Tratamento Flexível do Reescalonamento de Processos em Aplicações BSP: Migração em Nível de Middleware ou de Aplicação

Lucas Graebin, Rodrigo da Rosa Righi
Programa Interdisciplinar de Pós-Graduação em Computação Aplicada (PIPCA)
Universidade do Vale do Rio dos Sinos (UNISINOS)
Av. Unisinos, 950 – 93.022-000 – São Leopoldo – RS – Brazil
lgraebin@acm.org, rrrighi@unisinos.br

***Resumo*—Este artigo descreve as razões para o desenvolvimento de jMigBSP - uma biblioteca de programação Java que oferece reescalonamento de objetos. jMigBSP foi projetado para trabalhar em ambientes de *Grids* Computacionais e oferece uma interface que segue o modelo BSP (*Bulk Synchronous Parallel*). A principal contribuição de jMigBSP diz respeito as facilidades de reescalonamento em duas maneiras: (i) usando diretivas de migração no código da aplicação e; (ii) através do balanceamento de carga automático em nível de *middleware*. Em especial, esta segunda ideia é viável graças a característica de herança do Java, que permite transformar uma aplicação jMigBSP simples em outra migrável, alterando apenas uma linha de código. Além disso, jMigBSP facilita a interação entre os objetos através de métodos para comunicação do tipo *one-sided* e assíncrona. A avaliação compreendeu o desenvolvimento da aplicação de FFT. As execuções mostram jMigBSP como uma biblioteca competitiva quando comparada com BSPlib, uma biblioteca C para a escrita de aplicações BSP. Além de sua interface simples, os pontos fortes de jMigBSP também consideram os testes de migração que supera o tempo de BSPlib.**

I. Introdução

Balanceamento de carga é uma questão pertinente para obter desempenho em ambientes heterogêneos e dinâmicos. Ele pode ser ativado pelo reescalonamento de entidades como processos, tarefas ou objetos. Esta técnica é útil para migrar entidades para a execução mais rápida em recursos levemente carregados e/ou para aproximar aquelas que se comunicam com maior frequência. Além disso, a facilidade de migração pode ser oferecida em nível de aplicação ou de *middleware* [1]. A primeira ideia pode usar chamadas explícitas no código fonte da aplicação, enquanto a segunda representa uma extensão da biblioteca de programação para oferecer um mecanismo transparente para o usuário. Pesquisas sobre reescalonamento de entidades incluem a definição de métricas unificadas para agir em resposta a dinamicidade da aplicação e da infraestrutura, além de interfaces de programação para prover a facilidade de migração [1].

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 *Grids* Computacionais [2]. BSP representa um estilo comum para a escrita de aplicações paralelas bem sucedidas [2]. Lattice Boltzmann, sequenciamento de DNA e previsão meteorológica são exemplos de problemas implementados com este modelo. Enquanto isso, Java tem a característica de ser multiplataforma e oferece classes para a escrita de aplicações distribuídas que escondem detalhes técnicos de desenvolvedores. Além disso, o fato de Java ser interpretado não é mais um obstáculo impeditivo de uso. A diferença de desempenho entre Java e linguagens nativas (como C e Fortran) tem diminuído nos últimos anos graças ao compilador *Just-In-Time* (JIT) da máquina virtual [3].

Nesse contexto, estamos desenvolvendo uma biblioteca de programação chamada jMigBSP. Ela permite a escrita de aplicações BSP em Java e seu diferencial diz respeito as facilidades de reescalonamento em dois níveis: (i) **aplicação** e; (ii) ***middleware***. O desenvolvimento de jMigBSP é suportado pela biblioteca ProActive, da qual herda ideias como a migração de objetos e a possibilidade de trabalhar em ambientes compostos por diversos *clusters* [4]. Além disso, nossa biblioteca oferece diretivas para comunicação *one-sided* e assíncrona. A avaliação de jMigBSP compreendeu o desenvolvimento das aplicações da Soma de Prefixos e Transformada Rápida de Fourier (FFT). Em especial, a FFT também foi desenvolvida em BSPlib, uma biblioteca C para a escrita de aplicações BSP [5]. Além de uma descrição de jMigBSP, este artigo discute os resultados obtidos na sua avaliação. Eles mostram que quanto maior o grão de computação, menor a sobrecarga imposta por uma linguagem interpretada. Em adição, testes de migração revelam situações em que jMigBSP supera BSPlib, enfatizando os benefícios desta técnica.

