que agora está no topo da lista ordenada. A simulação continua até que acabem os valores de PM ou a simulação produza um valor de super-passo S_{n+1} com tempo maior do que o valor mais atual. Baseado nas características do modelo BSP, o APBSP utiliza uma função que retorna o tempo total do super-passo S_{n+1} seguinte que é representada como:

$$fp(S_{n+1}) = max(tc_{S_n}) + max(cc_{S_n}) + cm_{S_n} \tag{4.12}$$

Na equação 4.12, $fp(S_{n+1})$ indica o tempo estimado do super-passo S_{n+1}. Esta operação é realizada realizando a soma dos valores máximos dos conjuntos tc_{S_n} e cc_{S_n}. O conjunto tc_{S_n} representa os tempos de computação (custo computacional) dos processadores no super-passo atual S_n. O conjunto cc_{S_n} representa o maior custo computacional entre as tarefas no super-passo S_n. E cm_{S_n} indica o sobrecusto acumulado devido aos processos já escolhidos para a migração. O algoritmo 6 descreve este procedimento.

É importante observar que neste algoritmo é necessário ter conhecimento das características dos processos e processadores que estão envolvidos no sistema. Esta informação fica a cargo

Algoritmo 6: Escolha dos processos para migração.

Entrada: $M : matriz[I \times J]$ *de* $PM, G : vetor[0..I-1]$ *de processos*, $P : vetor[0..J-1]$ *de* $Processadores$
Saída : mig_list : *lista de* $tarefas$ *a migrar*

```
1  mig_list ← null
2  novo_fp ← 0
3  fp_atual ← fp(G)
4  G_temp ← G
5  Repete
6      indice_pm ← max(M)
7      G_temp ← Sim(G_temp, indice_pm, P)
8      novo_fp ← fp(G_temp)
9      Se novo_fp ≤ fp_atual Então
10         adiciona(indice_pm, mig_list)
11         fp_atual ← novo_fp
12         remove(indice_pm, M)
13     Fim Se
14 Até fp_atual ≠ novo_fp;
15 Retorna mig_list
```

do *middleware* fornecer ao MigBSP++. A função $Sim()$ realiza a simulação da migração do processo p_i para processador P_j considerando as mudanças necessárias. Para isso, ela utiliza o $indice\_pm$, o conjunto de informações de processadores P e o grafo de tarefas temporário G_temp. Isto inclui um novo cálculo de tempo de computação, ocupação e comunicação, tanto do processador de destino como o de origem. Ao término do APBSP, a lista mig_list é então devolvida ao *middleware* para que as mudanças no sistema sejam realizadas sem a intervenção do programador.

4.6 Discussão Sobre o Modelo

Este capítulo demonstrou o modelo matemático utilizado pelo MigBSP++. Baseado no MigBSP, ele combina o custo de computação, comunicação e migração na decisão de migração. O MigBSP++ possui dois aprimoramentos importantes. O primeiro é a solução para o problema de detecção de desequilíbrio de carga no sistema paralelo. Nesta versão, são utilizados os somatórios de tempos de todos os processos em um processador para a detecção. Assim, não existe mais a necessidade dos processos possuírem o tempo de sincronização semelhantes para o sistema estar em equilíbrio e permite a utilização de mais de um processo por processador. O segundo aprimoramento é a adição do *Algoritmo de Predição BSP*. Antes de decidir migrar um ou mais processos, este algoritmo permite avaliar quais processos podem ser transferidos sem que o próximo super-passo tenha tempo maior do que o atual.

Através da observação da execução de aplicações BSP utilizando o MigBSP, se percebeu que, após o equilíbrio alcançado, caso ocorresse algum desequilíbrio depois de um determinado período, a ativação do balanceador de carga corria de forma muito espaçada. Isso gerava muitos passos para que o sistema retomasse o equilíbrio novamente. Por isso, houve a necessidade de redução brusca de α para o intervalo de 1 passo, permitindo uma resposta mais rápida do sistema.

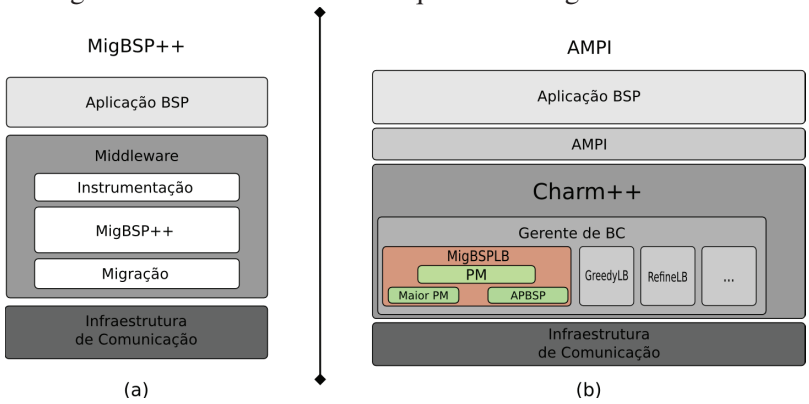
5 MIGBSPLB: IMPLEMENTANDO O MODELO MIGBSP++

Este capítulo apresenta duas implementações. Uma, baseada no MigBSP, utiliza a abordagem com a detecção de desbalanceamento de carga em função do tempo de computação das tarefas e realiza a migração de apenas uma tarefa de acordo com o maior PM encontrado. Esta utiliza uma das heurísticas proposta pelo modelo principal. A outra estratégia faz um tratamento considerando o modelo de reescalonamento proposto, MigBSP++, utilizando o *Algoritmo de Predição BSP* (APBSP) para determinar quantos processos irão migrar.

De acordo com a pesquisa realizada na elaboração desta dissertação, o Charm++/AMPI é o *middleware* que fornece os recursos que mais se aproximam das necessidades apontadas para a implementação do MigBSP++. Das necessidades apontadas pela tabela 6, o arcabouço de balanceamento de carga fornece praticamente todas as informações necessárias. Ele não efetua a detecção de sincronia em aplicações BSP para lançamento automático da estratégia de BC, mas disponibiliza a função *MPI_Migrate()* que permite ao desenvolvedor indicar quando realizar a intervenção de BC. Neste capítulo também estão apresentados os recursos do *Charm++* e suas limitações, os detalhes de implementação, metodologia utilizada e a avaliação geral dos resultados.

Mantendo a regra de nomes do Charm++ para as estratégias de balanceamento de carga, o nome da estratégia é denominada MigBSPLB e ela é apresentada em duas versões, “A” e “B”. A versão “A” diz respeito ao MigBSP e a “B” está relacionada ao MigBSP++. A biblioteca AMPI possui recursos que muito se assemelham a necessidade do MigBSP++. Na Figura 37 é realizado um paralelo entre elas.

Figura 37 – Paralelo entra as arquiteturas MigBSP++ e AMPI



Fonte: Elaborado pelo próprio autor.

A Figura 37(a) apresenta o sistema no qual o MigBSP++ está inserido e a 37(b) apresenta os recursos do AMPI. O Charm++ demonstra diversos recursos que não fazem parte do MigBSP++, como tolerância à falha, mas fornece quase todos os recursos necessários para implementação do sistema proposto. Ainda, na figura 37(b) localiza-se a estratégia em destaque

que será utilizada nos testes do modelo proposto.

5.1 Recursos e Limitações no Charm++

O manual do *Charm++* (UNIVERSITY OF ILLINOIS, 2013b) descreve como desenvolver um módulo próprio para se atribuir uma nova estratégia ao RTS *Charm++*. Para isso, é necessário criar uma estrutura básica para um objeto de classe *BaseLB*. Este objeto precisa ter implementado o método *work(LDStats*)*, de acordo com o código da Figura 38.

Figura 38 – Estrutura base para um módulo de estratégia de BC.

```
//Implementação de uma estratégia Foo
void FooLB::work(LDStats *stats)
{
/*==Inicialização==*/
  ProcArray *parr = new ProcArray(stats);
  ObjGraph *org = new ObjGraph(stats);
/*==início da implementação da estratégia==*/
//a estratégia é desenvolvida aqui
/*==fim estratégia==*/
  org->convertDecisions(stats);
}
```

Fonte: University of Illinois (2013b)

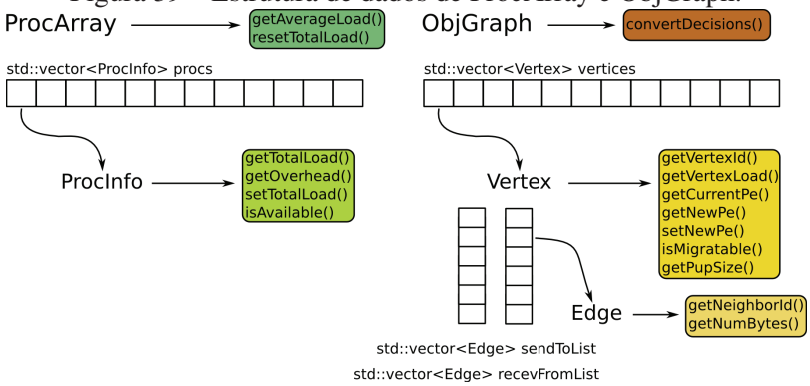
No caso de aplicações AMPI, método *work()* é invocado automaticamente pelo arcabouço de BC quando a função *MPI_Migrate()* é chamada. Baseado no conjunto de informações adquiridas durante a execução (por *LBDatabase*), *LDStats* é uma classe que contém os dados para a utilização na decisão do novo mapeamento dos VPs. As classes *ProcArray* e *ObjGraph* são vetores com informações preenchidas a partir de *LDStats*. Ainda na Figura 38, após a inicialização, *parr* possui os dados do vetor de processadores e *org* armazena os dados do grafo de tarefas da aplicação.

Os métodos existentes nos elementos de cada vetor (*ProcArray* e *ObjGraph*) são demonstrados na Figura 39. Os métodos da classe *ProcInfo* (membro de *ProcArray*) permitem obter as informações sobre as condições atuais dos processadores que compõem o sistema paralelo. A classe *Vertex* (elementos de *ObjGraph*) possui os métodos para coletar dados de carga da tarefa, a qual o processador está associado, definir uma nova associação, etc. Entretanto, as mudanças acontecerão somente quando ocorrer a chamada de *convertDecisions()*. Este método inicia as migrações necessárias movendo os objetos para os processadores correspondentes.

Mesmo com diversas informações disponíveis, algumas técnicas utilizadas no MigBSP são inviáveis de serem implementadas. Por exemplo, detectar o padrão de comunicação e computação (P_{comm} e P_{comp} respectivamente) necessárias para atender o modelo de modo fiel, na versão utilizada na escrita deste trabalho, são impraticáveis. Isso ocorre porque na classe *ObjGraph*, que manipula os VPs, não existem métodos que retornem as informações do *rank* relativo as tarefas MPI. Logo, não há como fazer a relação com os índices dos *Vertex* utilizados pelas es-

tratégias. Isso significa que não existem garantias de que, em uma chamada da estratégia o VP de índice 1, será na próxima chamada novamente o índice 1.

Figura 39 – Estrutura de dados de ProcArray e ObjGraph.



Fonte: University of Illinois (2013b).

Outra limitação imposta é a impossibilidade de coletar a informação da quantidade de instruções executadas pelo código do usuário. Porém, é possível fazer tal estimativa através da velocidade dos processadores e do tempo de execução em cada um deles. No MigBSP++ esta função já considerada como estimativa, sem prejuízo ao modelo.

Mais um fator que dificulta a realização da implementação completa tanto do MigBSP como do MigBSP++ é a falta de contagem de dados transmitidos no modo de comunicação coletiva. Isso pode acarretar em uma avaliação errada durante o cálculo de *Comm(i,j)*. Também não existe uma forma de retornar a taxa de transmissão dos dados entre os processos. Entretanto, algumas estratégias já contemplam uma forma de considerar esta métrica. Elas utilizam um valor constante definido nos cabeçalhos do *Charm++*. Este valor é utilizado pelas estratégias *GreedyCommLB* e *RefineCommLB*, por exemplo.

O custo de migração, fator que é um diferencial introduzido pelo modelo MigBSP e utilizado no MigBSP++, inicialmente era impossível de se obter no *Charm++*. Porém, em uma parceria com o Departamento de Ciência de Computação da Universidade de Ilinóis, este recurso foi adicionado ao arcabouço de BC para que seja utilizado pelas estratégias nas próximas versões [1]. Esta implementação, fundamental para a realização deste trabalho, é representada pelo método *getPupSize()*. Esse método retorna um valor aproximado da quantidade de *bytes* que serão transferidos quando o VP for migrado.

Considerando essas observações, ou as implementações estão de acordo com o modelo, ou sofreram pequenas contribuições para ajustar algum tipo de funcionalidade.

[1] O download desta versão pode ser realizado em "http://charm.cs.illinois.edu/cgi-bin/gitweb.cgi?p=charm.git;a=snapshot;h=81767bcd20dad39011cb479e3e5eb5e583bd2f94;sf=tgz". Acessado em ago/2013

Tabela 7 – Funções implementadas

Funções	Influência	Nível de impl.	Obs.
$PM(i,j)$	define o potêncial de migração	de acordo com o modelo	-
$Comp(i,j)$	custo de computação	adaptado	*MigBSPLB-A* não usa *Pcomm*
$P_{comp}(i,j)$	padrão de computação	não implementado	-
$Comm(i,j)$	custo de comunicação	parcial	utilizado valor constante para $K(i,j)$
$P_{comm}(i,j)$	padrão de comunicação	não implementado	-
$Mem(i,j)$	custo de migração	parcial	utilizado valor constante para $K(i,j)$
α	controle de intrusão	adaptado	MigBSP – dobra de valor 2 no equil. e divide por 2 no desequil. MigBSP++ – dobra de valor 2 no equil. e retorna a 1 no desequil.
D	nível de desbalanceamento	adaptado	valor permanece constante

Fonte: Elaborado pelo próprio autor.

5.2 Detalhes de Implementação

O modelo de reescalonamento MigBSP proporciona o balanceamento de carga em grandes estruturas computacionais, podendo ser compostos por aglomerados e grades. Neste cenário, é proposto um tratamento hierárquico. Contudo, no desenvolvimento da estratégia *MigBSPLB-A*, é realizado um tratamento de escalonamento centralizado igual ao proposto pelo MigBSP++ facilitando a implementação, monitoração e testes. O objetivo destes ensaios é mostrar as diferenças no tratamento durante a detecção de desbalanceamento e na escolha de tarefas a migrar.

Figura 40 – Implementação do módulo de estratégia *MigBSPLB*

```
//código principal
void MigBSPLB::work(LDStats *stats)
{
  ProcArray *parr = new ProcArray(stats);
  ObjGraph *ogr = new ObjGraph(stats);
  ObjGraph *tmp = new ObjGraph(stats);//grafo temporário para
                                      //uso na funçao de predição
  ProcFullInfo *pfi = new ProcFullInfo(stats);
  /** ================ STRATEGY ================== */
  if(MigBSPComputePM(pfi, parr, ogr, stats))
  {
        MigBSPMigrate(pfi,ogr,tmp);
        ogr->convertDecisions(stats);
  }
  delete parr;  delete pfi;
  delete ogr;  delete tmp;
}
```

Fonte: Elaborado pelo próprio autor

O conjunto de estratégias centralizadas que acompanham o pacote *Charm++* são derivados da classe *CentralLB*. Baseado nisso, o desenvolvimento ocorre de forma semelhante aos módulos já existentes. Dois métodos são criados. Um deles será responsável por preencher a matriz M calculando os valores dos *PM*s e informando a existência de algum valor maior que

0. Este é nomeado *MigBSPComputePM()*. O outro realiza um novo mapeamento dos VPs e é denominado *MigBSPMigrate()*. Diversos outros métodos são implementados, mas são menos importantes servindo como ferramentas auxiliares. A Figura 40 retrata o núcleo básico da implementação.

Um conjunto de testes foram realizados para avaliar os métodos propostos. Com o auxílio de programas específicos, desenvolvidos somente para obtermos uma avaliação dos dados coletados, pode-se verificar a funcionalidade dos parâmetros utilizados pela estratégia. A Tabela 8 apresenta estes testes realizados. Com ela, verificou-se a influência dos parâmetros em relação aos resultados esperados. Os testes foram baseados em controlar algumas variáveis e realizar a alteração de outras. Existem 5 parâmetros que foram avaliados: a influência de α, D, *Comp*, *Comm* e *Mem*.

Tabela 8 – Ensaios para a avaliação da implementação do *MigBSPLB*

Parâmetro	Ensaio proposto	Condição de teste	Resultado esperado
α	variar o valor dentro de um intervalo específico (a ser determinado)	Mantê-lo fixo	quanto menor o valor de α, mais vezes o balanceador de carga será invocado
D	definir valores específicos para observar a detecção de desbalanceamento	Mantê-lo fixo	para valores altos, a detecção do desbalanceamento ocorrerá com menos frequência
Mem	permitir só a sua variação	manter no valor máximo, mínimo e médio os valores de $Comm$ e $Comp$	verificar que o PM de migração terá uma relação inversa com a quantidade de memória
$Comp$	permitir só a sua variação	manter no valor máximo, mínimo e médio os valores de $Comm$ e Mem	verificar que o PM de migração terá uma relação direta a quantidade de dados trafegados
$Comm$	permitir só a sua variação	manter no valor máximo, mínimo e médio os valores de $Comp$ e Mem	verificar que o PM de migração terá uma relação direta com a estabilidade do processo

Fonte: Elaborado pelo próprio autor.

O parâmetro α tem influência na sobrecarga que o balanceador de carga terá sobre a aplicação. Mantendo este valor fixo, o balanceador de carga é chamado em um intervalo regular. Quanto maior este valor, maior será o intervalo entre uma invocação e outra. D tem a função de determinar a detecção do desbalanceamento na aplicação. Se este valor é muito pequeno, qualquer variação no tempo de execução dos processos é detectado e ativa o processo de balanceamento. Já para casos onde D é grande, será permitida uma variação maior sem a ativação. Os valores de *Comp*, *Comm* e *Mem* influenciam diretamente em *PM*. Já que as influências deles não podem ser analisadas isoladamente, um conjunto de testes para observar os valores de *PM* na saída do sistema. Para avaliar o comportamento de *Comp*, os valores de *Comm* e *Mem* foram mantidos constantes. Sendo PM uma função linear, cada variação de Comp pode ser observada na saída da função. Os mesmos testes foram empregados para a conferência das outras parcelas, *Comm* e *Mem*. Após a verificação, o comportamento dos métodos implementados corresponderam com os parâmetros esperados.

5.3 Avaliação e Metodologia de Testes

A avaliação da estratégias foram baseadas nos testes de trabalhos semelhantes realizados anteriormente. Trabalhos como Pilla *et al.* (2012b) e Rodrigues *et al.* (2010b) utilizaram softwares de referência e, devido a praticidade de troca de balanceador de carga por linha de comando, realizaram várias execuções alterando as estratégia e apresentaram uma análise crítica de cada resultado.

