UNISINOS

Programa Interdisciplinar de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Vladimir Magalhães Guerreiro

MigCube e MigHull:

HEURÍSTICAS PARA SELEÇÃO AUTOMÁTICA DE PROCESSOS PARA MIGRAÇÃO EM APLICAÇÕES BSP

São Leopoldo, 2014

Vladimir Magalhães Guerreiro

MIGCUBE E MIGHULL:
Heurísticas para Seleção Automática de Processos para Migração em Aplicações BSP

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre pelo Programa 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

G934m

Guerreiro, Vladimir Magalhães

MigCube e MigHull: Heurísticas para Seleção Automática de Processos para Migração em Aplicações BSP / Vladimir Magalhães Guerreiro. – 2014.

116 f.: il.,: 30cm.

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

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

1. Computação. 2. Processamento paralelo (Computadores).
3. Sistemas de computação em grade. I. Título. CDU 004

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecário: Flávio Nunes — CRB 10/1298)

Para os meus pais, Zerilton e Isaltina,
pelo incentivo, educação
e motivação, apesar da distância.

Para os meus irmãos, Roberto e Sabrina,
pela paciência e
por compensar nossos pais pela distância.

"*Educação para a Independência do Pensamento*".
(Albert Einstein)

RESUMO

Em ambientes paralelos, uma das alternativas para tratar o dinamismo, tanto em nível de infraestrutura quanto de aplicação é o uso de migração, principalmente em aplicações que executam em fases utilizando BSP (*Bulk Synchronous Parallel*). Neste contexto, o modelo de reescalonamento **MigBSP** foi desenvolvido para tratar da realocação de processos em aplicações paralelas. Assim como o modelo BSP, ele considera as três fases de execução de uma superetapa: (i) computação local, (ii) comunicação global e (iii) uma barreira de sincronização; coletando dados localmente durante a computação para efetuar o cálculo do Potencial de Migração (PM) do processo. Com o PM e parâmetros adicionais fornecidos no inicio da execução da aplicação, o MigBSP tem condições de escolher processos candidatos a migração em uma aplicação paralela executando em um ambiente distribuído. Entretanto, as duas heurísticas possíveis de serem utilizadas hoje, dependem de informações fornecidas pelo usuário e/ou podem não selecionar uma quantidade eficiente de processos no momento do reescalonamento, podendo ser necessário várias chamadas para balancear o ambiente. Desta forma, esta dissertação apresenta duas novas heurísticas, **MigCube** e **MigHull**. Elas utilizam o MigBSP e efetuam a seleção automática de processos candidatos à migração sem a interferência do programador. As informações fornecidas pelo MigBSP são utilizadas nas heurísticas, a combinação das três métricas mensurados, posicionadas em um plano tridimensional, define cada processo como um ponto no espaço que possui as coordenadas x, y e z, onde cada eixo representa uma métrica para tomada de decisão. A heurística MigCube monta um cubo a partir das médias das distâncias entre os pontos, utilizando o processo com o maior PM como centro do cubo. A heurística MigHull segue a definição da Envoltória Convexa, tentando envolver todos os pontos, porém utilizando duas adaptações que se fazem necessárias para a aplicação neste trabalho. O MigBSP foi desenvolvido no simulador SimGrid, e este segue sendo utilizado para a criação das duas heurísticas apresentadas nesta dissertação. Nos testes realizados neste simulador, foi possível verificar um ganho de até 45% no tempo de execução da aplicação utilizando a heurística MigHull, e até 42% utilizando a MigCube, quando comparado a aplicação sem o modelo de migração. Porém, em simulações com um maior número de processos, este ganho tende a cair, já que um dos maiores problemas do BSP e aplicações que executam em grades é o tempo de sincronização de tarefas, ou seja, quanto mais processos, maior a necessidade de sincronização, e mesmo o balanceamento dos processos acaba tendo um resultado prejudicado.

Palavras-chave: Processamento Paralelo. MigBSP. Grades Computacionais. SimGrid. Migração de Processos. Balanceamento de Carga. *Bulk Synchronous Parallel*. Envoltória Convexa.

ABSTRACT

In a parallel environment, one of the alternatives to address the dynamism, both at the infrastructure and application levels, is the use of migration, mostly with applications that execute in steps using BSP (Bulk Synchronous Parallel). In this context, the rescheduling model **MigBSP** was developed to deal with processes reallocation in parallel applications. As BSP model, MigBSP uses the three steps of a superstep: (i) computation, (ii) communication and (iii) a synchronization barrier; collecting local data during the computation step, to compute the processes' Potential of Migration (PM). With the PM and additional parameters provided in the beginning of the application's execution, MigBSP have conditions to choose the processes candidate to migrate in a parallel application running in a distributed system. However, the two heuristics possible to be used today depend of information provided by the user and/or may not select the proper quantity of processes in the rescheduling moment, being necessary many executions to balance the environment. This way, this dissertation present two new heuristics, **MigCube** and **MigHull**. They make use of MigBSP, and automatically will choose the processes to migrate without user interference. The information provided by MigBSP are used in the heuristics, the combination of the three measured metrics, positioned in a three-dimensional space, defines each process as a point in space and has the coordinates x, y e z, where each axis represents a metric for decision making. The MigCube heuristic build a cube from the average of the distances between points, using the process with the highest PM as the center of the cube. The MigHull follows the definition of a Convex Hull, trying to involve all points, but using two adaptations that are necessary to implement this work. The MigBSP was developed using SimGrid simulator, and it keeps being used to creation of the two heuristics presented in this dissertation. In the conducted tests in this simulator, was possible to achieve a gain of until 45% on application execution time using MigHull, and until 42% using MigCube, when compared with the application without the migration model. However, simulations with a bigger number of processes, this gain tends to fall, since one of the bigger problems of BSP and applications that run in grid is the time of tasks synchronization, that is, as more processes, more need of synchronization, and even the processes balancing ends up having an impaired outcome.

Keywords: Parallel Processing. MigBSP. Grid. SimGrid. Process Migration. Load Balance. Bulk Synchronous Parallel. Convex Hull.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS

dist	Distância
exec	Execução/Execute
Comm	*Comunication*/Comunicação
Comp	*Computation*/Computação
Infos	Informações
Max	Máximo
Memm	Memory/Memória
Min	Mínimo
recv	*receive*/Receber
std dev	*Standard Deviation*/Desvio Padrão
Sup	Suporte

LISTA DE SIGLAS

AMPI	*Adaptive Message Passing Interface*
API	*Application Programming Interface*
BSP	*Bulk Synchronous Parallel*
CAPES	Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
DSM	*Distributed Shared Memory*
FFT	Transformada Rápida de Fourier
FSF	*Free Software Fundation*
GNU	*GNU's Not Unix!*
GRAS	*Grid Reality And Simulation*
GPU	*Graphics Processing Unit*
HPFS	*Hadoop File System*
LGPL	*GNU Library General Public License*
MIMD	*Multiple Instruction, Multiple data*
MISD	*Multiple Instruction, Single Data*
MSG	*Simple Programming Environment*
NUMA	*Non-Uniform Memory Access*
PID	*Process Identifier*
PM	Potencial de Migração
RAM	*Random Access Memmory*
SAN	*Storage Area Network*
SIMD	*Single Instruction, Multiple Data*
SimDag	*Programming Environment for DAG Applications*
SimIX	*POSIX-like API*
SISD	*Single Instruction, Single Data*
SMPI	*Programming Environment for the Simulation of MPI Application*
SMURF	*XimIX Network Proxy*
SO	Sistema Operacional
SSH	*Secure Shell*
SURF	*SimGrid Internal Kernel Simulator*
UMA	*Uniform Memory Access*
VM	*Virtual Machine*/Máquina Virtual

VPN *Virtual Private Network*

XBT *SimGrid ToolBox Core*

SUMÁRIO

1 INTRODUÇÃO

"A persistência é o menor caminho do êxito."
Charles Chaplin

Nos dias atuais, a utilização de redes de computadores se faz cada vez mais presente em nosso cotidiano. Desde a sua criação, a conexão entre computadores tem auxiliado as mais diversas áreas científicas na comunicação, troca de informações e dados (MENDES, 2007). Atualmente, em ambientes de alto desempenho e de troca de grandes volumes de informação tem se utilizado sistemas distribuídos conectados por redes de diversas capacidades e inúmeros tipos de recursos, com o objetivo de resolver problemas de propósito geral (TONG; ZHU, 2010). Esta realidade representa um desafio e ao mesmo tempo um grande avanço para os recursos computacionais atuais.

Este crescimento permite à área de computação grandes avanços na pesquisa e desenvolvimento de novas soluções para suprir e também auxiliar as necessidades das demais áreas científicas (BERNABE; PLAZA, 2011). A distribuição de atividades, desde as tarefas mais cotidianas, é uma forma que se observa com um melhor rendimento no processamento, ou seja, utilizando o processamento paralelo, distribuindo partes de um problema maior em pequenos problemas para serem computados simultaneamente permitindo obter um resultado final em um menor tempo e proporcionalmente um menor custo (tempo em que apenas um processador iria efetuar todo o processamento). Atualmente o processamento paralelo é utilizado nos mais diversos meios, desde nossos celulares e computadores pessoais que permitem a execução de tarefas em mais de um processador simultaneamente (EPPSTEIN; GALIL, 1988), até super computadores, projetados para trabalharem com processamento de alto desempenho. Nesta linha de processamento de alto desempenho, diversas áreas se beneficiam do processamento paralelo, sendo considerada uma ferramenta eficaz para a busca de soluções, tais como: Bancos de dados não relacionais (NAKAGAWA; NAGAMI, 2012), mineração de dados, processamento de imagens de grandes tamanhos (KRISHNAKUMAR et al., 2011; LIU; WU; MARSAGLIA, 2012), entre outros.

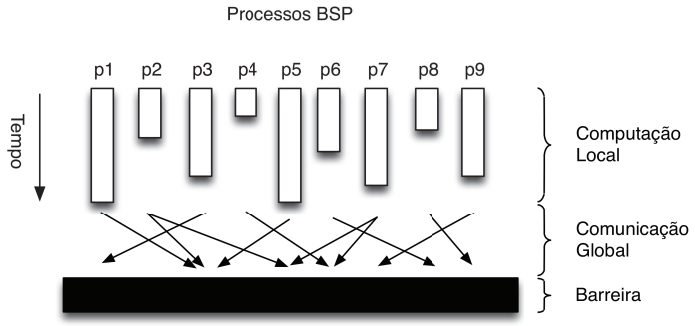
A busca pela evolução dos recursos computacionais para um maior poder de processamento tem sido um assunto discutido já há algum tempo. Conforme Heart et al. (1973) mostrou, uma grande evolução foi a utilização de multiprocessadores, permitindo que dados sejam processados simultaneamente obtendo um ganho proporcional ao número de processadores. Porém, vemos que mesmo trabalhando com muitos processadores, nem sempre conseguiremos obter o resultado esperado, utilizando um número de recursos consideravelmente limitado (KIYARAZM; MOEINZADEH; SHARIFIAN-R, 2011). Ao encontro deste problema, uma área que tem sido estudada há um longo tempo é a de Computação em Grade (*Grid Computing*) (RAJ; FIONA, 2013), que pode ser descrita como um conjunto de aglomerados de computadores,

que podem estar em diversos locais, mas conectados por uma rede (que pode ser pública ou privada) trabalhando para alcançar um objetivo comum (RATHORE; CHANA, 2011; SURI; SINGH, 2010).

Grades normalmente são construídas utilizando um grande número de recursos computacionais heterogêneos, com um poder de processamento e comunicação muito variável, mas que se sobressaem pela grande quantidade de nodos de processamento (computadores/nodos). Uma outra característica é o fato de serem, em geral, dispersas geograficamente em nível mundial e interconectadas através da Internet, sendo considerado uma rede de baixo acoplamento. Também são conhecidos como um grande conjunto de recursos computacionais com uma rede de comunicação que em geral não deve ser considerada para processamento conjunto. Cada nodo deve ser capaz de armazenar dados necessários para um processamento completo de uma tarefa e após a conclusão desta tarefa, comunicar com outros nodos para conclusão de todo o trabalho (PATNI et al., 2011). Grades são consideradas como uma máquina paralela para a computação de aplicações paralelas.

Um dos modelos de programação mais utilizados na computação paralela, e principalmente em aglomerados, é o BSP (*Bulk Synchronous Parallel*)(VALIANT, 1993), pois ele pode ser visto como um modelo de programação paralela onde os programas são organizados em superetapas (*supersteps*), cada qual dividida em três fases distintas, conforme mostra a Figura 1: (i) computação local (*Local Computation*), (ii) comunicação global (*Global Communication*) e (iii) uma barreira de sincronização (*synchronization Barrier*). Porém, um dos grandes problemas do BSP em grades é a sincronização, que geralmente é bastante custosa.

Figura 1 – Exemplo de uma superetapa ou *superstep*, utilizando o modelo BSP.



Fonte: Adaptado de (RIGHI et al., 2009).

Ambientes de grade devem ser projetados para que os processos executando em cada nodo, possam efetuar um processamento com a menor quantidade de comunicação com processos em outros nodos (RATHORE; CHANA, 2011; SURI; SINGH, 2010). Porém está é uma questão complexa em grades: como organizar processos de tal forma a se obter o melhor resultado na execução de toda a grade onde executem diversos processos com tamanhos e tempos variáveis. Considere por exemplo, a ocorrência de um processo em um nodo necessitar de um maior tempo

de processamento do que outros dois processos em outro nodo. Para resolver este problema, devemos utilizar o método de Balanceamento de Carga (DEVINE et al., 2005). Este método é utilizado para equilibrar cargas em ambientes distribuídos, em grades este método possibilita a melhor distribuição de processos entre nodos do *Cluster* (Aglomerado de Computadores) (TONG; ZHU, 2010; AGGARWAL; MOTWANI; ZHU, 2006). Por exemplo, em um *Cluster* com três nodos, onde a execução de processos em um nodo leva mais tempo do que os outros dois, deve-se avaliar o tamanho do processo e/ou a quantidade de processos executando no nodo. Caso existam dois processos neste nodo e apenas um em cada um dos outros nodos, deve-se considerar a migração de um processo do nodo sobrecarregado para outro menos carregado. Outra forma é redistribuir estes processos de acordo com a capacidade de processamento de cada nodo.

A Migração de Processos (RIGHI; GRAEBIN; COSTA, 2014) permite uma melhor distribuição e consequentemente uma melhor utilização dos recursos de grade tendo como resultado uma aplicação distribuída com uma melhor utilização da grade (RAJ; FIONA, 2013). A migração de processos utiliza quatro perguntas para a definição de um processo a ser migrado (RIGHI et al., 2009):

- Como: Define o mecanismo que será utilizado para efetuar a migração do processo;
- Quando: Em que momento, na execução da aplicação, o processo será migrado;
- Quem: Qual processo deve ser migrado;
- Onde: A partir do nodo atual do processo, para qual nodo devo migrar o processo definido.

O modelo de reescalonamento de processos definido por Righi, Graebin e Costa (2014), conhecido como MigBSP, utiliza a migração de processos e o modelo BSP para fornecer um método de balanceamento de carga em aglomerados. O MigBSP é um modelo que utiliza a execução de *supersteps* em cada nodo do aglomerado para definir o processo a ser migrado e para qual nodo será migrado. Ele coleta informações durante a computação local na execução dos processos, onde cada um é responsável por incrementar um índice de Potencial de Migração (PM). Após a computação, durante a comunicação global, cada processo troca informações com processos de outros aglomerados. Durante a barreira de sincronização, os processos gerentes trocam informações entre todos os gerentes dos aglomerados locais da grade. Antes de uma nova execução da *superstep*, o MigBSP efetua a migração do processo que possuir o maior PM, mas para isto ele verifica o custo da execução do processo no nodo atual e o custo no possível nodo destino. Caso o custo no nodo destino seja o menor, ele migra o processo e inicia uma nova *superstep*.

O MigBSP é um modelo que oferece grandes vantagens e recursos para migração de processos com o objetivo de reduzir o tempo de processamento de uma aplicação paralela. Porém,

a forma como as suas heurísticas atuais são calculadas o tornam limitado, é necessário um conhecimento prévio da arquitetura do ambiente e da aplicação que está sendo executada, um melhor resultado é bastante dependente da parametrização inicial da aplicação. Sendo assim, o desafio desta dissertação é oferecer heurísticas que superem estas limitações, afim de oferecer um modelo que automaticamente e com qualidade selecione processos para migração, de forma a reduzir ao máximo o tempo de execução da aplicação paralela.

Sendo assim, é papel da área de Processamento e Alto Desempenho fornecer soluções que permitam uma utilização de forma prática, ágil e segura de ambientes distribuídos.

1.1 Definição do Problema e Objetivos

Um das principais características em ambientes distribuídos é o balanceamento de recursos como processamento, memória, entre outros. Uma das alternativas para tratar o dinamismo, tanto em nível de infraestrutura quanto de aplicação, é o uso de migração. Em particular, o modelo de reescalonamento MigBSP (RIGHI; GRAEBIN; COSTA, 2014) foi desenvolvido utilizando o simulador SimGrid, com o objetivo de tratar a realocação de processos em aplicações que executam em fases do tipo BSP.

O MigBSP propõe uma arquitetura hierárquica de grade, na qual cada site (*Cluster* ou rede local) possui um gerente particular. Durante a execução de uma *superstep*, cada processo coleta dados localmente, calcula o Potencial de Migração (PM) para cada um dos processos e repassa o maior valor para o seu gerente no momento da barreira. O Potencial de Migração é um índice, calculado durante a fase de Computação Local da *superstep*, ele utiliza as métricas de Computação, Comunicação e Memória do processo no nodo em que ele está sendo executado e itera o índice do processo. Os gerentes trocam informações do PM dos processos sob suas jurisdições e a escolha dos candidatos é feita através de uma entre duas heurísticas: (i) escolha do processo com maior valor de PM; (ii) escolha dos processos que possuem PM maior do que Max(PM)$\times$**p**, onde **p** é uma percentagem definida na inicialização da aplicação. Testes com MigBSP mostram que a segunda abordagem é mais eficiente, uma vez que se tem a chance de reorganizar os processos para processadores de forma mais rápida e usufruir de um maior número de *supersteps* com um bom escalonamento (RIGHI et al., 2009).

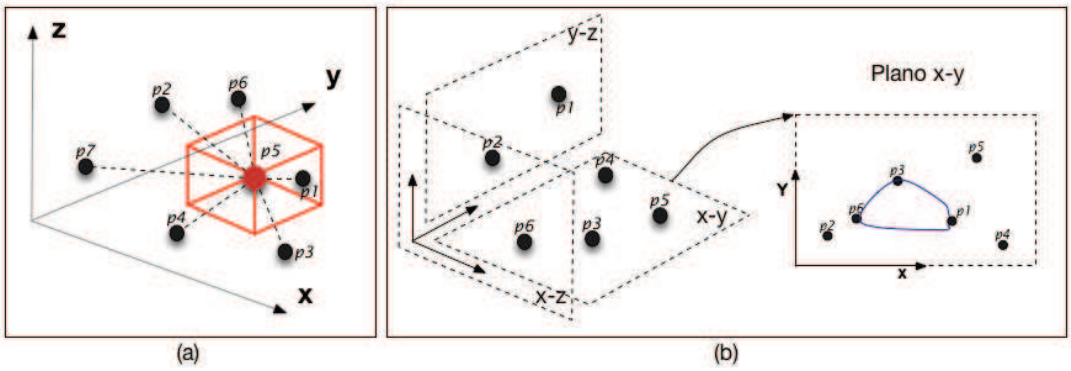
Mesmo com esta avaliação de cada processo, a heurística atual pode não selecionar uma quantidade eficiente de processos no momento do reescalonamento. Por exemplo, poderiam ser necessárias várias chamadas de reescalonamento para migrar processos de um aglomerado lento para outro rápido ao invés de transferi-los em uma única oportunidade. Para uma determinada aplicação, uma das métricas (computação, comunicação ou memória) pode ter uma relevância diferente do que para outro tipo de aplicação que utiliza o modelo MigBSP. Entretanto, para se obter uma maior assertividade na escolha de processos para migração, será necessário uma maior intrusividade durante a execução, ou seja, será necessário coletar mais informações dos processos e nodos executando na grade.

O MigBSP permite a utilização do Potencial de Migração, que oferece um método para seleção de processos para a migração em ambiente de grade. Este trabalho propõe o desenvolvimento de duas heurísticas, que utilizando o MigBSP, se encarregará de otimizar a quantidade de processos candidatos à migração sem a interferência do programador. Uma vez que ela é dada pela combinação de três componentes mensurados num espaço tridimensional, são apresentadas aqui as duas heurísticas para seleção de processos, MigCube e MigHull. A Figura 2 ilustra a ideia básica do funcionamento de cada uma. Cada processo é definido como um ponto no espaço e possui as coordenadas x, y e z. A heurísticas MigCube (a) trabalha no espaço tridimensional, efetuando cálculos em cima dos dados fornecidos. A MigHull (b) aplica duas adaptações em cima dos dados, transformando em três planos bidimensionais, onde a Envoltória Convexa é aplicada (REZENDE; STOLFI, 1994; KARAVELAS; TZANAKI, 2011).

O foco desta dissertação está em desenvolver heurísticas eficientes para a seleção automática de processos, que melhor utilizem as métricas (Computação, Comunicação e Memória) coletadas durante a execução de cada *superstep*, e também o PM, que é obtido pela combinação delas. Estas métricas podem ser representadas pelas coordenadas x, y e z, e posicionadas em um espaço tridimensional, servindo como um método de análise do ambiente, a fim de avaliar o impacto de um determinado processo *i* em um nodo *j*. Desta forma, é possível avaliar que uma aplicação que se caracteriza por bastante processamento (CPU *Bound*), e neste caso, é esperado ter a métrica computação elevada, mas sem impactar no tempo total de execução da aplicação.

- **Sentença do Problema:** *Dado uma aplicação BSP utilizando o modelo MigBSP, o desafio de pesquisa consiste em planejar como podem ser feitas heurísticas eficientes, tanto em tempo computacional quanto na qualidade dos resultados para a seleção automática de processos no MigBSP, considerando a combinação das três métricas atualmente suportadas pelo Potencial de Migração.*

Figura 2 – Representação dos processos com as duas Heurísticas no Espaço Tridimensional: (a) MigCube: o cubo de seleção de processos, (b) MigHull: a separação do espaço tridimensional em três planos, onde a envoltória adaptada é executada.



Fonte: Elaborado pelo Autor.

As heurísticas aqui apresentadas, trabalham de forma automática, sem a intervenção do usuário. Avaliando a arquitetura distribuída em uso e selecionando processos que ao serem migrados no ambiente de grade, permitam que a aplicação execute no menor tempo possível, efetuando uma utilização balanceada dos recursos computacionais disponíveis. Elas utilizam o MigBSP como base, e tem como objetivo prover funcionalidades automáticas para uma melhor seleção de processos. Desta forma, por se tratarem de modelos, podem ser aplicadas a outras aplicações e situações que envolvam o reescalonamento de processos em aplicações BSP.

Uma vez que os processos estão posicionados, as heurísticas gerenciam os processos candidatos à migração. Depois, é necessário avaliar o nodo destino para evitar que o processo a ser migrado nesta fase seja um candidato à migração na próxima.

1.2 Organização do Trabalho

Esta dissertação está estruturado de forma a permitir um entendimento prático e continuo deste trabalho. No capítulo 2 está detalhada a fundamentação teórica deste trabalho, definições sobre os processos de Escalonamento, Migração de Processos, Balanceamento de Carga em Aplicações Paralelas, Envoltória Convexa e Espaços Tridimensionais. Já o capítulo 3 apresenta os trabalhos que servem de base para esta pesquisa, além de um detalhamento sobre o modelo de reescalonamento de processos MigBSP, no final é apresentado um comparativo mostrando o impacto nas heurísticas apresentadas. Em seguida, no capítulo 4 é apresentada a estrutura das heurísticas, bem como o funcionamento de seus modelos. Adiante é apresentado o capítulo 5, onde são avaliadas as heurísticas, definidas sua implementação e comparadas com as heurísticas previamente utilizadas. Por fim, no capítulo 6 são apresentadas as conclusões desta dissertação, quais as contribuições e trabalhos futuros.

2 FUNDAMENTAÇÃO TÉORICA

Este capítulo apresenta os temas necessários e relevantes para o desenvolvimento deste trabalho, bem como os conceitos e definições dos assuntos abordados. Inicialmente é detalhado o conceito de Computação Paralela, bem como os modelos de Escalonamento, Balanceamento de Carga, Migração de Processos e BSP. Por fim é apresentada uma descrição do simulador de grades SimGrid, utilizado como ferramenta de modelagem deste trabalho.

2.1 Computação Paralela

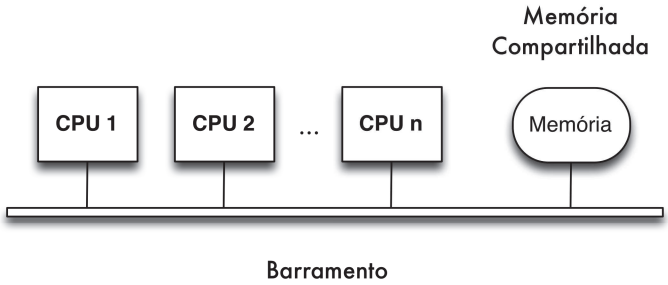
Computação Paralela é uma forma de computação que utiliza o paralelismo de tarefas para um maior aproveitamento de recursos e um melhor desempenho na execução de uma atividade. Inicialmente os programas eram desenvolvidos de forma sequencial, ou seja, uma tarefa era executada após a outra em um único processador, de modo que para aumentar o desempenho da tarefa era necessário aumentar o *clock* do processador. Entretanto, apenas aumentar a velocidade do processador não se mostrava viável, pois estava se atingindo limites físicos da arquitetura de processadores. Desta forma, a indústria encontrou uma nova solução, o paralelismo. A computação paralela tem se tornado dominante no paradigma de arquitetura de computadores, ela define um sistema computacional com múltiplos processadores em um computador ou múltiplos computadores interconectados (TANENBAUM; STEEN, 2007).

A ideia do paralelismo é permitir que *n* processadores trabalhando simultaneamente possam efetuar o processamento de *n* tarefas paralelamente. Porém, para obter o melhor resultado do paralelismo é necessário, além do ambiente computacional, que o programa a ser executado seja desenvolvido pensando nesta arquitetura paralela. As tarefas precisam ser divididas de forma a que possam ser processadas simultaneamente sem ou com a menor dependência de dados possível (DROZDOWSKI, 2010; SINNEN, 2007). Estas definições dificultam o desenvolvimento de aplicações paralelas (YAMIN, 2001). Desta forma, em geral ela é mais utilizada em atividades que demandem um grande processamento, como: previsões meteorológicas, pesquisas médicas, estudos de DNA, simulações de larga escala, entre outros problemas que demandem uma processamento de grandes volumes de dados. O processamento de tarefas paralelamente pode demandar outros recursos, como a sincronização de tarefas, espera para comunicação e barreira para sincronização de dados (TANENBAUM; STEEN, 2007; PACHECO, 2011). Além dos requisitos básicos no desenvolvimento de aplicações paralelas, elas devem ser moldadas de acordo com o ambiente computacional que será utilizado, podendo ser: Multiprocessador (2.1.1) e Multicomputador (2.1.2), conforme são detalhados a seguir.

2.1.1 Multiprocessador

Um sistema em que dois ou mais processadores compartilham acesso a uma memória RAM (*Random Access Memmory*) são chamados de **Multiprocessador de Memória Compartilhada** ou apenas Multiprocessador. Este sistema permite que um programa paralelo possa ser desenvolvido utilizando o mesmo endereçamento de memória local. Conforme mostra a Figura 3, todos os processadores se comunicam pelo mesmo barramento com uma unidade de memória compartilhada. Entretanto, existe a possibilidade de um processador escrever um dado e quando for lê-lo posteriormente, ele estar diferente, pois o outro processador pode já ter utilizado aquele endereço. Uma forma de eliminar esta restrição é organizar a memória de forma que cada processador possa escrever apenas em endereços específicos para cada um, mas possa ler toda ela, outra forma é tratar esta situação em nível de aplicação (TANENBAUM, 2011).

Figura 3 – Representação do Sistema Multiprocessador com Memória Compartilhada.



Fonte: Adaptado de (TANENBAUM, 2011).

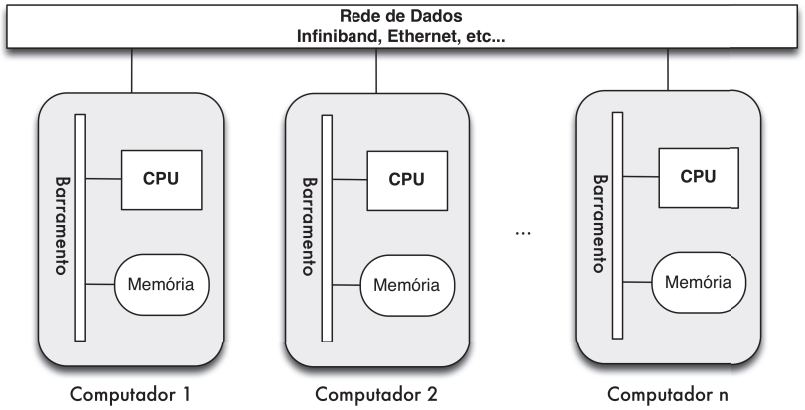
De forma geral, Multiprocessadores utilizam a arquitetura de memória UMA (*Uniform Memory Access*), ou seja, todos os processadores tem acesso uniforme à memória. Existe também a arquitetura de memórias NUMA (*Non-Uniform Memory Access*), onde existem blocos de memória separados, em que um processador está conectado diretamente pelo barramento a um bloco, mas o outro processador não. Estes processadores são conectados por uma rede de interconexão de alta velocidade. Um programa feito para UMA funciona em uma arquitetura NUMA, entretanto o tempo de acesso de um processador à memória de outro será maior. A arquitetura NUMA é utilizada em casos onde são necessários muitos processadores.

2.1.2 Multicomputador

Multiprocessadores são práticos e atrativos pois oferecem um meio de comunicação simples. Entretanto, multiprocessadores de grande porte são difíceis e caros de se montar. Por isso, muito tem se pesquisado sobre Multicomputadores. Dois ou mais computadores conectados através de uma rede (*Ethernet*, *Infiniband*, entre outras) com baixo acoplamento, mas com um processador com sua própria memória conectado diretamente um ao outro pelo mesmo barra-

mento, estes sistemas são conhecidos como Multicomputadores. Eles permitem a conexão de inúmeros computadores com configurações de *Hardware* heterogênea, sendo o único requisito básico uma interface de rede. Conforme mostra a Figura 4, cada computador possui sua CPU e memória com acesso local através do mesmo barramento em uma arquitetura UMA, e cada computador é conectado através de uma rede de baixo acoplamento (TANENBAUM, 2011).

