Obtendo Desempenho Através de Mobilidade Forte de Código

Gustavo Lermen, Fabiane Cristine Dillenburg, Jorge Luis Victoria Barbosa
Universidade do Vale do Rio dos Sinos (UNISINOS)
Programa Interdisciplinar de Pós-Graduação em Computação Aplicada (PIPCA)
São Leopoldo – RS, Brasil, 93022-000
glermen@unisinos.br, fdillenburg@turing.unisinos.br, jbarbosa@unisinos.br

Resumo

*Dentre as propostas desenvolvidas para tirar proveito da heterogeneidade de dispositivos e sistemas operacionais disponíveis atualmente, encontra-se a mobilidade de código. Através da utilização de uma camada software subjacente em comum, este artigo apresenta uma solução que utiliza mobilidade de código para obter paralelismo na execução de tarefas. O modelo de mobilidade forte de código apresentado chama-se Holo*Go*. Este modelo utiliza como plataforma de execução uma máquina virtual, chamada HoloVM, que oferece suporte à programação concorrente e* blackboards*. A utilização do modelo proposto é materializada através da implementação de uma aplicação que visa o ganho de desempenho na execução de uma tarefa computacionalmente intensa.*

1. Introdução

A mobilidade de código pode ser vista como uma alternativa a necessidade de personalização de software impulsionada pelo crescente avanço de diferentes tecnologias. Dentre as principais, pode-se citar as redes de computadores, que hoje independem de cabos para oferecer alta conectividade, bem como os avanços em hardware, que possibilitam que dispositivos cada vez menores ofereçam cada vez maior poder computacional. Aliado a estes avanços, tem-se também a redução de custos, possibilitando desta maneira que estes avanços estejam disponíveis para um número cada vez maior de pessoas.

Vários projetos de infra-estrutura computacional foram propostos para suprir as necessidades da infra-estrutura de comunicação. Estes esforços focaram diferentes características do problema em camadas diferentes de abstração. A maioria das abordagens, entretanto, tentou adaptar modelos e tecnologia bem conhecidos. Uma abordagem diferente, por sua vez, explora código móvel.

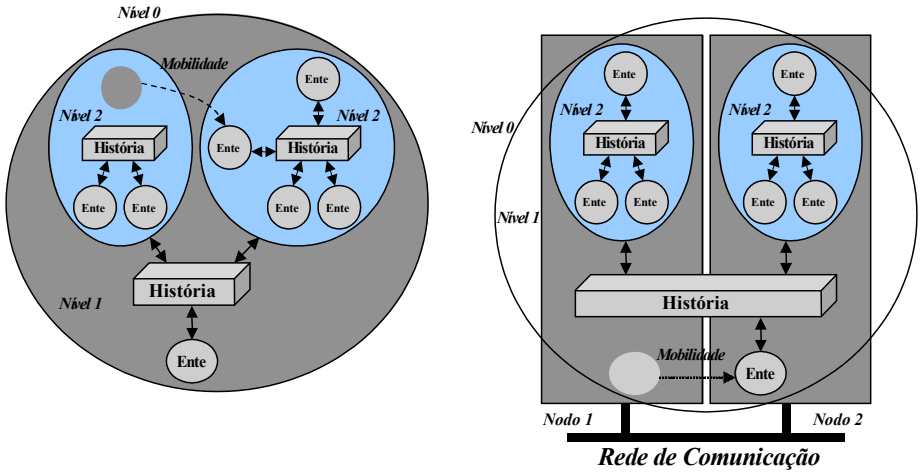
Mobilidade de código é um paradigma de programação importante e abre novas possibilidades para estruturar sistemas de software distribuídos em um ambiente aberto em constante mudança. Pode melhorar a velocidade, flexibilidade, estrutura, tratamento de desconexões e é especialmente indicado quando adaptação e a flexibilidade estão entre as exigências principais da aplicação.

Este artigo apresenta um modelo de mobilidade forte de código no contexto do Holoparadigma [1]. O foco é o projeto e a implementação de um sistema de mobilidade forte de código que usa a HoloVM [8] como plataforma. HoloVM é uma máquina virtual com suporte a *blackboards* [16] e programação concorrente. O artigo descreve, ainda, testes experimentais de validação do modelo e uma aplicação que utiliza mobilidade de código para ganhar desempenho em um ambiente heterogêneo.