As estratégias escolhidas para a comparação são: *NullLB*, *GreedyCommLB*, *RefineCommLB*, *SchotchLB*, *RandCentLB* e *TopoCentLB*. A estratégia *NullLB* permite avaliar a aplicação sem que o balanceamento de carga seja realizado. A *RandCentLB* faz uma associação aleatória dos VPs. A *TopoCentLB* realiza uma análise considerando a topologia do sistema. As outras estratégias foram apresentadas na seção 3.3. Elas foram escolhidas por considerarem tanto computação, como comunicação no processo de decisão.

Dois algoritmos BSP foram implementados para a avaliação: (i) Shear-Sort (SCHERSON; SEN; MA, 1989) e (ii) FIC (JACKSON; MAHMOUD, 1996). O algoritmo (i), que trata a ordenação de matrizes quadradas, foi adaptado para ser executado em modo BSP. O algoritmo (ii) realiza a compressão de imagens em uma configuração de computação em *pipeline*. Os programas foram executados 15 vezes cada e os valores apresentados são a média destes tempos de execução. Como existem dois parâmetros de ajuste no modelo MigBSP e MigBSP++, foi realizado a análise em função da variação dos valores de α e D sobre as duas implementações do *MigBSPLB*. O sistema foi avaliado em um aglomerado heterogêneo formado por 7 computadores, descritos na Tabela 9, interconectados através de uma rede *Fast Ethernet*. O sistema operacional instalado em todos os computadores é o Linux - kernel 3.2.0 (distribuição Debian Wheezy [2]).

Tabela 9 – Descrição dos computadores que compõe o sistema paralelo.

ID	Modelo CPU	Cache(kB)	RAM(GB)
0	Celeron 430 1,8 GHz	512	2
1	Celeron 430 1,8 GHz	512	2
2	Celeron 520 1,6 GHz	1000	2
3	Dual Core E5400 2,7 GHz	2000	4
4	I7-3632QM 2,2 GHz	6000	6
5	I3-2330M 2,2 GHz	3000	4
6	Atom Z530 1,6 GHz	512	1

Fonte: Elaborado pelo próprio autor.

Baseado nos trabalhos do estado-da-arte, as avaliações das estratégias de BC, comumente, são realizadas através de comparações com outras existentes. Os ensaios realizados observaram as seguintes métricas:

- Sobrecarga introduzida nas aplicações pela estratégia (sem realizar migração);

[2] Disponível em http://www.debian.org/ – Acesso em março de 2013

- Determinação do tempo de execução da aplicações com a estratégia ativada (realizando migração);
- Comparação do tempo de execução com as estratégias previamente definidas;
- Análise de cada um dos itens em função de α e D.

Os testes foram definidos de forma a registar o tempo de execução sob diversas condições. A primeira foi a coleta do tempo de execução da aplicação executando normalmente sem a interferência de nenhuma estratégia. Em seguida, as estratégias definidas para comparação foram utilizadas e seus valores registrados. Os ensaios com o *MigBSPLB-A* e *MigBSPLB-B* ocorreram realizando a combinação dos valores de α e D. Os valores que α assumiu foram 2, 4 e 8. Este parâmetro tende a se adaptar e variar durante a execução. Para D os valores determinados são 0,3, 0,2 e 0,1. Este parâmetro faz com que o balanceador seja mais ou menos tolerante ao desequilíbrio das tarefas. Todas as execuções foram realizadas considerando apenas um processador por máquina e que as velocidades dos processadores são utilizados pelas estratégias. Isto significa que a linha de comando possui os parâmetros *+LBTestPESpeed* e *+p7*.

5.4 Aplicações Desenvolvidas

As aplicações desenvolvidas foram escritas utilizando a biblioteca AMPI de forma a permitir a variação dos parâmetros de entrada e alterar a granularidade das tarefas. A seguir as principais características dos algoritmos.

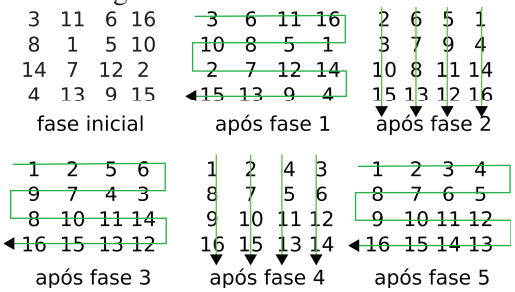
5.5 Ordenação de Matrizes: Algoritmo Shear-Sort

O algoritmo Shear-Sort (SCHERSON; SEN, 1989; SCHERSON; SEN; MA, 1989) possui uma característica de ser uma aplicação de fácil implementação e pode ser decomposta para operação em sistemas paralelos. Nas interações ímpares, é realizada uma ordenação das colunas de cima para baixo. Já nas interações pares, ele utiliza uma ordenação direta e reversa para cada linha. Esta fase de ordenação é conhecido como *snake-like*, Figura 41. A quantidade de fases necessárias para ocorrer a ordenação é dada por $2 \log n + 1$. O algoritmo 7 o descreve de forma geral.

Algoritmo 7: Ordenação Shear-Sort.

```
1 Para i Até n_passos Faça
2     Se i é par Então
3         Ordena_Coluna(M)
4     Senão
5         Ordena_Snake_Like(M)
6     Fim Se
7 Fim Para
```

Figura 41 – Fases Shear-Sort



Fonte: http://pages.cs.wisc.edu – Acesso em out/2013

Em um sistema paralelo, a ordenação pode ser realizada atribuindo a cada tarefa n linhas. Ao final de uma etapa, cada processador comunica aos outros de forma a atualizarem a situação dos elementos. A aplicação desenvolvida trata de matrizes quadradas e o número de VPs deve ser compatível com o lado da matriz. Se a matriz de entrada for $M(N \times N)$ e n a quantidade de linhas que cada um irá tratar, o número de VPs deve ser N/n. O algoritmo de ordenação local, utilizado nas linhas, é o *Quick-Sort*. Para que a execução tenha um tempo consideravelmente elevado e a observação do potencial do balanceamento de carga seja possível – sem exceder o tamanho máximo para a utilização de *isomalloc* em todos os nós – a quantidade de passos foi estendida ao valor da aresta. Assim também é possível tirar proveito da complexidade apresentada pelo algoritmo *Quick-Sort* no pior caso ($O(n^2)$).

O algoritmo 8 representa a implementação de acordo com o modelo BSP. Cada processo manipula a quantidade de n_linhas de dados. Na primeira etapa, ocorre a inicialização dos dados pertinentes a cada processo. A função $Init()$ resume a inicialização de variáveis auxiliares e a geração dos dados locais. Logo após, o bloco característico do Shear-Sort é executado. Porém, neste novo algoritmo há uma comunicação seguida de uma sincronização ao final de cada laço **Para**. A função $Ordena\_Coluna()$ executa o algoritmo *Quick-Sort* em sentido direto, e a função $Ordena\_Snake\_Like()$ faz tanto o *Quick-Sort* de forma direta, quanto de forma reversa, dependendo do número da linha a ser ordenada. A função $Finaliza()$ faz o envio dos dados para o processador principal, formando assim a matriz ordenada.

Algoritmo 8: Ordenação Shear-Sort BSP.

```
1  Init()
2  Para i Até n_passos Faça
3  |   Se i é par Então
4  |   |   Ordena_Coluna(n)
5  |   Senão
6  |   |   Ordena_Snake_Like(n)
7  |   Fim Se
8  |   Comunica(n)
9  |   Barreira()
10 Fim Para
11 Finaliza()
```

Os ensaios realizados com este algoritmo foram a ordenação das matrizes apresentadas pela Tabela 10. Estes arranjos foram organizados observando-se a quantidade de dados que o algoritmo implementado pode aceitar. Era necessário manter uma relação de proporcionalidade em relação ao tamanho da matriz e a quantidade de linhas a serem processadas em cada tarefa.

Tabela 10 – Ensaios com algoritmo Shear-Sort

Cenário	Tamanho	N° de linhas/VP	N° de VP
i	49x49	7	7
ii	98x98	14	7
iii	196x196	28	7
iv	98x98	7	14
v	196x196	14	14
vi	147x147	7	21
vii	294x294	14	21
viii	196x196	7	28

Fonte: Elaborado pelo próprio autor.

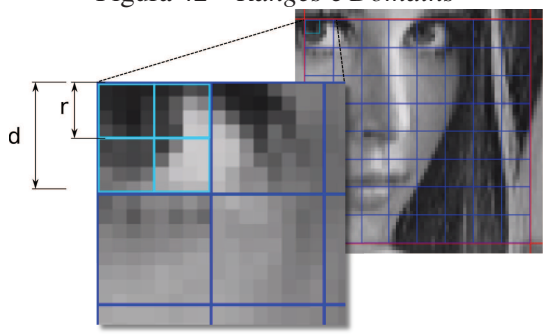
5.6 Compressão de Imagem

Este segundo algoritmo é baseado na solução proposta por Jackson e Mahmoud (1996). A compressão de imagens é realizada através da técnica de fractais. O método baseia-se na seleção de um bloco de imagem e na busca por outro que tenha uma correlação elevada. O *sistema de funções iterativas* e o *teorema do ponto-fixo* são as principais ferramentas utilizadas por este método (BARRIVIERA, 2009). Inicialmente, a imagem é dividida em blocos denominados *ranges* e *domains*. Os *ranges* são blocos não sobrepostos de tamanho $r \times r$ que formam o conjunto $R = \{r_0, r_1, ..., r_{\frac{n^2}{r^2}-1}\}$. Os *domains* são blocos, que podem ser sobrepostos ou não, de tamanho $d \times d$ que formam o conjunto $D = \{d_0, d_1, ..., d_j\}$, onde j depende da quantidade total de *domains* menos 1. O valor de d deve ser sempre maior do que r. Geralmente, a relação entre d e r é 2. Se o tamanho total da imagem for $n \times n$ e o tamanho de um range for $r \times r$, a quantidade de *ranges* a localizar é $(\frac{n^2}{r^2})$. A Figura 42 apresenta os *ranges* e os *domains* em uma imagem.

Cada *range* r_n é comparado com os *domains*. Os d_m são escalados na relação de n/N e sofrem transformações isométricas. São utilizadas 8 transformações: normal, espelhamento vertical e horizontal, rotação 90°, 180° e 270°, espelhamento HV e espelhamento HV invertido. A comparação é realizada através da função de Erro Quadrático Médio (MSE) (equação 5.1), quanto mais próximo de 0, mais parecido um *range* é de um *domain*. Este algoritmo é conhecido por ter complexidade de $O(n^4)$.

$$MSE(r_i, r'_i) = \frac{1}{r^2} \sum_{k=0,l=0}^{r-1} (r_i(k,l) - r'_i(k,l))^2 \tag{5.1}$$

Figura 42 – *Ranges* e *Domains*



Fonte: Elaborado pelo próprio autor.

Nesta equação, r_i indica o *range* original e r'_i representa o *domain* já isometricamente transformado. Após a escolha do melhor *domain*, d_i , são determinados os elementos da *Transformação Afim* que irá recompor os valores do *range*. A função que restaura os valores de um *range* r_i é dada pela equação 5.2

$$f(r_i) = s_i T_i(S(d_i)) + o_i, \tag{5.2}$$

onde:

- s_i é a relação de contraste entre r_i e d_i;
- $T_i()$ indica uma das oito transformações isométricas;
- $S(d_i)$ representa a mudança de escala do *domain*;
- o_i indica a relação de brilho entre r_i e d_i.

Os valores das constantes s_i e o_i são calculados com as equações 5.3 e 5.4.

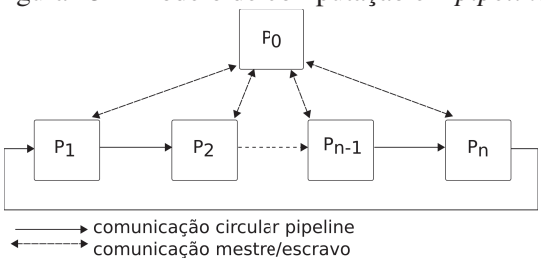
$$s_i = \frac{\sum S(d_i) \sum r_i - N^2 \sum S(d_i) r_i}{(\sum S(d_i))^2 - N^2 \sum S(d_i)^2} \tag{5.3}$$

$$o_i = \frac{\sum S(d_i) \sum r_i - \sum S(d_i)}{(\sum S(d_i))^2 - N^2 \sum S(d_i)^2} \tag{5.4}$$

A computação BSP ocorre com o modelo de comunicação circular *Pipeline* proposto por Jackson e Mahmoud (1996). Como mostrado na Figura 43, na primeira etapa, o processador P_0 tem o papel de mestre que faz a divisão dos *ranges* e *domains*. Em seguida, eles são distribuídos pelos processadores escravos do sistema paralelo. Então, cada processador realiza uma busca dos *ranges* locais pelos *domains* existentes. Ao término de cada busca, os *ranges* são enviados para o próximo processador, fazendo uma comunicação circular. Após a sincronização, os processadores iniciam novamente a busca até que todos os *ranges* retornem ao processador

de origem. Quando isto acontece, as informações são retornadas ao processador mestre e o algoritmo encerra.

Figura 43 – Modelo de computação em *pipeline*



Fonte: (JACKSON; MAHMOUD, 1996)

O Algoritmo 9 apresenta, de forma genérica, como é realizada a compressão de imagens. O vetor *ranges* é preenchido com os blocos da imagem a ser compactada e o vetor *domains* é populado com as partes da imagem a serem comparadas. Essa operação é realizada pela função *Init()*. Ao entrar no laço principal, cada elemento do vetor *ranges* é comparado com todos os elementos do vetor *domains* isotransformados (*m*), através da função *MSE*. Se o valor de *err_atual* for menor que o *err* do *range[i]*, eles são substituídos, e a informação do *domain*(*j*) e a isotransformação (z) são armazenadas.

Algoritmo 9: Compressão FIC

```
1  Init()
2  Para i Até n_ranges Faça
3      ranges[i].err ← ∞
4      Para j Até n_domains Faça
5          Para z Até n_trans Faça
6              m ← IsoTransform(domains[j], z)
7              err_atual ← MSE(m, i)
8              Se err_atual < ranges[i].err Então
9                  ranges[i].err ← err_atual
10                 ranges[i].d ← j
11                 ranges[i].iso ← z
12             Fim Se
13         Fim Para
14     Fim Para
15 Fim Para
16 Finaliza()
```

Já o programa implementado recebeu algumas modificações para ser tratado como BSP. O Algoritmo 10 apresenta estas alterações. A inicialização dos valores de *err* deve ocorrer fora do laço principal. Ao final de cada iteração de i, o *range* deve ser enviado para o próximo processador e receber um novo até que todos os *ranges* tenham sido comparados com todos os *domains* distribuídos pelo sistema paralelo.

Existem duas formas de manipular a granularidade da aplicação. A primeira é alterando o tamanho do *range*. A outra é variando a quantidade de processadores. Uma imagem de tamanho 1920x1080 foi escolhida para os ensaios e as configurações adotadas estão descritas na tabela 11.

Algoritmo 10: Compressão FIC BSP

```
1  Init()
2  Para i Até n_ranges/n Faça
3  |   ranges[i].err ← ∞
4  Fim Para
5  Para i Até n_ranges/n Faça
6  |   Para j Até n_domains/n Faça
7  |   |   Para z Até n_trans Faça
8  |   |   |   m ← IsoTransform(d[j], z)
9  |   |   |   err_atual ← MSE(m, i)
10 |   |   |   Se err_atual ≤ ranges[i].err Então
11 |   |   |   |   ranges[i].err ← err_atual
12 |   |   |   |   ranges[i].d ← j
13 |   |   |   |   ranges[i].iso ← z
14 |   |   |   Fim Se
15 |   |   Fim Para
16 |   Fim Para
17 |   Comunica(ranges[i])
18 |   Barreira()
19 Fim Para
20 Finaliza()
```

Tabela 11 – Ensaios com algoritmo de compressão de imagens

Cenário	Tamanho do *range*	N° de *domains*	N° de VP
i	32	1888	7
ii	16	7854	7
iii	8	32026	7
iv	32	1888	14
v	16	7854	14
vi	8	32026	14
vii	32	1888	21
viii	16	7854	21
ix	8	32026	21
x	32	1888	28
xi	16	7854	28
xii	8	32026	28

Fonte: Elaborado pelo próprio autor.

6 RESULTADOS E ANÁLISES

Neste capítulo estão os resultados e a análise dos ensaios realizados com a execução das estratégias desenvolvidas. Com os algoritmos implementados, ordenação de matrizes e compressão de imagens utilizando a AMPI, foram executados os cenários apresentados pelas Tabelas 10 e 11. Inicialmente, pode-se observar uma tabela com os resultados gerais, formada com os tempos médios das execuções. Em seguida, são expostos os resultados de forma mais específica, observando os tempos dos super-passos em cada intervenção considerando os cenários mais extremos: com menor e maior número de VPs. Por fim, apresenta-se um resumo das análises de cada algoritmo comentando sobre os pontos positivos e negativos observados.

6.1 Avaliação das Estratégias – Aplicação Shear-Sort

A Tabela 12 apresenta os resultados obtidos através da execução do programa baseado no algoritmo Shear-Sort. Nessa tabela, o tempo total de execução da estratégia principal em observação, *MigBSPLB-B* (representante do modelo MigBSP++), é apresentado como SC. Este valor representa auxilia na avaliação da sobrecarga na aplicação.

Tabela 12 – Média dos Tempos de Execução [s] – Shear-Sort

Estratégia		i	ii	iii	iv	v	vi	vii	viii
NullLB		12	25	181	25	124	38	505	98
GreedyCommLB		13	28	254	35	182	69	729	162
RefineCommLB		12	26	191	35	80	39	381	72
ScotchLB		15	34	285	42	197	85	816	176
RandCentLB		16	37	384	41	228	96	912	212
TopoCentLB		17	36	196	26	140	49	598	127
SC(D=0,3)		12	26	192	26	137	42	541	107
SC(D=0,2)		12	26	191	26	138	42	544	107
SC(D=0,1)		12	26	191	26	138	42	546	107
MigBSPLB-A(D=0,3)		12	26	166	26	77	40	375	65
MigBSPLB-A(D=0,2)	α=2	12	26	164	26	79	40	570	67
MigBSPLB-A(D=0,1)		12	26	167	26	82	40	344	76
MigBSPLB-B(D=0,3)		13	26	123	26	75	39	301	62
MigBSPLB-B(D=0,2)		14	26	124	26	99	41	316	61
MigBSPLB-B(D=0,1)		13	27	132	27	78	45	303	60
SC(D=0,3)		12	26	191	26	136	42	544	118
SC(D=0,2)		12	26	191	26	137	42	538	118
SC(D=0,1)		12	26	190	26	137	42	547	118
MigBSPLB-A(D=0,3)		12	26	143	26	80	40	400	66
MigBSPLB-A(D=0,2)	α=4	12	26	166	26	84	40	315	76
MigBSPLB-A(D=0,1)		12	26	151	26	87	40	344	69
MigBSPLB-B(D=0,3)		12	26	126	26	77	39	301	66
MigBSPLB-B(D=0,2)		12	26	153	26	98	45	309	68
MigBSPLB-B(D=0,1)		14	26	125	26	88	44	306	64
SC(D=0,3)		12	26	192	26	137	42	548	129
SC(D=0,2)		12	26	192	26	138	42	543	129
SC(D=0,1)		12	26	191	26	138	42	549	130
MigBSPLB-A(D=0,3)		12	26	139	26	82	40	363	69
MigBSPLB-A(D=0,2)	α=8	12	26	147	26	88	40	352	69
MigBSPLB-A(D=0,1)		12	26	150	26	84	40	347	70
MigBSPLB-B(D=0,3)		12	26	143	26	87	39	311	69
MigBSPLB-B(D=0,2)		12	26	126	26	86	43	321	65
MigBSPLB-B(D=0,1)		13	26	151	26	86	44	312	66
Menor tempo		12	25	123	25	75	38	301	60

Observando os dados, podemos avaliar a sobrecarga do *MigBSPLB-B* sobre a aplicação realizando a seguinte operação

$$Sbcg[\%] = \frac{T(SC) - T(NullLB)}{T(NullLB)} \times 100, \tag{6.1}$$

onde *Sbcg* representa a sobrecarga adicionada, *T(SC)* indica o tempo de execução com a estratégia sem migração, e *T(NullLB)* representa o tempo de estratégia nula. Os gráficos das Figuras 44 e 45 mostram a sobrecarga introduzida na aplicação por *MigBSPLB-B*. O cenário i apresentou uma sobrecarga de 0%. Aparentemente a escala utilizada não foi suficiente para para mostrar o valor real da sobrecarga e o tempo foi muito curto para a análise. Nas execuções dos cenários ii e iv a sobrecarga foi menor que 5%. Nos casos iii, vii e viii os valores são entre 5% e 10%. Nos cenários v e vi a sobrecarga ultrapassou 10%.

