3
ALua

No capítulo anterior falamos em um modelo de programação onde a aplicação é dividida em duas partes: a parte computacional, formada por processos envolvidos com a manipulação de dados, e a parte de coordenação, responsável pela comunicação e cooperação entre os processos. Este modelo pode ser um modelo dual de programação, onde usamos linguagens diferentes para desenvolver cada uma das partes da aplicação.

Para nós, coordenar uma aplicação vai além de trocar mensagens de dados entre os processos que a compõem ou dar uma forma final à aplicação, definindo sua configuração. Ao coordenar uma aplicação, podemos trocar trechos de código entre seus processos, de modo a adaptar a aplicação a mudanças em seu ambiente de execução. Além disso, é possível trocar mensagens não só entre os processos que compõem uma aplicação, mas entre eles e entidades externas à aplicação, que podem ser desde um programa que monitora o ambiente de execução e envia mensagens quando uma determinada condição ocorre no ambiente, até um usuário que envia comandos (mensagens com código) através de um console interativo. Dessa forma, acreditamos que pode haver uma interseção entre aspectos de configuração e coordenação.

3.1
O Modelo Básico do ALua

O projeto de uma linguagem de programação sempre envolve um equilíbrio entre desempenho e flexibilidade. Linguagens interpretadas, como Lua, Perl ou Tcl, são geralmente muito flexíveis, por exemplo, é bem fácil modificar uma aplicação sem ter que interrompê-la. Linguagens como C ou Fortran, por outro lado, se atêm ao desempenho. Elas têm um sistema de tipos mais estrito, precisam de informações

detalhadas sobre o uso da memória e são mais lentas para compilar e ligar, mas, elas podem ser mais que uma ordem de magnitude mais rápidas que uma linguagem interpretada. Usando ambos os tipos de linguagens em uma aplicação nos permite ter o melhor dos dois mundos, uma vez que podemos escolher o que deve ser mais rápido e o que deve ser mais flexível.

Neste capítulo, apresentamos o ALua [36, 15], um sistema para programação de aplicações paralelas em ambientes de memória distribuída, baseado em um modelo multilinguagem. ALua usa a linguagem de extensão Lua [16] para coordenar a interação entre componentes escritos em C. Assim como em outros ambientes distribuídos, uma aplicação em ALua é composta por um grupo de processos executados em vários computadores e que se comunicam através de uma rede. O código Lua trata de toda a comunicação entre os processos (e conseqüentemente define a arquitetura da aplicação), enquanto funções C tratam das tarefas de uso intensivo da CPU, em cada processo.

Uma outra característica importante de uma linguagem interpretada é o fornecimento de um mecanismo para execução de trechos de código criados dinamicamente. No ALua, mensagens são trechos de código que serão executados pelo processo receptor. Isto oferece um mecanismo de comunicação muito simples, no entanto muito poderoso. Existe apenas uma primitiva de comunicação, `send`, que envia um trecho de código para outro processo. Não existe o equivalente a uma primitiva `receive`. Ao invés disso, ALua usa um modelo de programação dirigido a eventos, onde a chegada de uma mensagem é tratada como um evento. Este mecanismo de comunicação é bastante flexível: Um programador pode usá-lo trivialmente para tarefas simples, como chamar uma função remota, mas ele também pode usá-lo para tarefas muito mais complexas, como trocar ou adaptar remotamente o algoritmo que um processo está executando. No contexto de aplicações paralelas de longa duração, e com a disponibilidade de um console interativo, esta é uma possibilidade poderosa, e permite ao programador redefinir dinamicamente o comportamento da aplicação.

O ALua utiliza um modelo *single-threaded*, dirigido a eventos, onde um programa é composto por processos, que são executados em uma ou mais máquinas e que se comunicam através de uma única operação assíncrona: *send*. Cada processo contém um interpretador Lua e um *loop* de eventos, que gerencia eventos de rede e de interface com o usuário. A Figura 3.1 mostra a estrutura de um processo. O *loop* de eventos recebe eventos (trechos de código Lua) continuamente e envia seu conteúdo (código Lua) para o interpretador Lua para que seja executado.

