Programa Interdisciplinar de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Lucas Graebin

Tratamento Flexível e Eficiente da Migração de Objetos Java em Aplicações Bulk Synchronous Parallel

São Leopoldo, 2012

Lucas Graebin

TRATAMENTO FLEXÍVEL E EFICIENTE DA MIGRAÇÃO DE OBJETOS JAVA EM APLICAÇÕES BULK SYNCHRONOUS PARALLEL

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
2012

Ficha catalográfica

G734t Graebin, Lucas

Tratamento Flexível e Eficiente da Migração de Objetos Java em Aplicações Bulk Synchronous Parallel / por Lucas Graebin — 2012.

87 f.: il.; 30 cm.

Dissertação (mestrado) — Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2012.

Orientação: Prof. Dr. Rodrigo da Rosa Righi.

1. Bulk Synchronous Parallel. 2. Java. 3. Reescalonamento. 4. Balanceamento de carga. I. Título.

CDU 004.42

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

Para os meus pais, Ademir e Sueli,
pelo incentivo, pela educação
e pela dedicação.

Para a minha esposa Angélica,
pelo carinho, pelo afeto
e pela paciência.

RESUMO

Migração de processos é um pertinente mecanismo para oferecer balanceamento dinâmico de carga, principalmente em ambientes dinâmicos e heterogêneos. Em especial, esse tópico é importante para aplicações BSP (*Bulk Synchronous Parallel*) uma vez que elas compreendem execuções em fases, onde o tempo de cada superetapa é determinado pelo processo mais lento. Nesse contexto, esse trabalho apresenta o sistema jMigBSP. Ele permite a escrita de aplicações BSP em Java e seu diferencial diz respeito às facilidades de reescalonamento de objetos em duas maneiras: (i) usando diretivas de migração no código da aplicação e; (ii) através do balanceamento de carga automático em nível de *middleware*. Além das abordagens de reescalonamento, jMigBSP facilita a interação entre os objetos através de métodos para comunicação assíncrona e *one-sided*. O desenvolvimento de jMigBSP foi guiado pelas ideias de eficiência e flexibilidade. Em primeiro lugar, a eficiência é marcada pela preocupação com o seu desempenho se comparado com linguagens compiladas, bem como no próprio algoritmo de reescalonamento. Além disso, a flexibilidade está presente no tratamento do reescalonamento automático de objetos. A avaliação de jMigBSP compreendeu o desenvolvimento e a execução de duas aplicações BSP em um ambiente *multicluster*: (i) transformada rápida de Fourier e; (ii) compressão de imagens. Duas heurísticas para a seleção dos objetos candidatos à migração foram aplicadas na avaliação. A primeira seleciona um objeto BSP com o maior valor de PM (Potencial de Migração). A segunda escolhe uma percentagem de objetos baseado no maior PM. Os resultados mostram que jMigBSP oferece a oportunidade de ganhos de desempenho sem alterações no código da aplicação. jMigBSP torna possível ganhos de desempenho na casa de 29%, bem como produz uma baixa sobrecarga quando comparado com uma biblioteca de código nativo. Além disso, uma sobrecarga média de 5,52% foi observada no algoritmo de reescalonamento. Em geral, os resultados obtidos mostram na prática a teoria da migração de processos, onde aplicações computacionalmente intensivas (*CPU-bound*) são mais beneficiadas com a transferência de entidades (processos, tarefas, objetos etc.) para processadores mais rápidos. Considerando que a seleção de uma percentagem de objetos para migração se mostrou uma heurística eficiente, trabalhos futuros compreendem o desenvolvimento de outras que selecionam uma coleção de objetos sem a necessidade de parâmetros particulares para o reescalonador.

Palavras-chave: Bulk Synchronous Parallel. Java. Reescalonamento. Balanceamento de carga.

ABSTRACT

Process migration is an useful mechanism for runtime load balancing, mainly in heterogeneous and dynamic environments. In particular, this technique is important for Bulk Synchronous Parallel (BSP) applications. This kind of application is based in rounds, or supersteps, where the time of each superstep is determined by the slowest process. In this context, this work presents the jMigBSP system. It was designed to act over BSP-based Java applications and its differential approach concerns the offering of the rescheduling facility in two ways: (i) by using migration directives in the application code directly and; (ii) through automatic load balancing at middleware level. In addition, the presented library makes the object interaction easier by providing one-sided asynchronous communication. The development of jMigBSP was guided by the following ideas: efficiency and flexibility. First of all, the efficiency topic involves the performance relation with compiled languages (native code), as well as the time spent in the rescheduling algorithm itself. Moreover, the flexibility is present in the treatment of automatic object rescheduling. The evaluation of jMigBSP comprised the development and execution of two BSP applications in a multicluster environment: (i) fast Fourier transform and; (ii) Fractal image compression. Two heuristics were used for selecting the candidate objects for migration in the evaluation. The first heuristic chooses the BSP object that presents the highest PM (Potential of Migration) value. The second heuristic selects a percentage of objects based on the highest PM value. The results showed that jMigBSP offers an opportunity to get performance in an effortless manner to the programmer since its does not need modifications in the application code. jMigBSP makes possible gains of performance up to 29% as well as produces a low overhead when compared with a C-based library. Furthermore, an average overhead of 5,52% was observed in the rescheduling algorithm. In general, the results demonstrate in practice the theory of process migration, where computationally intensive applications (CPU-bound) are most benefited by the entities transferring (processes, tasks or objects) to faster processors. Considering that the selection of a percentage of objects for migration showed an efficient heuristic, future work includes the development of new mechanisms that select a collection of objects without the need to setup particular parameters to the rescheduler.

Keywords: Bulk-Synchronous Parallel. Java. Rescheduling. Load balancing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

BSP	Bulk Synchronous Parallel
FFT	Fast Fourier Transform
JIT	Just-In-Time
JVM	Java Virtual Machine
MPI	Message Passing Interface
PVM	Parallel Virtual Machine
RDMA	Remote Direct Memory Access
RMI	Remote Method Invocation
SPMD	Simple Program Multiple Data
VN	Virtual Node
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Atualmente, o uso de redes de computadores está crescendo como plataforma para a resolução de problemas de propósito geral. Especialmente nas áreas de compartilhamento de recursos e de alto desempenho, podemos observar o uso de grandes sistemas distribuídos heterogêneos. Neles, questões como dinamicidade, balanceamento de carga e gestão de recursos devem ser cuidadosamente analisadas (KANG et al., 2009). Esse crescimento em escala ocorre porque, às vezes, um único sistema, como um computador ou um agregado de computadores (*cluster*), não oferece um desempenho satisfatório ou simplesmente não fornece capacidade suficiente de memória ou armazenamento em disco. Considerando isso, uma ideia é montar máquinas paralelas heterogêneas com um baixo custo financeiro, simplesmente unindo o poder de estações de trabalho e agregados que pertencem a uma ou mais instituições. Assim, as grades computacionais (*grid computing*) surgem para proporcionar uma infraestrutura computacional distribuída para aplicações da ciência e da engenharia (BATISTA et al., 2008).

Nesse tipo de ambiente, a carga dos processadores pode variar e as redes podem se tornar congestionadas enquanto as aplicações estão sendo executadas (CHEN; ZHU; AGRAWAL, 2006). Além da variação nas ações de computação e comunicação dos processos, podem ocorrer variações relacionadas à dinamicidade da infraestrutura. Considerando esse aspecto, uma alternativa para a obtenção de desempenho é o uso do reescalonamento através da migração de entidades como processos, tarefas ou objetos. Essa técnica é útil para migrar entidades para execução mais rápida em recursos levemente carregados e/ou aproximar aquelas que se comunicam com frequência para recursos mais rápidos. Além disso, a migração pode ser oferecida em nível de aplicação ou de *middleware* (PONTELLI; LE; SON, 2010). A primeira ideia pode usar chamadas explícitas no código fonte, enquanto a segunda representa uma extensão da biblioteca de programação para fornecer um mecanismo de migração transparente e sem esforço para o programador. Pesquisas sobre reescalonamento incluem a definição de métricas unificadas para agir em resposta a dinamicidade da aplicação e dos recursos, além de interfaces de programação para prover facilidade na migração de entidades (EL KABBANY et al., 2011).

Considerando o segundo tema de pesquisa, tanto o modelo de programação quanto a linguagem devem ser cuidadosamente analisados para se obter desempenho e usabilidade. Nesse sentido, o modelo BSP (*Bulk Synchronous Parallel*) e a linguagem Java aparecem como candidatos para atuarem em ambientes de grades computacionais (BONORDEN, 2007). O modelo BSP representa um estilo comum para a escrita de aplicações paralelas bem sucedidas (BONORDEN, 2007; DE GRANDE; BOUKERCHE, 2011). Ele foi proposto por Leslie Valiant como uma estrutura unificada para a análise, projeto e programação de sistemas paralelos de propósito geral (VALIANT, 1990). Aplicações BSP são compostas por um conjunto de processos que executam superetapas. Cada superetapa é subdividida em três fases ordenadas: (i) computação local; (ii) ações de comunicação global e; (iii) sincronização dos processos. Enquanto isso, a linguagem Java tem uma característica multiplataforma e oferece classes para a

Tabela 1: Índice Tiobe das linguagens de programação mais populares em fevereiro de 2012 (TIOBE SOFTWARE, 2012)

Posição em fevereiro de 2012	Posição em fevereiro de 2011	Linguagem de programação	Pontuação em fevereiro de 2012
1	1	Java	17,050%
2	2	C	16,523%
3	6	C#	8,653%
4	3	C++	7,853%
5	8	Objective-C	7,062%

escrita de aplicações distribuídas que escondem detalhes técnicos de desenvolvedores. Java tem se tornado uma das mais difundidas entre as linguagens orientadas a objetos, sendo uma opção emergente para a computação de alto desempenho (TABOADA; TOURIÑO; DOALLO, 2009). Além disso, o fato da linguagem ser interpretada não é mais um obstáculo. A diferença de desempenho entre Java e linguagens nativas (por exemplo, C e Fortran) tem diminuído nos últimos anos graças ao compilador *Just-In-Time* (JIT) da máquina virtual Java (JVM) (TABOADA; TOURIÑO; DOALLO, 2009; SHAFI et al., 2009). Em adição, de acordo com o índice Tiobe (2012), Java é a linguagem de programação mais popular para o desenvolvimento de sistemas. A Tabela 1 apresenta as cinco linguagens de programação mais utilizadas segundo esse índice. O estudo é baseado em fornecedores de *software* e pesquisas por meio dos sites de busca.

O modelo BSP não especifica como os processos devem ser atribuídos aos recursos. O programador deve lidar com questões de escalonamento para obter um melhor desempenho da aplicação, principalmente porque a fase da barreira sempre espera pelo processo mais lento antes de iniciar a superetapa seguinte (BONORDEN, 2007). Portanto, a tarefa de alocar processos BSP para processadores se torna um trabalho que requer um esforço considerável. A fim de explorar plenamente esse tipo de ambiente, o programador deve conhecer tanto a arquitetura de máquina paralela com antecedência quanto o código da aplicação BSP. Além disso, uma nova aplicação ou recurso na infraestrutura exige uma nova análise do algoritmo de escalonamento. Nesse sentido, um módulo de balanceamento de carga poderia ser implementado com a biblioteca de programação. Informações referentes ao comportamento dos processos e a carga e capacidade total dos processadores poderiam ser coletadas por esse módulo para auxiliar na tomada de decisão sobre o remapeamento dos processos. Considerando o ponto de vista do programador, essa abordagem representaria uma maneira fácil de obter desempenho.

1.1 Definição do Problema e Objetivos

Procurando integrar as facilidades das grades computacionais e da linguagem de programação Java, é definido como tema central dessa dissertação o desenvolvimento do sistema **jMigBSP** (GRAEBIN; RIGHI, 2011). A principal motivação para a sua construção está em prover um sistema que ofereça uma interface de programação simples para a escrita de apli-

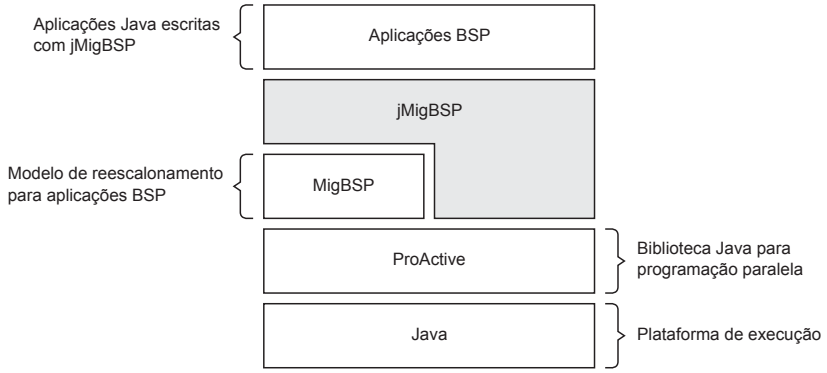

Figura 1: Organização das tecnologias e modelos que compõem o sistema jMigBSP

cações BSP com suporte ao reescalonamento de objetos. Nesse sentido, o sistema jMigBSP propõe uma solução para a sentença abaixo.

- **Sentença do Problema:** *Dado um ambiente potencialmente heterogêneo e dinâmico, o desafio consiste em disponibilizar um sistema flexível de modo que ofereça uma interface simples para a escrita de aplicações BSP e que suporte o reescalonamento eficiente de objetos de diferentes maneiras para o programador.*

O sistema jMigBSP se propõe a atuar em ambientes heterogêneos. Nesse sentido, a linguagem de programação Java foi adotada como plataforma alvo para a escrita de aplicações. Ela se destaca pelo seu caráter multiplataforma, permitindo portabilidade entre diferentes arquiteturas de máquinas e sistemas operacionais. Somado a isso, o recurso da orientação a objetos oferece as vantagens da herança, do polimorfismo e da reusabilidade de código, proporcionando clareza e simplicidade para a escrita de aplicações. Em adição, o desenvolvimento de jMigBSP é suportado pela biblioteca ProActive (CAROMEL; KLAUSER; VAYSSIERE, 1998). Ela oferece facilidades para a escrita de aplicações paralelas tais como comunicação em grupo, comunicação assíncrona e migração de objetos. Além disso, ProActive possibilita o lançamento de aplicações em ambientes heterogêneos formados por diversos agregados de computadores. Somado a isso, ProActive utiliza técnicas como VPN (*Virtual Private Network*) e SSH (*Secure Shell*) para viabilizar a comunicação entre esses ambientes (AMEDRO et al., 2010). Entretanto, ProActive é uma biblioteca complexa e requer que o programador possua conhecimentos em programação paralela e distribuída para tirar proveito de suas facilidades. Nesse sentido, jMigBSP se propõe a oferecer uma interface de programação mais simples em relação a ProActive, abstraindo toda a complexidade dessa biblioteca. A Figura 1 situa jMigBSP em relação as tecnologias e modelos adotados.

A arquitetura alvo de jMigBSP são ambientes de grades computacionais. Uma característica importante nesses ambientes é que a dinamicidade dos recursos pode variar significativamente ao longo do tempo (CHEN; ZHU; AGRAWAL, 2006). Nesse contexto, a tarefa de equilibrar o uso da carga para alocar ou redistribuir processos muitas vezes torna-se uma dificuldade para o programador devido às características heterogêneas da arquitetura. Portanto, o objetivo é fazer jMigBSP reagir contra a dinamicidade da aplicação e dos recursos através da migração de

objetos. Em especial, o sistema jMigBSP oferece uma extensão que proporciona o reescalonamento automático de objetos sem requerer a intervenção do programador. O desenvolvimento dessa extensão é guiado pelo modelo MigBSP, que trata do balanceamento de carga automático em aplicações BSP (RIGHI et al., 2010; GRAEBIN; ARAÚJO; RIGHI, 2011). Essa abordagem atua de forma contrária àquela comumente usada, na qual algoritmos de balanceamento de carga são implementados diretamente no código da aplicação (CHAUBE; CARINO; BANICESCU, 2007). Dessa forma, espera-se que o tempo de execução da aplicação com migração habilitada apresente uma eficiência superior em relação ao tempo de execução da aplicação sem o reescalonamento habilitado. Por fim, além da migração automática, jMigBSP oferece métodos para a migração explícita de objetos. Esse documento trata dessa tarefa como reescalonamento em nível de aplicação. Nessa linha, a maleabilidade de escolha entre uma abordagem ou outra define nosso objetivo de alcançarmos a flexibilidade em jMigBSP.

1.2 Organização do Texto

Esse documento está organizado em seis capítulos. Após o capítulo de introdução, apresentamos algumas informações sobre os conceitos abordados nesse trabalho. Assim, o Capítulo 2 descreve alguns tópicos, tais como escalonamento, balanceamento de carga, o modelo de programação BSP e a linguagem de programação Java. O Capítulo 3 apresenta trabalhos relacionados a sistemas e algoritmos de escalonamento e balanceamento de carga para o modelo BSP. Esse capítulo também aborda bibliotecas de comunicação BSP e sistemas Java que oferecem migração de objetos. Sua principal contribuição está em definir lacunas que estão faltando em sistemas de programação BSP.

O Capítulo 4 apresenta o sistema jMigBSP. Esse sistema é o foco principal dessa dissertação. Iremos descrever nesse capítulo as principais características de jMigBSP bem como sua interface de programação. Em especial, o Capítulo 4 aborda detalhes da implementação do mecanismo de reescalonamento automático de objetos. As estratégias utilizadas para a avaliação de jMigBSP são apresentadas no Capítulo 5. Capítulos intermediários têm uma seção especial chamada “Balanço”, que faz uma breve análise da parte em evidência dessa pesquisa. Por fim, nós apresentamos o Capítulo 6. Ele faz uma conclusão sobre o texto, enfatizando as principais contribuições e resultados de jMigBSP.

2 FUNDAMENTOS

O objetivo desse capítulo é apresentar algumas terminologias e conceitos que serão usados nos capítulos seguintes desse trabalho. Esse capítulo mostra a relevância do escalonamento de processos e do balanceamento de carga em sistemas paralelos. Em especial, ele aborda a importância desses tópicos em ambientes heterogêneos e dinâmicos. Além disso, esse capítulo descreve o modelo BSP e a linguagem de programação Java. Ambos serão utilizados no desenvolvimento de jMigBSP.

O presente capítulo está segmentado em cinco seções. A Seção 2.1 trata sobre o escalonamento de processos. Ela apresenta uma taxonomia amplamente utilizada nessa área e aborda o problema do reescalonamento. A Seção 2.2 mostra a importância do balanceamento de carga e como técnicas de migração de processos podem ser utilizadas para melhorar o desempenho de sistemas distribuídos. A Seção 2.3 descreve o modelo de programação paralela denominado BSP (*Bulk Synchronous Parallel*). Ela apresenta ideias para aumentar o desempenho de aplicações BSP e locais onde podemos agregar algoritmos de balanceamento de carga. A Seção 2.4 descreve algumas características da linguagem Java e aborda mecanismos que visam aumentar o desempenho de aplicações escritas nessa linguagem. Por fim, a Seção 2.5 apresenta um resumo desse capítulo, fazendo uma relação entre os principais temas abordados nele.

2.1 Escalonamento

O problema do escalonamento, de modo geral, compreende um conjunto de recursos e um conjunto de consumidores, conforme ilustrado na Figura 2 (YAMIN, 2001). Esse problema consiste em encontrar uma política eficiente para gerenciar o uso dos recursos pelos vários consumidores de forma que seja otimizada a medida de desempenho tida como parâmetro. Geralmente, uma proposta de escalonamento é avaliada por duas características: (i) desempenho e; (ii) eficiência. Dessa forma, a avaliação abrange tanto o escalonamento obtido como o tempo gasto para executar as políticas utilizadas pelo escalonador. Por exemplo, se o critério para julgar o escalonamento obtido for o tempo que um programa paralelo necessita para a sua execução, quanto menor esse, melhor o escalonador. Por outro lado, a política de escalonamento (utilizada pelo escalonador) pode ser avaliada em função da sua complexidade computacional. Assim, se duas políticas produzem escalonamentos iguais, a mais simples (de menor custo computacional) será a melhor (YAMIN, 2001).

A fim de formalizar a classificação dos escalonadores, Casavant e Kuhl (1988) apresentam

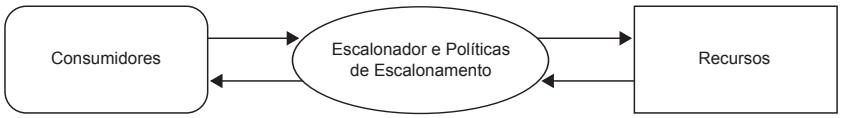

Figura 2: Visão geral do problema do escalonamento (YAMIN, 2001)

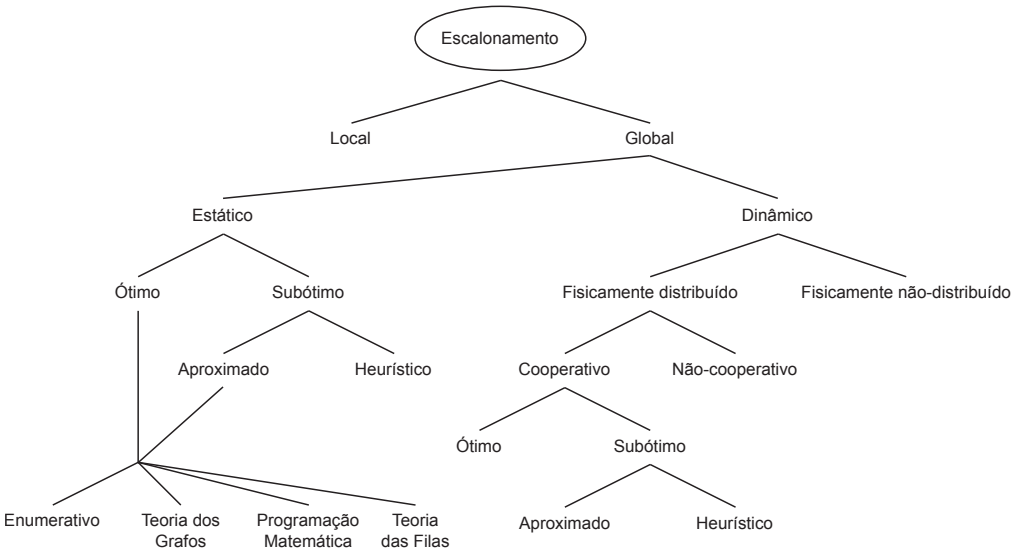

Figura 3: Taxonomia hierárquica proposta por Casavant e Kuhl (1988)

uma taxonomia para o escalonamento em sistemas distribuídos de propósito geral. Essa taxonomia é dividida em dois aspectos: (i) hierárquico e; (ii) horizontal. No aspecto hierárquico, os autores propõem uma estrutura em árvore para a classificação das técnicas de escalonamento de processos. Essa estrutura é apresentada na Figura 3. Em sua primeira divisão, o escalonamento é classificado em local e global. O escalonamento local diz respeito às metodologias de compartilhamento de tempo utilizadas quando vários processos concorrem a um único processador. O escalonamento global envolve decidir o lugar (processador) no qual determinado processo será executado, ficando a tarefa de escalonamento local entregue ao sistema operacional do processador de destino. O escalonamento global é dividido em duas abordagens: (i) estática e; (ii) dinâmica. Essas dizem respeito ao momento no qual as decisões sobre o escalonamento serão tomadas.

No escalonamento estático, as informações sobre os processadores, o tempo de execução das tarefas, o tamanho dos dados, o padrão de comunicação e a relação de dependência entre as tarefas são conhecidas de antemão. Nessa abordagem, as decisões sobre o escalonamento são tomadas antes da aplicação ser iniciada, sendo essas fixadas ao longo da mesma. A abordagem estática é simples de ser implementada. No entanto, Lu et al. (2006) apontam que ela tem duas grandes desvantagens. Em primeiro lugar, a distribuição da carga de trabalho e o comportamento de muitas aplicações não podem ser previstas antes da execução do programa. Em segundo lugar, o escalonamento estático supõe que as características dos recursos computacionais e da rede de comunicação são conhecidas com antecedência e permanecem constantes. Tal suposição não pode ser aplicada em grades computacionais, por exemplo, devido à dinamicidade desses ambientes (recursos podem entrar e sair a qualquer momento e a capacidade individual de cada recurso pode variar ao longo do tempo) (YU; SHI, 2007).

No caso onde todas as informações referentes ao estado do sistema e as necessidades de recurso dos processos são conhecidas, um escalonamento ótimo pode ser obtido. Mesmo com

todas as informações necessárias para o escalonamento, o método estático é computacionalmente caro, chegando ao ponto de ser inviável (YAMIN, 2001). Assim, esse fato resulta em soluções subótimas, dentre as quais pode-se adotar duas abordagens: (i) aproximada e; (ii) heurística. O escalonamento aproximado utiliza os mesmos algoritmos do escalonamento ótimo, mas ao invés de explorar todo o espaço das possíveis soluções ideais, ele se satisfaz quando encontra uma considerada boa. O escalonamento heurístico usa parâmetros e ideias que afetam o comportamento do sistema paralelo, como por exemplo, o agrupamento de processos com elevada taxa de comunicação em um mesmo processador (YAMIN, 2001). A expectativa é que tal procedimento melhore o desempenho do sistema como um todo.

O escalonamento dinâmico trabalha com algoritmos de escalonamento que tomam decisões durante a execução dos processos. Ele trabalha com a ideia de que pouco (ou nenhum) conhecimento a respeito das necessidades e do comportamento da aplicação estará disponível de antemão. Dessa forma, nenhuma decisão é tomada até que a aplicação inicie sua execução. Seguindo a interpretação da Figura 3, a responsabilidade do escalonamento dinâmico pode ser atribuída a um único processador (fisicamente não-distribuído) ou partilhada entre processadores (fisicamente distribuído). Nesse último, a taxonomia pode distinguir entre os mecanismos que envolvem a cooperação entre os componentes distribuídos (cooperativo) e aqueles em que os processadores individuais tomam decisões independentes das ações dos demais processadores (não-cooperativo). Nesse ponto da taxonomia, soluções ótimas, subótimas aproximadas e subótimas por heurísticas podem ser utilizadas. A discussão apresentada no ramo estático aplica-se, também, nesse caso.

Casavant e Kuhl (1988) também apresentam uma classificação horizontal para o escalonamento. As características descritas na classificação horizontal podem ser encaixadas em todos os ramos da classificação hierárquica. Nesse contexto, o escalonamento pode ser visto como adaptativo e não-adaptativo. Uma solução adaptativa para o problema do escalonamento é aquela em que os algoritmos e os parâmetros utilizados para implementar a política de escalonamento mudam dinamicamente de acordo com o comportamento do sistema. Em contraste, uma política não-adaptativa não modifica seu mecanismo de controle de escalonamento em resposta do comportamento do sistema. Em adição, a classificação horizontal também apresenta uma política de escalonamento com balanceamento de carga (CASAVANT; KUHL, 1988). A ideia básica dessa política é fazer com que os processos progridam em todos os computadores de um sistema distribuído a uma mesma razão. Para isso, a informação sobre a carga nos diferentes processadores é compartilhada através da rede de interconexão periodicamente ou sob demanda, de forma que todos os computadores tenham uma visão do estado global do sistema. Dessa forma, todos os computadores cooperam com o objetivo de redistribuir as tarefas dos processadores sobrecarregados para os demais.