Figura 44 – Sobrecarga x Cenário (I)– Shearsort
Sobrecarga da estratégia *MigBSPLB-B*

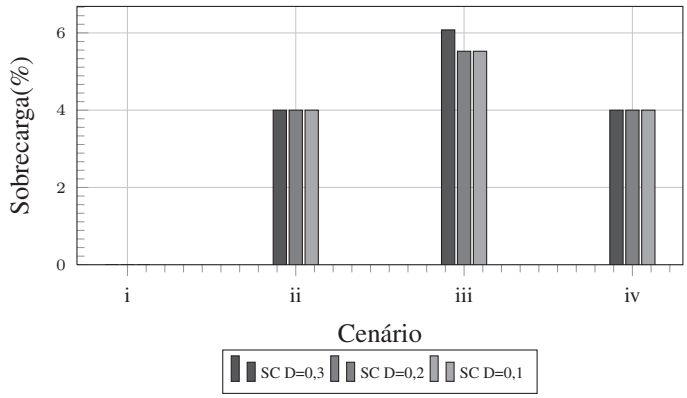
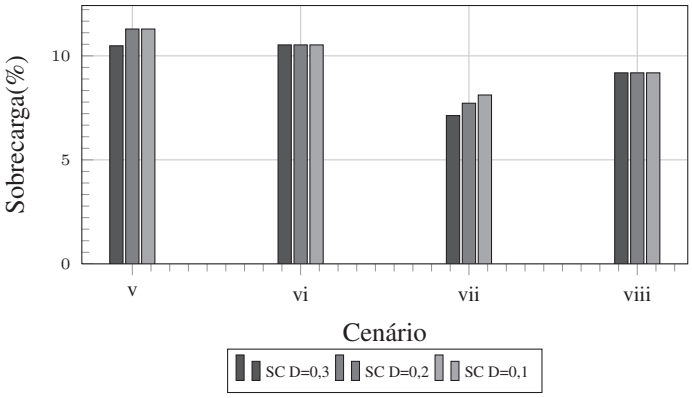


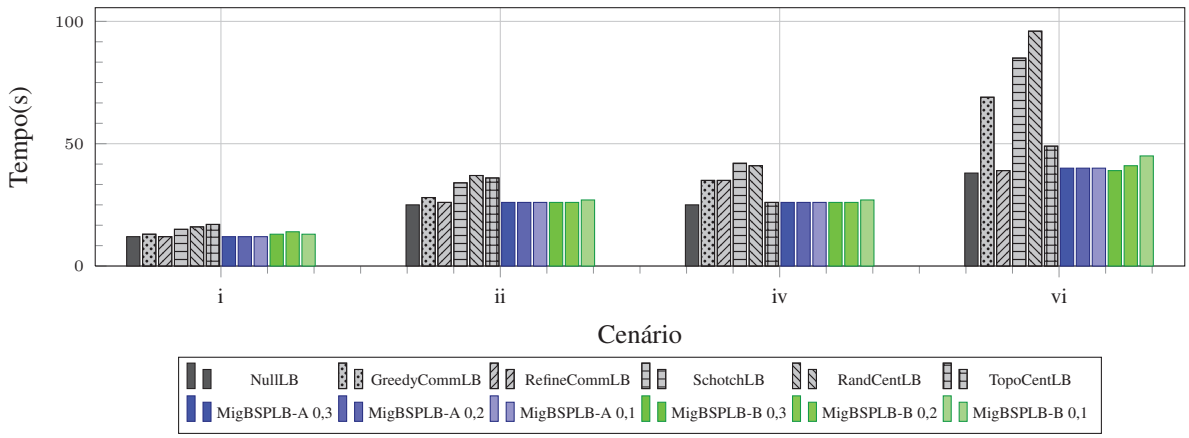
Figura 45 – Sobrecarga x Cenário (II)– Shearsort
Sobrecarga da estratégia *MigBSPLB-B*



Nos primeiros casos apontados (i,ii, iv e vi), a execução sem estratégia (utilizando o *NullLB*) obteve o menor tempo de execução. Este resultado sugere que há casos nos quais, apesar das

estratégias escolherem um suposto melhor escalonamento, elas adicionam um custo computacional mais elevado que não traz real benefícios à aplicação. Na análise geral da Tabela 12, considerando os cenários nos quais a execução obteve redução de tempo em relação a estratégia nula, a maioria das estratégias nativas do *Charm++* não obteve resultados satisfatórios. A que se destacou foi a *RefineCommLB*. Nos casos iii, v, vii e viii, com a utilização do *MigBSPLB-B* houve a redução de tempo na execução das tarefas de 32% , 39%, 40% e 38%, respectivamente. Os melhores resultados foram com $\alpha = 2$ e com valor de D, na maioria dos casos, igual a 0,3. Utilizando estes números em comparação com a estratégia nativa com melhor resultado, a redução de tempo foi de 35%, 6%, 20% e 16%. Uma observação sobre o uso do α: se o tempo de computação do super-passo for alto, apesar de ser adaptativo, a primeira intervenção ocorrerá somente no instante $\alpha\times$ tempo do super-passo. Desta forma, os dados acima sugerem um melhor resultado com valores de α baixo. Apesar do *MigBSPLB-A* não ter obtido o melhor resultado, na maioria dos casos, os tempos obtidos foram menores do que os ensaios com a *RefineCommLB*.

Figura 46 – Tempo de Execução x Cenário (I)– Shear-Sort

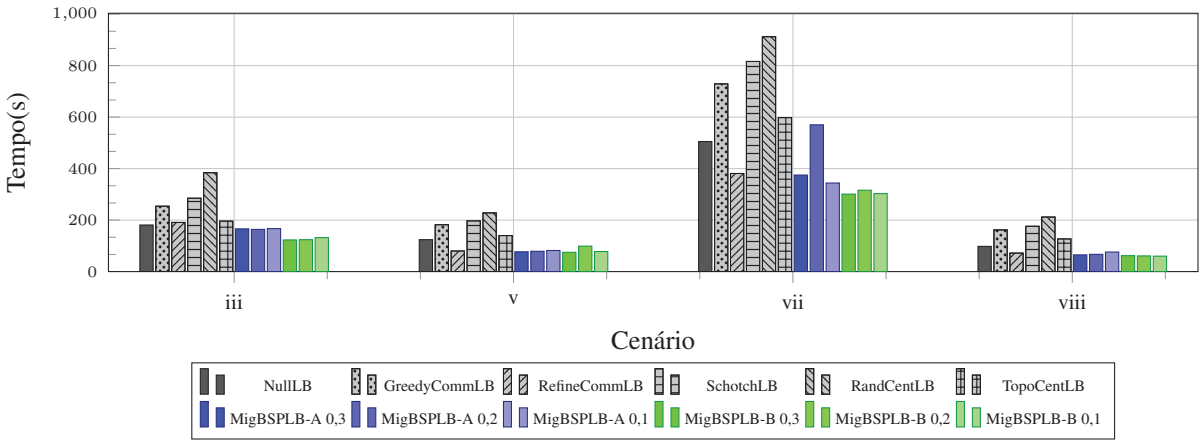


Os gráficos das Figuras 46 e 47 mostram de forma mais visual a comparação dos resultados do *MigBSPLB-A* e B, com $\alpha = 2$ e as demais estratégias. Os valores dos gráficos foram agrupados de acordo com a escala dos resultados obtidos.

A Figura 46 apresenta os cenários com o tempo de execução abaixo de 100 segundos. Nenhuma delas apresentou redução no tempo de execução em relação ao *NullLB*. Como dito anteriormente, isso indica que se a execução for de tempo relativamente baixo, o uso das estratégias pode não ser uma opção interessante para as aplicações BSP. Apesar deste gráfico não possuir resultados realmente satisfatórios, em relação ao *MigBSPLB-A* e *MigBSPLB-B*, observa-se que a versão que representa o MigBSP, na maioria das vezes apresentou tempos de execução menores do que os resultados com a versão do MigBSP++.

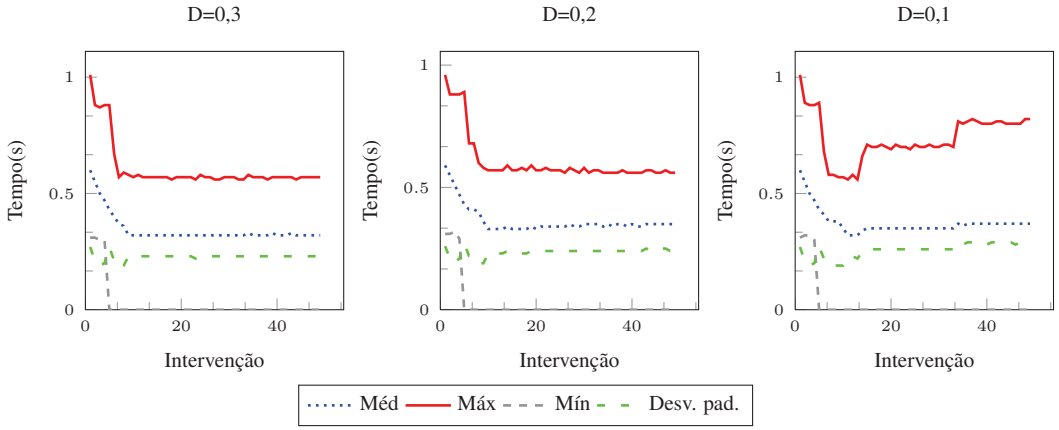
Já a Figura 47, que apresenta os cenários que possuem o tempo de execução acima de 100 segundos, em todas elas a *MigBSPLB-B* (MigBSP++) apresentou redução de tempo de

Figura 47 – Tempo de Execução x Cenário (II) – Shear-Sort



execução sobre qualquer estratégia. As execuções com a versão do MigBSP, em alguns casos, não foram tão satisfatórios. Nos próximos gráficos, o comportamento dos super-passos em casos específicos são apresentados.

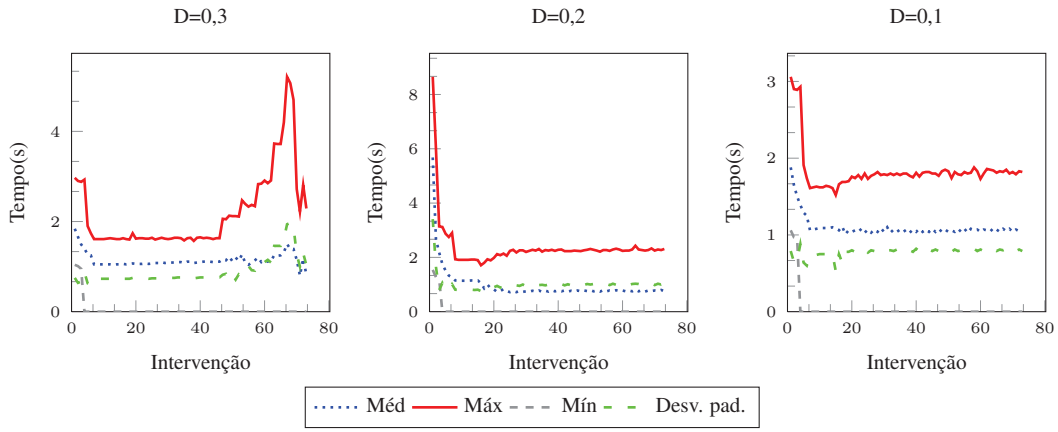
Figura 48 – Tempo do Super-passo x Intervenção da Estratégia *MigBSPLB-A* (viii)



Os gráficos da Figura 48 e os gráficos subsequentes similares, são formados observando cada processador e coletando as estatísticas do máximo tempo de execução (Máx), mínimo (Mín), médio (Méd) e desvio padrão (Desv.pad.). O eixo da abcissa representa o índice da intervenção e não o tempo de execução. O maior tempo dos processadores define a duração do super-passo da aplicação entre duas intervenções do balanceador de carga. Os gráficos demostram que a execução com a versão A do *MigBSPLB*, no cenário viii, mantém o valor máximo acima de 0.5 segundos (s). Isso indica que existe pelo menos 1 processador que demora mais de 0.5 s para entrar na chamada do $MPI\_Migrate()$. O super-passo mais rápido, representado pelo traçado Mín, indica que existe pelo menos 1 processador que teve todas as tarefas retiradas, ou seja, não contribuiu para a execução da aplicação no sistema. Outra observação é o valor do desvio padrão, na mesma ordem de grandeza da média, e que não diminui. Isso representa que há uma grande diferença entre o tempo de execução entre os processadores e que se mantém até

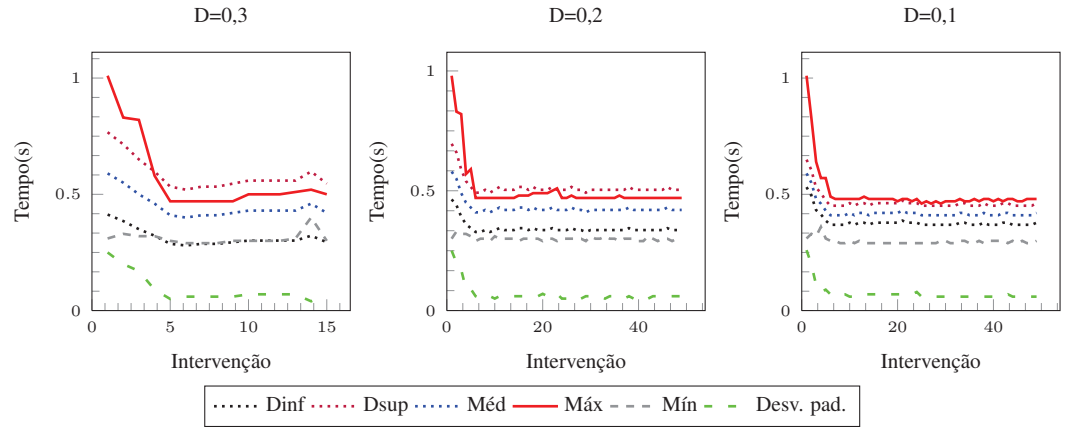
o fim da aplicação. O ideal seria que os tempos destes processadores estivessem equilibrados e o desvio padrão mais próximo de 0 possível.

Figura 49 – Tempo do Super-passo x Intervenção da Estratégia *MigBSPLB-A* (vii)



Nos gráficos representados na Figura 49, é possível observar que o comportamento do desvio padrão é similar aos apresentados anteriormente. Apesar do tempo do super-passo ficar abaixo dos 2 s logo no início da execução, o gráfico com D = 0,3 apresentou um comportamento inesperado próximo à intervenção 50, tornando mais longo o tempo do super-passo. Este resultado fez com que o valor do desvio padrão aumentasse, demostrando o desequilíbrio do sistema. Nas execuções com D=0,2 e D=0,1, em menos de 10 intervenções, o tempo do super-passo foi reduzido de 8 s para próximo de 2 s, e de 3 s para menos do que 2 s respectivamente. Novamente, os gráficos mostraram que um processador ficou sem utilização devido ao traçado de tempo mínimo chegar a 0 s.

Figura 50 – Tempo do Super-passo x Intervenção da Estratégia *MigBSPLB-B* (viii)

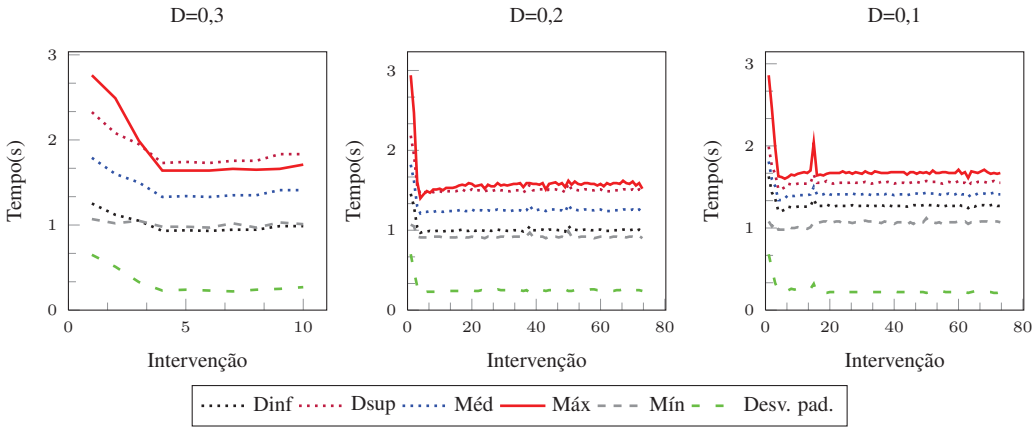


Na legenda dos gráficos que representam o *MigBSPLB-B* (MigBSP++), *Dinf* e *Dsup* auxiliam na visualização do intervalo de tempo entorno da média. O sistema foi considerado em

equilíbrio sempre que os traços *Máx* e *Mín* estiveram dentro do intervalo. Um cenário distinto a esse descrito expressa desarmonia no sistema.