O artigo está organizado da seguinte forma. A seção 2 apresenta conceitos básicos de mobilidade de código. Na seção 3 descreve-se o Holoparadigma juntamente com o ambiente de execução distribuído. A seção 4 aborda o Holo*Go*, o modelo de mobilidade forte de código proposto. Na mesma seção, resultados experimentais são expostos. Um cenário de aplicação do modelo é apresentado na seção 5. A seção 6 descreve trabalhos relacionados. Por fim, a seção 7 encerra o artigo com considerações finais e perspectivas de trabalhos futuros.

2. Mobilidade de Código

Diversas abordagens foram propostas para resolver os problemas relacionados com os desafios impostos pela evolução dos dispositivos computacionais e das redes de computadores. A maioria delas baseou-se em tecnologias bem estabelecidas que são adaptadas a esta nova realidade visando fornecer personalização e flexibilidade. Muitas dessas tecnologias, como CORBA [13], utilizam o paradigma cliente/servidor. Entretanto, outros trabalhos [7] [9] sugerem que uma abordagem utilizando mobilidade de código seja mais adequada a um ambiente carente de flexibilidade.



(a) Mobilidade Lógica

(b) Mobilidade física sem mobilidade lógica

Figura 1. Tipos de Mobilidade.

A mobilidade de código pode ser definida informalmente como a capacidade de mudar dinamicamente as "ligações" entre fragmentos de código e o lugar em que executam [7]. As tecnologias de mobilidade de código incluem linguagens de programação e seu suporte de tempo de execução correspondente. Um MCS (*Mobile Code System*) pode ser visto como uma camada de software colocada no topo do sistema operacional nativo. Esse sistema expõe a noção de localização, de maneira que o programador está sempre ciente da posição em que um fragmento de código está executando.

O modelo MCS apresenta os conceitos de "Ambiente Computacional" (CE) e de "Componente". O CE expõe o conceito de lugar. Um código estará sempre executando em um ambiente computacional. Um componente hospedado por um CE pode ser uma unidade de execução (*Execution Unit* - EU) ou um recurso. As unidades de execução podem ser vistas como fluxos seqüenciais da execução, assim como um processo ou uma *thread* em um processo *multi-threaded*. Os recursos são componentes que podem ser compartilhados entre múltiplas EUs.

Dois tipos de mobilidade de código são apresentados na literatura [7] [12] [9]: mobilidade forte e mobilidade fraca. A mobilidade forte de código permite que uma unidade de execução mova-se mantendo seu estado interno de execução. A execução pára no CE origem e é reiniciada no CE destino no mesmo ponto de execução. Na mobilidade fraca de código, somente o código é movido. Em alguns casos, dados necessários para iniciar a execução no CE destino também são movidos, mas o estado interno não é mantido. Duas abordagens são apresentadas para prover mobilidade forte de código: migração de código e clonagem remota. Na migração de código, a execução da EU é parada, serializada e, então, migrada ao CE destino. Na clonagem remota, a execução não é parada quando a migração ocorre. A EU continua sua execução no CE origem quando o código é movido para o CE destino. Ambas as abordagens podem ser pró-ativas ou reativas. Na abordagem pró-ativa, a própria EU provoca a mobilidade de código. Na abordagem reativa, a mobilidade ocorre devido a um evento externo, como uma mensagem recebida de uma outra EU.

O conceito de mobilidade de código, mesmo não sendo recente, aparece como uma alternativa para diversos desafios impostos pela evolução tecnológica. A mobilidade de código pode ser aplicada em áreas como distribuição de carga, tolerância a falhas, compartilhamento de recursos, localidade no acesso aos dados, computação móvel, personalização de serviços e sistemas ubíquos.

3. Holoparadigma

O Holoparadigma [3] [1] é baseado em uma abstração de contexto chamada ente, usada para suportar mobilidade. Um ente elementar é organizado em três partes: comportamento, *interface* e história. O comportamento define o conjunto de ações que o ente é capaz de executar. Dentre essas, as que podem ser acessadas externamente são descritas na *interface*. A história consiste em um espaço de tuplas que fica encapsulado no ente. Um ente composto possui a mesma organização; no entanto, suporta a existência de outros entes na sua composição (entes componentes). Nesse caso, a história é compartilhada pelos entes componentes.

No Holoparadigma, há dois tipos da mobilidade: (i) mobilidade lógica e (ii) mobilidade de código. A mobilidade lógica relaciona-se com o deslocamento em nível de modelagem, sem considerações sobre a plataforma de execução. A mobilidade de código relaciona-se com o deslocamento entre nodos de uma arquitetura distribuída. Desta forma, um ente move-se quando se desloca de um nodo para outro. A Figura 1 exemplifica os dois tipos em um ente com três níveis de composição. Após o movimento, o ente movido é incapaz de acessar a história e as ações da origem (Figura 1a). Entretanto, agora o ente tem acesso a história e as ações do destino. Neste cenário, a mobilidade de código ocorre somente se os entes origem e destino estiverem em nós diferentes da arquitetura distribuída (Figura 1b).