O escalonamento de processos trata do mapeamento inicial de um processo para um recurso específico. Normalmente, esse procedimento é realizado quando um novo processo é criado durante a execução de uma aplicação, ou quando ela é lançada. O problema do escalonamento

pode ser estendido e usado para redistribuir um processo durante a sua execução. Essa situação caracteriza o problema conhecido como reescalonamento (DU; SUN; WU, 2007). O reescalonamento pode ser visto como uma técnica que redistribui os consumidores (processos) que já estão atribuídos a um conjunto específico de recursos para executar (ou continuar sua execução) em outro. O reescalonamento é pertinente em grades computacionais e ambientes de rede compartilhados, onde o desequilíbrio de carga é causado pela dinamicidade das aplicações (variação quanto ao consumo de CPU, memória, rede etc.) e/ou pela variação da disponibilidade de seus recursos. Nesse sentido, o reescalonamento de processos pode ser implementado utilizando técnicas de migração de processos e geralmente utilizam mecanismos de balanceamento de carga em seus algoritmos.

2.2 Balanceamento de Carga

O desempenho de sistemas computacionais paralelos e distribuídos é fortemente influenciado pela distribuição de carga entre os elementos de processamento do sistema (EL KABBANY et al., 2011). O desequilíbrio de carga ocorre quando um ou mais computadores tem poucas tarefas para tratar enquanto outros possuem muitas. Dentre as razões para o desequilíbrio de carga, pode-se citar a insuficiência de paralelismo, a fraca distribuição de tarefas entre processadores, ou a desigualdade no tamanho das tarefas. Esse desequilíbrio reduz o desempenho global do sistema e para melhorá-lo, a carga de trabalho deve ser redistribuída entre todos os processadores. Esse processo é chamado de balanceamento de carga.

A ideia básica do balanceamento de carga é de tentar dividir de forma equitativa a quantidade de trabalho entre os diversos processadores pertencentes a um sistema distribuído (CASAVANT; KUHL, 1988). A principal razão para a sua adoção é o fato de que existe uma quantidade de poder de processamento que não é usado de forma eficiente, principalmente em ambientes dinâmicos e heterogêneos como as grades computacionais (YAGOUBI; MEDEBBER, 2007). Nesse contexto, políticas de escalonamento podem utilizar mecanismos de balanceamento de carga com o objetivo de (SINHA, 1996): (i) melhorar a utilização de todos os recursos; (ii) aumentar o desempenho total do sistema e; (iii) minimizar atrasos de comunicação, uma vez que processadores sobrecarregados tendem a demorar mais tempo para atender e processar as requisições pendentes. Para tanto, é necessário determinar a medida de carga (métrica que quantifica a utilização de um elemento do sistema) (LAWRENCE, 1998). Existem várias medidas de carga possíveis, incluindo: (i) o número de tarefas em uma fila; (ii) a carga média da CPU; (iii) a utilização da CPU em determinado momento; (iv) o percentual de CPU disponível; (v) a quantidade de memória livre; (vi) a quantidade de comunicação entre os processos, entre outros. Além disso, pode-se ter qualquer combinação dos indicadores referidos. Dessa forma, tais medidas podem influenciar na decisão sobre quando acionar o balanceamento de carga, quais processos estarão envolvidos e para onde eles serão transferidos.

Políticas de escalonamento podem utilizar técnicas de balanceamento de carga a fim de re-

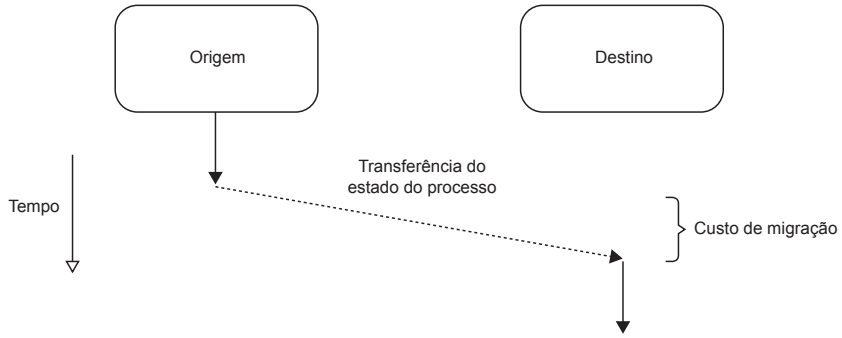

Figura 4: Representação da migração de processos (SMITH, 1988)

duzir o tempo de execução das aplicações. O balanceamento de carga pode ser classificado em estático e dinâmico (WARAICH, 2008; EL KABBANY et al., 2011). Os algoritmos estáticos distribuem as tarefas de uma aplicação paralela para os computadores de um sistema distribuído considerando a carga de trabalho dos processadores durante o lançamento da aplicação. A vantagem dessa abordagem é quanto a sua simplicidade em termos de execução e baixa sobrecarga, visto que não é necessário monitorar constantemente o desempenho das máquinas do sistema distribuído. No entanto, essa estratégia não apresenta bons resultados em ambientes com muita variação na carga de trabalho. Nesse sentido, algoritmos de balanceamento de carga dinâmicos podem ser utilizados. Essa abordagem realiza alterações no mapeamento inicial de processos para recursos em tempo de execução das aplicações. Ela monitora constantemente a carga de todos os processadores do sistema distribuído, e quando o desequilíbrio de carga atinge um valor pré-definido, o reescalonamento das tarefas é realizado. Nesse contexto, técnicas de migração de processos podem ser utilizadas para modificar o mapeamento inicial de processos para recursos e permitir uma melhor distribuição da carga no sistema.

Estratégias de reescalonamento decidem "quando" tratar da migração, "quais" processos são candidatos à transferência e para "onde" migrar os processos selecionados (MILOJICIC et al., 2000). A migração de processos refere-se a "como" o sistema irá transferir um processo da origem para o destino. A migração é dita transparente quando os efeitos da movimentação do processo são escondidos do usuário e da aplicação. Ela pode ser iniciada de dentro da computação (migração pró-ativa ou subjetiva) ou por um agente externo (reativa, objetiva, ou migração forçada). A Figura 4 ilustra o fluxo de execução de uma migração de processo. A transferência do estado do processo implica em custos de migração, que devem ser considerados no cálculo de viabilidade quando um processo é testado para a migração (DU; SUN; WU, 2007). A migração pode ser utilizada para o reescalonamento de processos, permitindo que o sistema tire proveito das mudanças no comportamento dos processos e/ou na utilização do ambiente durante a execução das aplicações. Além disso, os algoritmos de escalonamento ganham mais flexibilidade, já que a atribuição inicial de processos para recursos pode ser modificada a qualquer momento.

Os algoritmos de balanceamento de carga dependem de informações sobre o estado do sistema para a tomada de decisões (EL KABBANY et al., 2011). Para tanto, deve-se definir como

informações sobre a carga de trabalho de cada componente do sistema serão coletadas e mantidas atualizadas, uma vez que os métodos pelos quais os processadores trocam informações podem afetar a eficiência do algoritmo. Basicamente, existem duas estratégias para a troca de informações: (i) periódica e; (ii) aperiódica. Na estratégia periódica, os processadores informam sua carga de trabalho para outros em um intervalo de tempo pré-determinado. Um dos aspectos mais críticos dessa estratégia é definir o intervalo para a troca de informações. Um intervalo muito longo significaria imprecisão na tomada de decisões. Por outro lado, um intervalo curto pode gerar uma sobrecarga de comunicação desnecessária. Na maioria dos casos, a escolha do intervalo tende a ser dependente da característica da aplicação. Na estratégia aperiódica, por outro lado, um processador irá transmitir suas informações de carga sempre que a mesma variar. Essa estratégia favorece uma visão global do estado do sistema. No entanto, caso muitos eventos sejam desencadeados, pode-se gerar uma alta sobrecarga de comunicação. Em virtude disso, a estratégia aperiódica pode se tornar impraticável para ambientes grandes. Em suma, uma estratégia de troca de informações deve prover uma visão precisa do estado do sistema e ao mesmo tempo manter uma baixa sobrecarga de comunicação.

O balanceamento de carga pode ser oferecido diretamente dentro do código da aplicação. Essa abordagem resulta em um acoplamento entre a aplicação e o algoritmo de balanceamento de carga. Nesses casos, a implementação do algoritmo não pode ser utilizada com outras aplicações devido a sua especificidade (CHAUBE; CARINO; BANICESCU, 2007). Além do mais, às vezes o programador pode não ter experiência em otimizar o algoritmo de particionamento da aplicação e a implementação de um algoritmo de balanceamento de carga mudaria o foco principal do programador (a aplicação). Nesse sentido, uma estratégia é oferecer o balanceamento de carga em nível de *middleware*. Assim, um módulo de balanceamento de carga poderia ser implementado em conjunto com a biblioteca de programação. Informações referentes ao comportamento dos processos e a carga e capacidade total dos processadores poderiam ser coletadas por esse módulo para auxiliar na tomada de decisão sobre a realocação dos processos. Considerando o ponto de vista do programador, essa abordagem representaria uma maneira fácil de obter desempenho.

2.3 Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) é um modelo de programação paralela proposto por Leslie Valiant em 1990 (VALIANT, 1990). O modelo oferece um alto grau de abstração, permitindo portabilidade e estimativa do tempo de execução da aplicação em uma grande variedade de arquiteturas (GAVA; FORTIN, 2009). Dentre as aplicações que podem ser modeladas em BSP, pode-se citar o método de Lattice Boltzmann, transformada rápida de Fourier, sequenciamento de DNA e previsão meteorológica. As principais propriedades do modelo BSP são (SKILLICORN; HILL; MCCOLL, 1997):

- *A simplicidade na escrita de aplicações:* A escrita de uma aplicação BSP é semelhante a

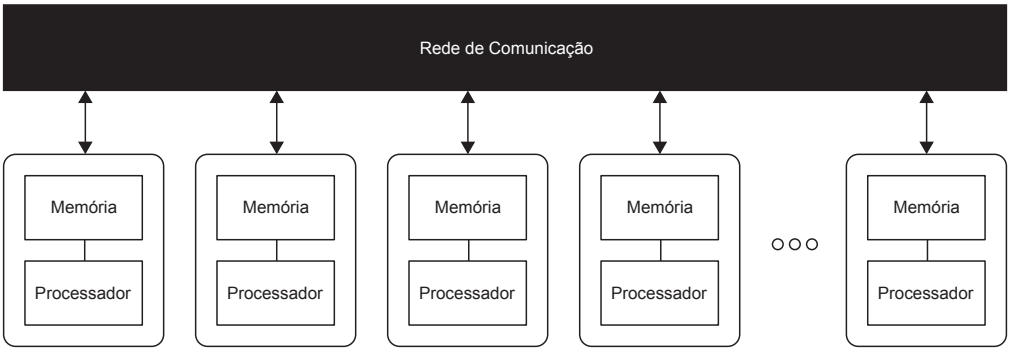

Figura 5: Arquitetura de um computador BSP (BISSELING, 2004)

escrita de uma aplicação sequencial. Apenas um mínimo de informações adicionais deve ser fornecido para descrever o uso do paralelismo;

- *A independência de arquitetura:* Ao contrário de muitos modelos de programação paralela, BSP é projetado para ser independente de arquitetura. Assim, os programas não precisam ser alterados quando movidos de uma arquitetura para outra;

- *A previsibilidade do desempenho da aplicação em uma determinada arquitetura:* O tempo de execução de uma aplicação BSP pode ser calculado considerando sua implementação e alguns parâmetros da arquitetura alvo.

O modelo BSP é composto por uma coleção de processadores (cada um com sua própria memória local) e uma rede de comunicação global (BISSELING, 2004). A Figura 5 ilustra a arquitetura de um computador BSP. Cada processador pode ler e escrever em cada célula de memória. Se a célula é local, as operações de leitura e escrita são relativamente rápidas. Se a célula pertence a outro processador, uma mensagem deve ser enviada através da rede de comunicação, sendo essa operação mais lenta. No modelo em questão, a rede de comunicação pode ser vista como uma caixa preta, onde a conectividade está escondida em seu interior. Dessa forma, o programador não precisa se preocupar com os detalhes da rede de comunicação, somente com o tempo de acesso remoto. É importante ressaltar que não há qualquer restrição sobre a proximidade dos processadores ou sobre o tipo de rede (rede local ou de larga cobertura). Assim, é possível desenvolver algoritmos portáveis, ou seja, algoritmos que possam ser utilizados em outras arquiteturas paralelas.

A principal ideia do modelo BSP é a separação explícita da computação e da comunicação. Para tanto, um algoritmo BSP é organizado em uma sequência de superetapas (BISSELING, 2004; SKILLICORN; HILL; MCCOLL, 1997). Cada superetapa contém fases de computação e comunicação, seguidas por uma barreira de sincronização, conforme ilustrado na Figura 6. Na fase de computação, cada processo executa uma sequência de operações utilizando somente dados locais. Na fase de comunicação, cada processo envia e recebe certa quantidade de mensagens. As mensagens enviadas durante uma superetapa estarão disponíveis para uso no início da próxima (GAVA; FORTIN, 2009). O adiamento das comunicações para o fim de uma superetapa remove a necessidade de realizar sincronizações entre processos e garante que eles

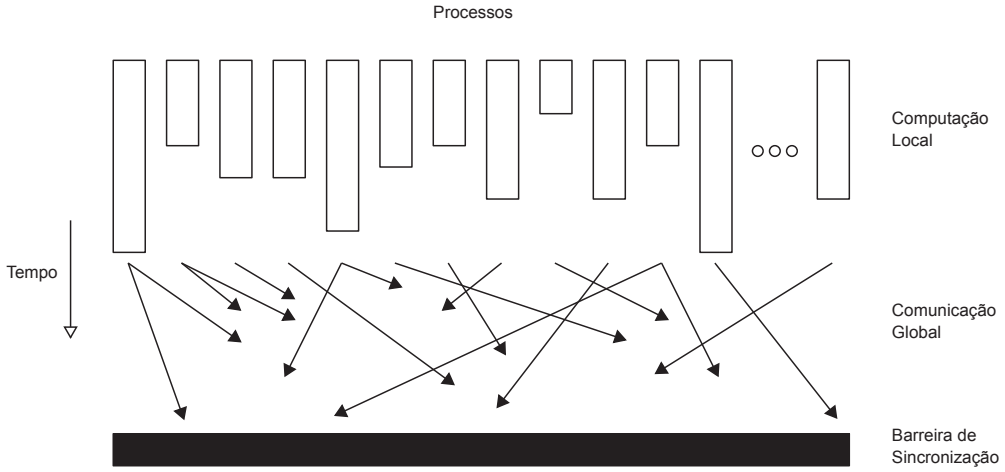

Figura 6: Representação de uma superetapa (SKILLICORN; HILL; MCCOLL, 1997)

sejam independentes entre si (DUTOT et al., 2005). No final de cada superetapa, os processos se sincronizam e esperam até que os demais tenham concluído suas atividades. Essa forma de sincronização é denominada de sincronização em massa (BISSELING, 2004).

O desempenho de uma aplicação BSP pode ser caracterizado por três parâmetros: (i) p = número de processos; (ii) l = custo de uma sincronização global e; (iii) g = tempo para transmitir uma mensagem pela rede de comunicação. O parâmetro g está relacionado a largura de banda da rede de comunicação e depende de diversos fatores, tais como: (i) os protocolos de comunicação utilizados; (ii) o gerenciamento de *buffer* pelos processadores e pela rede de comunicação; (iii) o roteamento utilizado na rede de comunicação, entre outros. A separação das fases de computação local, comunicação global e sincronização permite calcular o tempo limite de execução e prever o desempenho da aplicação utilizando equações simples. Nesse sentido, o custo de uma superetapa pode ser determinado como se segue. Supondo que w é o custo de uma computação local em cada processo durante a superetapa s, h_{out} a quantidade de mensagens enviadas por qualquer processo durante a superetapa s, h_{in} a quantidade máxima de mensagens recebidas por cada processo durante a superetapa s e l o custo da barreira de sincronização. Sendo assim, o tempo para computar a superetapa s pode ser calculado através da Equação 2.1. Dessa forma, o custo total de uma aplicação BSP pode ser calculado pela soma dos custos parciais obtidos em cada superetapa (ver Equação 2.2) (GOUDREAU et al., 1996; SKILLICORN; HILL; MCCOLL, 1997).

$$\textit{Custo da superetapa} = MAXw + MAX\{h_{in}, h_{out}\} \cdot g + l \tag{2.1}$$

$$\textit{Custo da aplicação} = \sum_{i=0}^{S-1} w_i + g \cdot \sum_{i=0}^{S-1} h_i + l \cdot s \tag{2.2}$$

Na Equação 2.2, a variável S representa o número de superetapas da aplicação. O modelo de custos também permite predizer o desempenho da aplicação em determinada arquitetura. Os valores de p, w e h para cada superetapa e a quantidade total de superetapas podem ser deter-

Tabela 2: Mecanismos para melhorar o desempenho de aplicações BSP (SKILLICORN; HILL; MCCOLL, 1997)

Mecanismo	Descrição
Balancear a computação	Balancear a computação entre os processos em cada superetapa, uma vez que o valor de w é o tempo máximo de computação e a barreira de sincronização deve aguardar o processo mais lento. Considerando uma arquitetura paralela dinâmica e heterogênea, pode-se reescalonar os processos entre os recursos disponíveis levando em consideração o valor de w.
Balancear a comunicação	Balancear a comunicação entre os processos, considerando que h é o valor máximo de mensagens enviadas e recebidas por um processo. O equilíbrio de comunicação é pertinente quando há processos que utilizam redes com baixa largura de banda. Dessa forma, pode-se reescalonar os processos e aproximar ao mesmo nível de rede aqueles que apresentam um padrão mais elevado de comunicação.
Reduzir as superetapas	Minimizar o número de superetapas, pois isso determina o número de vezes que l é contabilizado no custo final.

minados através da inspeção no código da aplicação. Os valores de g e l podem ser inseridos na fórmula para estimar o tempo de execução antes da aplicação ser executada. Nesse sentido, o modelo de custos pode ser usado como parte do processo de concepção das aplicações BSP e predizer o seu desempenho quando portadas para novas arquiteturas (SKILLICORN; HILL; MCCOLL, 1997). Além disso, algumas iniciativas podem ser tomadas com o objetivo de melhorar o desempenho de aplicações BSP, conforme apresentado na Tabela 2 (SKILLICORN; HILL; MCCOLL, 1997).

O modelo BSP pode ser expresso por uma ampla variedade de linguagens de programação. Por exemplo, programas BSP podem ser escritos usando bibliotecas de comunicação existentes como o PVM (SUNDERAM, 1990) e MPI (DONGARRA et al., 1995). A biblioteca de programação deve oferecer apenas uma função de sincronização e diretivas de comunicação. Comparado com PVM e MPI, a abordagem BSP oferece as seguintes facilidades (SKILLICORN; HILL; MCCOLL, 1997): (i) uma disciplina de programação simples (baseada em superetapas) que torna mais fácil a compreensão dos programas; (ii) um modelo de custos para análise e predição de desempenho e; (iii) independência de arquitetura. A Seção 3.2 apresenta algumas bibliotecas para programação BSP.

A abordagem mais comum para a programação BSP é o modelo SPMD (*Simple Program Multiple Data*) e o uso de linguagens de programação imperativas como C e Fortran. Além disso, programas BSP podem usar diretivas tradicionais para a troca de mensagens (envio e recepção) ou mecanismos para acesso direto a memória remota, ou RDMA (*Remote Direct Memory Access*), também conhecido como comunicação *one-sided* (SKILLICORN; HILL; MCCOLL, 1997). A vantagem da comunicação *one-sided* está em sua natureza assíncrona (GROPP; THAKUR, 2005). Diferente do modelo de comunicação ponto-a-ponto, onde o emis-

sor e o receptor chamam explicitamente funções para o envio e o recebimento de mensagens, na comunicação *one-sided*, apenas o processo de origem participa ativamente da comunicação. Ele invoca diretivas para o envio e/ou leitura de dados e a comunicação ocorre sem que o processo destino precise chamar funções de forma explícita para essas operações.

2.4 Linguagem de Programação Java

Java é uma linguagem de programação imperativa que está em constante desenvolvimento desde 1992 (LOBOSCO; AMORIM; LOQUES, 2002). A linguagem Java é desenvolvida pela Sun Microsystems (agora uma subsidiária da Oracle) e se destaca pelas características de orientação a objetos e independência de plataforma (POMINVILLE et al., 2010). Através da orientação a objetos, ela oferece as vantagens da herança, do polimorfismo, da reutilização de código e as propriedades de clareza e simplicidade para a escrita de aplicações (GETOV et al., 2001). Agregado a essas questões, Java possui um caráter multiplataforma, permitindo a portabilidade de aplicações entre diferentes arquiteturas de máquinas e sistemas operacionais. A principal questão inerente a portabilidade de Java está relacionada à sua representação de arquivos executáveis que é feita através de *bytecodes*. Um *bytecode* é o produto da compilação de um programa fonte Java para uma arquitetura neutra de máquina. Esse arquivo pode ser executado sobre quaisquer plataformas que tenham uma implementação da máquina virtual Java, ou JVM (*Java Virtual Machine*) (KOTZMANN et al., 2008). O termo virtual de uma JVM deve-se ao fato que ela é implementada em *software* sobre uma plataforma de *hardware* existente. Dessa forma, o fato de implementar JVMs em diferentes plataformas é o que faz a linguagem Java ser portável, favorecendo o seu uso em ambientes heterogêneos como as grades computacionais.

Além da orientação a objetos e da portabilidade, a distribuição padrão do Java oferece mecanismos para operar com memória distribuída, como o sistema de Sockets e de invocação remota de métodos, ou simplesmente RMI (*Remote Method Invocation*) (TABOADA; TOURIÑO; DOALLO, 2009). O sistema de Sockets provê mecanismos de conexão e troca de mensagens entre dois computadores, podendo ser utilizado para a construção de aplicações com interações de comunicação. Por outro lado, a RMI possibilita a invocação e o retorno de um método de um objeto remoto. Todavia, ela apresenta uma interface de mais alto nível, escondendo do programador questões referentes a conexão e o transporte de dados. Aplicações RMI são geralmente descritas em dois programas separados: um cliente e um servidor (YANG et al., 2006). Tipicamente, uma aplicação servidor cria objetos remotos e referências de nomes para que eles se tornem acessíveis e espera que os clientes invoquem métodos sobre tais objetos. A aplicação cliente, por sua vez, captura a referência para um ou mais objetos remotos do servidor e invoca métodos sobre eles.

Em um sistema RMI existem objetos procuradores que processam as informações passadas entre o cliente e o servidor. Na literatura, geralmente o procurador no lado do cliente (chamador) é conhecido como *stub*, enquanto que o procurador no lado do servidor é conhecido como

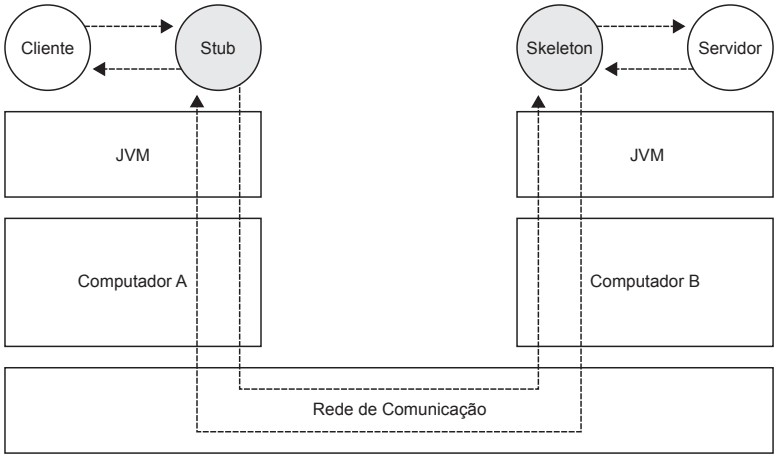

Figura 7: Organização dos objetos procuradores em um sistema RMI (LOBOSCO; AMORIM; LOQUES, 2002)

skeleton (LOBOSCO; AMORIM; LOQUES, 2002). A Figura 7 mostra a organização dos procuradores na aplicação cliente e servidor. O chamador invoca os métodos de um *stub* local, o qual é responsável por transportar a chamada de método para o objeto remoto. No lado do servidor, o *skeleton* recebe uma invocação de método e a repassa para o objeto remoto.

Em geral, um ponto em que Java é criticado é quanto ao seu desempenho para a execução de aplicações. As primeiras implementações de Java, baseadas somente na interpretação de *bytecodes*, apresentavam baixo desempenho quando comparadas a abordagens tradicionais como C e Fortran (SHAFI et al., 2009). Em virtude da crescente popularidade de Java, grandes esforços vêm sendo investidos para que o problema de desempenho seja amenizado. Nesse sentido, as distribuições atuais de JVMs fazem uso da compilação de trechos de um arquivo *bytecode* em tempo de execução, juntamente com a técnica padrão de interpretação (TABOADA; TOURIÑO; DOALLO, 2009). Para tanto, compiladores Java, como por exemplo, o *Just-In-Time* (JIT), transformam trechos de código mais frequentemente utilizados em instruções binárias para serem executadas em processadores alvo.

2.5 Balanço

Esse capítulo apresentou uma breve descrição sobre escalonamento e balanceamento de carga. Mecanismos de escalonamento gerenciam a interação entre consumidores e recursos. Os consumidores são o grão de escalonamento e podem ser representados por processos, tarefas ou objetos, por exemplo. Um escalonador atua de acordo com a sua política de escalonamento, que pode oferecer balanceamento de carga a fim de tentar dividir de forma equitativa a quantidade de trabalho entre os diversos recursos. Métodos de balanceamento de carga podem ser classificados em duas categorias: (i) estático e; (ii) dinâmico. Na abordagem estática, o mapeamento das tarefas na arquitetura ocorre antes do início da execução. Ela exige conhecimento prévio dos aspectos correspondentes à aplicação (tais como o tempo de execução) e não pode adaptar-se à variação de carga do sistema em tempo de execução. Por outro lado, na abordagem dinâmica,

decisões de escalonamento podem ser tomadas durante a execução da aplicação. Uma vez que a abordagem estática geralmente não consegue lidar com as mudanças dinâmicas nas condições do sistema, algoritmos dinâmicos são utilizados em situações práticas. No entanto, o algoritmo dinâmico deve coletar, armazenar e analisar informações sobre o sistema e, portanto, introduz um custo extra ao sistema computacional se comparado com a abordagem estática.