Nos ensaios com o *MigBSPLB-B*, conforme a Figura 50, observa-se que o desvio padrão do tempo de computação dos processadores reduz a valores abaixo de 0,1 s já na quinta intervenção. Outra situação interessante que o gráfico com D= 0,3 mostra é que a quantidade de intervenções até o fim da aplicação é menor do que nos casos onde D é igual a 0,2 e 0,1. Isso indica que, no primeiro caso, o valor de equilíbrio foi alcançado e as intervenções seguintes eram desnecessárias, uma vez que os valores de Máx e Mín permaneciam situados entre Dinf e Dsup. Quando D é igual a 0,2, apesar do valor máximo estar dentro da faixa na maior parte do tempo, pelo menos um dos processadores executou os seus super-passos de forma mais rápida do que os demais, e o MigBSP++ detectou esta situação como desequilíbrio. No caso onde D é igual 0,1,os valores de Máx e Mín estão fora da faixa indicada. Nessa situação, onde não é possível alcançar um equilíbrio melhor, a intervenção do balanceador de carga deve ser vista apenas como acréscimo de sobrecarga.

Figura 51 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-B* (vii)



Observando a sequência de gráficos da Figura 51, a utilização do *MigBSPLB-B* apresenta um resultado no qual o super-passo converge nas primeiras intervenções. Novamente na execução onde D é igual a 0,3, o número de intervenções são menores, demonstrando que o balanceamento definido foi alcançado antes da quinta intervenção. Na quarta, o desvio padrão é menor do que 0,2 s. Na execução com D igual a 0,2, no início das intervenções o valor máximo do super-passo ficou abaixo de 1,5s. Entretanto, os valores de Máx e Mín não se mantém dentro da faixa do estado de equilíbrio. No gráfico com D igual a 0,1, é possível notar que próximo da intervenção 20 o tempo do super-passo se eleva, mas rapidamente é reduzido pela atuação do MigBSP++.

6.2 Discussão dos Ensaios com Shear-Sort

Baseado nos resultados dos ensaios, observou-se que o MigBSP++ não demonstrou bons resultados em todos os casos. Nos ensaios i,ii, iv e vi, não houve melhora na execução das aplicações com a estratégia proposta. Em compensação, nenhuma das estratégias nativas utilizadas para comparação obteve um resultado melhor do que a estratégia nula. Nestes cenários a não utilização do balanceamento de carga é mais eficiente. Nos casos restantes, é notável a melhora que a estratégia *MigBSPLB-B* demonstrou em relação as demais, a medida que a quantidade de VPs aumentava. Apesar da estratégia *MigBSPLB-A* também ter demonstrado resultados melhores sobre as estratégias nativas nestes cenários, existem casos nos quais ela não foi superior a *RefineCommLB*. Em relação a sobrecarga apresentada pela estratégia em estudo, os valores ficaram abaixo de 15%.

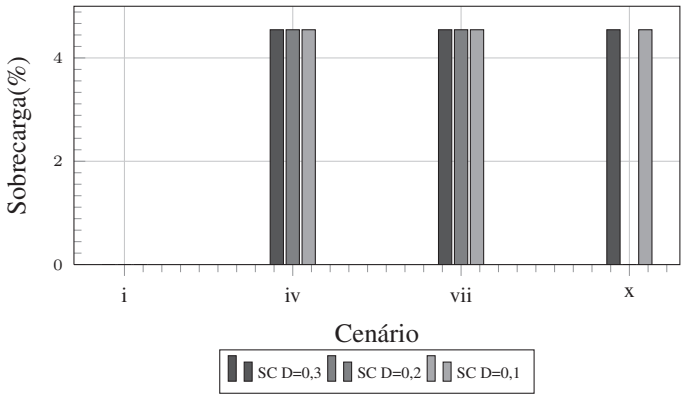
6.3 Avaliação das estratégias – Aplicação FIC

Os resultados foram agrupados em função da quantidade de *domains* utilizados para cada ensaio. A primeira parte apresenta os resultados com 1888 *domains*. Os tempos médios de execução estão expostos na Tabela 13. Para representar a sobrecarga introduzida pela estratégia *MigBSPLB-B* em cada execução, foi mantida a sigla de referência SC.

Tabela 13 – Média do Tempo de Execução – FIC 1888 Domains

Estratégia		i	iv	vii	x
NullLB		22	22	22	22
GreedyCommLB		25	54	30	28
RefineCommLB		22	15	22	14
ScotchLB		31	34	37	41
RandCentLB		34	35	32	33
TopoCentLB		25	26	34	36
SC(D=0.3)		22	23	23	23
SC(D=0.2)		22	23	23	22
SC(D=0.1)		22	23	23	23
MigBSPLB-A(D=0.3)		17	17	20	20
MigBSPLB-A(D=0.2)	α=2	17	17	20	21
MigBSPLB-A(D=0.1)		17	19	21	21
MigBSPLB-B(D=0.3)		38	17	18	17
MigBSPLB-B(D=0.2)		38	16	18	18
MigBSPLB-B(D=0.1)		38	16	18	17
SC(D=0.3)		22	22	22	22
SC(D=0.2)		22	22	22	22
SC(D=0.1)		22	22	22	22
MigBSPLB-A(D=0.3)		22	19	22	22
MigBSPLB-A(D=0.2)	α=4	22	18	21	22
MigBSPLB-A(D=0.1)		22	19	21	22
MigBSPLB-B(D=0.3)		35	17	19	20
MigBSPLB-B(D=0.2)		35	17	19	19
MigBSPLB-B(D=0.1)		35	17	19	19
SC(D=0.3)		22	22	23	22
SC(D=0.2)		22	22	22	22
SC(D=0.1)		22	22	23	22
MigBSPLB-A(D=0.3)		20	22	24	23
MigBSPLB-A(D=0.2)	α=8	20	22	24	23
MigBSPLB-A(D=0.1)		20	22	24	23
MigBSPLB-B(D=0.3)		32	19	21	21
MigBSPLB-B(D=0.2)		32	19	21	22
MigBSPLB-B(D=0.1)		32	19	21	22
Menor tempo		17	15	18	14

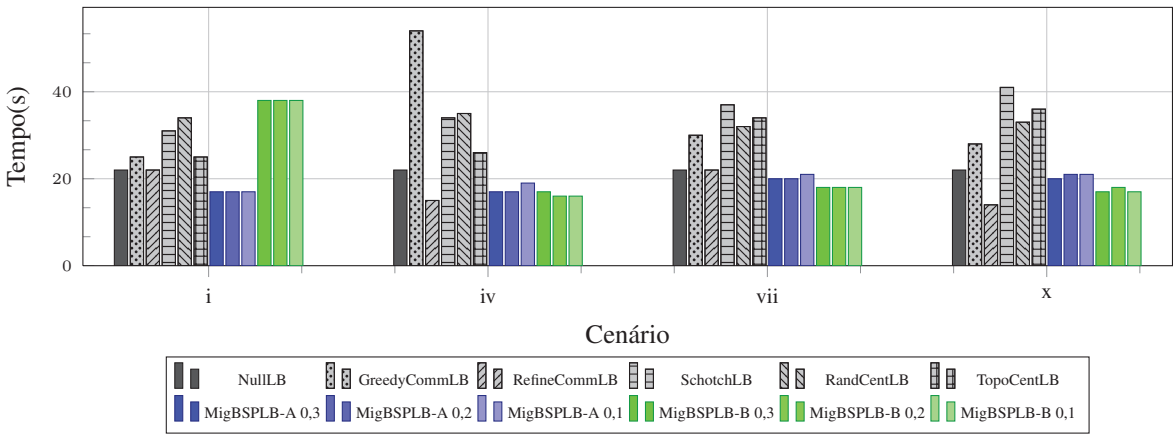
Figura 52 – Sobrecarga x Cenário – Fractal 1888 Domains
Sobrecarga da estratégia *MigBSPLB-B*



No entanto, considerando os cenários i, iv, vii e x, o *MigBSPLB-B* não foi o modelo mais eficaz na tentativa de redução de tempo de tarefas. Novamente pode-se observar que, das estratégias nativas utilizadas para comparação, a *RefineCommLB* obteve melhor resultado do que as outras. A estratégia *MigBSPLB-A* teve um desempenho melhor nas execuções no cenário i e vii. Já a estratégia *MigBSPLB-B*, obteve uma execução entre 6 e 123% mais lenta do que o melhor tempo de cada um desses cenários. Ou seja, em nenhum destes resultados a estratégia *MigBSPLB-B* obteve um desempenho favorável.

Realizando uma análise da sobrecarga apresentada pela *MigBSPLB-B*, o gráfico da Figura 52 mostra que o valor máximo ficou abaixo de 5%. Estes resultados representam que os valores são muito próximos aos da estratégia nula, indicando que a sobrecarga nestes casos é irrelevante para escala da medida de tempo utilizada. O método de cálculo é o mesmo apresentado pela equação 6.1.

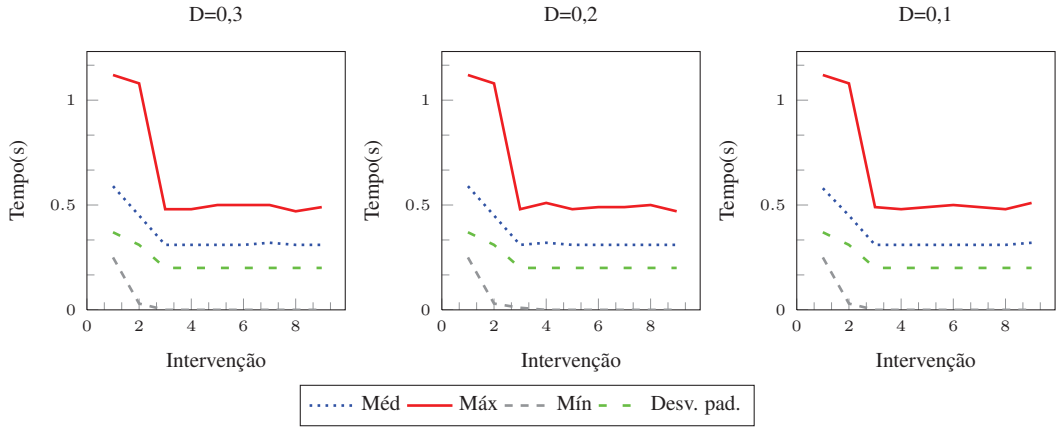
Figura 53 – Tempo de Execução x Cenário – FIC 1888 Domains
Tempos de Execução – FIC



Observando o gráfico 53, é possível notar que apenas no cenário vii a *MigBSPLB-B* obteve ganho sobre as demais. Porém, o cenário i apresenta um resultado favorável ao *MigBSPLB-A*.

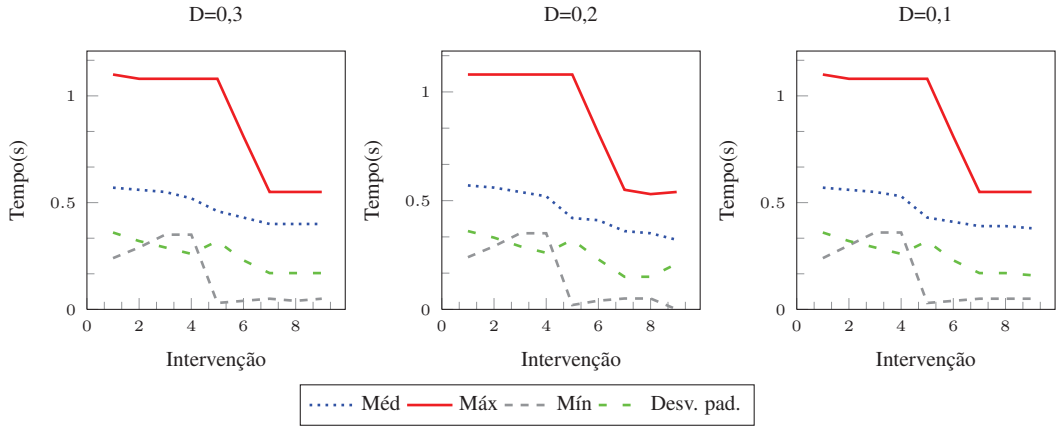
Neste cenário, a quantidade de processos é igual ao número de processadores, e este resultado de desempenho favorável ao MigBSPLB-A já era esperado. De acordo com os ensaios com o algoritmo anterior, o *MigBSPLB-B* não obteve bons resultados com tempos de execução baixos. Apesar de não ter o melhor desempenho das estratégias demonstradas, a *MigBSPLB-B* obteve uma redução de tempo de até 27% em relação a estratégia nula, demostrando ser solução viável em alguns casos.

Figura 54 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-A* (i)



Na análise dos tempos das tarefas (Figura 54) no cenário onde a estratégia *MigBSPLB-A* obteve o melhor resultado, é possível notar que duas intervenções foram necessárias para que o valor do processador com tempo maior fosse reduzido de 1,1 s para aproximadamente 0,5 s. Entretanto, houve pelo menos 1 processador que ficou ocioso nos três casos. Isso aconteceu devido a quantidade de VPs utilizada, 7, e a estratégia procurou alocar os processos onde ocorresse redução de tempo da tarefa.

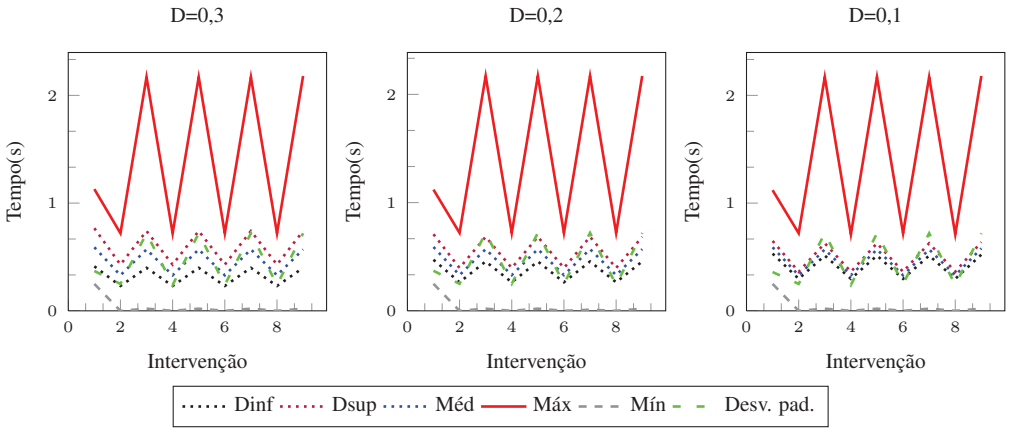
Figura 55 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-A* (x)



Na execução da *MigBSPLB-A* utilizando 28 processos, Figura 55, nos três casos os super-

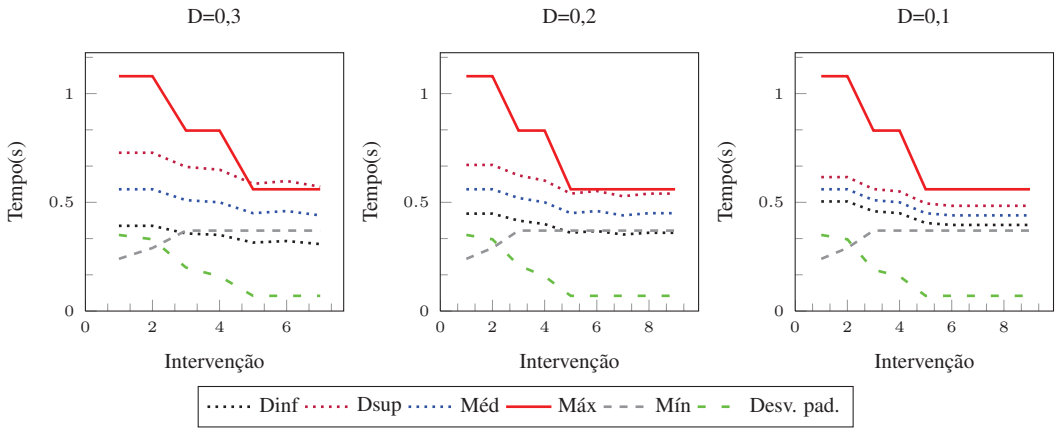
passos permanecem em 1 s durante as 5 primeiras intervenções. No super-passo seguinte, o tempo é reduzido a quase 0,5 s. Diferentemente do caso i, neste cenário não existe momento em que algum processador fica inativo. Isso representa um uso mais adequado dos recursos. O desvio padrão do tempo dos processadores varia durante as intervenções demonstrando que, apesar da diferença diminuir, o tempo das sincronizações não se aproximam.

Figura 56 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-B* (i)



Na observação da execução com a estratégia *MigBSPLB-B* do cenário i, Figura 56, os tempos dos super-passos variaram constantemente devido a disponibilidade de um processador ocioso sempre que alguma migração ocorria. Logo, a cada intervenção acontecia um desbalanceamento que era detectado. Este comportamento é consequência de uma quantidade de processos e processadores iguais. Dado que a recomendação de operação com o *Charm++* deve ocorrer com o número de processos maior do que a de processadores (KALE; ZHENG, 2009), este caso dificilmente acontecerá em uma aplicação de ambiente de produção. Observando o desvio padrão percebe-se que não há uma convergência e existe uma tendência oscilatória.

Figura 57 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-B* (x)



Com 28 VPs, a estratégia em estudo realizou um escalonamento mais adequado do que no caso anterior i. Na Figura 57, o gráfico com D igual a 0,3, mostra que o balanceamento foi alcançado na quinta intervenção. A quantidade de intervenções foi menor do que nos outros experimentos. Nos gráficos com D igual 0,2 e 0,1 o desempenho seguiu conforme previsto, porém não superou os resultados obtidos com a *RefineCommLB*.

A Tabela 14 possui os resultados das execuções com 7854 *domains*. Neles é possível perceber que no cenário ii a *MigBSPLB-A* obteve o melhor resultado. Nos outros cenários, tanto a *MigBSPLB-B*, como a *RefineCommLB* obtiveram tempos equivalentes, sem apresentar grandes vantagens de uma estratégia sobre a outra. Nos casos onde houve redução, o *MigBSPLB-B* alcançou até 41% de melhora em relação a estratégia nula.

