5
Monitoramento e Adaptação

A computação em grade se distingue de outros ramos da área de sistemas distribuídos pelo seu foco em compartilhamento de recursos em larga escala, aplicações inovadoras e, em alguns casos, orientação a alto desempenho. A computação em grade pode ser definida como um compartilhamento de recursos de forma flexível, segura e coordenada entre grupos dinâmicos de indivíduos, instituições e recursos, chamados de organizações virtuais (VO, *virtual organizations*) [11]. Nesse cenário, vários dos desafios de sistemas distribuídos, tais como autenticação, autorização, descoberta e acesso a recursos são majorados.

Além disso, vários requisitos de aplicações distribuídas estão relacionados a propriedades do sistema que evoluem ao longo do tempo. Por isso, gerenciar requisitos não-funcionais tipicamente requer o monitoramento do ambiente de execução. Um desafio nesta área é como projetar facilidades de monitoramento que sejam flexíveis o suficiente para serem usadas em diferentes aplicações, e como fazer a interface entre essas facilidades e as aplicações clientes. Especificamente, programas auto-adaptativos devem ser capazes de detectar e reagir a mudanças no cumprimento desses requisitos.

No contexto de computação em grade, uma ferramenta como o ALua torna-se bastante importante, devido ao fato da grade possuir uma configuração dinâmica em sua essência. O modelo do ALua pode ser usado em grades para diferentes tarefas e acreditamos que ele pode ser bastante eficiente para o monitoramento e gerenciamento da própria grade.

Nos capítulos anteriores, apresentamos o modelo do ALua e mostramos como ele pode ser usado em aplicações paralelas distribuídas para configuração e comunicação de dados e de controle. O ALua é particularmente interessante para aplicações de longa duração, onde a possibilidade de interação com o usuário final nos permite monitorar a aplicação, inspecionando seu estado de execução (variáveis, etc), e adaptá-la, alterando seu comportamento (redefinindo funções, etc).

Neste capítulo, apresentamos um mecanismo de monitoramento e políticas de adaptação mais automáticas para as aplicações. Em seguida, mostramos como esses mecanismos podem ser aplicados a aplicações paralelas para grades, mais especificamente para ambientes de grade que utilizam o Globus [12]. No final do capítulo, mostramos como integrar esse mecanismo a aplicações já existentes, usando o ALua em conjunto com outras bibliotecas de comunicação, em particular o MPI [54, 5].

5.1 ALuaMonitor

Em [55], foi feito um estudo sobre reconfiguração dinâmica de aplicações distribuídas em ambiente CORBA [56]. Nesse trabalho, foi construída uma infra-estrutura baseada em Lua e CORBA, usando o sistema LuaOrb [57], que permitia que aplicações selecionassem dinamicamente os componentes que melhor atendessem seus requisitos, verificassem se o sistema continuava a satisfazer esses requisitos ao longo de sua execução e reagissem, quando apropriado, a variações das propriedades não funcionais dos serviços em uso. Um dos componentes principais dessa infra-estrutura é um mecanismo de monitoramento de requisitos definidos dinamicamente através de Lua. Também usamos a linguagem Lua para especificar estratégias de adaptação e um mecanismo de *proxy* inteligente para aplicar essas estratégias.

Inspirado nesse mecanismo de monitoramento, empregamos alguns resultados de [55] ao modelo do ALua [58]. Nesta seção, apresentamos o ALuaMonitor, um mecanismo de monitoramento extensível, que explora as características dinâmicas de Lua para dar suporte ao desenvolvimento de monitores.

Um monitor é um programa que de tempos em tempos verifica o estado de uma lista de propriedades definidas pelo usuário, que faz o papel de administrador da aplicação. O ALuaMonitor[1] oferece o método `defineProperty` que define um tipo especial de objeto — uma *propriedade* — que representa uma propriedade específica que se deseja observar. As propriedades observadas podem ser tanto propriedades do sistema, como o número de usuários correntes no sistema ou a porcentagem de CPU utilizada por uma determinada máquina, quanto propriedades da aplicação, como o número de clientes que um servidor está atendendo.

Um objeto `Property` implementa a funcionalidade de uma propriedade básica e oferece o método `getValue`, através do qual podemos obter o valor corrente da

[1] A API do ALuaMonitor é descrita no Apêndice B.

propriedade representada.