Considerando o grão de escalonamento como um processo do sistema operacional, seu mapeamento para recursos acontece, normalmente, antes da execução da aplicação, caracterizando uma abordagem estática. O procedimento para mapear processos para recursos antes da execução da aplicação não considera as possíveis modificações no ambiente. Dessa forma, uma ideia pertinente é o uso da abordagem dinâmica para coletar informações durante a execução da aplicação e assim obter um melhor mapeamento de processos para recursos. Entretanto, mesmo utilizando uma abordagem dinâmica, modificações após o mapeamento inicial podem torná-lo ineficiente com o passar do tempo. Nesse contexto, o fato de propor um novo mapeamento tendo em conta um anterior sugere a ideia do reescalonamento de processos.

O suporte ao reescalonamento através da migração de processos é uma das técnicas mais importantes para melhor aproveitar os recursos de um sistema distribuído. Esse mecanismo pode ser gerenciado por um algoritmo de balanceamento de carga dinâmico, que constantemente verifica a situação do ambiente oferecendo uma reorganização dos processos a fim de manter um determinado nível de desempenho no sistema. O reescalonamento proporciona a distribuição da carga e pode melhorar o desempenho de uma aplicação no que diz respeito às fases de comunicação e computação. O tempo de cálculo pode ser otimizado através da transferência de processos localizados em processadores sobrecarregados para outros com carga moderada. A comunicação pode ser melhorada minimizando o custo de troca de mensagens entre processos. Para tanto, o reescalonamento pode trazer para uma mesma rede dois processos que apresentam um padrão mais elevado de comunicação.

O modelo BSP e a linguagem de programação Java foram escolhidos para o desenvolvimento de jMigBSP. O BSP oferece simplicidade na escrita de programas paralelos, visto que ele é independente de arquitetura e fornece uma ideia do custo de execução da aplicação que combina suas três fases (computação, comunicação e barreira). O estilo de computação em fases do modelo BSP tem sido utilizado para a escrita de aplicações de propósito geral, sendo uma organização frequente na escrita de programas paralelos de sucesso (BONORDEN, 2007; DE GRANDE; BOUKERCHE, 2011). A escolha de Java ressalta o objetivo da portabilidade de BSP. Como uma linguagem interpretada, as aplicações construídas sobre ela podem ser distribuídas para quaisquer máquinas que tenham uma implementação da máquina virtual Java. Além disso, através dos conceitos da orientação a objetos, Java torna muito mais fácil a organização das ideias para a escrita de uma aplicação. Nesse contexto, Java foi escolhido como linguagem alvo para a escrita de aplicações em jMigBSP. A linguagem Java também foi adotada visto a sua crescente utilização perante a comunidade científica, em especial para a construção de aplicações de alto desempenho (TABOADA; TOURIÑO; DOALLO, 2009).

3 TRABALHOS RELACIONADOS

Esse capítulo descreve alguns trabalhos relacionados que são inseridos no contexto dessa dissertação. Esse capítulo trata de diferentes sistemas e algoritmos, bem como as suas abordagens para lidar com os seguintes tópicos: escalonamento, balanceamento de carga e migração de processos. Além disso, o presente capítulo aborda alguns trabalhos relacionados com o modelo BSP e a linguagem Java. Por último, apresentamos uma seção especial que enfatiza as lacunas existentes e guia o leitor para o próximo (e mais importante) capítulo dessa dissertação.

O presente capítulo está organizado em quatro seções. É importante notar que muitos dos trabalhos aqui apresentados poderiam estar distribuídos em duas ou mais seções. Essa organização foi feita apenas para simplificar a leitura do capítulo. A Seção 3.1 apresenta alguns trabalhos relacionados no contexto de escalonamento e balanceamento de carga. A Seção 3.2 descreve algumas iniciativas de bibliotecas de comunicação BSP, bem como algumas mudanças e melhorias feitas em tal modelo. A Seção 3.3 apresenta alguns sistemas Java que implementam migração de objetos e recursos para a escrita de aplicações de alto desempenho. Por fim, a Seção 3.4 encerra o capítulo com uma discussão sobre os principais pontos abordados nele.

3.1 Escalonamento e Balanceamento de Carga

Righi et al. (2010) desenvolveram o modelo MigBSP, que trata do balanceamento de carga automático em aplicações BSP através da migração de processos. O modelo trabalha sem o conhecimento prévio do ambiente e a realocação dos processos é feita baseada em informações capturadas durante a execução da aplicação. A ideia de MigBSP é aplicar o balanceamento de carga com o objetivo de reduzir o tempo de execução das superetapas. Entre as três formas de minimizar o tempo total de uma aplicação BSP (SKILLICORN; HILL; MCCOLL, 1997), MigBSP se propõe a atuar na redução dos tempos de computação e comunicação. O modelo controla a migração de processos BSP e trabalha de acordo com a estabilidade do sistema. Além da computação e da comunicação, MigBSP considera em seu algoritmo questões como a memória e custos de migração de um processo para avaliar a viabilidade das transferências.

O modelo MigBSP provê um formalismo que responde as seguintes questões sobre a migração de processos: (i) “quando” tratar da migração de processos; (ii) “quais” processos são candidatos à migração; (iii) “onde” colocar os processos selecionados. O modelo trabalha sobre uma arquitetura heterogênea composta por agregados, supercomputadores e/ou redes locais. A heterogeneidade trata de processadores com diferentes velocidades e redes com larguras de banda distintas. Tal arquitetura é montada com a ideia de Conjuntos e Gerentes de Conjuntos, conforme ilustrado na Figura 8. Um Conjunto pode ser uma rede local ou um agregado. Cada Conjunto possui um Gerente de Conjunto, o qual captura informações sobre seu Conjunto e as troca com os demais Gerentes.

A decisão de remapeamento dos processos é tomada no final de cada superetapa. Com o

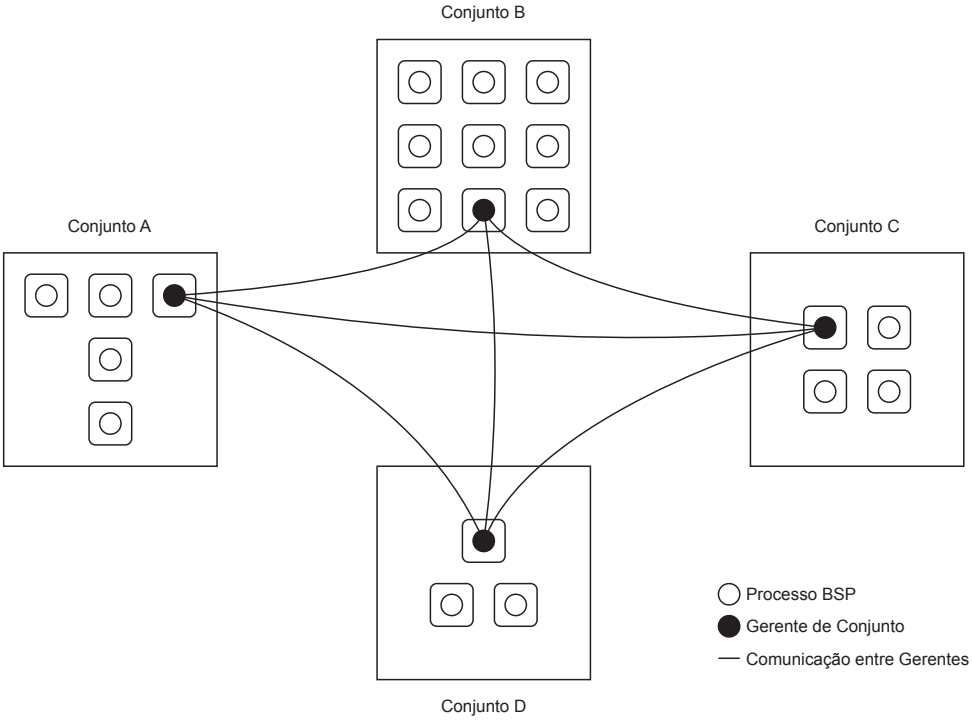

Figura 8: Organização hierárquica de MigBSP (RIGHI et al., 2010)

intuito de gerar menos intrusividade na aplicação, MigBSP atua com duas adaptações que controlam o valor de α ($\alpha \in \mathbb{N}^*$). α é atualizado a cada chamada de reescalonamento e indica o intervalo para o lançamento da próxima. A questão "Quais" é respondida através da função de decisão chamada Potencial de Migração (PM). Cada processo i computa n funções $PM(i,j)$, onde n é o número de conjuntos e j significa um conjunto específico. $PM(i,j)$ é obtido através da combinação das métricas Computação, Comunicação e Memória (Equação 3.1). Computação e Comunicação são métricas que atuam a favor da migração, enquanto a Memória trabalha em direção oposta. Na medida em que o valor de $PM(i,j)$ aumenta, aumenta também a possibilidade de migração do processo i para o Conjunto j.

$$PM(i,j) = Comp(i,j) + Comm(i,j) - Mem(i,j) \tag{3.1}$$

Outro trabalho relacionado a balanceamento de carga em aplicações BSP compreende os esforços de Jiang et al. (2007) para com o desenvolvimento do modelo ServiceBSP. ServiceBSP combina o modelo BSP com o conceito de serviços. O algoritmo de balanceamento de carga proposto pelos autores é baseado na carga de trabalho dos nós pertencentes ao sistema distribuído. Assim, quando uma nova tarefa é inserida no sistema, o algoritmo de balanceamento de carga é acionado e a tarefa em questão é enviada para o nó com o menor valor de carga. Para a escolha do nó candidato, informações sobre o consumo de CPU, memória, quantidade de tarefas em execução, a largura de banda e o tempo de resposta são considerados. Para tanto, os autores definiram M_j, que representa o peso do recurso j de um nó que uma tarefa necessita. Assim, o peso de M_j varia de acordo com as características das tarefas. Por exemplo, uma

tarefa com um alto valor de peso para o recurso de memória representa a necessidade de uma elevada quantia desse recurso para a sua execução. A Equação 3.2 expressa o cálculo para obter o valor de carga da tarefa i.

$$V_i = M_1 * V_{cpu} + M_2 * V_{mem} + M_3 * V_{tasks} + M_4 * V_{network} + M_5 * V_{response} \quad (3.2)$$

Na Equação 3.2, as variáveis V_{cpu}, V_{mem}, V_{tasks}, $V_{network}$ e $V_{response}$ representam, respectivamente, o consumo de CPU, o consumo de memória, a quantidade de tarefas em execução, a largura de banda e o tempo de resposta do sistema. Assim, o valor de carga de um nó que possui várias tarefas em execução é obtido por $V = \sum_{i=0}^{n} V_i$. Utilizando o valor corrente de carga V é possível calcular um novo valor quando uma nova tarefa é alocada.

Bonorden et al. (2006) desenvolveram a biblioteca PUBWCL. PUBWCL é uma biblioteca Java para a escrita de aplicações BSP e destina-se a utilizar o poder de processamento de computadores distribuídos pela Internet. Os participantes dispostos a doar ciclos de CPU devem instalar uma aplicação cliente do PUBWCL. Sempre que um usuário executar uma aplicação BSP, o sistema escolherá um subconjunto de clientes para executar o programa.

A biblioteca PUBWCL pode migrar processos BSP durante a barreira de sincronização e, adicionalmente, em pontos arbitrários especificados pelo programador. Além disso, processos BSP podem ser reiniciados em outros clientes quando um computador da Internet deixar de responder de forma inesperada. Esse recurso também é utilizado para permitir estratégias de balanceamento de carga. Assim, um processo BSP pode ser abortado caso ele não termine em um determinado tempo para que esse possa ser reiniciado em um cliente mais rápido. PUBWCL implementa as seguintes estratégias de balanceamento de carga:

- Algoritmo *PwoR*: O algoritmo *Parallel Execution without Restarts* (*PwoR*) executa todos os processos de uma superetapa em paralelo. Sempre que uma superetapa é concluída, todos os clientes são inspecionados quanto ao tempo de execução de seus processos. Caso o tempo de execução de um processo seja r vezes o tempo médio de duração da superetapa, o processo será redistribuído entre os clientes ativos de tal forma que o tempo de execução da próxima superetapa seja minimizado.

- Algoritmo *PwR*: Usando o algoritmo de balanceamento de carga *Parallel Execution with Restarts* (*PwR*), a execução de uma superetapa é realizada em fases. A duração da fase é r vezes o tempo de execução do processo $[s \cdot p^*]$, onde p^* é o número de processos BSP que ainda não concluíram a superetapa corrente. No término de uma fase, todos os processos BSP que não tenham sido concluídos são abortados. Assim, na próxima fase, eles serão reiniciados em clientes mais rápidos.

- Algoritmo *SwoJ*: Enquanto as estratégias *PwoR* e *PwR* executam todos os processos BSP em paralelo, o algoritmo *Sequential Execution without Just-in-Time Assignments* (*SwoJ*) executa somente um processo BSP por cliente de cada vez, sendo que os demais são

mantidos em filas. Assim como o algoritmo *PwR*, *SwoJ* opera em fases. No final de uma fase, todos os processos que não concluíram sua execução são abortados e transferidos para outros clientes.

- Algoritmo *SwJ*: Assim como o *SwoJ*, o algoritmo *Sequential Execution with Just-in-Time Assignments* (*SwJ*) executa apenas um processo BSP por cliente de cada vez e mantém os demais em filas. A principal diferença, entretanto, é que as filas são balanceadas. Mais precisamente, sempre que um cliente concluir a execução do último processo da fila, ele irá receber um processo da fila do cliente mais lento.

Bonorden também desenvolveu uma extensão da biblioteca PUB para suporte à migração de processos em aplicações BSP (BONORDEN, 2007). Ele implementou três estratégias de balanceamento de carga. A primeira consiste em uma abordagem centralizada onde um nó coleta informações sobre a carga dos demais e toma todas as decisões sobre a migração. As demais estratégias tomam decisões sobre a migração sem o conhecimento global do sistema. Todas as estratégias são explicadas abaixo:

- Estratégia global: Todos os nós enviam informações sobre a sua capacidade de processamento e carga atual para um nó mestre. O nó mestre calcula o nó menos sobrecarregado (P_{min}) e o mais sobrecarregado (P_{max}) e migra um processador virtual de P_{max} para P_{min} se necessário.

- Estratégia distribuída simples: Cada nó questiona c outros selecionados de forma aleatória sobre a atual carga de trabalho desses. Se a carga mínima de todos os c nós é menor do que a própria carga menos uma constante $d \geq 1$, um processo é migrado para o nó menos carregado.

- Estratégia de predição global: Cada nó possui uma matriz de carga de todos os outros nós, entretanto, essas entradas não são atualizadas todo o tempo. Cada nó envia periodicamente a sua carga para k outros nós escolhidos de forma aleatória.

A biblioteca PUB implementa os algoritmos de balanceamento de carga na barreira de sincronização (BONORDEN, 2007). Eles decidem "quando" migrar um processo, "qual" processo é candidato a migração e para "onde" migrá-lo. Em adição, assim como o trabalho de Jiang et al. (2007), tais algoritmos não consideram a comunicação entre os processos nem os custos de migração.

3.2 Bulk Synchronous Parallel

BSP é um modelo de programação paralela que foi planejado para a execução de aplicações em ambientes homogêneos e dedicados. No entanto, podemos usá-lo em grades computacionais. Assim, alguns trabalhos adaptam o modelo BSP para ambiente de grade. Vasilev (2003)

propôs o modelo BSPGRID, que explora o paradigma BSP para permitir que algoritmos existentes possam ser facilmente adaptados e utilizados em grades. Martin e Tiskin (2004) apresentaram uma alteração na abordagem BSPGRID, a qual permite tratar questões de tolerância a falhas em grades computacionais. Eles criaram o modelo Dynamic BSP, que oferece uma maior flexibilidade na escrita de aplicações e capacidade de gerar processos adicionais dentro das superetapas quando necessário.

Cha e Lee (2001) criaram o modelo H-BSP (*Bulk Synchronous Parallel* hierárquico). Ele oferece mecanismos para tirar proveito da localidade dos processadores e, portanto, permite o desenvolvimento de algoritmos mais eficientes. Para tanto, além do princípio básico de BSP, H-BSP usa um mecanismo especial para dividir todo o sistema em um certo número de grupos menores. Williams e Parsons (2000) desenvolveram o HBSP (*Bulk Synchronous Parallel* heterogêneo). O HBSP aumenta a aplicabilidade de BSP, uma vez que o modelo original considera que todos os componentes computacionais possuem as mesmas características de computação e comunicação. Para tanto, HBSP incorpora parâmetros adicionais que refletem as diferentes velocidades de tais componentes em ambientes heterogêneos para o cálculo da predição de desempenho.

Além das extensões ao modelo BSP apresentadas, o estado da arte em programação BSP compreende uma série de bibliotecas de programação que foram desenvolvidas ao longo do tempo sob diferentes motivações. Uma das primeiras bibliotecas propostas foi a BSPlib (HILL et al., 1998). BSPlib é uma biblioteca C composta por 20 primitivas (ver Tabela 3) e que segue o estilo de programação SPMD. Basicamente, ela contém funções para delimitar superetapas e provê operações para comunicação através de troca de mensagens e acesso remoto a memória (*one-sided*). A comunicação por troca de mensagens em BSPlib ocorre com o uso das primitivas $bsp\_send()$ e $bsp\_move()$. O processo origem invoca a primitiva $bsp\_send()$ para enviar uma mensagem ao processo destino. O processo destino, então, poderá acessar a mensagem enviada somente na superetapa subsequente através da operação $bsp\_move()$. No que diz respeito à comunicação *one-sided*, BSPlib impõe que o programador registre as variáveis que podem ser acessadas remotamente. O registro das variáveis ocorre através da primitiva $bsp\_push\_reg()$. Após o seu registro, processos remotos podem acessar variáveis remotas utilizando $bsp\_put()$ e $bsp\_get()$. Além de BSPlib, podemos citar outras bibliotecas para a escrita de aplicações BSP em C como Paderborn University BSP (PUB) (BONORDEN et al., 2003) e BSPonMPI (SUIJLEN; BISSELING, 2011).

A biblioteca PUB oferece as mesmas funcionalidades que BSPlib e se destaca por possuir primitivas para comunicação assíncrona e uma segunda semântica para a sincronização de processos. Essa semântica é baseada no trabalho de Gonzalez et al. (2000) e parte do princípio que o programador conhece a quantidade de mensagens que cada processo irá receber. Assim, os processos deixam a barreira quão logo forem recebendo as mensagens. BSPonMPI, por sua vez, possui as mesmas características de BSPlib e é executado em todos os ambientes que possuem uma implementação de MPI. Essa característica difere BSPonMPI das demais bibliotecas.

Tabela 3: Conjunto de primitivas de BSPlib (HILL et al., 1998)

Classificação	Primitiva	Descrição
Inicialização	$bsp\_begin()$	Inicia o código SPMD.
	$bsp\_end()$	Finaliza o código SPMD.
	$bsp\_init()$	Inicia a aplicação.
Consulta	$bsp\_pid()$	Retorna o código identificador do processo no grupo.
	$bsp\_nprocs()$	Retorna a quantidade de processos no grupo.
	$bsp\_time()$	Retorna o tempo decorrido do processo.
Sincronização	$bsp\_sync()$	Barreira de sincronização.
Acesso remoto a memória	$bsp\_push\_reg()$	Registra uma área de memória de acesso global.
	$bsp\_pop\_reg()$	Remove uma área de memória de acesso global.
	$bsp\_put()$	Copia dados para a memória do processo remoto.
	$bsp\_hpput()$	Similar a $bsp\_put()$, mas sem o uso de *buffer*.
	$bsp\_get()$	Copia dados da memória do processo remoto.
	$bsp\_hpget()$	Similar a $bsp\_get()$, mas sem o uso de *buffer*.
Troca de mensagens	$bsp\_set\_tagsize()$	Define o tamanho das *tags*.
	$bsp\_send()$	Envia uma mensagem para a fila remota.
	$bsp\_qsize()$	Retorna a quantidade de mensagens na fila.
	$bsp\_get\_tag()$	Retorna a *tag* da primeira mensagem na fila.
	$bsp\_move()$	Move uma mensagem da fila.
	$bsp\_hpmove()$	Similar a $bsp\_move()$, mas sem o uso de *buffer*.
Pausa	$bsp\_abort()$	Aborta todos os processos.

Além das bibliotecas de código nativo, podemos citar algumas iniciativas desenvolvidas em Java. Nesse contexto, JBSP (GU; LEE; CAI, 2001) e MulticoreBSP (YZELMAN; BISSELING, 2011) aparecem como as mais significativas. JBSP é um sistema para ambientes ponto-a-ponto e busca tirar proveito de recursos computacionais ociosos. JBSP permite que o usuário especifique os computadores em que deseja executar suas tarefas. Entretanto, quando novas tarefas são submetidas ao sistema e nenhum local é especificado, o mapeamento delas para recursos ocorre de forma aleatória. MulticoreBSP, por sua vez, é voltado para a execução de aplicações em ambientes com memória compartilhada e processadores com diversos núcleos. A comunicação entre os objetos ocorre através de memória compartilhada (YZELMAN; BISSELING, 2011). As variáveis implementam a classe BSP_COMM e herdam os métodos $bsp\_get()$ e $bsp\_put()$ para permitir a consulta e a manipulação dos dados pelos demais objetos. Semelhante ao modelo BSP, toda a comunicação em MulticoreBSP é efetiva na etapa da barreira. Nesse sentido, múltiplas alterações e consultas em uma mesma variável durante uma mesma superetapa resultam em apenas uma das operações sendo efetivadas.

3.3 Bibliotecas Java para Alto Desempenho

Apesar do crescente interesse na adoção de Java para a computação de alto desempenho, a linguagem apresenta algumas lacunas para a escrita de aplicações para esse propósito (TABO-

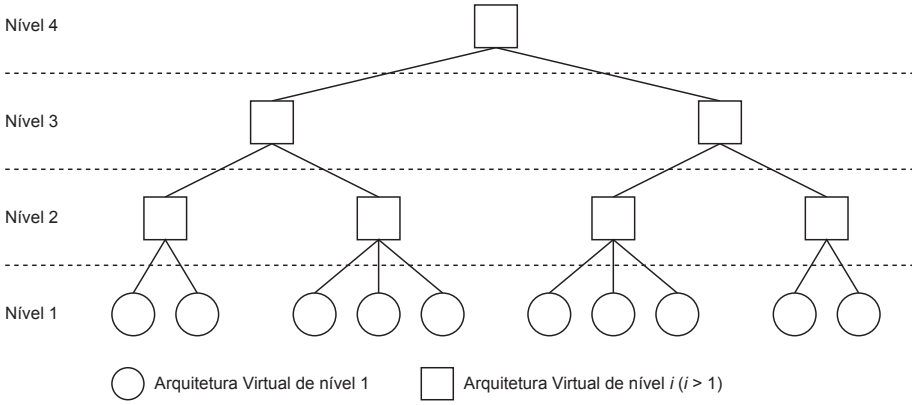

Figura 9: Organização do sistema JavaSymphony (FAHRINGER; JUGRAVU, 2005)

ADA; TOURIÑO; DOALLO, 2009). Nesse sentido, podemos citar a ausência de recursos para a comunicação assíncrona transparente, programação em grupo, tolerância a falhas e migração de objetos. Dessa realidade, surgem sistemas Java voltados para grades computacionais que procuram expor uma interface para cobrir tais características. Nesse contexto, podemos citar sistemas como o JavaSymphony (FAHRINGER; JUGRAVU, 2005), o JavaParty (HÜTTER; MOSCHNY, 2008) e ProActive (CAROMEL; KLAUSER; VAYSSIERE, 1998).

O JavaSymphony é uma biblioteca Java que permite ao programador controlador o paralelismo, o balanceamento de carga e a localidade de objetos em alto nível (FAHRINGER; JUGRAVU, 2005). Ela é desenvolvida no Instituto de Ciência de Software da Universidade de Viena, na Áustria, e pode ser utilizada em ambientes de execução como agregados ou grades computacionais. JavaSymphony suporta balanceamento de carga, migração de objetos e o mapeamento de objetos para recursos de forma explícita e implícita. Para tanto, ela introduz o conceito de arquiteturas distribuídas virtuais e dinâmicas, chamadas de VAs (*Virtual Architectures*) no decorrer dessa seção.

As VAs permitem a organização dos elementos do sistema distribuído de forma hierárquica. Com isso, JavaSymphony pode realizar melhor o balanceamento de carga entre os nós e estabelecer características individuais para cada integrante. A Figura 9 apresenta a organização de um conjunto de VAs em um sistema distribuído. O nível 1 corresponde a um simples nó de computação, como uma estação de trabalho com um ou vários processadores. O nível 2 se refere a um agregado composto de nós presentes no nível abaixo. O nível 3 define uma rede geograficamente distribuída conectada por várias VAs do nível 2, e assim por diante. Dessa forma, um nível i, onde $i \geq 2$, representa um agregado de elementos de nível $i - 1$, que pode incluir arquiteturas heterogêneas largamente distribuídas. JavaSymphony pode distribuir objetos em diferentes VAs. Para tanto, é necessário encapsulá-los em objetos JavaSymphony para torná-los acessíveis remotamente. A biblioteca pode transformar um objeto normal Java em um objeto JavaSymphony ou criar um objeto JavaSymphony diretamente.

A biblioteca JavaSymphony também oferece chamadas RMI síncronas, assíncronas não bloqueante e unidirecionais (*one-sided*). Os três modelos de RMI suportados por esse sistema pos-

suem uma assinatura similar: o nome do método seguido da lista de parâmetros. Contudo, o retorno do resultado das chamadas é diferente. A chamada assíncrona retorna um objeto que permite que o programador obtenha os resultados posteriormente. A comunicação *one-sided* é uma mensagem só de ida e não retorna resultados. As chamadas de métodos sobre um objeto do JavaSymphony devem necessariamente usar um método específico da biblioteca: $sinvoke()$, $ainvoke()$ ou $oinvoke()$. Respectivamente, eles representam comunicação RMI síncrona, assíncrona e comunicação *one-sided*. Em adição, o JavaSymphony permite que objetos sejam migrados para outras VAs de forma automática ou controlada pelo programador. Os objetos podem ser migrados durante a execução da aplicação, entretanto, JavaSymphony verifica antes se algum método do objeto está atualmente em execução. Se sim, a migração é adiada até que todas as invocações tenham sido concluídas. Caso contrário, o objeto é migrado imediatamente.

A biblioteca JavaParty estende a linguagem Java através de um pré-processador e um ambiente de execução de aplicações paralelas e distribuídas para agregados de computadores (HÜTTER; MOSCHNY, 2008). Ela introduz a palavra-chave $remote$ na linguagem Java para identificar os objetos que devem ser distribuídos pela rede. Tais objetos são registrados em um espaço de endereçamento compartilhado. Dessa forma, JavaParty permite que objetos de classes remotas localizados em diferentes máquinas tenham seus métodos e propriedades acessados de forma transparente e semelhante a objetos convencionais de Java. Esse espaço de endereçamento é gerenciado de forma centraliza pelo *RuntimeManager*. Somado a isso, cada máquina do agregado executa um componente chamado *LocalJP*, que é registrado no *RuntimeManager*. O *RuntimeManager* conhece todos os *LocalJPs* e também a localização de todas as classes. Essa informação é replicada para cada *LocalJP* com o objetivo de reduzir a carga do *RuntimeManager*.