Uma importante característica do ALua é que ele trata cada mensagem como um bloco de código atômico. Por adotar um paradigma orientado a eventos, ele trata

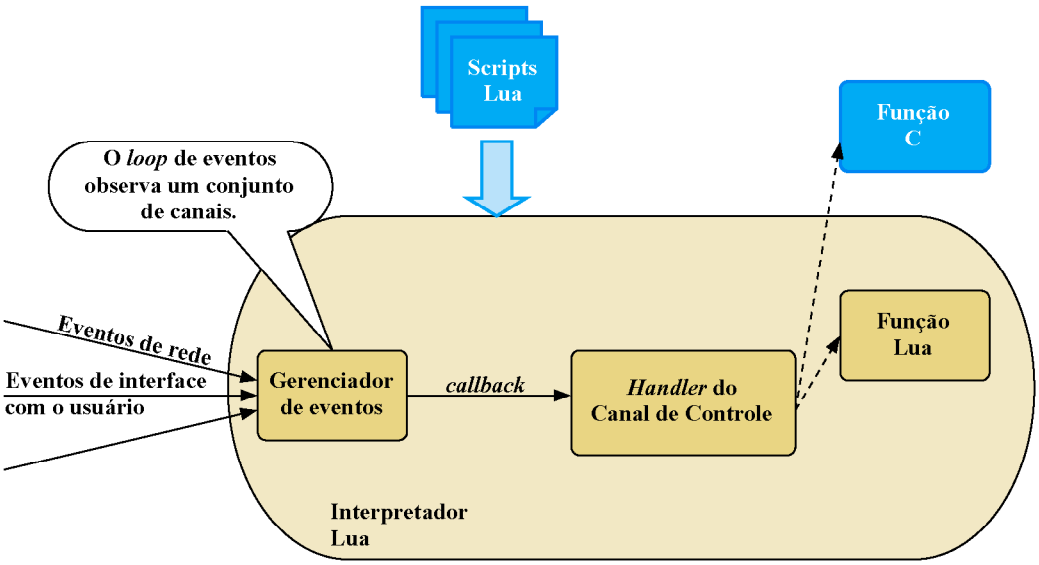

Figura 3.1: A estrutura de um processo ALua.

cada evento até seu término, antes de começar o tratamento do evento seguinte. Isto quer dizer que não existe concorrência interna em um processo ALua. O modelo de programação resultante é similar ao modelo assíncrono descrito em [38], com eventos disparando ações atômicas. Em nossa experiência, isso não mostrou ser uma limitação. O uso de um paradigma orientado a eventos, assim como qualquer outro paradigma de programação, leva o programador a criar estruturas de programação específicas. As mensagens devem tipicamente ser trechos de código pequenos e não bloqueantes. Se uma aplicação necessita de ações mais longas, um processo sempre pode recorrer a enviar uma mensagem para si mesmo, como meio de quebrar seu código em ações não atômicas, e portanto permitir que outras mensagens sejam recebidas entre essas ações. Por outro lado, a falta de concorrência simplifica bastante muitos aspectos de programação distribuída, uma vez que não há necessidade de sincronização dentro de um processo [40].

Eventos de rede correspondem a mensagens recebidas de outros processos. A operação de *send* envia um trecho de código para ser executado em outro processo. ALua não tem uma operação específica para receber uma mensagem. O *loop* de eventos do receptor irá executar automaticamente o código recebido. O resultado é um modelo de programação dirigido a eventos, compatível com o perfil de linguagens interpretadas: não muito seguras, porém altamente flexíveis. Com esse modelo é muito fácil implementar várias tarefas distribuídas típicas, como chamada remota de procedimentos, ou inspecionar e modificar variáveis remotas [15, 33].