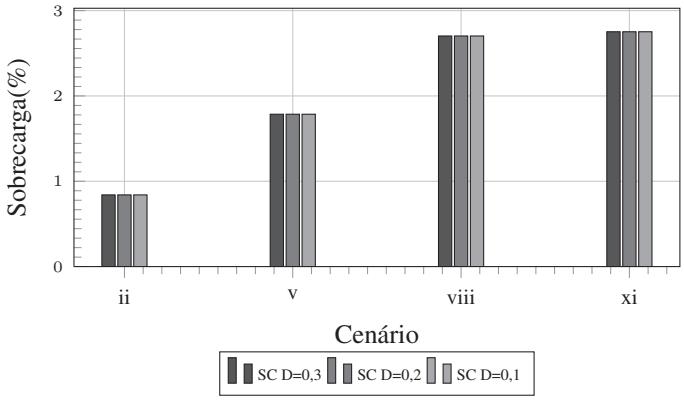
Tabela 14 – Média do Tempo de Execução – FIC 7854 Domains

Estratégia		ii	v	viii	xi
NullLB		119	112	111	109
GreedyCommLB		124	181	146	147
RefineCommLB		120	65	83	63
ScotchLB		165	162	153	177
RandCentLB		203	176	170	175
TopoCentLB		127	139	162	157
SC(D=0.3)	α=2	120	114	114	112
SC(D=0.2)		120	114	114	112
SC(D=0.1)		120	114	114	112
MigBSPLB-A(D=0.3)		70	69	83	71
MigBSPLB-A(D=0.2)		78	71	83	72
MigBSPLB-A(D=0.1)		94	76	83	89
MigBSPLB-B(D=0.3)		196	65	75	68
MigBSPLB-B(D=0.2)		200	65	76	70
MigBSPLB-B(D=0.1)		199	70	83	70
SC(D=0.3)	α=4	120	113	113	111
SC(D=0.2)		120	113	113	111
SC(D=0.1)		121	113	113	111
MigBSPLB-A(D=0.3)		74	71	84	74
MigBSPLB-A(D=0.2)		79	73	84	74
MigBSPLB-A(D=0.1)		80	75	75	75
MigBSPLB-B(D=0.3)		203	67	84	75
MigBSPLB-B(D=0.2)		203	66	84	69
MigBSPLB-B(D=0.1)		203	74	83	71
SC(D=0.3)	α=8	121	113	113	112
SC(D=0.2)		120	113	113	112
SC(D=0.1)		120	113	113	112
MigBSPLB-A(D=0.3)		76	75	88	77
MigBSPLB-A(D=0.2)		79	75	88	77
MigBSPLB-A(D=0.1)		90	76	84	77
MigBSPLB-B(D=0.3)		199	69	85	76
MigBSPLB-B(D=0.2)		199	69	85	75
MigBSPLB-B(D=0.1)		198	74	85	74
Menor tempo		70	65	75	63

Analisando a sobrecarga nestes cenários, a Figura 58 mostra que ocorreu um crescimento com o aumento de VPs. Porém os valores não ultrapassaram 3% do tempo da estratégia nula. A maior sobrecarga ocorreu nos cenários viii e xi, com valor de 2,7%. Esses percentuais indicam que a *MigBSPLB-B* introduziu menos sobrecarga em relação ao tempo total da aplicação se comparado aos casos anteriores.

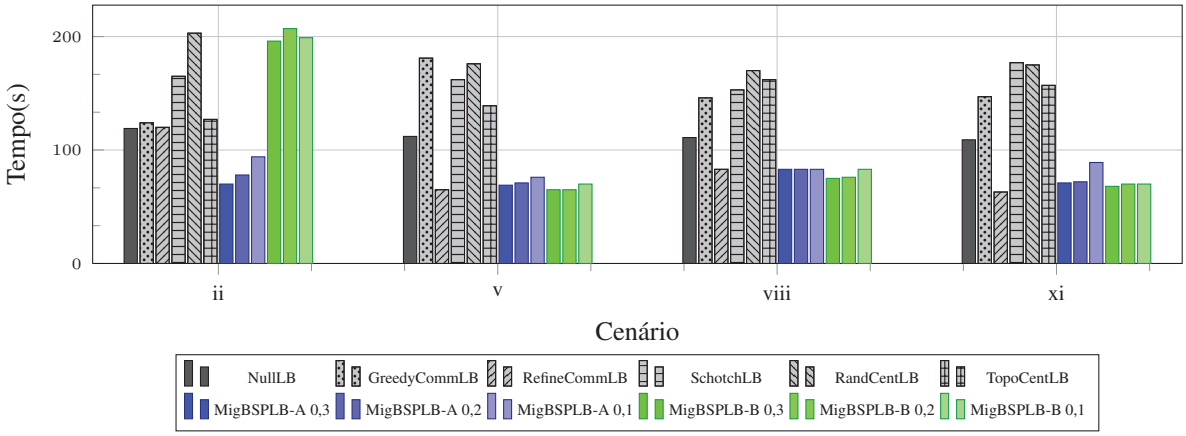
No gráfico da Figura 59, a estratégia *MigBSPLB-A* apresentou o melhor resultado no cenário ii. Mesmo com a redução de até 41%, os resultados não se mantiveram tão satisfatórios nos demais cenários. Porém, permaneceu mais adequada do que sem estratégia. Em v, a *RefineCommLB* e a *MigBSPLB-B* obtiveram o mesmo desempenho, ficando com também 41% de

Figura 58 – Sobrecarga x Cenário – Fractal 7854 Domains
Sobrecarga da estratégia *MigBSPLB-B*



vantagem em relação a estratégia nula. No caso viii, a *MigBSPLB-B* obteve uma melhora de 32% sobre a estratégia nula, porém ficou apenas com o tempo 9% melhor do que a estratégia *RefineCommLB*. Já no cenário xi, apesar de obtido um resultado de 7% inferior em relação ao *RefineCommLB*, apresentou uma melhora de 37% em relação a estratégia nula.

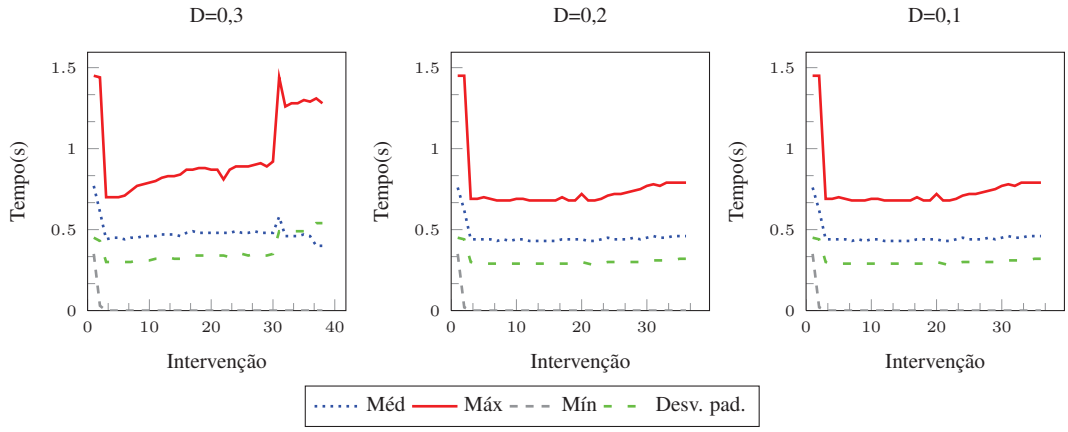
Figura 59 – Tempo de Execução x Cenário – FIC 7854 Domains
Tempos de Execução – FIC



Ao análisar os super-passos na execução da estratégia *MigBSPLB-A* no cenário ii (Figura 60) percebe-se nos três casos que, pelo menos, um processador ficou ocioso. Mesmo assim a vantagem desta estratégia foi visível nesta relação de 1:1 na quantidade de processos e processadores. As curvas dos ensaios com D igual a 0,2 e 0,1 mantiveram um comportamento similar. Porém, no caso onde D é igual a 0,3, ocorreu um aumento do tempo de sincronização. O desvio padrão, na maioria das intervenções, permaneceu em 0,3 s, demonstrando a variação existente entre os tempos de sincronização.

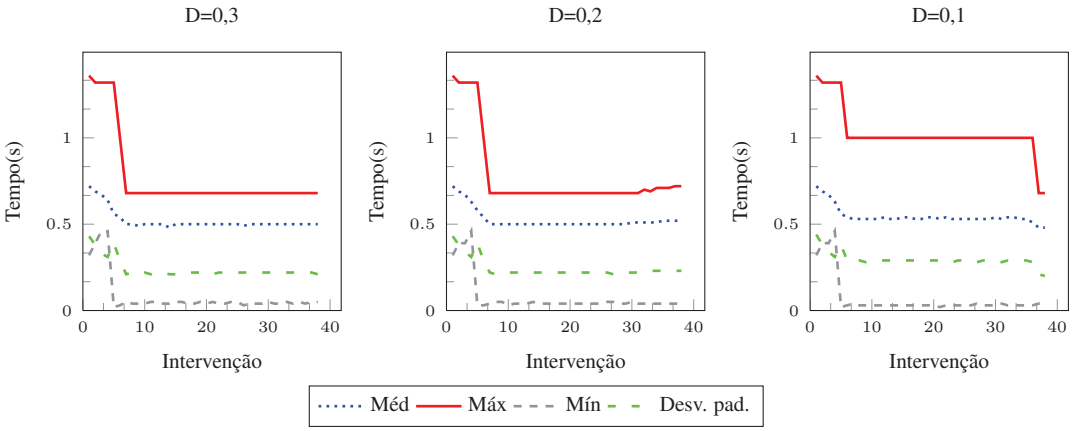
Quando a quantidade de VP é maior, observamos que a redução do tempo dos processadores até a sincronização foi de aproximadamente 0,7 s nos dois primeiros gráficos da Figura 61. Quando a restrição de balanceamento é 0,1, a estratégia não foi capaz de reduzir o tempo do

Figura 60 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-A* (ii)



super-passo que permaneceu em 1 s na maior parte do tempo da execução. É importante observar que pelo menos um processador sincronizava com menos de 0,1 s em todos os casos. Isso indica que os recursos do sistema foram utilizados, porém, não aconteceu um reescalonamento adequado.

Figura 61 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-A* (xi)



Os gráficos da Figura 62 demonstram que a falta de processos provocam uma instabilidade na estratégia *MigBSPLB-B*. Nos três casos houveram momentos nos quais os tempos dos super-passos variaram a 3s a 1s na maioria das intervenções. Esta variação do desvio padrão mostra que os processadores não mantiveram uma regularidade. No primeiro e no último gráfico, ocorreu um curto intervalo de tempo onde o super-passo ficou abaixo de 1s. Entretanto, os ensaios apresentaram resultados insatisfatórios.

No cenário xi, observamos que os processadores partem de um valor de tempo de sincronização elevado e a estratégia os reduzem para 0,7 s. Nestes cenários, foram necessárias de 5 a 6 intervenções para chegar a este valor. Novamente, como em ensaios anteriores, a execução

Figura 62 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-B* (ii)

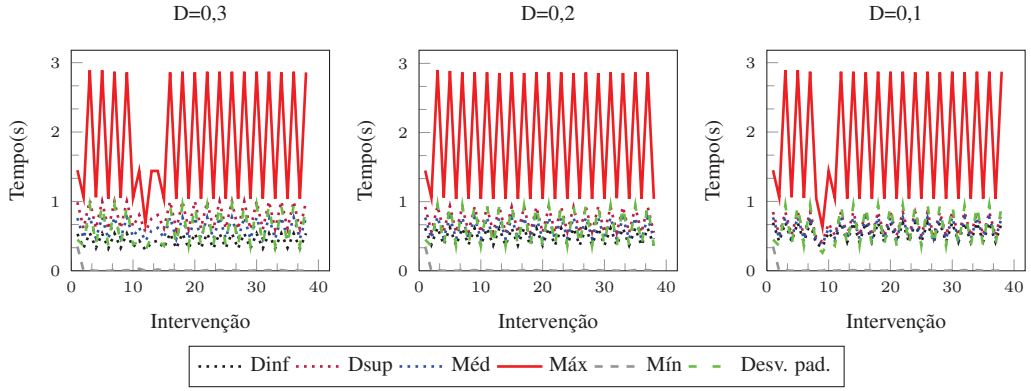
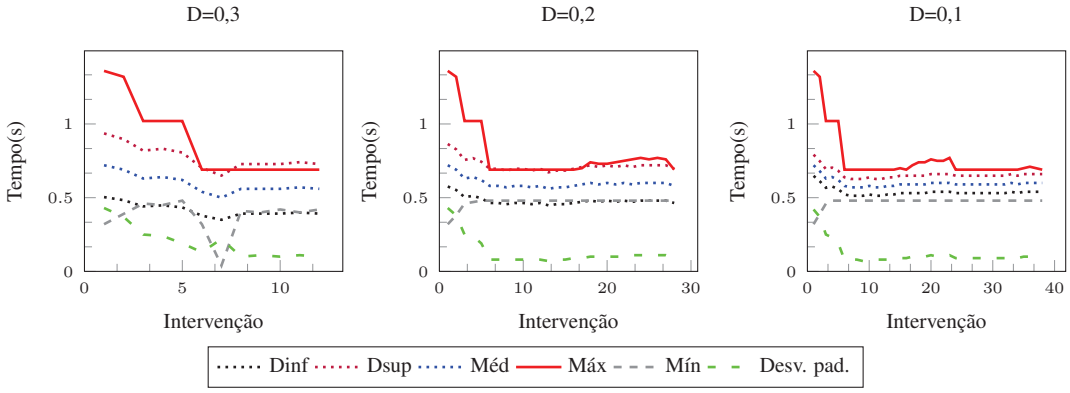


Figura 63 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-B* (xi)



com D igual a 0,3 alcançou o equilíbrio. No gráfico D igual a 0,2, também houve momentos de equilíbrio dentro da faixa estipulada e com D igual 0,1, o balanceamento não foi atingido e teve constantes intervenções. Na Figura 63 observamos que, nos três gráficos, o desvio padrão reduziu rapidamente. Após a 7ª intervenção, o valor estava abaixo de 0,1s.

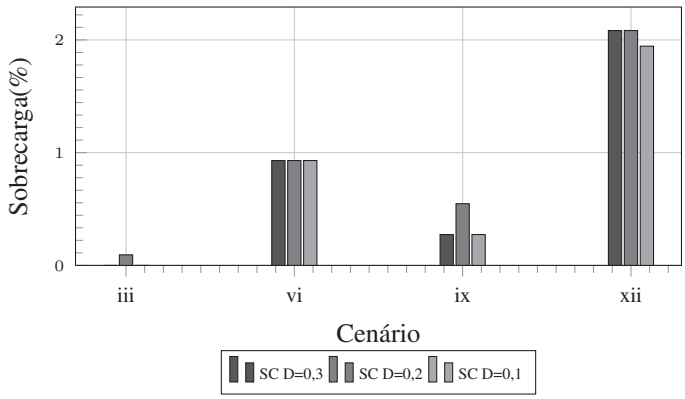
A Tabela 15 demonstra os ensaios com os tempos mais elevados de execução. Nestes ensaios é possível observar os casos nos quais a *MigBSPLB-B* obteve vantagens sobre as estratégias convencionais. Com base também nos ensaios anteriores, pode-se afirmar que a estratégia em estudo demonstrou melhores resultados quando a quantidade de VPs era 21 ou 28. Com exceção dos casos com 7 VPs, em todos os outros o tempo de execução foi mais rápido do que o caso onde a estratégia nula estava em uso.

Comparando a sobrecarga que a *MigBSPLB-B* coloca sobre a aplicação, a Figura 64 demonstra que, nestes cenários, a sobrecarga máxima foi de aproximadamente 2%. Isso ocorreu porque, ao reduzir o tamanho dos *ranges*, a busca realizada entre os *domains* ocupou mais tempo de processamento, mas reduziu o tempo de comunicação. E como o tempo de análise da estratégia se mantém relativamente semelhante, a relação da sobrecarga é reduzida.

Tabela 15 – Média do Tempo de Execução – FIC 32026 Domains

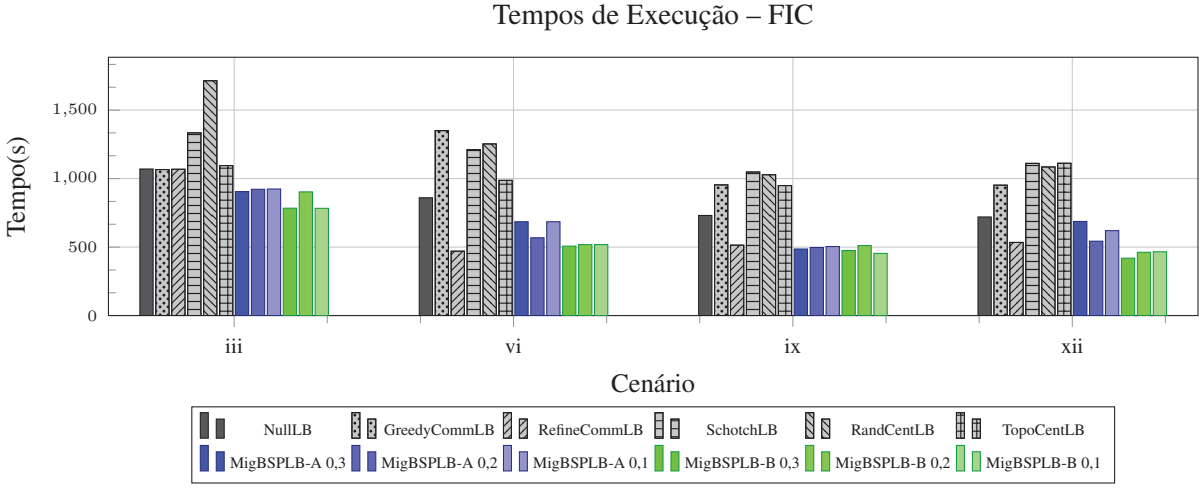
Estratégia		iii	vi	ix	xii
NullLB		1070	860	731	720
GreedyCommLB		1066	1350	954	952
RefineCommLB		1068	470	514	534
ScotchLB		1334	1211	1048	1111
RandCentLB		1714	1253	1028	1085
TopoCentLB		1095	987	949	1112
SC(D=0.3)		1070	868	733	735
SC(D=0.2)		1071	868	735	735
SC(D=0.1)		1070	868	733	734
MigBSPLB-A(D=0.3)		905	685	486	687
MigBSPLB-A(D=0.2)	α=2	922	568	497	543
MigBSPLB-A(D=0.1)		924	685	504	620
MigBSPLB-B(D=0.3)		784	507	475	419
MigBSPLB-B(D=0.2)		903	518	511	462
MigBSPLB-B(D=0.1)		783	518	454	466
SC(D=0.3)		1071	867	743	735
SC(D=0.2)		1072	869	742	734
SC(D=0.1)		1071	868	743	735
MigBSPLB-A(D=0.3)		893	496	465	654
MigBSPLB-A(D=0.2)	α=4	915	624	499	592
MigBSPLB-A(D=0.1)		914	680	480	672
MigBSPLB-B(D=0.3)		785	514	487	417
MigBSPLB-B(D=0.2)		785	518	488	467
MigBSPLB-B(D=0.1)		785	520	486	464
SC(D=0.3)		1073	867	742	730
SC(D=0.2)		1072	869	745	730
SC(D=0.1)		1073	867	743	730
MigBSPLB-A(D=0.3)		912	676	480	705
MigBSPLB-A(D=0.2)	α=8	961	510	511	540
MigBSPLB-A(D=0.1)		979	693	496	617
MigBSPLB-B(D=0.3)		788	516	491	422
MigBSPLB-B(D=0.2)		806	520	491	470
MigBSPLB-B(D=0.1)		866	522	468	469
Menor tempo		783	470	454	417

Figura 64 – Sobrecarga x Cenário – Fractal 32026 Domains
Sobrecarga da estratégia *MigBSPLB-B*



No gráfico da Figura 65, a estratégia *MigBSPLB-B* apresentou o melhor resultado nos cenários iii, ix e xiii. Em relação a estratégia nula, a redução de tempo foi significante. No cenário iii a redução no tempo de execução atingiu 27%. No cenário vi, 46%. Em ix a redução foi de 37% e 42% no cenário xii. Em relação a estratégia *RefineCommLB* nos cenários iii , ix e xii ela obteve uma redução de 7%, 11% e 21% respectivamente. No cenário vi a *RefineCommLB* obteve uma vantagem de 5% no tempo de execução. É importante ressaltar que ocorreram casos onde a estratégia *MigBSPLB-A* obteve um desempenho melhor do que a estratégia *RefineCommLB*.