A Figura 5.1 mostra a definição da propriedade `NUsers`, no processo `Monitor`, que representa o número de usuários correntes em um sistema Linux. O primeiro parâmetro do método `defineProperty` é o nome da propriedade que será monitorada, o segundo é a função de atualização da propriedade e o terceiro determina a freqüência com a qual ela será atualizada (em segundos).

```
----------------------
-- Processo Monitor --
----------------------
prop = ALuaMonitor:defineProperty("NUsers", function(self)
   local nusers
   local unix_cmd = "who | cut -d ' ' -f1 | sort | uniq | wc -l"

   execute(unix_cmd .. " > /tmp/nusers")
   readfrom("/tmp/nusers")
   nusers = read("*n")
   readfrom()

   return nusers
end , 30)
```

Figura 5.1: Definição da propriedade `NUsers`.

Uma chamada ao método `getValue` do objeto `prop` (`prop:getValue()`) nos fornece o o número de usuários correntes no sistema.

Um usuário, executando um processo `C` que faça uso de um console, pode obter o valor da propriedade `NUsers` a qualquer momento, através do comando

```
msg = [[ alua.send("C", "nusers = "..prop:getValue()) ]]
alua.send("Monitor", msg)
```

Para evitar que as aplicações fiquem testando os valores das propriedades continuamente, o mecanismo do `ALuaMonitor` permite que as aplicações se registrem como observadoras (através da chamada ao método `attachEventObserver` do ALuaMonitor), especificando os eventos nos quais elas estão interessadas. Um objeto observador observa a ocorrência de um determinado evento, baseado no valor de uma ou mais propriedades. Desta forma, aplicações podem ser notificadas apenas quando mudanças específicas no estado ocorrerem, ao invés de em cada modificação no valor da propriedade.

Na implementação de monitores, um mecanismo interno de tempo dá suporte à geração de notificações. Ele dispara atualizações dos valores das propriedades e

ativa o mecanismo para detecção de eventos. A transferência da detecção de eventos para monitores permite a redução no número de interações entre os monitores e as aplicações.

Na Figura 5.2, o processo `A` cria um observador `observeUsers` no processo `Monitor`. Este observador notifica o processo `A` quando o evento `ManyUsers` ocorre, o que acontece quando o número de usuários é maior do que 3 (ver Figura 5.1). A função `greaterThanThree` será executada no processo `A` quando este evento ocorrer e poderá adaptar a aplicação conforme suas necessidades.

```
----------------
-- Processo A --
----------------
monitor_msg = [[
   notifyEventFunc = function(self, event_name)
      alua.send("A", "greaterThanThree()")
   end

   observeUsers = {notifyEvent = notifyEventFunc}
   ALuaMonitor:attachEventObserver(observeUsers, "ManyUsers", "$NUsers > 3")
]]
alua.send("Monitor", monitor_msg)
```

Figura 5.2: Adição de um observador.

Mais uma vez, tiramos proveito da natureza interpretada de Lua para definir novos tipos de eventos através de strings de código Lua, evitando a necessidade de definir-se previamente os tipos de eventos que podem ser observados.

O método `attachEventObserver`, de `ALuaMonitor`, recebe como parâmetros um objeto observador, que deve conter o método `notifyEvent`, o nome do evento que será observado e uma string com a expressão booleana que determina a ocorrência do evento em questão. Os valores das propriedades usados na expressão booleana devem ser descritos como `$<nome da propriedade>` e serão traduzidos para `<propriedade>:getValue()`. Um observador não está amarrado a uma única propriedade e pode fazer uso de um conjunto delas para determinar um evento que se deseja observar. Para ser notificado quando a propriedade `X` for maior que 5 e a propriedade `Y` for igual a 2, basta criar um observador que espera pelo evento `"$X > 5 and $Y == 2"`.

Em muitos casos uma aplicação pode estar interessada não apenas no valor específico de uma propriedade, mas também em estatísticas ou perfis da evolução de alguma condição. Para acomodar estes interesses, o ALuaMonitor oferece facilidades

para definir e obter aspectos de uma propriedade. Através do método `defineAspect`, do objeto `Property`, podemos definir um aspecto.

Na Figura 5.3 definimos o aspecto `greater3` para a propriedade `prop`, definida na Figura 5.1.

```
aspectFunc = function(self, currval, property)
   local resp = nil
   if currval > 3 then
      resp = 1
   end
   return resp
end

prop:defineAspect("greater3", aspectFunc)
```

Figura 5.3: Definição de um aspecto.

A seguir, mostramos outra forma de escrever a expressão booleana que define o evento do observador da Figura 5.2, usando o aspecto `greater3`:

```
ALuaMonitor:attachEventObserver(observeUsers, "ManyUsers", "$NUsers:greater3")
```

Os aspectos usados na expressão booleana devem ser descritos como `$<nome da propriedade>:<nome do aspecto>` e serão traduzidos para `<propriedade>:getAspectValue(<nome do aspecto>)`.