II. jMigBSP: Biblioteca de Programação Java para o Reescalonamento de Objetos

A biblioteca jMigBSP oferece uma interface de programação (API) Java para a escrita de aplicações BSP. Ela herda características de ProActive e propõe novos mecanismos para seguir o modelo de programação BSP. jMigBSP tira proveito de quatro ideias de ProActive: (i) a facilidade no mapeamento de objetos para recursos; (ii) o modelo de programação OOSPMD; (iii) a migração de objetos e; (iv) o modelo de comunicação assíncrona. A primeira ideia permite informar os recursos computacionais existentes na rede em arquivos XML. No que tange o modelo OOSPMD, ele permite o lançamento de objetos de uma mesma classe em diferentes nós. Todos os objetos começam sua execução em um determinado método e a interação entre eles acontece através de RMI. Além disso, ProActive oferece migração e deixa um procurador no nó de origem que é usado para realizar chamadas sobre o

objeto migrado. Por fim, o assincronismo em jMigBSP é expresso pelas técnicas de espera pela necessidade e objetos futuros de ProActive [4]. A RMI sempre retorna um objeto futuro para o chamador, permitindo que ele continue sua execução imediatamente.

A biblioteca jMigBSP esconde a necessidade de o programador compreender o funcionamento de ProActive. Por exemplo, detalhes sobre a leitura dos arquivos XML e a criação e o gerenciamento de grupos são abstraídos do desenvolvedor através de um único método em jMigBSP. Uma aplicação jMigBSP deve implementar a semântica de uma superetapa no método $run()$ (ver Seção II-A). Uma superetapa é composta pelas fases de computação e comunicação, seguidas por uma barreira de sincronização. A coleção destas fases representa uma aplicação BSP. Assim como o modelo BSP, jMigBSP garante que uma mensagem enviada em uma superetapa será entregue somente no início da próxima.

A. Escrevendo uma Aplicação em jMigBSP

Uma aplicação jMigBSP deve estender a classe $jMigBSP$ e implementar o método $run()$. Este método contém o código que será executado em paralelo. A Tabela I apresenta as funções que podem ser usadas no método $run()$. Primeiramente, a quantidade de objetos paralelos pode ser consultada com a chamada ao método $bsp\_nprocs()$, e o identificador de cada objeto pode ser recuperado com $bsp\_pid()$. O método $bsp\_sync()$ é uma barreira que sincroniza todos os objetos. Ao terminá-lo, todas as mensagens enviadas na superetapa anterior estarão disponíveis para uso no *buffer* do objeto de destino.

Tabela I
COLEÇÃO DE MÉTODOS DE JMIGBSP

Classificação	Método	Descrição
Consulta	bsp_nprocs	Quantidade de objetos
	bsp_pid	Identificador do objeto
Superetapa	bsp_sync	Barreira de sincronização
Comunicação	bsp_put	Copia para a memória remota
	bsp_get	Copia da memória remota
Reescalonamento	$bsp\_migrate$	Migra o objeto chamador

A biblioteca jMigBSP possui métodos de comunicação *one-sided* que permitem ao programador ler e escrever diretamente na memória de um objeto remoto. O método $bsp\_put()$ armazena dados na memória de um objeto destino sem a participação ativa deste. Este método tem um comportamento assíncrono, permitindo ao programador executar cálculos úteis enquanto a RMI é realizada. Em aspectos de implementação, todos os objetos possuem dois vetores de *buffer* com o objetivo de implementar a semântica de comunicação do modelo BSP. Um vetor tem os dados que foram recebidos na superetapa anterior, enquanto o outro atua como receptor dos dados na superetapa corrente. O primeiro vetor é chamado de Ativo, enquanto o segundo de Temporário. Além disso, ambos têm o tamanho de $n - 1$, onde n é a quantidade de objetos. Após o fim de uma superetapa, o vetor Ativo é preenchido com o conteúdo do Temporário. Assim, a operação $bsp\_get()$ atinge o vetor Ativo de outro objeto a fim de copiar os valores mantidos ali.