Através de comandos simples é possível inspecionar variáveis, alterar o valor de variáveis, enviar mensagens para outros processos ou até mesmo executar um

programa.

3.2 Suporte de Execução

Apesar do ALua oferecer apenas uma função primitiva de comunicação (`send`), o sistema oferece outras funções, não só para atender a necessidade de criação e terminação de processos, mas também para facilitar a comunicação. A seguir mostramos uma lista das principais funções usadas neste texto e no Apêndice A listamos todas as funções disponíveis no ALua.

alua.spawn Dispara novos processos.

alua.send Envia código Lua para um processo.

alua.mcast Envia código Lua para múltiplos processos.

alua.tostring Converte um valor Lua em uma *string* que o representa, de modo que o valor possa ser inserido em uma mensagem para outro processo.

alua.exit Termina a execução do processo.

alua.exit_all Termina a execução de todos os processos disparados.

Antes de executar uma aplicação ALua, deve-se disparar um processo *daemon* em cada uma das máquinas que se deseja utilizar. Embora conceitualmente os processos troquem mensagens diretamente entre eles, os *daemons* ALua agem como intermediários nesta troca, assim como acontece em PVM [4]. A Figura 3.2 ilustra este modelo.

Quando um processo A1, na máquina A, envia uma mensagem para o processo B1, na máquina B, esta mensagem será enviada para o *daemon* ALua em execução na máquina A. Em seguida o *daemon* A enviará a mensagem para o *daemon* ALua na máquina B e finalmente este *daemon* enviará a mensagem para o processo B1. Os *daemons* comunicam-se entre si e com os processos ALua através de conexões TCP, via soquetes.

Na implementação do ALua, utilizamos a biblioteca LuaSocket [41], uma biblioteca que exporta soquetes Berkeley para Lua. A integração com a biblioteca LuaSocket permite que código Lua manipule soquetes diretamente, através de uma interface simplificada. A maior parte da implementação do ALua é escrita em Lua

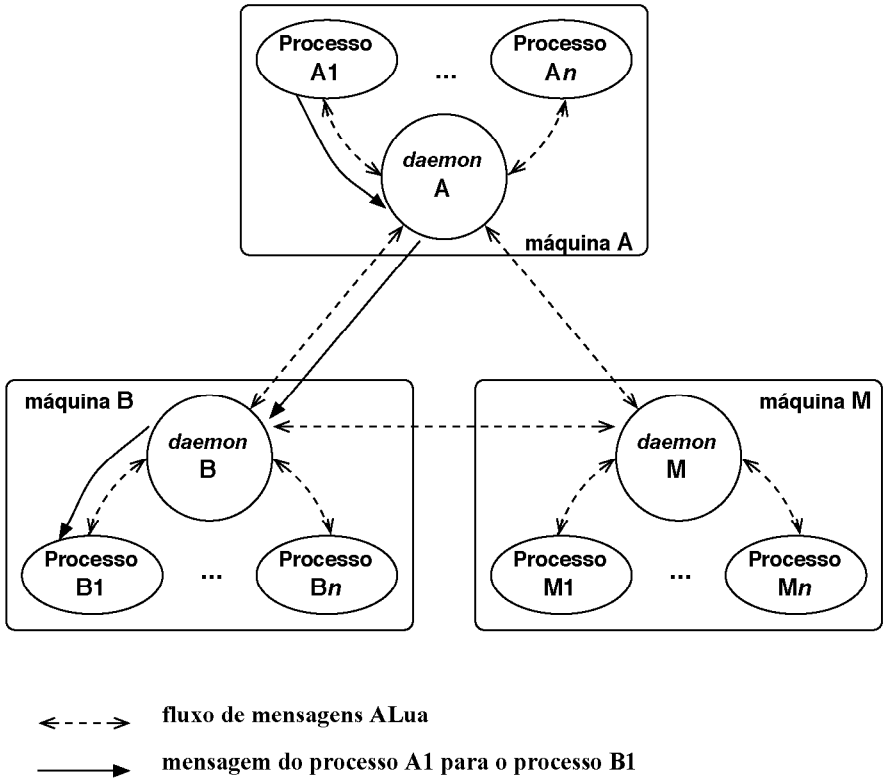

Figura 3.2: O modelo de comunicação do ALua.

com chamadas a LuaSocket, e o programador ALua também pode ter acesso direto à biblioteca de soquetes se assim ele desejar.