Com este mecanismo podemos construir uma aplicação de gerência independente da aplicação principal que se está executando, o que nos permite reaproveitar as estruturas de monitoramento definidas em um projeto para outro. Para permitir que se faça adaptação, as aplicações podem ser modificadas ou estendidas para possibilitar uma instrumentação com o ALua.

Enquanto o modelo orientado a eventos utilizado no ALua permite que um código, do qual a aplicação não tinha conhecimento, seja executado, inserindo-se facilmente mecanismos de monitoramento não previstos originalmente pela aplicação, no trabalho [55], no qual o ALuaMonitor foi baseado, o mecanismo de interfaces rígidas de CORBA torna muito mais complicada a possibilidade de se integrar um mecanismo de monitoramento em aplicações que não o previam em sua implementação.

5.2
Monitoramento e Adaptação de Aplicações Paralelas em Grades

Uma das tecnologias mais populares para grade é o *toolkit* Globus [12, 59]. O Globus é composto de um conjunto de serviços e bibliotecas de software com arquitetura e código abertos, que dão suporte a grades e a aplicações para grade. O *toolkit* aborda questões como segurança, descoberta de informações, gerenciamento de recursos, gerenciamento de dados, comunicação, detecção de falhas e portabilidade.

Os componentes mais relevantes desse *toolkit* são:

- o protocolo GRAM (*Grid Resource Allocation Management*) [60], para alocação e gerenciamento de recursos para grades;
- o serviço de *gatekeeper*, que oferece a criação e gerenciamento de recursos confiáveis e seguros;
- o MDS-2 [61], um meta serviço de diretórios que captura e torna disponível informações sobre o sistema, modelagem de dados e um registro local (GRAM reporter); e
- o GSI, infraestrutura de segurança para grades, que oferece um sistema simplificado de autenticação, delegação restrita e mapeamento de credenciais.

Estes componentes oferecem os elementos essenciais de uma arquitetura orientada a serviços. Uma característica importante dessa infra-estrutura é que seus serviços são independentes uns dos outros, o que nos permite selecionar apenas os mais relevantes para a grade que se deseja construir.

Além das questões puramente técnicas, existem ainda questões políticas, sociológicas e econômicas para a implementação de grades computacionais, que incluem o encorajamento da participação de centros ricos em recursos. Dentre as principais tecnologias para grades, os mecanismos oferecidos pelo *toolkit* do Globus se destacam por serem usados em centenas de locais por dúzias de grandes projetos de grade espalhados por todo o mundo.

Neste trabalho, utilizamos a infra-estrutura do *Grid-Rio* [62], uma grade computacional baseada no Globus. O Grid-Rio foi estabelecido em janeiro de 2003 e atualmente conta com a participação do LabPar, Laboratório de Paralelismo do Departamento de Informática da PUC-Rio, e do Instituto de Computação da Universidade Federal Fluminense (UFF). Existe a previsão de integração de pelo menos mais três centros de pesquisa do Rio de Janeiro: o LNCC, Laboratório

Nacional de Computação Científica, o CBPF, Centro Brasileiro de Pesquisas Físicas e a COPPE-Sistemas da Universidade Federal do Rio de Janeiro (UFRJ). A UFF é a instituição responsável pelo gerenciamento dessa grade, realizando tarefas como emissão de certificados para máquinas e usuários, suporte para instalação do Globus e até mesmo construindo um site sobre projetos de grade em todo o Brasil.

Ao mesmo tempo em que o Globus é uma tecnologia popular para grades pela quantidade e independência dos serviços que ele oferece, utilizar tantos serviços e bibliotecas diferentes torna-se uma tarefa bastante complexa. Para utilizar as facilidades oferecidas pelo ALuaMonitor no contexto de grades e facilitar a programação com uso do Globus, integramos o ALua com mecanismos de comunicação e gerenciamento de recursos existentes para o Globus [63]. Cada serviço foi integrado separadamente, o que mantém a característica de independência dos serviços no ambiente do ALua, além de facilitar a própria integração. A maior dificuldade desta integração foi a precária documentação desses diversos serviços. Integramos o mecanismo de monitoramento descrito na Seção 5.1, com o serviço de diretórios do Globus (MDS) para obter informações sobre propriedades do ambiente de execução. Em trabalhos anteriores, os monitores eram tipicamente distribuídos pelas diversas máquinas utilizadas na execução da aplicação, porém neste ambiente de grades, um único monitor foi suficiente na maioria dos casos, uma vez que o MDS pode dar informações sobre as diversas máquinas que participam da aplicação e sobre as que podem vir a participar.