A Figura 1 ilustra o uso das funções de jMigBSP. Ela demonstra a operação da Soma de Prefixos, onde cada processo p armazena um número diferente para o cálculo. O algoritmo usa a técnica logarítmica que computa log_p superetapas. Assim, os objetos na faixa de $2^{k-1} \leq i \leq p$ combinam suas somas parciais durante a k-*ésima* superetapa. Depois de criar uma instância da classe $PrefixSum$, a aplicação invoca o método $start()$ informando o número de objetos paralelos como parâmetro.

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

Figura 1. Algoritmo da Soma de Prefixos escrito em jMigBSP

No algoritmo da Soma de Prefixos, cada objeto inicia com o seu próprio identificador. Na primeira superetapa, cada objeto i (exceto o último) envia o seu valor para o seu vizinho da direita $i + 1$. Nessa operação, o objeto emissor coloca o seu valor no vetor Temporário do objeto destino. Assim, todos os objetos (exceto o primeiro) possuirão um número que é a soma de dois inteiros. Seguindo o algoritmo, o último objeto terá em seu *buffer* o resultado final da computação na terceira e última superetapa. Esse valor poderá ser recuperado através do método $bsp\_get()$.

B. Reescalonamento Explícito de Objetos

O reescalonamento permite o remapeamento inicial de objetos para recursos em resposta ao comportamento da aplicação e da infraestrutura. Para tanto, jMigBSP oferece duas implementações para o método $bsp\_migrate()$. A primeira consiste na migração do objeto chamador para um nó remoto que é recebido como parâmetro. A segunda consiste em transferir o objeto chamador para o mesmo nó em que outro objeto está em execução. Esta chamada recebe o código identificador do objeto como parâmetro. O reescalonamento explícito requer que o programador possua experiência em algoritmos de balanceamento de carga. Ele deve coletar dados sobre o escalonamento manualmente. Além disso, uma nova aplicação e/ou infraestrutura exigirá um novo esforço em estudar os melhores lugares para adicionar chamadas de migração.

C. Balanceamento de Carga Automático

jMigBSP também tem como objetivo oferecer balanceamento de carga automático sem a intervenção do programador. Para tanto, o modelo de reescalonamento MigBSP [6] será implementado e oferecido através da interface de jMigBSP. Nosso objetivo final é reduzir o tempo da aplicação, tornando as superetapas mais curtas sem o esforço do programador. MigBSP responde as seguintes questões sobre a migração de processos: (i) "Quando"

lançar a migração; (ii) "Quais" objetos são candidatos e; (iii) "Onde" colocar os objetos eleitos. MigBSP atua sobre uma arquitetura que é montada com a ideia de Conjuntos (diferentes *clusters*) e Gerentes de Conjuntos. Gerentes de Conjuntos são responsáveis pela captura de dados sobre o escalonamento de um Conjunto específico e a troca deles com os demais Gerentes. A decisão do reescalonamento é tomada no final de uma superetapa. Além disso, MigBSP responde a pergunta "Quais" usando uma função de decisão chamada Potencial de Migração (PM). Cada objeto i computa n funções $PM(i,j)$, onde n é o número de Conjuntos e j representa um Conjunto específico. $PM(i,j)$ é encontrado usando as métricas de Computação, Comunicação e Memória da seguinte forma: $PM(i,j) = Comp(i,j) + Comm(i,j) - Mem(i,j)$.