Figura 4 – Representação do Sistema Multicomputador com Troca de Mensagens.



Fonte: Adaptado de (TANENBAUM, 2011).

A forma de comunicação destes sistemas é a troca de mensagens, um processador de um computador não pode gravar diretamente na memória de outro computador. Por isso, no programa a ser desenvolvido, a troca de informações precisa ser efetuada através de troca de mensagens. A troca de mensagens é funcional e tem sido bem utilizada. Entretanto, para sistemas Multicomputadores existe a arquitetura DSM (*Distributed Shared Memory*), uma memória distribuída compartilhada, em que o programa em execução enxerga a memória como se fosse local, entretanto o Sistema Operacional se encarrega de acessar o endereço de memória solicitado em outro computador ou no local que ele esteja (TANENBAUM, 2011; TANENBAUM; STEEN, 2007).

Com a atual arquitetura dos computadores, é possível utilizar sistemas Híbridos, onde cada computador de um Multicomputador é na verdade um Multiprocessador com mais de um processador e memória local.

2.1.3 Taxonomia de Flynn

Existem diversas maneiras de classificar computadores, entretanto, uma das mais conhecidas é a Taxonomia de Flynn (FLYNN, 1966). Ela se baseia no número de instruções concorrentes e a quantidade de fluxos de dados disponíveis para classificar a arquitetura. Ela foi criada há bastante tempo, porém é valida até hoje, e segue sendo bastante controversa, principalmente no modelo MISD (*Multiple Instruction, Single Data*). A Tabela 1 representa a Taxonomia de Flynn em forma de matriz.

Tabela 1 – Representação da Taxonomia de Flynn

	Dados Únicos	**Dados Múltiplos**
Instruções Únicas	**SISD**	**SIMD**
Instruções Múltiplas	**MISD**	**MIMD**

Fonte: Adaptado de (FLYNN, 1966).

Conforme a Tabela 1 com a Taxonomia de Flynn, segue o descritivo de cada uma das quatro classificações (FLYNN, 1966).

- **SISD** (*Single Instruction, Single Data*): Modelo de computador serial, uma instrução trabalhando sobre um fluxo de dados sequenciais. É a mais antiga e segue sendo utilizada atualmente principalmente em computadores pessoais;
- **SIMD** (*Single Instruction, Multiple Data*): Modelo em que uma instrução é aplicada sobre múltiplos fluxos de dados, são considerados um tipo de computador paralelo, pois são capazes de efetuar uma simples instrução sobre um grande volume de dados. As GPU's são consideradas SIMD;
- **MISD** (*Multiple Instruction, Single Data*): Neste modelo várias instruções atuam sobre um mesmo fluxo de dados, é uma classificação que gera muita discussão, pois é complexo a comprovação de sua utilização. Sistemas de tolerância falha são eventualmente considerados nesta classificação;
- **MIMD** (*Multiple Instruction, Multiple Data*): Este modelo utiliza múltiplas instruções sobre múltiplos fluxos de dados, são considerados um modelo de computação paralela, sendo amplamente utilizado em diversos ambientes computacionais. Eles permitem a execução de diferentes instruções sobre diferentes fluxos de dados.

Esta classificação auxilia na definição de ambientes computacionais baseado na arquitetura utilizada. Dos quatro modelos, a Taxonomia mais utilizada na computação paralela é a *MIMD*, ela nos permite executar inúmeros processos ao mesmo tempo em um computador. Porém para isto é necessário um escalonamento destas tarefas, conforme a seção a seguir (TANENBAUM; STEEN, 2007; TANENBAUM, 2011; PACHECO, 2011).

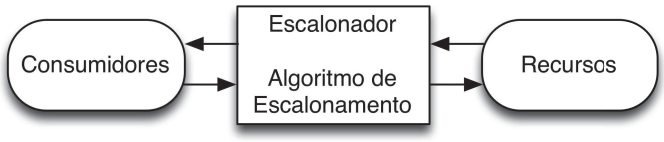
2.2 Escalonamento

Em um computador, em geral existem múltiplos processos e *threads* que competem pela CPU ao mesmo tempo, cabendo a um componente do Sistema Operacional efetuar a escolha de quem deverá utilizar a CPU (TANENBAUM, 2011). O escalonamento contempla um conjunto de recursos e outro de consumidores. A parte do Sistema Operacional que efetua a escolha de

qual processo/*thread* deve utilizar a CPU é o **Escalonador**, e ele utiliza o **Algoritmo de Escalonamento**, conforme mostra a Figura 5 (YAMIN, 2001; TANENBAUM, 2011; DROZDOWSKI, 2010; SINNEN, 2007).

Considerando as restrições de recursos e consumidores, Yamin (2001) define que o problema do escalonamento consiste em encontrar uma política/algoritmo eficiente para gerenciar o uso dos recursos pelos consumidores. Uma política de escalonamento normalmente é avaliada por duas características: (i) desempenho e (ii) eficiência, ou seja, deve-se avaliar tanto o resultado do escalonamento efetuado quanto o custo para executar o escalonamento. Se duas políticas produzem escalonamentos iguais, a de menor custo computacional deve ser utilizada (YAMIN, 2001).

Figura 5 – Estrutura de escalonamento.



Fonte: Adaptado de (YAMIN, 2001).

O Escalonamento é um problema clássico que está presente em diversas áreas do cotidiano (sistema de produção em industriais, serviço de atendimento ao usuário, etc.). Entretanto, um ponto importante no escalonamento de tarefas é avaliar se as sub-tarefas são dependentes das outras ou necessitam de algum tipo de sincronização. É necessário efetuar um mapeamento das tarefas, desta forma, se considera o escalonamento uma função que associa cada tarefa a uma data de início e um local de execução, conforme a equação (2.1), onde: *exec(T)* representa o tempo de execução da tarefa T, *data(T)* o momento que inicia a tarefa *T* e *com(T,T')* o custo de comunicação (YAMIN, 2001).

$$data(T') \geq data(T) + exec(T) + com(T, T'). \tag{2.1}$$

Normalmente, quando as tarefas tem a mesma alocação (*aloc(T)* = *aloc(T')*), o custo de comunicação é nulo (*com(T,T')* = 0). O tempo total de execução de uma aplicação paralela é determinado na equação (2.2).

$$C = Max(data(T) + exec(T)). \tag{2.2}$$

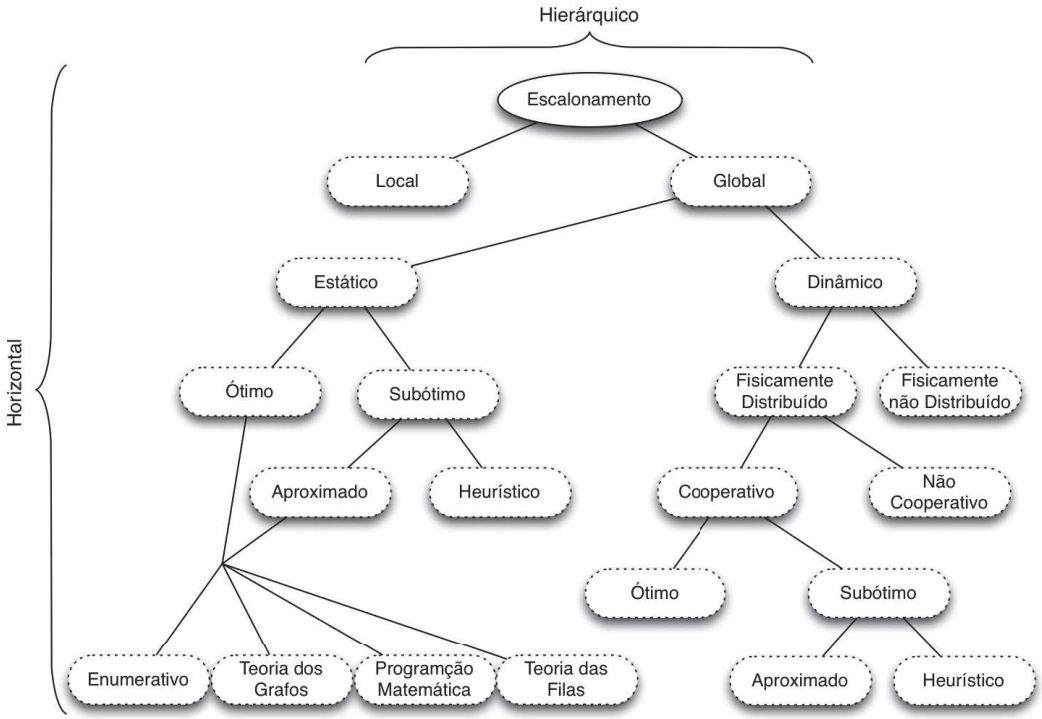
O problema do escalonamento será eficiente quando se obtiver um menor *C* possível. Entretanto, encontrar um *C* mínimo é difícil e custoso, com uma complexidade da classe NP-Difícil (DROZDOWSKI, 2010; SINNEN, 2007).

Para evitar o uso de terminologias e conceitos contraditórios, algumas Taxonomias foram definidas, a Taxonomia de Casavant e Kuhl é a mais utilizada.

2.2.1 Taxonomia de Casavant e Kuhl

A Taxonomia definida por Casavant e Kuhl (1988) é uma das mais conhecidas, ela se baseia na natureza do algoritmo de escalonamento utilizado. Ela possui duas abordagens, a (i) hierárquica e (ii) horizontal, conforme mostra a Figura 6 com suas características. Na abordagem hierárquica, é definida uma estrutura em árvore para a classificação das características de escalonamento, já a abordagem horizontal utiliza os ramos da hierárquica para sua classificação (GRAEBIN; RIGHI, 2012; YAMIN, 2001).

Figura 6 – Características da Taxonomia de Casavant.



Fonte: Adaptado de (CASAVANT; KUHL, 1988).

Com base na Figura 6, fica mais evidente a definição das duas abordagens que a Taxonomia define, conforme sua descrição a seguir.

2.2.1.1 Abordagem Hierárquica

A abordagem Hierárquica inicialmente é dividida em (i) Local e (ii) Global, Local se refere ao compartilhamento de tempo quando múltiplos processos executam em um único processador; no Global é necessário decidir qual o local (processador) que determinado processo será executado. O escalonamento Global é dividido em (i) Estático e (ii) Dinâmico. No escalonamento Estático as informações sobre a arquitetura, as tarefas e dados a serem processados

são conhecidos previamente e todas as decisões de escalonamento são tomadas antes da execução, é uma abordagem simples de ser implementada. Entretanto ela não considera que as cargas e tarefas possam mudar ao longo da execução e é necessário conhecer todo o ambiente computacional a ser utilizado. Nos casos em que estas duas restrições sejam aceitas, pode ser considerado um escalonamento (i) ótimo.

Nos demais casos, o escalonamento será (ii) subótimo, de forma que possa ser até mesmo inviável sua execução, mesmo assim ele permite duas abordagens, a (i) Aproximada e (ii) Heurística. A Aproximada utiliza a mesma estrutura do Ótimo, mas considera apenas as possíveis soluções ideais, ele termina quando encontra uma solução considerada boa; a Heurística utiliza parâmetros de ideais que impliquem no comportamento do sistema com a expectativa de que estes procedimentos melhorem o resultado geral do escalonamento.

Por outro lado, o escalonamento Dinâmico trabalha com a ideia de baixo conhecimento sobre a aplicação no ambiente disponível, decisões são tomadas apenas durante a execução da aplicação. Ele trabalha com a abordagem de processadores locais, (i) fisicamente não distribuídos e também (ii) fisicamente distribuídos onde, neste caso o escalonamento pode ser (i) Não Cooperativo, tal que os processadores tomam decisões independentes e (ii) Cooperativo, tendo este método a possibilidade de soluções (i) ótimas e (ii) subótimas. Seguindo a mesma ideia do escalonamento Estático, as soluções Subótimas utilizam a abordagem (i) Aproximada e (ii) Heurístico (SINNEN, 2007; YAMIN, 2001; GRAEBIN; RIGHI, 2012).

2.2.1.2 Abordagem Horizontal

As definições da abordagem Horizontal se encaixam em todos os ramos da abordagem Hierárquica. O escalonamento pode ser definido como (i) Adaptativo e (ii) Não Adaptativo. Na forma Adaptativa os algoritmos e parâmetros utilizados para implementar a política de escalonamento mudam dinamicamente de acordo com o comportamento do sistema; já a Não Adaptativa não se altera pelo comportamento do sistema.

A abordagem Horizontal utiliza uma política de escalonamento com balanceamento de carga, em que os processos são persistidos em todos os computadores de um sistema distribuído. Para isto, as informações sobre a utilização dos nodos é compartilhada pela rede periodicamente, ou sob demanda, para todos os nodos que estejam no nível global do sistema. Desta forma, todos cooperam para melhor distribuir as tarefas no ambiente distribuído.

Normalmente, quando um novo processo é criado, ou quando ele é iniciado, o escalonamento de processos efetua o mapeamento inicial deste processo para um recurso específico. O escalonamento pode ser estendido para redistribuir um processo durante a execução, desta forma efetuando um reescalonamento, redistribuindo os processos que já estão atribuídos a um processador, a um outro menos utilizado (SINNEN, 2007; YAMIN, 2001; GRAEBIN; RIGHI, 2012).

O escalonamento é um problema que vem a tempos sendo tratado para um melhor resultado

em ambientes paralelos, utilizando as técnicas conhecidas, empregadas de acordo com o ambiente e solução propostas. Juntamente com um método de Balanceamento de Carga, permitem uma utilização com um alto desempenho de um ambiente computacional.

2.3 Balanceamento de Carga

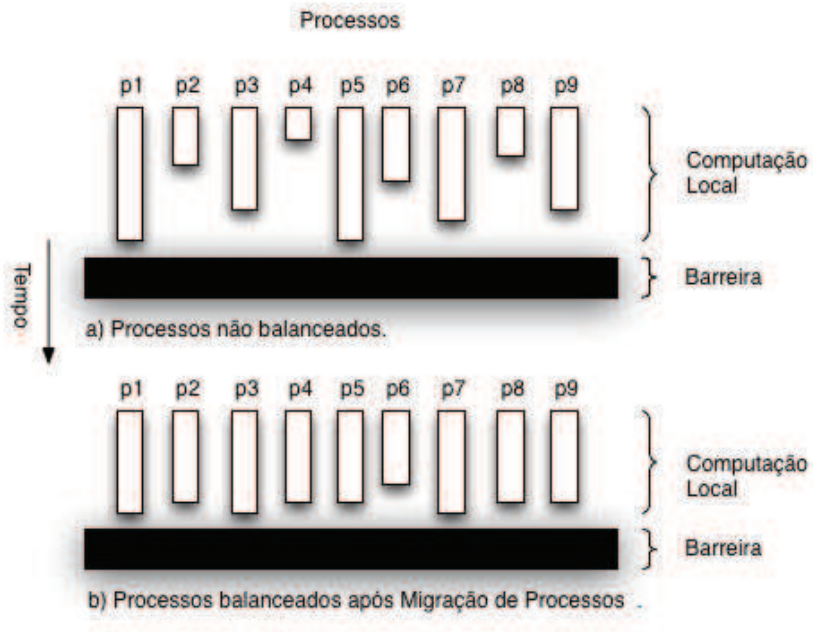
A ideia básica da política de Balanceamento de Carga é fazer com que os processos utilizem os recursos disponíveis de forma homogênea. Para isto, é necessário que as informações sobre a carga nos processadores do ambiente sejam compartilhadas através da rede de forma periódica ou sob demanda. Para que assim todos os computadores (nodos) tenham uma visão do estado global do sistema. Desta forma, todos os nodos cooperam com o objetivo de transferir a carga dos processadores com maior utilização para os menos carregados. Quando os nodos são homogêneos, esta estratégia é mais viável. Deve-se sempre levar em consideração que quanto maior a intrusividade no ambiente para coletar informações para o balanceamento de carga, maior será o impacto na aplicação (YAMIN, 2001).

Para se obter um bom desempenho em sistemas computacionais paralelos e distribuídos, é necessário balancear a carga em cada um dos recursos. Uma fraca distribuição de tarefas, baixo nível de paralelismo e desigualdade no tempo de processamento das tarefas, são alguns itens que desequilibram o sistema, reduzindo o desempenho global (NAVAUX, 2011; GRAEBIN; RIGHI, 2012). O balanceamento de carga possui algumas políticas que auxiliam a distribuição de carga em processos: (i) escolha da quantidade de trabalho a ser enviada para um processo, (ii) mover a carga de um processo mais carregado para outro com menor carga, (ii) escolher o processador para iniciar uma nova carga e (iv) decidir quais processos podem ser migrados no ambiente. Balanceamento de carga tem um maior impacto em aplicações que utilizam sincronização, pois a cada barreira para sincronizar é possível redistribuir as tarefas de acordo com a carga coletada nas últimas execuções, conforme mostra a Figura 7 (RIGHI et al., 2009).

O balanceamento de carga tenta, analisando o volume de dados, distribuir de forma igualitária a quantidade de trabalho entre os diversos processadores do ambiente distribuído (CASAVANT; KUHL, 1988). Isto ocorre pois em toda execução existe uma considerável quantidade de poder de processamento que não está sendo utilizada eficientemente. Algumas métricas foram definidas para auxiliar a identificar processos e processadores com maior e menor utilização (RATHORE; CHANA, 2011; SURI; SINGH, 2010; EL KABBANY et al., 2011): (i) quantidade de tarefas na fila, (ii) carga média nos processadores, (iii) utilização dos processadores em determinado momento, (iv) percentual de processamento disponível, (v) quantidade de memória livre e o (vi) volume de comunicação entre processos. Utilizando estas métricas, de forma a que satisfaçam o ambiente em questão, é possível efetuar um balanceamento de carga que resulte em um bom desempenho.

O balanceamento de carga pode utilizar algoritmos: (i) estático, onde as tarefas são distribuídas para os computadores do sistema distribuído considerando a carga de trabalho durante o

Figura 7 – Exemplo de Processos (a) sem balanceamento de carga e (b) após balanceamento de carga.



Fonte: Adaptado de (RIGHI et al., 2009).

lançamento da aplicação onde, a vantagem neste caso é a simplicidade na execução e o baixo nível de intrusividade, inexistente neste caso. Porém este sistema acaba sendo comprometido em ambientes que possuam muita variação na carga de trabalho; (ii) dinâmico, onde as tarefas são alteradas e mapeadas durante a execução do programa, sendo necessário um constante monitoramento do ambiente para que ao atingir um valor pré-definido de desequilíbrio, os processos sejam redistribuídos (EL KABBANY et al., 2011).

Segundo Righi et al. (2009), um ambiente em desequilíbrio pode ser detectado de forma (i) síncrona, ocorre de forma natural para a maioria das aplicações, durante as barreias, no momento da sincronização o balanceamento de carga pode ser acionado, analisando o ambiente e efetuando o reescalonamento dos processos; já na forma (ii) assíncrona, existe um processo ou tarefa dedicado a analisar e acompanhar o histórico de carga do sistema, ele então faz a solicitação de reescalonamento. Um balanceador de carga dinâmico precisa responder às questões, pois a resposta a elas mostra que tipo de balanceador está sendo utilizado (ZHANG et al., 2005):

- Quem decide quando o balanceador de carga atua?
- Quais informações são utilizadas para atuação do balanceador de carga?
- Onde o balanceador de carga decide a atuação?

A resposta a “Quem” decide quando o balanceador de carga atua é respondida baseada na política de quem inicializa a migração, *sender-initiated* e *receiver-initiated*, ou seja, o *sender*

são os nodos mais carregados solicitando o balanceamento para aliviar sua carga e o *receiver* são os nodos com menor carga solicitando o balanceamento para receberem mais carga.

Já, a resposta à "Quais" remete às abordagens (i) Global e (ii) Local. Quando Global, todas as informações do sistema distribuído serão utilizadas para efetuar o rebalanceamento; no caso Local, apenas as informações do conjunto local são utilizadas para a redistribuição dos processos.

Por fim, para responder à "Onde", são definidas as características (i) Centralizado e (ii) Distribuído. No primeiro, existe um único nodo mestre que recebe informações de todo o ambiente, com base nas informações que ele recebe, ele decide quem deve migrar; no caso Distribuído, cada aglomerado local recebe os dados de todo o ambiente, desta forma todos os processos possuem as informações referente a todos os outros processos (RIGHI et al., 2009; ZHANG et al., 2005; ZHANG; KOELBEL; COOPER, 2009).

O Balanceamento de Carga é basicamente um conjunto de políticas e métricas que servem para definir quais processos, em qual momento e para onde um processo deve ser migrado. Na seção a seguir serão explanadas as técnicas para migração de processos.

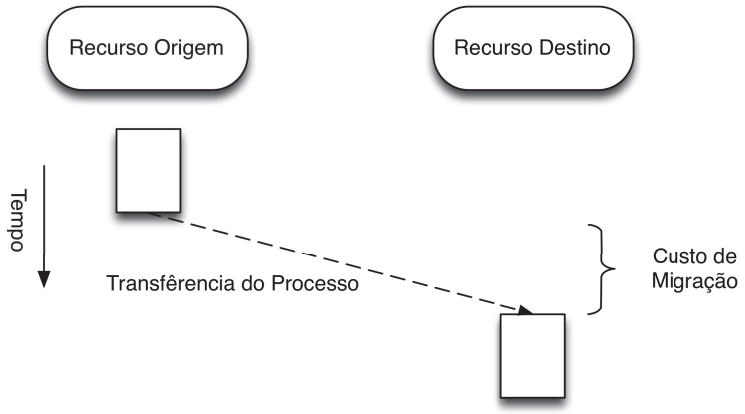
2.4 Migração de Processos

Migração de Processos é o movimento de um processo em execução para um outro processador. Um processo é uma abstração do Sistema Operacional e representa uma instância de um programa (YAMIN, 2001). A migração na verdade executa uma ilusão de movimento do processo, pois na verdade ele encerra o processo em execução em um processador e inicia uma nova execução em outro processador. Porém com os dados e informações no exato momento em que ele foi encerrado no processador anterior, conforme mostra a Figura 8. Entretanto, a migração de processos está atrelada ao custo de migração. Dependendo da quantidade de informações ou volume de dados que um processo utilize, por vezes o custo de migração pode se sobressair à vantagem de sua migração. Esta é uma análise que deve ser efetuada durante a execução do balanceamento de carga (MILOJICIC et al., 2000; RIGHI et al., 2009).

A migração de processos tem como vantagem aproveitar mudanças de *hardware* ou em outros pontos do sistema distribuído. Pois ela consegue executar praticamente sem apresentar impacto ao usuário e ter um melhor resultado, caso a alteração de *hardware* seja uma ampliação do mesmo (AGGARWAL; MOTWANI; ZHU, 2006). De forma geral, a migração de processos pode ser amplamente aplicada as seguintes áreas (ZHANG et al., 2005; RIGHI et al., 2009): (i) redistribuição dinâmica de cargas, (ii) processamento de grandes volumes de dados, (iii) sistemas tolerantes a falhas, (iv) melhora no desempenho ao acesso de dados locais e (v) administração de sistemas.

O conceito de migração de processos se aplica desde redes locais a redes globais, com conexão de baixa velocidade. Existem três componentes importantes para qualquer política de migração de processos (RIGHI et al., 2009):

Figura 8 – Representação da migração de processos.



Fonte: Adaptado de (MILOJICIC et al., 2000).

- Política de Balanceamento de Carga;
- Mecanismo de Migração de Processos;
- Execução Remota.

A política de balanceamento de carga decide "Quando" e "Onde" um processo será migrado, baseado em "Qual" o processo a ser migrado. O mecanismo de migração de processos define "Como" o processo será migrado do processador origem para o destino. A execução Remota é responsabilidade do processo no processador destino. Todas estas informações fazem sentido apenas quando a definição de "Quando" está relacionada com a importância da migração. Migrar um processo que está terminando sua execução acabará gerando um custo maior ao sistema todo; ou migrar um processo para um processador que já está com uma carga razoável, certamente irá impactar nos demais processos (RIGHI et al., 2009).

Este trabalho utiliza o modelo MigBSP, que responde as perguntas de "Qual" e "Onde"; ele utiliza *supersteps* BSP. Desta forma, a pergunta "Como" não é importante, pois ela está na implementação de migração utilizada (RIGHI et al., 2009).

2.5 O Modelo *Bulk Synchronous Parallel*

O modelo *Bulk Synchronous Parallel* (BSP) foi apresentado por Valiant (1990) como uma junção de partes de arquitetura de máquina e *software*. O BSP pode ser considerado um estilo de programação paralela para propósitos gerais em aplicações e variadas arquiteturas. BSP tem como características principais (VALIANT, 1990): (i) ser simples de programar, (ii) ser independente da arquitetura onde vai ser utilizado e (iii) ser possível prever o desempenho de uma aplicação em uma arquitetura conhecida.

Segundo Righi et al. (2009), BSP se assemelha muito à programação sequencial, sendo necessário apenas algumas informações extras para descrever o paralelismo; e com alguns parâmetros adicionais ao código é possível coletar informações e computar o tempo de execução. Um programa BSP quando movido de uma arquitetura para outra não necessita alterações, ele foi feito para ser independente de arquiteturas (VASILEV, 2003). O BSP compreende uma coleção de processadores, cada qual com sua memória; e uma rede que interconecte todos, não importando que seja local ou global (de baixa velocidade), apenas requer comunicação assíncrona entre todos os processos.

O projeto BSP se torna mais assertivo ao permitir um modelo de custos tratável e confiável. Considerando isto, a Tabela 2 apresenta algumas iniciativas para melhorar o desempenho em aplicações BSP (GERBESSIOTIS; SINIOLAKIS, 2001).

Tabela 2 – Iniciativas para um melhor desempenho em BSP.

Mecanismo	**Descrição**
Equilíbrio na Computação	Balancear a computação entre processos a cada *superstep*, considerando que w é o maior tempo de computação e a barreira precisa aguardar pelos processos mais lentos. Considerando uma arquitetura heterogênea, é possível reescalonar os processos levando em consideração o valor de w.
Equilíbrio na Comunicação	Balancear a comunicação entre processos, considerando que h é a maior entrada e saída de dados. O equilíbrio na comunicação é importante se utilizarmos links sobrecarregados ou através da Internet. É possível remapear os processos de forma a que fiquem aproximados no mesmo nível de rede.
Redução de *SuperSteps*	Minimizar o número de *supersteps*, considerando o valor de l que aparece no final da execução.

Fonte: Adaptado de (RIGHI et al., 2009).

O modelo BSP pode ser utilizado em uma variedade de linguagens de programação, ele apenas precisa oferecer barreira de sincronização e diretivas de comunicação assíncrona. BSP é mais utilizado em SPMD, com linguagem C ou FORTRAN utilizando MPI (RIGHI et al., 2009; GERBESSIOTIS; SINIOLAKIS, 2001; SRIVATSA; KAWADIA; YANG, 2012).

Programas BSP possuem duas estruturas, Vertical e Horizontal, conforme descrito a seguir (GERBESSIOTIS; SINIOLAKIS, 2001; RIGHI et al., 2009).

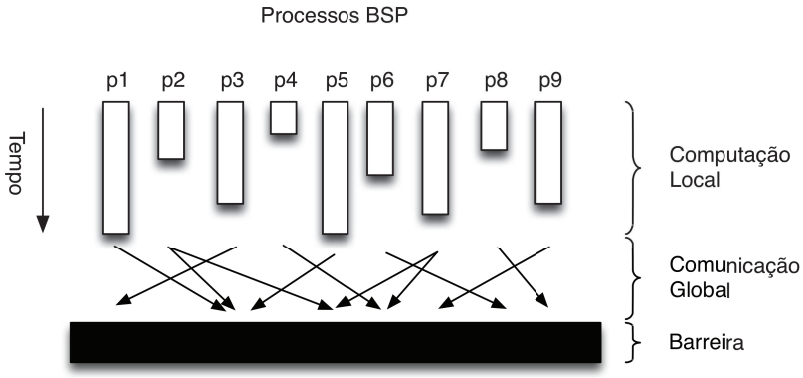
2.5.1 Estrutura Vertical

A estrutura vertical segue o processamento de forma sequencial, iniciando pelo processamento em três fases (*steps*), onde as três *steps* formam uma *superstep*, cada *superstep* é com-

posta pelos itens a seguir descritos e também apresentados na Figura 9.

- Computação Local, utilizando apenas valores armazenados localmente;
- Comunicação entre processos, transferindo dados entre processadores;
- Barreira de Sincronização, que aguarda a comunicação de todos os processos.

Figura 9 – Representação de uma *Superstep*.



Fonte: Adaptado de (RIGHI et al., 2009).

Uma das vantagens do BSP é deixar a comunicação para o final da execução, desta forma não há necessidade de sincronizações durante a execução. O formato em *supersteps* com a barreira no final não permite *deadlocks*, e se qualquer problema ocorrer durante uma execução, basta refazer aquela *superstep*, não é necessário refazer toda a execução.

2.5.2 Estrutura Horizontal

A estrutura horizontal é definida pela consistência e concorrência de um número finito de processos BSP, que são mapeados de forma não ordenada aos processadores independente de sua localização física. O desempenho em aplicações BSP pode ser descrito por três parâmetros: (i) p = número de processadores, (ii) l = tempo gasto por uma barreira de sincronização e (iii) g = taxa em que o dado acessado pode ser entregue de maneira aleatória.

O parâmetro g está relacionado à largura de banda da rede e depende de alguns fatores como: (i) protocolos utilizados na conexão, (ii) *buffer* de gerenciamento do processador e rede, (iii) estratégia de roteamento utilizada pela rede e o (iv) sistema de execução do BSP. Em resumo, a variável g representa o custo para iniciar uma transferência de dados em cada *superstep* (RIGHI et al., 2009). Considerando que w seja o custo da computação local de cada processo em uma *supertstep* s; e h_{out} e h_{in} sejam respectivamente a quantidade de mensagens enviadas e recebidas durante uma *superstep* s; e l o custo da barreira de sincronização, então o tempo de execução

de uma *superstep s* pode ser dado pela equação (2.3) (GERBESSIOTIS; SINIOLAKIS, 2001; VASILEV, 2003).

$$t_S = MAX\{w\} + MAX\{h_{out}, h_{in}\} \times g + l. \tag{2.3}$$

A partir da equação (2.3), se considerarmos *W* e *H* respectivamente como $MAX\{w\}$ e $MAX\{h_{out}, h_{in}\}$, e considerarmos *n* como o número de *supersteps*, o custo total da aplicação pode ser previsto pela equação (2.4).