Mobilidade lógica e mobilidade de código são independentes. A ocorrência de uma não implica na ocorrência de outra. No exemplo, a Figura 1b mostra a mobilidade de código sem mobilidade lógica. Nesse caso, o ente movido não muda sua visão da história (suportada pelo *blackboard*). Esse tipo de situação somente pode acontecer se o ambiente de execução fornecer a localização dos entes.

Uma linguagem de programação, conhecida como Hololinguagem, foi projetada especialmente para explorar os conceitos apresentados pelo Holoparadigma [2]. Ela foi criada para o desenvolvimento de sistemas distribuídos.

A execução de um programa Holo cria uma estrutura hierárquica de entes, denominada Árvore de Entes (HoloTree). Essa estrutura é usada para organizar entes durante a execução. A árvore implementa o encapsulamento dos mesmos em níveis de composição, conforme proposto pelo Holoparadigma. A Figura 2a exemplifica a HoloTree para o ente mostrado na Figura 1a. A HoloTree suporta ainda o aspecto dinâmico da política de grupos, mudando continuamente durante a execução de um programa. Ações como a clonagem e a mobilidade de entes são exemplos de ações que modificam a HoloTree. Uma mobilidade lógica é executada movendo uma folha (ente elementar) ou um ramo da árvore (ente composto) origem para o ente destino. O ente movido tem acesso direto ao espaço do ente destino (história do ente composto). A Figura 2b mostra a mudança na HoloTree causada pela mobilidade apresentada na Figura 1a.

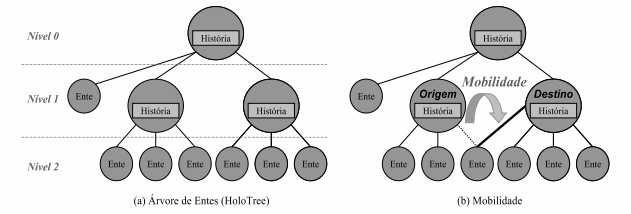


Figura 2. HoloTree.

3.1. HoloVM

O suporte à execução de programas baseados em *bytecode* Holo é provido por uma máquina virtual chamada HoloVM [8]. A HoloVM cria uma camada de abstração entre o programa e o hardware subjacente. Isso permite que os programas em Holo sejam executados em toda a plataforma que a HoloVM for suportada, facilitando os pré-requisitos da distribuição. Atualmente, há implementações da HoloVM para as plataformas Windows, Windows Mobile, MacOSX e Linux.

A execução de um programa ocorre através de um arquivo binário predefinido. Esse arquivo contém uma tabela de símbolos e as instruções (*bytecode*) a serem executadas. Qualquer compilador pode gerar *bytecode* suportado pela HoloVM, desde que respeite o conjunto de instruções da mesma.

A HoloVM oferece um conjunto de instruções específico para fornecer as funcionalidades propostas pelo Holoparadigma. A mobilidade de código ocorre quando a instrução *move* é chamada e o ente destino está situado em um outro dispositivo. Essa instrução muda a composição de um ente movendo-o para outro ente. Neste trabalho, a instrução *move* é focalizada.

3.2. Ambiente de Execução Distribuído

Um modelo de execução foi proposto e implementado a fim explorar características de sistemas distribuídos fornecidas pelo Holoparadigma. Esse é chamado de HNS (*Holo Naming System*) [4] e compreende dois componentes: o servidor HNS e uma camada de comunicação acoplada à HoloVM.

O objetivo principal do HNS é prover um serviço que permita a execução de programas distribuídos entre diversas HoloVMs. Para tanto, ele mantém a localização de todos os entes que estão executando no ambiente. Essa informação de localização possibilita que duas HoloVMs possam comunicar-se a fim de explorar os recursos propostos pelo Holoparadigma.

A comunicação entre a HoloVM e o servidor HNS é provida por uma camada de software acoplada à HoloVM. Essa camada executa consultas e informa ao servidor HNS sobre mudanças ocorridas internamente na HoloVM. Os mecanismos envolvidos nesse processo são descritos em [4].