Nós criamos uma classe denominada $LBjMigBSP$ que estende $jMigBSP$ a fim de adicionar a captura de dados para o escalonamento. A Figura 2 ilustra os métodos que LBjMigBSP sobrescreve de jMigBSP. Os métodos apresentados nas linhas 2 e 6 capturam informações sobre o escalonamento e acionam os métodos de jMigBSP para a troca de mensagens. O método $bsp\_sync()$ cria um vetor para armazenar o tempo da superetapa de cada objeto. Quando o reescalonamento é ativado, o método $computeBalance()$ troca os vetores entre os Gerentes, permitindo-lhes conhecer o próximo intervalo de superetapas para o reescalonamento de objetos. Após obter o PM, a viabilidade de migração é testada.

```
1. public class LBjMigBSP extends jMigBSP {
2.   public void bsp_put(Object o, int destination) {
3.     captureDataPut(o, destination);
4.     super.bsp_put(o, destination);
5.   }
6.   public Object bsp_get(int destination) {
7.     Object temp = super.bsp_get(destination);
8.     captureDataReceive(temp, destination);
9.     return temp;
10.  }
11.  public void bsp_sync(){
12.    double[] vec_steps = new double[MAX_SUPERSTEPS];
13.    computeSuperstepTime(vec_steps);
14.    if (isSuperstepRescheduling()) {
15.       next_call = computeBalance(vec_steps);
16.       exchageDataAmongSetManagers(computePM());
17.       machine = willMigrate();
18.       if (machine != null)
19.          bsp_migrate(machine);
20.    }
21.    super.bsp_sync();
22.  }
23.}
```

Figura 2. Derivando jMigBSP para oferecer balanceamento de carga

Para usar nosso protótipo de reescalonamento, o programador deve alterar a linha que define a classe da aplicação. Assim, ao invés de estender a classe $jMigBSP$ (ver Figura 1), a aplicação deve estender $LBjMigBSP$. LBjMigBSP representa a nossa abordagem de *middleware*, agindo como um *wrapper* para o balanceamento de carga. LBjMigBSP é um trabalho em andamento. A métrica de Computação de PM considera as instruções entre as barreiras, assim como o tempo gasto em ações de computação. A métrica Comunicação trabalha com o número de bytes enviados ou recebidos de/para um Conjunto. A métrica Memória considera a memória de um objeto, a largura de banda para alcançar um Conjunto e os custos relacionados a migração.

III. Resultados Experimentais

Os testes foram executados em um *cluster* composto por 16 nós Intel Core 2 Duo de 2,93 GHz com 4 GB de memória. Eles estão ligados por uma rede de 10 Mbps e toda comunicação ocorre através de TCP/IP. Mais precisamente, ProActive usa Java RMI, *threads* e reflexão para viabilizar a comunicação em jMigBSP. Quanto as versões, utilizou-se ProActive 5.0.3 e BSPlib 1.4.

A. Análise do Custo de Migração

A avaliação comparou os tempos de migração de objetos com ProActive e os tempos de transferência de objetos com Java Sockets. Três nós foram reservados para os testes. O primeiro executa uma aplicação sintética que cria um objeto ativo no segundo com uma quantidade específica de dados. A aplicação chama a diretiva de migração sobre o objeto ativo, movendo-o para o terceiro nó. Além desta aplicação, uma outra foi escrita para criar um objeto Java e transferi-lo usando Sockets.

Dados de 1KB até 32MB foram alocados pelos objetos. Basicamente, os resultados obtidos apresentam um comportamento linear. Por exemplo, 8,49 segundos foram obtidos com a migração de um objeto de 10MB com ProActive, enquanto 7,29 segundos foram alcançados com Java Sockets. Quando alocado 32MB, 30,32 e 28,89 segundos foram observados com ProActive e Java Sockets, respectivamente. Assim, 1,20 e 1,43 segundos são obtidos quando subtraído os tempos de Java a partir de ProActive. Em outras palavras, a sobrecarga imposta pela migração de ProActive não depende diretamente do tamanho dos dados manipulados.