Para alcançar seus objetivos, JavaParty utiliza um pré-processador que analisa os arquivos fontes e gera um código Java convencional (PHILIPPSEN; ZENGER, 1997). Dessa forma, a biblioteca não impõe nenhuma modificação na máquina virtual Java. JavaParty também se destaca por implementar primitivas que permitem a migração de objetos. A migração é possível somente se nenhum método estiver sendo executado sobre o objeto. Ao mover-se, o objeto remoto deixa um procurador no seu local de origem, o qual atenderá eventuais chamadas que ainda não tenham sido direcionadas para o novo endereço. No atendimento da chamada, o procurador lança uma exceção que contém o novo endereço do objeto remoto. No tratamento da exceção, no lado do cliente, o JavaParty atualiza o endereço da referência e redireciona a chamada de forma transparente.

O ProActive, por sua vez, é uma biblioteca Java de código livre para a computação paralela (CAROMEL; KLAUSER; VAYSSIERE, 1998). Ela é desenvolvida pelo Instituto INRIA, na França, em parceria com a empresa ActiveEon e permite a escrita de aplicações distribuídas as quais o ambiente de execução pode ser redes locais, agregados ou grades computacionais. Dentre as empresas que utilizam ProActive, pode-se citar a Renault e a Hewlett-Packard, que empregam a biblioteca para o desenvolvimento de soluções para grades computacionais e com-

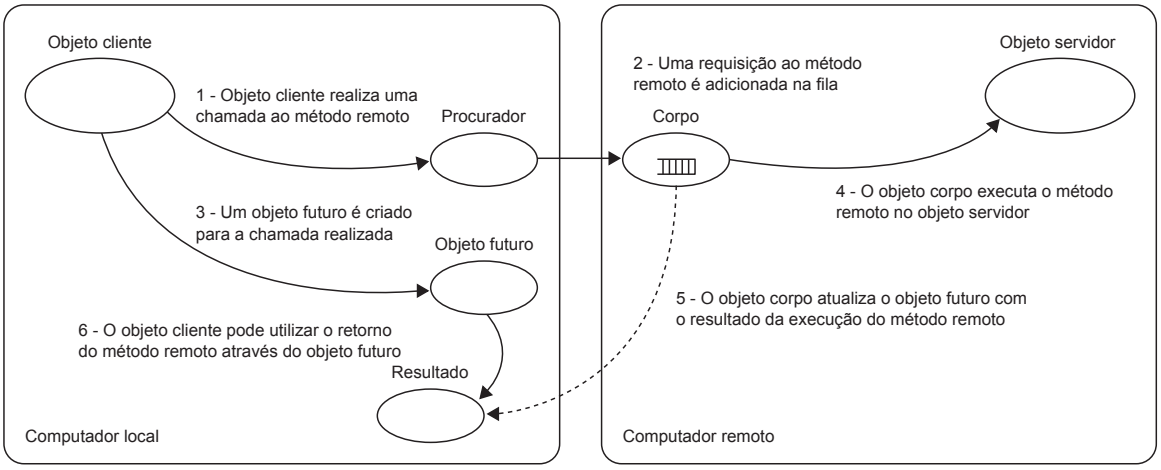

Figura 10: Comunicação assíncrona em ProActive (BADUEL; BAUDE; CAROMEL, 2002)

putação nas nuvens.

A biblioteca ProActive é construída inteiramente utilizando as classes padrões de Java (CAROMEL; KLAUSER; VAYSSIERE, 1998). O ProActive não impõe modificações no código fonte da distribuição da máquina virtual ou a necessidade de compiladores ou pré-processadores especiais. O principal conceito inerente ao desenvolvimento de ProActive está relacionado ao tratamento de objetos ativos. Um objeto ativo é análogo a um objeto remoto. Ele pode ser acessível remotamente e é composto por um objeto padrão Java e um fluxo de execução chamado corpo. O corpo não é visível ao usuário e é encarregado de tratar das operações de recepção e invocação de métodos do objeto associado, do armazenamento destas em uma fila de requisições e do envio dos resultados para os chamadores de cada requisição. Um objeto ativo pode ser instanciado em qualquer das máquinas envolvidas para a solução da aplicação e a sua manipulação é igual a dos objetos normais do Java.

O ProActive é capaz de realizar a invocação de métodos remotos de maneira assíncrona. Para tal, ela faz uso dos recursos de objetos futuros e de espera pela necessidade (CAROMEL, 1993). Um objeto futuro compreende o retorno imediato da invocação de um método sobre um objeto ativo. Ele representa o resultado da chamada de um método ainda não processado. Quando o processamento remoto é finalizado, o objeto futuro é automaticamente substituído pelo objeto que representa o resultado de fato. A Figura 10 apresenta o fluxo de execução da invocação de um método remoto. Do ponto de vista da aplicação, não existe diferença entre um objeto futuro e aquele que realmente representa o resultado da chamada. Dessa forma, o ProActive permite que a aplicação local continue o seu processamento enquanto ocorre o processamento remoto. Caso o programa local efetue alguma operação no objeto futuro, a aplicação é bloqueada até a chegada efetiva do resultado da invocação. Essa política de sincronização caracteriza o recurso de espera pela necessidade. Em adição, o ProActive gere a comunicação assíncrona de forma transparente. Nesse sentido, sempre que possível, a biblioteca executará métodos de objetos ativos de forma assíncrona sem a intervenção do programador.

Além da comunicação assíncrona, ProActive possibilita que objetos de uma mesma classe

possam ser agrupados (BADUEL; BAUDE; CAROMEL, 2005). A organização dos objetos em grupos favorece o desenvolvimento de aplicações paralelas por oferecer uma abstração quanto ao acesso a métodos e campos desses objetos. O ProActive implementa a técnica de programação SPMD e permite que o programador manipule um grupo de maneira semelhante a um objeto Java convencional. Para tanto, o objeto do grupo deve possuir a mesma interface de seus membros. Além disso, ProActive permite o lançamento e a execução simultânea de objetos ativos em diferentes máquinas ou processadores. Nesse cenário, cada objeto membro do grupo possui um identificador, o que possibilita a comunicação entre eles.

Ainda sobre as características do ProActive, a biblioteca oferece migração de objetos para outras máquinas e facilidade no mapeamento de objetos para recursos. O recurso de migração habilita a transferência de objetos para outras máquinas do sistema distribuído (BAUDE et al., 2000). Ele permite move-los para um novo nó ou para um nó no qual outro objeto está em execução. Para tal, é deixado um procurador na máquina origem que é usado para realizar chamadas ao objeto de forma transparente. Quando o objeto transferido recebe um pedido, ele imediatamente envia a sua nova localização para o chamador. No que diz respeito ao mapeamento de objetos para recursos, ProActive permite que os desenvolvedores indiquem os recursos computacionais disponíveis na rede e aqueles necessários para a execução das aplicações. O mapeamento deve ser descrito em arquivos no formato XML e elimina a necessidade de o programador especificar nomes de máquinas e protocolos de comunicação no código fonte (BAUDE et al., 2002). Assim, é possível executar qualquer aplicação escrita em ProActive em diferentes ambientes sem alterações no código da mesma.

3.4 Balanço

Considerando os trabalhos relacionados apresentados nesse capítulo, podemos destacar alguns aspectos positivos dos sistemas e algoritmos estudados. Relacionado aos temas escalonamento e balanceamento de carga, observa-se que a combinação de diversas métricas (como computação e comunicação) para calcular a carga do ambiente e selecionar os candidatos à migração mostra-se importante em ambientes dinâmicos e heterogêneos. Essa combinação pode ser vista nos trabalhos de Jiang et al. (2007) e Righi et al. (2010). Jiang et al. (2007) desenvolveram o modelo ServiceBSP, que observa a capacidade e o consumo de processamento, memória e largura de banda dos diversos recursos para decidir onde lançar as tarefas. O algoritmo de balanceamento de carga de ServiceBSP atua no lançamento da aplicação e mantém o mapeamento inicial de processos para recursos até a conclusão das tarefas. Essa abordagem pode não ser eficiente em ambientes dinâmicos uma vez que a alocação dos processos não é alterada em função do comportamento da aplicação e/ou infraestrutura. Nesse sentido, Righi et al. (2010) desenvolveram o modelo MigBSP que trabalha com múltiplas métricas e trata a realocação automática de processos durante a execução da aplicação. O ato de realizar o balanceamento de carga de forma implícita é pertinente uma vez que não são necessárias modificações no código

Tabela 4: Comparativo entre as bibliotecas de programação BSP

Biblioteca	Linguagem de programação	Migração explícita	Migração implícita	Comunicação assíncrona	Comunicação *one-sided*
BSPlib	C				•
PUB	C		•	•	•
BSPonMPI	C				•
PUBWCL	Java		•		•
JBSP	Java				•
MulticoreBSP	Java				•

da aplicação. Além disso, não há a necessidade de interação do programador com o ambiente de execução.

O modelo de aplicação escolhido para o desenvolvimento de jMigBSP é o BSP. Esse modelo é atraente para ambientes dinâmicos e heterogêneos (CAMARGO; KON; GOLDMAN, 2005). Para tanto, considerando que o modelo é originalmente planejado para ambientes homogêneos e dedicados, a literatura propõe duas estratégias (SONG; TONG; ZHI, 2006): (i) desenvolver uma plataforma que suporte a execução de aplicações BSP em ambientes dinâmicos e heterogêneos e; (ii) aplicar modificações no modelo BSP para ajustá-lo para um ambiente de destino. PUBWCL (BONORDEN; GEHWEILER; HEIDE, 2006), BSPonMPI (SUIJLEN; BISSELING, 2011) e JBSP (GU; LEE; CAI, 2001) são exemplos da primeira abordagem. Essas implementações não mudam o conceito original do modelo BSP. Elas utilizam padrões e arquiteturas consolidadas (no caso, o MPI e a máquina virtual Java) para abstrair questões de dinamicidade e heterogeneidade. Por outro lado, modelos como HBSP e Dynamic BSP são exemplos de extensões que apresentam adaptações para ambientes heterogêneos, no caso do primeiro, e tolerância a falhas, no caso do Dynamic BSP.

No contexto de aplicações BSP, podemos citar duas iniciativas para a migração de processos. A primeira descreve a biblioteca PUBWCL (BONORDEN; GEHWEILER; HEIDE, 2006). PUBWCL tira proveito do poder computacional ocioso de computadores distribuídos pela Internet. Os algoritmos de balanceamento de carga de PUBWCL podem realizar a migração de processos durante a execução de uma superetapa bem como no seu final. Para tanto, são utilizadas informações como o tempo de conclusão médio de cada superetapa. Outros trabalhos incluem uma extensão da biblioteca PUB para apoiar a migração de processos (BONORDEN, 2007). Como no trabalho anterior, esse também considera a capacidade de processamento dos computadores e o tempo de computação. Além disso, PUB realiza a migração de processos no fim de uma superetapa, onde o impacto da modificação é observado na próxima. Diferente do modelo apresentado por Righi et al. (2010), as estratégias de Bonorden et al. (2007) não levam em consideração a comunicação entre os processos nem os custos de migração. A Tabela 4 resume as principais características das bibliotecas BSP estudadas neste capítulo.

Depois de observar os trabalhos relacionados descritos nesse capítulo, vislumbra-se a lacuna de um sistema capaz de atuar em ambientes dinâmicos, heterogêneos e com suporte ao

reescalonamento implícito e explícito de processos. Nesse contexto, o sistema aqui apresentado adotará em seu desenvolvimento o modelo MigBSP para suportar ambientes dinâmicos. MigBSP se destaca dos demais trabalhos por considerar múltiplas métricas e tratar a realocação de processos durante a execução da aplicação. Em adição, o modelo considera a comunicação entre os processos e o custo de migração deles. Relacionado a heterogeneidade, será adotada a linguagem Java em virtude de sua portabilidade e por esta ser uma opção emergente para a computação de alto desempenho (TABOADA; TOURIÑO; DOALLO, 2009). Em adição, o desenvolvimento de jMigBSP será suportado pela biblioteca ProActive (CAROMEL; KLAUSER; VAYSSIERE, 1998). A escolha de ProActive reflete as suas facilidades e as constantes atualizações do projeto em relação as demais ferramentas estudadas. Nesse sentido, recursos como migração de objetos, comunicação assíncrona e o estilo de programação SPMD auxiliarão no desenvolvimento de jMigBSP.

4 SISTEMA JMIGBSP

Os capítulos anteriores servem de base para entender as decisões de projeto do sistema jMigBSP (GRAEBIN; RIGHI, 2011). Ele foi desenvolvido para proporcionar uma interface de programação para a escrita de aplicações BSP em Java. Em especial, sua principal contribuição diz respeito às facilidades de reescalonamento em duas maneiras:

- Com a intervenção do programador: usando diretivas de migração no código da aplicação;
- Sem a intervenção do programador: através do balanceamento de carga automático em nível de *middleware*.

O uso de diretivas de migração no código da aplicação permite ao programador decidir quando transferir um objeto para outro processador. Nessa abordagem, o reescalonamento ocorre na fase de computação e requer que o programador possua experiência em algoritmos de balanceamento de carga para um melhor aproveitamento dos recursos computacionais. Por outro lado, o reescalonamento em nível de *middleware* representa uma extensão da biblioteca de programação para oferecer um mecanismo transparente para o usuário. Nessa estratégia, o remapeamento de objetos ocorre após a barreira de sincronização de forma implícita.

Esse capítulo apresenta o sistema jMigBSP e está organizado da forma que segue. A Seção 4.1 descreve em linhas gerais as decisões de projeto que norteiam o desenvolvimento de jMigBSP. A Seção 4.2 apresenta a interface de programação de jMigBSP. Essa seção também descreve o modelo de comunicação empregado e uma aplicação de exemplo. A Seção 4.3 apresenta as estratégias de jMigBSP para oferecer reescalonamento em nível de aplicação e de *middleware*. Em especial, essa seção detalha a implementação do modelo MigBSP em jMigBSP. Por fim, a Seção 4.4 resume o capítulo e enfatiza os principais tópicos abordados.

4.1 Decisões de Projeto

Como apresentado na introdução do presente capítulo, o sistema jMigBSP foi construído para possibilitar a escrita de aplicações BSP em Java. Sua principal contribuição está em oferecer reescalonamento em dois níveis: (i) aplicação e; (ii) *middleware*. Em especial, essa última ideia é alcançada com a implementação do modelo MigBSP em jMigBSP. O modelo MigBSP controla o reescalonamento de processos BSP de acordo com o estado do sistema. Segundo a taxonomia de Casavant e Kuhl (1988) (ver Seção 2.1), MigBSP pode ser classificado como dinâmico e global. O item dinâmico considera que as informações para o reescalonamento dos processos são coletadas em tempo de execução da aplicação. Além disso, as decisões quanto ao escalonamento são distribuídas entre os vários processos que cooperam entre si a fim de melhorar a utilização dos recursos. Dessa forma, o modelo realiza um escalonamento fisicamente distribuído e cooperativo. Para alcançar essas ideias, o sistema jMigBSP captura todos

os dados necessários referente ao escalonamento diretamente nas fases de comunicação e barreira, além de outras fontes, como por exemplo, o sistema operacional. A captura desses dados ocorre durante a execução das aplicações de forma transparente para o programador. Assim, as informações capturadas por jMigBSP tornam-se os dados de entrada para o modelo MigBSP, que decidirá quando lançar o reescalonamento dos objetos, quais objetos serão candidatos à migração e para onde os objetos selecionados serão transferidos.

O desenvolvimento de jMigBSP é suportado pela biblioteca ProActive. O sistema jMigBSP herda características dessa biblioteca e propõe novos mecanismos para seguir o estilo de programação BSP. Dentre esses mecanismos, podemos destacar a semântica de comunicação do modelo BSP, onde as mensagens enviadas em uma superetapa são recebidas pelos processos somente na superetapa seguinte (GAVA; FORTIN, 2009). Quanto as características que jMigBSP herda de ProActive, podemos citar: (i) a facilidade no mapeamento de objetos para recursos; (ii) o modelo de programação SPMD; (iii) a migração de objetos e; (iv) o modelo de comunicação assíncrona. A primeira ideia permite que o programador indique os recursos computacionais existentes na rede e aqueles necessários para a execução das aplicações. O mapeamento deve ser descrito em arquivos no formato XML e elimina a necessidade de especificar nomes de máquinas e protocolos de comunicação no código da aplicação. Assim, torna-se possível executar qualquer aplicação escrita com jMigBSP em diferentes ambientes sem a necessidade de realizar alterações no código da mesma. Em adição, jMigBSP é construído inteiramente utilizando as classes padrões da linguagem Java. Nesse sentido, jMigBSP não impõe modificações na máquina virtual ou a necessidade de compiladores especiais ou pré-processadores.

O sistema jMigBSP segue o estilo de programação SPMD. A justificativa para essa adoção deve-se ao fato de esta ser a abordagem mais utilizada para a escrita de aplicações BSP (SKILLICORN; HILL; MCCOLL, 1997). Além disso, jMigBSP tira proveito das facilidades de ProActive para a migração de objetos. Com isso, jMigBSP oferece métodos para a migração explícita de objetos para outras máquinas do sistema distribuído. Esses métodos permitem transferir um objeto para um novo nó ou para um nó no qual outro objeto está em execução no momento. Por fim, jMigBSP possui a capacidade de gerir comunicação assíncrona. A comunicação assíncrona é entendida da seguinte forma: o transmissor não espera (não bloqueante) pela chegada da sua mensagem no objeto receptor. O fluxo de execução no transmissor retorna para a aplicação tão logo a requisição da troca de mensagem é analisada. Para expressar o assincronismo, jMigBSP faz uso dos recursos de objetos futuros e de espera pela necessidade de ProActive.

4.2 Programando com jMigBSP

A seção que segue apresenta em detalhes os métodos de jMigBSP para a escrita de aplicações BSP. Ela aborda questões como o modelo de comunicação utilizado e os mecanismos para a migração objetos. Para o desenvolvimento desses temas, priorizou-se a simplicidade no

Tabela 5: Conjunto de métodos de jMigBSP

Classificação	Método	Descrição
Consulta	$bsp\_pid()$	Retorna o código identificador do objeto no grupo.
	$bsp\_nprocs()$	Retorna a quantidade de objetos no grupo.
Superetapa	$bsp\_sync()$	Barreira de sincronização.
Comunicação	$bsp\_put()$	Copia dados para a memória do objeto remoto.
	$bsp\_get()$	Copia dados da memória do objeto remoto.
Reescalonamento	$bsp\_migrate()$	Migra o objeto chamador para outro local.

uso de jMigBSP. Nesse sentido, jMigBSP se propõe a facilitar a escrita de aplicações paralelas quando comparado com ProActive. Para tanto, todas as exigências impostas por essa biblioteca são contempladas por jMigBSP. Por exemplo, ProActive impõe que o programador invoque rotinas para carregar os arquivos descritores do ambiente e gerenciar a criação dos objetos em processadores. Além disso, o uso de anotações em classes e métodos é recomendado para evitar exceções em tempo de execução da aplicação (CUNHA; SOBRAL, 2007). Esses requisitos são contemplados nas classes e métodos de jMigBSP, que tornam transparente o fato dele ser desenvolvido com ProActive.

4.2.1 Interface de Programação

O sistema jMigBSP se propõe a oferecer uma interface simples para a escrita de aplicações BSP. Sendo assim, para escrever uma aplicação utilizando jMigBSP, o programador precisa apenas estender a classe $jMigBSP$ e implementar o método $run()$. Esse método deve conter o código da aplicação que será executado em paralelo. Considerando que uma aplicação escrita em jMigBSP é, em essência, uma classe, para usá-la, uma instância dela precisa ser criada. Entretanto, criar a instância de uma classe escrita em jMigBSP não implica em sua execução paralela. Para tanto, o método $start()$ deve ser invocado. Esse método recebe como parâmetro um número inteiro que representa a quantidade de objetos que serão criados em paralelo. Em adição, o método $start()$ lê os arquivos descritores do ambiente e realiza o mapeamento inicial de objetos para recursos.

O estilo de programação SPMD é adotado em jMigBSP. Nesse sentido, cada objeto executa exatamente o mesmo trecho de código definido no método $run()$. jMigBSP oferece um conjunto de métodos que permitem a interação entre esses objetos e que podem ser utilizados na implementação do método $run()$. Esses métodos são apresentados na Tabela 5. Os métodos $bsp\_nprocs()$ e $bsp\_pid()$ permitem, respectivamente, consultar a quantidade de objetos paralelos e o código identificador de cada objeto no grupo. O método $bsp\_sync()$ implementa uma barreira de sincronização e identifica o término de uma superetapa e o início da próxima. Esse método garante que os dados recebidos durante uma superetapa estarão disponíveis para uso somente no início da próxima, seguindo assim as premissas do modelo BSP.

O sistema jMigBSP implementa o modelo de comunicação *one-sided* através dos métodos

$bsp\_put()$ e $bsp\_get()$. O método $bsp\_put()$ grava dados na memória do objeto remoto sem a intervenção direta dele. Esse método possui um comportamento assíncrono, permitindo que o programador execute computações úteis enquanto é realizado o envio dos dados. Em aspectos de implementação, todos os objetos possuem dois vetores que atuam como *buffers* para dar suporte a semântica de comunicação do modelo BSP. Um vetor possui as mensagens que foram recebidas na superetapa anterior, enquanto o outro atua como receptor das mensagens na superetapa corrente. O primeiro vetor é chamado de Buffer Ativo, enquanto o segundo de Buffer Temporário. Além disso, ambos os vetores possuem tamanho $n - 1$, onde n é a quantidade de objetos paralelos. Isso significa que cada objeto possui uma área de memória reservada para receber mensagens dos demais. Assim, diferente de BSPlib, jMigBSP não impõe que o programador registre variáveis para habilitar a comunicação *one-sided* entre os objetos. Caso um objeto emissor envie duas mensagens para um mesmo objeto em uma mesma superetapa, somente uma das operações será efetivada na superetapa subsequente. Após o fim de uma superetapa, o Buffer Ativo é preenchido com o conteúdo do Buffer Temporário. Dessa forma, a operação $bsp\_get()$ atinge o Buffer Ativo do objeto remoto a fim de copiar os valores mantidos ali. Em contra partida, a operação $bsp\_put()$ atinge o Buffer Temporário.

A Figura 11 ilustra a semântica de comunicação de jMigBSP. Nela, o objeto $o1$ envia uma mensagem m para o objeto $o2$ usando o método $bsp\_put()$. Essa operação ocorre na primeira superetapa, onde a mensagem recebida é armazenada no Buffer Temporário do objeto $o2$. Ainda na primeira superetapa, o objeto $o1$ realiza uma operação de leitura na memória de $o2$ usando o método $bsp\_get()$. Essa operação atinge o Buffer Ativo do objeto de destino. Nesse sentido, os dados retornados são referentes às mensagens trocadas na superetapa anterior, e não os dados referentes à operação $bsp\_put()$ realizada na superetapa corrente. Ainda referente a Figura 11, o conteúdo do Buffer Ativo é substituído pelo Buffer Temporário durante a barreira de sincronização. Dessa forma, a operação $bsp\_get()$ realizada na segunda superetapa retornará os dados enviados do objeto $o1$ para $o2$ na superetapa anterior.

4.2.2 Exemplo de Programa

A Figura 12 ilustra o uso das funções de jMigBSP. Ela demonstra a implementação do algoritmo da soma de prefixos onde cada processo p possui um número diferente para o cálculo. O algoritmo usa a técnica logarítmica que computa log_p superetapas. Sendo assim, os processos na faixa $2^{k-1} \leq i \leq p$ combinam suas somas parciais durante a k*-ésima* superetapa. Após criar uma instância da classe $PrefixSum$, a aplicação invoca o método $start()$ para iniciar a execução paralela do programa. No exemplo da Figura 12, a aplicação é lançada com quatro processos (ver linha 19). O sistema jMigBSP provê transparência de localização. Nesse sentido, os objetos podem estar em execução em diferentes agregados de computadores.

A Figura 13 apresenta as superetapas envolvidas na resolução do algoritmo apresentado. Cada objeto inicia com o seu próprio identificador. Na primeira superetapa, cada objeto envia

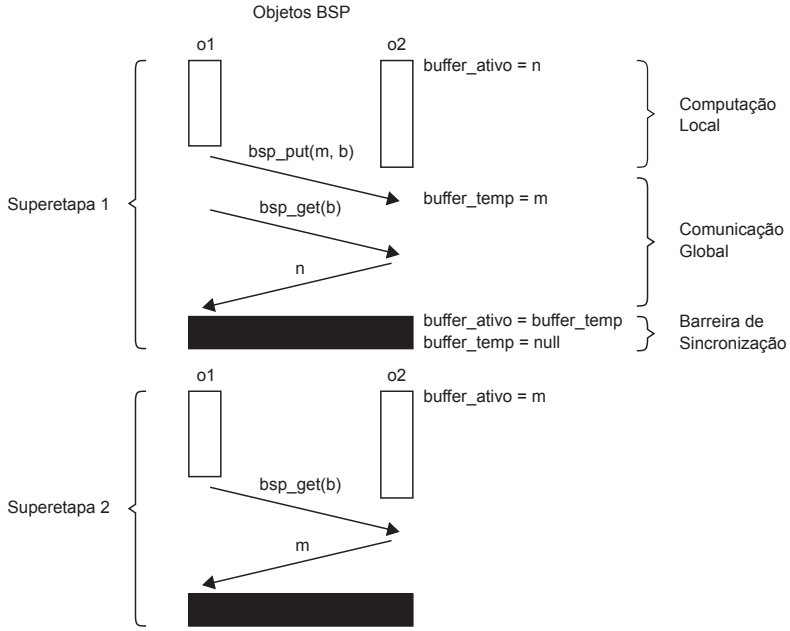

Figura 11: Semântica de comunicação de jMigBSP

```
1.  public class PrefixSum extends jMigBSP
2.  {
3.     public void run()
4.     {
5.        int n = bsp_pid() + 1;
6.        for (int i = 1; i < bsp_nprocs(); i *= 2)
7.        {
8.           if (bsp_pid() + i < bsp_nprocs())
9.           {
10.             bsp_put((Object) n, bsp_pid() + i);
11.          }
12.          bsp_sync();
13.          n = n + getBuffer(bsp_pid() - i);
14.       }
15.    }
16.    public static void main(String[] args)
17.    {
18.       PrefixSum s = new PrefixSum();
19.       s.start(4);
20.    }
21. }
```

Figura 12: Algoritmo da soma de prefixos implementado em jMigBSP

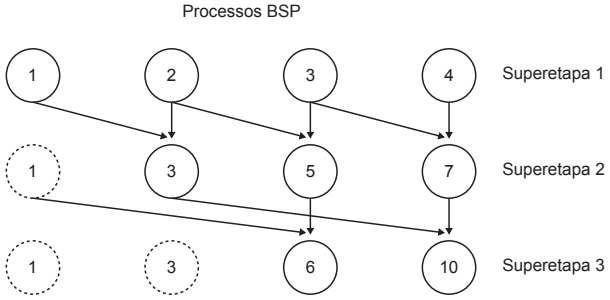

Figura 13: Operação da soma de prefixos utilizando a técnica logarítmica

o seu valor para o seu vizinho da direita. Assim, o objeto emissor da mensagem irá colocá-la no Buffer Temporário do objeto destino. Nessa etapa, todos os objetos (exceto o primeiro) possuirão um número que representa a soma de dois inteiros. Seguindo o algoritmo, o último objeto terá o resultado final da computação na terceira superetapa.