3.3 Exemplos de Programação

Tipicamente, quando executamos um programa ALua, criamos um processo "console", que simplesmente abre um console interativo para o usuário e executa qualquer comando digitado neste console. Através deste console, um usuário possui os meios para adicionar ou alterar dinamicamente a funcionalidade de outros processos ALua. Por exemplo, suponha que temos vários processos ALua executando um programa, e todos eles imprimem suas saídas através da função `print`. Agora o usuário decide redirecionar a saída de todos os processos para seu próprio console, anexando a cada mensagem sua origem. Tudo o que ele precisa fazer é digitar o seguinte código em seu console:

```
-- "[[" e "]]" são delimitadores de strings em Lua.
```

```
prog = [[
   print = function (msg)
      alua.send("A", format("print('%s: %s\n')", alua.myname, msg))
   end
]]
alua.mcast(allProcs, prog)
```

Em algum momento, cada processo receberá e executará o código `prog`, que redefine a função `print`. Esta nova função `print`, quando chamada, envia ao processo `A` (o qual executa o console) uma mensagem do tipo `print('nomeprocesso: msg')`.

O usuário não precisa antecipar esta mudança, e não precisa interromper o programa. A linguagem interpretada permite que ele altere o programa em execução, criando um novo código durante a execução do programa.

Nosso objetivo inicial era avaliar esse modelo em aplicações paralelas distribuídas. Queríamos investigar não só o poder de expressão do ALua, mas o tipo de flexibilidade que poderia ser obtida através desse modelo. Implementamos então algoritmos clássicos distribuídos como *probe/echo* e *heart beat* [42]. Implementamos também algoritmos para detecção de terminação e alguns paradigmas de programação paralela. Para avaliar a facilidade de desenvolvimento com ALua e estimar seu custo em termos de tempo de execução, reimplementamos em ALua duas aplicações paralelas, originalmente implementadas com linguagens compiladas tradicionais. Esperávamos uma perda significativa de desempenho, devido ao uso de uma linguagem interpretada, entretanto o uso combinado das linguagens Lua e C nos permitiu obter resultados inesperadamente bons [13].

Outra aplicação interessante que reimplementamos em ALua foi uma sobre Times de Agentes Assíncronos (A-Team) para o problema de recobrimento de conjuntos. Esta aplicação é composta por entidades autônomas, trabalhando cooperativamente, e memórias compartilhadas. Dependendo dos dados de entrada, esta aplicação pode ficar em execução por alguns dias, antes de atingir um resultado interessante, o que torna particularmente atraente a idéia de poder alterar o comportamento da aplicação durante sua execução. Esta aplicação mostra a flexibilidade que podemos ganhar por usar um modelo de comunicação orientado a eventos e que usa uma linguagem interpretada.

Nesta seção, vamos mostrar como implementar tarefas comuns através deste modelo. No nosso primeiro exemplo, um processo `A` apenas define um valor num processo `B`.

```
n = 50
```

```
-- envia a variável 'n' para o processo B.
msg = format("n = %d", n)
alua.send("B", msg)
```

A função `format` é similar a `sprintf` de C, mas ao invés de usar um espaço de memória fornecido, ela cria e retorna uma nova *string* (Lua possui alocação dinâmica de memória e coleta de lixo.). O resultado desta chamada a `format` é a *string* `"n = 50"`. A última linha do nosso exemplo envia esta *string* para o processo `B`, que irá, em algum momento, executar este trecho de código, atribuindo o valor 50 à sua própria variável global `n`.