4. Holo*Go*

O Holo*Go* tem por objetivo principal disponibilizar um serviço de mobilidade forte de código aos programas baseados em *bytecode* Holo. Holo*Go* pode ser visto como uma camada de software acoplada à HoloVM. Antes do desenvolvimento de Holo*Go*, a HoloVM suportava apenas a

mobilidade lógica de entes. A mobilidade forte de código não requer modificações na Hololinguagem. Quando a instrução *move* é executada, a mobilidade de código pode ocorrer dependendo da localização do ente destino. Se o ente destino não estiver na mesma HoloVM, então ocorre mobilidade de código. Se ele estiver na mesma, ocorre apenas mobilidade lógica. A HoloVM pode ser vista como um ambiente computacional utilizado como plataforma de execução aos programas baseados em *bytecode* Holo. O ente, por sua vez, pode ser visto como uma unidade de execução, uma vez que irá mover-se entre HoloVMs (dispositivos). A Figura 3 mostra como HoloVM implementa o conceito de localização [7].

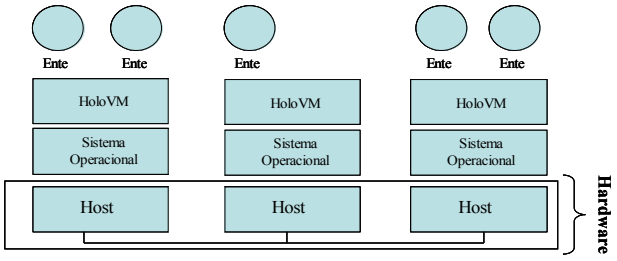


Figura 3. HoloVM como Ambiente Computacional.

O ente sempre está ligado a uma HoloVM específica. No Holo*Go*, os entes podem mover-se através de HoloVMs junto com o seu estado de execução. O estado de execução é composto pelos seguintes elementos: pilha de operandos, pilha de controle, comportamento, história e *interface*. A Figura 4 descreve a estrutura interna de um ente.

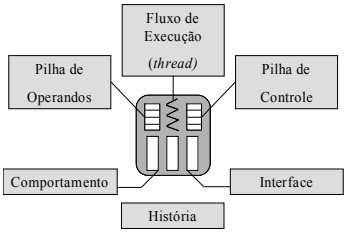


Figura 4. Estrutura do Ente.

Os seguintes passos são necessários para obtenção da mobilidade forte de código: (i) parar a execução de um ente; (ii) serializar os dados de um ente; (iii) transmitir os dados a outro ambiente; (iv) reconstruir, a partir dos dados recebidos, o ente e seu estado de execução no ambiente destino; e (v) reiniciar a execução no ambiente destino.

Todas essas etapas são tratadas por Holo*Go* quando a mobilidade de código é requerida pela instrução *move*. Essa instrução verifica se o ente destino está na mesma HoloVM. Se não estiver, a HoloVM solicita ao HNS a localização da HoloVM destino. Usando essa informação Holo*Go* envia o ente a HoloVM destino. Na chegada, é o Holo*Go* da HoloVM destino quem irá, a partir dos dados serializados recebidos, recriar o ente, inseri-lo na hierarquia e reiniciar sua execução. Esse mecanismo é demonstrado na Figura 5.

Na etapa 1, a HoloVM pede ao HNS a localização do destino. O HNS responde o pedido com a localização na etapa 2. A mobilidade forte de código ocorre efetivamente na etapa 3. Na etapa 4, a HoloVM destino notifica o HNS sobre a posse do novo ente. Na etapa 5, o HNS muda seu estado interno para refletir o novo cenário.

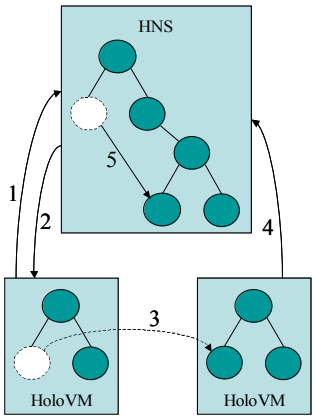


Figura 5. Etapas da Mobilidade.

No modelo proposto, a mobilidade de código pode ser pró-ativa ou reativa [7]. O ente pode executar uma ação em outro ente [3] [4]. Essa ação pode provocar a mobilidade de código no ente destino, caracterizando a mobilidade reativa de código. Da mesma maneira, o próprio ente pode mover-se para outro ambiente caracterizando uma mobilidade de código pró-ativa.

No Holo*Go*, um ente composto ao ser movido leva consigo seus entes componentes. Ao detectar uma mobilidade de código, o ente composto avisa seus entes componentes para que estes parem suas execuções e estejam aptos a serem movidos. Do ponto de vista da HoloTree, ao mover um ente composto, todo um ramo da árvore é movido enquanto que a mobilidade de um ente elementar implica na movimentação de uma folha da árvore.