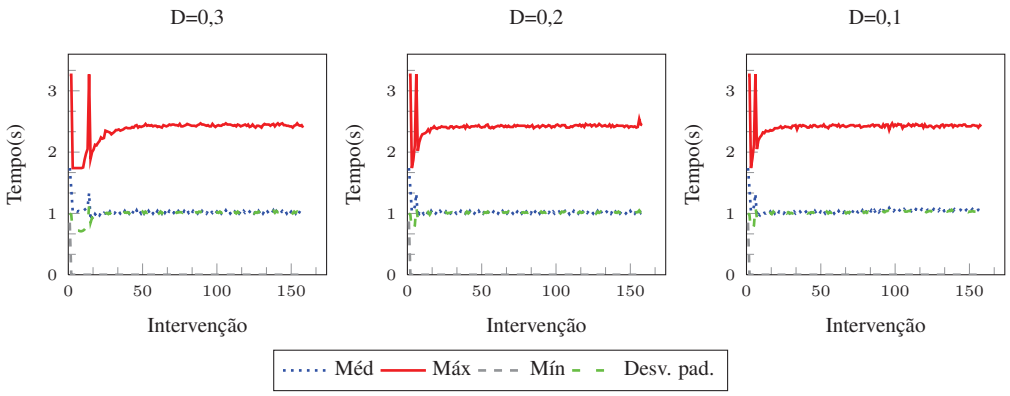
Figura 65 – Tempo de Execução x Cenário – FIC 32026 Domains



No cenário ix, foi obtida uma redução de 5%.

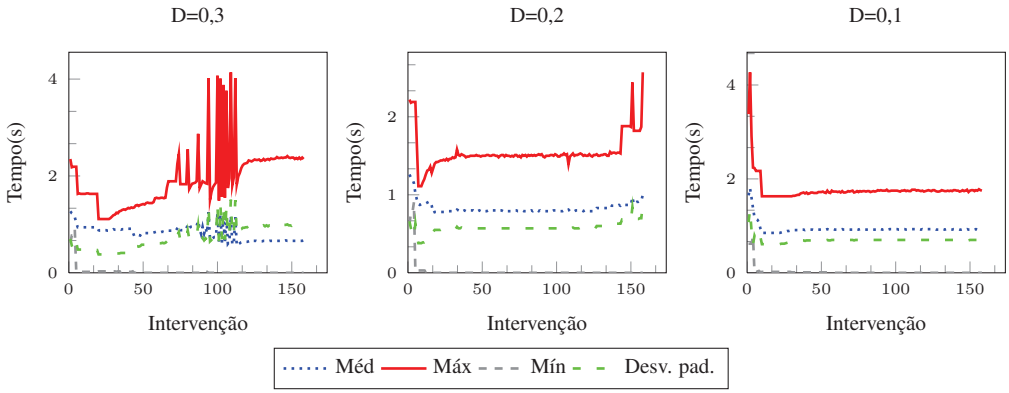
Na análise dos super-passos durante a execução da estratégia *MigBSPLB-A* no cenário iii (Figura 66), observou-se que nos três casos, pelo menos, um processador fica ocioso e o tempo do super-passo é de quase 2,5 s. Com base no desvio padrão, os processadores estavam com valores muito diferentes e isto demonstrou que a aplicação ficou com o tempo de sincronização muito elevado. Mesmo assim, esta estratégia obteve um resultado melhor do que as nativas utilizadas nos ensaios.

Figura 66 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-A* (iii)



Os gráficos da Figura 67 demonstram uma situação interessante para observação. No caso em que D é igual a 0,3, nas primeiras intervenções ocorreu uma redução do tempo total do super-passo. Como nesta estratégia não há o APBSP, ele permaneceu migrando os VPs. Em seguida, o tempo do super-passo cresceu lentamente até atingir um estado de instabilidade, alcançando valores de até aproximadamente 4s. Esse comportamento ocorreu devido a uma maior quantidade de tarefas do que processadores. Consequentemente os VPs mais lentos são transferidos para os processadores mais rápidos. Em seguida, a estratégia encontra um novo

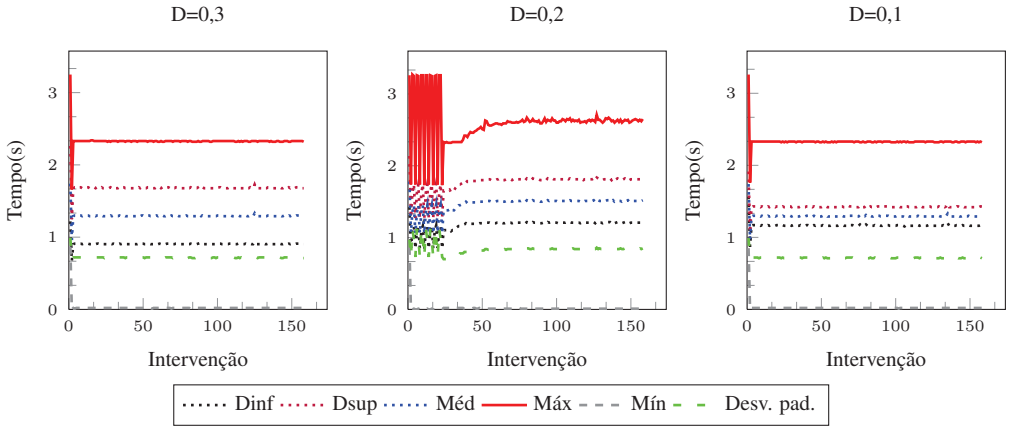
Figura 67 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-A* (xii)



escalonamento onde permanece com o tempo de super-passo de 2,5 s. No segundo gráfico, como a quantidade de processos prováveis fora do intervalo de equilíbrio é maior, a estratégia possui mais opções para ajustar o escalonamento.

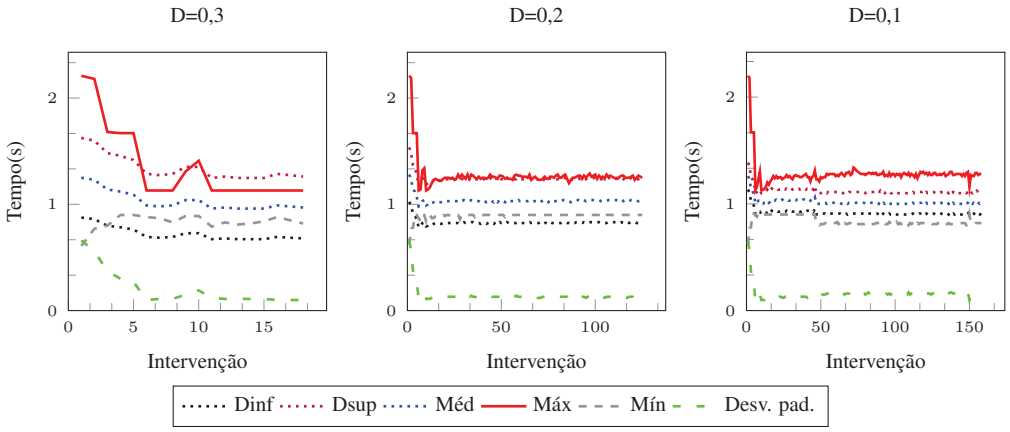
Os gráficos da Figura 68 demonstram que a falta de processos provocou uma instabilidade na estratégia *MigBSPLB-B*. Nos três casos houve momentos nos quais o super-passo oscilou entre 3s e 1s. Porém estabilizou em valores muito elevados. Este efeito é decorrente da impossibilidade da estratégia encontrar uma opção melhor de escalonamento. A variação do desvio padrão mostra que os processadores não mantiveram uma regularidade.

Figura 68 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-B* (iii)



No cenário xi, observamos que a redução do tempo dos processadores até a sincronização é de aproximadamente 0,7s. Novamente, como em ensaios anteriores, a execução com D igual a 0,3 alcançou o equilíbrio. No gráfico D igual a 0,2, houve momentos de balanceamento. Já com D igual 0,1, o balanceamento não foi alcançado e constantemente sofreu intervenções. Na Figura 69 observamos que, nos três gráficos, o desvio padrão reduziu rapidamente. Após a 7ª intervenção o valor estava abaixo de 0,1s.

Figura 69 – Tempo do Super-passo x Intervenção da estratégia *MigBSPLB-B* (xii)



6.4 Discussão Sobre os Ensaios com FIC

Baseado nesses ensaios realizados, a estratégia *MigBSPLB-B*, de acordo com o MigBSP++, mostrou que é capaz de apresentar os melhores resultados com aplicações onde o número de tarefas é maior do que o de processadores. Com exceção dos 3 casos onde a quantidade de processos é igual ao número de processadores, os tempos alcançados foram melhores do que com a utilização da estratégia nula. Porém, os melhores resultados ocorreram nos casos iii, iv, v, vii, ix e xii. A estratégia nativa que se ressaltou foi a *RefineCommLB*. As outras estratégias não obtiveram um resultado com a satisfatoriedade mínima, ou seja, não reduziram o tempo da execução. Isso pode ser um indicativo de que não são apropriadas para este tipo de abordagem com aplicações BSP. Das execuções nas quais foram alcançadas uma similaridade ou melhoria em relação a *RefineCommLB*, a maioria delas ocorreu com D igual 0,3 e α igual a 2.

6.5 Execução Intervenção por Intervenção

Nesta seção é realizada uma análise de uma execução passo-a-passo com a demonstração dos PMs e as decisões tomadas pela estratégia. A execução escolhida pela análise é a que utiliza o algoritmo de compressão de imagens, com a configuração xii.

Intervenção 2: Com o desbalanceamento detectado, a matriz M abaixo é gerada.

M	0	1	2	...	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	...	0	0	0	0,99	0,98	0,99	0,97	1,22	1,21	1,22	1,21
1	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	...	0	0	0	0,22	0,2	0,23	0,17	0,87	0,86	0,87	0,85
3	0	0	0	...	0	0	0	1,27	1,27	1,27	1,25	1,48	1,48	1,48	1,48
4	0	0	0	...	0	0	0	0,56	0,56	0,57	0,53	1,07	1,07	1,07	1,06
5	0	0	0	...	0	0	0	0,62	0,61	0,62	0,58	1,11	1,11	1,11	1,1
6	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0

Nesta etapa, a estratégia escolhe a combinação que possui o último maior PM encontrado. O valor selecionado foi o do VP_{27} para o P_3 . O tempo para chamada da estratégia foi 2,56s. O

APBSP inicia a execução e função de predição, fp (4.12), conclui que o próximo super-passo duraria 2,21s. Então foi realizada uma nova tentativa de migração, porém a função fp retorna um valor acima do anteriormente encontrado, 2,24s. Nesta intervenção somente uma migração ocorre. O valor de α foi carregado com o valor 1.

Intervenção 4: Novamente, o desbalanceamento é detectado e a próxima matriz M é gerada.

M	0	1	2	...	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	...	0	0	0	0	0,96	0,96	0,97	0,95	0,89	0,89	0,9
1	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	...	0	0	0	0	0,2	0,19	0,21	0,15	0,55	0,54	0,56
3	0	0	0	...	0	0	0	0	1,07	1,07	1,07	1,05	1,01	1	1,01
4	0	0	0	...	0	0	0	0	0,54	0,54	0,55	0,51	0,75	0,74	0,76
5	0	0	0	...	0	0	0	0	0,6	0,59	0,61	0,57	0,8	0,79	0,81
6	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0

O primeiro valor selecionado foi o que indica a migração do VP_{23} para o P_3. O tempo na entrada da estratégia era de 2,27s. A fp conclui que o próximo passo iria durar 1,87s. Então foi realizada uma tentativa de uma nova migração, porém a fp retornou um valor acima do anteriormente encontrado, 1,91s. Nesta intervenção somente uma migração ocorreu. O valor de α foi carregado com o valor 1.

Intervenção 6: Novamente o desbalanceamento foi detectado e a próxima matriz M é gerada.

M	0	1	...	16	17	18	19	20	21	22	23	24	25	26	27
0	0	0	...	0,43	0,41	0,42	0	0	0	0	0	0	0,71	0,7	0,71
1	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	...	0	0	0	0	0	0	0	0	0	0,26	0,25	0,26
3	0	0	...	0,3	0,27	0,28	0	0	0	0	0	0	0,63	0,62	0,62
4	0	0	...	0	0	0	0	0	0	0	0	0	0,5	0,49	0,49
5	0	0	...	0	0	0	0	0	0	0	0	0	0,55	0,54	0,54
6	0	0	...	0	0	0	0	0	0	0	0	0	0	0	0

O primeiro valor selecionado foi o que indicava a migração do VP_{27} para o P_1. O tempo na entrada da estratégia foi 1,76s. A fp conclui que o próximo passo irá durar 1,66s. Então foi realizado uma tentativa de nova migração, porém a fp retornou um valor acima do anteriormente encontrado, 1,69s. Nesta intervenção, somente uma migração ocorre. O valor de α foi novamente carregado com o valor 1.

Intervenção 8: Novamente o desbalanceamento foi detectado e a próxima matriz M é gerada.

M	0	1	2	...	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	...	0,23	0,2	0,21	0	0	0	0	0	0	0,27	0,26
1	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	...	0,3	0,27	0,28	0	0	0	0	0	0	0,34	0,34
4	0	0	0	...	0	0	0	0	0	0	0	0	0	0,21	0,21
5	0	0	0	...	0	0	0	0	0	0	0	0	0	0,26	0,25
6	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0

O primeiro valor selecionado foi o que indica a migração do VP_{27} para o P_3. É importante ressaltar que o VP_{27} não é o mesmo das intervenções anteriores. Como explicado anteriormente, entre duas intervenções de uma estratégia não existe garantias que os VPs se manterão os mesmos. O tempo na entrada da estratégia foi 1,75s. A fp conclui que o próximo passo irá durar 1,66s. Então foi realizado uma tentativa de nova migração, porém a fp retornou um valor

acima do anteriormente encontrado, 1,69s. Nesta intervenção somente uma migração ocorre. O valor de α foi carregado com o valor 1.

Intervenção 10: Novamente o desbalanceamento é detectado e a próxima matriz M é gerada.

M	0	1	2	...	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	...	0,26	0,23	0,24	0	0	0	0	0	0	0	0
1	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	...	0,05	0,02	0,03	0	0	0	0	0	0	0	0
4	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0

O primeiro valor selecionado era o que indicava a migração do VP_{17} para o P_0. O tempo na entrada da estratégia foi 1,75s. A fp conclui que o próximo passo irá durar 1,13s. Então é realizado uma tentativa de nova migração, porém a fp retorna um valor acima do anteriormente encontrado, 1,17s. Nesta intervenção somente uma migração ocorre. O valor de α foi carregado com o valor 1.

Intervenção 12: Balanceamento detectado e α recebeu o valor 2.

Intervenção 15: Balanceamento detectado e α recebeu o valor 4.

Intervenção 20: Balanceamento detectado e α recebeu o valor 8.

Intervenção 29: Balanceamento detectado e α recebeu o valor 16.

Intervenção 46: Balanceamento detectado e α recebeu o valor 32.

Intervenção 79: Desbalanceamento foi detectado e α recebeu o valor 1. Porém, os valores da matriz M eram todos 0.

Intervenção 81: Desbalanceamento foi detectado e α recebeu o valor 1. Porém, os valores da matriz M eram todos 0.

Intervenção 83: Desbalanceamento foi detectado e α recebeu o valor 1. Porém, os valores da matriz M eram todos 0.

Intervenção 85: Desbalanceamento foi detectado e α recebeu o valor 1. Porém, os valores da matriz M eram todos 0.

Intervenção 87: Desbalanceamento foi detectado e α recebeu o valor 1. Porém, os valores da matriz M são todos 0.

Intervenção 89: Desbalanceamento foi detectado e α recebeu o valor 1. Porém, os valores da matriz M são todos 0.

Intervenção 91: Desbalanceamento foi detectado e α recebeu o valor 1. Porém, os valores da matriz M são todos 0.

Intervenção 93: Desbalanceamento foi detectado e α recebeu o valor 1. A matriz M é gerada

M	0	1	2	...	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	...	0	0	0	0,03	0	0	0	0	0	0	0
1	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0

O primeiro valor selecionado era o que indicava a migração do VP_{20} para o P_0. O tempo na entrada da estratégia foi 1,37s. A fp concluiu que o próximo passo iria durar 1,26s. Então foi realizado uma tentativa de nova migração, não existiu uma tarefa com PM acima de 0. Nesta intervenção somente uma migração ocorreu.

Na próxima intervenção, 95 o balanceamento foi detectado e a próxima intervenção com valores de PM válidos ocorreu na intervenção 162.

M	0	1	2	...	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	...	0,29	0,29	0,28	0,27	0	0	0	0	0	0	0
4	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	...	0	0	0	0	0	0	0	0	0	0	0

O primeiro valor selecionado foi o que indicava a migração do VP_{18} para o P_3. O tempo na entrada da estratégia foi 1,67s. A fp concluiu que o próximo passo iria durar 1,21s. Então foi realizado uma tentativa de migração. O próximo valor selecionado foi o que indicava a migração do VP_{17} para o P_3. A fp concluiu que o próximo passo iria durar 1,18s. Na próxima tentativa, a fp retornou um valor acima do anteriormente encontrado, 1,33s. Nesta intervenção duas migrações ocorreram. O valor de α foi carregado com o valor 1.

Intervenção 164: Desbalanceamento detectado. Duas migrações foram efetuadas. Intervenção 166: Desbalanceamento detectado. Duas migrações foram efetuadas. Intervenção 168: Desbalanceamento detectado. Uma migração efetuada.

Da intervenção 170 até a 195 não ocorreram migrações por que estava balanceado, ou desbalanceado mas com matriz M com elementos nulos. Até o final da aplicação são 314 intervenções onde a maioria estavam em estado de equilíbrio ou o desbalanceamento não gerou PMs válidos.

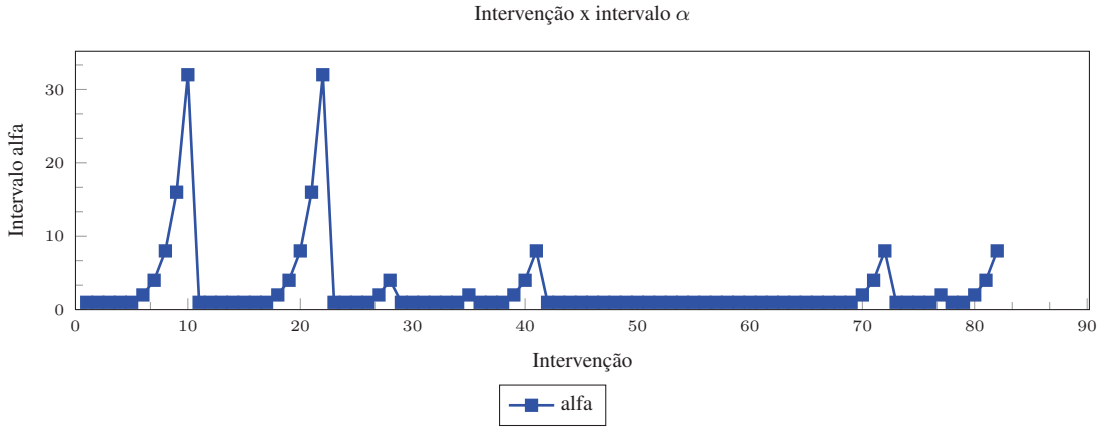
Com esta análise podemos apresentar a curva dos valores de α em relação a intervenção ocorrida. Neste gráfico, Figura 70, podemos deduzir que o equilíbrio foi alcançado na intervenção 5 e se manteve até a intervenção de número 10. Após este instante o sistema permaneceu desequilibrado até a intervenção 17 e se manteve até a 22ª em equilíbrio. Até o final da execução houve mais 13 intervenções em que o sistema esteve balanceado.