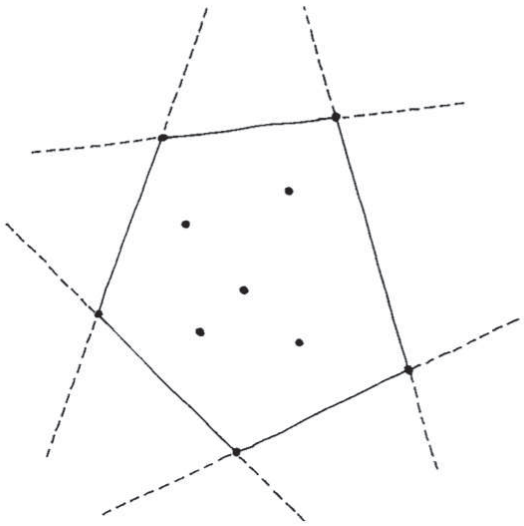
$$t = \sum_{i=0}^{n} W_i + g \times \sum_{i=0}^{n} H_i + n \times l. \tag{2.4}$$

Os parâmetros utilizados podem ser facilmente adicionados ao código do programa BSP, e os custos totais podem ser calculados e utilizados como mecanismos para prever o desempenho de programas em arquiteturas paralelas (RIGHI et al., 2009).

2.6 Envoltória Convexa

Uma Envoltória Convexa, também conhecida como Casco Convexo ou Fecho Convexo, é um conjunto de pontos em um espaço plano, é a menor convexa que contenha todos os pontos em um plano. Um exemplo comum da envoltória convexa é um elástico que envolva todos os pontos, a figura que ele formar ao abraçar todos eles forma a envoltória. Porém, ele não deve se contrair na ligação entre um ponto e outro (KARAVELAS; TZANAKI, 2011).

Figura 10 – Representação do cálculo de uma Envoltória Convexa.



Fonte: Retirado de (REZENDE; STOLFI, 1994).

Formalmente, a envoltória convexa pode ser definida como a interseção de todos os conjuntos convexos que contenham X ou o conjunto de todas as combinações convexas de pontos em

X, conforme mostra a Figura 10. Envoltórias também são extensões do espaço euclidiano (mais informações na seção (2.7) a seguir) para vetores de espaço real. Consideramos P um conjunto, a envoltória convexa de P é o conjunto $Conv(P) = \bigcap_{X \in C_p} X$, onde $C_p = \{X \supset P \mid X$ é um conjunto convexo $\}$ (REZENDE; STOLFI, 1994).

O algoritmo para encontrar o conjunto finito de pontos em um plano é um dos problemas fundamentais da geometria computacional (KARAVELAS; TZANAKI, 2011). Existem diversos algoritmos para construção da envoltória convexa, são utilizadas uma das duas abordagens para identifica-lo, (i) determinar os pontos do conjunto que são vértices de $Conv(S)$ (conjunto dos pontos que formam o convexo) ou (ii) identificar os pontos que não são vértices e elimina-los. Estes algoritmos podem chegar a uma complexidade de $\bigcirc(n^3)$. A seguir será descrito brevemente 3 algoritmos para encontrar a Envoltória Convexa de um conjunto (REZENDE; STOLFI, 1994).

- **Algoritmo Ingênuo**: O objetivo deste algoritmo é eliminar os pontos do conjunto que sejam inferiores a envoltória do convexo. Desta forma, um ponto é interno a envoltória se e somente é interior a algum triângulo formado por alguma tripla de pontos do conjunto. A complexidade total deste algoritmo é de n^4 (REZENDE; STOLFI, 1994).

- **Algoritmo não tão ingênuo**: Este algoritmo considera um sub-conjunto qualquer de todo plano. Traçando uma reta qualquer, se ela estiver fechada ao sub-conjunto, dizemos que ela é uma reta de suporte do sub-conjunto. Sendo assim, cada par de pontos que passa por uma reta de suporte é uma aresta da envoltória convexa. Sendo a complexidade deste caso $\bigcirc(n^3)$ (REZENDE; STOLFI, 1994).

- **QuickHull**: Este algoritmo utiliza um método de divisão e conquista. Ele inicialmente busca os pontos com menor e maior coordenada X. Considera como uma linha e de forma recursiva, busca os pontos mais distantes em Y abaixo e acima da linha de X. Este caso possui uma complexidade de $\bigcirc(n^2)$ (BARBER; DOBKIN; HUHDANPAA, 1996)

Desta forma, vemos que por mais que possa parecer natural generalizar a visualização destes objetos sendo envolvidos por um tipo de membrana plástica. Encontrar uma superfície em equilíbrio que forme uma real envoltória convexa pode ser uma tarefa mais complexa e custosa em termos de complexidade. Esta descrição representa apenas a ideia do plano em duas dimensões, porém quando consideramos um plano tridimensional, encontramos uma complexidade ainda maior, a seção a seguir nos auxilia a entender melhor o espaço e suas dimensões.

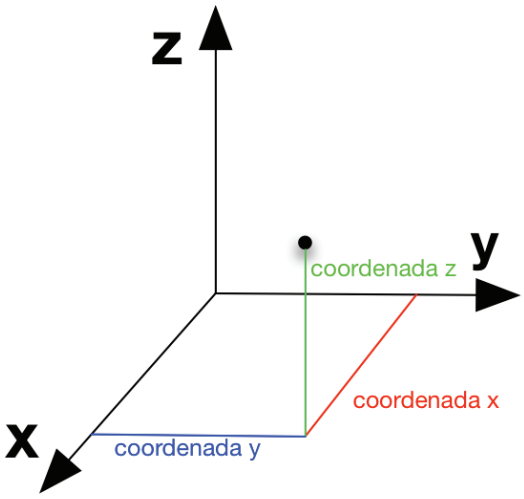
2.7 Espaço Tridimensional

Por definição, um espaço tridimensional é aquele que possui três dimensões: altura, largura e profundidade. A geometria analítica e a descritiva tratavam o espaço tridimensional de maneiras diferentes, a primeira utilizava a linguagem algébrica e a segunda a geométrica. Com a terceira

dimensão alguns problemas geométricos passaram a ser estudados, como: distancia entre planos paralelos e retas, calculo de volumes, entre outros (CAMARGO; BOULOS, 2009).

Na matemática, a geometria analítica ou geometria cartesiana descreve cada ponto em um espaço tridimensional como três coordenadas, conforme a Figura 11. Um eixo de três coordenadas normalmente é constituído de três retas perpendiculares, com um ponto central (origem para um eixo positivo) onde elas se cruzam. Normalmente estes eixos são conhecidos como X, Y e Z (CAMARGO; BOULOS, 2009; REZENDE; STOLFI, 1994).

Figura 11 – Sistema cartesiano de coordenadas.



Fonte: Adaptado de (CAMARGO; BOULOS, 2009).

Relativo aos eixos, a posição de cada ponto no espaço tridimensional é dado por uma tripla ordenada de números reais, sendo que cada numero é a distancia entre o ponto de origem até a sua posição em cada plano. A área de cada eixo, que pode ir do número menor infinito até o maior infinito é chamado de plano, o espaço tridimensional é composto de três planos (CAMARGO; BOULOS, 2009).

De acordo com o eixo de três coordenadas, é definido o espaço cartesiano, onde se considera cada eixo um espaço com dimensões, e utilizando um sistema de referência, pode-se definir matematicamente a localização dos pontos. O sistema de referência possui um ponto origem, direção e sentido, dados que podem ser encontrados na tripla ordenada de números reais para cada ponto. Esta tripla (x, y, z) corresponde a uma posição numérico em cada eixo, o sinal de positivo ou negativo representa o sentido. O ponto de origem é conhecido por (0, 0, 0), onde os eixos se cruzam (MENDELSON, 2011; CAMARGO; BOULOS, 2009).

Como todas estas informações e ferramentas que a utilização do plano cartesiano em um espaço tridimensional, o cálculo de distância entre pontos é facilitado, utilizando a fórmula da equação (2.5) é possível encontrar a distância entre os pontos, mesmo em um espaço tridimensional (MENDELSON, 2011; CAMARGO; BOULOS, 2009).

$$D = \sqrt[2]{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{2.5}$$

Para esta equação (2.5), consideramos D a distância entre dois pontos no espaço tridimensional, e cada par de valores para o ponto 1 e 2 em cada respectivo eixo (x, y e z). Desta forma, podemos traçar todas as distâncias de um ponto para os demais e utilizar estas informações para definir pontos que estejam dentro de uma sub-área (CAMARGO; BOULOS, 2009; REZENDE; STOLFI, 1994).

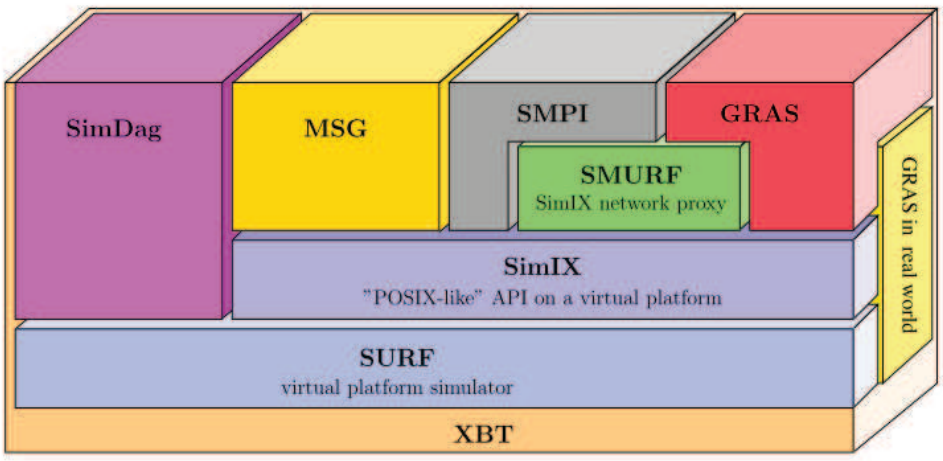
2.8 Simulador SimGrid

SimGrid é uma ferramenta que provê funcionalidades para a simulação de aplicações distribuídas em ambientes distribuídos heterogêneos. O objetivo do SimGrid é auxiliar na pesquisa em sistemas paralelos e distribuídos de larga escala como: *Grid*, *P2P* e *Cloud* (SIMGRID, 2013).

A necessidade de efetuar simulações ao invés de experimentos reais, fez com que Casanova (2001) aprimorasse um trabalho na Universidade da Califórnia em San Diego em meados de 1999, para em 2001 apresentar o simulador SimGrid. Desde então o SimGrid vem sendo aprimorado de acordo com as necessidades da área de sistemas paralelos e distribuídos. Atualmente ele está na versão 3.10 (Novembro de 2013).

O SimGrid permite uma série de parâmetros de checagem, mas analisa principalmente a latência (tempo de acesso a um recurso) e a taxa de serviço (número de unidades de trabalhado realizadas por unidade de tempo). Ele requer uma topologia de rede totalmente conectada, ou seja, uma matriz para moldar toda a estrutura de rede que é requerida.

Figura 12 – Módulos do SimGrid.



Fonte: Retirado de (CASANOVA; LEGRAND; QUINSON, 2008).

O SimGrid oferece quatro interfaces de uso: (i) **SimDag** e (ii) **MSG**, voltadas a pesquisadores e a (iii) **SMPI** e (iv) **GRAS** voltadas a desenvolvedores. A Figura 12 apresenta a estrutura e

os módulos do SimGrid. As outras camadas são os recursos internos do simulador: (i) **XBT** é a caixa de ferramentas do simulador, desenvolvida em ANSI C, ela fornece recursos para log e exceções; (ii) **SURF** é o núcleo do simulador, ele é modular para permitir modificações, porém deve ser alterado com cautela para não comprometer o desempenho das mesmas; (iii) o módulo **SimIX** provê a API POSIX-like, que permite o desenvolvimento de APIs que implementam a abstração de múltiplos processos concorrentes; e o módulo (iv) **SMURF** permite distribuir a simulação de processos em um aglomerado de computadores (CASANOVA; LEGRAND; QUINSON, 2008).

Módulos do SimGrid:

- **SimDag**: Permite simular heurísticas de escalonamento para aplicações estruturadas como grafos de tarefas, é possível criar tarefas com dependências de outras tarefas e analisar informações da arquitetura;
- **MSG**: É o módulo mais utilizado e por isso não está sendo alterado, apenas ampliado. Seu foco é grades de Computadores Pessoais e fornecer recursos para algoritmos de escalonamento. O MSG será melhor detalhado a seguir;
- **SPMI**: Módulo mais recente, permite a utilização de aplicações nativas MPI, interceptando suas chamadas internas;
- **GRAS**: permite o desenvolvimento de aplicações distribuídas, porém, para ser utilizada por aplicações sem alteração de código, foi adicionado o módulo **GRAS in Real World**, em que aplicações reais podem ser utilizadas.

O módulo utilizado pelo MigBSP e pelas heurísticas aqui apresentadas é o **MSG**. Ele possui uma série de funções que permitem a construção de um sistema distribuído realístico, de forma simples e eficiente. Suas principais funções são:

- Central da Simulação: funções para controle e configuração da simulação;
- Gerenciamento de Processos: descreve a estrutura para gerenciamento dos principais processos do simulador;
- Gerenciamento de Hosts: define a estrutura dos hosts do ambiente, e recursos para gerenciá-los;
- Gerenciamento de Tarefas: permite o gerenciamento das tarefas dentro da estrutura do MSG;
- Gerenciamento de Mailbox: recursos para gerenciar as formas de comunicação no ambiente;
- Gerenciamento de Arquivos: ferramentas para tratar e utilizar arquivos na estrutura.

Para utilizar o simulador são necessários três arquivos: (i) código do sistema que está sendo desenvolvido, (ii) arquivo de plataforma conforme a Figura 13, onde constam as definições do ambiente a ser simulado, e (iii) arquivo de implantação (*deploy*) da aplicação, exemplificado na Figura 14, onde ficam as definições da aplicação e o mapeamento dos processos nos recursos .

Figura 13 – Exemplo de XML de Plataforma do SimGrid.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "surfxml.dtd">
<platform version="2">
        <AS id="AS0" routing="Full">
                <!-- Definicao dos Hosts -->
                <host name="host1" power="1E8"/>
                <host name="host2" power="1E8"/>

                <!-- Links -->
                <link name="link1" bandwidth="1E6" latency="1E-2" />
                <route src="host1" dst="host2">
                        <link:ctn id="link1"/>
                </route>
        </AS>
</platform>
```

Fonte: Adaptado de (SIMGRID, 2013).

Figura 14 – Exemplo de XML de Implantação(*Deploy*) do SimGrid.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
        <!--Processo Principal-->
        <process host="Tremblay" function="master">
                <argument value="20"/>       <!--Numero de Tarefas-->
                <argument value="50000000"/> <!--Tam. de Computacao-->
                <argument value="1000000"/>  <!--Tam. de Comunicacao-->
                <argument value="Jupiter"/>  <!--Primeiro Escravo-->
                <argument value="Fafard"/>   <!--Segundo Escravo-->
                <argument value="Ginette"/>  <!--Terceiro Escravo-->
                <argument value="Bourassa"/> <!--Quarto Escravo-->
                <argument value="Tremblay"/> <!--Processo Principal-->
        </process>
        <!-- Processos Escravos, sem argumentos-->
        <process host="Tremblay" function="slave"/>
        <process host="Jupiter" function="slave"/>
        <process host="Fafard" function="slave"/>
        <process host="Ginette" function="slave"/>
        <process host="Bourassa" function="slave"/>
</platform>
```

Fonte: Adaptado de (SIMGRID, 2013).

O simulador foi desenvolvido utilizando C, Java, Ruby e Lua, podendo ser utilizado sob a

licença LGPL[1] (*GNU Library General Public License*) trabalha tanto com C quanto com Java e nos Sistemas Operacionais Linux, Windows, Mac OS e AIX. O SimGrid suporta um grande número de simulações, estimados em torno de 2 milhões de processos para um computador com 16GB de memória. No anexo A está um bloco de código com exemplo do SimGrid em C (CASANOVA, 2001).

2.9 Considerações Parciais

Este capítulo apresentou uma breve descrição sobre os principais assuntos relevantes para este trabalho. Inicialmente foi abordado sobre computação paralela, dando a ideia de execução de múltiplos processadores trabalhando de forma paralela para resolver um problema de forma a obter uma solução em menor tempo. Para isto, existem os (i) Multiprocessadores, sistemas em que dois ou mais processadores compartilham memória e os (ii) Multicomputadores, em que vários computadores se comunicam através de uma rede, que pode ser de baixo acoplamento. Também foi abordado sobre a classificação de computadores com base na Taxonomia de Flynn, podendo ser definidos como computadores do tipo SISD, MISD, SIMD e MIMD.

Posteriormente foi descrito escalonamento, que permite a distribuição de recursos para consumidores, a fim de gerenciar a execução de tarefas/*threads* em um sistema. Este escalonamento pode ser tanto em nível de sistema operacional, em que o SO divide o mesmo recurso por tempo para múltiplos processos, ou em nível de rede, em que processos são escalonados em computadores. Para este componente, existe a classificação com base na Taxonomia de Cassavant, em que o processamento é distribuído de forma (i) Heurística, sendo classificada como Global e Local, Estático e Dinâmico, com escalonamento Ótimo e Subotimo; e Horizontal, em que o escalonamento pode ser adaptativo ou não-adaptativo, em que a informação dos processos é persistida em todo o ambiente.

O problema do escalonamento vem sendo tratado há um longo tempo, e normalmente com ele se relaciona o balanceamento de carga, com o objetivo de fazer com que os processos utilizem de forma homogênea os recursos. Algumas questões são utilizadas para analisar um balanceamento de carga, como: (i) quem decide quando ele atua, (ii) quais informações ele utiliza e (iii) onde ele irá efetuar o balanceamento de carga. Também com execução em nível Global e Local, ele pode ser iniciado pelo nodo mais carregado, ou com os nodos com menor carga.

A migração de processos é na verdade uma abstração em que um processo que está executando em um processador é interrompido, seu status guardado, suas informações movidas para outro processador e inicializado novamente, dando a ideia de que ele foi movido. Esta movimentação tem como objetivo aproveitar mudanças de *hardware* e da aplicação em execução, sendo relevante três questões: (i) Quando um processo será movido, (ii) para Onde este pro-

[1] A LGPL é um formato de licença de *Software* que foi desenvolvido pela FSF (*Free Software Foundation*) de Richard Stallman, ela visa regulamentar a utilização de bibliotecas de código aberto por outras aplicações. Disponível em: https://www.gnu.org/.

cesso será movido e (iii) Qual processo será movido. A resposta destas três perguntas permite uma migração eficiente, e a melhor forma de executa-la é utilizando o modelo BSP, que executa em fases de uma *superstep*, sendo as três fases normalmente definidas como: (i) computação local, (ii) comunicação global e (iii) uma barreira de sincronização. O BSP é considerado um modelo simples de implementar, independente de arquitetura e possível prever o desempenho da aplicação que o utilizará, ele pode trabalhar tanto de forma vertical quanto horizontal.

Verificamos também dois conceitos matemáticos muito importantes neste trabalho, a Envoltória Convexa e a utilização de Espaços Tridimensionais. A envoltória propõe identificar os pontos mais distantes em um plano traçando sua convexa, já a definição do Espaço Tridimensional da o entendimento da própria envoltória, que aqui está sendo trabalhada em duas dimensões, para sua aplicacao no modelo tridimensional em que as duas heurísticas trabalham.

Por fim, foi apresentado o simulador SimGrid, ele foi utilizado para o desenvolvimento do modelo MigBSP e será utilizado para as heurísticas aqui propostas. Ele oferece recursos para simulações de aplicações distribuídas em ambientes distribuídos heterogêneos. O SimGrid possui alguns componentes que permitem uma simulação realística e de alto desempenho, tendo quatro módulos: (i) SimDag, (ii) SPMI, (iii) GRAS e (iv) MSG, sendo o último o módulo mais utilizado com foco em grades computacionais, e é o módulo que será utilizado nesta proposta.

3 TRABALHOS RELACIONADOS

Os trabalhos aqui relacionados servem como base para a pesquisa que está sendo desenvolvida, eles apresentam as técnicas e ferramentas mais recentes para lidar com balanceamento de carga e migração de processos utilizando bibliotecas BSP. O objetivo é mostrar a relevância deste trabalho e quais lacunas ele se propõe a preencher.

Os trabalhos que são apresentados nas seções a seguir tem como foco resolver problemas de migração de processos e balanceamento de carga em ambientes distribuídos. Desta forma, consideramos a utilização de BSP como uma vantagem, e não como um requisito.

3.1 Modelo MigBSP

O MigBSP é um modelo de reescalonamento de processos apresentado por Righi et al. (2009) como uma ferramenta para auxiliar nas decisões de balanceamento de carga em aplicações BSP. Esta ferramenta responde as questões necessárias para a migração de processos: "Qual?", "Onde?" e "Quando?". O modelo aproveita a etapa da barreira de sincronização das aplicações BSP para ajustar a localização dos processos para melhor distribuir a carga no sistema, e consequentemente reduzir o tempo da *superstep* (RIGHI et al., 2010). As informações são coletadas durante a execução da aplicação e são utilizadas para compor o valor do *potencial de migração* (PM) de cada processo.

Conforme a Tabela 2 na seção 2.5, que mostra a relação de equilíbrio em aplicações BSP, o MigBSP atua na intenção de equilibrar a execução das tarefas e reduzir o custo de comunicação. Ele trabalha de forma reativa, analisando a aplicação durante a execução e executando quando ela está parada na barreira de sincronização. Ele oferece um balanceamento de carga automático na forma de um *middleware* sem a necessidade de conhecimento prévio da aplicação.

Trabalhando com a menor intrusividade possível, o MigBSP utiliza, além do PM, outros dois parâmetros adaptativos de controle de intervalo de reescalonamento: um (i) baseado no balanceamento de processos e outro (ii) baseado no número de chamadas sem a necessidade de migração, conforme descritos abaixo:

- **Adaptação 1: Considerando o balanceamento de processos**:

  Este fator contabiliza o número de *supersteps* α ($\alpha \in N^*$) entre um reescalonamento e outro, no passo em que não é necessário uma intervenção, aumenta-se o valor de α de modo a evitar intrusões desnecessárias. Quando um desbalanceamento é detectado, o valor de α é reduzido para que as decisões de migração futuras ocorram mais rapidamente, desta forma permitindo uma estabilização do sistema com mais rapidez.

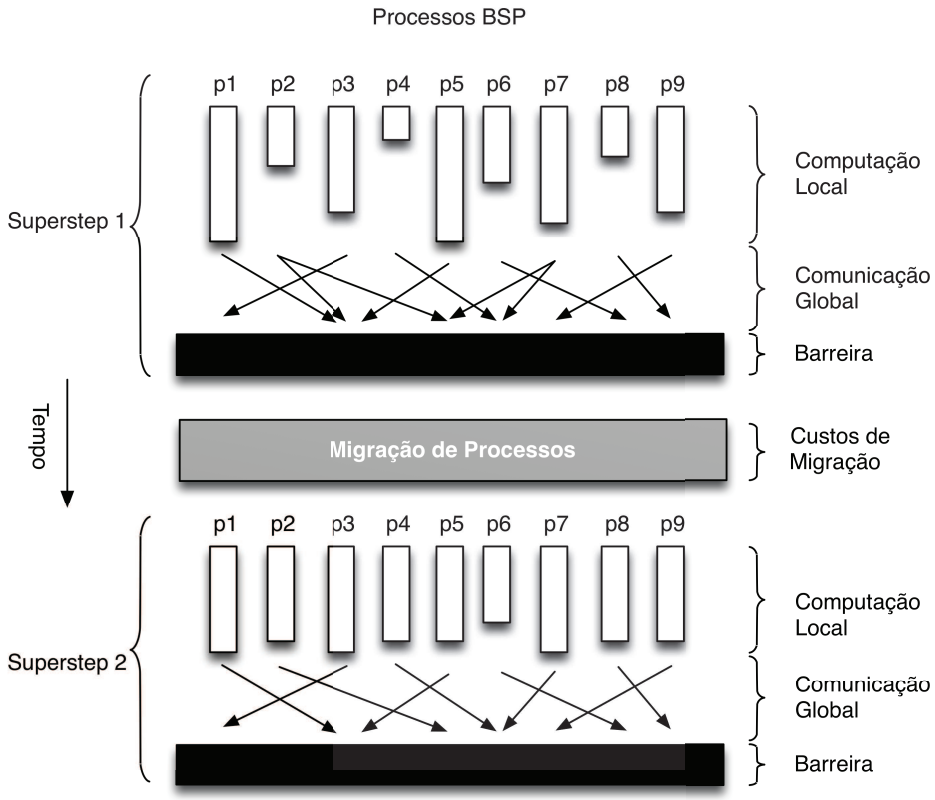
  No início da execução da aplicação, um parâmetro D é definido, este valor indica um percentual de tolerância para processos estáveis. Este valor multiplicado pelas equações (3.1) e (3.2) informa, caso uma delas for falsa, o desbalanceamento do sistema. A equação

(3.1) avalia se o tempo do processo mais lento no sistema for menor do que o tempo médio dos processos, já a equação (3.2) verifica se o processo com menor tempo for maior do que o tempo médio dos processos. A Figura 15 auxilia na definição deste processo.

$$tempo\ processo\ mais\ lento < tempo\ medio\ processos \times (1 + D). \quad (3.1)$$

$$tempo\ processo\ mais\ rapido > tempo\ medio\ processos \times (1 - D). \quad (3.2)$$

Figura 15 – Representação de processos não balanceados e balanceados.



Fonte: Adaptado de (RIGHI et al., 2010).

- **Adaptação 2: Considerando a quantidade de chamadas para Reescalonamento que não ocasionaram migração**:

  Este fator ocorre na variação do parâmetro D, aumenta-se o valor de D a cada ω consecutivas barreiras sem nenhuma migração acontecer. O aumento de D permite uma variância maior dos tempos de término do processo em cada *superstep*. Entretanto, enquanto ocorrer uma migração, o valor de D é reduzido apresentando uma rigidez maior na detecção do desbalanceamento dos processos. Segundo Righi et al. (2010), este ajuste é importante, principalmente onde os custos de migração são altos.

A definição de qual processo será migrado ocorre através do valor do PM. A cada barreira de sincronização é computado para o processo i, n funções $PM(i,j)$, onde n é a quantidade de processadores destinos possíveis e $j = 1, ..., n$ representa um destino específico. A utilização das métricas de computação $Comp(i,j)$, comunicação $Comm(i,j)$ e memória $Memm(i,j)$ permitem uma análise de $PM(i,j)$ como um somatório no qual $Comp$ e $Comm$ possuem uma influência favorável a migração enquanto $Memm$ representa um fator contrário. Assim, a equação (3.3) mostra como PM é encontrado.

$$PM(i,j) = Comp(i,j) + Comm(i,j) - Memm(i,j). \tag{3.3}$$

A métrica $Comp$ tem o objetivo de simular o desempenho do processo i no processador j. A cada α ***supersteps***, este valor é calculado coletando as informações de tempo de execução (CT) e instruções (I_t) realizadas em cada ***superstep***. 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 indicando uma grande variância da quantidade de instruções e 1 apresentado um valor contínuo. Já $PI_t(i)$ é definido como o valor que prevê o comportamento do valor de I_t em uma ***superstep***. Ele é definido considerando os valores I_t das ***supersteps*** anteriores a partir da equação (3.4).

$$PI_t(i) = \begin{cases} I_t(i) & \text{se } t = k; \\ \frac{1}{2}PI_{t-1}(i) + \frac{1}{2}I_t(i) & \text{se } k < t \leq k + \alpha - 1. \end{cases} \tag{3.4}$$

Este método economiza memória e tempo de cálculo por considerar apenas as informações das duas últimas execuções, dado que estas são as mais significativas (RIGHI et al., 2009). Porém, o valor de $P_{comp}(i)$ se mantém independente da quantidade de realocações de processos realizadas.

A computação do processo i é considerada estável se a previsão está dentro de uma margem 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 pela equação (3.5).

$$CTP_t(i) = \begin{cases} CT_t(i) & \text{se } t = k; \\ \frac{1}{2}CTP_{t-1}(i) + \frac{1}{2}CT_t(i) & \text{se } k < t \leq k + \alpha - 1. \end{cases} \tag{3.5}$$

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 $ISet$, $Comp(i,j)$ é definido como:

$$Comp(i,j) = P_{comp}(i) \times CTP_{k+\alpha-1}(i) \times ISet_{k+\alpha-1}(j). \tag{3.6}$$

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

$$Comm(i,j) = P_{comm}(i,j) \times BTP_{k+\alpha-1}. \tag{3.7}$$

Para ponderar a comunicação entre os processos, o MigBSP considera apenas as recepções envolvidas provenientes do processador j para o processo i. $Comm(i,j)$ 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 transferidos de j para i, $PB(i,j)$ (RIGHI et al., 2009).

$$PB_t(i) = \begin{cases} B_t(i) & \text{se } t = k; \\ \frac{1}{2}PB_{t-1}(i) + \frac{1}{2}B_t(i) & \text{se } k < t \leq k + \alpha - 1. \end{cases} \tag{3.8}$$

Neste contexto, $B_t(i,j)$ indica a quantidade de *bytes* recebidos por i do processador j. A previsão do tempo de comunicação é representado por $BTP_{k+\alpha-1}$ que é representado na equação (3.9). Análogo a σ, β é o valor de aceitação de variação no padrão de comunicação.

$$BTP_t(i) = \begin{cases} BT_t(i) & \text{se } t = k; \\ \frac{1}{2}BTP_{t-1}(i) + \frac{1}{2}BT_t(i) & \text{se } k < t \leq k + \alpha - 1. \end{cases} \tag{3.9}$$

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

$$Memm(i,j) = M(i) \times T(i,j) + Mig(i,j). \tag{3.10}$$

Quanto menor o valor de $Memm$, mais interessante 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.

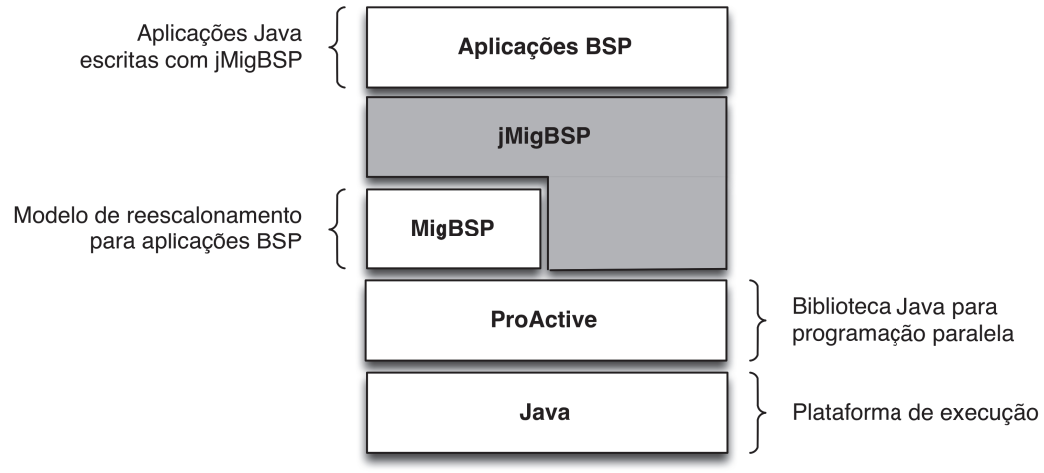
O modelo MigBSP foi desenvolvido utilizando o simulador SimGrid. Ele foi portado para Java, resultando no jMibBSP (GRAEBIN, 2012). Paralelamente a este trabalho, está sendo desenvolvido por Gomes (2013) um balanceador em AMPI baseado no MigBSP.