4.1. Resultados Experimentais

Experimentos foram realizados a fim de validar o modelo proposto e de medir tempos necessários para as etapas que definem uma operação de mobilidade de código.

Quando ocorre uma operação de mobilidade de código, três elementos determinam o tempo necessário da mesma: tempo de serialização, tempo de envio à HoloVM destino

```
1  // FUNCAO: Simulacao de datamining
2  // OBSERVACOES:
3  // 1) Cria um mineiro;
4  // 2) O mineiro entra em um mina (de outra HoloVM)
5  //     e minera;
6  // 3) A mineracao consiste em uma simples leitura
7  //     na historia.
8
9  //**********  ENTE PRINCIPAL **********
10 holo() {     // Ente principal.
11   holo() {  // Acao guia.
12     writeln('HOLO: Vou criar a casa de um mineiro.');
13     clone(miner_home, miner_office);
14   }
15 }
16
17 //****** ENTE CASA DE UM MINEIRO ******
18 miner_home() {
19   miner_home() {
20     writeln('Casa mineiro: Criando um mineiro.');
21     clone(miner, miner_d);
22     time(Inicio);
23     //Obtem o resultado de uma mineracao
24     history#list(#Ident,#Num,#Fibo);
25     writeln('Mineiro terminou o servico.');
26     writeln('Terminou a mineracao da mina:',Ident);
27     writeln('Fibonacci de ',Num, 'e ', Fibo);
28     time(Fim);
29     writeln('Casa mineiro: Terminou a mineracao.');
30     writeln('Tempo = ',(Fim-Inicio),' milisegundos');
31   }
32 }
33
34 //**********  ENTE MINEIRO **********
35 miner() {
36   miner() {
37     writeln('MINEIRO: Inicio da mineracao.');
38     // Passo 1 - Entra na mina 1
39     move(self, mine_d1);
40     writeln('Minerando.');
41     // Passo 2 - Minera a mina 1
42     mining(1,#Num1,#Res1);
43     // Passo 3 - Salva o resultado
44     out(history)!list(1,Num1,Res1);
45     writeln('Mineiro voltando para casa.');
46     // Passo 4 - Sai da mina 1
47     move(self, miner_office);
48     // Passo 5 - Salva o resultado
49     out(history)!list(1,Num1,Res1);
50   }
51
52   // acao que realiza a mineracao
53   mining(Ident,Num,Result) {
54     // Minera a historia externa para obter um dado
55     out(history)#list(Ident,#Num);
56     // Chama acao para determinar Fibonacci
57     fib(Num,#Result);
58   }
59
60   fib(Num, Result) {
61     if (Num <= 1) {
62       Result := 1;
63     }
64     else {
65       Value1 := Num - 1;
66       Value2 := Num - 2;
67       fib(Value1, #Result1);
68       fib(Value2, #Result2);
69       Result := Result1 + Result2;
70     }
71   }
72 }
```

Figura 6. Código Fonte da Aplicação de Teste.

e tempo de deserialização. Inicialmente, foram testados os tempos de serialização e deserialização. Para tanto, foi criado um ente elementar que computa esses tempos. Esse mesmo ente foi utilizado nas execuções com entes compostos, quando outros entes foram movidos para dentro dele antes da serialização. Os resultados obtidos nestes experimentos são mostrados na Tabela 1.

Tabela 1. Serialização e Deserialização

Entes Filhos	Tempo de Serialização	Tempo de Deserialização	Desvio Padrão
0	0,015	0,015	0,0008
1	0,016	0,016	0,0009
2	0,017	0,017	0,0009
3	0,018	0,018	0,0010

Esses experimentos foram realizados em uma máquina com um processador Athlon XP de 1,8 Ghz com 512 Mb de memória RAM utilizando o sistema operacional Windows XP Professional. Os resultados apresentados nas colunas dois e três representam a média de dez execuções. A partir dos dados apresentados na Tabela 1, observa-se que o tempo de serialização de um ente elementar em relação ao ente composto é proporcionalmente maior. Além disso, nota-se que o tempo de serialização aumenta linearmente à medida em que são acrescentados entes componentes. Isto ocorre devido à estrutura de execução de um ente na HoloVM. No protótipo atual de Holo*Go*, todos os entes componentes de um ente composto compartilham a mesma *Constant Pool* (estrutura que mantém uma lista de todas as constantes necessárias para a execução de um ente). Desta maneira, quando se move um ente elementar ou um ente composto, somente uma instância da *Constant Pool* é movida.