Não é difícil enviar dados mais complexos com nosso mecanismo. Tudo o que precisamos é de uma forma de serializar os dados. ALua oferece a função `tostring` para isto:

```
x = {1, 4, 9, 16}    -- tabela
-- define o valor de 'x' na variável y do processo B.
msg = format("y=%s", alua.tostring(x))
alua.send("B", msg)
```

Tabelas em Lua são vetores associativos dinâmicos, elas são o mecanismo básico de estruturação de dados da linguagem. Nesse exemplo, primeiramente nós criamos uma tabela, que age como um vetor regular com índices numéricos, com os elementos 1, 4, 9 e 16. O valor final de `msg` neste exemplo é `"y={1, 4, 9, 16}"`. Quando `B` executa este trecho de código, ele cria uma tabela com os elementos fornecidos e a atribui a `y`.

No código a seguir, um processo `A` define a *string* `msg` que contém a definição da função `somatb`. Esta função calcula e retorna a soma dos elementos de uma tabela. Note que `A` conhece apenas a *string* `msg`, mas não conhece a função `somatb`. Esta *string* é então enviada para `B`, que ao receber a mensagem executará este código, definindo o valor da função em seu escopo global.

```
msg = [[ function somatb(t)  -- "[[" e "]]" são delimitadores de string
           local tot=0
           for i,v in t do
             tot = tot + v
           end
           return tot
         end
      ]]
alua.send("B", msg)      -- Envia a definição da função somatb
                         -- para o processo B.
```

Em Lua, funções são valores de primeira ordem, e nomes de funções são variável globais comuns, que contêm uma função. Também poderíamos ter escrito a mensagem anterior como

```
msg = [[ somatb = function (t)
                    ...
                  end
      ]]
```

Portanto, no nosso exemplo, `B` poderia ou não ter uma função `somatb` definida anteriormente em seu ambiente. Se a variável `somatb` já estiver definida em `B` quando a mensagem chegar, a mensagem irá redefini-la.

No ALua também é possível chamar uma função remotamente, mas ao contrário do estilo RPC, o processo que envia a mensagem não fica bloqueado enquanto espera a resposta. No próximo exemplo, usamos esta técnica para determinar a soma dos elementos da tabela `y` no processo `B`. Primeiro o processo `A` executa o código abaixo[1]:

```
msg = [[ alua.send("A", "print(" .. somatb(y) .. ")") ]]

alua.send("B", msg)
```

O processo `B` receberá a mensagem

```
alua.send("A", "print(" .. somatb(y) .. ")")
```

Em seguida, ele calcula os argumentos para a função `send`. Assumindo que a tabela `y` no processo `B` é `{10, 20, 40}`, o resultado de

```
"print(" .. somatb(y) .. ")"
```

será `"print(70)"`. Esta *string* é a mensagem enviada de volta ao processo `A`, que irá então executá-la e exibir o número 70.

O próximo exemplo ilustra outra técnica de programação útil em ALua, que vem de outras arquiteturas orientadas a eventos e consiste em estruturar um processo como uma máquina de estados. Para ilustrar esta técnica, suponha que o processo `A` precisa enviar uma mensagem a dois outros processos, `B` e `C`, e em seguida terminar sua execução quando receber uma confirmação de ambos os processos. Em um sistema de troca de mensagens convencional, poderíamos fazer algo como:

[1] O símbolo `..` representa a concatenação de *strings* em Lua.

```
-- Processo A
msg_recebidas = 0
msg=[[ alua.send("A", "msg_recebidas=msg_recebidas + 1") ]]
alua.mcast({"B", "C"}, msg)

-- loop vazio para esperar respostas
while msg_recebidas ~= 2 do  end

print("FIM.")
alua.exit()
```