3.2 jMibgBSP

O *jMigBSP* é uma biblioteca apresentada por Graebin e Righi (2011), ela fornece uma API para o desenvolvimento de aplicações BSP utilizando Java. O jMigBSP utiliza os recursos de programação paralela do ProActive (CAROMEL; KLAUSER; VAYSSIèRE, 1998), fornecendo interfaces adaptadas para o desenvolvimento de aplicações BSP.

A biblioteca ProActive permite a escrita de aplicações paralelas, suas principais características são: programação orientada a objetos, migração de objetos, comunicação assíncrona e gerência de recursos. Ela oferece uma série de recursos, entre eles, comunicação em grupo e assíncrona, trabalha em ambientes heterogêneos composto por agregados de computadores e permite a comunicação entre ambientes utilizando VPN (*Virtual Private Network*) e SSH (Secure Shell). A Figura 16 mostra as camadas do jMigBSP.

Figura 16 – Estrutura de camadas do jMigBSP.



Fonte: Adaptado de (GRAEBIN; RIGHI, 2011).

O jMigBSP foi desenvolvido utilizando a ideia de (i) eficiência e (ii) flexibilidade, ou seja, permitindo o desenvolvimento de aplicações utilizando um *middleware* que oferece o reescalonamento de tarefas ou através da ativação explícita na aplicação; ele se torna flexível ao tratar o reescalonamento de objetos de forma automática.

Inicialmente o jMigBSP foi testado utilizando soma de prefixos para validar a troca de mensagens entre objetos e a sincronização das tarefas na aplicação. Posteriormente, duas aplicações foram utilizadas e comparadas com experimentos conhecidos: (i) Compressão de imagens utilizando Fractais e (ii) Transformada Rápida de Fourier (FFT) na compressão de imagens. O jMigBSP obteve um ganho de desempenho de mais de 90% se comparado com o código sequencial; já a FFT foi comparada com o BSPlib e o jMigBSP obtendo um resultado similar. Isto ocorre devido à sobrecarga nas chamadas do Java e ProActive(GRAEBIN; RIGHI, 2011, 2012).

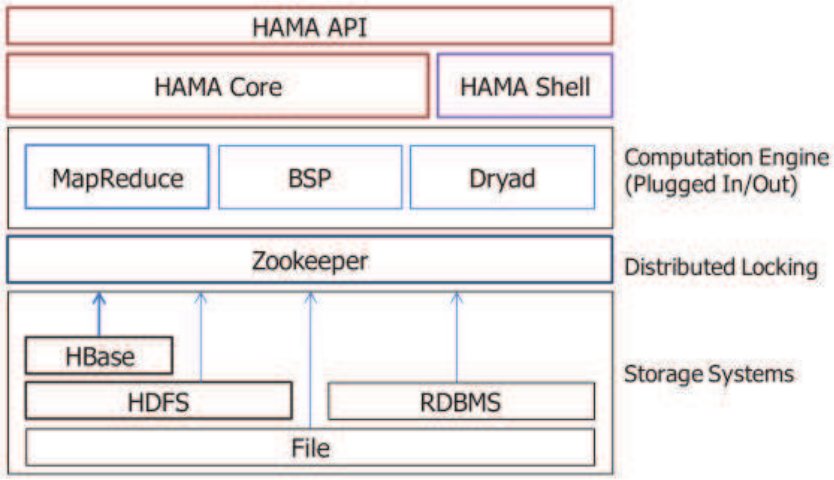
3.3 HAMA

HAMA (HAMA, 2013) é um *Framework* baseado na linguagem Java e que trabalha com BSP. Desenvolvido para aplicações com troca de mensagens, ele trabalha na camada acima do HDFS (*Hadoop Distributed File System*) (HADOOP, 2013) e faz uso do recurso de balanceamento de carga dele (CHUNG et al., 2012).

Conforme mostra a Figura 17, o HAMA pode ser dividido em três camadas (SEO et al., 2010): (i) *HAMA Core*, a camada que fornece recursos para computação de matrizes e grafos, ela também determina qual dos processadores de computação (*MapReduce*, BSP e *Dryad*) (MING et al., 2011; HAMA, 2013) mais apropriados a se utilizar; (ii) *HAMA Shell*, nesta camada é onde ocorre a interação com o usuário, onde as informações da execução são enviadas

e recebidas; e a (iii) *HAMA API* é a camada da aplicação em si.

Figura 17 – Estrutura de camadas do HAMA.



Fonte: Retirado de (SEO et al., 2010).

O HAMA suporte as características descritas abaixo:

- **Aplicável**: Pode ser aplicado a diversas aplicações que requeiram cálculos de matrizes e grafos, segundo Seo et al. (2010), um exemplo é o *me2day*, um serviço semelhante ao *Twitter* utilizado na Coreia do Sul;
- **Compatível**: Por incorporar recursos do Hadoop, o HAMA utiliza todas as funcionalidades dele, sendo compatível com todos os tipos de conectividades existentes no Hadoop;
- **Flexível**: Atualmente o HAMA dispõe de três ferramentas de cálculo (*MapReduce*, BSP e *Dryad*), entretanto, qualquer ferramenta que esteja em acordo com as interfaces do HAMA pode ser utilizada e conectada a ele;
- **Escalável**: Por sua compatibilidade com o Hadoop, o HAMA permite a utilização da infraestrutura de internet em serviços de nuvem como a Amazon EC2.

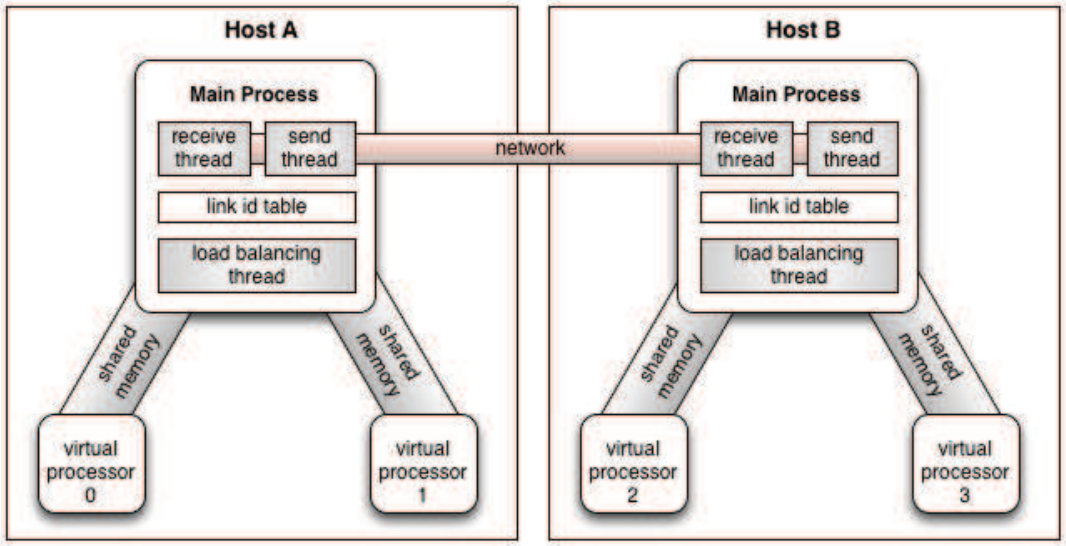
Diversos trabalhos já apresentam resultados pelo uso do HAMA, entre eles os de Zhang e Ge (2012) e Ting, Lin e Wang (2011). O segundo faz um estudo de um algoritmo paralelo para encontrar comunidades em redes baseado em grafos ponderados. já Zhang e Ge (2012) faz um comparativo entre a ferramenta de cálculos BSP e *MapReduce* para armazenar dados de redes sociais, neste caso o BSP se mostrou com melhor desempenho.

3.4 PUB

PUB ou *Paderborn University BSP Library* é uma biblioteca em C, inicialmente desenvolvida por Bonorden et al. (2003), com recursos de comunicação para o desenvolvimento de

algoritmos BSP (GAVA; FORTIN, 2009). Em 2007, Bonorden (2007) apresentou um trabalho utilizando a biblioteca PUB fazendo uso de de processadores virtuais através da criação de processos. Com isto ele mostrou os novos recursos da biblioteca, como a função de balanceamento de carga com a migração de processos.

Figura 18 – Estrutura de comunicação do PUB.



Fonte: Retirado de (BONORDEN, 2007).

A Figura 18 apresenta uma visão geral da estrutura de comunicação e funcionamento da biblioteca. As *threads* de (i) recepção e (ii) transmissão tem a função de: receber as mensagens de rede, colocá-las na fila da memória compartilhada e encaminhar para o processo correspondente; já a transmissão envia as mensagens da fila da memória para o processador virtual da rede. A *thread* do balanceador de carga coleta as informações do sistema e decide quando migrar um processador virtual. Cada nodo possui uma tabela de identificação de processadores. Quando um é migrado de nodo, a informação na tabela não é alterada imediatamente, o nodo sabe para qual nodo o processador virtual foi migrado, entretanto são necessárias trocas de informações a mais para rotear as mensagens para o processador virtual no seu novo destino a partir do nodo antigo.

A biblioteca permite algumas estratégias conhecidas de balanceamento de carga: Global Centralizada, Distribuída Simples, Distribuída Conservativa e Preditiva Global. Os processos a serem migrados são definidos em uma ideia de tolerância a falha, ou seja, quando um processo fica indisponível, devido à sobrecarga do nodo ou real indisponibilidade, ele marca o processo para migração. Para efetuar a migração, as decisões são tomadas considerando a carga atual dos processadores sem levar em consideração o histórico de comunicação no custo de migração. Com os resultados encontrados (BONORDEN, 2007), o autor confirma que a migração de processos é um recurso que oferece muitos benefícios em ambientes distribuídos.

3.5 MulticoreBSP

A biblioteca MulticoreBSP foi apresentada por Yzelman e Bisseling (2012), ela foi desenvolvida de forma orientada a objetos na linguagem Java, porém foi idealizada na linguagem C. Os autores preferiram desenvolver em Java pois ele oferece um conjunto menor de instruções, e a intenção dos autores é oferecer uma ferramenta de fácil aprendizado e transparente em relação à máquina paralela.

Para utilizar a biblioteca, são necessárias algumas alterações na aplicação, porém elas são simples, estender a classe $BSP\_PROGRAM$ e implementar duas funções: (i) $main\_part()$ que é executada por um único processo e (ii) $parallel\_part()$ que será incluída na parte do código que será executada em paralelo. Para a troca de mensagens, foi definida a classe BSP_COMM, que implementa as funções: $bsp\_put()$, $bsp\_get()$ e $bsp\_send()$, apenas estas três funções realizam comunicação, nenhum outro objeto pode efetuar comunicação entre *threads*.

Os autores apresentam uma comparação efetuada entre a biblioteca MulticoreBSP e um trabalho com o Algoritmo *BSPedupack* que Bisseling (2004) desenvolveu, demonstrando a utilização do MulticoreBSP.

Os resultados do MulticoreBSP foram interessantes para ambientes de memória compartilhada, onde ele apresentou uma boa eficiência. Entretanto, por se tratar de uma implementação Java, segundo Yzelman e Bisseling (2012), ela não é viável para aplicações de alto desempenho, sendo proposto uma nova implementação em C++ no futuro.

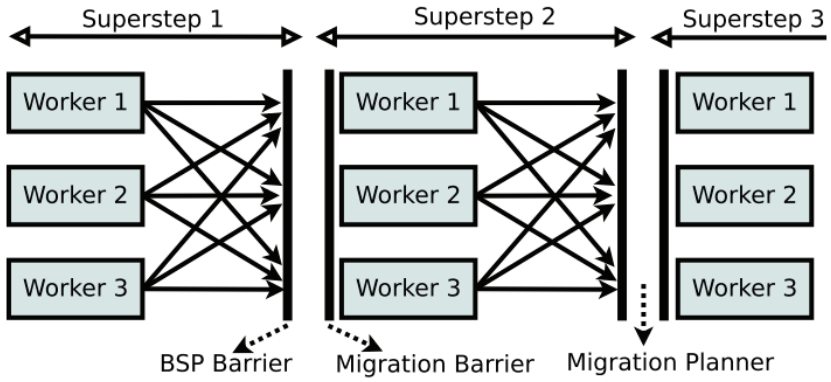
3.6 Mizan

O sistema Mizan foi apresentado por Khayyat et al. (2013), ele é baseado no sistema *Pregel* (MALEWICZ; AUSTERN; BIK, 2010), um sistema que suporta balanceamento de carga entre *supersteps*, ele tem como base para aplicações o modelo BSP. Conforme mostra a Figura 19, ele é composto por *workers* em uma arquitetura mestre-escravo, os *workers* são computadores ou máquinas virtuas (VM) que recebem tarefas através de um *worker* mestre.

Este sistema proporciona tolerância a falhas, eficiência e escalabilidade, pois monitora todas as tarefas durante sua execução (tempo, envio e recepção de mensagens). Com base nestes parâmetros ele monta um plano para minimizar a variação entre processos. Ele ordena os processos em duas listas, uma começando com o processo mais carregado e a outra com o menos carregado, desta forma, com base nesta lista e a análise das informações de cada processo, ele define os processos que devem ser migrados ao final de cada *superstep*.

O Mizan não necessita de um controlador central, ele possui total suporte a execução distribuída. Ele possui uma estrutura baseada na linguagem C++. Seguindo a determinação MPI, são definidas as funções para troca de mensagens em aplicações BSP. A seguir constam as cinco etapas que representam a arquitetura de balanceamento de carga do Mizan:

Figura 19 – Representação de *Supersteps* do Mizan.



Fonte: Retirado de (KHAYYAT et al., 2013).

- **Identificar Origem de Desbalanceamento**: Ao final da execução de cada *superstep*, efetuar uma comparação de todas as tarefas com a curva de uma distribuição Normal, e selecionar as que estivem fora;
- **Selecionar o Objetivo de Migração**: Etapa em que se define sobre utilizar uma política para equilibrar o custo de envio e recebimento de mensagens ou o tempo da *superstep*;
- **Alinhar Sub-Carregados com Sobre-Carregados**: Em uma lista ordenada, definir as tarefas que possuem uma relação oposta em sua carga, eles devem ser organizados respeitando o critério anterior;
- **Selecionar Tarefas para Migração**: A diferença entre uma tarefa e seu par equivalente irá definir o número de tarefas para a migração, ele será selecionado se estiver fora da curva da Normal;
- **Efetuar a Migração**: A migração da tarefa ocorre durante a fase da barreira de uma *superstep*, a tarefa é codificada em um *stream* que inclui seu ID, estado e outras informações relevantes para continuação de seu processamento em *supersteps* seguintes.

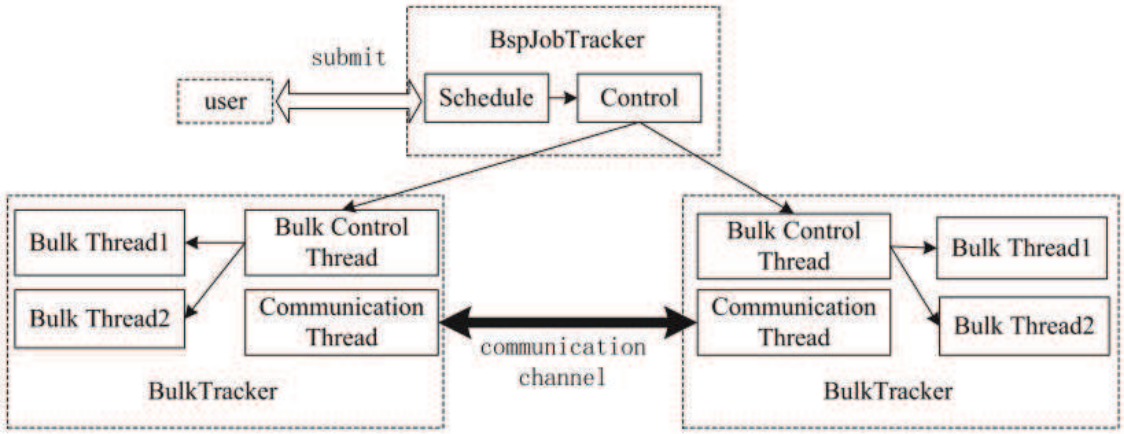
O Giraph! (GIRAPH!, 2013) é uma implementação Java de código aberto do sistema *Pregel*, ela foi utilizada para efetuar comparações com o Mizan. De forma geral, o Mizan obteve resultados muito superiores ao Giraph! (i) em execução de algoritmos estáticos ele obteve ganhos de 200% no tempo de execução; (ii) já em algoritmos dinâmicos o ganho foi de aproximadamente 87% no tempo de execução; (iii) em aplicações estáticas a redução foi de 40% onde foi inserido um *overhead* na aplicação e (iv) redução de 10% durante migrações.

3.7 BSPCloud

O BSPCloud é um modelo de escalonamento em ambiente de nuvem desenvolvida por Liu, Tong e Hou (2012). O objetivo dele é permitir a utilização de recursos como máquinas virtuais.

Ele oferece um modelo simples com custos realistas para migração de recursos em aplicações desenvolvidas para computação em nuvem.

Figura 20 – Estrutura geral do BSPCloud.



Fonte: Retirado de (LIU; TONG; HOU, 2012).

O modelo foi desenvolvido em Java, os componentes que fazem parte dele são mostrados na Figura 20 conforme sua descrição: (i) **BspJob** é a aplicação a ser submetida, (ii) **Bulk** é uma sub-tarefa da aplicação submetida no *BspJob*, (iii) **BspJobTracker** é a fila e o controlador de *Jobs*, e por fim, o (iv) **BulkTracker** é quem recebe as sub-tarefas *Bulk* de um *BspJobTracker*.

Conforme a Figura 20 mostra (LIU; TONG; HOU, 2012), o BSPCloud é composto por um *BspJobTracker* e, neste exemplo, dois *BulkTracker*; o *BspJobTracker* possui uma *thread* de escalonamento e outra de controle. Quando uma aplicação (BspJob) é submetida, ela não é imediatamente processada, apenas quando o escalonamento a escolhe ela inicia seu processamento. Após, a *thread* de controle particiona o *BspJob* em *Bulks* de acordo com os recursos disponíveis e repassa para os *BulkTrackers*, estes por sua vez, também possuem duas *threads*, a de Controle de *Bulk* e a de Comunicação. A *thread* de Controle de *Bulk* recebe o *BspJob* de um *BspJobTracker* e inicia a quantidade de *threads Bulk* indicada pelo Controle. Caso seja necessário um *Bulk* comunicar com ele, ele utiliza a *thread* de Comunicação que utiliza um canal de comunicação ponto-a-ponto entre eles. Este modelo requer que a aplicação a ser utilizada seja adaptada, incluindo as primitivas *bspOperate()* e *bspDataMap()* em sua codificação.

Segundo Xiaodong et al. (2013), três avaliações sobre o modelo foram realizadas: (i) análise de desempenho na multiplicação de matrizes, variando o número de máquinas virtuais com apenas um núcleo; (ii) novamente simulado a multiplicação de matrizes, porém utilizando máquinas virtuais com mais de um núcleo; e por fim (iii) foi avaliada a escalabilidade do modelo, eliminando e criando novas máquinas virtuais durante a execução. Os resultados mostraram que o modelo obtém um melhor resultado em arquiteturas com mais de um núcleo de processamento, permite uma boa escalabilidade e resulta em um considerável *SpeedUp*[1].

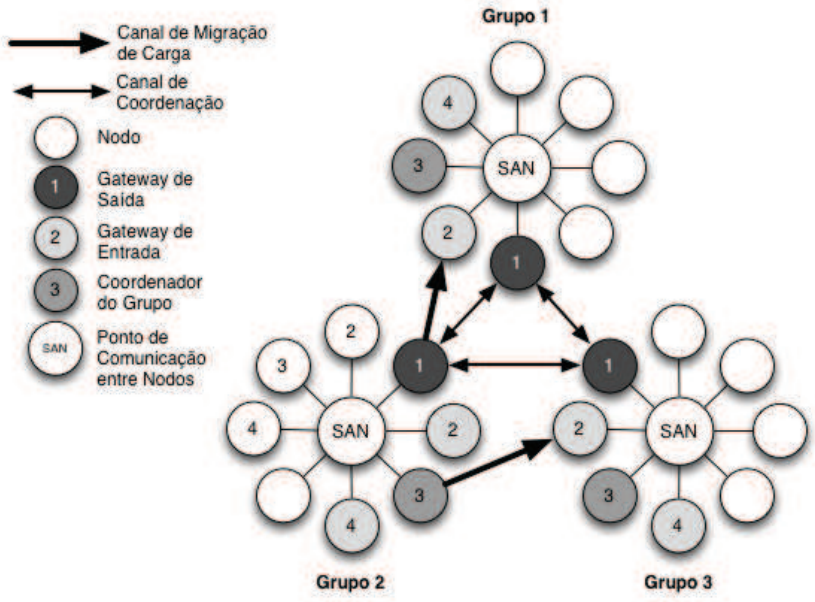
[1] *SpeedUp* se refere, em computação paralela, a quanto um algoritmo paralelo é mais rápido que o mesmo em um ambiente sem paralelismo.

3.8 DistPM

O trabalho de Li e Lan (2005) propõe um esquema de migração chamado DistPM, com uma nova abordagem hierárquica e avaliação de desempenho em tempo real. Este esquema trabalha com a heterogeneidade e características dinâmicas de sistemas distribuídos. Para reduzir o *overhead* na migração de cargas de trabalho, através de redes compartilhadas entre arquiteturas heterogêneas, foi desenvolvido um método heurístico baseado em um algoritmo de programação linear.

A principal ideia do esquema de Li e Lan (2005) é o conceito de grupos, conforme mostra a Figura 21. Nela é possível verificar que cada grupo é composto por um conjunto de processadores homogêneos conectados através de uma rede dedicada. Cada grupo pode ter computadores (i) paralelos de memória compartilhada, (ii) paralelos de memória distribuída e (iii) em um aglomerado de Linux. Cada grupo possui dois processos especiais: (i) Coordenador de Grupo e (ii) *Gateway* de Migração.

Figura 21 – Estrutura de grupos do DistPM.



Fonte: Adaptado de (LI; LAN, 2005).

Cada grupo elege um Coordenador de Grupo, e estes coordenadores elegem um Coordenador Global. Cada coordenador de grupo tem a função de coletar dados de carga, monitorar e predizer o status do sistema e aplicações. Durante cada passo de execução, o coordenador de grupo coleta as informações e repassa ao coordenador Global. Com estas informações ordenadas, o Global decide quais ações de balanceamento tomar para equilibrar o sistema.

A ideia de utilizar estas duas camadas é manter o conhecimento global e ao mesmo tempo ser escalável, e ainda reduzindo o número de mensagens trocadas pelo sistema. Em cada grupo, vários processos são considerados *gateways* de migração, eles são responsáveis por mover a

carga dos computadores com maior utilização para os com menor carga através do canal de migração. Durante a migração, este processo envia todas as mensagens para a migração através de uma única conexão direta ao computador destino, desta forma evitando a comunicação entre aglomerados e reduzindo a utilização de rede em ambientes com alta latência.

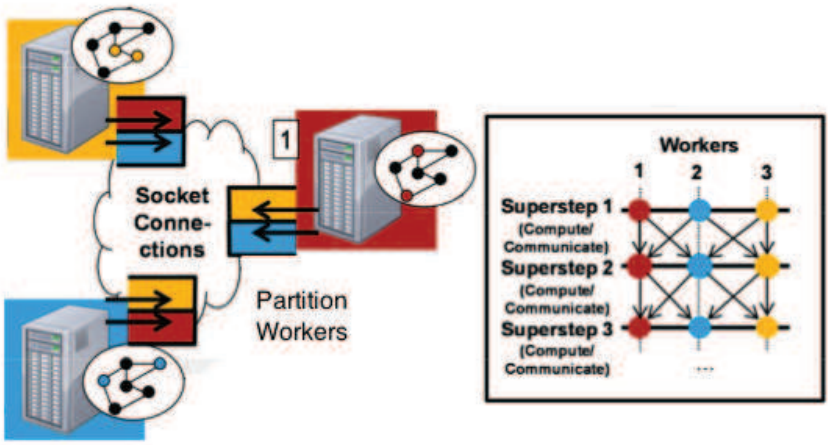
3.9 Pregel.NET

Originalmente o Google desenvolveu seu próprio *framework*, o Pregel, com o objetivo de oferecer escalabilidade para análise gráfica de grandes volumes de dados. Ele provê um design de arquitetura para o processamento gráfico com BSP. Entretanto, o Pregel é considerado não totalmente publico pelo Google, portanto testa-lo não é possível (REDEKOPP; SIMMHAN; PRASANNA, 2013).

Este trabalho propôs a utilização do Pregel em ambiente de *Cloud*, ou seja, utilizando a arquitetura dele. Desta forma, foi desenvolvida uma versão em .NET para ser utilizada juntamente ao Windows Azure[2] da Microsoft (REDEKOPP; SIMMHAN; PRASANNA, 2013).

O pregel.NET foi desenvolvido por Redekopp, Simmhan e Prasanna (2013), utilizando alguns algoritmos novos para prover análise gráfica complexa, entre eles: *betweenness-centrality* e *all-pairs shortest paths* (entre dados centrais e caminho mais curto). Também foi utilizada uma nova abordagem para escalonamento do processamento dos vértices gráficos, que mostrou que mesmo um gráfico bem particionado pode nem sempre levar ao melhor desempenho devido a barreira de sincronização.

Figura 22 – Distribuição de Workers e Processadores do pregel.NET.



Fonte: Adaptado de (REDEKOPP; SIMMHAN; PRASANNA, 2013).

Conforme a Figura 22 mostra, o pregel.NET trabalha no particionamento e escalonamento das atividades em cada *superstep*, alocando cada parte da tarefa para um *worker* especifico, e

[2]Plataforma de *Cloud* da Microsoft, disponível em http://www.windowsazure.com.

utilizando a elasticidade do ambiente em nuvem, sendo possível criar e destruir quantas máquinas virtuais forem necessárias a cada etapa.

Este trabalho mostrou que a melhor utilização da elasticidade do ambiente de nuvem, escalando dinamicamente os BSP *Workers*, apresenta um melhor desempenho, do que uma quantidade estática de *Workers* por um custo maior. Também foi possível obter um ganho de desempenho no volume de processamento de aproximadamente 3,5x ao melhor utilizar as maquinas virtuais (REDEKOPP; SIMMHAN; PRASANNA, 2013).

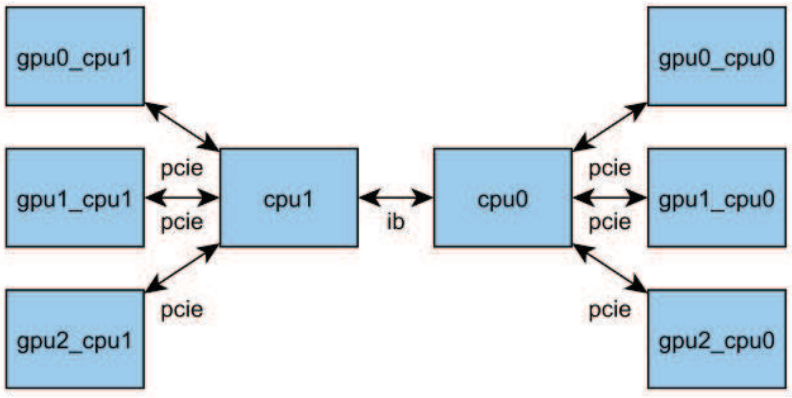
3.10 CPU-GPU Cluster

O trabalho de Mansouri et al. (2013) tem uma abordagem diferente, a ideia é propor um novo tipo de *Cluster*. Provendo maior desempenho no processamento de Fluxos de Dados Gráficos (*Data Flow Graph*) que representam o processamento de sinais digitais. A ideia é utilizar o processador gráfico como uma outra unidade de processamento no mesmo computador. Desta forma, criando um *cluster* heterogêneo dentro do mesmo computador (MANSOURI et al., 2013).

Este método permite que durante a execução do processo, caso haja necessidade de mais recursos, ao invés de migrar todo o processo para um outro computador, basta utilizar o processador gráfico para complementar o processamento necessitado (MANSOURI et al., 2013).

A abordagem aqui apresentada utiliza a estrutura de BSP, com o processamento entre barreiras de sincronização. Porém, a proposta aqui é efetuar a migração de nodos de dados durante a *superstep*, sem esperar a barreira. Desta forma, não interrompendo o fluxo de dados em processamento e reduzindo a variação no tempo de processamento (MANSOURI et al., 2013).

Figura 23 – Arquitetura Gráfica do Cluster.



Fonte: Adaptado de (MANSOURI et al., 2013).

A Figura 23 mostra a arquitetura deste trabalho, cada CPU interage com um bloco de GPU's através de uma conexão de alta velocidade. Também são utilizados *buffers* de memória para facilitar a comunicação através da arquitetura (MANSOURI et al., 2013).

Para a migração durante a execução, a ideia é ir removendo dados de uma CPU e passando ao mesmo tempo para a GPU. Para isto, é necessário ir interrompendo os fluxo de processamento aos poucos, de forma a garantir a execução continua do processo (MANSOURI et al., 2013).

Por fim, este trabalho obteve resultados apenas na execução, pois o foco da simulação foi uma aplicação gráfica, que já estava preparada para o processamento em GPU. Quando a aumento de performance, não ficaram claro os resultados, mas a implementação permitiu a continuidade da execução ao mover os fluxos de dados da CPU para GPU (MANSOURI et al., 2013).

3.11 Análise

Os trabalhos que foram detalhados servem como apoio a proposta aqui apresentada, eles embasam e demonstram a relevância do tema abordado. A Tabela 3 apresenta um resumo dos trabalhos descritos, demonstrado suas características e capacidades. Analisando ela é possível perceber a lacuna que este trabalho pretende preencher e como ele é relevante ao cenário atual.

Tabela 3 – Comparativo de trabalhos relacionados.