Visando obter o tempo necessário para o envio de um ente a outro ambiente computacional, o mesmo ente utilizado no experimento anterior foi serializado, enviado a outra HoloVM onde foi deserializado e enviado de volta à origem. O tempo necessário para completar a operação foi computado no primeiro ambiente computacional, de onde o ente foi instanciado e enviado ao segundo. Os tempos de serialização e deserialização em ambos os ambientes foram subtraídos do tempo total para obter somente o tempo de envio.

Para este experimento, duas máquinas foram utilizadas. Ambas com processador Athlon XP 2800 com 512 Mb de memória RAM utilizando o sistema operacional Windows XP Professional e conectadas através de uma rede ethernet de 10 Mbits.

A mesma abordagem do primeiro experimento foi utilizada: inicialmente, foi enviado um ente elementar. Em um segundo momento, foi enviado um ente composto com apenas um filho. Em seguida, o experimento foi repetido com entes compostos com dois e três filhos. Os resultados obti-

Tabela 2. Envio e Recebimento

Entes Filhos	Tempo (segundos)	Desvio Padrão
0	0,009	0,0009
1	0,014	0,0002
2	0,016	0,0001
3	0,019	0,0001

dos, em dez execuções, são mostrados na Tabela 2 e correspondem somente ao tempo necessário para o envio de um ente a outro ambiente computacional.

Através dos dados obtidos, pode-se observar uma tendência linear no tempo de envio dos entes. À medida que entes filhos são adicionados à hierarquia, o tempo cresce proporcionalmente. O ente utilizado neste experimento possui somente uma ação que realiza o cálculo da série de Fibonacci. A Figura 6 mostra uma instância do código fonte da aplicação utilizada nos testes, neste caso o ente movido é elementar.

5. Aplicações

Esta seção mostra uma aplicação desenvolvida usando o Holo*Go*. O objetivo principal é situar o modelo proposto no contexto de aplicações reais. Assim, a subseção seguinte apresenta um cenário que se beneficia da utilização de mobilidade de código. Para o cenário apresentado foi implementado um programa, utilizando a Hololinguagem, que foi testado em um ambiente com suporte do Holo*Go*.

5.1. Ganho de Desempenho

A aplicação desenvolvida visa o ganho de desempenho na execução de uma tarefa computacionalmente intensa. A aplicação recebe como entrada uma matriz e, baseada nos recursos disponíveis no ambiente, divide a mesma em vários pedaços iguais. Cada pedaço da matriz é atribuído a um ente que se move para um ente do tipo *holder* a fim de aplicar o processamento ao seu pedaço da entrada. O ente que se move carrega os dados (pedaço da matriz) e também o código a ser aplicado. Utilizando esta abordagem, um novo algoritmo pode ser adicionado à aplicação apenas inserindo uma nova ação no ente que o aplica. A Figura 7 mostra o código fonte referente ao ente que se move para um ente do tipo *holder*.

Inicialmente, a aplicação requisita ao HNS a quantidade de entes do tipo *holder* disponíveis no ambiente. Baseada nesta informação, a aplicação divide a tarefa a ser executada e para cada parte da tarefa um ente é instanciado e move-se para um dos entes *holder* do ambiente.

O código localizado entre as duas instruções *move* na linha 4 é executado em uma máquina remota. Ao ser clonado, o ente descrito na Figura 7 recebe como argumento o nome de um ente do tipo *holder* para onde ele deve se mover juntamente com o pedaço da matriz a ser processada. O primeiro comando executado por este ente (linha 3) é um comando move que efetiva a mobilidade de código. Depois de aplicar o filtro na máquina remota este ente volta para a máquina de origem através de outra chamada ao comando move (linha 5).

```
1  Task() {
2      Task(Part, Target_Being) {
3          move(self, Target_Being);
4          DoTask(Part);
5          move(self, launch);
6      }
7
8      DoTask(Part) {
9      ...
10     }
11 }
```

Figura 7. Trecho de Código Fonte da Aplicação Focada em Ganho de Desempenho.

Os testes foram realizados com quatro tamanhos diferentes de entradas: 500x500, 1000x1000, 2000x2000 e 4000x4000 respectivamente. Os resultados obtidos demonstram que a tendência manteve-se a mesma para todas as entradas. Executando a aplicação com apenas uma máquina disponível no ambiente e utilizando uma entrada de tamanho 1000x1000, o tempo médio de execução foi de 19,72 segundos. Na medida em que mais máquinas entraram no ambiente e anunciaram seus entes *holder* para o HNS, a aplicação percebe esta mudança e divide a tarefa entre estas máquinas. Sendo assim, executando a aplicação com duas, três e quatro máquinas no ambiente os tempos médios obtidos foram 8,94, 6,50 e 5,70 segundos respectivamente.