4.3 Flexibilidade no Reescalonamento de Objetos

O reescalonamento permite o remapeamento inicial de processos para recursos em resposta ao comportamento da aplicação. O uso dessa técnica é pertinente em grades computacionais, uma vez que a disponibilidade dos recursos pode variar de forma significativa ao longo do tempo (CHEN; ZHU; AGRAWAL, 2006). Nesse sentido, a técnica de reescalonamento pode ser empregada para adaptar o escalonamento corrente ou gerar um novo durante a execução da aplicação (TAN; AUFENANGER, 2011). O reescalonamento de objetos em jMigBSP pode ocorrer em dois níveis: (i) usando diretivas de migração no código da aplicação e; (ii) através do balanceamento de carga automático em nível de *middleware*. As próximas subseções detalham o uso de ambas as técnicas.

4.3.1 Reescalonamento Explícito: Gerenciando Diretivas de Migração dentro da Aplicação

O sistema jMigBSP permite a migração de qualquer objeto BSP entre diferentes JVMs através da chamada explícita ao método $bsp\_migrate()$. Essa abordagem flexibiliza a escrita de aplicações, uma vez que o programador pode escolher os pontos para a chamada da migração de acordo com as características da aplicação, podendo assim implementar o seu próprio algoritmo de reescalonamento. O sistema jMigBSP oferece duas implementações para o método $bsp\_migrate()$. A primeira permite migrar o objeto chamador para um computador remoto. Ela recebe como parâmetro um objeto do tipo $Node$, que é a referência de ProActive para uma JVM que está em execução em um processador. Essa estratégia é útil quando sabemos que o computador de destino está levemente carregado e possui recursos computacionais suficientes para a execução da aplicação. A outra maneira consiste em transferir o objeto chamador para um computador remoto no qual outro objeto está sendo executado. Para tanto, o método $bsp\_migrate()$ deve receber como parâmetro de entrada o código identificador do objeto de referência. Tal abordagem é pertinente para aproximar os objetos que se comunicam com frequência. Em especial, o uso dessa técnica em redes lentas pode reduzir o tempo de execução da aplicação, uma vez que o custo de comunicação será menor.

Em termos de aplicação BSP, uma maneira trivial para o lançamento do reescalonamento é colocar diretivas de migração após a barreira. Isso porque esse ponto representa um estado global consistente, fazendo com que a implementação da migração e da captura dos dados de escalonamento sejam fáceis. Essa última sentença é argumentada pelo fato de que o início de

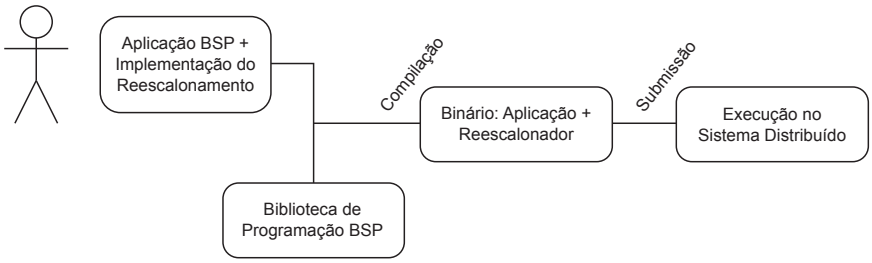

Figura 14: Acoplamento entre o código da aplicação e o algoritmo de escalonamento na abordagem explícita

uma superetapa permite a tomada de decisão através de um conhecimento global. Em outras palavras, dados atualizados de todos os processos e nós podem ser capturados na barreira de sincronização para o remapeamento (KWOK; CHEUNG, 2004).

O reescalonamento explícito requer que o programador possua experiência em algoritmos de balanceamento de carga. Isso porque, além dos pontos de uma superetapa, deve-se escolher quais delas possuirão chamadas para a migração. Dessa forma, o programador deve coletar dados sobre a capacidade e a carga dos processadores para a tomada de decisão sobre a realocação manual dos objetos. Apesar de essa abordagem oferecer flexibilidade quanto ao momento da migração, o seu uso impõe um acoplamento entre o código da aplicação e a implementação do reescalonamento. A Figura 14 ilustra essa dependência. Nesse sentido, uma nova aplicação e/ou infraestrutura de recursos exigirá um novo esforço em estudar os melhores lugares para adicionar chamadas de migração, bem como para escolher superetapas pertinentes para isso.

4.3.2 Migração Implícita: Balanceamento de Carga Automático em Nível de Middleware

Além da migração explícita, o sistema jMigBSP oferece balanceamento de carga automático sem requerer a intervenção do programador. Para tanto, o modelo de reescalonamento MigBSP é implementado e oferecido através de jMigBSP. O modelo MigBSP funciona sobre uma arquitetura que é montada com a ideia de Conjuntos (diferentes agregados) e Gerentes de Conjuntos. Gerentes de Conjuntos são responsáveis pelo escalonamento, pela captura de dados de um Conjunto e pela troca dessas informações com os demais Gerentes. A decisão para o reescalonamento de processos é tomada no final de uma superetapa. Com o intuito de gerar menos intrusividade na aplicação, MigBSP aplica adaptações que controlam o intervalo entre as superetapas. A ideia básica é ampliar esse índice caso os processos estejam balanceados ou reduzi-lo, caso contrário.

A Figura 15 ilustra duas superetapas em diferentes situações. Nosso objetivo é fazer jMigBSP reagir contra a dinamicidade da aplicação e dos recursos através da migração de objetos após a barreira de sincronização. Assim, o objetivo final é reduzir o tempo da aplicação, tornando as superetapas mais curtas (ver Superetapa 2 da Figura 15). O reescalonamento automático em jMigBSP é oferecido através da classe $LBjMigBSP$. Essa classe estende $jMigBSP$ a

fim de adicionar rotinas para a captura de dados para o escalonamento. A Figura 16 ilustra os métodos que LBjMigBSP reimplementa a partir de jMigBSP. As linhas entre 3 e 19 mostram a implementação dos métodos de comunicação. Eles capturam informações para o reescalonamento e acionam os métodos de jMigBSP para o tratamento da comunicação. Por exemplo, o método $bsp\_put()$ captura o tempo de sistema antes e após efetivar o envio dos dados. A diferença de ambos os tempos é considerado o tempo de comunicação entre os objetos de origem e destino. O método $bsp\_sync()$ captura o tempo de execução das superetapas e os armazena em uma estrutura de dados (ver linhas entre 23 e 25). Quando o reescalonamento é ativado, o método $computeBalance()$ troca o seu vetor de tempos com os demais Gerentes de Conjunto, permitindo-lhes conhecer o próximo intervalo de superetapas para o reescalonamento de objetos. Depois de obter o PM, a viabilidade de migração é testada.

A Figura 17 ilustra nossa ideia para oferecer balanceamento de carga automático. O programador não precisa alterar qualquer linha de código da aplicação. Ele simplesmente precisa compilar a aplicação com a biblioteca que oferece tanto o modelo de reescalonamento quanto a interface para a escrita de aplicações BSP. O resultado é um binário que pode ser executado no sistema distribuído, juntamente com a implementação do modelo. Para utilizar o recurso de balanceamento de carga automático, o programador precisa apenas alterar a linha que define a classe da aplicação (por exemplo, a linha 1 da Figura 12). Assim, ao invés de estender a classe $jMigBSP$, a aplicação deve estender $LBjMigBSP$. Nesse sentido, LBjMigBSP representa nossa abordagem de *middleware*, agindo como um *wrapper* para o balanceamento de carga automático. Para tanto, LBjMigBSP calcula o objeto potencial de migração através da combinação das métricas de Computação, Comunicação e Memória de MigBSP. A métrica Computação considera as instruções realizadas entre as superetapas, bem como o tempo gasto para as ações de computação dentro de uma superetapa. A métrica Comunicação considera a quantidade de bytes enviados e recebidos de um Conjunto, além da largura de banda para alcançá-lo. Finalmente, a métrica Memória leva em consideração o tamanho do objeto e os custos relacionados à migração do mesmo. A próxima subseção descreve em detalhes a implementação do modelo MigBSP em jMigBSP.

4.3.3 Implementação do Modelo MigBSP

O modelo MigBSP utiliza-se de escalonamento hierárquico com o intuito de otimizar a passagem de informações (RIGHI et al., 2010). Nesse sentido, os nós são reunidos de forma a criar uma abstração de Conjunto. Um Conjunto pode ser uma rede local ou um agregado, e cada Conjunto possui um Gerente de Conjunto. Na implementação do sistema jMigBSP, um Gerente de Conjunto é um objeto Java do tipo $SetManager$. Esse objeto é criado durante o lançamento da aplicação no primeiro nó de cada agregado. A Figura 18 apresenta a interface de métodos públicos da classe $SetManager$. Tais métodos são invocados remotamente pelos objetos BSP sobre a jurisdição do Gerente. Primeiramente, o método $receiveSuperstepData()$

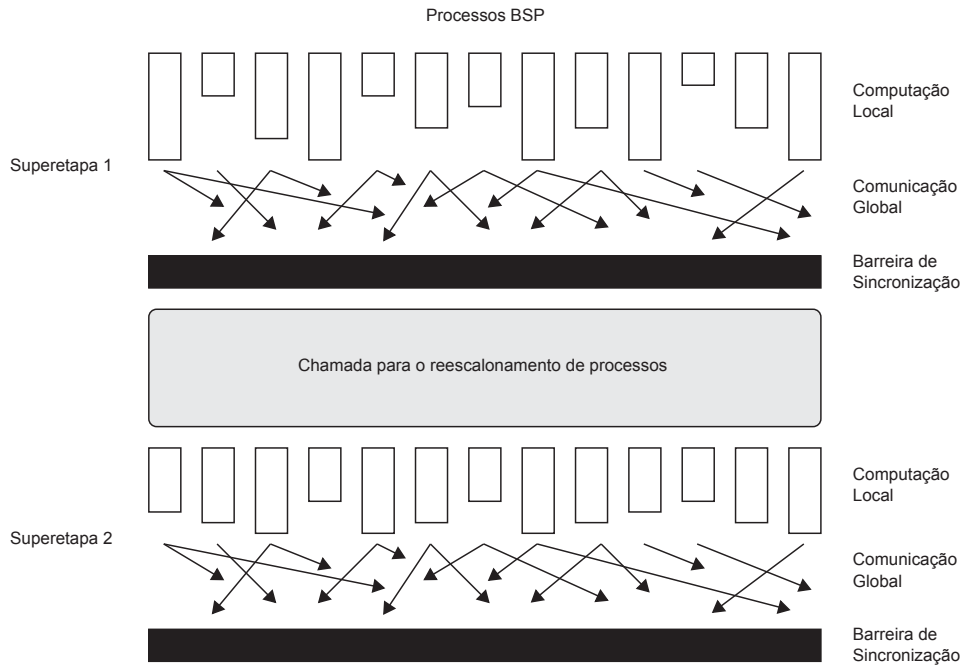

Figura 15: Observando a situação de diferentes superetapas

```
1.  public class LBjMigBSP extends jMigBSP
2.  {
3.     public Object bsp_get(int destination)
4.     {
5.        long t1 = System.nanoTime();
6.        Object data = super.bsp_get(destination);
7.        long t2 = System.nanoTime();
8.        long time = t2 - t1;
9.        addCommunicationData(getCurrentSuperstep(), time, getObjectSize(data));
10.       return data;
11.    }
12.    public void bsp_put(Object data, int destination)
13.    {
14.       long t1 = System.nanoTime();
15.       super.bsp_put(data, destination);
16.       long t2 = System.nanoTime();
17.       long time = t2 - t1;
18.       addCommunicationData(getCurrentSuperstep(), destination, time, getObjectSize(data));
19.    }
20.    public void bsp_sync()
21.    {
22.       super.bsp_sync();
23.       long t2 = System.nanoTime();
24.       long time = t2 - superstepStartTime;
25.       addSuperstepData(getCurrentSuperstep(), bsp_pid(), time);
26.       if (isSuperstepRescheduling())
27.       {
28.          next_call = computeBalance();
29.          exchageDataAmongSetManagers();
30.          if (willMigrate())
31.          {
32.             bsp_migrate(getDestinationNode());
33.          }
34.       }
35.       superstepStartTime = System.nanoTime();
36.    }
37. }
```

Figura 16: Derivando a classe jMigBSP para oferecer balanceamento de carga automático

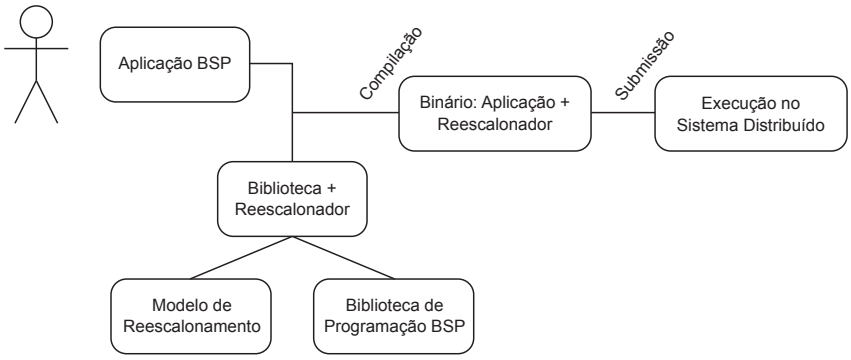

Figura 17: Oferecendo balanceamento de carga automático diretamente na biblioteca de programação BSP

```
1.  public class SetManager implements Serializable
2.  {
3.      public void receiveSuperstepData(int pid, int superstep, long[] times) {}
4.      public int getAlpha() {}
5.      public float[][] getCapacity() {}
6.      public DestinationNode getANode(long bytes, long memory) {}
7.  }
```

Figura 18: Métodos públicos da classe SetManager

é responsável por receber os tempos das superetapas quando o reescalonamento é ativo. Esse método aguarda como parâmetro um vetor com os tempos da superetapa k até a superetapa $k+\alpha-1$[1]. Na medida em que o método é invocado, os tempos das superetapas são armazenados em um vetor local no Gerente. Em seguida, o método $receiveSuperstepData()$ realiza a troca desse vetor com os demais Gerentes de Conjunto. Com isso, todos os Gerentes possuem os tempos de todas as superetapas dos objetos BSP, podendo assim calcular a estabilidade do sistema. O cálculo da estabilidade é realizado pelo método $getAlpha()$. Esse método retorna um número inteiro que representa o novo valor do índice α. O $getAlpha()$ é bloqueante até que todos os Gerentes recebam todos os vetores de tempos para assim calcular a variação de α. Na Figura 16, o método $receiveSuperstepData()$ da classe $SetManager$ é invocado pelo método privado $addCommunicationData()$ (linhas 18 e 25), enquanto que o método $getAlpha()$ é chamado através de $computeBalance()$ (linha 28).

O método $getCapacity()$ retorna o desempenho de todos os Conjuntos. Ao ser invocado, ele recupera a carga de todos os processadores sobre a jurisdição do Conjunto e multiplica esses valores por sua capacidade teórica. Em seguida, todos os Gerentes calculam o desempenho médio de seus Conjuntos e trocam esses valores entre si. Por fim, cada Gerente calcula o desempenho médio de seu Conjunto considerando a capacidade dos demais. O método $getANode()$ devolve o processador mais adequado para receber um objeto candidato a migração. Diferente dos métodos anteriormente citados, o método $getANode()$ não é invocado pelos objetos BSP, mas sim pelos Gerentes. Supondo que o objeto i do Conjunto j_1 é candidato a ser transferido para o Conjunto j_2. Nesse caso, o Gerente de Conjunto j_1 invoca o método $getANode()$ do Gerente j_2. Nessa etapa, o método $getANode()$ calcula o processador com o menor tempo

[1]O índice α indica o próximo intervalo para a chamada de reescalonamento (ver Seção 3.1).

de processamento, retornando assim o tempo obtido e a referência ao processador em questão. Para alcançar esse resultado, o método $getANode()$ considera o número de bytes recebidos pelo objeto i de outros objetos do Conjunto j_2 na última superetapa, bem como o custo de migração.

A função de decisão chamada Potencial de Migração (PM) responde por quais objetos são candidatos à migração. Conforme discutido na Seção 3.1, o valor de PM é alcançado através da combinação das métrica Computação, Comunicação e Memória (ver Equação 3.1). A métrica Computação - $Comp(i, j)$ - considera o Padrão de Computação do processo i, a predição do tempo de computação de i e o nível de desempenho do Conjunto j. Os dados utilizados para calcular essa métrica iniciam na superetapa k e terminam na superetapa $k+\alpha-1$. De acordo com o modelo MigBSP, o cálculo do Padrão de Computação é obtido através da quantidade de instruções do processo i (RIGHI et al., 2010). Esse padrão mede a regularidade do processo conforme o número de instruções executadas a cada superetapa. Para simplificar nossa implementação, estamos adotando o tempo de conclusão da fase de computação ao invés da quantidade de instruções. Esse tempo é obtido da seguinte forma: o tempo de sistema $t1$ é capturado no início da superetapa, e o tempo de sistema $t2$ é registrado no início da fase de comunicação. Sendo assim, o tempo de computação é o resultado de $t2 - t1$. Esse valor também é utilizado para obter a predição do tempo de computação do objeto. Por fim, o nível de desempenho do Conjunto j é calculado considerando o desempenho dos processadores sobre sua jurisdição. Esse índice é recuperado na $k + \alpha - 1$-*ésima* superetapa com a chamada ao método $getCapacity()$ da classe $SetManager$ (ver Figura 18). Esse método é invocado n vezes, sendo n a quantidade de Conjuntos.

Assim como a métrica anterior, a métrica Comunicação - $Comm(i, j)$ - calcula o Padrão de Comunicação entre os processos BSP e os Conjuntos. Esse padrão considera os dados recebidos pelo processo i provenientes dos processos do Conjunto j a cada superetapa. Esses dados são recuperados na superetapa k até $k+\alpha-1$ pelos métodos $bsp\_get()$ e $bsp\_put()$. Esses métodos armazenam o tempo de comunicação e a quantidade de bytes transferidos. Por exemplo, o método $bsp\_get()$ armazena em uma estrutura de dados local o tempo de comunicação e o tamanho da mensagem recebida (ver linha 9 da Figura 16). Por outro lado, o método $bsp\_put()$ envia para o objeto de destino o tempo da comunicação e o tamanho da mensagem transmitida (ver linha 18). Dessa forma, o cálculo do Padrão de Comunicação é obtido com a quantidade de bytes capturados com esses métodos. Ainda, a métrica Comunicação considera a predição do tempo de comunicação entre duas chamadas de reescalonamento. Ele é calculado considerando o tempo de comunicação envolvendo o objeto i e o Conjunto j na superetapa $k + \alpha - 1$.

A métrica Memória - $Mem(i, j)$ - é composta pela memória do processo i, a taxa de transferência entre i e o Conjunto j e os custos relacionados a migração. Nossa implementação simplifica o cálculo dessa métrica ao considerar o tamanho do objeto em memória e os custos de migração obtidos com ProActive (ver Seção 5.2). O tamanho do objeto é recuperado ao subtrairmos o espaço de memória ocupado por uma JVM antes e depois do lançamento da aplicação. Sendo assim, $Mem(i, j)$ é o custo de migração equivalente ao tamanho do objeto a

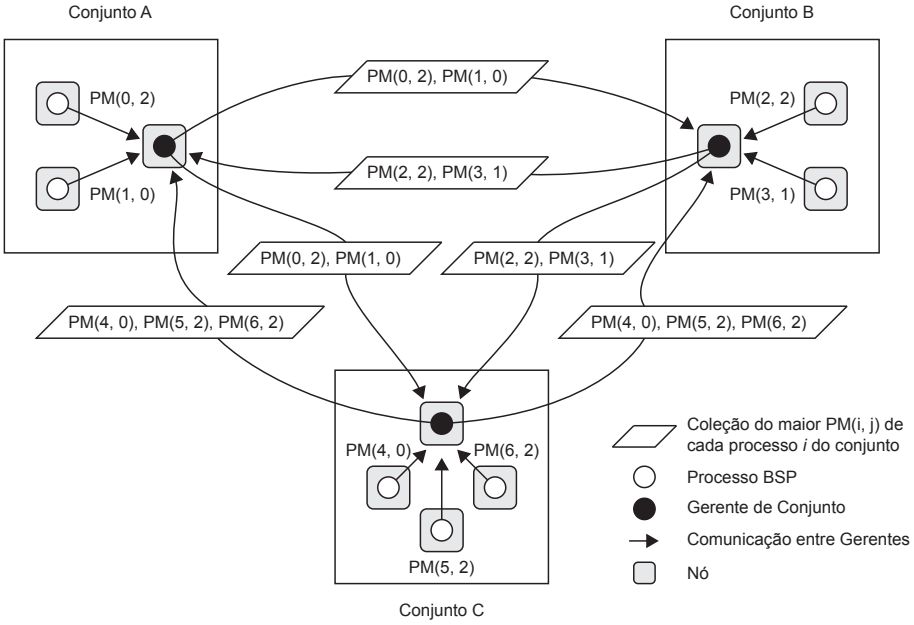

Figura 19: Exemplo de passagem de mensagens entre os Gerentes de Conjunto com a coleção dos maiores $PM(i, j)$ de cada objeto BSP

ser transferido.

Após obter o valor das métricas acima, cada objeto i calcula n vezes a Equação 3.1 localmente, onde n é a quantidade de Conjuntos no ambiente. Depois disso, o objeto i envia o seu maior PM para o seu Gerente. Logo após, os Gerentes de Conjunto trocam os seus valores de PM. Essas interações são realizadas pelo método $exchageDataAmongSetManagers()$ da Figura 16. A Figura 19 ilustra um exemplo dessa comunicação. Cada Gerente de Conjunto cria uma lista ordenada de forma decrescente com o maior PM de cada objeto BSP. Os valores negativos de PM são descartados da lista. O modelo MigBSP usa essa lista para aplicar uma das duas heurísticas com o objetivo de selecionar os candidatos à migração. A primeira heurística escolhe um objeto BSP, o primeiro da lista, que possui o maior PM. A segunda heurística escolhe os objetos que possuem um PM maior que $MAX(PM).x$, onde $MAX(PM)$ é o maior PM e x um percentual.

4.3.4 Lançamento da Aplicação

O desempenho de aplicações paralelas depende de muitos fatores, tais como o padrão de comunicação, a distância entre os processos da aplicação, a contenção da rede etc. (PASCUAL; NAVARIDAS; MIGUEL-ALONSO, 2009). O primeiro é uma característica que depende da aplicação, entretanto, os demais são afetados pela maneira como a aplicação é lançada. Nós implementamos a estratégia de alocação contínua para o mapeamento de objetos para recursos. Essa estratégia reduz a distância entre os processos da aplicação, acelerando a operação de troca de mensagens e reduzindo a utilização da rede. Sendo assim, a ideia é preencher um agregado e, em seguida, passar para o próximo. Inicialmente, jMigBSP mapeia um objeto por máquina.

```
1.  <GCMApplication>
2.    <environment></environment>
3.    <application></application>
4.    <resources>
5.      <nodeProvider id="myNodeProvider1">
6.        <file path="Set1.xml"/>
7.      </nodeProvider>
8.      <nodeProvider id="myNodeProvider2">
9.        <file path="Set2.xml"/>
10.     </nodeProvider>
11.   </resources>
12. </GCMApplication>
```

Figura 20: Exemplo de arquivo XML descrendo os VNs disponíveis no ambiente computacional

```
1.  <GCMDeployment>
2.    <environment></environment>
3.    <resources></resources>
4.    <infrastructure>
5.      <hosts>
6.        <host id="myHost" os="unix" hostCapacity="2" vmCapacity="1"/>
7.      </hosts>
8.      <groups>
9.        <sshGroup id="myGroup" hostList="labgeral01 labgeral02 labgeral03"/>
10.     </groups>
11.   </infrastructure>
12. </GCMDeployment>
```

Figura 21: Exemplo de arquivo XML descrevendo os nós pertencentes a um VN e suas características

Se a quantidade de objetos é maior do que o número de máquinas, o mapeamento começa novamente a partir do primeiro agregado.

O lançamento da aplicação ocorre na chamada ao método $start()$. Esse método implementa o modelo de alocação citado acima. Para tanto, ele cria os objetos ativos na medida em que encontra um recurso computacional nos arquivos descritores. Nesses arquivos, os recursos são agrupados em Nós Virtuais, ou simplesmente VNs (*Virtual Nodes*) (BAUDE et al., 2009). Os VNs são uma abstração da infraestrutura física em que a aplicação será lançada. A Figura 20 exemplifica um arquivo XML que descreve dois VNs disponíveis no ambiente computacional. Os nós pertencentes a cada VN e suas características são especificados em arquivos separados. Considerando a Figura 20, esses arquivos são definidos nas linhas 6 e 9. Um VN pode conter um ou mais nós. O nó representa a localização onde um objeto pode ser criado e executado. Ele pode ser simplesmente um computador físico ou, nos casos de arquiteturas com múltiplos processadores e diversos núcleos, um único processador ou um único núcleo de uma máquina. A Figura 21 demonstra um arquivo XML que define as características e os nós pertencentes a um VN. O sistema operacional, a quantidade de JVMs por nó e a quantidade de objetos por JVM são definidos na linha 6. Nesse caso, estamos trabalhando com duas JVMs por nó e somente um objeto por JVM. A linha 9 descreve os nós que pertencem a esse VN. Nesse caso, três nós estão sendo especificados: labgeral01, labgeral02 e labgeral03.

O sistema jMigBSP considera um VN como sendo um agregado de computadores. Essa organização nos permite criar com facilidade um ambiente *multicluster*. Nesse sentido, quando um VN é encontrado durante o lançamento da aplicação, um Gerente de Conjunto é criado no primeiro nó desse agregado. Quando o lançamento da aplicação é concluído, todos os Gerentes recebem uma estrutura de dados com o procurador para os demais Gerentes, permitindo a

comunicação entre eles.

4.4 Balanço

Esse capítulo apresentou em detalhes o desenvolvimento do sistema jMigBSP. Ele se propõe a oferecer uma interface simples para a escrita de aplicações BSP em Java. Nesse sentido, para escrever uma aplicação utilizando jMigBSP, o programador precisa apenas estender a classe $jMigBSP$ e implementar o método $run()$. Em aspectos de implementação, jMigBSP herda quatro características de ProActive: (i) a facilidade no mapeamento de objetos para recursos; (ii) o modelo de programação SPMD; (iii) a migração de objetos e; (iv) o modelo de comunicação assíncrona. Em adição, jMigBSP propõe novos mecanismos para seguir o modelo de programação BSP. No BSP, as mensagens enviadas para um processo durante uma superetapa ficam disponíveis para uso somente na superetapa seguinte. Para atender a essa regra, cada objeto jMigBSP possui dois vetores que atuam como *buffers*. O primeiro, denominado de Buffer Ativo, possui as mensagens que foram recebidas na superetapa anterior e estão disponíveis para uso através do método $bsp\_get()$. O segundo, denominado de Buffer Temporário, atua como receptor das mensagens que são enviadas com o método $bsp\_put()$. Após o término de uma superetapa, o conteúdo do Buffer Ativo é substituído pelo conteúdo do Buffer Temporário.