Biblioteca	Adaptação para Uso	Grid/Cluster	Seleção Automática	Métrica Computação	Métrica Comunicação	Métricas Combinadas	Custo de Migração	Suporte a BSP	Sup. a Dinamicidade	Sup. Heterogeneidade	Suporte a Migração
jMigBSP/MigBSP	Não	Todos	Não*	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim
HAMA	HDFS	Cluster	Não	Não	Não	Não	Não	Não	Não	Sim	Sim
PUB	Não	Todos	Sim	Não	Não	Sim	Não	Sim	Sim	Sim	Sim
MulticoreBSP	Sim	Não	NsA***	Não	Não	Não	NsA***	Sim	Não**	Não**	Não
Mizan	Não	Cluster	Sim	Não	Não	Sim	Não	Sim	Sim	Sim	Sim
BSPCloud	Não	Cluster	Não	Não	Não	Sim	Não	Sim	Sim	Não	Sim
DistPM	Não	Todos	Sim	Não	Não	Sim	Não	Não	Sim	Sim	Sim
Pregel.NET	Sim	Cloud	Sim	Não	Não	Não	Não	Sim	Sim	Sim	Sim
CPU-GPU	Sim	Cluster	Sim	Não	Não	Não	Não	Sim	Sim	Sim	Sim

Fonte: Elaborado pelo Autor.

* Depende da definição do usuário no inicio da aplicação;

** Não informado;

*** Não se Aplica

- **Seleção Automática de Processos**: Esta é a lacuna a ser preenchida por esta dissertação.

Lista de itens e definições que compõem a Tabela 3:

1. **Adaptação para o Uso**: Informa a necessidade de adaptar a aplicação que se deseja balancear à biblioteca/sistema de reescalonamento de processos/tarefas, se é necessário efetuar alterações de código no programa original;

2. **Ambiente (Grade ou Aglomerado)**: Ambiente em que a aplicação será utilizada, um aglomerado (*cluster*) ou uma grade (*grid*) com inúmeras máquinas heterogêneas;

3. **Seleção Automática de Processos**: Define a escolha dos processos por parte do sistema, se é necessário uma pré definição do usuário no início da aplicação, ou se ela tem total controle sobre escolha dos processo;

4. **Métrica Computação**: Se a aplicação utiliza a análise do processamento como métrica para definir processos a serem migrados;

5. **Métrica Comunicação**: Semelhante à Computação, se a aplicação analisa a comunicação entre processos para definir quem será migrado;

6. **Métricas Combinadas**: Define se o sistema analisa as métricas de computação e comunicação, mas não de forma separada, e sim conjunta, utilizando cada uma em um momento distinto;

7. **Métrica Custo de Migração**: Se analisa o custo para um processo/tarefa ser migrado, ou apenas analisa se ele deve ser migrado;

8. **Suporte a BSP**: Se suporta aplicações desenvolvidas utilizando o modelo BSP;

9. **Suporte à Dinamicidade**: Aceita a dinamicidade das cargas, aplicações e ambientes que possam vir a ser utilizados pela aplicação;

10. **Suporte a Heterogeneidade**: Suporta aglomerados/grades com recursos heterogêneos ou apenas utiliza ambientes homogêneos;

11. **Suporte à Migração**: Se a aplicação apenas analisa o sistema identificando possíveis candidatos à migração mas não implementa este recurso.

Conforme os trabalhos que foram apresentados e a Tabela 3, é possível verificar a variedade de possibilidades que cada biblioteca/modelo tenta preencher, a única lacuna que o MigBSP não preenche de forma eficiente é a seleção automática de processos. Porém, atualmente ele é o único a combinar três métricas na seleção de processos para migração, considerando o $PM(i,j)$, do processo lento *i* para um conjunto rápido *j* e que tal processo tenha um grau de comunicação com os processos que estão em *j*, e que este possua um baixo custo de migração.

Algumas bibliotecas/modelos, como PUB (BONORDEN et al., 2003), Mizan (KHAYYAT et al., 2013) e DistPM (LI; LAN, 2005), permitem a escolha automática de processos, ou seja,

não necessitam de uma análise ou conhecimento prévio do ambiente e características da aplicação que será executada pelo usuário. Elas automaticamente analisam o ambiente e sugerem ou migram os processos. Este recurso é muito útil para o desenvolvimento de aplicações de grande porte, permitindo que ela seja eficiente e prática. Entretanto, por exemplo a biblioteca PUB, que consegue definir o processo automaticamente, não leva em conta o sobrecusto de migração de um processo. Em geral, cada trabalho foca em resolver uma parte de um problema,

Dentre as características apresentadas, se verifica que seleção automática de processos é a que implica em uma maior complexidade ao sistema. Conforme vimos acima, as três bibliotecas/modelos que permitem a migração automática (PUB, Mizan e DistPM), pecam em outros itens, como avaliação do sobrecusto de migração e análise de métricas, para formarem um sistema completo. Desta forma, se verifica que o MigBSP é o único modelo/biblioteca que utiliza diversos recursos, como: análise de métricas individualmente e de forma combinada, analise do sobrecusto de migração de um processo para outro nodo menos carregado, avalia as últimas execuções de um processo para entender se a migração nesta *superstep* não irá afetar toda a aplicação na próxima. Porém, ele depende de uma intervenção manual no inicio da execução, e de conhecimento prévio da arquitetura e da aplicação a ser executada.

As duas heurísticas aqui propostas procuram preencher a **lacuna** da Seleção Automática de Processos, utilizando todos os recursos do MigBSP, que já supera muitas bibliotecas/modelos na questão de recursos e ferramentas de análise. E ainda oferecer a seleção automática de processos, sem intervenção manual. Oferecendo um modelo mais completo e com grande potencial de ser aplicado a diversas aplicações paralelas.

3.12 Considerações Parciais

O capítulo apresentou uma descrição sobre diversas bibliotecas/modelos que utilizam e resolvem problemas de balanceamento de carga, escalonamento e migração de processos de diversas formas. Alguns seguem o modelo BSP, outros utilizam padrões consolidados como o HDFS.

De forma geral é possível afirmar que o jMigBSP e MigBSP são os modelos que mais recursos oferecem, como migração de processos, análise individual e combinada das métricas de computação e comunicação. Além de utilizar o modelo BSP, também avalia o sobrecusto de migração de um processo e suporta a heterogeneidade e dinamicidade, apenas não oferece a seleção automática e eficiente de processos.

Com relação a seleção automática de processos, as bibliotecas PUB, Mizan e DistPM oferecem este recurso, mas não resolvem outras questões, como avaliar o sobrecusto de migração de um processo. Também podemos falar sobre o BSPCloud, com um foco um pouco diferente deste trabalho, mas oferece de forma eficiente o balanceamento de carga para computação na nuvem, mas também sem avaliar o sobrecusto de migração.

A biblioteca HAMA tem outro objetivo, ela trabalha sob a camada do *Hadoop*, utilizando o

HDFS, ela não analisa as métricas do ambiente, ela considera um ambiente homogêneo a partir do Hadoop, mas permite a migração de processos em sistemas que o utilizem. Já o modelo MulticoreBSP é o com mais limitações, por ser um modelo inicial, ele apenas suporta o BSP, mas não implementa nenhum outro recurso considerado essencial, o foco dele é permitir a análise de balanceamento de carga de forma fácil e simples, utilizando Java.

O pregel.NET é uma implementação focada em nuvem, e que depende de uma grande escalabilidade do ambiente. Porém, obteve um bom desempenho neste tipo de ambiente. A heurística avaliada referente ao *cluster* CPU-GPU tem uma objetivo muito interessante, permitir que uma aplicação seja balanceada durante a execução da *superstep* e não apenas a cada barreira. Porém ela depende da implementação de código da aplicação em CUDA[3] para GPU, o que pode tornar mais complexo a sua utilização.

Após a análise das diversas bibliotecas/modelos, verifica-se que todos tentam resolver algum problema em específico ou com um propósito determinado. O modelo MigBSP é o que possui a maior gama de recursos. Por isto, o trabalho aqui proposto irá herdar as características do MigBSP, implementando duas heurísticas de seleção automática de processos candidatos à migração de forma eficiente, oferecendo dois modelos com todas as características aqui analisadas.

[3]CUDA (*Compute Unified Device Architecture*) é uma linguagem de programação para computação paralela desenvolvida pela NVIDIA para utilização em placas gráficas.

4 MIGCUBE E MIGHULL: HEURÍSTICAS PARA SELEÇÃO AUTOMÁTICA DE PROCESSOS

Todas as informações previamente apresentadas nesta dissertação servem como base para entender as decisões tomadas para o desenvolvimento deste trabalho. Utilizamos como base o MigBSP, um modelo de reescalonamento de processos desenvolvido por Righi et al. (2009). Ele é um modelo bem definido que permite o reescalonamento de processos dentro de sistemas distribuídos em uma arquitetura híbrida em ambientes dinâmicos. Com ele, aplicações que seguem o modelo BSP conseguem obter melhores resultados com o balanceamento de carga que ele fornece. Entretanto, atualmente o MigBSP possui duas heurísticas, uma que seleciona sempre o processo com maior PM, e outra que seleciona um percentual de processos com base no maior PM, ou seja, requer uma configuração prévia de um percentual de migração no início da execução da aplicação, para definir qual o nível de intrusividade do modelo. Para isso, é necessário que o usuário possua conhecimento prévio da arquitetura para definir este parâmetro.

4.1 Características Comuns

Na busca de um balanceamento de carga eficiente, reduzindo o tempo de execução de uma aplicação em ambiente distribuído, esta dissertação apresenta duas novas heurísticas para o MigBSP. A ideia das duas novas heurísticas, MigCube e MigHull, é permitir uma melhor assertividade na seleção de processos para migração em ambientes distribuídos.

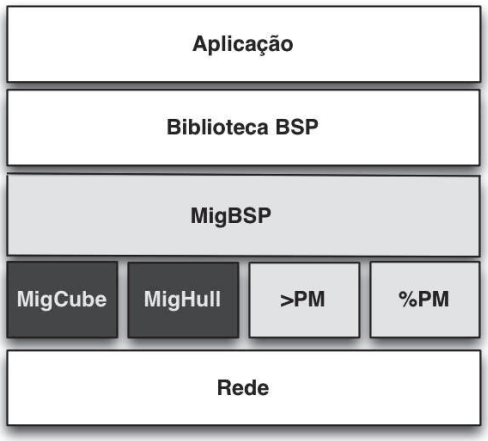
Ambas as heurísticas seguem utilizando os dados fornecidos pelo MigBSP, como o Potencial de Migração (PM) e métricas de computação, comunicação e memória de cada processo. Com base nestes dados, elas utilizam o espaço tridimensional para avaliar os processos em execução no ambiente. As decisões de projeto das heurísticas seguem as já definidas pelo MigBSP:

- Global, em todo ambiente de grade; Local, no aglomerado e de forma Híbrida, trocando informações entre estas duas arquiteturas;
- Escalonamento dinâmico, as informações para o reescalonamento são coletadas durante a execução;
- A análise de migração é efetuada pela coleta de informações dos processos, não pelo balanceamento dos recursos;
- As heurísticas foram desenvolvidas para programas que utilizem o modelo BSP.

A Figura 24 mostra a estrutura de camadas do modelo, e como as duas novas heurísticas se relacionam com ele. Elas trabalham dentro da estrutura atual do modelo, sendo executadas apenas dentro da chamada para migração que pode ocorrer na barreira. Estas chamadas não

ocorrem em todas as *supersteps*, elas dependem do número de execuções dos processos, e são impactadas pela migração ou não durante uma *superstep*.

Figura 24 – Nova Estrutura de Camadas do Modelo MigBSP.



Fonte: Elaborado pelo Autor.

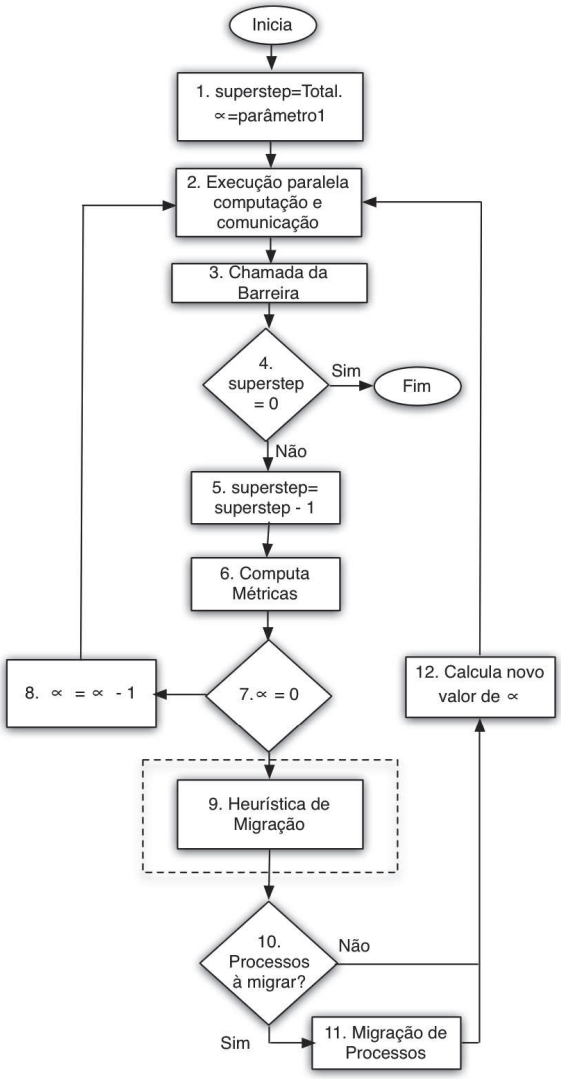
Se considera que o modelo MigBSP permite a seleção de processos candidatos à migração utilizando heurísticas com base em informações parametrizadas no início da execução do programa. Portanto, o desafio das novas heurísticas é efetuar uma melhor análise da arquitetura em que a aplicação está sendo executada. Desta forma, selecionando automaticamente os processos que melhor beneficiarão a todo o sistema.

Conforme a Figura 25, as duas novas heurísticas se situam dentro do modelo, no momento em que a chamada para migração dentro da execução é solicitada. Porém, nem todas as *supersteps* irão executar o MigCube e MigHull. Apenas as que, dentro das definições do modelo, chegarem ao parâmetro α, referente ao número de *supersteps* sem migrações.

O Algoritmo 1 apresenta a visão da função $bsp()$, mostrando como o modelo MigBSP atualmente utiliza as três métricas básicas de um processo: (i) computação, (ii) comunicação e (ii) memória. Dados que são coletados durante a execução do processo em uma *superstep*. Caso o parâmetro α tenha sido alcançado pelo número de *supersteps*, é chamado o reescalonamento, onde o Potencial de Migração de cada processo é calculado em cada conjunto do Aglomerado Global. O Algoritmo 2 mostra o papel do gerente, ao receber as informações e repassar aos demais processos.

A coleta de informações de cada processo ocorre durante a sua execução. Ao final dela, cada processo computa suas três métricas básicas (computação, comunicação e memória), e calcula o Potencial de Migração de cada processo em cada um dos demais aglomerados locais do conjunto global. Conforme a Figura 26, cada aglomerado local possui N processos em execução, um deles é escolhido como o gerente do aglomerado local. Este gerente conversa com todos os outros gerentes durante a chamada para reescalonamento. Cada gerente repassa aos outros uma lista ordenada, com o maior PM de cada processo. Após a troca de informações

Figura 25 – Arquitetura do Modelo, a caixa 9 representa a posição das heurísticas.



Fonte: Elaborado pelo Autor.

do gerentes, todos possuem a lista de todos os processos do ambiente e seus PM's. De posse desta lista, o modelo efetua a chamada para execução de uma das duas heurísticas de seleção de processos.

O MigBSP originalmente possui duas heurísticas de seleção, conforme mostram os Algoritmos 3 e 4, a primeira (Algoritmo 3) marca o processo com o maior PM de todo o conjunto para migração. Já a segunda heurística (Algoritmo 4), utiliza um parâmetro de percentual de migração definido no inicio da execução como um percentual do PM, ou seja, a partir do maior PM, todos os processos que tiverem um valor de PM dentro do percentual definido, a partir do maior, serão marcados para migrar.

As duas heurísticas fazem parte do modelo desenvolvido por Righi et al. (2009), e foram amplamente testadas, obtendo resultados bastante significativos em simulações de diversos ambientes distribuídos. Porém, elas são limitadas a um processo a ser migrado, e/ou à um percentual

Algoritmo 1: Modelo de um programa paralelo.

```
Para i = 0 Até Supersteps Faça
    computacao();
    comunicacao();
    barreira();
Fim Para
barreira() begin
    Se i = α Então
        computa_metrica_computacao();
        computa_metrica_comunicacao();
        computa_metrica_memoria();
        calcula_PM();
        send_to_Manager(PM, tempo_Supersteps);
        novo_α ← recv_from_Manager();
    Fim Se
end
```

Algoritmo 2: Modelo de chamada para reescalonamento.

```
while 1 Até Supersteps do
    lista_PM[n_processos] ← recv_from_BSP()
    send_to_Managers(PM[n_processos], tempo_Supersteps[n_processos]);
    calcula_novo_α();
    Heuristicas_de_Selecao();   %%execute Heuristica de selecao escolhida
    Para i = 0 Até n_processos Faça
        Se PID_manager[i] = PID_atual Então
            Se processo[i].migrate = true Então
                migra_processo();
                atualiza_novo_manager();
            Fim Se
        Fim Se
    Fim Para
    Para i = 0 Até n_processos Faça
        Se PID_manager[i] = PID_atual Então
            send_to_BSP[i](novo_α);
        Fim Se
    Fim Para
end
```

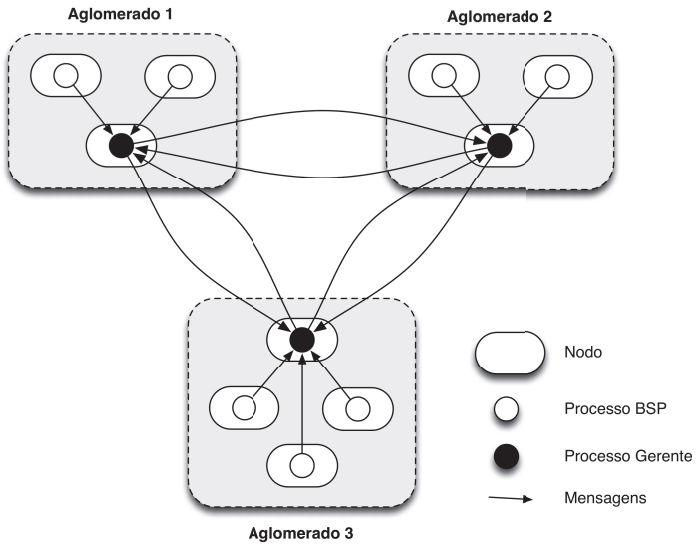
Algoritmo 3: Algoritmo da Heurística de Seleção de Um Processo.

```
begin
    recebe_lista_PMs_ordenada;
    migrar(lista[1]);   %%marca o primeiro para migracao
end
```

previamente definido.

Devido as estas limitações, esta dissertação apresenta a seguir duas novas heurísticas, Mig-

Figura 26 – Representação do ambiente distribuído para execução do modelo.



Fonte: Elaborado pelo Autor.

Algoritmo 4: Algoritmo da Heurística de Seleção pelo Percentual.

begin
  $recebe\_lista\_PMs\_ordenada$;
  $menor\_valor\_PM = lista[1] \times \alpha$;
  Para $x =$ **1 Até** $Processos$ **Faça**
    Se $lista[x] > menor\_valor\_PM$ **Então**
      $migrar(lista[x])$; $\%\%marca\ o\ processo\ para\ migracao$
    Fim Se
  Fim Para
end

Algoritmo 5: Algoritmo da Heurística MigCube.

begin
  $recebe\_lista\_PMs\_ordenada$;
  Para $i =$ **2 Até** $Processos$ **Faça**
    $media \leftarrow calcula\_distancia(ponto[1], ponto[i])$
  Fim Para
  $\triangle_{cubo} \leftarrow \frac{media}{(Processos-1)}$;
  Para $i =$ **1 Até** $Processos$ **Faça**
    Se $(lista[1] - \triangle_{cubo}) \leq lista[i] \geq (lista[1] + \triangle_{cubo})$ **Então**
      $\%\%em\ x,\ y\ e\ z$
      $migrar(lista[i])$; $\%\%marca\ o\ processo\ para\ migracao$
    Fim Se
  Fim Para
end

Algoritmo 6: Algoritmo da Heurística MigHull.

```
begin
    recebe_lista_PMs_ordenada;
    desvioX ← lista.X;   %%para y e z tambem
    Para i = 1 Até Processos Faça
        %%calcula para cada plano : x − y, x − z e y − z
        distancia ← calcula_distancia(menorX, maiorX, ponto[i])
        Se distancia > 0 and distancia < desvio Então
            migrar(lista[i]);   %%marca o processo para migracao
        Fim Se
        distancia ← calcula_distancia(maiorX, menorX, ponto[i])
        Se distancia > 0 and distancia < desvio Então
            migrar(lista[i]);   %%marca o processo para migracao
        Fim Se
    Fim Para
end
```

Cube e MigHull. Os Algoritmos 5 e 6 apresentam brevemente o funcionamento delas comparado aos Algoritmos 3 e 4. Ambas utilizam toda a estrutura do MigBSP e são facilmente adaptáveis a qualquer sistema real. Elas demandam um maior processamento para seleção de processos. Porém o maior sobrecusto imposto a aplicação deve ser desconsiderado quando avaliarmos os resultados no ganho do tempo de execução.

4.2 Heurística de Seleção MigCube

A principal ideia da heurística MigCube é a utilização do espaço tridimensional para selecionar os processos para migração. Posicionando cada processo como um ponto em um espaço definido pelas coordenadas X, Y e Z. Desta forma, relacionado-os com afinidade de informações.

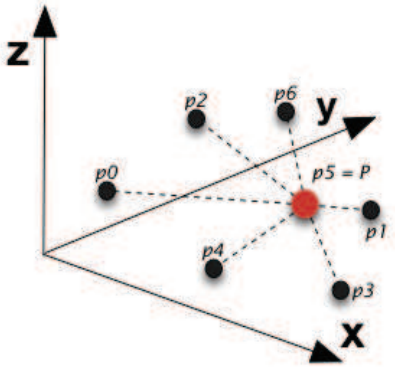
A questão é como posicionar estes processos e quais informações podem ser utilizadas. A resposta são as métricas básicas já coletadas pelo MigBSP, computação, comunicação e memória. Seguindo toda sua estrutura, quando cada processo passa suas informações para o seu Gerente, ele repasse aos outros gerentes os PM's de cada processo. No gerente, ele calcula o PM de cada processo e repassa aos outros gerentes todos os PM's de seus processos, neste momento, passamos também as três métricas coletadas durante a execução.

Cada gerente repassa a lista com informações de seus processos para os demais, ao final, um é escolhido para executar a heurística de migração. Neste momento, o gerente repassa a lista de PM's e demais informações para a heurística selecionada. No caso da MigCube, ela considera cada métrica, computação, comunicação e memória, respectivamente como uma coordenada X, Y e Z no espaço tridimensional. E desta forma, posicionando os processos de acordo com os três eixos básicos de uma plano cartesiano.

Conforme a Figura 27, os pontos que representam os sete processos (deste exemplo) estão

dispostos de acordo com o valor de cada métrica conforme um eixo: (i) computação como X, (ii) comunicação como Y e (iii) memória como Z. Agora, com o espaço posicionado, é possível selecionar um processo como o centro do nosso cubo. Para isso, será utilizado o processo que possui o maior PM.

Figura 27 – Representação do processo de maior PM P e os demais.



Fonte: Elaborado pelo Autor.

Na execução do modelo, ele repassa a heurística de seleção uma lista ordenada com os dados de cada processo. Na Figura 27 verificamos o ponto $p5$ como o de maior PM. Na lista ordenada recebida, ele é o processo na posição 1 da lista. A partir dele, será calculada a distância dos demais processos até ele no espaço tridimensional, utilizando a equação (4.1).

$$D = \sqrt[2]{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{4.1}$$

Com o somatório destas distâncias, será calculado a média das distâncias de todos os processos até o processo de maior PM, que a partir de agora será definido como P. Para isto será utilizado a equação (4.2), ela mostra que será somado a distância de cada ponto $p(n)$ até o ponto P, depois dividido o somatório destas distâncias pelo número de processos n menos P.

$$\triangle_{cubo} = \frac{1}{(n-1)} \sum_{i=2}^{n} p(i)_{distD} P \tag{4.2}$$

Esta equação (4.2) fornece o valor da medida de metade do lado de um quadrado, a partir de agora definido como $\triangle_{cubo}$. Com as coordenadas X, Y e Z do processo P, adicionamos o $\triangle_{cubo}$ em cada coordenada para o valor maior, e descontamos $\triangle_{cubo}$ para os valores menores, conforme o Algoritmo 7.

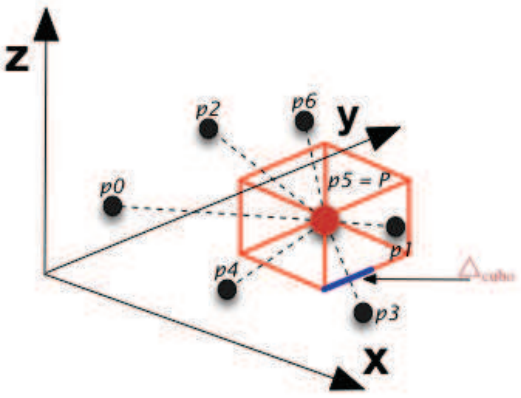
A Figura 28 mostra a montagem do cubo, o valor $\triangle_{cubo}$ é adicionado no valor de cada coordenada do ponto P para formar a maior, e descontado para formar as menores. Desta forma, criando um cubo com o centro no processo P, e cada processo que estiver dentro deste cubo, pela proximidade das métricas, será considerado um processo a ser migrado.

Após a heurística marcar quais processos devem ser migrados, o modelo segue com o pro-

Algoritmo 7: Algoritmo de Seleção dos processos a partir das coordenadas menores e maiores calculadas.

```
define_coordenadas();
begin
    menorX = pontoP.X − △cubo;
    maiorX = pontoP.X + △cubo;
    menorY = pontoP.Y − △cubo;
    maiorY = pontoP.Y + △cubo;
    menorZ = pontoP.Z − △cubo;
    maiorZ = pontoP.Z + △cubo;
end
seleciona_processos();
begin
    Para i = 2 Até Processos Faça
        Se processo[i].X ≥ menorX and processo[i].X ≤ maiorX Então
            Se processo[i].Y ≥ menorY and processo[i].Y ≤ maiorY Então
                Se processo[i].Z ≥ menorZ and processo[i].Z ≤ maiorZ Então
                    migrar(processo[i]);    %%marca o processo para migracao
                Fim Se
            Fim Se
        Fim Se
    Fim Para
end
```

Figura 28 – Representação da montagem do cubo a partir do processo P.



Fonte: Elaborado pelo Autor.

cessamento normal, migrando ou não os processos selecionados, de acordo com as definições do nodo destino para o processo.

Esta heurística sempre irá selecionar pelo menos um processo, o do centro. Sendo que as métricas dele sempre estarão dentro dos pontos maiores e menores calculados em cada eixo. A MigCube permite uma seleção de processos com base na relação de suas métricas e não no PM, dois processos que possuam PMs muito diferentes podem ser selecionados, desde que

suas métricas os aproximem no posicionamento no espaço tridimensional.

4.3 Heurística de Seleção MigHull

A heurística MigHull é uma adaptação da Envoltória Convexa, do inglês *Convex Hull*. O conceito de Envoltória Convexa já foi descrito na seção 2.6, mas a ideia básica é envolver todos os pontos em um plano bidimensional, de maneira a criar uma forma convexa de todo o perímetro dos pontos (REZENDE; STOLFI, 1994).

Segue-se a ideia das demais heurísticas, com base na estrutura do modelo MigBSP, utilizamos as métricas já definidas e as posicionamos em um espaço tridimensional. Referenciando cada métrica como uma coordenada, sendo: computação como X, comunicação como Y e memória como Z. Os cálculos necessários para esta heurística são efetuados apenas quando o número de execuções de *supersteps* alcançar o α pre-determinado.

Neste trabalho, a ideia é aplicar a envoltória convexa. Porém, devido a suas características, duas adaptações tiveram que ser efetuadas, (4.3.2) execução do espaço tridimensional em três planos bidimensionais, e (4.3.2) reduzir a abrangência do envolvimento dos pontos.

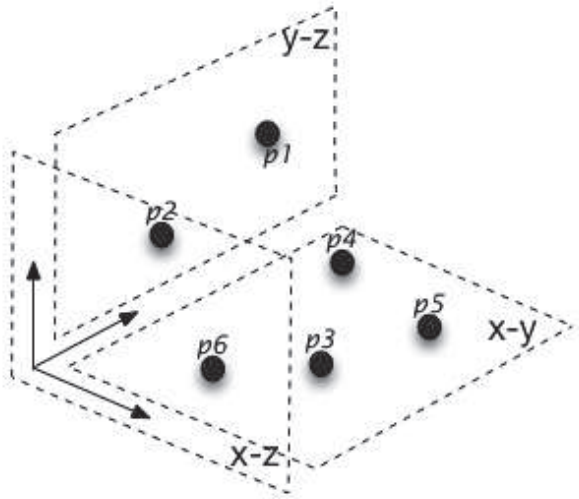
4.3.1 Espaço Tridimensional para Bidimensional

Envoltórias podem ser calculadas para dois e três planos. Porém, a complexidade de se calcular em dois planos é da ordem de $\bigcirc(n^2)$, e para três planos a complexidade tende a NP Difícil. Então, considerando que o objetivo desta dissertação é reduzir o custo de computação, não faria sentido agregar tamanha complexidade a execução. Sendo assim, calculamos a envoltória de forma bidimensional, mas como possuímos três coordenadas, executamos em três relações de coordenadas: x - y, x - z e y - z, como mostra a Figura 29; Desta forma consideramos como o cálculo de três planos, sempre utilizado a ideia de X e Y, independente de qual eixo real esteja sendo trabalhado. O processo a ser migrado, será definido pelo que estiver marcado para migrar nos três planos.

4.3.2 Abrangência da Envoltória

Considerando então, que esteja sendo trabalhado com planos bidimensionais, de coordenadas X e Y, posicionamos todos os processos neste plano. Se aplicássemos a envoltória convexa nativamente, ela selecionaria todos os pontos, o que não faria sentido. Está sendo utilizado o algoritmo *QuckHull* (BARBER; DOBKIN; HUHDANPAA, 1996) para o cálculo da envoltória. Ele trabalha de forma recursiva, e possui dois mecanismos principais, a definição da linha de separação X e a distância dos pontos a partir desta linha. Para seleção da linha, escolhemos os dois processos com maior valor de PM, e a partir da coordenada X deles (sendo X o eixo

Figura 29 – Separação dos três Planos (x-y, x-z e y-z) a partir do Espaço Tridimensional.