Entretanto, em ALua este código iria bloquear a aplicação. Por causa da falta de concorrência interna do ALua, o código ficaria bloqueado no *loop*, e o processo nunca iria tratar os novos eventos. Em um paradigma orientado a eventos, como no ALua, poderíamos refazer este exemplo como mostrado a seguir:

```
-- Processo A
function First_Answer ()
  Answer = Second_Answer
end

function Second_Answer ()
  print("FIM.")
  alua.exit()
end

Answer = First_Answer   -- A variável Answer representa
                        -- o estado do processo A
alua.mcast({"B", "C"}, [[ alua.send("A","Answer()") ]])
```

Neste exemplo, a variável `Answer` representa o estado do processo `A`, que começa esperando pela primeira mensagem. Após enviar a mensagem para os processos `B` e `C`, ele espera até que um novo evento chegue. Ambos os processos `B` e `C` irão receber a mensagem

```
  alua.send("A","Answer()")
```

Quando a primeira mensagem chegar em `A`, a função `First_Answer` será chamada, de modo que `A` irá para um novo estado, esperando pela segunda mensagem. Quando a segunda mensagem chegar, a função `Second_Answer` será chamada, o que terminará a execução de `A`.

3.4
Flexibilidade

Nesta seção, discutimos a flexibilidade que se pode obter através do uso de um modelo de comunicação interpretado e orientado a eventos. Em contraste com os exemplos anteriores, que visavam ilustrar como programar tarefas comuns de comunicação em ALua, os exemplos descritos nesta seção são tipicamente difíceis de replicar em sistemas convencionais de programação.

3.4.1
Mensagem Auto-Replicante

O primeiro exemplo lida com a transmissão de uma ação para todos os nós de uma configuração em *pipeline* lógico, na qual cada nó não tem conhecimento sobre a configuração inteira.

Assumimos que a variável `prox`, em cada processo, contém o nome do processo seguinte no *pipeline*. A ação que queremos transmitir é

```
alua.send("A", "print("..n..")")
```

que fará com que cada processo envie o valor de sua própria variável `n` de volta para o processo `A`, pedindo que ele a exiba. Quando o sistema inicia a execução da aplicação, cada processo não tem conhecimento sobre a necessidade de transmitir uma mensagem para o próximo processo, de modo que a própria mensagem deve ser responsável por sua retransmissão.

A Figura 3.3 mostra uma solução parcial, onde a mensagem é retransmitida apenas uma vez (de `B` para `C`). Observe que esta mensagem nunca irá chegar ao processo `D`. No exemplo de código mostrado nesta figura, usamos a função de Lua `dostring` que recebe como parâmetro uma *string* e a executa como um trecho de código Lua.

Estendendo esta mesma solução para lidar com um número maior de retransmissões iria requerer que um trecho de código arbitrariamente grande fosse transmitido e só iria funcionar caso o transmissor original soubesse o número exato de processos no *pipeline*. Ao invés disso, nós usamos uma mensagem auto-replicante, como mostrado na Figura 3.4 [2].

[2]Deixamos o entendimento deste exemplo como um exercício para o leitor. A referência [43] contém uma discussão interessante sobre programas auto-replicantes.

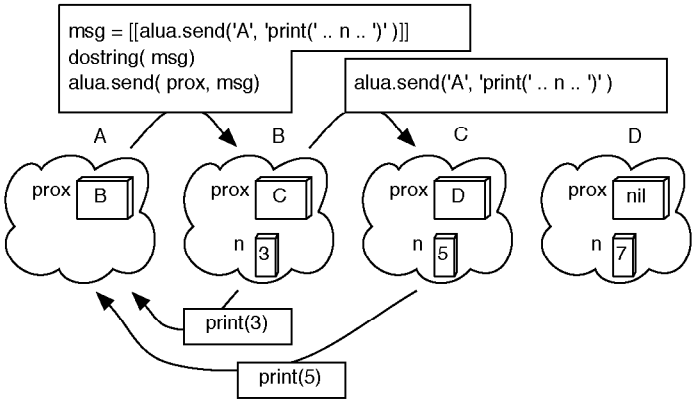

Figura 3.3: Mensagem quase auto-replicante.