A Figura 8 demonstra graficamente os resultados apresentados na Tabela 3, obtidos através dos experimentos realizados em um ambiente de testes. Foram utilizados para realizar os experimentos três computadores Athlon XP e um computador Pentium IV. Todos com 512 MB de memória RAM e sistema operacional Windows XP. Para o gerenciamento da aplicação, foi utilizado um computador Core 2 Duo com 1 GB de memória RAM e sistema operacional Windows XP. Cada um dos experimentos foi executado dez vezes.

A latência da rede e o tamanho de um ente serializado são fatores que influenciam no tempo total de execução. Neste experimento, entretanto, o foco principal era a funcionalidade do modelo de mobilidade de código desenvolvido. Neste sentido, HoloGo mostrou-se uma alternativa viável para a redução do tempo total de execução de uma

Tabela 3. Tempo de Execução x Número de Máquinas

	1		2		3		4	
	Média	D. P.	Média	D. P.	Média	D. P.	Média	D. P.
500x500	10,26	0,93	05,62	0,25	03,39	0,09	01,77	0,04
1000x1000	19,72	0,55	09,72	0,51	06,51	0,31	05,71	0,37
2000x2000	41,37	0,36	19,91	0,25	12,63	0,29	09,41	0,32
4000x4000	82,55	0,65	41,62	0,33	24,71	0,22	20,22	0,24

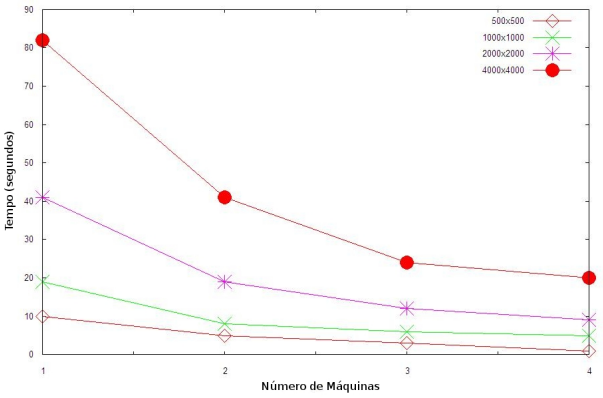


Figura 8. Tempo de Execução x Número de Máquinas.

tarefa computacionalmente intensa, uma vez que permitiu o aproveitamento dos recursos disponíveis no ambiente.

6 Trabalhos Relacionados

Algumas linguagens de programação, como Java [10] e C Sharp [6], provêem mobilidade de código como uma de suas funcionalidades. Nesses casos, a característica é obtida através de um mecanismo chamado de carregamento dinâmico de classe e torna a execução da mobilidade de código mais fácil.

Para prover mobilidade forte de código, o sistema JavaGo [15] usa um pré-processador que modifica o código fonte gerado em tempo de compilação. A modificação permite que JavaGo capture o estado de execução de um programa antes de enviá-lo a um outro ambiente computacional. Além disso, permite que o programador escolha quais as partes da pilha de execução que serão conservadas. JavaGoX [14] é baseado no JavaGo e oferece a mobilidade forte de código através da modificação de bytecode. Holo*Go* difere de JavaGo e de JavaGoX principalmente porque não há necessidade de modificação do código fonte ou do *bytecode*.

Desenvolvido pela IBM, Java Aglets [11] é outro sistema de mobilidade de código baseado em Java. As unidades de execução nesse sistema são *threads* Java. A mobilidade provida por Java Aglets é fraca. Quando uma *thread* é movida de um ambiente computacional para outro, seu estado de execução é perdido.

Entre os sistemas apresentados, JavaGo e JavaGoX são os que provêem mobilidade forte de código como o Holo*Go*. JavaGo e JavaGoX pertencem a uma classe de *Mobile Code Systems* que fazem transformações no *bytecode* ou modificações na JVM a fim de fornecer mobilidade forte de código. As técnicas de transformação de *bytecode*, mesmo baseadas somente em Java (e assim realmente portáveis) não fornecem uma completa gerência de *threads* e sofrem de problemas relacionados ao desempenho [5]. As modificações na JVM, por sua vez, podem introduzir problemas de confiança e erros de segurança. Esses são principalmente relacionados a *threads* [5].

Estendendo a comparação, JavaGo não fornece a mobilidade de entidades compostas. Holo*Go*, por sua vez, suporta a mobilidade de entidades compostas (entes compostos). JavaGo requer conhecimento da localização exata do ambiente computacional de destino de antemão. No Holo*Go*, esta informação é encapsulada na entidade ente. Assim, nenhum dado específico da rede, como o número da porta, precisa ser fornecido a fim de executar a mobilidade de código.