Como esperado a cada intervenção em que o equilíbrio foi encontrado, o valor de α dobrou. A cada momento de desbalanceamento o valor retornou a 1.

6.6 Discussão sobre os resultados encontrados

A partir da biblioteca AMPI foram desenvolvidas duas estratégias de balanceamento de carga. A primeira foi implementada seguindo o modelo de reescalonamento MigBSP e a segunda de acordo com o MigBSP++. Foi demonstrado que a estratégia baseada no MigBSP foi eficiente na máquina paralela utilizada, ou seja, apresentou uma redução de tempo de execução da aplicação, em cenários onde o número de processos era igual ao de processadores. Quando a quantidade de processos aumentava, o MigBSP++ apresentou melhores resultados.

Figura 70 – Controle de Intrusão de acordo com MigBSP++



Uma observação importante é aparente falta de necessidade de se indicar o parâmetro α para o MigBSP++. Na grande maioria dos casos onde a houve redução de tempo, o valor de α utilizado foi o menor possível.

Com a observação dos gráficos que mostraram os tempos dos super-passos, há indícios que o MigBSP++ realiza um reescalonamento mais apropriado utilizando melhor os recursos quando a quantidade de processos se eleva. Na análise passo-a-passo é possível ver a atuação do APBSP, que em algumas intervenções realiza seleção de até 3 processos para a migração baseado na simulação realizada.

7 CONSIDERAÇÕES FINAIS

Com o objetivo de obter o melhor desempenho das aplicações, a busca por uma melhor utilização dos recursos disponíveis é um desafio constante. Em particular, as aplicações paralelas geralmente possuem soluções específicas para o balanceamento de carga e, muitas vezes, não reaproveitáveis em outros ambientes. Esta dissertação apresenta um modelo de reescalonamento de processos para aplicações paralelas que tem a característica de ser independente da aplicação e da infraestrutura onde será utilizado.

O modelo de aplicações BSP, amplamente utilizado no desenvolvimento de aplicações paralelas, atualmente, tem sido alvo de pesquisas através de desenvolvimento de bibliotecas, *Runtime Systems* e modelos que utilizam recursos em nuvem para facilitar a implementação de aplicações. Uma iniciativa de migração de processos para aplicações escritas de acordo com o BSP, denominado **MigBSP**, vem sendo objeto de estudos na linha de pesquisa de Redes de Computadores e Sistemas Distribuídos do PIPCA da UNISINOS. O MigBSP é um modelo de reescalonamento de processos que realiza a combinação de múltiplas métricas na decisão de balanceamento de carga. Entretanto, após uma série de testes, verificou-se que duas características do MigBSP demonstram a necessidade de aperfeiçoamento. A primeira refere-se a detecção de desbalanceamento de carga. Em alguns casos, nos quais era necessário utilizar mais de um processo por processador, o MigBSP apresentou resultados equivocados, sinalizando falsos positivos, bem como, falsos negativos. A segunda característica, também explorada por esta dissertação, refere-se a decisão de quantos processos serão migrados de fato. Com o objetivo de resolver essas duas questões, apresenta-se modelo **MigBSP++**. O MigBSP++ é um modelo de reescalonamento que soluciona estas duas demandas, porém não substitui o MigBSP. Diferentemente do MigBSP, que é idealizado para grandes estruturas computacionais incluindo grades e aglomerados, o MigBSP++ possui uma abordagem centralizada e é direcionado apenas a aglomerados.

Para a primeira demanda, o MigBSP++ propôs a alteração do modo de detecção de desbalanceamento utilizada, observando-se o tempo total de computação de cada processador ao invés da análise individual dos processos. Através dele é possível realizar a detecção de desequilíbrio de modo adequado, mesmo que exista mais do que um processo por processador. Este trabalho demonstrou que o equilíbrio em aplicações BSP pode ser alcançado, ainda que os processos possuam tempos de execução diferentes.

O MigBSP++ soluciona a segunda questão utilizando o **Algoritmo de Predição BSP**. Após a seleção de processos propícios à migração, este algoritmo realiza uma série de simulações com os processos elencados pelo potencial de migração (PM). Ao seu término, esse algoritimo retorna a lista de processos que migrarão e produzirão um super-passo com tempo menor que o atual. Esta simulação só é possível devido ao conhecido conjunto de etapas das aplicações BSP.

A biblioteca AMPI que, na pesquisa realizada demonstrou-se mais próxima das características para a implementação do modelo em questão, foi a escolhida para os ensaios. Utilizando as

ferramentas do *Charm++*, nesta dissertação foram desenvolvidas duas estratégias de balanceamento de carga para a observação do comportamento do MigBSP e do MigBSP++. Apesar da impossibilidade de realizar uma implementação fiel dos modelos desejados devido a limitações técnicas, os resultados foram satisfatórios e sugerem que o MigBSP++ é capaz de realizar o balanceamento de carga adequado no ambiente utilizado.

7.1 Contribuições

O modelo apresentado neste trabalho baseia-se no MigBSP. Muitas das suas características são heranças mantidas ou que sofreram pequenas adaptações. As principais contribuições científicas que MigBSP++ apresenta são:

- Algoritmo de Predição BSP;
- Ajuste na detecção de desbalanceamento de carga;
- Adaptação para o cálculo da parcela *Comp()*;
- Alteração no comportamento de α.

O MigBSP, apesar de apontar as possíveis tarefas para a migração, não define quantas destas irão migrar de fato. Isso, até então, ficava a critério de uma entre duas heurísticas previstas. Entretanto, sabendo-se que as aplicações seguem o modelo BSP, um algoritmo foi elaborado com o objetivo de predizer quantas tarefas poderiam migrar sem prejuízo à aplicação. Denominado Algoritmo de Predição BSP, a partir dos ensaios realizados, apresentou resultados eficientes preenchendo a lacuna deixada pelo modelo anterior.

Quando mais de um processo está em execução em um processador, eles são executados de forma concorrente e não paralela. Este comportamento não permite que a análise proposta pelo MigBSP para detecção de desbalanceamento seja realizada adequadamente. Desta forma, o MigBSP++ foi modelado considerando este novo contexto. Assim, observou-se a necessidade de usar o tempo total do processador, ao invés de cada processo individualmente. Esta alteração trouxe automaticamente uma nova forma de calcular a parcela referente a computação (Comp) para a formação de PM. Outras otimizações também foram realizadas, melhorando o desempenho do modelo evitando cálculos desnecessários.

Outra observação que permitiu alterações foi o comportamento demorado nas intervenções quando o valor de α era muito elevado. A mudança realizada trouxe benefício ao MigBSP, pois a tentativa de balanceamento ficou mais intensa após a detecção do desequilíbrio entre os processadores. Além disso, os resultados dos ensaios deram indícios de que não há necessidade da passagem do parâmetro α por linha de comando. Na maioria dos casos, o menor valor de α produziu melhores resultados. Deste modo, a própria estratégia se encarregará de decidir quando intervir desde o início da aplicação.

7.2 Resultados

Foram realizados ensaios com duas aplicações científicas. Nem todos os resultados mostraram que a estratégia que utilizava o modelo MigBSP++ foi a melhor opção de balanceamento. Entretanto, nos casos onde o tempo de execução sem estratégia é alto e existem diversos superpassos, o MigBSP++ mostrou melhores resultados. Em situações onde o número de processos não eram superiores ao número de processadores, a estratégia MigBSP++ obteve um resultado insatisfatório e muitas vezes ficou com o tempo de execução acima das estratégias nativas do *Charm++*. Já o MigBSP obteve um resultado mais adequado. Nos casos em que a quantidade de tarefas era de até quatro vezes maior do que a de processadores, realizou-se uma distribuição de tarefas mais adequada com a utilização do MigBSP++ na maioria dos cenários.

Na maioria dos resultados em que o MigBSP++ obteve uma redução maior de tempo da aplicação, os parâmetros utilizados foram $\alpha = 2$ e D de 0,3. Esta observação sugere que a escolha de um parâmetro α inicial não seria mais necessária. Em relação ao valor de D, foi possível observar que. com os valores de 0,2 e 0,1. poucas vezes o balanceamento foi alcançado. Nestes casos, a estratégia detectou constantemente o desbalanceamento de carga e inseriu uma sobrecarga desnecessária a aplicação.

Em relação as estratégias nativas do Charm++, se desconsiderarmos os casos onde o modelo *MigBSP++* atuou com o número de tarefas igual a de processadores, a estratégia *RefineCommLB* foi a que mais se aproximou dos tempos de execução. Nos resultados onde a *MigBSP++* teve um pior desempenho, o tempo de execução foi de no máximo 7% maior. Porém, nos casos onde obteve vantagens, a média de execução alcançou o valor de até 37% mais rápido do que a melhor estratégia nativa utilizada nas comparações.

Esta estratégia foi concebida considerando que não há conhecimento da atividade que a aplicação paralela possui e/ou da infraestrutura na qual ela será utilizada. Para a utilização do MigBSP++ o usuário deve considerar apenas duas condições: se a aplicação é do tipo BSP e se a quantidade de tarefas é maior do que a quantidade de processadores. Com base nos resultados, o modelo *MigBSP++* apresentou-se como opção de estratégia viável. Porém, não existe nenhuma garantia de obtenção do resultado ótimo.

7.3 Trabalhos Futuros

Utilizando as informações contidas nesta dissertação, algumas ideias de pesquisas podem ser colocadas em prática de modo a enriquecer o MigBSP++.

- Abordagem Hierárquica – O MigBSP++, atualmente, utiliza uma abordagem centralizada. Para aperfeiçor este modelo, propõe-se o desenvolvimento de um modelo hierárquico seguindo os moldes do MigBSP, adicionando o ente gerentes que permitirá a aplicação em sistemas computacionais maiores

- Utilização de heurísticas mais sofisticadas – De modo eficiente, o MigBSP++, através do APBSP, indica os processos que devem migrar. Porém, utiliza-se um escalonamento inicial que não é responsabilidade do modelo. Assim, não há garantias de que o sistema está operando com o escalonamento inicial mais adequado. É possível propor métodos que realizem o mapeamento inicial de modo a encontrar uma distribuição melhor para as tarefas

- Conhecimento de quantos passos existem na aplicação – O MigBSP++ faz o reescalonamento considerando apenas os dados do super-passo atual. Ter o conhecimento da quantidade de super-passos necessários para a conclusão da aplicação, permite avaliar, de forma mais refinada, o custo computacional integrando os super-passos previstos para a descoberta do impacto total no tempo da aplicação. E assim, tomar decisões mais adequadas.

REFERÊNCIAS

ALAM, T.; RAZA, Z. A dynamic load balancing strategy with adaptive threshold based approach. In: PARALLEL DISTRIBUTED AND GRID COMPUTING (PDGC), 2012 2ND IEEE INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 927–932.

AMEDRO, B.; BAUDE, F.; CAROMEL, D.; DELBÉ, C.; FILALI, I.; HUET, F.; MATHIAS, E.; SMIRNOV, O. An Efficient Framework for Running Applications on Clusters, Grids, and Clouds. In: ANTONOPOULOS, N.; GILLAM, L. (Ed.). **Cloud Computing**. [S.l.]: Springer London, 2010. p. 163–178. (Computer Communications and Networks).

BADUEL, L.; BAUDE, F.; CAROMEL, D.; CONTES, A.; HUET, F.; MOREL, M.; QUILICI, R. Programming, Deploying, Composing, for the Grid. In: **Grid Computing**:software environments and tools. [S.l.]: Springer-Verlag, 2006.

BARRIVIERA, R. **Compressão Fractal de Imagens**. 2009. 1–251 p. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Paraná, 2009.

BARROSO, L. A.; DEAN, J.; HöLZLE, U. Web Search for a Planet: the google cluster architecture. **IEEE Micro**, Los Alamitos, CA, USA, v. 23, n. 2, p. 22–28, Mar. 2003.

BAUDE, F.; CAROMEL, D.; HUET, F.; MESTRE, L.; VAYSSIERE, J. Interactive and descriptor-based deployment of object-oriented grid applications. In: HIGH PERFORMANCE DISTRIBUTED COMPUTING, 2002. HPDC-11 2002. PROCEEDINGS. 11TH IEEE INTERNATIONAL SYMPOSIUM ON, 2002. **Anais...** [S.l.: s.n.], 2002. p. 93–102.

BHATELÉ, A.; BOHM, E.; KALÉ, L. V. A Case Study of Communication Optimizations on 3D Mesh Interconnects. In: INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING, 15., 2009, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2009. p. 1015–1028. (Euro-Par '09).

BISSELING, R. H. **Parallel Scientific Computation:a structured approach using bsp and mpi**. [S.l.]: Oxford University Press, 2004.

BONORDEN, O. Load Balancing in the Bulk-Synchronous-Parallel Setting using Process Migrations. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2007. IPDPS 2007. IEEE INTERNATIONAL, 2007. **Anais...** [S.l.: s.n.], 2007. p. 1–9.

BONORDEN, O.; JUURLINK, B.; OTTE, I. von; RIEPING, I. The Paderborn University BSP (PUB) library. **Parallel Comput.**, Amsterdam, The Netherlands, The Netherlands, v. 29, n. 2, p. 187–207, Feb. 2003.

CAROMEL, D.; KLAUSER, W.; VAYSSIèRE, J. Towards seamless computing and metacomputing in Java. **Concurrency: Practice and Experience**, [S.l.], v. 10, n. 11-13, p. 1043–1061, 1998.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v. 14, n. 2, p. 141–154, Feb. 1988.

CATALYUREK, U. V.; BOZDAG, D.; BOMAN, E. G.; DEVINE, K. D.; HEAPHY, R.; RIESEN, L. A. Hypergraph-Based Dynamic Partitioning and Load Balancing. In: **Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications**. [S.l.]: John Wiley & Sons, Inc., 2009.

CHUNG, H.-Y.; CHANG, C.-W.; HSIAO, H.-C.; CHAO, Y.-C. The Load Rebalancing Problem in Distributed File Systems. In: CLUSTER COMPUTING (CLUSTER), 2012 IEEE INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 117–125.

DA ROSA RIGHI, R.; PILLA, L. L.; MAILLARD, N.; CARISSIMI, A.; NAVAUX, P. O. A. Observing the Impact of Multiple Metrics and Runtime Adaptations on BSP Process Rescheduling. **Parallel Processing Letters**, [S.l.], v. 20, n. 02, p. 123–144, 2010.

DEVINE, K. D.; BOMAN, E. G.; HEAPHY, R. T.; HENDRICKSON, B. A.; TERESCO, J. D.; FAIK, J.; FLAHERTY, J. E.; GERVASIO, L. G. New challenges in dynamic load balancing. **Appl. Numer. Math.**, Amsterdam, The Netherlands, The Netherlands, v. 52, n. 2-3, p. 133–152, Feb. 2005.

DIAMOS, G.; WU, H.; WANG, J.; LELE, A.; YALAMANCHILI, S. Relational algorithms for multi-bulk-synchronous processors. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 18., 2013, New York, NY, USA. **Proceedings...** ACM, 2013. p. 301–302. (PPoPP '13).

DIAZ, J.; MUOZ-CARO, C.; NIO, A. A Fault Tolerant Adaptive Method for the Scheduling of Tasks in Dynamic Grids. In: ADVANCED ENGINEERING COMPUTING AND APPLICATIONS IN SCIENCES, 2009. ADVCOMP '09. THIRD INTERNATIONAL CONFERENCE ON, 2009. **Anais...** [S.l.: s.n.], 2009. p. 51–56.

DROST, N.; MAASSEN, J.; MEERSBERGEN, M. van; BAL, H.; PELUPESSY, F.; ZWART, S.; KLIPHUIS, M.; DIJKSTRA, H.; SEINSTRA, F. High-Performance Distributed Multi-Model / Multi-Kernel Simulations: a case-study in jungle computing. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM WORKSHOPS PHD FORUM (IPDPSW), 2012 IEEE 26TH INTERNATIONAL, 2012. **Anais...** [S.l.: s.n.], 2012. p. 150–162.

FAN, X.; WU, R. an; CHEN, P.; NING, Z.; LI, J. Parallel Computing of Large Eigenvalue Problems for Engineering Structures. In: FUTURE COMPUTER SCIENCES AND APPLICATION (ICFCSA), 2011 INTERNATIONAL CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 43–46.

FAN, Y.; WU, W.; CAO, H.; ZHU, H.; WEI, W.; ZHENG, P. LBVP: a load balance algorithm based on virtual partition in hadoop cluster. In: CLOUD COMPUTING CONGRESS (APCLOUDCC), 2012 IEEE ASIA PACIFIC, 2012. **Anais...** [S.l.: s.n.], 2012. p. 37–41.

FUNG, C.; CHOW, S. Y.; WONG, K. A low-cost parallel computing platform for power engineering applications. In: ADVANCES IN POWER SYSTEM CONTROL, OPERATION AND MANAGEMENT, 2000. APSCOM-00. 2000 INTERNATIONAL CONFERENCE ON, 2000. **Anais...** [S.l.: s.n.], 2000. v. 2, p. 354–358 vol.2.

GAVA, F.; FORTIN, J. Two Formal Semantics of a Subset of the Paderborn University BSPlib. In: PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING, 2009 17TH EUROMICRO INTERNATIONAL CONFERENCE ON, 2009. **Anais...** [S.l.: s.n.], 2009. n. 2, p. 44–51.

GERBESSIOTIS, A. V.; SINIOLAKIS, C. J. Merging on the BSP model. **Parallel Computing**, [S.l.], v. 27, n. 6, p. 809 – 822, 2001.

GHOSH, T. K.; GOSWAMI, R.; BERA, S.; BARMAN, S. Load balanced static grid scheduling using Max-Min heuristic. **2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing**, [S.l.], p. 419–423, Dec. 2012.

GIRAPH. Disponível em: <http://giraph.apache.org/>. Acesso em: jan. 2013.

GOMES, R. D. Q.; RIGHI, R. d. R. Implementação de um Balanceador de Carga para a Biblioteca AMPI Baseado no Modelo de Escalonamento MigBSP. In: XIII ERAD, 2013. **Anais...** [S.l.: s.n.], 2013. p. 19–20.