B. Desempenho de jMigBSP em Aplicações Paralelas

Nós implementamos duas aplicações BSP: (i) Soma de Prefixos e; (ii) Transformada Rápida de Fourier (FFT). A Soma de Prefixos foi implementada e executada em nosso *cluster* para validar o correto funcionamento de jMigBSP. A FFT foi desenvolvida para analisar o desempenho de jMigBSP. Nesse sentido, implementamos duas versões da FFT: (i) uma escrita em jMigBSP e; (ii) outra em BSPlib. FFT é um algoritmo rápido para o cálculo da Transformada Discreta de Fourier (DFT). Nossa implementação recebe como entrada um vetor de tamanho n com números complexos que são distribuídos em p processadores [7]. Cada processador atua sobre $\frac{n}{p}$ valores. O algoritmo calcula a transformada desses elementos, redistribui os dados e sincroniza os objetos. A segunda superetapa combina os resultados para produzir a transformada de Fourier.

A Tabela II mostra os tempos de execução da aplicação da FFT. O tamanho do vetor variou de 2^{22} até 2^{25}. Como esperado, BSPlib teve um melhor desempenho em relação a jMigBSP. Este resultado é justificado pela sobrecarga que Java e ProActive impõem sobre jMigBSP. No entanto, podemos observar que quanto maior o grão de computação, menor a diferença no tempo entre ambas as bibliotecas. Por exemplo, 7,73 segundos foram obtidos

com jMigBSP quando utilizado 16 processadores e vetor de tamanho 2^{24}, enquanto 5,75 segundos foram alcançados com BSPlib. Estes tempos representam uma sobrecarga de 25,61%. No entanto, com um vetor de tamanho 2^{25}, 16,22 e 14,02 segundos foram obtidos com jMigBSP e BSPlib, respectivamente. Esta execução indica uma redução do custo de jMigBSP estimada em 13,56%.

Tabela II
TEMPO DE EXECUÇÃO (EM SEGUNDOS) DA FFT

Tamanho do vetor	8 processadores		16 processadores	
	jMigBSP	BSPlib	jMigBSP	BSPlib
2^{22}	2,89	1,85	1,69	0,99
2^{23}	5,83	3,92	3,91	2,42
2^{24}	13,78	10,54	7,73	5,75
2^{25}	27,32	22,01	16,22	14,02

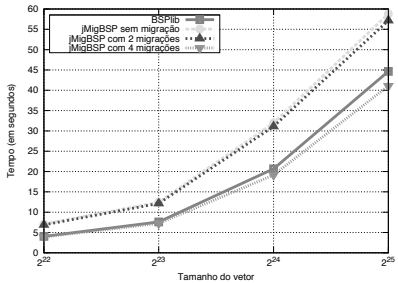

Figura 3. Desempenho da FFT com migração explícita em jMigBSP

Os resultados da execução de jMigBSP com migração habilitada são apresentados na Figura 3. Chamadas explícitas para $bsp\_migrate()$ após a barreira foram utilizadas nos testes. Nós executamos a FFT em 8 nós e uma sobrecarga foi simulado em 4 deles. A migração de um objeto de um processador sobrecarregado para outro levemente carregado busca minimizar o tempo de execução da aplicação. A Figura 3 mostra que jMigBSP supera BSPlib ao migrar 4 objetos. A migração de dois objetos não apresenta resultados satisfatórios devido a própria regra de BSP. Apesar da transferência de dois objetos, dois outros permanecem em processadores sobrecarregados, limitando o tempo da superetapa.