Na infra-estrutura de monitoramento e adaptação, estamos utilizando serviços tais como GRAM (alocação e gerenciamento de recursos para grades), GSI (infra-estrutura de segurança para grades) e MDS-2 (meta serviço de diretório), disponíveis no Globus *toolkit*. Com recursos como o MDS, podemos descobrir não só quais os nós disponíveis para a execução de uma aplicação, mas também suas características e seus recursos. A idéia é tornar esses recursos disponíveis para o programador ALua, de modo que ele possa usar a própria infra-estrutura disponível na plataforma de grade de forma mais dinâmica e flexível, tanto de dentro das aplicações, quanto de um console de gerenciamento.

O MDS [61] é composto pelos serviços GRIS e GIIS, que são responsáveis pela atualização das informações dinâmicas do ambiente. Essas informações são armazenadas num serviço de diretórios que usa o protocolo LDAP [64].

Para implementar uma infra-estrutura para monitoramento e adaptação dinâmica de aplicações, ligamos a biblioteca de Lua com os componentes do GRAM e do MDS (LDAP). Com o ALuaMonitor, podemos criar propriedades que usam o LDAP para obter informações a respeito do ambiente de grade. As *callbacks* re-

gistradas pelos observadores (`notifyEvent`) podem ser escritas em Lua ou em C e podem usar outras bibliotecas, como por exemplo a de MPI [5].

A Figura 5.4 mostra como definir uma propriedade que indica qual a porcentagem de CPU livre nos últimos 1, 5 e 15 minutos. Note que esta informação é fornecida pelo MDS na função que atualiza o valor da propriedade `lookupF`.

```
ldap_obj = LDAP:new{server = "server.par.inf.puc-rio.br"}

lookupF = function(self) -- Retorna o valor corrente da propriedade,
                         -- que neste caso é uma tabela com as três
                         -- informações desejadas.

   local DN = "Mds-Device-Group-name=processors, "..
              "Mds-Host-hn=%s, "..
              "Mds-Vo-name=GIIS-labpar.inf.puc-rio.br, o=grid"

   ldap_obj.filter="(objectClass=MdsCpuFree)"
   ldap_obj.base  =format(DN, alua.myhost)

   entries = ldap_obj:search{"Mds-Cpu-Free-1minX100",
                             "Mds-Cpu-Free-5minX100",
                             "Mds-Cpu-Free-15minX100"}

   -- Neste exemplo, o valor da propriedade será o valor os
   -- atributos Mds-Cpu-Free-*, retornados na primeira entrada.

   local currval = {entries[1]["Mds-Cpu-Free-1minX100"],
                    entries[1]["Mds-Cpu-Free-5minX100"],
                    entries[1]["Mds-Cpu-Free-15minX100"]}
   return currval
end

-- O valor desta propriedade será atualizado a cada 30 segundos!
propCPU = ALuaMonitor:defineProperty("CPU", lookupF, 30)
```

Figura 5.4: Propriedade que indica a porcentagem de CPU livre.

Para cada propriedade podemos definir diversos aspectos. Na Figura 5.5, mostramos como definir o aspecto *Decreasing* da propriedade *CPU* criada anteriormente. Este aspecto indica se a quantidade de CPU livre está diminuindo ao longo do tempo. Neste exemplo, consideramos que esta quantidade diminui se nos últimos 14 minutos houver uma queda de 10% de CPU livre.

Para obter o valor do aspecto `Decreasing` desta propriedade podemos usar o método `propCPU:getAspectValue("Decreasing")`.

Além de poder definir propriedades e aspectos, podemos criar observadores para as diversas propriedades e seus aspectos. Na Figura 5.6 criamos um observador

```
aspectF = function(self, currval, property)
   local cons = currval[3] - currval[1]  -- (15 min - 1 min)
   if cons > 10 then
      return 1
   else
      return nil
   end
end

propCPU:defineAspect("Decreasing", aspectF)
```

Figura 5.5: Aspecto que indica se o uso da CPU está mais intenso.

baseado na propriedade `CPU` e em seu aspecto `Decreasing`. A função de *callback* do observador (chamada `notifyEvent`) será chamada caso a quantidade de CPU livre no último minuto seja inferior a 75% e tenha diminuído mais de 10% nos últimos 14 minutos.

```
observer = { notifyEvent=function(self, event)
                <<código para adaptar a aplicação>>
                alua.send(<<targetProcess>>, <<adaptingCode>>)
             end}

ALuaMonitor:attachEventObserver(observer, "CPUDecrease",
           "$CPU[1] < 75 and $CPU:Decreasing")
```

Figura 5.6: Observador baseado na propriedade `CPU` e em seu aspecto `Decreasing`.