Fonte: Elaborado pelo Autor.

de trabalho para qualquer relação x-y, x-z e y-z), traçamos a linha. No cálculo da distância, ao invés de buscar os pontos mais distantes, selecionamos apenas os pontos em que o valor da distância for menor do que o desvio padrão das coordenadas para aquele plano.

Inicialmente, quando o modelo passa à heurística MigHull a lista de processos ordenada pelo PM, é calculado o Desvio Padrão para cada eixo utilizando a equação padrão (4.3), sendo que $\bar{X}$ é a média dos valores, e n o número de elementos. Depois, para cada plano de análise, x-y, x-z e y-z, é definido o maior valor de cada plano.

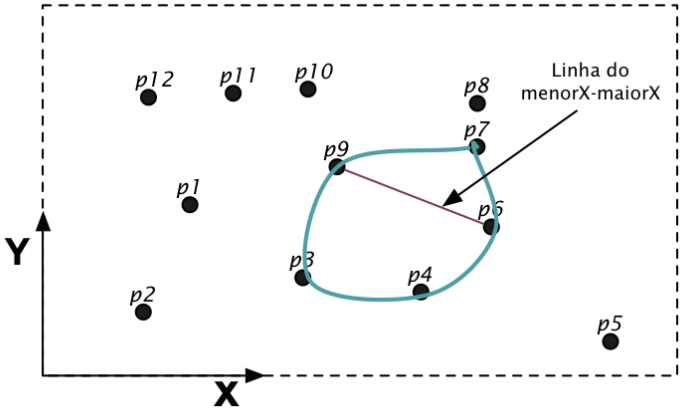
$$std\_dev = \sqrt{\frac{\sum (x - \bar{X})^2}{n}} \tag{4.3}$$

Com as três relações de coordenadas e o desvio padrão para cada uma delas, é calculado a envoltória adaptada. Selecionamos os dois processos no eixo X com maior valor de PM, consideramos eles como $menorX$ e $maiorX$, conforme mostra a Figura 30. Caso o valor da coordenada X do $menorX$ seja maior do que $maiorX$, os processos devem ser invertidos.

Algoritmo 8: Algoritmo para calcular a distância entre a linha $menorX$-$maiorX$ até um ponto P qualquer.

```
calcula_distancia();
begin
    linhaA = maiorX.X − menorX.X;
    linhaB = maiorX.Y − menorX.Y;
    pontoA = pontoP.X − menorX.X;
    pontoB = pontoP.Y − menorX.Y;

    return = (pontoB × linhaA) − (pontoA × linhaB);
end
```

Figura 30 – Representação da aplicação da Envoltória Convexa adaptada no plano.



Fonte: Elaborado pelo Autor.

Após estas definições, é possível calcular a distância de cada ponto a partir da linha $menorX$-$maiorX$. Para isto utilizamos o Algoritmo 8, sendo necessário informar o valor de X e Y de $menorX$ e $maiorX$, e também o X e Y do ponto a ser calculado a distância. O retorno deste algoritmo é um valor maior do que zero para pontos acima da linha $menorX$-$maiorX$, e negativos para pontos abaixo. Para calcular a distância dos pontos abaixo da linha, devemos inverter os pontos $menorX$-$maiorX$. Esta seria a divisão para a recursividade no QuickHull. Porém, como estamos adaptando este algoritmo, calculamos a distância para todos os pontos, das duas formas, com o $menorX$ e $maiorX$ normal e invertidos e verificamos se a distância de cada ponto é maior do que zero e menor do que o desvio padrão para aquele plano, conforme o Algoritmo 9.

Algoritmo 9: Cálculo do QuickHull adaptado ao MigHull.

```
seleciona_processos();
begin
    Para i = 1 Até Processos Faça
        distanciaA = calcula_distancia(menorX, maiorX, pontoP);
        distanciaB = calcula_distancia(maiorX, menorX, pontoP.X);

        Se distanciaA ≥ 0 and distanciaA ≤ std_dev_plano Então
            migrar_plano(processo[i]);   %%marca o processo neste plano
        Fim Se
        Se distanciaB ≥ 0 and distanciaB ≤ std_dev_plano Então
            migrar_plano(processo[i]);   %%marca o processo neste plano
        Fim Se
    Fim Para
end
```

Por fim, após executar os algoritmos 8 e 9 nas três relações de planos (x-y, x-z e y-z), temos uma marcação para cada processo em cada plano e verificamos, caso o processo possua

marcações para migrar nos três, o processo é marcado para migrar. A partir da definição de quais processos devem ser migrados, o modelo segue o fluxo normal, avaliando de acordo com o nodo destino a migração ou não do processo durante a barreira.

Esta heurística não necessariamente irá migrar os dois processos com maior PM definidos como $menorX$ e $maiorX$. Considerando que a distancia é calculada em relação as duas coordenadas do processo em cada plano (x-y, x-z e y-z). E em um determinado plano o processo pode ter o valor da distância muito diferente dos demais, o processo pode ser considerado em um, mas em outro não, vai depender da relação das métricas do processo. Desta forma, permitindo uma maior relação entre as características dos processos no ambiente em execução.

4.4 Análise dos Modelos MigCube e MigHull

Esta dissertação apresenta duas novas heurísticas propostas para o modelo MigBSP. Ambas tem uma ideia semelhante, porém com uma abordagem diferente cada. Elas posicionam os processos no espaço tridimensional a fim de melhor relaciona-los de acordo com suas métricas. MigCube trabalha diretamente no espaço tridimensional, pois calcula a distância entre pontos utilizando uma técnica para este espaço geométrico, e a partir disto monta uma figura geométrica em três dimensões que engloba os processos dentro do plano cartesiano do eixo de coordenadas X, Y e Z. Ela faz uma relação entre processos no espaço tridimensional, pois da forma como eles estão posicionados, e o calculo utilizado para encontrar a distância entre cada ponto e o ponto P. Faz com que trabalhem uma relação global entre eles. Também, devido a este método, o MigCube sempre irá migrar o processo P de maior PM.

Já a MigHull utiliza o posicionamento dos processos no espaço tridimensional, porém transforma estes dados, criando três planos bidimensionais com a relação de cada plano (x-y, x-z e y-z). O MigHull trabalha na variação de cada relacionamento, ao analisar (i) computação com comunicação (x-y), (ii) computação com memória (x-z) e (iii) comunicação com memória (y-z). Desta forma, força uma análise da variação de cada relação, estudando caso a caso o seu impacto, ao invés de uma análise global do sistema.

A heurística MigCube tende a migrar menos processos a cada *superstep* conforme a sua quantidade for aumentando. Pois como ele utiliza o valor médio das distâncias, quanto mais processos existirem no ambiente, maior será a divisão das distâncias, e menor a média e por consequência o tamanho do cubo. Porém, pelo posicionamento no espaço tridimensional, a disposição dos pontos acaba deixando próximos os processos que possuem métricas muito semelhantes. Desta forma, selecionando uma quantidade adequada de processos para migração.

Por outro lado, a heurística MigHull possui um comportamento menos linear, pelo fato de considerar o desvio padrão de cada eixo de coordenadas, ela acaba criando um limite para cada plano. Pois a variação dos valores na métrica de computação pode ser muito diferente dos de comunicação. E desta forma a distância entre a linha X dos processos com maiores PM e cada um dos demais processos pode variar bastante. Desta forma, esta análise plano a plano permite

uma melhor abordagem em uma simulação com uma maior quantidade de processos. Já que ela sempre irá selecionar os processos em cada relação, independente de uma amostragem global.

Ambas as heurísticas possuem seu modo particular de analisar dados. Porém, a técnica empregada nas duas visa a análise real do ambiente e seus processos, buscando definir a relação ideal entre migração e estabilidade para garantir o balanceamento da aplicação distribuída.

4.5 Considerações Parciais

Este capítulo apresentou a estrutura das heurísticas MigCube e MigHull aqui desenvolvidas. Como a base deste trabalho é o MigBSP de Righi et al. (2009). Elas seguem os mesmos princípios, como sendo *sender-initiated*, em que os nodos carregados iniciam o balanceamento, trabalhando de forma híbrida, local e global, e com base no modelo BSP. O objetivo destas duas heurísticas é a seleção automática de processos candidatos à migração em ambientes paralelos e distribuídos.

As duas heurísticas trabalham com base na análise das três métricas de um processo, a (i) computação, (ii) comunicação e (iii) memória. Estas métricas são utilizadas para calcular o Potencial de Migração (PM) de um processo. Segue sendo utilizado o PM para análise dos processos, mas também está sendo considerado cada uma das métricas como uma coordenada x, y e z em um plano tridimensional. Desta forma, no aglomerado local, cada heurística irá utilizar as métricas coletadas durante a execução para analisar estes dados e com base neles definir os processos a serem migrados.

O sistema executa normalmente dentro de uma *superstep*, ao chegar na barreira, ele verifica se o sistema deve ter balanceamento. Desta forma, chama o mecanismo de reescalonamento do MigBSP para computar as métricas e calcular o PM. Após, uma das heurísticas é iniciada.

O MigCube utiliza os processos posicionados no espaço tridimensional para calcular a distância entre o processo de maior PM e os demais, com estas distâncias é calculada a média, e com ela definido o tamanho do cubo. Este cubo serve como limite para definir quais processos estao dentro do cubo e serão selecionados para migração.

O MigHull utiliza a ideia de envoltória convexa, porém de forma adaptada, primeiramente os processos no espaço tridimensional são divididos em três planos com a relação entre cada coordenada x-y, x-z e y-z. Depois, ao invés de calcular toda a envoltória dos pontos, são calculados apenas as distancias entre os pontos e a linha formada pelos dois processos de maior PM definidos de acordo com a lista de processos informada a heurística.

Ambas as heurísticas trabalham a ideia de uma análise matemática do ambiente para definir os processos a serem migradas. O MigCube trabalha puramente no espaço tridimensional, já o MigHull se utiliza de duas adaptações deste ambiente.

5 AVALIAÇÃO DAS HEURÍSTICAS

Este capítulo apresenta as estratégias que foram utilizadas para desenvolver e validar as heurísticas MigCube e MigHull. Além disso, apresenta os resultados das heurísticas em simulações no SimGrid com aplicações BSP. Serão apresentados testes demonstrado os ganhos que elas oferecem na execução de aplicações distribuídos. Também é detalhado situações em que as heurísticas acrescentam um sobrecusto no tempo total de simulação das aplicações.

O objetivo de avaliar um sistema é verificar se ele cumpre os objetivos a que se propôs. Neste caso, está sendo utilizado o modelo MigBSP em sua estrutura original, utilizando suas duas heurísticas: (i) seleção de um processo com maior PM e (ii) que utiliza uma porcentagem definida no início da execução; e comparando com as duas heurísticas desenvolvidas.

5.1 Implementação

Na inicialização da execução da heurística é acionado a função $bsp()$ para cada processo e o $manager()$ de acordo com a arquitetura. A função $bsp()$ irá executar conforme a quantidade de *supersteps* estipulada. As três métricas são coletadas e enviadas à função $barrier()$, que por sua vez envia para a função $rescheduling()$, que computa os dados e gera o PM. Caso o valor de α tenha sido alcançado, os dados de PM são enviados ao $manager()$ que aciona as heurísticas de seleção MigCube ou MigHull. Elas marcam os processos a serem migrados e retornam ao $manager()$ que efetua a migração, retornando ao $bsp()$ as informações sobre o estado dos processos para a computação recomeçar.

A estrutura inicial da simulação se baseia no modelo MigBSP, a aplicação é modelada na função $bsp()$. É necessário definir a quantidade de *supersteps* que se deseja simular em $\#define\ SUPERSTEPS$. Ao lançar o programa, ele inicia tantas *threads* de funções $bsp()$ e $manager()$ conforme a estrutura detalhada nos arquivos de Implantação e Plataforma previamente definidos na seção SimGrid (2.8).

Conforme mostra o código na Figura 31, a função $bsp()$ executa a aplicação modelada quantas vezes forem informadas no parâmetro $SUPERSTEPS$ utilizando um laço de repetição.

Ao acionar a barreira ($barrier()$), todas os dados coletados de computação, comunicação e memória são computados em cada processo e repassados para a função $rescheduling()$. Ela por sua vez finaliza os cálculos de comunicação para todos os processos e verifica se o parâmetro α foi alcançado, ou seja, se o número de *supersteps* para ocorrer uma migração chegou. Caso não tenha alcançado o α, a barreira encerra e uma nova *superstep* inicia. Caso o parâmetro tenha sido alcançado, a função de $rescheduling()$ inicia uma simulação do processo a ser migrado, sendo executado nos outros aglomerados da grade. Neste momento, cada processo calcula o impacto de cada uma das métricas (computação, comunicação e memória) do processo i sendo executado no aglomerado j. No processo seguinte ele verifica em qual aglomerado obteve o maior PM, ou seja, para qual aglomerado j seria melhor migrar o processo i.

Figura 31 – Trecho de código da função bsp().

```
bsp(){
 for (i=0;i<SUPERSTEPS;i++){
   //Modelagem da aplicacao simulada
   //Simulacao processamento
   //Simulacao comunicacao

   barrier(superstep);
 }
}
```

Fonte: Elaborado pelo Autor.

Na última etapa, o processo envia ao seu gerente o seu valor de PM, e assim, cada gerente local tem a informação dos valores de PM de todos os processos sob sua gerência. Toda esta estrutura de troca de informações entre gerentes já existe no MigBSP. Para o desenvolvimento deste trabalho foram necessárias algumas adaptações no MigBSP, a primeira se refere a troca de informações de cada processo e o seu gerente, apenas o PM, Identificador do processo, identificador do gerente e a *superstep* são enviados para o gerente. Para a utilização das métricas como coordenadas dos pontos/processos foi necessário incluir os valores das três métricas nas informações repassadas ao gerente, conforme mostra o código na Figura 32.

Figura 32 – Trecho de código da função rescheduling().

```
rescheduling(){
 largest = get_largest(data[i].pm,sets,i,superstep,
                       data[i].comp,data[i].comm,data[i].memm);

 task_send = MSG_task_create("sending", 0, 30, largest);
 sprintf(mailbox_send, "manager-%d", data[i].id_manager);

 //Envio das informacoes para o gerente
 MSG_task_send(task_send, mailbox_send);
```

Fonte: Elaborado pelo Autor.

O gerente recebe das informações de todos os processos e armazena em uma lista. Neste momento, ele aciona a função $verify\_migration()$ que ordena a lista do maior PM para o menor e passa para a função $migration\_heuristic()$. Esta função executa o que cada heurística de seleção de processos manda, conforme a que for escolhida para execução nesta simulação. Caso seja a (i) heurística de um processo, ela pega o o item zero da lista e marca para migrar, caso seja a (ii) do percentual, ela multiplica o valor do percentual informado no inicio da execução ao valor do PM do processo zero da lista e percorre todos os processos verificando qual possui

valor de PM maior do que o percentual do maior PM, e marca estes para migração. Desta forma, foram incluídas duas novas opções na lista de heurísticas de seleção do MigBSP:

- $SELECTION\_HEURISTIC\_MIGCUBE \Rightarrow cube\_calculate(lista\_ordenada)$;
- $SELECTION\_HEURISTIC\_MIGHULL \Rightarrow hull\_calculate(lista\_ordenada)$.

Caso aso elas sejam as escolhidas pelo usuário, elas acionam respectivamente as funções informações a frente de cada opção, passando a lista ordenada de processos e suas informações, inciando os cálculos de cada uma das novas heurísticas.

5.1.1 MigCube

No caso da heurística MigCube, inicialmente é percorrido todos os processos a partir do segundo processo da lista. Calculando a distância entre eles e o primeiro. Este valor é adicionado a uma variável. Ao final, o valor da variável é dividido pelo número de processos, menos o primeiro. Este valor médio é usado como valor de metade de um lado do cubo $\triangle_{cubo}$. Depois este valor é adicionado em cada valor de coordenada do primeiro processo da lista para formar a maior coordenada do cubo, e descontado das mesmas coordenadas para formar as menores, conforme mostra o código na Figura 33.

Após verificar a posição de todos os processos, a heurística MigCube encerra e retorna para o manager que se encarregará da tarefa de efetuar as migrações.

5.1.2 MigHull

A heurística MigHull tem um inicio semelhante ao MigCube, recebe uma lista ordenada e chama uma função para calcular o desvio padrão dos valores de cada eixo. Esta função retorna uma vetor com os três valores de desvio.

Depois, ela percorre a lista com cada processo, pega cada coordenada e cria três novas listas, ou planos. Contendo os pares de coordenadas, $plano\_xy$, $plano\_xz$ e $plano\_yz$. Esta é a primeira adaptação no algoritmo desta heurística. Também é gerado uma lista $result$ com três posições com estes mesmos nomes e um campo de identificador do processo. Esta lista será atualizada em cada calculo de lado, onde será informado se naquele lado o processo deve migrar. Em seguida, ele verifica para cada par de coordenadas o maior valor de desvio padrão retornado, o maior dos dois é considerado para aquele plano.

Após a preparação dos dados, a função genérica $getConvexHull()$ é chamada passando a lista de cada plano, um identificador informando planos 1, 2 ou 3, e também o maior desvio padrão para aquele plano. Nativamente a envoltória vai tentar localizar todos os pontos mais distantes da linha, formando uma convexa dos pontos. Como o objetivo aqui é delimitar uma

Figura 33 – Trecho de código da heurística MigCube.

```
cube_calculate(){

   min_x = lista[0].x - media;
   max_x = lista[0].x + media;
   min_y = lista[0].y - media;
   max_y = lista[0].y + media;
   min_z = lista[0].z - media;
   max_z = lista[0].z + media;

 //Verifica quais pontos estao dentro da area do cubo
 for(i=0; i< PROCESSES; i++){
   if(
     lista[i].x >= min_x &&
     lista[i].x <= max_x &&
     lista[i].y >= min_y &&
     lista[i].y <= max_y &&
     lista[i].z >= min_z &&
     lista[i].z <= max_z
    ){
      //Marca processo lista[i] para migracao
  }
 }
}
```

Fonte: Elaborado pelo Autor.

área que englobe os pontos que se relacionem, foi escolhido o desvio padrão, por melhor representar uma amostragem de dados ao invés da média, que poderia destoar muito devido a variações nas métricas.

A função $getConvexHull()$ é a segunda adaptação desta heurística. Ela foi inspirada no Algoritmo *QuickHull* (BARBER; DOBKIN; HUHDANPAA, 1996). Conforme a primeira adaptação, no sentido de separar as três coordenadas em três planos. Por ser esta função genérica, sempre que se falar em eixos x e y, está se generalizando, independente se os dados correspondem aos planos x-z ou y-z. O primeiro passo é definir os dois pontos no eixo X para traçar a linha que inicia a envoltória convexa. Para isto, são utilizados os dois primeiros processos da lista, índice zero e um, verificamos o menor e maior deles e os chamamos de $menorX$ e $maiorX$, respectivamente. Estes dois pontos traçam a linha que serve de base para a função do cálculo da distância.

A lista do plano é percorrida, primeiramente é criada uma variável vazia chamada $distancia$. Para cada ponto, a função $getPointDistance()$ é chamada e passados os valores X e Y do $menorX$-$maiorX$ e ponto ($lista[indice]$) a ser avaliado.

O código na Figura 34 representa a função do calculo da distância, que retorna um valor positivo caso o ponto esteja acima da linha do $menorX$ e $maiorX$ na coordenada Y, con-

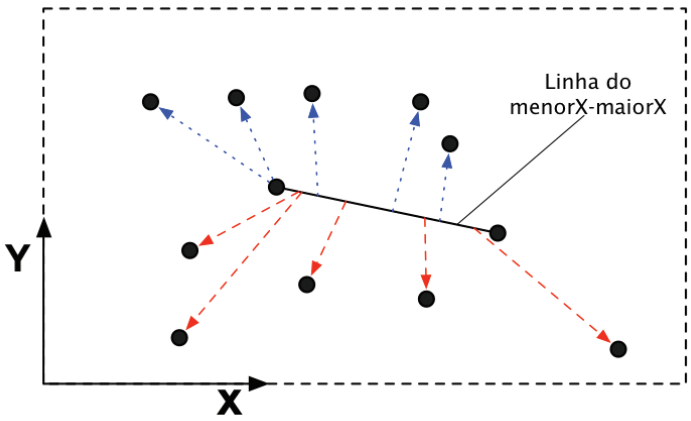
Figura 34 – Função do calculo da distância do *QuickHull*.

```
getPointDistance(menorX, maiorX, ponto){

      vLineA = maiorX->X - menorX->X;
      vLineB = maiorX->Y - menorX->Y;
      vPointA = ponto->X - menorX->X;
      vPointB = ponto->Y - menorX->Y;

      return (vPointB * vLineA) - (vPointA * vLineB);
}
```

Fonte: Elaborado pelo Autor.

forme mostra a Figura 35, onde as setas pontilhadas representam a distância positiva dos pontos acima da linha do $menorX$-$maiorX$, e as setas tracejadas representam os valores negativos dos pontos abaixo da linha.

Figura 35 – Calculo da distância no *QuickHull* de acordo com a linha do $menorX$-$maiorX$.



Fonte: Elaborado pelo Autor.

Depois de calcular a distância, a função retorna o valor calculado. Na $getConvexHull()$ é verificado se o valor for maior do que zero, e menor do que o desvio padrão para aquele plano. Caso seja, aquele ponto/processo é mercado como para migração naquele plano, ou seja, a lista $result$ é atualizada com a marcação para migrar.

Em seguida, é necessário calcular os pontos abaixo da linha do $menorX$ e $maiorX$, para isso, basta inverter a ordem dos valores da linha, passando $maiorX$ como menor, e $menorX$ como maior. Desta forma, as setas pontilhadas com valor negativo na Figura 35 será positivo. O valor da distância é novamente avaliado para verificar se é positivo, já que estamos calculando todos os pontos novamente, inclusive os que já foram marcados para migração. Também é

verificado se o novo valor da distância calculado é menor do que o desvio padrão.

Por fim, a lista $result$ é percorrida e verificada se cada processo consta o valor 1 nos três planos, então o processo é marcado para migrar. Após verificar todos os processos, a heurística MigHull encerra e o manager inicia sua tarefa de efetuar as migrações.

Após a execução das heurísticas, o gerente aciona a função $perform\_migration()$ que percorre cada processo. Caso ele esteja marcado para migração, ele verifica se o nodo destino j informado no PM tem condições de receber este novo processo. Caso seja possível, inicia migração para o novo nodo, altera a informação do processo dizendo que o seu gerente mudou e notifica o novo gerente de que ele recebeu um processo.

Por fim, os gerentes trocam mensagens informando que seu reescalonamento encerrou, retornando para a função de barreira do $bsp()$ e uma nova *superstep* é iniciada.

5.2 Metodologia

A estratégia de avaliação se dará pela simulação de aplicações reais e tidas como usuais para este tipo de trabalho. O objetivo é registrar o tempo de cada execução e comparar com as diversas situações e ponto de vista de cada heurística. Para mostrar o impacto e resultados de cada uma, três cenários serão analisados:

- **Cenário 1**: Aplicação paralela sem o modelo MigBSP;
- **Cenário 2**: Aplicação paralela com o modelo MigBSP mas sem migrações;
- **Cenário 3**: Aplicação paralela com o modelo MigBSP e com migrações.

A ideia destes cenários é, no primeiro, avaliar o tempo atual e conhecido da simulação da aplicação paralela executando em um ambiente distribuído, sem aplicar nenhuma sobrecarga de qualquer natureza; no segundo item, o objetivo é mostrar o impacto que a utilização do modelo de balanceamento de carga exerce sobre o tempo da aplicação. Um bom balanceador não deve alterar o tempo normal da aplicação. Porém, dependendo a situação, pode se aceitar o aumento do tempo na situação sem migrações, desde que quando a migração de processos seja ativada, se obtenha um melhor tempo. Por fim, a aplicação será executada efetuando migrações de acordo com cada heurística, mostrando o ganho, ou não no tempo total de execução da aplicação.

Para efetuar estes cenários, uma aplicação paralela amplamente conhecida será utilizada, a Dinâmica de Fluídos.

5.2.1 Aplicação Dinâmica de Fluídos

A dinâmica de fluídos é uma importante área de pesquisa na tecnologia (ANDERSON, 1995). Ela pode ser descrita como a simulação numérica de todos aqueles processos físicos

e/ou físico químicos que representam escoamento.

Para isso, iremos utilizar o método Lattice-Boltzmann de Schepke e Maillard (2007). Uma potente técnica para o cálculo de aplicações que podem ser caracterizadas como da área de dinâmica de fluídos. É um método computacional discreto que considera um volume tipico de fluídos compostos por uma coleção de partículas, e uma partícula é representada por uma função de distribuição para cada componente de fluído em cada ponto da grade.

Algoritmo 10: Algoritmo do Método da Dinâmica de Fluídos.

```
divide_dados(); Para i = 0 Até Supersteps Faça
    Processa();
    Se i = PID_processo Então
        receive_From_Manager(i);
    Senão
        send_To_Manager(i);
    Fim Se
    Barreira();
Fim Para
```

O Algoritmo 10 mostra o desenvolvimento dele em BSP (RIGHI; GRAEBIN; COSTA, 2014). O volume de dados é dividido em blocos contínuos de tamanhos iguais, na quantidade de processos a serem executados. Cada bloco é copiado e roda em um processo bsp independente. Após o processamento, no final de cada interação, as informações são repassadas aos demais processos e uma nova *superstep* inicia.

5.2.2 Escalonamento Inicial

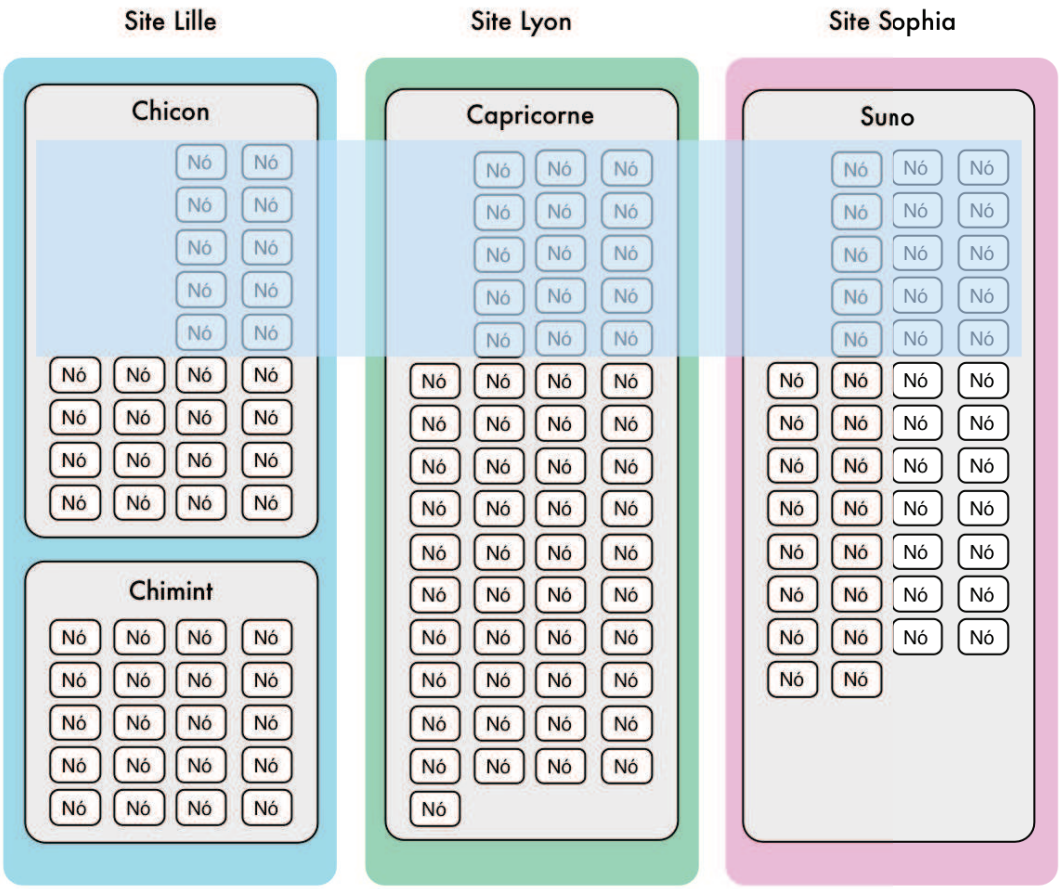
Para o desenvolvimento deste trabalho, está sendo utilizado o simulador SimGrid, conforme descrito anteriormente na seção 2.8, ele utiliza três arquivos de entrada, sendo (i) um arquivo de plataforma, onde deve ser descrito a estrutura, quais computadores disponíveis, capacidades, conectividade; e outro de (ii) implantação, onde é descrito a forma como os processos são destinados a cada recurso e quem são os gerentes de cada aglomerado local; e o (iii) arquivo do código da aplicação simulada.

Estes arquivos permitem definir uma série de características do ambiente, podendo simular inúmeros ambientes e grades. Entretanto, deve se considerar a forma como o escalonamento inicial dos processos nos nodos pode impactar consideravelmente no resultado da simulação. Em um caso em que muitos processos sejam atribuídos a poucos nodos, pode se demorar um tempo razoável até que o sistema estabilize, perdendo tempo na execução. Também existe a possibilidade de processos serem alocados de forma não proporcional a capacidade de processamento dos nodos, mesmo que em uma quantidade balanceada de processos, porém com recursos computacionais muito dispares. Todas estas questões podem impactar de forma positiva ou negativa

na aplicação em execução.

Para modelar a grade de forma similar a um ambiente real, utilizamos a arquitetura da grade Grid5000[1], os dados disponibilizados em um arquivo de plataforma define a utilização de 147 Nodos, distribuídos em 3 sites distintos, sendo que 2 sites possuem cada, um aglomerado, e o terceiro possui dois aglomerados. A Figura 36 mostra a distribuição dos nodos: no site *Lille* estão os aglomerados *Chicon* e *Chimint* com respectivamente 26 e 20 nodos; no site *Lyon* está o aglomerado *Capricorne* com 56 nodos; e por fim, no site *Sophia* está o aglomerado *Suno* com 45 nodos.

Figura 36 – Estrutura do Grid5000 e o modelo utilizado nestas simulações.