A transmissão de dados em jMigBSP ocorre de forma assíncrona. Para tanto, os métodos $bsp\_put()$ e $bsp\_get()$ implementam as ideias de objetos futuros e de espera pela necessidade de ProActive. Assim, o transmissor e o receptor podem utilizar os ciclos ganhos para realizar alguma computação útil enquanto a RMI é processada. Além disso, o sistema jMigBSP implementa mecanismos para oferecer reescalonamento de objetos. O reescalonamento pode ser gerenciado através de chamadas ao método $bsp\_migrate()$ dentro do código da aplicação, ou através do balanceamento de carga automático em nível de *middleware*. A Tabela 6 resume as vantagens e desvantagens no uso de ambas as abordagens. Em especial, o reescalonamento em nível de *middleware* é oferecido através da classe $LBjMigBSP$. Essa classe implementa as ideias do modelo MigBSP e reimplementa os métodos de comunicação e sincronização de jMigBSP para a captura de dados sobre o escalonamento. No modelo MigBSP, a escolha dos processos candidatos à migração é baseada no Potencial de Migração (PM). O PM é alcançado através da combinação das métricas Computação, Comunicação e Memória. Para calcular essas métricas, jMigBSP utiliza os seguintes dados:

- Métrica Computação: (i) tempo de conclusão da fase de computação; (ii) capacidade média de processamento dos Conjuntos.
- Métrica Comunicação: (i) quantidade de bytes recebidos pelo objeto i provenientes dos objetos do Conjunto j; (ii) tempo de comunicação envolvendo o objeto i e o Conjunto j;
- Métrica Memória: (i) tamanho do objeto em memória; (ii) custo de migração do objeto com ProActive.

Tabela 6: Vantagens e desvantagens entre as abordagens de reescalonamento de jMigBSP

Nível de migração	Vantagens	Desvantagens
Aplicação	(i) Flexibilidade quanto ao momento da migração; (ii) possibilidade de o programador implementar suas próprias ideias de reescalonamento.	(i) Acoplamento entre aplicação e a implementação do reescalonador; (ii) a implementação do algoritmo de reescalonamento não pode ser utilizada com outras aplicações e/ou infraestrutura; (iii) o programador deve conhecer algoritmos de reescalonamento, bem como o ambiente computacional.
Middleware	(i) Desacoplamento entre a aplicação e o algoritmo de reescalonamento; (ii) não requer a alteração da aplicação quanto executada em uma nova infraestrutura.	(i) Impõe uma sobrecarga no tempo de execução da aplicação; (ii) migrações podem ocorrer perto do final da execução da aplicação e gerar mais sobrecarga.

Com o PM calculado, é possível definir quais objetos são candidatos à migração. Essa decisão é tomada após o final de uma superetapa, pois nesse momento é possível analisar dados de todos os objetos BSP sobre suas fases de computação e comunicação. O sistema jMigBSP implementa as duas heurísticas de MigBSP para selecionar os objetos candidatos a migração. A primeira heurística escolhe o objeto BSP com o maior valor de PM em relação aos demais. A segunda heurística escolhe os objetos que possuem um PM maior que $MAX(PM).x$, onde $MAX(PM)$ é o maior PM e x um percentual. Além disso, o presente capítulo abordou a estratégia de jMigBSP para o lançamento da aplicação. Nós implementamos a estratégia de alocação contínua para o mapeamento do objetos para recursos. A ideia é preencher um agregado e, em seguida, passar para o próximo. O lançamento da aplicação ocorre na invocação do método $start()$. Na medida em que jMigBSP lê os arquivos descritores do ambiente, ele cria os objetos ativos nos processadores disponíveis.

5 AVALIAÇÃO DE JMIGBSP

Esse capítulo apresenta as estratégias que foram utilizadas para avaliar jMigBSP. Para tal, ele apresenta os resultados de jMigBSP na execução de aplicações BSP. Vamos apresentar situações onde o emprego do balanceamento de carga automático de jMigBSP alcança um melhor desempenho em relação a execução da aplicação paralela sem migrações. Além disso, vamos explicar as situações em que jMigBSP inclui uma sobrecarga no tempo total da aplicação, não colaborando para reduzir o tempo de conclusão dela.

O presente capítulo é segmentado em quatro seções. A primeira delas apresenta os agregados que serviram de base para a execução dos experimentos com jMigBSP. A Seção 5.2 apresenta uma análise do custo de migração de objetos com ProActive. A Seção 5.3 mostra os resultados ao implementar duas aplicações com jMigBSP: (i) transformada rápida de Fourier e; (ii) compressão de imagens. Essa seção é o núcleo desse capítulo, uma vez que ela aborda o comportamento de jMigBSP e sua viabilidade, ou não, para a obtenção de um melhor desempenho em aplicações BSP. Finalmente, a Seção 5.4 mostra os principais tópicos escritos nesse capítulo.

5.1 Ambiente de Experimentos e de Execução

Com o advento das grades computacionais, a interligação de agregados de computadores é cada vez mais comum para a criação de arquiteturas *multiclusters* para o processamento de aplicações científicas e comerciais (ZHANG; KOELBEL; COOPER, 2009). Nesse sentido, nós construímos uma infraestrutura com dois agregados conforme ilustrado na Figura 22. Cada agregado é mapeado para um Conjunto que possui um Gerente de Conjunto em execução no primeiro nó. Tal Gerente é pertinente para o funcionamento do modelo MigBSP, como é possível ver na Seção 3.1. Nossa infraestrutura é heterogênea em termos de capacidade de processamento dos nós, bem como em termos de largura de banda. O agregado A possui nós com processadores Intel Core 2 Duo de 2.93 GHz e 4 GB de memória principal, interligados por uma rede de 100 Mbits/s. O agregado B possui nós biprocessados Intel Xeon de 2.4 GHz e 2 GB de memória principal, ligados por uma rede de 10 Mbits/s. Conforme ilustrado na Figura 22, a comunicação entre ambos os agregados ocorre através de uma conexão de 10 Mbits/s. Todas as máquinas de nosso ambiente utilizam o sistema operacional GNU/Linux, com o núcleo na sua versão 2.6.26. Continuando a descrição de *software*, é utilizada a máquina virtual Java na sua versão 1.6.0 e a biblioteca ProActive na sua revisão 5.0.3. Em adição, os experimentos envolvendo a transformada rápida de Fourier utilizam a biblioteca BSPlib em sua versão 1.4.

Qualquer nó de nossa arquitetura paralela oferece a capacidade de executar até dois processos BSP. A Figura 23 ilustra o mapeamento inicial de processos para recursos em nosso ambiente. Uma vez que as aplicações revelam um caráter de comunicação entre objetos cujos índices são próximos, foi adotada a abordagem contínua na qual cada agregado é totalmente pre-

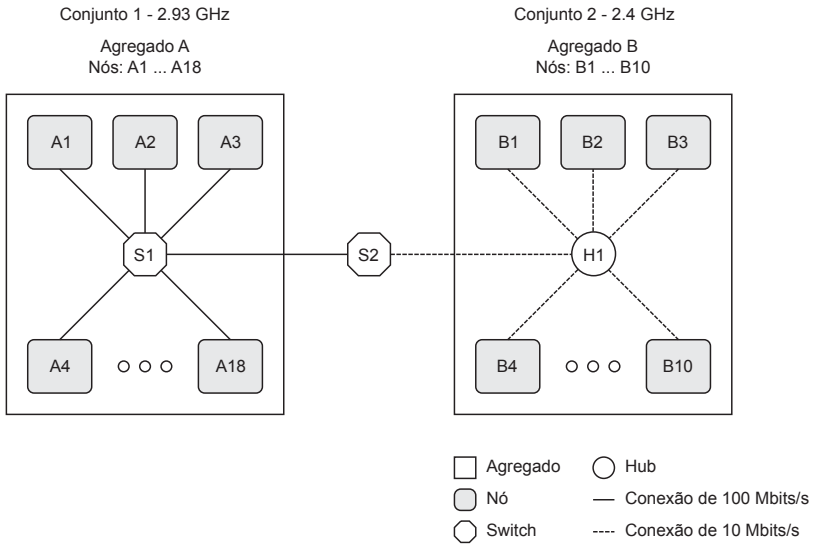

Figura 22: Infraestrutura composta por dois agregados utilizada na avaliação de jMigBSP

Mapeamento inicial de processos para recursos	
2 processos	= A {1-2}
4 processos	= A {1-4}
8 processos	= A {1-8}
16 processos	= A {1-16}
32 processos	= A {1-18}, B {1-10}, A {1-4}

Figura 23: Mapeamento inicial de processos para recursos utilizado na avaliação de jMigBSP

enchido antes de completar o próximo (PASCUAL; NAVARIDAS; MIGUEL-ALONSO, 2009). Sendo assim, a ideia é preencher um agregado e, em seguida, passar para o próximo. Inicialmente, jMigBSP mapeia um objeto por máquina. Se a quantidade de objetos é maior do que a quantidade de máquinas, o mapeamento começa novamente a partir do primeiro Conjunto. Isso acontece quando 32 objetos são mapeados em nossa infraestrutura. Além disso, o agregado B somente é utilizado quando trabalhamos com 32 objetos (ver Figura 23). As próximas seções irão discutir os resultados de jMigBSP ao executar diferentes aplicações BSP.

5.2 Análise do Custo de Migração

O objetivo desse experimento é de comparar os tempos de migração de objetos com ProActive e os tempos de transferência desses mesmos objetos com Java Sockets. Essa observação é pertinente para analisar o custo de ProActive na transferência de dados pela rede. Ela vai mostrar a usabilidade, ou não, dessa ferramenta. Dois nós do agregado A (ver Figura 23) foram reservados para essa avaliação. O primeiro executa uma aplicação sintética que cria um objeto ativo com uma quantidade específica de dados. A aplicação chama a diretiva de migração sobre o objeto criado, movendo-o para o segundo nó. O tempo de sistema é computado antes e após a chamada da diretiva de migração. Além dessa aplicação, outra foi escrita para criar um objeto Java e transferi-lo usando Sockets. Essa aplicação também envolve somente duas máquinas do

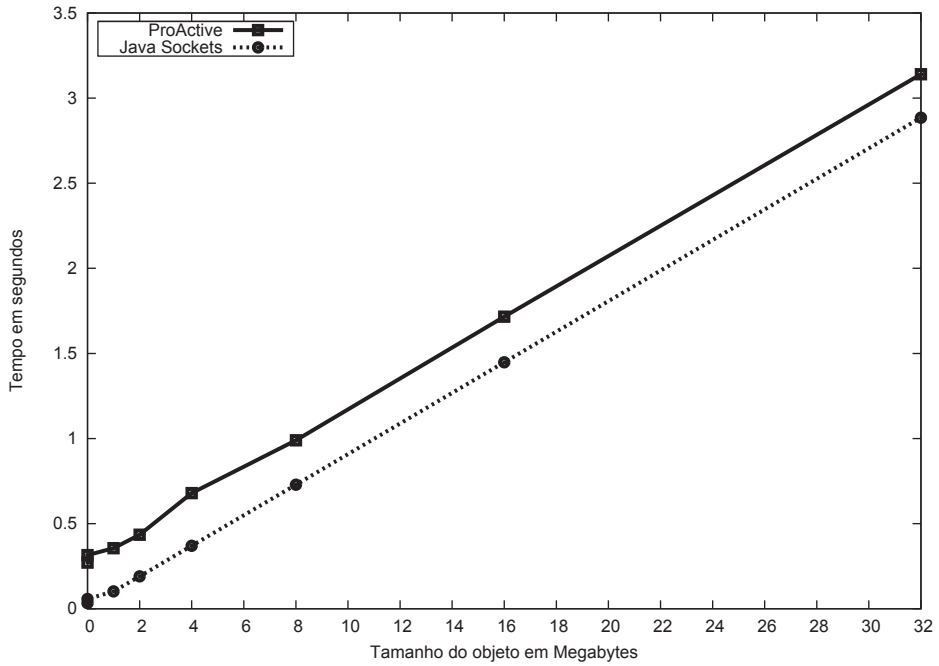

Figura 24: Analisando o tempo de migração e transferência de objetos Java em uma rede de 100 Mbits/s

agregado A.

Os dados alocados para os objetos variaram de 1 KB a 32 MB. Os tempos finais obtidos são o resultado da média aritmética de 10 execuções das aplicações para cada tamanho de objeto. A Figura 24 mostra o comportamento dos resultados ao executar as aplicações sintéticas no agregado A. Basicamente, o gráfico apresenta um comportamento linear, onde um aumento na quantidade de dados alocados acarreta em um aumento proporcional no tempo de migração e transferência dos objetos. Por exemplo, ao migrar um objeto Java de 8 MB, alcança-se o tempo de 0,98 segundos com ProActive e 0,72 segundos ao transferir o mesmo objeto usando Java Sockets (ver Tabela 7). Ao alocar 16 MB, 1,71 segundos são observados com ProActive e 1,44 segundos com Java Sockets. Assim, 0,26 segundos são medidos em ambos os casos quando subtraído os tempos de ProActive e Java Sockets. Em outras palavras, a sobrecarga imposta pelo recurso de migração de ProActive não depende do tamanho dos dados manipulados. Essa mesma avaliação foi realizada em dois nós do agregado B com o objetivo de conhecermos o custo de migração em uma rede de 10 Mbits/s. A Figura 25 mostra os resultados obtidos e a Tabela 8 detalha os tempos alcançados.

Os resultados alcançados mostram que o custo de migração de ProActive diminui com o aumento do tamanho do objeto. Por exemplo, a migração de um objeto de 2 MB com ProActive em uma rede de 10 Mbits/s impôs um sobrecusto de 37,54% em relação a Java Sockets. Por outro lado, esse sobrecusto diminui para 18,60% ao migrar um objeto de 4 MB. Tal sobrecusto é justificado pelas operações que a biblioteca ProActive realiza para garantir a migração e o acesso transparente ao objeto transferido. Dentre essas operações, pode-se citar a transferência do estado do objeto, sua serialização, a criação de um objeto procurador e a criação de uma nova instância do objeto na máquina de destino (XU; LIAN; GAO, 2003). Ao utilizar Java Sockets,

Tabela 7: Tempo de migração e transferência de objetos Java em uma rede de 100 Mbits/s (tempo em segundos)

Tamanho do objeto	ProActive		Java Sockets		Sobrecarga	Percentual
	Tempo	Desvio padrão	Tempo	Desvio padrão		
1 KB	0,2714	0,0219	0,0329	0,0007	0,2385	725,24%
512 KB	0,3152	0,0196	0,0570	0,0001	0,2582	453,41%
1024 KB	0,3557	0,0240	0,1015	0,0001	0,2541	250,31%
2048 KB	0,4351	0,0233	0,1907	0,0001	0,2444	128,18%
4096 KB	0,6796	0,0128	0,3698	0,0001	0,3097	83,74%
8192 KB	0,9896	0,0189	0,7290	0,0001	0,2606	35,75%
16384 KB	1,7160	0,0118	1,4478	0,0001	0,2682	18,53%
32768 KB	3,1406	0,0235	2,8849	0,0001	0,2557	8,86%

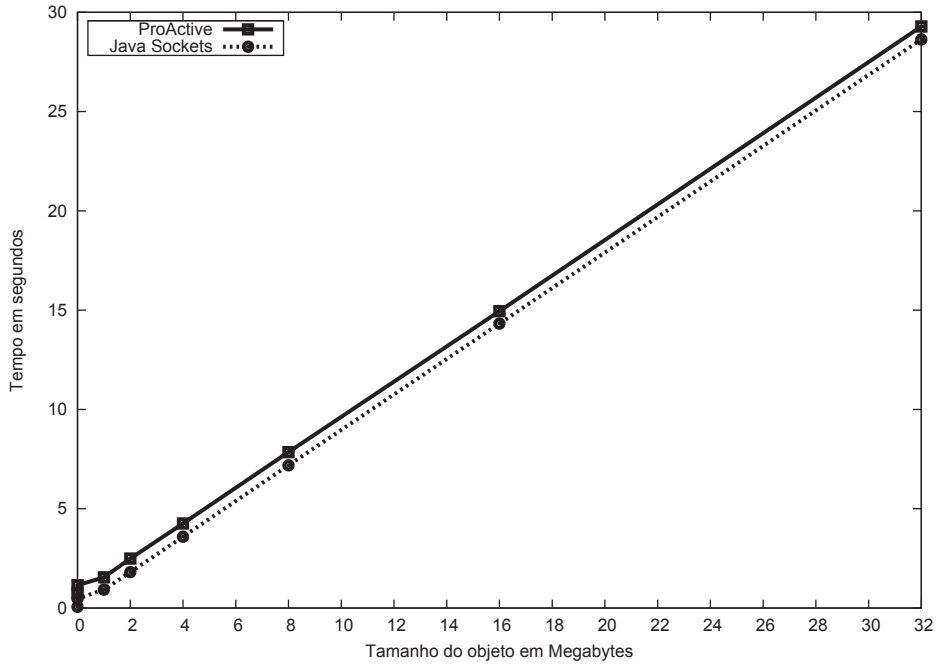

Figura 25: Analisando o tempo de migração e transferência de objetos Java em uma rede de 10 Mbits/s

Tabela 8: Tempo de migração e transferência de objetos Java em uma rede de 10 Mbits/s (tempo em segundos)

Tamanho do objeto	ProActive		Java Sockets		Sobrecarga	Percentual
	Tempo	Desvio padrão	Tempo	Desvio padrão		
1 KB	0,7617	0,0262	0,0620	0,0006	0,6997	1128,75%
512 KB	1,1460	0,0415	0,4725	0,0007	0,6735	142,55%
1024 KB	1,5479	0,0417	0,9272	0,0194	0,6207	66,94%
2048 KB	2,4891	0,0354	1,8097	0,0009	0,6794	37,54%
4096 KB	4,2598	0,0463	3,5919	0,0009	0,6680	18,60%
8192 KB	7,8486	0,0286	7,1856	0,0227	0,6630	9,23%
16384 KB	14,9495	0,0461	14,3261	0,0257	0,6234	4,35%
32768 KB	29,2907	0,0321	28,6340	0,0093	0,6567	2,29%

o objeto é simplesmente serializado e transferido para o nó de destino.

5.3 Análise do Desempenho de jMigBSP em Aplicações Paralelas

O objetivo dessa seção é demonstrar a viabilidade de jMigBSP para a escrita de aplicações BSP. Nesse sentido, nossa ideia é verificar se o balanceamento de carga automático de jMigBSP permite uma redução no tempo da aplicação se comparado com a execução paralela sem a reorganização de objetos. Além disso, vamos explicar situações em que jMigBSP não colabora para o melhor desempenho da aplicação. Sendo assim, as subseções que seguem descrevem os resultados obtidos ao implementarmos a transformada rápida de Fourier e o algoritmo que realiza a compressão de imagens segundo o método de Fractal. Os experimentos com a transformada de Fourier buscaram analisar o desempenho de jMigBSP em relação a uma biblioteca de código nativo. Por outro lado, o algoritmo de compressão de imagens teve como objetivo avaliar o recurso de balanceamento de carga automático de jMigBSP. Além dessas, uma terceira aplicação foi desenvolvida e utilizada para validar as primeiras versões de jMigBSP. O algoritmo que calcula a soma de prefixos foi a aplicação escolhida para essa tarefa. O mesmo foi executado em nosso ambiente para validarmos questões como a troca de mensagens e a sincronização entre os objetos de jMigBSP.

5.3.1 Transformada Rápida de Fourier

A transformada rápida de Fourier, ou simplesmente FFT (*Fast Fourier Transform*), é um algoritmo capaz de computar a transformada discreta de Fourier com uma complexidade menor em relação ao método direto (COOLEY; TUKEY, 1965). A FFT usa um número reduzido de operações aritméticas, alcançando assim um menor esforço computacional. Nossa implementação é baseada no algoritmo de Bisseling (2004). O algoritmo realiza uma sequência de operações sobre um vetor complexo de comprimento n, onde n é uma potência de dois. Em cada estágio da computação, elementos do vetor são modificados em pares, onde cada par de entrada produz um novo par de saída. No estágio k, $0 \leq k \leq \log_2 n$, os elementos de um par estão a uma distância de 2^k. Assumindo que temos p processadores, onde p é uma potência de dois, a ideia básica é permutar o vetor de tal forma que durante o estágio $\log_2 n - \log_2 p$ ambos os elementos de um par estejam no mesmo processador. Nos casos em que $p \leq \sqrt{n}$, o algoritmo é resolvido em duas superetapas, realizando somente uma permutação. O Algoritmo 1 apresenta a modelagem da função de permutação da FFT. O algoritmo paralelo pode ser caracterizado como regular, onde cada superetapa apresenta o mesmo número de instruções a ser calculado por processo, bem como o mesmo comportamento de comunicação (BISSELING, 2004).

A FFT foi implementada com o objetivo de analisarmos o desempenho de jMigBSP em relação a uma biblioteca de código nativo. Naturalmente, espera-se que a própria linguagem

Algoritmo 1 Modelagem da função de permutação do algoritmo da FFT

1: Considerando um vetor x de tamanho n, $np = \frac{n}{bsp\_nprocs()}$ e p igual a quantidade de processos paralelos.
2: **for** $j = 0$ até $j < np$ **do**
3: $\sigma = j * p + bsp\_pid()$
4: $dst\_pid = \frac{\sigma}{np}$
5: O processo $bsp\_pid()$ envia o elemento $x[j]$ para o processo dst_pid.
6: **end for**
7: Chamada para a barreira de sincronização.

Tabela 9: Tempo de execução da FFT ao ser paralelizada com 8 processos com jMigBSP e BSPlib (tempo em segundos)

Tamanho do vetor	jMigBSP		BSPlib		Sobrecarga	Percentual
	Tempo	Desvio padrão	Tempo	Desvio padrão		
2^{22}	2,8913	0,0012	1,8507	0,0029	1,0406	56,23%
2^{23}	5,8341	0,0031	3,9213	0,0103	1,9128	48,78%
2^{24}	13,7827	0,0078	10,5497	0,0265	3,2330	30,65%
2^{25}	27,3284	0,0121	22,0151	0,0065	5,3133	24,13%

interpretada Java e o *middleware* ProActive imponham um tempo maior de execução se comparados com uma abordagem compilada e executada sem máquina virtual. Por outro lado, é pertinente analisarmos essa sobrecarga, uma vez que nosso trabalho está dirigido para a área da computação paralela. Sendo assim, desenvolvemos duas versões da FFT: (i) uma escrita em jMigBSP e; (ii) outra em BSPlib. Ao compararmos jMigBSP com uma biblioteca de código nativo queremos conhecer a sobrecarga que Java e ProActive impõem sobre jMigBSP. Em especial, queremos demonstrar que essa sobrecarga diminui com o aumento do grão de computação. As Tabelas 9 e 10 apresentam o tempo de execução de ambas as implementações ao variarmos o tamanho do vetor de entrada e a quantidade de processos. Como esperado, BSPlib obteve um melhor desempenho em relação a jMigBSP. No entanto, é possível observar que quanto maior o grão de computação, menor é a diferença entre ambas as bibliotecas. Por exemplo, ao executar a FFT com 16 processos e um vetor de tamanho 2^{24}, 7,73 segundos e 5,75 segundos são obtidos com jMigBSP e BSPlib, respectivamente. Esses tempos representam uma sobrecarga de 25,63%. No entanto, para um vetor de tamanho 2^{25}, 16,21 segundos e 14,02 segundos são alcançados com jMigBSP e BSPlib. Esses resultados representam uma redução de 12,09% na sobrecarga de jMigBSP.

Seguindo a comparação entre as bibliotecas, nossa avaliação também observou os possíveis ganhos de jMigBSP sobre BSPlib ao habilitarmos o recurso de migração explícita. Para tanto, ambas as implementações da FFT foram executadas em oito nós do agregado A e uma sobrecarga foi simulada em quatro deles. Para simular a sobrecarga, limitamos o percentual máximo de uso dos quatro processadores em 50% usando a ferramenta cpulimit[1] (CESARIO et al.,

[1] http://cpulimit.sourceforge.net/

Tabela 10: Tempo de execução da FFT ao ser paralelizada com 16 processos com jMigBSP e BSPlib (tempo em segundos)

Tamanho do vetor	jMigBSP		BSPlib		Sobrecarga	Percentual
	Tempo	Desvio padrão	Tempo	Desvio padrão		
2^{22}	1,6931	0,0011	0,9871	0,0009	0,7060	41,69%
2^{23}	3,9172	0,0024	2,4209	0,0012	1,4963	38,19%
2^{24}	7,7356	0,0089	5,7531	0,0008	1,9825	25,63%
2^{25}	16,2197	0,0065	14,0234	0,0113	2,1963	13,54%

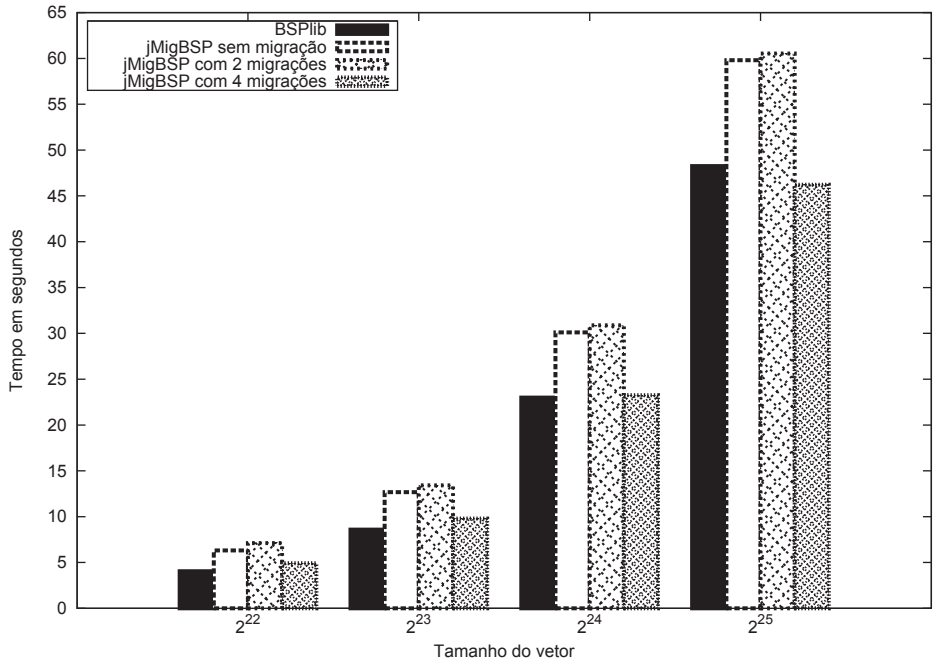

Figura 26: Observando o desempenho da FFT quando habilitada a migração explícita de jMigBSP

2011; VERA; SUPPI, 2010). Chamadas explícitas ao método $bsp\_migrate()$ foram inseridas na barreira de sincronização da aplicação escrita em jMigBSP. A ideia é transferir os objetos localizados em processadores sobrecarregados para outros levemente carregados, buscando minimizar o tempo de execução da aplicação. A Figura 26 apresenta os resultados obtidos com esse cenário. Ela mostra que jMigBSP supera em 4,19% o desempenho de BSPlib ao trabalhar com um vetor de tamanho 2^{25} e transferir quatro objetos para processadores levemente carregados (ver Tabela 11). Os testes envolvendo a migração de dois objetos não apresentam ganhos de desempenho devido a regra do modelo BSP. Isso porque o modelo possui uma barreira de sincronização que sempre espera pelo processo mais lento (nesse caso, os dois processos que permanecem nos processadores sobrecarregados e limitam o tempo de execução da aplicação).