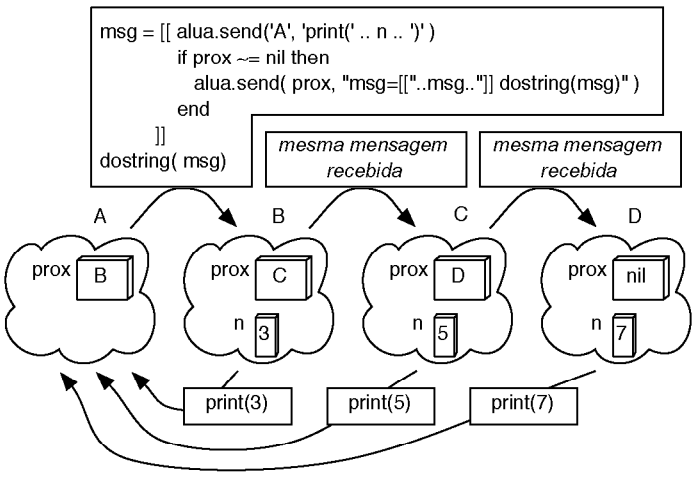

Figura 3.4: Mensagem auto-replicante.

3.4.2

Multiplicação de Tuplas

O trabalho [44] discute cinco paradigmas de programação paralela, cada um dos quais aplicável a um conjunto de algoritmos com alguma estrutura em comum. Como um experimento em ALua, implementamos uma versão paralela e outra seqüencial para alguns destes paradigmas. Cada uma destas implementações consiste de duas partes: um esqueleto independente da aplicação e algumas funções dependentes da aplicação. Para cada paradigma temos dois esqueletos, um para uma implementação seqüencial e outro para uma implementação paralela.

Este exemplo de programação ilustra como o ALua pode ser útil para fazer testes e prototipação. Para cada aplicação, as mesmas funções dependentes da aplicação são usadas tanto na versão seqüencial do algoritmo quanto na paralela.

Com isso, o programador pode juntar diferentes esqueletos com diferentes aplicações dinamicamente: Ele pode, por exemplo, carregar a versão seqüencial para testar e depurar o código específico da aplicação, e em seguida carregar o esqueleto paralelo e executá-lo na mesma aplicação. Vamos discutir um destes paradigmas, chamado de *Multiplicação de Tuplas* em [44], e mostrar como usá-lo para implementar dois algoritmos: multiplicação de matrizes e todos os pares de caminhos mais curtos.

Uma multiplicação de tuplas calcula uma matriz c, $n \times n$, como o "produto" de duas matrizes a e b. Os elementos da matriz são obtidos aplicando-se uma função f a cada par ordenado, que é composto por uma n-tupla de a (uma linha) e uma n-tupla de b (uma coluna), ou seja, $c_{ij} = f(a_{i,.}, b_{.,j})$.

Esta estrutura reflete bem a similaridade entre a multiplicação de uma matriz e um algoritmo baseado em matrizes para calcular os caminhos mais curtos entre todos os pares de nós em um grafo orientado [45]. No caso da multiplicação de matrizes, os elementos da tupla são linhas e colunas, respectivamente, e a função f é o produto cartesiano. Para o caso do todos os pares de caminhos mais curtos, vamos primeiro rever o algoritmo.