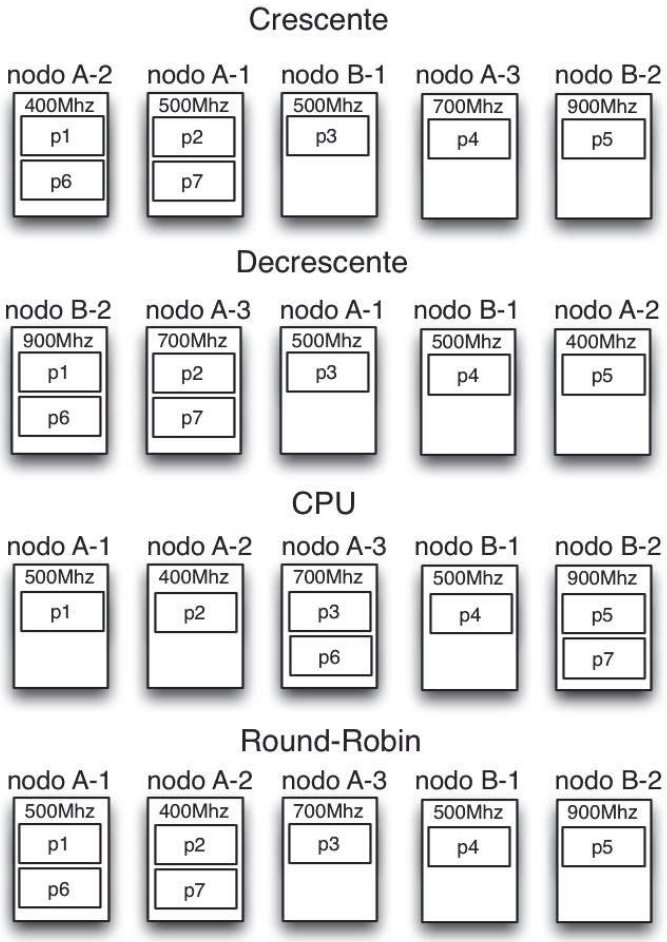
Fonte: Elaborado pelo Autor.

Para melhor adequar as simulações efetuadas, e considerando os 60 processos simulados, a grade foi adaptada de forma a auxiliar na análise dos resultados. Conforme a área em foco na Figura 36, foram utilizados os 3 sites, porém com apenas 10 nodos do site *Lille* no aglomerado *Chicon*, 15 nodos no site *Lyon* no aglomerado *Capricorne* e 15 nodos no site *Sophia* no aglomerado *Suno*. As definições de hardware e rede foram mantidas, apenas limitado a quantidade de nodos em cada aglomerado.

[1]Grid5000, dados disponíveis em: http://lists.gforge.inria.fr

Com base nas informações adaptados do Grid5000, foram modelados quatro escalonamentos iniciais, cada um tende a modificar o resultado da simulação pois impacta diretamente no inicio de toda a execução, a Figura 37 mostra a representação dos escalonamentos, a definição de cada escalonamento é descrita na sequencia:

Figura 37 – Representação dos Escalonamentos Iniciais.



Fonte: Elaborado pelo Autor.

- Crescente: Nodos são ordenados pelos de menor capacidade de processamento para o maior, independente de qual aglomerado está. Processos são distribuídos um a um em cada nodo, caso haja mais processos do que nodos, a distribuição recomeça, e novamente a partir do de menor capacidade. Neste escalonamento, existe o risco dos nodos de menor capacidade iniciarem muito carregados;

- Decrescente: A ideia inversa do Crescente, nodos ordenados na ordem decrescente de capacidade de processamento. Neste caso, os nodos com maiores capacidades podem receber mais processos, mas a execução não inicia balanceada;

- CPU: Os processos são alocados de forma a oferecer maior quantidade de processamento possível, um nodo pode receber mais processos e um outro ficar sem nenhum, caso sua capacidade seja muito superior, por exemplo: dois nodos, um de 900Mhz e outro de 300Mhz, se distribuirmos dois processos, é mais vantajoso alocar os dois processos no nodo de 900Mhz, pois cada um terá 450Mhz, ao invés de alocar um em cada nodo;

- *Round-Robin*: Ordena os nodos por seu nome/descrição dentro de cada aglomerado, não considera capacidade, apenas distribui os processos nos nodos, um a um.

A seção a seguir irá mostrar as informações e detalhes de todos os dados das simulações executadas de acordo com as definições descritas nas seções anteriores.

5.3 Resultados

Os resultados a seguir mostram uma série de testes e simulações efetuadas para validar o funcionamento e resultados das heurísticas apresentadas nesta dissertação. Simulações considerando o número de *supersteps*, os três tipos de cenários descritos anteriormente e com a aplicação modelada são apresentados.

As subseções MigCube (5.3.1) e MigHull (5.3.2) apresentam resultados de testes efetuados apenas com cada uma das heurísticas de forma separada, considerando os três cenários descritos anteriormente e a aplicação simulada. Na subseção 5.3.3 são apresentados comparativos entre as duas heurísticas, mostrando as diferenças e similaridades entre elas. Por fim, na seção 5.3.4, um comparativo entre as duas heurísticas MigCube e MigHull e as duas já existentes no MigBSP é apresentado.

As informações apresentadas nas próximas seções são resultado de diversas simulações efetuadas no SimGrid. Conforme as informações descritas nas seções anteriores, será utilizada a aplicação paralela de Dinâmica de Fluídos: Os arquivos de entrada tem como base o Grid5000, com quatro aglomerados e um total de 40 nodos. Serão utilizados os escalonamentos iniciais de processos descritos na seção anterior. Foram utilizados 60 processos para uma melhor análise dos resultados. As simulações foram efetuadas utilizando 20, 40, 60, 80 e 100 *supersteps*. E também com três níveis de α: 4, 8 e 16. Os testes foram efetuados conforme os três cenários descritos previamente na seção de Metodologia (5.2), sendo uma primeira execução apenas da aplicação depois com o modelo MigBSP sem migrações e por fim, com migrações.

5.3.1 Avaliação MigCube

A Tabela 4 mostra os experimentos de avaliação do MigCube. Nela são apresentados os resultados das simulações considerando as combinações de 60 processos, em 20, 40, 60, 80 e 100 *Supersteps*, para α: 4, 8 e 6; nos 3 cenários simulados e com os 4 escalonamentos iniciais.

Tabela 4 – MigCube: Avaliações da aplicação Dinâmica de Fluídos (tempo em segundos).

Crescente							
Superstep	Cenário 1	$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Cenário 2	Cenário 3	Cenário 2	Cenário 3	Cenário 2	Cenário 3
20	16,10	17,39	11,79	16,57	10,85	16,33	14,33
40	32,19	34,59	23,41	33,42	20,00	32,66	20,66
60	48,28	51,86	36,29	49,93	28,32	48,99	27,42
80	64,37	67,00	47,97	66,78	36,66	65,60	38,00
100	80,47	84,90	60,85	83,28	46,20	81,88	44,61

Decrescente							
Superstep	Cenário 1	$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Cenário 2	Cenário 3	Cenário 2	Cenário 3	Cenário 2	Cenário 3
20	16,27	16,68	11,67	16,21	11,50	16,39	14,41
40	32,94	33,09	21,47	32,13	21,36	31,87	21,99
60	48,82	49,50	32,41	47,87	29,91	47,35	29,88
80	65,09	65,00	42,21	63,78	38,58	63,03	40,82
100	81,37	81,40	53,14	79,53	48,40	78,54	47,67

CPU							
Superstep	Cenário 1	$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Cenário 2	Cenário 3	Cenário 2	Cenário 3	Cenário 2	Cenário 3
20	20,08	20,33	13,78	19,93	16,97	20,13	19,50
40	40,15	40,45	25,94	39,59	27,91	39,40	33,26
60	60,22	60,56	39,16	59,11	38,28	58,66	41,28
80	80,29	80,50	51,31	78,76	48,83	78,11	53,96
100	100,36	100,36	64,50	98,28	60,27	97,42	62,12

Round-Robin							
Superstep	Cenário 1	$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Cenário 2	Cenário 3	Cenário 2	Cenário 3	Cenário 2	Cenário 3
20	16,16	17,11	10,73	16,42	10,95	16,30	14,28
40	32,33	34,12	20,30	33,01	20,45	32,49	20,89
60	48,46	51,13	30,94	49,29	29,38	48,68	27,73
80	65,56	66,20	39,94	65,89	38,19	65,09	40,00
100	80,66	83,50	49,99	82,17	48,00	81,20	47,09

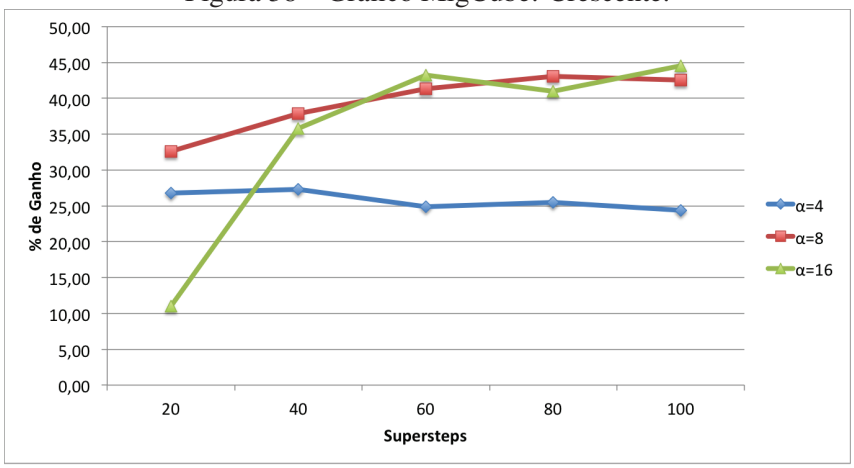
Fonte: Elaborado pelo Autor.

A Tabela 4 mostra os cenários de simulação, e como resultado, os tempos de cada experimento. Além da redução de tempo na execução apresentado quando comparados o Cenário 1 com o Cenário 3, devemos levar em consideração sempre o Cenário 2, que mostra o quanto a execução do MigCube adicionou de sobrecarga na aplicação, sem efetuar migrações. De forma geral, sempre ocorre um aumento no tempo do Cenário 2, esta sobrecarga pode ser considerada o custo de execução do MigCube. Se comparado com os resultados do Cenário 3, fica bastante visível os ganhos quanto as migrações são efetuadas. Também é interessante avaliar os campos marcados na tabela, nas simulações de 20 *supersteps*, com α de 16, o tempo reduz muito pouco,

pois apenas uma migração é efetuada, e desta forma o ambiente não é balanceado, isto ocorre para os quatro escalonamentos.

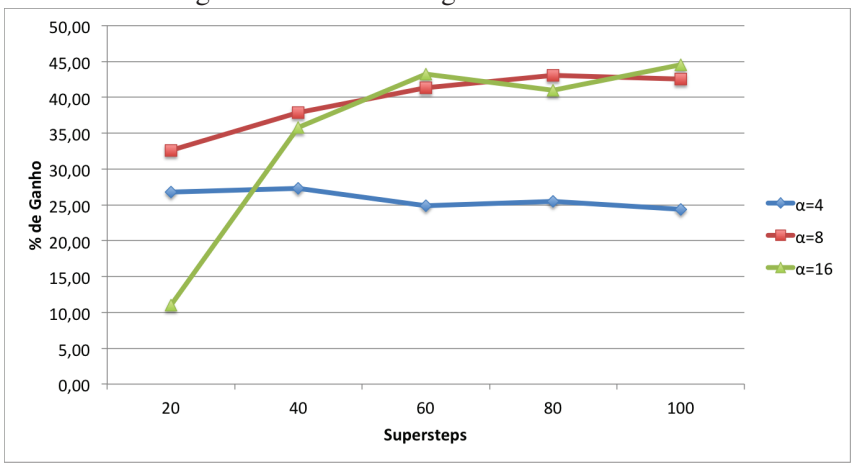
As figuras 38, 39, 40 e 41 a seguir mostram o percentual de ganho no tempo de execução das simulações. Elas foram calculadas considerando o quanto o valor do cenário três, em cada α ficou abaixo do cenário um, utilizando a fórmula $\frac{(Cenario\ 1 - Cenario\ 3) \times 100}{Cenario\ 1}$, considerando os três α avaliados. Elas estão ordenadas de acordo com os quatro escalonamentos iniciais testados.

Figura 38 – Gráfico MigCube: Crescente.



Fonte: Elaborado pelo Autor.

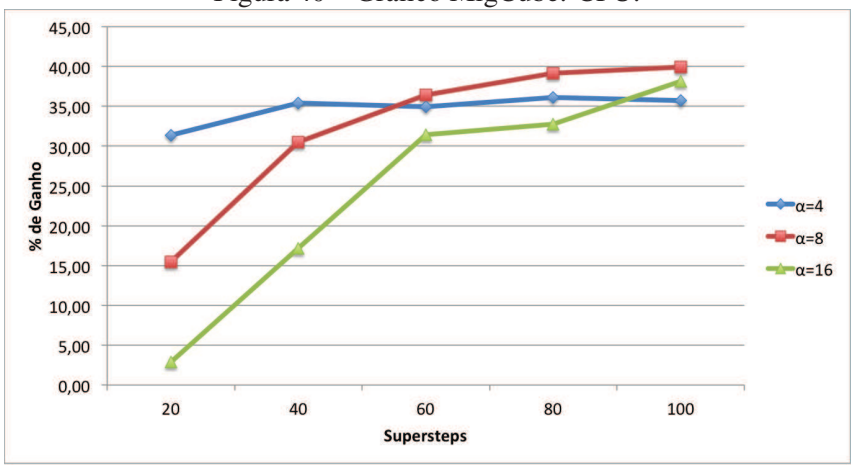
Figura 39 – Gráfico MigCube: Decrescente.



Fonte: Elaborado pelo Autor.

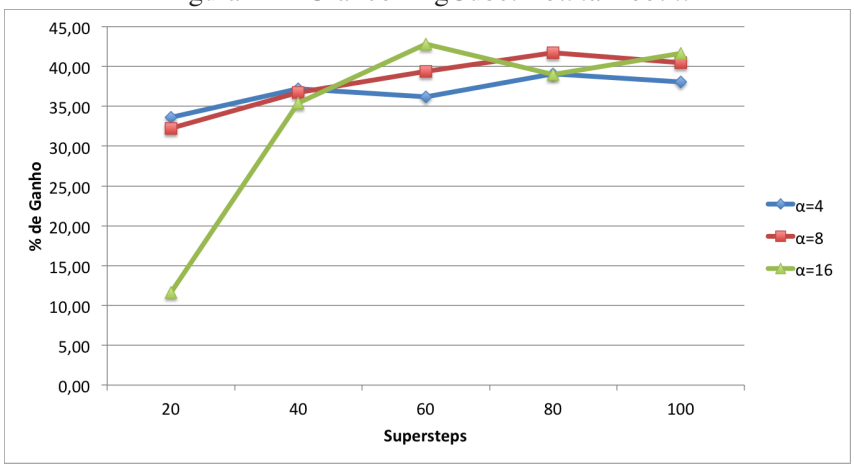
Analisando as informações das tabelas e os gráficos, é possível verificar que um $\alpha = 8$ é o mais estável no ganho de tempo, enquanto o $\alpha = 16$ tende a aumentar conforme o aumento do número de *supersteps* em execução. O $\alpha = 4$ apresentou valores consideravelmente disformes dos demais. Os melhores resultados alcançados com o $\alpha = 8$ se devem ao fato de ser um valor mediano no intervalo de migrações, nem tão baixo como 4, onde são executadas muitas migrações, ou tão alto quanto 16, em que se espera muito tempo para o ambiente se rebalancear.

Figura 40 – Gráfico MigCube: CPU.



Fonte: Elaborado pelo Autor.

Figura 41 – Gráfico MigCube: *Round-Robin*.



Fonte: Elaborado pelo Autor.

Já os escalonamentos testados afetam diretamente os α baixos, mas de forma geral é possível verificar que conforme forem aumentado parâmetros, *supersteps* e o α, existe uma tendência do ganho se manter alto e estável.

5.3.2 Avaliação MigHull

A Tabela 5 mostra os cenários de simulação, e como resultado, os tempos de cada experimento. Além da redução de tempo na execução apresentado quando comparados o Cenário 1 com o Cenário 3, devemos levar em consideração sempre o Cenário 2, que mostra o quanto a execução do MigHull adicionou de sobrecarga na aplicação, sem efetuar migrações. De forma geral, sempre ocorre um aumento no tempo do Cenário 2, esta sobrecarga pode ser considerada o custo de execução do MigHull, e se comparado com os resultados do Cenário 3, fica bastante visível os ganhos quanto as migrações são efetuadas.

Tabela 5 – MigHull: Avaliações da aplicação Dinâmica de Fluídos (tempo em segundos).

Crescente							
Superstep	Cenário 1	$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Cenário 2	Cenário 3	Cenário 2	Cenário 3	Cenário 2	Cenário 3
20	16,10	17,33	11,75	16,57	11,89	16,33	14,56
40	32,19	34,59	21,52	33,42	21,95	32,66	23,04
60	48,28	51,56	32,20	49,93	30,93	48,99	31,57
80	64,37	67,04	42,07	66,78	39,91	64,60	42,39
100	80,47	84,48	52,85	83,28	49,90	81,88	49,96

Decrescente							
Superstep	Cenário 1	$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Cenário 2	Cenário 3	Cenário 2	Cenário 3	Cenário 2	Cenário 3
20	16,27	16,68	12,15	16,21	11,99	16,39	14,31
40	32,54	33,09	21,53	32,13	22,15	31,87	23,32
60	48,82	49,50	32,27	47,87	31,12	47,35	32,12
80	65,09	65,10	41,97	63,78	40,21	63,09	42,71
100	81,37	81,70	53,11	79,53	50,22	78,54	51,37

CPU							
Superstep	Cenário 1	$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Cenário 2	Cenário 3	Cenário 2	Cenário 3	Cenário 2	Cenário 3
20	20,08	20,33	14,47	19,93	15,72	20,13	17,46
40	40,15	40,45	27,04	39,58	27,05	39,40	28,46
60	60,22	60,56	38,87	59,11	37,07	58,66	40,91
80	80,29	81,10	51,84	78,76	47,37	78,11	55,03
100	100,36	101,13	63,67	98,28	56,57	97,42	67,51

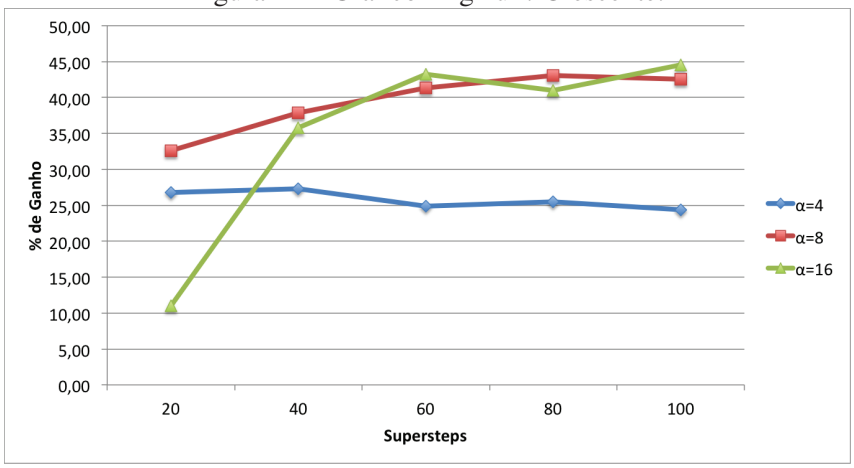
Round-Robin							
Superstep	Cenário 1	$\alpha = 4$		$\alpha = 8$		$\alpha = 16$	
		Cenário 2	Cenário 3	Cenário 2	Cenário 3	Cenário 2	Cenário 3
20	16,16	17,11	11,61	16,42	11,81	16,30	14,50
40	32,33	34,12	21,20	33,01	21,93	32,49	23,04
60	48,46	51,13	31,85	49,29	30,87	48,68	31,97
80	64,56	65,15	41,44	65,89	39,87	65,09	43,20
100	80,60	83,60	52,10	82,17	49,85	81,20	50,70

Fonte: Elaborado pelo Autor.

Na Tabela 5 é interessante avaliar que a mesma situação do α 16 que acontece nos testes do MigCube mostrados na Tabela 4 também ocorre. Neste caso, os campos marcados apresentam a grande diferença do tempo inicial da aplicação quando comparamos os quatro escalonamentos iniciais. Toda a aplicação, mesmo avaliando os três cenários, fica com um tempo mais elevado no escalonamento que considera a melhor distribuição de processamento de cada nodo (CPU). Isto ocorre pois mesmo que cada processo tenha uma fatia igual de processamento, as tarefas de cada um nem sempre são iguais. E como o ambiente inicia muito balanceado, demora ate ele redimensionar a carga e efetuar migrações mais assertivas.

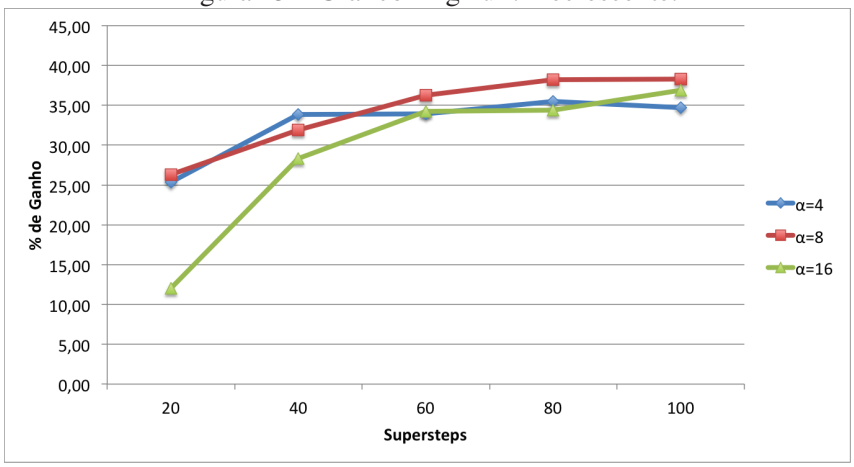
As figuras 42, 43, 44 e 45 a seguir mostram o percentual de ganho no tempo de execução das simulações, considerando os três α avaliados. Elas foram calculadas considerando o quanto o valor do cenário três, em cada α ficou abaixo do cenário um, utilizando a fórmula $\frac{(Cenario\ 3 - Cenario\ 1) \times 100}{Cenario\ 1}$, considerando os três α avaliados. Elas estão ordenadas de acordo com os quatro escalonamentos iniciais testados.

Figura 42 – Gráfico MigHull: Crescente.



Fonte: Elaborado pelo Autor.

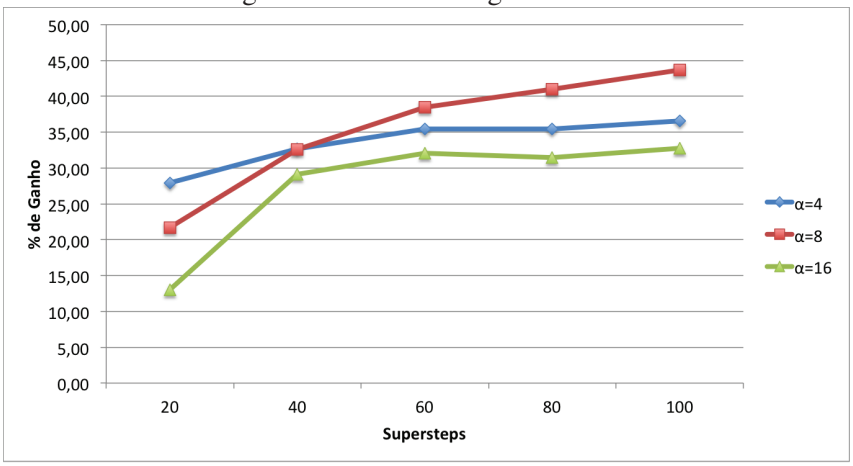
Figura 43 – Gráfico MigHull: Decrescente.



Fonte: Elaborado pelo Autor.

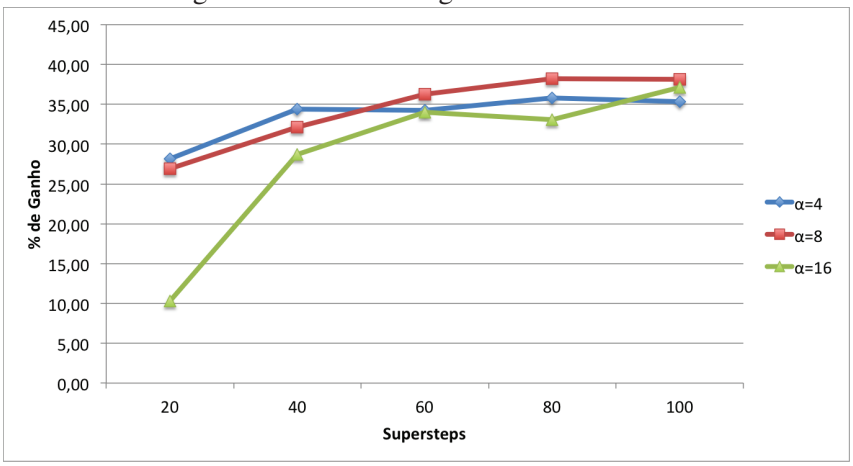
Analisando as informações das tabelas e os gráficos das simulações utilizando a heurística MigHull, é possível verificar que um $\alpha = 8$ é o mais estável no ganho de tempo, porém, diferente do MigCube, o $\alpha = 16$ não mostra uma tendência de aumento de ganho. O MigHull tende a ser mais complexo, e aumentar o custo de execução conforme for o aumento do número de aglomerados, pois para cada processo o MigBSP necessita verificar a possibilidade de migração em cada aglomerado diferente do atual do processo, ou seja, aumentando os custos de computação proporcionalmente ao número de aglomerados.

Figura 44 – Gráfico MigHull: CPU.



Fonte: Elaborado pelo Autor.

Figura 45 – Gráfico MigHull: *Round-Robin*.



Fonte: Elaborado pelo Autor.

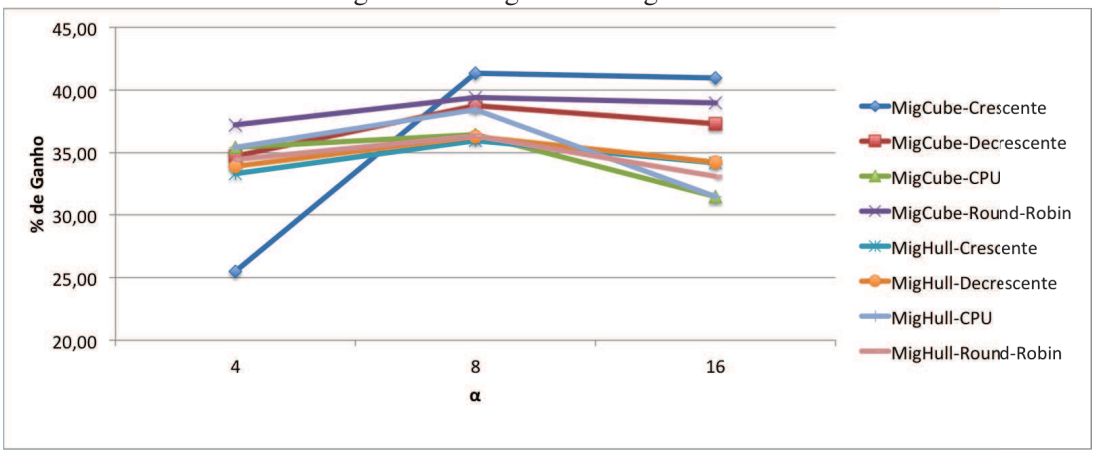
O $\alpha = 4$ também apresentou valores consideravelmente disformes dos demais. e a variação nos escalonamentos iniciais foi pouco perceptível, exceto no crescente, em que o $\alpha = 4$ tende a cair bastante o percentual de ganho.

5.3.3 Comparativo entre MigCube e MigHull

A Figura 46 mostra as médias dos ganhos em cada escalonamento inicial do MigCube e MigHull, foram consideradas as cinco simulações de *supersteps* e gerado um valor médio dos ganhos de tempo do Cenário 3 para o Cenário 1, para cada α em cada escalonamento inicial.

Os valores mostrados na Figura 46, apresentam uma proximidade nos dados médios, porém com o MigCube utilizando um escalonamento crescente, com α=4, o valor de ganho de 25% destoa bastante. Isto ocorre, pois como o α é baixo, e os processos estavam alocados em sua maioria dos processadores de menor capacidade, gerou uma grande pressão para migração em

Figura 46 – MigCube x MigHull x α.



Fonte: Elaborado pelo Autor.

curtos espaços de tempo, gerando muita comunicação e aproximando as métricas dos processos. Por isso foram selecionados para migração no MigCube.

Com os testes efetuados foi possível verificar principalmente que o MigHull apresentou um ganho de até 45% no tempo de execução, enquanto o MigCube até 42%. A parametrização que obteve os melhores resultados no MigCube foi com um $\alpha = 16$, 100 *supersteps* e tanto no escalonamento crescente quando decrescente. Já no MigHull, o melhor caso foi com um $\alpha = 16$, também com 100 *supersteps*, porém apenas no escalonamento crescente. Quando analisamos as Tabelas 4 e 5, ao comparar os resultados dos Cenários 1 e 2, podemos perceber no geral um pequeno aumento no tempo de execução da aplicação quando ativamos o modelo MigBSP, este tempo é em média 2% do tempo da aplicação sem o modelo. Se considerarmos apenas a execução do modelo sem migrações tendo um custo de 2% no tempo, é algo aceitável dependendo da exigência da aplicação. Porém, conforme descrito anteriormente, ao ativar as migrações os ganhos de tempo superam em muito o sobrecusto que o modelo emprega.

Para mostrar também como o sobrecusto maior das heurísticas é justificado, a Tabela 6 apresenta quantas migrações de processos cada heurística executou durante as simulações, ou seja, é possível avaliar o quanto o modelo analisou todo o ambiente, marcou processos para migração e os migrou.

Com os dados apresentados na Tabela 6 é possível verificar o volume de migrações efetuadas. Em casos como do MigCube, no escalonamento de CPU, onde migrou todos os processos, ou seja, apesar de apresentar ganhos no tempo, a heurística está sempre avaliando e selecionando uma grande quantidade de processos, no esforço de reduzir cada vez mais o tempo da execução.

O MigHull apresentou melhores resultados em grades maiores, devido a sua adaptação na divisão do espaço tridimensional em três planos bidimensionais, ela consegue avaliar a relação entre cada métrica e o seu impacto na aplicação. O MigCube, por utilizar um valor médio, acaba reduzindo o tamanho do seu cubo, conforme a quantidade de processos for aumentado. Por isso

Tabela 6 – Volume de migrações em cada *superstep* com um α de 8.