5.3.2 Compressão de Imagens

A compressão de imagens segundo a técnica de Fractal tem gerado interesse na comunidade como um possível competidor de formatos já estabelecidos como JPEG e Wavelets (GUO

Tabela 11: Tempo de execução da FFT quando habilitada a migração explícita de jMigBSP (tempo em segundos)

Tamanho do vetor	jMigBSP		BSPlib		jMigBSP com 2 migrações		jMigBSP com 4 migrações	
	Tempo	Desvio padrão	Tempo	Desvio padrão	Tempo	Desvio padrão	Tempo	Desvio padrão
2^{22}	6,3219	0,0451	4,0437	0,2471	7,1145	0,3918	4,9214	0,1819
2^{23}	12,6812	0,0704	8,5910	0,4218	13,4121	0,9687	9,7943	0,3982
2^{24}	30,1097	1,2546	22,9856	1,4685	30,8742	1,4742	23,2524	1,2812
2^{25}	59,8124	1,5691	48,2449	2,2981	60,5310	2,1097	46,2210	2,3753

et al., 2009). Um dos principais problemas na abordagem de Fractal é a alta complexidade de compressão. Entretanto, a decodificação acontece em um tempo muito menor (XING, 2008). O algoritmo consiste na transformação de blocos, os quais aproximam partes menores da imagem por partes maiores. As partes menores são chamadas de intervalos, enquanto que as maiores de domínios. A união de todos os intervalos forma uma imagem. Os domínios são selecionados livremente dentro da imagem. Para cada intervalo, um domínio apropriado deve ser encontrado. A distância *rms* é calculada com o objetivo de julgar a qualidade de um mapeamento. Basicamente, o tempo de codificação depende do número de domínios que devem ser comparados aos intervalos.

A modelagem do algoritmo de compressão combina o modelo de programação BSP com o paradigma de *pipeline* circular (GRAEBIN; SILVA; RIGHI, 2011). O Algoritmo 2 apresenta a organização de uma simples superetapa. Primeiramente, conjuntos de domínios são distribuídos uniformemente entre os processos. Cada processo é designado a calcular um conjunto de intervalos. As melhores combinações de domínio para os intervalos calculados são armazenadas no processo. Em seguida, esse processo envia ao vizinho da direita essas combinações, juntamente com o conjunto de intervalos calculado anteriormente e que serão a carga de trabalho do receptor na próxima superetapa. Quando todos os conjuntos de intervalos tiverem passado por todos os processos do *pipeline*, a última superetapa será computada. Nela, um único processo é responsável por solicitar aos demais as melhores combinações obtidas de domínios e intervalos. De posse dessas combinações, o processo escolhido produzirá a imagem resultante.

A Tabela 12 apresenta os tempos obtidos com o algoritmo sequencial para uma imagem de 512×512 *pixels* quando variado a quantidade de domínios. Os tempos finais obtidos são o resultado da média aritmética de 10 execuções da aplicação para cada variação na quantidade de domínios. A Figura 27 (a) ilustra a imagem utilizada na avaliação. As Figuras 27 (b), (c) e (d) apresentam as imagens resultantes após a execução do algoritmo utilizando 16384, 4096 e 1024 domínios, respectivamente. Em especial, observa-se que a Figura 27 (b) apresenta poucas perdas de qualidade em relação a imagem original, obtendo uma taxa de compressão de 66,03%. Nesse sentido, nota-se que a taxa de compressão da imagem aumenta na medida em que a quantidade de domínios é reduzida. Por outro lado, quanto maior a quantidade de do-

Algoritmo 2 Modelagem de uma superetapa para o problema da compressão de Fractal

```
1: Considerando um conjunto de intervalos r para a imagem.
2: for cada domínio pertencente a um processo específico do
3:    for cada intervalo em r do
4:       for cada isomeria de um domínio do
5:          Calcule rms(intervalo, domínio)
6:       end for
7:    end for
8: end for
9: Cada processo i (0 ≤ i ≤ n − 1) envia dados para o seu vizinho da direita i + 1. O processo
   n − 1 transmite para o processo 0 (onde n é o número de processos).
10: Chamada para a barreira de sincronização.
```

Tabela 12: Tempo de execução do algoritmo sequencial de compressão Fractal

Quantidade de domínios	Tempo de execução em segundos	Desvio padrão	Taxa de compressão
16384	1913,20	18,79	66,03%
4096	1863,72	14,78	91,42%
1024	1560,79	12,61	97,79%

mínios, maior a qualidade da imagem resultante. Os tempos obtidos ao paralelizar o algoritmo com jMigBSP no agregado A são apresentados nas Tabelas 13 e 14. Podemos observar que o algoritmo sequencial precisou de mais de 30 minutos para alcançar uma taxa de compressão de 66,03% sobre a Figura 27 (a). Esse cenário, ao ser paralelizado com 16 processos, resultou em um tempo de processamento de apenas 2,36% minutos (ver Tabela 14). Esse resultado representa uma redução no tempo da aplicação de 92,57% e uma eficiência de 84,19% do algoritmo paralelo.

Nós modelamos três cenários para avaliar o balanceamento de carga automático de jMigBSP com o algoritmo de compressão: (i) execução da aplicação simplesmente com jMigBSP; (ii) execução da aplicação com LBjMigBSP e sem migração; (iii) execução da aplicação com LBjMigBSP e migração habilitada. A Tabela 15 resume esses cenários. A ideia com essa organização é mostrar o comportamento normal de LBjMigBSP (Cenário 3) e a situação em que

(a) (b) (c) (d)

Figura 27: Observando a imagem resultante com o algoritmo de compressão Fractal: (a) imagem original; (b) imagem com 16384 domínios; (c) imagem com 4096 domínios; (d) imagem com 1024 domínios.

Tabela 13: Tempo de execução do algoritmo paralelo de compressão Fractal ao trabalhar com 2 e 4 processos (tempo em segundos)

Quantidade de domínios	2 processos			4 processos		
	Tempo	Desvio padrão	Eficiência	Tempo	Desvio padrão	Eficiência
16384	996,54	2,23	95,99%	522,53	7,97	91,53%
4096	983,82	2,25	94,71%	517,45	1,47	90,04%
1024	852,19	6,19	91,57%	438,61	0,92	88,96%

Tabela 14: Tempo de execução do algoritmo paralelo de compressão Fractal ao trabalhar com 8 e 16 processos (tempo em segundos)

Quantidade de domínios	8 processos			16 processos		
	Tempo	Desvio padrão	Eficiência	Tempo	Desvio padrão	Eficiência
16384	274,67	9,15	87,06%	142,02	7,61	84,19%
4096	269,15	2,96	86,56%	139,92	1,77	83,24%
1024	227,52	2,11	85,75%	118,47	5,87	82,34%

todas as migrações são inviáveis (Cenário 2). O Cenário 2 mede a sobrecarga relacionada aos cálculos de escalonamento, os custos com trocas de mensagens entre os Gerentes, bem como trocas de mensagens entre os processos BSP e seus respectivos Gerentes de Conjunto. Em outras palavras, o Cenário 2 consiste na realização de todos os cálculos de escalonamento e todas as decisões dos processos que realmente vão migrar, mas não inclui qualquer migração efetiva. O Cenário 3 permite migrações e adiciona os custos de migração entre processadores.

A comparação entre os Cenários 2 e 3 refere-se a sobrecarga imposta por LBjMigBSP na execução da aplicação quando migrações não ocorrem. O Cenário 2 sempre apresentará tempos maiores em relação ao Cenário 1. A diferença entre eles representa exatamente a sobrecarga imposta por LBjMigBSP. A análise dos Cenários 1 e 3 mostrará os ganhos ou perdas de desempenho quando a migração é habilitada. Se o tempo de execução da aplicação é reduzido quando se utiliza migração, pode-se afirmar que a realocação de um ou mais processos BSP obtém um melhor desempenho em relação aos custos de migração e aqueles relacionados aos cálculos de escalonamento.

A avaliação compreendeu a execução do algoritmo de compressão utilizando 16384 domínios em um ambiente formado por três agregados heterogêneos, conforme ilustrado na Figura 28. Os agregados A e B possuem nós com processadores Intel Core 2 Duo de 400 MHz e 1.5

Tabela 15: Diferentes cenários para a avaliação de LBjMigBSP

Cenário	Execução da aplicação	Execução de LBjMigBSP	Migração habilitada
Cenário 1	•		
Cenário 2	•	•	
Cenário 3	•	•	•

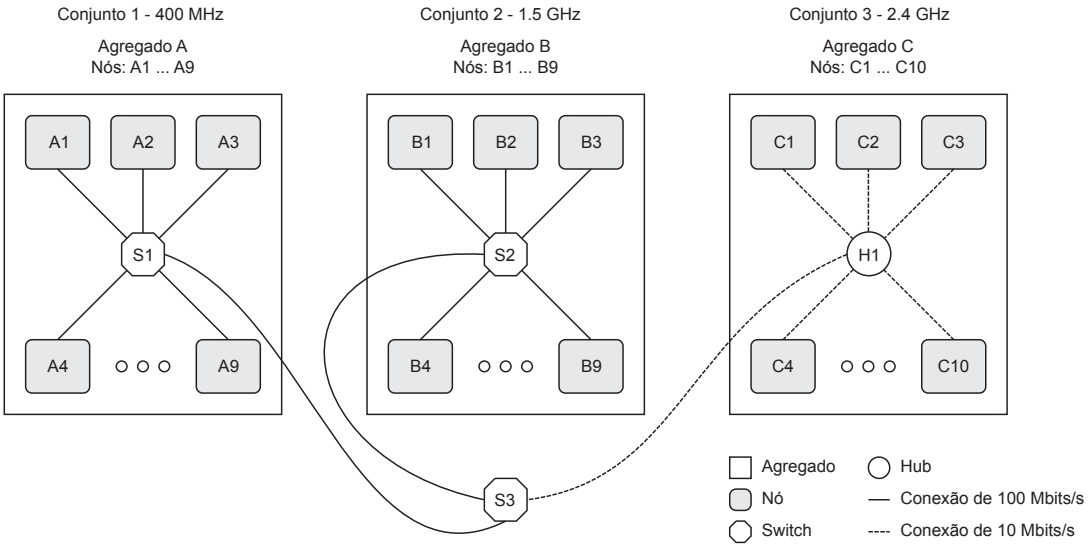

Figura 28: Infraestrutura composta por três agregados utilizada na avaliação de LBjMigBSP

Mapeamento inicial de processos para recursos	
2 processos	= A {1-2}
4 processos	= A {1-4}
8 processos	= A {1-8}
16 processos	= A {1-9}, B {1-7}
32 processos	= A {1-9}, B {1-9}, C {1-10}, A {1-4}

Figura 29: Mapeamento inicial de processos para recursos utilizado na avaliação de LBjMigBSP

GHz, respectivamente. Ambos possuem nós com 4 GB de memória principal, interligados por uma rede de 100 Mbits/s. O agregado C possui nós biprocessados Intel Xeon de 2.4 GHz e 2 GB de memória principal, ligados por uma rede de 10 Mbits/s. Conforme ilustrado na Figura 28, os agregados A e B se comunicam através de uma rede de 100 Mbits/s, enquanto que a ligação desses com o agregado C ocorre através de uma conexão de 10 Mbits/s. O mapeamento de processos para recursos segue a abordagem contínua e é expresso na Figura 29.

A Tabela 16 mostra os resultados obtidos com α igual a 4 e a heurística que avalia somente um processo como candidato a migração. Primeiramente, podemos observar que a intrusividade de LBjMigBSP é pequena quando comparado os Cenários 1 e 2 (5,52% são observados ao paralelizar a aplicação com 8 processos). Nesse cenário, todos os processos são mapeados para o agregado mais lento A (ver Figura 29). Uma migração é sugerida na quarta superetapa: {(p1, b3)}. Essa notação informa que o processo p1 será transferido para ser executado no nó b3. O agregado A obteve valores negativos de PM quando o reescalonamento foi testado. Apesar do agregado B não possuir os processadores com maior poder computacional do ambiente, a migração de p1 para o agregado C não é considerada viável devido ao custo de migração. Nesse sentido, quatro migrações ocorrem do agregado A para o agregado B com 16 processos: {(p1, b8), (p7, b2), (p3, b1), (p4, b3)}. Por outro lado, a paralelização do algoritmo com 32 processos torna viável a migrações do agregado A para o C. Isso porque o grão de computação é menor e o espaço de memória ocupado pelos processos é inferior quando comparado com os cenários

Tabela 16: Avaliação utilizando α igual a 4 e a heurística que seleciona apenas um processo para migração (tempo em segundos)

Processos	Cenário 1		Cenário 2		Cenário 3	
	Tempo	Desvio padrão	Tempo	Desvio padrão	Tempo	Desvio padrão
8	3518,09	21,07	3712,43	33,98	3783,81	34,16
16	1969,05	17,38	2214,36	30,14	2299,73	32,67
32	1048,23	22,43	1303,33	31,57	1414,56	35,71

Tabela 17: Avaliação utilizando α igual a 8 e a heurística que seleciona apenas um processo para migração (tempo em segundos)

Processos	Cenário 1		Cenário 2		Cenário 3	
	Tempo	Desvio padrão	Tempo	Desvio padrão	Tempo	Desvio padrão
8	3518,09	21,07	3610,87	30,34	3651,23	21,98
16	1969,05	17,38	2073,30	21,06	2118,44	29,05
32	1048,23	22,43	1175,96	23,81	1226,32	25,34

anteriores. Mesmo assim, esse cenário não apresenta ganhos de desempenho. Apesar de oito migrações terem ocorrido do agregado A para o B, cinco processos permanecem em execução no agregado mais lento.

A Tabela 17 apresenta os resultados com α igual a 8 e a heurística que seleciona apenas um processo como candidato a migração. Nessa configuração, temos uma quantidade menor de chamadas para o reescalonamento e uma sobrecarga menor de LBjMigBSP. Por exemplo, 2,63% são observados ao comparar os Cenários 1 e 2 com 8 processos. Entretanto, ganhos de desempenho não são obtidos devido às poucas chamadas de reescalonamento e o fato de somente um processo ser analisado quando elas ocorrem. Nesse sentido, a Tabela 18 apresenta os ganhos alcançados com α igual a 4 e a heurística que avalia mais de um processo para migração. Assim, optamos pela análise de processos que apresentam $PM > MAX(PM) \times 0,30$. A Tabela 18 mostra que ganhos de desempenho foram alcançados ao paralelizarmos a aplicação com 16 e 32 processos. Com esses cenários, obteve-se ganhos de, respectivamente, 60,69% e 71,10%. No primeiro caso, cinco migrações são observadas na primeira chamada de reescalonamento de processos em execução no agregado A para B: {(p1, b9), (p5, b3), (p2, b1), (p7, b8), (p3, b6)}. A próxima chamada de reescalonamento ocorre na oitava superetapa e compreende a migração de outros quatro processos em execução no agregado A: {(p4, b2), (p6, b5), (p5, b7), (p8, b4)}. Em ambos os casos, migrações para o agregado C não são consideradas viáveis devido ao seu custo. Além disso, o agregado A obtém valores negativos de PM quando o reescalonamento é testado. Nesse sentido, migrações não ocorrem nas superetapas subsequentes.

Os resultados apresentados na Tabela 18 para 32 processos são caracterizados pela variação de α e D durante a execução do algoritmo. A primeira chamada de reescalonamento ocorre na quarta superetapa e compreende a migração de nove processos do agregado A para B e um

Tabela 18: Avaliação utilizando α igual a 4 e a heurística que seleciona mais de um processo para migração (tempo em segundos)

Processos	Cenário 1		Cenário 2		Cenário 3	
	Tempo	Desvio padrão	Tempo	Desvio padrão	Tempo	Desvio padrão
8	3518,09	21,07	3712,43	33,98	3814,78	21,89
16	1969,05	17,38	2214,36	30,14	1225,30	14,06
32	1048,23	22,43	1303,33	31,57	612,64	9,91

processo do agregado A para C: {(p32, b2), (p29, b1), (p30, b7), (p31, b4), (p1, b9), (p4, b8), (p7, b6), (p5, b3), (p9, b5), (p2, c8)}. A próxima chamada de reescalonamento ocorre na oitava superetapa e envolve a migração de três processos do agregado A para C: {(p6, c4), (p8, c2), (p3, c7)}. A partir da nona superetapa, os processos estão equilibrados entre si, fazendo com que o valor de α aumente a cada chamada de reescalonamento. Além disso, o valor de D é alterado na superetapa 17 de 0.5 para 0.75 e de 0.75 para 1.125 na superetapa 30.

Por fim, a Figura 30 resume os resultados obtidos com LBjMigBSP. Primeiramente, é possível observar que ganhos de desempenho não foram alcançados ao paralelizar a aplicação com 8 processos. Esse comportamento se justifica pelo número reduzido de superetapas e, consequentemente, a pequena quantidade de chamadas para o reescalonamento. De forma semelhante, o uso da heurística que avalia somente um processo para migração não apresenta redução no tempo da aplicação. Apesar desses cenários apresentarem uma maior quantidade de chamadas de reescalonamento, nem todos os processos localizados em agregados mais lentos são migrados para outros mais rápidos, limitando assim o tempo das superetapas. Nesse sentido, é possível observar ganhos de desempenho nas avaliações com α igual a 4 e o uso da heurística que avalia mais de um processo para migração. Em especial, a paralelização da aplicação com 16 e 32 processos apresenta ganhos de 60,69% e 71,10%, respectivamente.

5.4 Balanço

Esse capítulo apresentou a avaliação de jMigBSP através da utilização de duas aplicações BSP. As aplicações escolhidas foram o algoritmo que calcula a transformada rápida de Fourier (FFT) e o algoritmo de compressão de imagens segundo o método de Fractal. Inicialmente, a avaliação observou os custos de migração de ProActive. A ideia é conhecer a percentagem paga por essa biblioteca para oferecer migração de objetos. Para tanto, foi comparado o tempo de migração de objetos com o tempo de transferência desses mesmos objetos utilizando Java Sockets. A avaliação ocorreu em redes de 100 Mbits/s e 10 Mbits/s. Em ambos os ambientes, os resultados alcançados demonstram um comportamento linear, onde um aumento na quantidade de dados alocados para os objetos acarreta em um aumento proporcional no tempo de migração e transferência. Nesse sentido, a sobrecarga imposta pelo recurso de migração de ProActive não depende do tamanho dos dados manipulados. Assim, é possível observar que o custo de

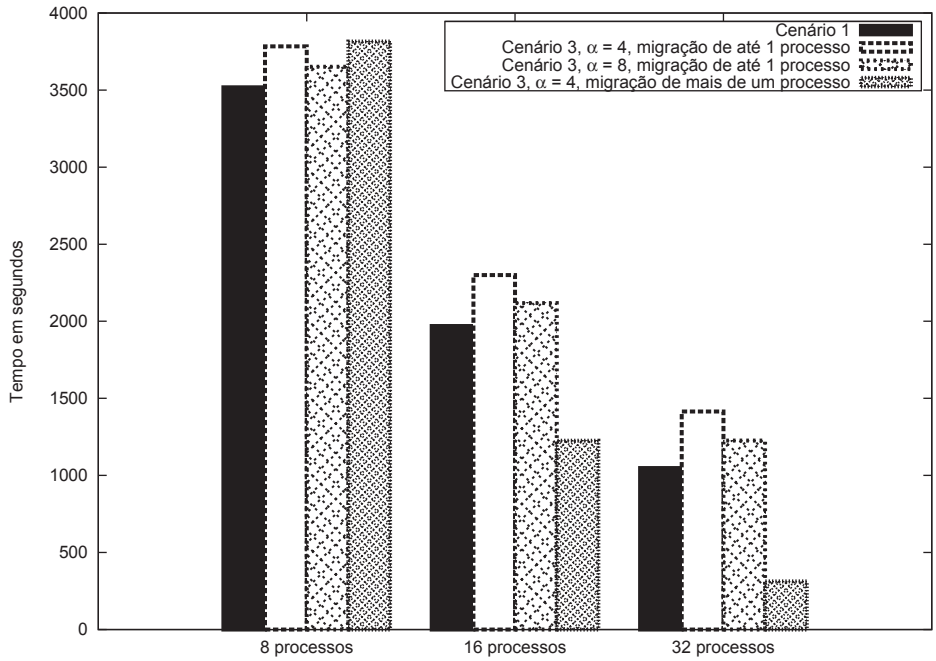

Figura 30: Observando o desempenho de LBjMigBSP em diferentes situações

ProActive diminui com o aumento do tamanho dos objetos.

A FFT foi implementada com o objetivo de conhecermos a sobrecarga de jMigBSP em relação a uma biblioteca de código nativo. Para tanto, duas versões da FFT foram desenvolvidas: (i) uma escrita em jMigBSP e; (ii) outra em BSPlib. Como esperado, o tempo de execução de jMigBSP foi superior em relação a BSPlib. Entretanto, os resultados obtidos mostram que quanto maior o grão de computação, menor é a diferença entre ambas as bibliotecas. Por exemplo, ao trabalharmos com 16 processos e um vetor de tamanho 2^{24}, 7,73 segundos e 5,75 segundos são observados com jMigBSP e BSPlib, respectivamente. Esses tempos representam uma sobrecarga de 25,63%. Por outro lado, para um vetor de tamanho 2^{25}, 16,21 segundos e 14,02 segundos são obtidos com jMigBSP e BSPlib. Esses resultados representam uma redução de 12,09% na sobrecarga de jMigBSP. Além disso, nossa avaliação observou os ganhos com a migração explícita de jMigBSP. Nesse sentido, ambas as aplicações foram executadas em um ambiente com processadores sobrecarregados. A ideia foi de inserir chamadas ao método $bsp\_migrate()$ para transferir os objetos localizados em processadores sobrecarregados para outros levemente carregados. Ganhos de 4,19% no tempo de execução da aplicação escrita em jMigBSP foram observados em relação a BSPlib.

A aplicação que realiza a compressão de imagens foi desenvolvida para avaliarmos a implementação de LBjMigBSP. Nossa avaliação abrangeu: (i) a execução da aplicação simplesmente com jMigBSP; (ii) a execução da aplicação com LBjMigBSP e sem migração e; (iii) a execução da aplicação com LBjMigBSP e migração habilitada. Nós montamos uma infraestrutura com três agregados, onde cada nó possui a capacidade de executar até dois processos simultaneamente. Além disso, os agregados são heterogêneos entre si em termos de capacidade de processamento e largura de banda. A avaliação do algoritmo de compressão compreendeu o

uso de ambas as heurísticas do modelo MigBSP. A primeira heurística escolhe somente um processo BSP para migrar, o que apresentar maior valor de PM. A segunda heurística escolhe uma percentagem de processos baseado no maior valor de PM. Os resultados mostram que quando temos um pequeno número de superetapas, não temos tempo para superar a sobrecarga imposta por LBjMigBSP. Sendo assim, quando o número de superetapas é pequeno, a quantidade de processos escolhidos para migração em cada chamada de reescalonamento pode ser determinante para o desempenho da aplicação. Por exemplo, nos testes que envolveram a primeira heurística, não foram observados ganhos de desempenho ao trabalharmos com até 32 superetapas. Por outro lado, o uso da segunda heurística apresentou ganhos de desempenho de 60,69% e 71,10% ao trabalharmos com, respectivamente, 16 e 32 processos.

6 CONCLUSÃO

Aplicações BSP são compostas por um conjunto de processos que executam superetapas. Cada superetapa compreende fases de computação e comunicação, terminando com uma barreira de sincronização entre os processos. O modelo BSP não especifica como os processos devem ser mapeados para os recursos, deixando essa questão para o programador. O tempo de cada superetapa é determinado pelo processo mais lento, devido ao uso de uma barreira de sincronização. Consequentemente, o mapeamento de processos para recursos é um tema importante para conseguir um bom desempenho ao executar aplicações baseadas no modelo BSP. Especialmente, esse reescalonamento é ainda mais importante em ambientes heterogêneos e dinâmicos como as grades computacionais.

Geralmente, as otimizações de escalonamento são implementadas no código da aplicação para atender as necessidades do ambiente computacional (EL KABBANY et al., 2011). Embora uma alta otimização possa ser alcançada, essa tarefa pode ser tediosa e precisa de uma experiência considerável. Além disso, essa abordagem não é portável para diferentes ambientes computacionais e cada aplicação diferente exige um novo esforço para o escalonamento dos processos. Apesar do escalonamento inicial ser fortemente relevante para obter desempenho, o mapeamento inicial de processos para recursos pode não permanecer eficiente durante o tempo. Assim, uma possibilidade é redistribuir os processos durante a execução da aplicação através da migração de processos. Nesse contexto, algumas iniciativas usam chamadas de migração explícitas no código da aplicação (GALINDO; ALMEIDA; BADÍA-CONTELLES, 2008). Nessa técnica, o desenvolvedor deve conhecer detalhes sobre a arquitetura de máquina paralela e a aplicação. Uma abordagem diferente para o rebalanceamento de carga acontece em nível de *middleware*. Comumente, essa abordagem não exige mudanças no código da aplicação, nem conhecimento prévio sobre o sistema. Portanto, o desenvolvedor pode se concentrar na escrita da aplicação sem se preocupar com decisões de implementação de migrações.

Esse trabalho apresentou o desenvolvimento e a avaliação do sistema jMigBSP, bem como as tecnologias e os sistemas de *software* necessários para a sua confecção. Ele oferece uma interface de programação Java para a escrita de aplicações BSP. Além disso, jMigBSP possui mecanismos para o reescalonamento de objetos em nível de aplicação e de *middleware*. Em especial, a abordagem em nível de *middleware* realiza a realocação dos objetos de forma transparente para o programador, que só vai perceber as mudanças no tempo de execução da aplicação. O desenvolvimento de jMigBSP é suportado pela biblioteca ProActive. O sistema jMigBSP tira proveito de quatro ideias dessa biblioteca: (i) a facilidade no mapeamento de objetos para recursos; (ii) o modelo de programação SPMD; (iii) a migração de objetos e; (iv) o modelo de comunicação assíncrona. Além dos mecanismos para o reescalonamento de objetos, jMigBSP tem a capacidade de gerir comunicação assíncrona e *one-sided*. Em adição, os métodos de comunicação e de sincronização implementam a semântica de comunicação do modelo BSP. Dessa forma, todas as mensagens trocadas em uma superetapa estarão disponíveis para

uso somente na superetapa seguinte.