IV. TRABALHOS RELACIONADOS

BSPlib foi uma das primeiras bibliotecas BSP [5]. Basicamente, ela contém funções em C para delimitar superetapas e operações para acesso remoto a memória (DRMA) e troca de mensagens (BSMP). Outras bibliotecas BSP escritas em C são a Paderborn University BSP (PUB) [2] e BSPonMPI [8]. PUB oferece as mesmas funcionalidades de BSPlib, mas inclui primitivas para comunicação assíncrona e uma segunda semântica para a barreira de sincronização. BSPonMPI oferece as funcionalidades básicas de BSPlib e executa em implementações de MPI. Além de bibliotecas de código nativo, podemos citar algumas iniciativas em Java. JBSP [9], MulticoreBSP [10] e PUBWCL [11] aparecem como as mais significativas. JBSP é um sistema para ambientes *peer-to-peer* e oferece métodos de comunicação DRMA e BSMP. Por outro lado, MulticoreBSP atua em ambientes com diversos núcleos. Nele, processos interagem entre si através de memória compartilhada. Em adição, PUBWCL e PUB suportam migração. PUBWCL pode migrar um processo BSP durante a sua fase de computação, bem como após a barreira de sincronização. PUB oferece estratégias centralizadas e distribuídas para o balanceamento de carga. Ambas oferecem apenas a abordagem de migração com chamadas explícitas no código da aplicação.

V. CONCLUSÃO

A migração com jMigBSP pode ser acionada pelo programador ou pelo *middleware* de escalonamento. Essa decisão depende do grau de controle desejado. Assim, o programador pode escolher em colocar chamadas a $bsp\_migrate()$ na aplicação ou estender $LBjMigBSP$ ao invés de $jMigBSP$. A primeira abordagem é pertinente para os desenvolvedores que almejam construir os seus algoritmos de migração. A segunda representa uma abordagem "sem esforço" em nível de programação para tentar um melhor desempenho para a aplicação. A avaliação experimental mostrou resultados encorajadores, uma vez que jMigBSP obteve um desempenho competitivo se comparada com BSPlib. Além disso, conclui-se que aplicações que demandam alta computação tendem a ter mais benefícios da facilidade de migração. Trabalhos futuros compreendem a implementação de uma aplicação *CPU-Bound* de compressão de imagens com jMigBSP.

REFERÊNCIAS

[1] G. El Kabbany, N. Wanas, N. Hegazi, and S. Shaheen, "A dynamic load balancing framework for real-time applications in message passing systems," *International Journal of Parallel Programming*, vol. 39, pp. 143–182, 2011.

[2] O. Bonorden, "Load balancing in the bulk-synchronous-parallel setting using process migrations," in *IPDPS, 2007. IEEE International*, pp. 1 –9.

[3] G. L. Taboada, J. Touriño, and R. Doallo, "Java for high performance computing: assessment of current research and practice," ser. PPPJ '09. New York, NY, USA: ACM, 2009.

[4] D. Caromel, W. Klauser, and J. Vayssiere, "Towards seamless computing and metacomputing in java," in *Concurrency Practice and Experience*, G. C. Fox, Ed., vol. 10, no. 11–13. Wiley & Sons, Ltd., September-November 1998.

[5] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, "Bsplib: The bsp programming library," *Parallel Computing*, vol. 24, no. 14, pp. 1947 – 1980, 1998.

[6] R. d. R. Righi, L. L. Pilla, N. Maillard, A. Carissimi, and P. O. A. Navaux, "Observing the impact of multiple metrics and runtime adaptations on bsp process rescheduling," *Paral. Proc. Letters*, vol. 20, no. 2, pp. 123–144, 2010.

[7] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004.

[8] W. J. Suijlen and R. H. Bisseling, "BSPonMPI," 2011, http://bsponmpi.sourceforge.net/.

[9] Y. Gu, B.-S. Lee, and W. Cai, "Jbsp: A bsp programming library in java," *Journal of Parallel and Distributed Computing*, vol. 61, no. 8, pp. 1126 – 1142, 2001.

[10] A. N. Yzelman and R. H. Bisseling, "An object-oriented BSP library for multicore programming," 2011.

[11] O. Bonorden, J. Gehweiler, and F. der Heide, "Load balancing strategies in a web computing environment," in *Paral. Proc. and Applied Math.*. Springer, vol. 3911, pp. 839–846.