Através de um console e com a capacidade de poder enviar um código que altere o comportamento de um nó da grade, um operador pode adaptar uma aplicação dinamicamente, sem precisar previamente definir qual será esta adaptação. Uma das facilidades que o ALua oferece é permitir a depuração da aplicação sem que ela tenha sido preparada para isto. É possível enviar código não só para alterar o comportamento de um nó, mas para inspecionar suas variáveis.

Considere o exemplo de uma aplicação mestre/escravo onde o processo mestre possui um determinado número de tarefas a executar (`MAX`). O código da implementação dessa aplicação em ALua é mostrado na Figura 5.7.

Inicialmente, o processo mestre dispara `N` processos escravos e envia `AMNT` tarefas para cada um deles. A medida em que cada processo escravo termina suas tarefas o mestre envia mais `AMNT` tarefas, até que não haja mais tarefas para serem executadas.

```
MAX = 2000  -- Número de tarefas
N   = 3     -- Número de escravos
AMNT= 50    -- Quantidade de tarefas por ciclo

ntasks = 0

function nextTask(slave)
   local first = ntasks + 1
   local last  = ntasks + AMNT
   if last > MAX then last = MAX end
   if first <= MAX then
      local msg = format("doTasks(%d, %d)", first, last)
      alua.send(slave, msg)
   else
      alua.exit_all()
      alua.exit()
   end
   ntasks = last
end

slave_code =
[[function doTasks(i, j)
     for k=i,j do
       print("Doing task "..k)
     end
     alua.send("Master", format("nextTask('%s')", alua.myname))
  end
]]

do
  alua.spawn(N, function (slaves)  -- dispara N processos escravos.
     SLAVES = slaves
     print("Slaves are OK.")

     alua.mcast( slaves, slave_code)
     for i,v in slaves do
        nextTask(v)
     end
  end)
  print("Server is ready.")
end
```

Figura 5.7: Aplicação mestre/escravo.

Através de um console, é possível enviar para o processo mestre, a qualquer momento, a mensagem '`AMNT = 10`' e alterar o número de tarefas enviado para cada escravo.

Para fazer uma adaptação de forma mais automática, usamos o exemplo de monitores mostrado nesta seção. Criamos um processo monitor na mesma máquina que o processo mestre conforme apresentado na Figura 5.8.

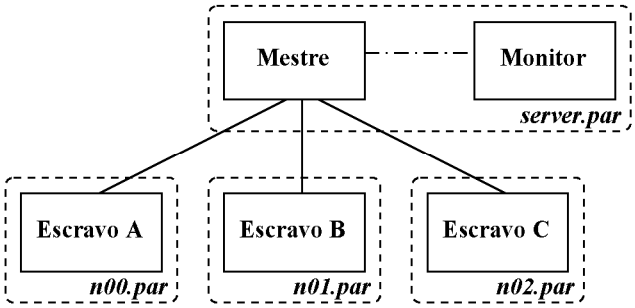

Figura 5.8: A estrutura da aplicação mestre/escravo.

Com o observador `CPUDecrease`, podemos aumentar a quantidade de tarefas por rodada enviada para cada escravo, de modo que o processo mestre fique menos sobrecarregado com pedidos de novas tarefas quando o consumo de CPU estiver aumentando. Neste exemplo, como existe um observador para cada escravo, o mestre pode funcionar como um escalonador dinâmico. O objeto `observer` passa a ser:

```
observer = { notifyEvent=function(self, event)
               alua.send("Master", "N = 100")
             end}
```

As propriedades e aspectos que mostramos nesta seção são independentes de aplicações, pois são propriedades do ambiente de execução. Por isso, podemos registrar simultaneamente observadores de diferentes aplicações, especializando o método `notifyEvent` de cada um deles para atender as necessidades de adaptação de cada aplicação.

Em particular, aplicações de longa duração, como a de times de agentes assíncronos discutida na Seção 3.4.3, podem tirar grande proveito desse mecanismo.

5.3 Uso do ALua com outras Bibliotecas de Comunicação

Na seção anterior, mostramos como é possível integrar o mecanismo de monitoramento do ALuaMonitor com ambientes de grades. Para isso, mostramos uma

aplicação implementada em ALua, que sofria adaptação quando uma determinada mudança no ambiente de execução acontecia.