6.1 Resultados Obtidos

Nós estudamos e implementamos duas aplicações BSP com o objetivo de avaliar jMigBSP. Os algoritmos avaliados foram: (i) transformada rápida de Fourier (FFT) e; (ii) compressão de imagens. Primeiramente, nós concluímos que jMigBSP apresenta uma baixa sobrecarga em relação a BSPlib, uma biblioteca C para a escrita de aplicações BSP. Quanto maior o grão de computação, menor é a sobrecarga imposta por jMigBSP. Levando em consideração todas as execuções do algoritmo da FFT, a menor sobrecarga observada foi de 13,54%. Em adição, os nossos testes observaram o desempenho de ambas as bibliotecas em um ambiente com processadores sobrecarregados. Nessa avaliação, jMigBSP superou em 29,44% o desempenho de BSPlib ao habilitarmos a migração em nível de aplicação. Chamadas explícitas ao método de migração foram inseridas na aplicação escrita com jMigBSP para transferir os objetos localizados em processadores sobrecarregados para outros levemente carregados.

O algoritmo de compressão de imagens foi desenvolvido para avaliarmos o recurso de balanceamento de carga automático de jMigBSP. Esse algoritmo foi executado em um ambiente heterogêneo formado por três agregados. Os resultados obtidos mostram que ganhos de desempenho são alcançados com aplicações que apresentam uma grande quantidade de superetapas. Em adição, esse resultado é mais fácil de ser alcançado se mais de um objeto é migrado durante uma chamada de reescalonamento. Nesses cenários, ganhos de até 71,10% foram alcançados ao executar o algoritmo de compressão com 32 superetapas e α igual a 4.

6.2 Contribuições

Essa seção trata das contribuições do sistema jMigBSP. Ele oferece uma interface para a escrita de aplicações BSP em Java. No âmbito de bibliotecas de programação BSP, as contribuições de jMigBSP são:

(i) Abordagem de reescalonamento de objetos BSP em nível de aplicação e de *middleware*;

(ii) Estratégia de alocação contínua para o mapeamento de objetos para recursos.

A abordagem de reescalonamento de objetos em nível de aplicação e de *middleware* é uma das bases do desenvolvimento de jMigBSP e diz respeito à contribuição científica desse trabalho. Essas abordagens são ilustradas na Figura 31. O reescalonamento em nível de aplicação é aquele controlado pelo programador. Através do método $bsp\_migrate()$, o programador transfere qualquer objeto BSP durante a fase de computação (ver Figura 31 (a)). A migração pode ocorrer para um novo computador ou para um computador em que outro objeto está em execução. A primeira ideia é pertinente quando sabemos que o computador de destino está levemente

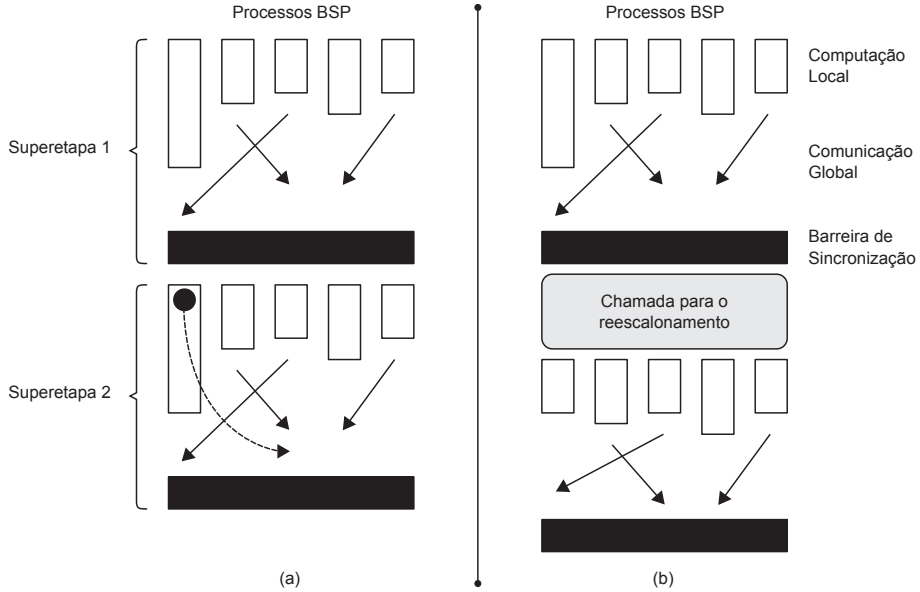

Figura 31: Abordagens de reecalonamento de jMigBSP: (a) reescalonamento em nível de aplicação; (b) reescalonamento em nível de *middleware*.

carregado e possui recursos computacionais suficientes para a execução da aplicação. Já a segunda ideia é favorável para aproximar os objetos que se comunicam com frequência. Por outro lado, o reescalonamento em nível de *middleware* é aquele que ocorre sem a intervenção do programador. O desenvolvimento dessa abordagem é guiado pelo modelo MigBSP e o reescalonamento ocorre após a barreira de sincronização (ver Figura 31 (b)). Chamadas para o reescalonamento de processos acontecem a cada α superetapas. O índice α é utilizado para informar o intervalo de superetapas entre as chamadas de reescalonamento. Esse índice aumenta se o sistema se encontra estável no tempo de conclusão das superetapas e diminui caso contrário.

A contribuição técnica desse trabalho refere-se a implementação da estratégia de alocação contínua no mapeamento de objetos para recursos. O uso dessa estratégia é pertinente para a nossa avaliação, uma vez que as aplicações revelam um caráter de comunicação entre objetos cujos índices são próximos. Em especial, diferente da abordagem oferecida por ProActive, a estratégia de alocação adotada por jMigBSP prioriza o uso de todas as máquinas do ambiente computacional. Apesar das máquinas de nosso ambiente suportarem a execução de até dois processos em paralelo, durante o lançamento da aplicação, cada nó recebe, em um primeiro momento, somente um objeto. Sendo assim, quando a quantidade de objetos é maior em relação ao número de máquinas, o mapeamento começa novamente a partir do primeiro agregado.

Por fim, nossos trabalhos sobre migração e jMigBSP foram publicados em congressos, revistas científicas e em capítulo de livro. Além de congressos regionais como a ERAD (Escola Regional de Alto Desempenho) e ERRC (Escola Regional de Redes de Computadores), ambos os tópicos geraram publicações em eventos como PDCAT (*Parallel and Distributed Computing, Applications and Technologies*), ICPADS (*International Conference on Parallel and Distributed Systems*) e WSCAD-SSC (Simpósio em Sistemas Computacionais). Além disso, tem-se um

capítulo no livro *Production Scheduling*, publicado pela editora InTech, e artigos nas revistas JACR (*Journal of Applied Computing Research*) e JCC (*Journal of Communication and Computer*). O artigo da revista JACR foi publicado em dezembro de 2011, enquanto que o artigo da revista JCC será publicado na edição de maio de 2012.

6.3 Trabalhos Futuros

A continuação desse trabalho consiste nas seguintes direções:

(i) Implementar a heurística AutoMig, que seleciona um conjunto de processos candidatos à migração automaticamente;

(ii) Selecionar uma aplicação que demande uma quantidade maior de superetapas;

(iii) Avaliar jMigBSP quando alterada a disponibilidade dos recursos ao longo da execução da aplicação.

Originalmente, o modelo MigBSP sugere duas heurísticas para selecionar os processos candidatos à migração (RIGHI et al., 2010). A primeira escolhe o processo candidato no topo de uma lista decrescente construída com os valores de PM. A segunda escolhe os processos com PM maior que $MAX(PM).x$, onde x é um percentual. Apesar do uso da segunda heurística ter apresentado ganhos de desempenho em nossas avaliações, concorda-se que uma questão pertinente é sobre como achar uma percentagem otimizada para aplicações dinâmicas e ambientes heterogêneos. Nesse contexto, o trabalho de Graebin et al. (2011) apresenta a heurística AutoMig, que seleciona um ou mais candidatos a migração automaticamente. Ela tira proveito dos conceitos de *List Scheduling* e *Backtracking* para avaliar, de maneira autônoma, o impacto da migração sobre cada elemento da lista de PM. Para cada processo i descrito em $PM(i, j)$, AutoMig simula sua execução aplicando uma predição pf com o intuito de medir o desempenho do novo escalonamento. Esse escalonamento é calculado através do teste de realocação do processo i para o Conjunto j denotado como parâmetro de PM.

A segunda atividade consiste em avaliar jMigBSP com uma aplicação que demande uma quantidade maior de superetapas. Isso porque, em nossos testes, não foi possível observar ganhos de desempenho utilizando a heurística que seleciona apenas um objeto para migração. Nesse sentido, uma nova aplicação permitiria comparar os ganhos de desempenho dessa heurística em relação àquela que seleciona mais de um objeto. Finalmente, sabemos que as grades computacionais apresentam problemas de disponibilidade dos recursos. Assim, pretendemos avaliar jMigBSP ao modificar essa disponibilidade. Para tanto, a ferramenta cpulimit pode ser utilizada para variar o poder computacional dos recursos ao longo do tempo.

REFERÊNCIAS

AMEDRO, B.; BAUDE, F.; CAROMEL, D.; DELBÉ, C.; FILALI, I.; HUET, F.; MATHIAS, E.; SMIRNOV, O. An Efficient Framework for Running Applications on Clusters, Grids, and Clouds. In: SAMMES, A. J.; ANTONOPOULOS, N.; GILLAM, L. (Ed.). **Cloud Computing**. [S.l.]: Springer London, 2010. p. 163–178. (Computer Communications and Networks, v. 0).

BADUEL, L.; BAUDE, F.; CAROMEL, D. Efficient, flexible, and typed group communications in Java. In: ACM-ISCOPE CONFERENCE ON JAVA GRANDE, 2002., 2002, New York, NY, USA. **Proceedings...** ACM, 2002. p. 28–36. (JGI '02).

BADUEL, L.; BAUDE, F.; CAROMEL, D. Object-oriented SPMD. In: FIFTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID'05) - VOLUME 2 - VOLUME 02, 2005, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2005. p. 824–831. (CCGRID '05).

BATISTA, D. M.; FONSECA, N. L. S. da; MIYAZAWA, F. K.; GRANELLI, F. Self-adjustment of resource allocation for grid applications. **Computer Networks**, New York, NY, USA, v. 52, p. 1762–1781, June 2008.

BAUDE, F.; CAROMEL, D.; DALMASSO, C.; DANELUTTO, M.; GETOV, V.; HENRIO, L.; PéREZ, C. GCM: a grid extension to fractal for autonomous distributed components. **Annals of Telecommunications**, [S.l.], v. 64, p. 5–24, 2009.

BAUDE, F.; CAROMEL, D.; HUET, F.; MESTRE, L.; VAYSSIÈRE, J. Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 11., 2002, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2002. p. 93–102. (HPDC '02).

BAUDE, F.; CAROMEL, D.; HUET, F.; VAYSSIÈRE, J. Communicating Mobile Active Objects in Java. In: INTERNATIONAL CONFERENCE ON HIGH-PERFORMANCE COMPUTING AND NETWORKING, 8., 2000, London, UK. **Proceedings...** Springer-Verlag, 2000. p. 633–643. (HPCN Europe 2000).

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

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

BONORDEN, O.; GEHWEILER, J.; HEIDE, F. der. A Web Computing Environment for Parallel Algorithms in Java. In: WYRZYKOWSKI, R.; DONGARRA, J.; MEYER, N.; WASNIEWSKI, J. (Ed.). **Parallel Processing and Applied Mathematics**. [S.l.]: Springer Berlin / Heidelberg, 2006. p. 801–808. (Lecture Notes in Computer Science, v. 3911).

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

CAMARGO, R. de; KON, F.; GOLDMAN, A. Portable checkpointing and communication for BSP applications on dynamic heterogeneous grid environments. In: COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2005. SBAC-PAD 2005. 17TH INTERNATIONAL SYMPOSIUM ON, 2005. **Anais...** [S.l.: s.n.], 2005. p. 226 – 233.

CAROMEL, D. Toward a method of object-oriented concurrent programming. **Commun. ACM**, New York, NY, USA, v. 36, p. 90–102, September 1993.

CAROMEL, D.; KLAUSER, W.; VAYSSIERE, J. Towards Seamless Computing and Metacomputing in Java. In: CONCURRENCY PRACTICE AND EXPERIENCE, 1998. **Anais...** Wiley & Sons: Ltd., 1998. v. 10, n. 11–13, p. 1043–1061.

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

CESARIO, E.; GRILLO, A.; MASTROIANNI, C.; TALIA, D. A Sketch-Based Architecture for Mining Frequent Items and Itemsets from Distributed Data Streams. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING, 2011., 2011, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2011. p. 245–253. (CCGRID '11).

CHA, H.; LEE, D. H-BSP: a hierarchical bsp computation model. **J. Supercomput.**, Hingham, MA, USA, v. 18, n. 2, p. 179–200, 2001.

CHAUBE, R.; CARINO, R.; BANICESCU, I. Effectiveness of a Dynamic Load Balancing Library for Scientific Applications. In: PARALLEL AND DISTRIBUTED COMPUTING, 2007. ISPDC '07. SIXTH INTERNATIONAL SYMPOSIUM ON, 2007. **Anais...** [S.l.: s.n.], 2007. p. 32.

CHEN, L.; ZHU, Q.; AGRAWAL, G. Supporting dynamic migration in tightly coupled grid applications. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2006., 2006, New York, NY, USA. **Proceedings...** ACM, 2006. (SC '06).

COOLEY, J. W.; TUKEY, J. W. An Algorithm for the Machine Calculation of Complex Fourier Series. **Mathematics of Computation**, [S.l.], v. 19, n. 90, p. 297–301, 1965.

CUNHA, C. A.; SOBRAL, J. L. An Annotation-Based Framework for Parallel Computing. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING, 15., 2007, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p. 113–120. (PDP '07).

DE GRANDE, R. E.; BOUKERCHE, A. Dynamic balancing of communication and computation load for HLA-based simulations on large-scale distributed systems. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v. 71, p. 40–52, January 2011.

DONGARRA, J. J.; OTTO, S. W.; SNIR, M.; WALKER, D. **An Introduction to the MPI Standard**. Knoxville, TN, USA: University of Tennessee, 1995.

DU, C.; SUN, X.-H.; WU, M. Dynamic Scheduling with Process Migration. In: SEVENTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2007, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p. 92–99. (CCGRID '07).

DUTOT, P.-F.; NETTO, M.; GOLDMAN, A.; KON, F. Scheduling Moldable BSP Tasks. In: FEITELSON, D.; FRACHTENBERG, E.; RUDOLPH, L.; SCHWIEGELSHOHN, U. (Ed.). **Job Scheduling Strategies for Parallel Processing**. [S.l.]: Springer Berlin / Heidelberg, 2005. p. 157–172. (Lecture Notes in Computer Science, v. 3834).

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, p. 143–182, 2011.

FAHRINGER, T.; JUGRAVU, A. JavaSymphony: a new programming paradigm to control and synchronize locality, parallelism and load balancing for parallel and distributed computing: research articles. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, v. 17, p. 1005–1025, June 2005.

GALINDO, I.; ALMEIDA, F.; BADÍA-CONTELLES, J. M. Dynamic Load Balancing on Dedicated Heterogeneous Systems. In: EUROPEAN PVM/MPI USERS' GROUP MEETING ON RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 15., 2008, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2008. p. 64–74.

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.

GETOV, V.; LASZEWSKI, G. von; PHILIPPSEN, M.; FOSTER, I. Multiparadigm communications in Java for grid computing. **Commun. ACM**, New York, NY, USA, v. 44, p. 118–125, October 2001.

GONZALEZ, J.; LEON, C.; PICCOLI, F.; PRINTISTA, M.; RODA, J.; RODRIGUEZ, C.; SANDE, F. de. Oblivious BSP. In: BODE, A.; LUDWIG, T.; KARL, W.; WISMüLLER, R. (Ed.). **Euro-Par 2000 Parallel Processing**. [S.l.]: Springer Berlin / Heidelberg, 2000. p. 682–685. (Lecture Notes in Computer Science, v. 1900).

GOUDREAU, M.; LANG, K.; RAO, S.; SUEL, T.; TSANTILAS, T. Towards efficiency and portability: programming with the bsp model. In: ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 1996, New York, NY, USA. **Proceedings...** ACM, 1996. p. 1–12. (SPAA '96).

GRAEBIN, L.; ARAÚJO, D. P. d.; RIGHI, R. d. R. Auto-Organização na Seleção de Candidatos a Migração em Aplicações Bulk Synchronous Parallel. In: X WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAçÃO, 2011, Natal, RN. **Anais...** [S.l.: s.n.], 2011. p. 2061–2074.

GRAEBIN, L.; RIGHI, R. d. R. jMigBSP: Object Migration and Asynchronous One-Sided Communication for BSP Applications. In: THE 12TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING, APPLICATIONS AND TECHNOLOGIES, 2011, Gwangju, Korea. **Anais...** [S.l.: s.n.], 2011. p. 35–38.

GRAEBIN, L.; SILVA, M. R.; RIGHI, R. d. R. Modelagem e Implementação Bulk Synchronous Parallel da Compressão de Imagens Baseada em Fractais. In: X SIMPÓSIO DE INFORMÁTICA DA REGIÃO CENTRO/RS, 2011, Santa Maria, RS. **Anais...** [S.l.: s.n.], 2011. p. 26–31.

GROPP, W.; THAKUR, R. An Evaluation of Implementation Options for MPI One-Sided Communication. In: DI MARTINO, B.; KRANZLMüLLER, D.; DONGARRA, J. (Ed.). **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. [S.l.]: Springer Berlin / Heidelberg, 2005. p. 415–424. (Lecture Notes in Computer Science, v. 3666).

GU, Y.; LEE, B.-S.; CAI, W. JBSP: a bsp programming library in java. **Journal of Parallel and Distributed Computing**, [S.l.], v. 61, n. 8, p. 1126 – 1142, 2001.

GUO, Y.; CHEN, X.; DENG, M.; WANG, Z.; LV, W.; XU, C.; WANG, T. The Fractal Compression Coding in Mobile Video Monitoring System. In: COMMUNICATIONS AND MOBILE COMPUTING, 2009. CMC '09. WRI INTERNATIONAL CONFERENCE ON, 2009. **Anais...** [S.l.: s.n.], 2009. v. 3, p. 492–495.

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

HÜTTER, C.; MOSCHNY, T. Runtime Locality Optimizations of Distributed Java Applications. In: EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP 2008), 16., 2008, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p. 149–156.

JIANG, Y.; TONG, W.; ZHAO, W. Resource Load Balancing Based on Multi-agent in ServiceBSP Model. In: COMPUTATIONAL SCIENCE, PART III: ICCS 2007, 7., 2007, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2007. p. 42–49. (ICCS '07).

KANG, P.; SELVARASU, N.; RAMAKRISHNAN, N.; RIBBENS, C.; TAFTI, D.; VARADARAJAN, S. Modular, Fine-Grained Adaptation of Parallel Programs. In: ALLEN, G.; NABRZYSKI, J.; SEIDEL, E.; ALBADA, G. van; DONGARRA, J.; SLOOT, P. (Ed.). **Computational Science - ICCS 2009**. [S.l.]: Springer Berlin / Heidelberg, 2009. p. 269–279. (Lecture Notes in Computer Science, v. 5544).

KOTZMANN, T.; WIMMER, C.; MOSSENBOCK, H.; RODRIGUEZ, T.; RUSSELL, K.; COX, D. Design of the Java HotSpot client compiler for Java 6. **ACM Trans. Archit. Code Optim.**, New York, NY, USA, v. 5, p. 7:1–7:32, May 2008.

KWOK, Y.-K.; CHEUNG, L.-S. A new fuzzy-decision based load balancing system for distributed object computing. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v. 64, p. 238–253, February 2004.

LAWRENCE, R. **A survey of process migration mechanisms**. [S.l.]: University of Manitoba, 1998.

LOBOSCO, M.; AMORIM, C.; LOQUES, O. Java for high-performance network-based computing: a survey. **Concurrency and Computation: Practice and Experience**, [S.l.], v. 14, n. 1, p. 1–31, 2002.

LU, K.; SUBRATA, R.; ZOMAYA, A. Towards Decentralized Load Balancing in a Computational Grid Environment. In: CHUNG, Y.-C.; MOREIRA, J. (Ed.). **Advances in Grid and Pervasive Computing**. [S.l.]: Springer Berlin / Heidelberg, 2006. p. 466–477. (Lecture Notes in Computer Science, v. 3947).

MARTIN, J. M. R.; TISKIN, A. V. Dynamic BSP: Towards a Flexible Approach to Parallel Computing over the Grid. In: COMMUNICATING PROCESS ARCHITECTURES 2004, 2004. **Anais...** [S.l.: s.n.], 2004. p. 219–226.

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

PASCUAL, J.; NAVARIDAS, J.; MIGUEL-ALONSO, J. Effects of Topology-Aware Allocation Policies on Scheduling Performance. In: FRACHTENBERG, E.; SCHWIEGELSHOHN, U. (Ed.). **Job Scheduling Strategies for Parallel Processing**. [S.l.]: Springer Berlin / Heidelberg, 2009. p. 138–156. (Lecture Notes in Computer Science, v. 5798).

PHILIPPSEN, M.; ZENGER, M. JavaParty - Transparent Remote Objects in Java. **Concurrency: Practice and Experience**, [S.l.], v. 9, n. 11, p. 1225–1242, 1997.

POMINVILLE, P.; QIAN, F.; VALLéE-RAI, R.; HENDREN, L.; VERBRUGGE, C. A framework for optimizing Java using attributes. In: CASCON FIRST DECADE HIGH IMPACT PAPERS, 2010, New York, NY, USA. **Anais...** ACM, 2010. p. 225–241.

PONTELLI, E.; LE, H. V.; SON, T. C. An investigation in parallel execution of answer set programs on distributed memory platforms: task sharing and dynamic scheduling. **Comput. Lang. Syst. Struct.**, Amsterdam, The Netherlands, v. 36, p. 158–202, July 2010.

RIGHI, R. d. 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. 2, p. 123–144, 2010.

SHAFI, A.; CARPENTER, B.; BAKER, M.; HUSSAIN, A. A comparative study of Java and C performance in two large-scale parallel applications. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, v. 21, p. 1882–1906, October 2009.

SINHA, P. K. **Distributed Operating Systems**: concepts and design. 1st. ed. [S.l.]: Wiley-IEEE Press, 1996.

SKILLICORN, D. B.; HILL, J. M. D.; MCCOLL, W. F. Questions and Answers about BSP. **Scientific Programming**, [S.l.], v. 6, n. 3, p. 249–274, 1997.

SMITH, J. M. A survey of process migration mechanisms. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v. 22, p. 28–40, July 1988.

SONG, J.; TONG, W.; ZHI, X. ServiceBSP Model with QoS Considerations in Grids. In: ADVANCED WEB AND NETWORK TECHNOLOGIES, AND APPLICATIONS, 2006. **Anais...** Springer, 2006. p. 827–834. (Lecture Notes in Computer Science, v. 3842).

SUIJLEN, W. J.; BISSELING, R. H. **BSPonMPI**. http://bsponmpi.sourceforge.net/.

SUNDERAM, V. S. PVM: a framework for parallel distributed computing. **Concurrency: Pract. Exper.**, Chichester, UK, v. 2, p. 315–339, November 1990.

TABOADA, G. L.; TOURIÑO, J.; DOALLO, R. Java for high performance computing: assessment of current research and practice. In: INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA, 7., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p. 30–39. (PPPJ '09).

TAN, Y.; AUFENANGER, M. A real-time rescheduling heuristic using decentralized knowledge-based decisions for flexible flow shops with unrelated parallel machines. In: IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL INFORMATICS, 9., 2011, Washington, DC, USA. **Proceedings...** IEEE Comp. Society, 2011. p. 431–436. (INDIN '11).

TIOBE SOFTWARE. **TIOBE Programming Community Index for February 2012**. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em: 29/02/2012.

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

VASILEV, V. P. BSPGRID: variable resources parallel computation and multiprogrammed parallelism. **Parallel Processing Letters**, [S.l.], v. 13, n. 3, p. 329–340, 2003.

VERA, G.; SUPPI, R. ATLS - A parallel loop scheduling scheme for dynamic environments. **Procedia Computer Science**, [S.l.], v. 1, n. 1, p. 583–591, 2010.

WARAICH, S. S. Classification of Dynamic Load Balancing Strategies in a Network of Workstations. In: FIFTH INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: NEW GENERATIONS, 2008, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p. 1263–1265.

WILLIAMS, T. L.; PARSONS, R. J. The Heterogeneous Bulk Synchronous Parallel Model. In: IPDPS '00: PROCEEDINGS OF THE 15 IPDPS 2000 WORKSHOPS ON PARALLEL AND DISTRIBUTED PROCESSING, 2000, London, UK. **Anais...** Springer-Verlag, 2000. p. 102–108.

XING, C. An Adaptive Domain Pool Scheme for Fractal Image Compression. In: EDUCATION TECHNOLOGY AND TRAINING, 2008. AND 2008 INTERNATIONAL WORKSHOP ON GEOSCIENCE AND REMOTE SENSING. ETT AND GRS 2008. INTERNATIONAL WORKSHOP ON, 2008. **Anais...** [S.l.: s.n.], 2008. v. 2, p. 719–722.

XU, B.; LIAN, W.; GAO, Q. Migration of active objects in proactive. **Information and Software Technology**, [S.l.], v. 45, n. 9, p. 611–618, 2003.

YAGOUBI, B.; MEDEBBER, M. A load balancing model for grid environment. In: COMPUTER AND INFORMATION SCIENCES, 2007. ISCIS 2007. 22ND INTERNATIONAL SYMPOSIUM ON, 2007. **Anais...** [S.l.: s.n.], 2007. p. 1 –7.

YAMIN, A. C. Escalonamento em Sistemas Paralelos e Distribuídos. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 1., 2001, Gramado, RS. **Anais...** Porto Alegre: RS, 2001. p. 75–126.

YANG, C.-C.; CHEN, C.-K.; CHANG, Y.-H.; CHUNG, K.-H.; LEE, J.-K. Streaming support for Java RMI in distributed environments. In: PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA, 4., 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p. 53–61. (PPPJ '06).

YU, Z.; SHI, W. An Adaptive Rescheduling Strategy for Grid Workflow Applications. **Parallel and Distributed Processing Symposium, International**, Los Alamitos, CA, USA, v. 0, p. 115, 2007.

YZELMAN, A. N.; BISSELING, R. H. An Object-Oriented Bulk Synchronous Parallel Library for Multicore Programming. **Concurrency and Computation: Practice and Experience**, [S.l.], 2011.

ZHANG, Y.; KOELBEL, C.; COOPER, K. Hybrid Re-scheduling Mechanisms for Workflow Applications on Multi-cluster Grid. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2009., 2009, Washington, DC, USA. **Proceedings…** IEEE Computer Society, 2009. p. 116–123. (CCGRID '09).