GRAEBIN, L. **Tratamento Flexível e Eficiente da Migração de Objetos Java em Aplicações Bulk Synchronous Parallel**. 2012. Dissertação (Mestrado em Ciência da Computação) — Universidade do Vale do Rio dos Sinos – UNISINOS, 2012.

GRAEBIN, L.; da Rosa Righi, R. jMigBSP:object migration and asynchronous one-sided communication for bsp applications. **2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies**, [S.l.], p. 35–38, Oct. 2011.

GRAEBIN, L.; RIGHI, R. D. R. Flexible Management on BSP Process Rescheduling:offering migration at middleware and application levels. **Journal of Applied Computing Research**, [S.l.], v. 1, n. 2, p. 84–94, Feb. 2012.

GUERREIRO, V.; RIGHI, R. d. R. Proposta de uma Heurística 3D para Seleção de Candidatos à Migração em Aplicações BSP. In: XIII ERAD, 2013. **Anais...** [S.l.: s.n.], 2013. p. 19–20.

HADOOP. Disponível em: <http://hadoop.apache.org/>. Acesso em: jan. 2013.

HAMA. Disponível em: <http://hama.apache.org/>. Acesso em: jan. 2013.

HARTUNG, S.; SHUKLA, H.; MILLER, J. P.; PENNYPACKER, C. GPU acceleration of image convolution using spatially-varying kernel. In: IMAGE PROCESSING (ICIP), 2012 19TH IEEE INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 1685–1688.

HILL, J. M.; MCCOLL, B.; STEFANESCU, D. C.; GOUDREAU, M. W.; LANG, K.; RAO, S. B.; SUEL, T.; TSANTILAS, T.; BISSELING, R. H. BSPlib: the bsp programming library. **Parallel Computing**, [S.l.], v. 24, n. 14, p. 1947 – 1980, 1998.

HUAN, Z.; QI-LONG, Z.; RUI, W. Estimation of BSP Network Parameters Based on MPI-2 One-Sided Operation. In: PARALLEL ARCHITECTURES, ALGORITHMS AND PROGRAMMING (PAAP), 2011 FOURTH INTERNATIONAL SYMPOSIUM ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 151–155.

HUANG, C.; ZHENG, G.; KALÉ, L.; KUMAR, S. Performance Evaluation of Adaptive MPI. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p. 12–21. (PPoPP '06).

HUET, F.; CAROMEL, D.; BAL, H. E. A High Performance Java Middleware with a Real Application. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2004., 2004, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2004. p. 2–. (SC '04).

JACKSON, D. J.; MAHMOUD, W. Parallel Pipelined Fractal Image Compression using Quadtree Recomposition. **The Computer Journal**, [S.l.], v. 39, n. 1, p. 1–13, 1996.

JETLEY, P.; GIOACHIN, F.; MENDES, C.; KALE, L.; QUINN, T. Massively parallel cosmological simulations with ChaNGa. In: PARALLEL AND DISTRIBUTED PROCESSING, 2008. IPDPS 2008. IEEE INTERNATIONAL SYMPOSIUM ON, 2008. **Anais...** [S.l.: s.n.], 2008. p. 1–12.

JETLEY, P.; WESOLOWSKI, L.; GIOACHIN, F.; KALÉ, L. V.; QUINN, T. R. Scaling Hierarchical N-body Simulations on GPU Clusters. In: ACM/IEEE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2010., 2010, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. (SC '10).

JIAO, X.; ZHENG, G.; ALEXANDER, P. A.; CAMPBELL, M. T.; LAWLOR, O. S.; NORRIS, J.; HASELBACHER, A.; HEATH, M. T. A system integration framework for coupled multiphysics simulations. **Engineering with Computers**, London, UK, v. 22, n. 3, p. 293–309, 2006.

JIAO, X.; ZHENG, G.; LAWLOR, O.; ALEXANDER, P.; CAMPBELL, M.; HEATH, M.; FIEDLER, R. An Integration Framework for Simulations of Solid Rocket Motors. In: AIAA/ASME/SAE/ASEE JOINT PROPULSION CONFERENCE, 41., 2005, Tucson, Arizona. **Anais...** [S.l.: s.n.], 2005.

KAJDANOWICZ, T.; INDYK, W.; KAZIENKO, P.; KUKUL, J. Comparison of the Efficiency of MapReduce and Bulk Synchronous Parallel Approaches to Large Network Processing. In: DATA MINING WORKSHOPS (ICDMW), 2012 IEEE 12TH INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 218–225.

KALE, L. V.; KRISHNAN, S. CHARM++: a portable concurrent object oriented system based on c++. **SIGPLAN Not.**, New York, NY, USA, v. 28, n. 10, p. 91–108, Oct. 1993.

KALE, L. V.; ZHENG, G. Charm++ and AMPI: adaptive runtime strategies via migratable objects. In: PARASHAR, M. (Ed.). **Advanced Computational Infrastructures for Parallel and Distributed Applications**. [S.l.]: Wiley-Interscience, 2009. p. 265–282.

KARYPIS, G.; KUMAR, V. **METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0**. [S.l.]: University of Minnesota, 1995.

KAUL, S.; SHUKLA, U.; SRIHARSHA, S.; SONKAR, V.; KANT, K. Robustness analysis of a new distributed scheduling approach - Relprox model using Mosix. In: NETWORKED COMPUTING AND ADVANCED INFORMATION MANAGEMENT (NCM), 2010 SIXTH INTERNATIONAL CONFERENCE ON, 2010. **Anais...** [S.l.: s.n.], 2010. p. 11–16.

KHAYYAT, Z.; AWARA, K.; ALONAZI, A.; JAMJOOM, H.; WILLIAMS, D.; KALNIS, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In: ACM EUROPEAN CONFERENCE ON COMPUTER SYSTEMS, 8., 2013, New York, NY, USA. **Proceedings...** ACM, 2013. p. 169–182. (EuroSys '13).

KRIZANC, D.; SAARIMAKI, A. Bulk Synchronous Parallel: practical experience with a model for parallel computing. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 1996., PROCEEDINGS OF THE 1996 CONFERENCE ON, 1996. **Anais...** [S.l.: s.n.], 1996. p. 208–217.

KRIZANC, D.; SAARIMAKI, A. Bulk synchronous parallel: practical experience with a model for parallel computing. **Parallel Computing**, [S.l.], v. 25, n. 2, p. 159 – 181, 1999.

LI, H.; FOX, G.; QIU, J. Performance Model for Parallel Matrix Multiplication with Dryad: dataflow graph runtime. In: CLOUD AND GREEN COMPUTING (CGC), 2012 SECOND INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 675–683.

LIU, W.; SCHMIDT, B. Parallel Pattern-Based Systems for Computational Biology: a case study. **Parallel and Distributed Systems, IEEE Transactions on**, [S.l.], v. 17, n. 8, p. 750–763, 2006.

LIU, X.; TONG, W.; HOU, Y. BSPCloud: a programming model for cloud computing. In: COMPUTER AND INFORMATION TECHNOLOGY (CIT), 2012 IEEE 12TH INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 1109–1113.

LMBENCH benchmark. Disponível em: <http://www.gelato.unsw.edu.au/IA64wiki/lmbench3>. Acesso em: jun. 2013.

LOWE, D.; OROU, N. Interdependence of booking and queuing in remote laboratory scheduling. In: REMOTE ENGINEERING AND VIRTUAL INSTRUMENTATION (REV), 2012 9TH INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 1–6.

MALEWICZ, G.; AUSTERN, M. H.; BIK, A. J.; DEHNERT, J. C.; HORN, I.; LEISER, N.; CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2010., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p. 135–146. (SIGMOD '10).

MATEOS, C.; ZUNINO, A.; CAMPO, M. On the evaluation of gridification effort and runtime aspects of JGRIM applications. **Future Generation Computer Systems**, [S.l.], v. 26, n. 6, p. 797–819, 2010.

MCHEICK, H.; MOHAMMED, Z. R.; LAKISS, A. Evaluation of Load Balance Algorithms. **2011 Ninth International Conference on Software Engineering Research, Management and Applications**, [S.l.], p. 104–109, Aug. 2011.

ME2DAY. Disponível em: <http://me2day.net/>. Acesso em: mai. 2013.

MIAO, W.; TONG, W. Agent based ServiceBSP Model with Superstep Service for Grid Computing. In: GRID AND COOPERATIVE COMPUTING, 2007. GCC 2007. SIXTH INTERNATIONAL CONFERENCE ON, 2007. **Anais...** [S.l.: s.n.], 2007. p. 255–260.

MILOJICIC, D. S.; DOUGLIS, F.; PAINDAVEINE, Y.; WHEELER, R.; ZHOU, S. Process migration. **ACM Comput. Surv.**, New York, NY, USA, v. 32, n. 3, p. 241–299, Sept. 2000.

MING, L.; GUANG-HUI, X.; LI-FA, W.; YAO, J. Performance Research on MapReduce Programming Model. In: INSTRUMENTATION, MEASUREMENT, COMPUTER, COMMUNICATION AND CONTROL, 2011 FIRST INTERNATIONAL CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 204–207.

MIZAN. Disponível em: <http://thegraphsblog.wordpress.com/the-graph-blog/mizan/>. Acesso em: jun. 2013.

MPI Standard. Disponível em: <http://www.mcs.anl.gov/research/projects/mpi/standard.html>. Acesso em: jan. 2013.

MULTICOREBSP for C. Disponível em: <http://www.multicorebsp.com/>. Acesso em: jun. 2013.

PACHECO, P. **An Introduction to Parallel Programming**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

PARASHAR, M.; LI, X. Introduction: enabling large-scale computational science – motivations, requirements, and challenges. In: **Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications**. [S.l.]: John Wiley & Sons, Inc., 2009. p. 1–7.

PELLEGRINI, F.; ROMAN, J. Scotch: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: LIDDELL, H.; COLBROOK, A.; HERTZBERGER, B.; SLOOT, P. (Ed.). **High-Performance Computing and Networking**. [S.l.]: Springer Berlin Heidelberg, 1996. p. 493–498. (Lecture Notes in Computer Science, v. 1067).

PHILLIPS, J. C.; BRAUN, R.; WANG, W.; GUMBART, J.; TAJKHORSHID, E.; VILLA, E.; CHIPOT, C.; SKEEL, R. D.; KALé, L.; SCHULTEN, K. Scalable molecular dynamics with NAMD. **Journal of computational chemistry**, [S.l.], v. 26, n. 16, p. 1781–802, Dec. 2005.

PILLA, L.; NAVAUX, P.; RIBEIRO, C.; COUCHENEY, P.; BROQUEDIS, F.; GAUJAL, B.; MEHAUT, J. Asymptotically Optimal Load Balancing for Hierarchical Multi-Core Systems. In: PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS), 2012 IEEE 18TH INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 236–243.

PILLA, L.; RIBEIRO, C.; CORDEIRO, D.; MEI, C.; BHATELE, A.; NAVAUX, P.; BROQUEDIS, F.; MEHAUT, J.; KALE, L. A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems. In: PARALLEL PROCESSING (ICPP), 2012 41ST INTERNATIONAL CONFERENCE ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 118–127.

QUADROS GOMES, R. de; GUERREIRO, V.; ROSA RIGHI, R. da; SILVEIRA JR., L. G. da; YANG, J. Analyzing Performance of the Parallel-based Fractal Image Compression Problem on Multicore Systems. **AASRI Procedia**, [S.l.], v. 5, n. 0, p. 140 – 146, 2013. 2013 AASRI Conference on Parallel and Distributed Computing and Systems.

RIGHI, R.; CARISSIMI, A.; NAVAUX, P.; HEISS, H. **MigBSP: nova abordagem para reescalonamento de processos em aplicações bsp**. Caxias do Sul: [s.n.], 2009.

RIGHI, R. d. R. **MigBSP:a new approach for processes rescheduling management on bulk synchronous parallel applications**. 2009. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul – UFRGS, 2009. (October).

RIGHI, R. d. R.; PILLA, L.; CARISSIMI, A. Process rescheduling: enabling performance by applying multiple metrics and efficient adaptations. In: **Future Manufacturing Systems**. [S.l.: s.n.], 2008. p. 39–64.

RODRIGUES, E.; NAVAUX, P.; PANETTA, J.; MENDES, C.; KALE, L. Optimizing an MPI weather forecasting model via processor virtualization. In: HIGH PERFORMANCE COMPUTING (HIPC), 2010 INTERNATIONAL CONFERENCE ON, 2010. **Anais...** [S.l.: s.n.], 2010. p. 1–10.

RODRIGUES, E. R. **Dynamic load-balancing: a new strategy for weather forecast models**. 2011. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul – UFRGS, 2011. (September).

RODRIGUES, E. R.; NAVAUX, P. O. A.; PANETTA, J.; FAZENDA, A.; MENDES, C. L.; KALE, L. V. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), 22., 2010, Itaipava, Brazil. **Proceedings...** [S.l.: s.n.], 2010.

RODRIGUES, E. R.; NAVAUX, P. O.; PANETTA, J.; MENDES, C. L. Preserving the original MPI semantics in a virtualized processor environment. **Science of Computer Programming**, [S.l.], v. 78, n. 4, p. 412–421, Apr. 2013.

ROSA RIGHI, R. da; PILLA, L.; CARISSIMI, A.; NAVAUX, P.; HEISS, H.-U. MigBSP: a novel migration model for bulk-synchronous parallel processes rescheduling. In: HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, 2009. HPCC '09. 11TH IEEE INTERNATIONAL CONFERENCE ON, 2009. **Anais...** [S.l.: s.n.], 2009. p. 585–590.

SANDHYA, K.; RAJU, G. Single System Image clustering using Kerrighed. In: ADVANCED COMPUTING (ICOAC), 2011 THIRD INTERNATIONAL CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 260–264.

SAROOD, O.; KALE, L. V. A 'Cool' Load Balancer for Parallel Applications. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2011., 2011, New York, NY, USA. **Proceedings...** ACM, 2011. p. 21:1–21:11. (SC '11).

SCHERSON, I. D.; SEN, S.; MA, Y. Two nearly optimal sorting algorithms for mesh-connected processor arrays using shear-sort. **Journal of Parallel and Distributed Computing**, [S.l.], v. 6, n. 1, p. 151 – 165, 1989.

SCHERSON, I.; SEN, S. Parallel sorting in two-dimensional VLSI models of computation. **Computers, IEEE Transactions on**, [S.l.], v. 38, n. 2, p. 238–249, 1989.

SEMAR SHAHUL, A. Z.; SINNEN, O. Scheduling task graphs optimally with A*. **The Journal of Supercomputing**, [S.l.], v. 51, n. 3, p. 310–332, Mar. 2010.

SEO, S.; YOON, E. J.; KIM, J.; JIN, S.; KIM, J.-S.; MAENG, S. HAMA: an efficient matrix computation with the mapreduce framework. **2010 IEEE Second International Conference on Cloud Computing Technology and Science**, [S.l.], p. 721–726, Nov. 2010.

SINNEN, O. **Task Scheduling for Parallel Systems**. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2007. (Wiley Series on Parallel and Distributed Computing).

SKILLICORN, D. B.; HILL, J. M. D.; MCCOLL, W. F. Questions and Answers About BSP. **Journal of Scientific Programming**, [S.l.], v. 6, n. November, 1997.

SQUYRES, J. M. Open MPI. In: **The Archicture of Open Source Applications**. [S.l.]: Self published, 2012. v. ii.

SRIVATSA, M.; KAWADIA, V.; YANG, S. Distributed graph query processing in dynamic networks. In: IEEE INTERNATIONAL CONFERENCE ON PERVASIVE COMPUTING AND COMMUNICATIONS WORKSHOPS (PERCOM WORKSHOPS), 2012., 2012. **Anais...** [S.l.: s.n.], 2012. p. 20–25.

TANAKA, M.; TATEBE, O. Workflow Scheduling to Minimize Data Movement Using Multi-constraint Graph Partitioning. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING (CCGRID 2012), 2012., 2012, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2012. p. 65–72. (CCGRID '12).

TANENBAUM, A. S. **Computer Networks**. 4. ed. Upper Saddle River: Prentice Hall PTR, 2003. 892 p.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 3. ed. São Paulo, SP, BRA: Prentice-Hall, Inc., 2011.

TANENBAUM, A. S.; STEEN, M. v. **Distributed Systems Principles and Paradigms**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

TANG, S.; YU, C.; SUN, J.; LEE, B.-S.; ZHANG, T.; XU, Z.; WU, H. EasyPDP: an efficient parallel dynamic programming runtime system for computational biology. **Parallel and Distributed Systems, IEEE Transactions on**, [S.l.], v. 23, n. 5, p. 862–872, 2012.

TING, I.-H.; LIN, C.-H.; WANG, C.-S. Constructing a Cloud Computing Based Social Networks Data Warehousing and Analyzing System. In: INTERNATIONAL CONFERENCE ON ADVANCES IN SOCIAL NETWORKS ANALYSIS AND MINING, 2011., 2011, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2011. p. 735–740. (ASONAM '11).

UNIVERSITY OF ILLINOIS. **Adaptive MPI Manual**. 2013. Disponível em: <http://charm.cs.illinois.edu/manuals/pdf/ampi.pdf>. Acesso em: ago. 2012.

UNIVERSITY OF ILLINOIS. **The Charm++ Parallel Programming System Manual**. 2013. Disponível em: <http://charm.cs.illinois.edu/manuals/pdf/charm++.pdf>. Acesso em: ago. 2013.

VALIANT, L. G. A Bridging Model for Parallel Computation. **Commun. ACM**, New York, NY, USA, v. 33, n. 8, p. 103–111, Aug. 1990.

WHITFIELD, T. W.; MARTYNA, G. J. A unified formalism for many-body polarization and dispersion: the quantum drude model applied to fluid xenon. **Chemical Physics Letters**, [S.l.], v. 424, p. 409 – 413, 2006.

WILKINSON, B.; MICHAEL, A. **Parallel Programming: techniques and applications using networked workstations and parallel computers**. 2. ed. [S.l.]: Upper Saddle River, NJ: Prentice Hall, 2005.

YAMIN, A. Escalonamento em Sistemas Paralelos e Distribuídos. **Escola Regional de Alto Desempenho**, [S.l.], p. 75–126, 2001.

YZELMAN, A.; BISSELING, R. H. An object-oriented bulk synchronous parallel library for multicore programming. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, v. 24, n. 5, p. 533–553, Apr. 2012.

ZHANG, J.; GE, S. A Parallel Algorithm to Find Overlapping Community Structure in Directed and Weighted Complex Networks. **2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control**, [S.l.], p. 1561–1564, Dec. 2012.

ZHENG, G. **Achieving high performance on extremely large parallel machines**: performance prediction and load balancing. 2005. Tese (Doutorado em Ciência da Computação) — University of Illinois at Urbana-Champaign, 2005.

ZHENG, G.; KALE, L.; LAWLOR, O. Multiple flows of control in migratable parallel programs. In: PARALLEL PROCESSING WORKSHOPS, 2006. ICPP 2006 WORKSHOPS. 2006 INTERNATIONAL CONFERENCE ON, 2006. **Anais...** [S.l.: s.n.], 2006. p. 10 pp.–444.