Nesta seção, mostramos como é possível conciliar a flexibilidade oferecida por esse mecanismo de monitoramento com aplicações já existentes, que usam outras bibliotecas de comunicação. Devido a popularidade da biblioteca de comunicação MPI, utilizamos nos exemplos dessa seção o MPICH-G2 [54], uma implementação de MPI para o Globus.

A seguir, vamos mostrar passo a passo como integrar o mecanismo do ALuaMonitor e do ALua com uma implementação MPI da aplicação mestre/escravo utilizada na seção anterior, mostrando também que o programador pode decidir o grau de integração entre o núcleo da aplicação e o ALua. Esse grau de integração vai determinar o grau de flexibilidade e as possibilidades de adaptação que podem ser alcançadas.

5.3.1
Aplicação mestre/escravo em MPI

Nesta seção, mostramos a implementação da aplicação mestre/escravo (Figura 5.8) em MPI e adicionamos a ela um processo "monitor", com uma funcionalidade limitada. Primeiramente desenvolvemos uma implementação MPI similar ao exemplo da Figura 5.7, e a testamos no Grid-Rio. A Figura 5.9 mostra o *loop* principal do processo mestre. Seu código completo pode ser visto no Apêndice C.2.1.

Alteramos então este *loop* principal para que o processo mestre pudesse receber uma mensagem (MPI) de adaptação e, como conseqüência, alterar o valor de `amount`, variável que indica o número de tarefas enviadas para cada escravo. Após receber uma mensagem, o processo mestre verifica se ela é uma mensagem de adaptação:

```
if (status.MPI_TAG == ADAPT)
   amount = received_num;
else
{
   slave = received_num;
   nextTask(slave);
}
```

Em seguida, adicionamos à aplicação um processo "monitor", implementado em MPI, que apenas envia uma mensagem de adaptação para o mestre:

```
int new_amount = 100;
```

```
int i;
int received_num;
int slave;
for (i=1; i < numprocs; i++)
   nextTask(i);
while (ntasks < MAX)
{
   MPI_Recv(&received_num,    /* buff   */
            1,                /* count  */
            MPI_INT,          /* type   */
            MPI_ANY_SOURCE,   /* source */
            MPI_ANY_TAG,      /* tag    */
            MPI_COMM_WORLD,   /* comm   */
            &status);         /* status */
   slave=received_num;
   nextTask(slave);
}
```

Figura 5.9: *Loop* principal do processo mestre.

```
MPI_Send(&new_amount,     /* buff  */
         1,               /* count */
         MPI_INT,         /* type  */
         MASTER,          /* dest  */
         ADAPT,           /* tag   */
         MPI_COMM_WORLD); /* comm  */
```

A seguir, mostramos como transformar este monitor MPI em um programa que utiliza a infra-estrutura de monitores oferecida pelo ALuaMonitor.

5.3.2

Aplicação mestre/escravo em MPI com monitor ALuaMonitor

Nesta seção, o processo monitor passa a ser um processo ALua + MPI, de modo que ele possa utilizar o mecanismo de monitoramento do ALuaMonitor para disparar uma mensagem (MPI) de adaptação para o processo mestre. Nos exemplos mostrados nesta seção, os processos mestre/escravo permanecem inalterados.

Para que o processo monitor possa usar a infra-estrutura do ALuaMonitor e enviar mensagens de adaptação (MPI) para o processo mestre, exportamos para o ambiente Lua do monitor a função `MPI_Send`, que envia mensagens MPI. Implementada em C, essa função pode ser chamada por código Lua.

Neste exemplo, o processo monitor não prevê o recebimento de mensagens

MPI[2], mas faz as chamadas MPI de inicialização e finalização para que possa fazer parte da aplicação, e portanto enviar mensagens MPI para os demais processos.

Utilizando as propriedades e aspectos apresentados na Seção 5.2, podemos definir um observador, no processo monitor, cujo o método `notifyEvent` é:

```
notifyEvent=function(self, event)
   local amount = getNewAmount()
   local MASTER = 0
   MPI_Send(MASTER, amount)
   return 1 -- não remove o observador!
end
```

Neste caso, conseguimos tirar proveito do mecanismo de monitoramento para disparar a adaptação da aplicação, mas o tipo de adaptação que pode ser feita está amarrado ao código da aplicação MPI. Como dissemos antes, quanto maior a integração entre Lua e a aplicação principal, maiores as possibilidades de adaptação que podem ser alcançadas. Quando implementamos esta aplicação exclusivamente com o ALua, as possibilidades de adaptação são praticamente ilimitadas.

Agora, vamos mostrar como os processos desta aplicação podem receber tanto mensagens MPI quanto mensagens ALua.