MigCube										
	supersteps									
	1	8	16	32	40	48	56	64	72	80
Crescente	37	17	5	40	19	19	5	40	35	19
Decrescente	37	10	16	43	2	16	16	41	43	2
CPU	38	12	45	2	59	58	50	60	58	60
Round-Robin	37	13	48	11	56	49	21	11	19	11
MigHull										
	supersteps									
	1	8	16	32	40	48	56	64	72	80
Crescente	27	36	23	36	23	23	36	23	36	23
Decrescente	13	54	32	21	18	10	11	21	14	35
CPU	18	19	13	13	10	10	10	50	10	50
Round-Robin	37	21	37	22	38	22	36	24	37	22

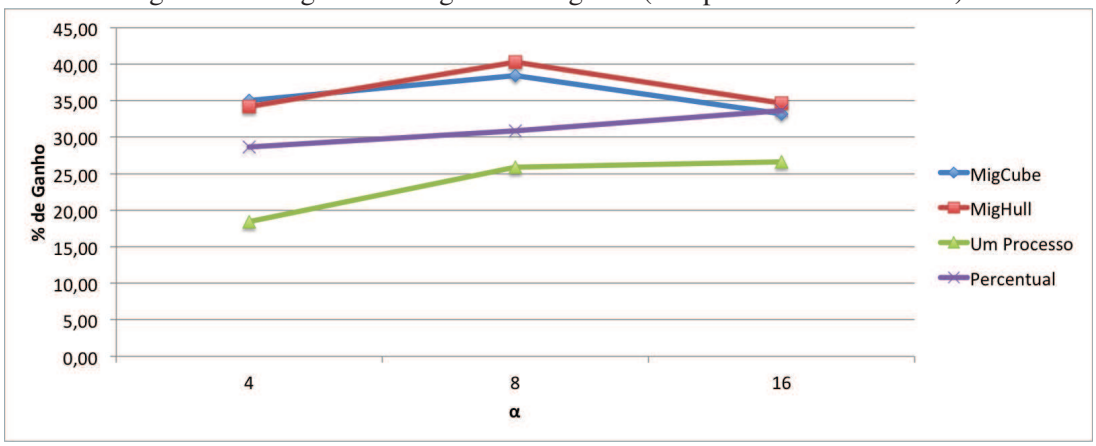
Fonte: Elaborado pelo Autor.

tende a selecionar menos processos, mas em uma situação de um α baixo, em que ocorram mais chamadas para reescalonamento, ele pode oferecer um resultado muito similar ao MigHull.

5.3.4 Comparativo entre MigCube e MigHull e Heurísticas do MigBSP

O objetivo desta dissertação é desenvolver o modelo de duas heurísticas que efetuam a seleção automática de processos candidatos a migração em aplicações paralelas desenvolvidas utilizando modelo BSP. A base para este trabalho foi o MigBSP, modelo de reescalonamento que já possui duas heurísticas de seleção, a de um processo e o Percentual do PM.

Figura 47 – MigCube x MigHull x MigBSP (Um processo e Percentual).



Fonte: Elaborado pelo Autor.

Entretanto, as heurísticas dependem de informações passadas no inicio da aplicação e de conhecimento prévio da aplicação pelo usuário. Possuem uma seleção limitada do número de

processos a serem migrados. Sendo assim, se em um comparativo entre as duas heurísticas existentes e as duas novas, o ganho de tempo for aproximado, a MigCube e MigHull alcançou seu objetivo, pois de forma automática conseguiu selecionar os processos que irão melhorar o desempenho de toda a aplicação, oferecendo um melhor ganho no tempo de execução de aplicações paralelas.

A Figura 47 apresenta o comparativo mencionado acima, os valores da Figura 46 foram agrupados para as heurísticas MigCube e MigHull, de forma a gerar uma linha com percentuais de ganho para os três α simulados. No caso do MigBSP, as heurísticas de um processo e percentual foram efetuadas quatro simulações de cada escalonamento inicial com os três α, os resultados foram agrupados e suas médias gerou uma linha para cada heurística do MigBSP. No gráfico apresentado é possível observar que a heurística do percentual se aproxima do MigCube/MigHull, principalmente com $\alpha = 16$, mas possui um resultado continuo e abaixo das novas heurísticas. Já a de seleção de Um Processo fica bem abaixo dos demais, devido a sua lógica de buscar apenas um processo por vez, acaba demorando mais tempo para mover mais processos, por isso com $\alpha = 16$ é onde se obtém menor ganho da heurística de Um processo.

6 CONCLUSÃO

O advento dos ambientes paralelos e distribuídos trouxe grandes vantagens para o processamento de grandes quantidades de tarefas e processos. Entretanto, para se obter um bom desempenho destes sistemas, é necessário que os programas rodando nestes ambientes aproveitem o máximo de sua capacidade. O modelo BSP desponta com uma forma prática e com bons resultados para a execução em ambientes como estes, por se tratar de um modelo que trabalha em etapas dentro de *supersteps*. O modelo não define como os processos devem ser mapeados para os recursos, ou de que forma e quando efetuar uma redistribuição destes processos, mas a forma como ele auxilia a análise de processos mais lentos dentro de uma *superstep* o torna um modelo eficiente para o reescalonamento de processos.

De forma a retirar a tarefa do desenvolvedor da aplicação a obter um melhor desempenho de sistemas distribuídos, diversas bibliotecas e modelos foram desenvolvidos para resolver este problema. Esta dissertação avaliou alguns trabalhos publicados, e a partir desta pesquisa foi constatado que as diversas bibliotecas/modelos disponíveis focam em alguns aspectos na migração de processos. Porém deixam a desejar em alguns quesitos, como avaliação de métricas e custos de migração. O modelo MigBSP é o que apresentou o maior número de ferramentas e recursos para o balanceamento de carga em aplicações paralelas. Entretanto ele apresenta uma limitação com relação a seleção automática de processos, dependendo de informações e conhecimento prévio do ambiente e aplicação em execução.

Esta dissertação apresenta MigCube e MigHull, duas heurísticas que permitem a seleção automática de processos candidatos à migração dentro de ambientes de grade computacional distribuída. Para isso, as heurísticas fazem uso do MigBSP, um modelo que oferece uma série de recursos para que aplicações que utilizem BSP obtenham um bom balanceamento no uso de recursos computacionais, utilizando o reescalonamento de processos.

A heurística MigCube utiliza ao máximo o espaço tridimensional ao posicionar os processos num plano cartesiano de três coordenadas. Com isto, ele calcula a distância dos pontos até um ponto principal selecionado pelo seu valor de Potencial de Migração (PM). A partir das distâncias, uma média é calculada e utilizada como medida de lado de um cubo centrado no processo com maior PM. A partir disto, processos que devido a sua distância do processo central, estejam dentro do cubo, serão considerados candidatos a migração. Desta forma, o MigCube tende a migrar processos que possuam um PM mais aproximado, devido a relação das métricas de cada processo, podendo apresentar uma deficiência em aplicações com um alto número de processos, porém tende a selecionar uma quantidade mais adequada em cada *superstep* para migração.

A heurística MigHull efetua duas adaptações da ideia básica de um espaço tridimensional e uma Envoltória Convexa. Devido a complexidade do calculo de uma envoltória em três dimensões, dispomos dos processos como pontos no plano cartesiano de três dimensões e separamos em três planos, considerando a relação de coordenadas: x-y, x-z e y-z. Criando três planos

bidimensionais em que a envoltória é calculada, neste momento surge a segunda adaptação, a envoltória por definição tenta envolver todos os pontos, mas como queremos encontrar apenas os que se relacionam, utilizamos apenas o calculo da distância do algoritmo QuickHull (BARBER; DOBKIN; HUHDANPAA, 1996), e utilizamos os dois pontos com maior valor de PM para considerar como linha central no eixo X do plano. A partir desta linha calculamos as distâncias, e os pontos que tiverem valor menor do que o desvio padrão dos valores de cada eixo, serão considerados candidatos a migração naquele plano, por fim, os processos que foram considerados candidatados nos três planos, serao considerados para migração na aplicação.

Sendo assim, MigHull possui um comportamento menos linear, pelo fato de considerar o desvio padrão de cada eixo de coordenadas, ela acaba criando um limite para cada plano pois, por exemplo, os valores da métrica de computação podem ser muito diferentes dos de comunicação. E desta forma a distância entre a linha X dos processos com maiores PM e cada um dos demais processos pode variar bastante. Entretanto, esta análise seleciona os processos em cada relação, independente de uma amostragem global.

As heurísticas fazem uso do monitoramento das três métricas que regem a aplicação distribuída. Desta forma, cada processo em execução é representado por estas três métricas. Dentro do sistema elas são analisadas de acordo com o perfil da aplicação, com isso é possível efetuar uma definição com maior assertividade do processo que, se migrado para outro recurso dentro do ambiente distribuído, melhor beneficiará a aplicação em execução.

6.1 Contribuições

As heurísticas apresentadas tem como principal contribuição o desenvolvimento de novos métodos para a seleção automática e eficiente de processos para migração em ambientes distribuídos. Elas foram desenvolvidas com base no modelo de reescalonamento de processos MigBSP. Entretanto, elas são consideradas como modelos a parte, e podem ser utilizadas em aplicações que trabalhem com reescalonamento de processos em aplicações BSP.

As heurísticas apresentam alguns benefícios, conforme listados a seguir:

- Seleção automática de processos;
- Análise de todas as informações dos processos, através do cálculo da distância no espaço tridimensional utilizando o MigCube;
- Análise em fases pelo MigHull, calculando através de pares de coordenadas, para uma solução ao final da execução;
- Adição de duas novas heurísticas ao modelo MigBSP;
- Utilização do Espaço Tridimensional como ferramenta de análise dos processos em execução;

- Adaptação da Envoltória Convexa para trabalhar dinamicamente de acordo com a necessidade das heurísticas.

Os itens anteriores foram estudados e testados como formas para obter sempre um melhor resultado na redução do tempo de execução e balanceamento do ambiente distribuído.

Conforme apresentado na Tabela 7, se verifica que a seleção automática de processos é um recurso implementado por poucos modelos/bibliotecas, e os que implementam pecam em outros requisitos como a análise individual de métricas e análise do custo de migração de um processo. Desta forma, apresentamos MigCube e MigHull, heurísticas que utilizam o modelo MigBSP para balanceamento de carga.

Tabela 7 – Comparativo de trabalhos relacionados com o MigCube/MigHull.

Biblioteca	Adaptação para Uso	Grid/Cluster	Seleção Automática	Métrica Computação	Métrica Comunicação	Métricas Combinadas	Custo de Migração	Suporte a BSP	Sup. a Dinamicidade	Sup. Heterogeneidade	Suporte a Migração
jMigBSP/MigBSP	Não	Todos	Não*	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim
HAMA	HDFS	Cluster	Não	Não	Não	Não	Não	Não	Não	Sim	Sim
PUB	Não	Todos	Sim	Não	Não	Sim	Não	Sim	Sim	Sim	Sim
MulticoreBSP	Sim	Não	NsA***	Não	Não	Não	NsA***	Sim	Não**	Não**	Não
Mizan(e)	Não	Cluster	Sim	Não	Não	Sim	Não	Sim	Sim	Sim	Sim
BSPCloud	Não	Cluster	Não	Não	não	Sim	Não	Sim	Sim	Não	Sim
DistPM	Não	Todos	Sim	Não	Não	Sim	Não	Não	Sim	Sim	Sim
Pregel.NET	Sim	Cloud	Sim	Não	Não	Não	Não	Sim	Sim	Sim	Sim
CPU-GPU	Sim	Cluster	Sim	Não	Não	Não	Não	Sim	Sim	Sim	Sim
MigCube/MigHull	Não	Todos	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim

Fonte: Elaborado pelo Autor.

* Depende da definição do usuário no inicio da aplicação;

** Não informado;

*** Não se Aplica

A Tabela 7 acima mostra a lacuna que as heurísticas MigCube e MigHull preenchem, oferecendo a seleção automática de processo. Além dos recursos já disponíveis, como sendo um modelo que utiliza o BSP, permitindo a análise das métricas de computação e comunicação dos processos de forma individual e/ou combinada. Também avalia o sobrecusto de migração de um processo, trabalhando tanto em aglomerados quanto em grades computacionais, de fácil utilização e com a migração de processos em ambiente distribuído.

6.2 Resultados

Avaliando os resultados apresentados no capítulo Avaliação de Heurísticas (5) é possível verificar que o objetivo definido na sentença do problema foi alcançado. O percentual reduzido no tempo total de execução de aplicações paralelas foi de até 45% para a MigHull e até 42% para a MigCube, ou seja, foi possível aumentar a intrusividade na análise de processos de forma a garantir uma seleção totalmente automática na análise dos candidatos a migração.

- **Sentença do Problema:** *Dado uma aplicação BSP utilizando o modelo MigBSP, o desafio de pesquisa consiste em planejar como podem ser feitas heurísticas eficientes, tanto em tempo computacional quanto na qualidade dos resultados para a seleção automática de processos no MigBSP, considerando a combinação das três métricas atualmente suportadas pelo Potencial de Migração.*

As heurísticas previamente existentes dos MigBSP já ofereciam a migração de processos com ganhos no tempo de execução, porém dependem de parâmetros definidos no inicio da execução, e para isso o usuário necessita ter um conhecimento prévio da arquitetura e aplicação. As heurísticas MigCube e MigHull aqui apresentadas coletam as informações do processo durante a execução, as analisam durante a etapa de reescalonamento. E de forma automática, com base em seus cálculos, marcam os processos candidatos a migração naquela *superstep*. Em seguida o modelo efetua as migrações conforme o fluxo normal.

Apesar de serem desenvolvidas com base no modelo MigBSP, as heurísticas MigCube e MigHull são modelos de seleção voltados para aplicações paralelas desenvolvidas utilizando o modelo BSP, ou seja, as heurísticas, como um modelo, podem ser adaptadas e utilizadas em outras aplicações e situações que demandem migração de processos com o objetivo de reduzir o tempo de execução.

6.3 Trabalhos Futuros

A continuação desta dissertação consiste em trabalhos nas seguintes linhas:

(i) Avaliar a possibilidade de migrar processos durante uma *superstep*, e não apenas após a barreira de sincronização;

(ii) Utilizar uma heurística para avaliar as variações do α durante as *supersteps*;

(iii) Analisar os padrões de Computação e Comunicação de uma série de *supersteps* ao invés de verificar à cada uma;

(iv) Desenvolver um nova heurística para seleção de processos utilizando redes neurais.

Atualmente, o MigBSP trabalha na ideia do BSP, com coleta de métricas durante a computação e sincronização. A migração de processos ocorre apenas durante a barreira, com todo o processamento parado. A ideia do item (i) é avaliar a migração de processos durante a fase

de computação, de forma a não aguardar a chegada da barreira para efetuar o reescalonamento de processos, com o objetivo de ganhar tempo na execução. O item (ii) sugere a avaliação do α a cada *superstep*, atualmente ele é definido no inicio da execução e vai sendo incrementado durante a fase de reescalonamento, a ideia é avaliar a cada *superstep* este parâmetro, e variar seu valor de acordo com o comportamento do ambiente. O item (iii) propõe avaliar padrões de comunicação ao longo da execução, mantendo um histórico destas informações, de modo a rastrear e gerar um perfil da eficiência de cada processo, e a partir deste perfil, analisar as migrações. Por fim, o item (iv) sugere desenvolver e treinar uma rede neural com a capacidade de analisar cada processo, seu comportamento e métricas, definindo uma migração tendo o conhecimento exato do impacto em todo o ambiente.

REFERÊNCIAS

AGGARWAL, G.; MOTWANI, R.; ZHU, A. The Load Rebalancing Problem. **Journal of Algorithms**, [S.l.], 2006.

ANDERSON, J. **Computational Fluid Dynamics**. [S.l.]: McGraw-Hill Education, 1995. (Computational Fluid Dynamics: The Basics with Applications).

BARBER, C. B.; DOBKIN, D. P.; HUHDANPAA, H. The Quickhull Algorithm for Convex Hulls. **ACM Trans. Math. Softw.**, New York, NY, USA, v. 22, n. 4, p. 469–483, Dec. 1996.

BERNABE, S.; PLAZA, A. Commodity Cluster-Based Parallel Implementation of an Automatic Target Generation Process for Hyperspectral Image Analysis. In: PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS), 2011 IEEE 17TH INTERNATIONAL CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 1038–1043.

BISSELING, R. **Parallel Scientific Computation - A Structured Approach Using BSP and MPI**. [S.l.]: OUP Oxford, 2004. (Oxford scholarship online).

BONORDEN, O. Load Balancing in the Bulk-Synchronous-Parallel Setting using Process Migrations. **2007 IEEE International Parallel and Distributed Processing Symposium**, [S.l.], p. 1–9, 2007.

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.

CAMARGO, E. O. Ivan de; BOULOS, P. **Geometria Analítica**: um tratamento vetorial. [S.l.]: Pearson Prentice Hall, 2009.

CAROMEL, D.; KLAUSER, W.; VAYSSIèRE, J. Towards seamless computing and metacomputing in Java. In: JAVA. CONCURRENCY PRACTICE AND EXPERIENCE, 1998. **Anais...** [S.l.: s.n.], 1998. p. 1043–1061.

CASANOVA, H. Simgrid: a toolkit for the simulation of application scheduling. In: CLUSTER COMPUTING AND THE GRID, 2001. PROCEEDINGS. FIRST IEEE/ACM INTERNATIONAL SYMPOSIUM ON, 2001. **Anais...** [S.l.: s.n.], 2001. p. 430–437.

CASANOVA, H.; LEGRAND, A.; QUINSON, M. SimGrid: a generic framework for large-scale distributed experiments. In: COMPUTER MODELING AND SIMULATION, 2008. UKSIM 2008. TENTH INTERNATIONAL CONFERENCE ON, 2008. **Anais...** [S.l.: s.n.], 2008. p. 126–131.

CASAVANT, T.; KUHL, J. A taxonomy of scheduling in general-purpose distributed computing systems. **Software Engineering, IEEE Transactions on**, [S.l.], v. 14, n. 2, p. 141–154, 1988.

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.

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. **Applied Numerical Mathematics**, [S.l.], v. 52, n. 2-3, p. 133–152, Feb. 2005.

DROZDOWSKI, M. **Scheduling for Parallel Processing**. [S.l.]: Springer, 2010. (Computer Communications and Networks).

EL KABBANY, G.; WANAS, N.; HEGAZI, N.; SHAHEEN, S. A Dynamic Load Balancing Framework for Real-time Applications in Message Passing Systems. **International Journal of Parallel Programming**, [S.l.], v. 39, n. 2, p. 143–182, 2011.

EPPSTEIN, D.; GALIL, Z. Annual Review of Computer Science: vol. 3, 1988. In: TRAUB, J. F. (Ed.). . Palo Alto, CA, USA: Annual Reviews Inc., 1988. p. 233–283.

FLYNN, M. Very high-speed computing systems. **Proceedings of the IEEE**, [S.l.], v. 54, n. 12, p. 1901–1909, 1966.

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. 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.

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

GOMES, R. **Implementação de um Balanceador de Carga para a Biblioteca AMPI baseado no Modelo de Escalonamento MigBSP**. 2013.

GRAEBIN, L. **Tratamento Flexível e Eficiente da Migração de Objetos Java em Aplicações Bulk Synchronous Parallel**. 2012.

GRAEBIN, L.; RIGHI, R. 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. 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.

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

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

HEART, F. E.; ORNSTEIN, S. M.; CROWTHER, W. R.; BARKER, W. B. A new minicomputer/multiprocessor for the ARPA network. **Proceedings of the June 4-8, 1973, national computer conference and exposition on - AFIPS 73**, New York, New York, USA, p. 529, 1973.

KARAVELAS, M. I.; TZANAKI, E. Convex Hulls of Spheres and Convex Hulls of Convex Polytopes Lying on Parallel Hyperplanes. In: TWENTY-SEVENTH ANNUAL SYMPOSIUM ON COMPUTATIONAL GEOMETRY, 2011, New York, NY, USA. **Proceedings...** ACM, 2011. p. 397–406. (SoCG '11).

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).

KIYARAZM, O.; MOEINZADEH, M.-H.; SHARIFIAN-R, S. A New Method for Scheduling Load Balancing in Multi-processor Systems Based on PSO. In: INTELLIGENT SYSTEMS, MODELLING AND SIMULATION (ISMS), 2011 SECOND INTERNATIONAL CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 71–76.

KRISHNAKUMAR, Y.; PRASAD, T. D.; KUMAR, K. V. S.; RAJU, P.; KIRANMAI, B. Realization of a parallel operating SIMD-MIMD architecture for image processing application. In: COMPUTER, COMMUNICATION AND ELECTRICAL TECHNOLOGY (ICCCET), 2011 INTERNATIONAL CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 98–102.

LI, Y.; LAN, Z. A novel workload migration scheme for heterogeneous distributed computing. In: CLUSTER COMPUTING AND THE GRID, 2005. CCGRID 2005. IEEE INTERNATIONAL SYMPOSIUM ON, 2005. **Anais...** [S.l.: s.n.], 2005. v. 2, p. 1055–1062 Vol. 2.

LIU, J.; WU, Y.; MARSAGLIA, J. Making Learning Parallel Processing Interesting. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM WORKSHOPS PHD FORUM (IPDPSW), 2012 IEEE 26TH INTERNATIONAL, 2012. **Anais...** [S.l.: s.n.], 2012. p. 1307–1310.

LIU, X.; TONG, W.; HOU, Y. BSPCloud: a programming model for cloud computing. **2012 IEEE 12th International Conference on Computer and Information Technology**, [S.l.], p. 1109–1113, Oct. 2012.

MALEWICZ, G.; AUSTERN, M.; BIK, A. Pregel: a system for large-scale graph processing. **Proceedings of the ...**, [S.l.], p. 135–145, 2010.

MANSOURI, F.; HUET, S.; FRISTOT, V.; HOUZET, D. **Task migration of DSP application specified with a DFG and implemented with the BSP computing model on a CPU-GPU cluster**. [S.l.: s.n.], 2013. 326-333 p.

MENDELSON, E. **Teoria e Probabilidade de Introdução ao Calculo**. [S.l.]: BOOKMAN COMPANHIA ED, 2011.

MENDES, D. **Redes de Computadores Teoria e Pratica**. [S.l.]: Novatec Editora, 2007.

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. **2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control**, [S.l.], p. 204–207, Oct. 2011.

NAKAGAWA, I.; NAGAMI, K. Jobcast - Parallel and Distributed Processing Framework: data processing on a cloud style kvs database. In: APPLICATIONS AND THE INTERNET (SAINT), 2012 IEEE/IPSJ 12TH INTERNATIONAL SYMPOSIUM ON, 2012. **Anais...** [S.l.: s.n.], 2012. p. 123–128.

NAVAUX, P. O. **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.

PACHECO, P. **An Introduction to Parallel Programming**. [S.l.]: Elsevier Science, 2011. (An Introduction to Parallel Programming).

PATNI, J.; ASWAL, M. S.; PAL, O.; GUPTA, A. Load balancing strategies for Grid computing. In: ELECTRONICS COMPUTER TECHNOLOGY (ICECT), 2011 INTERNATIONAL CONFERENCE ON, 2011. **Anais...** [S.l.: s.n.], 2011. v. 3, p. 239–243.

RAJ, J.; FIONA, R. Load balancing techniques in grid environment: a survey. In: COMPUTER COMMUNICATION AND INFORMATICS (ICCCI), 2013 INTERNATIONAL CONFERENCE ON, 2013. **Anais...** [S.l.: s.n.], 2013. p. 1–4.

RATHORE, N.; CHANA, I. A cognitive analysis of load balancing and job migration technique in Grid. In: INFORMATION AND COMMUNICATION TECHNOLOGIES (WICT), 2011 WORLD CONGRESS ON, 2011. **Anais...** [S.l.: s.n.], 2011. p. 77–82.

REDEKOPP, M.; SIMMHAN, Y.; PRASANNA, V. Optimizations and Analysis of BSP Graph Processing Models on Public Clouds. In: PARALLEL DISTRIBUTED PROCESSING (IPDPS), 2013 IEEE 27TH INTERNATIONAL SYMPOSIUM ON, 2013. **Anais...** [S.l.: s.n.], 2013. p. 203–214.

REZENDE, P. de; STOLFI, J. **Fundamentos de geometria computacional**. [S.l.]: Universidade Federal de Pernambuco, Departamento de Informatica, 1994. 103-126 p.

RIGHI, R. d. R.; GRAEBIN, L.; COSTA, C. A. da. On the replacement of objects from round-based applications over heterogeneous environments. **Software: Practice and Experience**, [S.l.], p. n/a–n/a, 2014.

RIGHI, R. d. R.; PILLA, L. L.; CARISSIMI, A.; NAVAUX, P.; HEISS, H.-U. MigBSP: a novel migration model for bulk-synchronous parallel processes rescheduling. In: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, 2009., 2009, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p. 585–590. (HPCC '09).

RIGHI, R. 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, June 2010.

SCHEPKE, C.; MAILLARD, N. Performance Improvement of the Parallel Lattice Boltzmann Method Through Blocked Data Distributions. In: COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2007. SBAC-PAD 2007. 19TH INTERNATIONAL SYMPOSIUM ON, 2007. **Anais...** [S.l.: s.n.], 2007. p. 71–78.

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.

SIMGRID. Disponível em: <http://simgrid.gforge.inria.fr/>. Acesso em: jun. 2013.

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

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

SURI, P. K.; SINGH, M. An efficient decentralized Load Balancing Algorithm for grid. In: ADVANCE COMPUTING CONFERENCE (IACC), 2010 IEEE 2ND INTERNATIONAL, 2010. **Anais...** [S.l.: s.n.], 2010. p. 10–13.

TANENBAUM, A. **Sistemas operacionais modernos**. [S.l.]: Pearson Prentice Hall, 2011. n. 3.

TANENBAUM, A.; STEEN, M. van. **Distributed systems**: principles and paradigms. [S.l.]: Pearson Prentice Hall, 2007.

TING, I.-H.; LIN, C.-H.; WANG, C.-S. Constructing a Cloud Computing Based Social Networks Data Warehousing and Analyzing System. **2011 International Conference on Advances in Social Networks Analysis and Mining**, [S.l.], p. 735–740, July 2011.

TONG, R.; ZHU, X. A Load Balancing Strategy Based on the Combination of Static and Dynamic. **2010 2nd International Workshop on Database Technology and Applications**, [S.l.], n. 2, p. 1–4, Nov. 2010.

VALIANT, L. Why BSP computers? **Parallel Processing Symposium, 1993., ...**, [S.l.], p. 2–5, 1993.

VALIANT, L. G. A bridging model for parallel computation. **Commun. ACM**, New York, NY, USA, v. 33, n. 8, p. 103–111, aug 1990.

VASILEV, V. P. BSPGRID: variable resources parallel computation and multiprogrammed parallelism. **Parallel Processing Letters**, [S.l.], v. 13, n. 03, p. 329–340, 2003.

XIAODONG, L.; WEIQIN, T.; FU, Z.; LIAO, W. BSPCloud - A Hybrid Distributed-memory and Shared-memory Programming Model. **International Journal of Grid Distributed Computing**, [S.l.], v. 6, n. 1, p. 87 – 97, 2013.

YAMIN, A. C. Escalonamento em Sistemas Paralelos e Distribuídos. **ERAD 2001 - Escola Regional de Alto Desempenho**, [S.l.], 2001.

YZELMAN, A.; BISSELING, R. H. An object-oriented bulk synchronous parallel library for multicore programming. **Concurrency and Computation: Practice and Experience**, [S.l.], v. 24, n. 5, p. 533–553, 2012.

ZHANG, B.-Y.; MO, Z.-Y.; YANG, G.-W.; ZHENG, W.-M. An efficient dynamic load-balancing algorithm in a large-scale cluster. In: ALGORITHMS AND ARCHITECTURES FOR PARALLEL PROCESSING, 6., 2005, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2005. p. 174–183. (ICA3PP'05).

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.

ZHANG, Y.; KOELBEL, C.; COOPER, K. Hybrid Re-scheduling Mechanisms for Workflow Applications on Multi-cluster Grid. In: CLUSTER COMPUTING AND THE GRID, 2009. CCGRID '09. 9TH IEEE/ACM INTERNATIONAL SYMPOSIUM ON, 2009. **Anais...** [S.l.: s.n.], 2009. p. 116–123.

ANEXO A – EXEMPLO CÓDIGO SIMGRID

Bloco de exemplo do código extraído do trabalho de Casanova (2001).

```
#include "simgrid.h"

int main() {

SG_Resource p1,p2,link;
SG_Task c1,c2,c3,c4;   /*Computacao*/
SG_Task t1,t2,t3,t4;   /*Transferencia de Dados*/
SG_Task*tasks_done;    /*Lista de tarefas completadas*/

double clock;

SG_init();    /*Inicializacao do SimGrid*/

/*Dois processadores e uma rede*/
/*p2 duas vezes mais rapido do que p1*/

p1 = SG_createHost("p1",1.O',I ./cpu1");
p2 = SG_createHost("p2",2.0,"./cpu2");
link = SG_createLink("link",* I . /latency",' I . /bandwidth") ;

/*Criacao das Tarefas*/
c1 = SG_newTask("c1",50.00);      /* 50 s */
c2 = SG_newTask("c2",100.00);     /* 100 s */
c3 = SG_newTask("c3",200.00);     /* 200 s */
c4 = SG_newTask("c4",80.00);      /* 80 s */
t1 = SG_newTask("t1",1000.00);    /* 1 Kb */
t2 = SG_newTask("t2",1000.00);    /* 10 Kb */
t3 = SG_newTask("t3",200000.00); /* 200 Kb */
t4 = SG_newTask("t4",200000.00); /* 200 Kb */

/*Configuracao das dependencias*/
SG_addDependent(t1,c1) ;SG_addDependent(t2,c1);
SG_addDependent(c2,t1) ;SG_addDependent(c3,t2);
SG_addDependent(t3,c2) ;SG_addDependent(t4,c3);
SG_addDependent(c4,t3) ;SG_addDependent(c4,t4);
```

```
/*Agendamento de Comunicacao entre Hosts*/
SG_scheduleTaskOnResource(c1,p1);
SG_scheduleTaskOnResource(c2,p1);
SG_scheduleTaskOnResource(c3,p2);
SG_scheduleTaskOnResource(c4,p2);

/*Transfere t1,t4 no mesmo Host*/
SG_scheduleTaskOnResource(t1,SG_LOCAL_LINK);
SG_scheduleTaskOnResource(t4,SG_LOCAL_LINK);

/*Transfere t2,t3, agendados no mesmo link*/
SG_scheduleTaskOnResource(t2,link);
SG_scheduleTaskOnResource(t3,link);

/*Executa a simulacao ate que o c4 termine*/
SG_simulate(0.0,c4,SG_VERBOSE); clock=SG_getclock();

/*Executa por 100sec em tempo virtual*/
SG_reset();
tasks_done=SG_simulate(100.0,NULL,SG_VERBOSE);

/*Executa ate a ultima tarefa concluir*/
SG_reset();
tasks_done=SG_simulate(_1.0,NULL,SG_VERBOSE);
clock=SG_getclock();

/*Libera memoria utilizada pelo SimGrid*/
SG_shutdown();
exit(0);

}
```