7 Considerações Finais

Este artigo apresentou o Holo*Go*, um modelo que implementa mobilidade forte de código no Holoparadigma. Holo*Go* pode ser visto como uma camada de software acoplada à HoloVM que implementa funcionalidades de mobilidade forte de código sem a necessidade de modificação na Hololinguagem.

Os resultados experimentais validaram o uso de Holo*Go*. Uma das características principais do modelo é a facilidade do desenvolvimento de aplicações móveis. A mobilidade forte de código ocorre de forma transparente com a instrução *move*. Todas as etapas necessárias para mover o ente e seu estado interno de execução são providas automaticamente, livrando o desenvolvedor do software dessa preocupação.

Holo*Go* demonstrou ser um modelo adequado para a exploração de mobilidade forte de código em aplicações reais. Neste artigo, isto pode ser verificado na aplicação apresentada: ganho de desempenho. Com a utilização da HoloVM, todo o dispositivo que a suporta pode beneficiar-se

das funcionalidades disponibilizadas pelo Holo*Go*.

Testes de desempenho em diferentes plataformas estão entre os trabalhos futuros relacionados a questões de escalabilidade. Pretende-se, ainda, propor uma solução para a mobilidade de código de entes que estejam aguardando retorno de ações. No protótipo atual, um ente nesta situação invariavelmente aguarda pelo retorno da ação, o que pode causar um atraso considerável na execução de uma mobilidade de código. Em versões futuras, um tratamento mais aprimorado para esta situação é pretendido. Além disso, pretende-se explorar outras aplicações de mobilidade de código.

Referências

[1] J. Barbosa, C. Costa, A. Yamin, and C. Geyer. A multiparadigm model oriented to development of grid systems. *International Journal of Grid Computing: Theory, Methods and Applications (Future Generation Computer Systems)*, 21(1):227–237, 2005.

[2] J. Barbosa and C. Geyer. Uma linguagem multiparadigma orientada ao desenvolvimento de software distribuído. In *Proceedings of the V Simpósio Brasileiro de Linguagens de Programação (SBLP)*, Curitiba, maio 2001.

[3] J. Barbosa, A. Yamin, P. Vargas, I. Augustin, and C. Geyer. Holoparadigm: a multiparadigm model oriented to development of distributed systems. In *International Conference on Parallel and Distributed Systems (ICPADS 2002)*, volume 8, pages 165–170. IEEE Press, 2002.

[4] D. T. Bonatto. *HNS: Uma Solução para Suporte à Execução Distribuída Considerando Aspectos da Pervasividade*. Dissertação de mestrado, Universidade do Vale do Rio dos Sinos, São Leopoldo, 2006.

[5] G. Cabri, L. Ferrari, L. Leonardi, and R. Quitadamo. Strong agent mobility for aglets based on the ibm jikesrvm. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 90–95, New York, NY, USA, 2006. ACM Press.

[6] M. Delamarco and G. P. Picco. Mobile code in .net: A porting experience. In *6th International Conference on Mobile Agents (MA 2002)*, pages 16–31. N. Suri ed. Lecture Notes on Computer Science vol. 2355., 2002.

[7] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[8] A. S. Garzão and J. Barbosa. A virtual machine with concurrency, mobility and blackboards support. In *XXIX Conferência Latinoamericana de Informática (CLEI)*, volume 24, La Paz, Bolívia, 2003. Universidad Mayor de San Andréas.

[9] C. Ghezzi and G. Vigna. Mobile code paradigms and technologies: A case study. In *Proceedings of the First International Workshop on Mobile Agents (MA'97)*, pages 39–49, Berlin, Germany, 1997. Springer Verlag.

[10] J. Gosling and J. G. Steele. *The Java Language Specification*. Addison Weseley, 1996.

[11] D. Lange and M. Oshima. *Programming and Deploying Java Agents with Aglets*. Addison Weseley, 1998.

[12] M. K. e. a. Naseen. Implementing Strong Code Mobility. *Information Technology Journal*, pages 188–191, 2004.

[13] OMG. *Common Object Request Broker Architecture: Core Specification*. Object Management Group, 2004.

[14] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in java. In *Agent Systems, Mobile Agents, and Applications*, pages 16–28, 2000.

[15] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension for java language for controllable transparent migration and its portable implementation. In *Coordination Models and Languages*, 1999.

[16] S. Vraned and M. Stanojevic. Integrating multiple paradigms within the blackboard framework. *IEEE Transactions on Software Engineering*, 21(3):244–262, 1995.