5.3.3

Aplicação mestre/escravo em ALua + MPI com monitor ALuaMonitor

Em geral, um processo ALua fica em um *loop* esperando eventos de rede ou de interface com o usuário:

```
while (1) do
  alua.event_manager()
end
```

A chamada à função `alua.event_manager()` faz com que o processo fique bloqueado até que um evento seja recebido e processado. Esta função também pode receber um parâmetro indicando um *timeout* em segundos. Neste caso, se não ocorrer nenhum evento dentro do tempo especificado, ela retorna.

Processos ALuaMonitor, além de esperar eventos de rede ou de interface, também devem verificar o estado de uma lista de propriedades definidas pelo usuário.

[2] A primitiva `MPI_Recv` é bloqueante, o que não se encaixa bem com o modelo do ALua. Na próxima seção, vamos mostrar como processos podem receber tanto mensagens MPI quanto mensagens ALua.

Nesta seção, mudamos a estrutura dos processos mestre/escravo de modo que eles passassem a ter um *loop* de eventos que recebe mensagens ALua e MPI. Uma mensagem MPI dispara a parte do código original responsável pelo passo do *loop* principal, enquanto uma mensagem ALua pode desempenhar tarefas de instrumentação e adaptação.

Para que processos MPI pudessem se beneficiar do mecanismo do ALua, criamos a biblioteca *aluampi*, com funções Lua e C. A função `alua.poll` (também disponível em C), permite que o programa MPI chame o *loop* de eventos do ALua. A estrutura típica de uma aplicação ALua + MPI pode ser observada na Figura 5.10.

```
#include <mpi.h>
#include <aluampi.h>

int main(int argc, char **argv)
{
    lua_State* L;
    int numprocs, my_id;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

    L = alua_mpiopen(argc, argv, my_id, numprocs);

    // Corpo da Aplicação
    // incluindo a chamada ao loop de eventos
    // No exemplo, o timeout é de 5 segundos.
    while (alua_poll(L, 5)) { ... }

    MPI_Finalize();

    return 0;
} /* end main() */
```

Figura 5.10: Aplicação típica ALua + MPI.

Na aplicação mestre/escravo, alteramos o *loop* principal do processo mestre mostrado na Figura 5.9, para que ele também pudesse receber e enviar mensagens ALua. O processo mestre passou a ser um processo híbrido, ALua + MPI, pois ele fica num *loop* de eventos esperando uma mensagem MPI ou ALua.

O *loop* principal do processo mestre é apresentado na Figura 5.11. A chamada a função `MPI_Recv` não deve ser bloqueante, para permitir a chegada de outros eventos. Por isso, usamos as funções `MPI_Recv_init` e `MPI_Start`, que indicam o interesse da aplicação em receber uma mensagem MPI específica, e a função `MPI_Test`, que indica o recebimento efetivo da mensagem especificada.

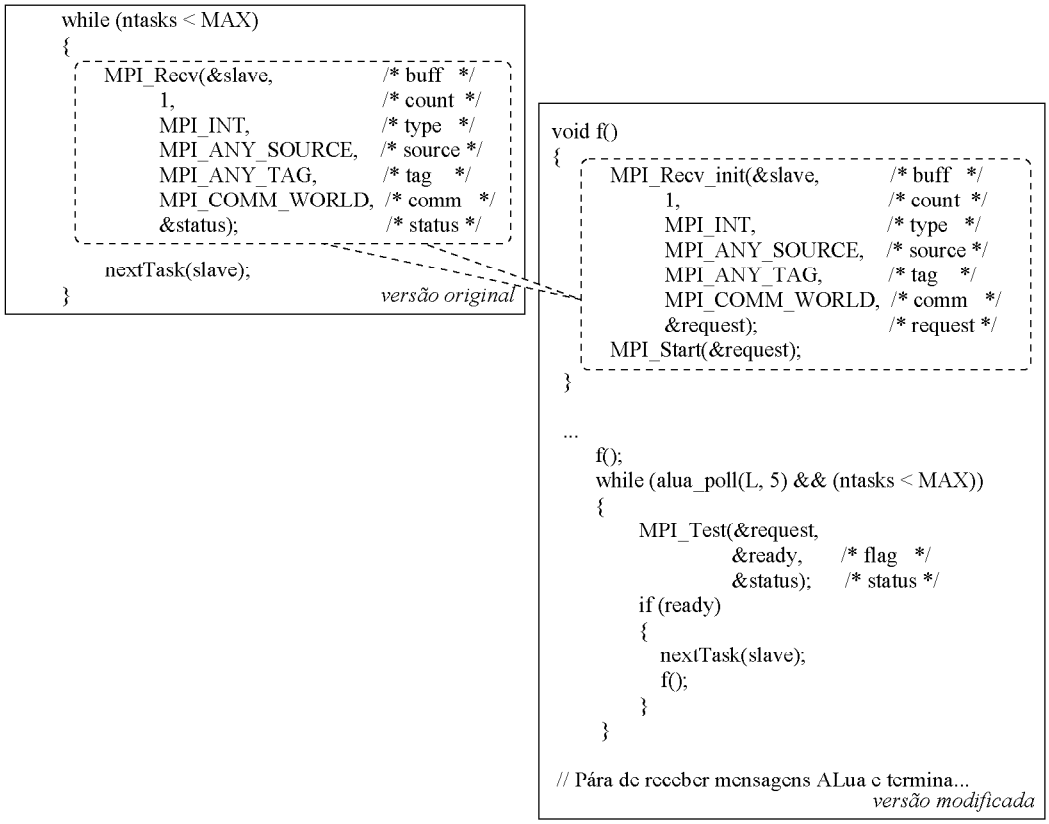

Figura 5.11: *Loop* principal do processo mestre ALua + MPI.

Esta nova estrutura do processo mestre não explora muito a flexibilidade que o ALua oferece. É preciso ainda aumentar a integração entre o código original e Lua. Por exemplo, podemos exportar para Lua as variáveis `ntasks` e `amount` para inspecionar o número de tarefas calculadas até o momento ou mudar o número de tarefas por rodada, através de mensagens de adaptação em ALua.

Existem ferramentas interessantes para este tipo de integração, como o *toLua* e a biblioteca *loadlib*. *toLua* [65] é uma ferramenta que simplifica bastante a integração de código C/C++ com Lua. Baseado num arquivo de protótipos de funções, o *toLua* gera automaticamente um *binding* que oferece acesso a recursos C/C++ a partir de Lua, mapeando constantes, variáveis externas, funções, classes e métodos para Lua. A biblioteca *loadlib* [66] carrega uma biblioteca dinamicamente e torna possível o acesso as suas funções através de Lua. Na aplicação A-Teams, discutida na Seção 3.4.3, podemos usar estas facilidades para alterar os algoritmos de refinamento usados pelos *agentes trabalhadores*.

Uma outra possibilidade de se estruturar uma aplicação ALua + MPI é usar um modelo de máquina de estados. O processo pode ficar no loop de eventos do ALua onde ele recebe mensagens que definem funções, variáveis, etc e o configuram

para entrar num estado "tarefa MPI", onde o processo pode ficar bloqueado numa primitiva `MPI_Recv` até receber os dados da tarefa a ser executada. Uma vez terminada a tarefa, o processo sai do estado "tarefa MPI" e volta ao loop de eventos do ALua.

No Globus, uma vez que uma aplicação foi disparada não há meios para adicionar ou remover processos a esta aplicação. Em um ambiente dinâmico como a grade, isto torna-se uma limitação. Para tirar proveito das características adaptativas do modelo do ALua para grades, desenvolvemos em [63] um mecanismo para simular a criação dinâmica de processos. Para uma aplicação MPI que precisa de n processos, este mecanismo irá escolher um número $m > n$ de processos. n processos serão processos MPI e os outros $m - n$ processos serão processos ALua. Na realidade, todos esses processos são híbridos (ALua + MPI). A idéia é que processos ALua não farão uso intensivo da CPU, mas estarão disponíveis como parte da aplicação caso sejam necessários.

5.4 Considerações Finais

Neste capítulo mostramos que aplicações que usam outras bibliotecas para comunicação de dados também podem se beneficiar do uso do ALua para envio de mensagens de monitoramento e de adaptação.

Apresentamos um mecanismo de monitores que possibilita o monitoramento de um conjunto de propriedades definidas pelo usuário. Aplicamos esse mecanismo ao domínio de grades, em particular a ambientes que usam a infra-estrutura Globus, e mostramos como integrar esse mecanismo a uma aplicação MPI.

Se por um lado o ALua naturalmente oferece um mecanismo de adaptação interativo para aplicações, através de consoles de comando, o mecanismo de monitores pode ser usado para adaptação mais automática de uma aplicação.

Com esse mecanismo, podemos implementar ferramentas de gerenciamento e monitoramento de grades customizáveis, que possam ser mais facilmente estendidas com recursos e procedimentos específicos para as aplicações que estão executando.

Novamente, chamamos atenção para o fato de que a integração do ALua com uma aplicação pode ser feita em diferentes níveis. É o desenvolvedor da aplicação quem deve decidir o grau de flexibilidade que a aplicação necessita.