Como descrito em [45], o algoritmo baseado em matrizes que é mostrado a seguir pode ser usado para determinar caminhos mais curtos. O algoritmo constrói uma seqüência de matrizes de distância D^r, onde cada entrada $d^r_{i,j}$ representa o peso do caminho mais curto de i para j que passa em no máximo r nós intermediários. A matriz inicial D^0 (contendo caminhos sem nós intermediários) é construída diretamente da matriz de adjacência:

$$d^0_{i,j} = \begin{cases} 0 & \text{se } i = j \\ \text{peso da aresta edge (i,j)} & \text{se existir} \\ \infty & \text{caso contrário} \end{cases}$$

Assim, em cada passo $r+1$, substituímos o peso do caminho corrente entre i e j, que passa por no máximo r nós, pelo menor peso de um caminho entre i e j o qual percorre no máximo $r+1$ nós:

$$d^{r+1}_{i,j} = \min_{1 \le k \le n} (d^r_{i,k} + d^1_{k,j})$$

Este algoritmo precisa de $n-1$ iterações para construir a matriz de todos os pares de caminhos mais curtos, contendo os pesos mínimos dos caminhos percorrendo no máximo $n-1$ nós. Uma vez que a computação de cada $d_{i,j}$ na iteração r necessita de n comparações, este algoritmo calcula os caminhos mais curtos em $O(n^4)$.

Entretanto, como nosso objetivo é computar apenas a matriz final D^{n-1}, não precisamos calcular todas as matrizes D^r intermediárias para $1 \leq r \leq n-1$. Ao invés disso, podemos calcular D^{n-1} com $\log(n-1)$ passos combinando D^r com si mesmo a cada iteração:

$$d_{i,j}^{2r} = \min_{1 \leq k \leq n} (d_{i,k}^r + d_{k,j}^r)$$

Com esta melhoria, o tempo de execução para o algoritmo é $O(n^3 \log n)$.

Agora podemos voltar ao paradigma de multiplicação de tuplas. O paradigma descreve a computação de uma matriz como um produto de outras duas matrizes através da aplicação da função f em pares de n-tuplas.

A Figura 3.5 apresenta o esqueleto seqüencial para a multiplicação de tuplas. A função `multiply` recebe duas matrizes e suas dimensões, calcula a matriz resultante e retorna o resultado. Para simplificar, assumimos que ambas as matrizes são quadradas.

```
function multiply(a, b, n)
   local c = {}
   for i = 1, n do
      c[i] = {}         -- cria uma tabela Lua
      for j = 1, n do
         c[i][j] = f(a[i], b[j])
      end
   end
   return c
end
```

Figura 3.5: Paradigma de multiplicação de tuplas: esqueleto seqüencial.

A Figura 3.6 mostra o código específico para o problema de multiplicação de matrizes. Ele define uma única função `f`, que é chamada por `multiply`. Esta função recebe uma linha da primeira matriz e uma coluna da segunda, e retorna um elemento da matriz resultante.

A Figura 3.7 mostra o código específico para o problema de todos os pares de caminhos mais curtos. A função `allpaths` cria a matriz de distâncias $n \times n$ inicial de um grafo de n nós e chama a função `multiply` $\log n$ vezes para calcular a matriz resultante. Mais uma vez, a função `f` é chamada por `multiply`.

A Figura 3.8 mostra um esqueleto para o paradigma paralelo. O processo mestre executa a versão paralela da função `multiply`, distribuindo o cálculo das linhas entre todos os processos disponíveis. Cada processo calcula pelo menos `qmin` linhas e no máximo `qmax` linhas da matriz resultante. Cada processo recebe apenas

```
function matrix(a, b, n)
   local c
   c = multiply(a,
                transpose(b),
                n)
   if c then finish(c) end
end

function finish(a)
   show(a)
end
```

```
function f(ai, bj)
  local n = getn(ai) -- obtém o
                     -- tamanho
                     -- da matriz.
  local cij = 0
  for k = 1, n do
    cij = cij + ai[k]*bj[k]
  end
return cij
end
```

Figura 3.6: Multiplicação de matrizes - código específico da aplicação.

```
function allpaths(a, n)
  d = a
  m = 1
  while (m < n) and d do
    d = multiply(d,
                 transpose(d),
                 n)
    m = 2 * m
  end
  if d then finish(d) end
end
```

```
function f(ai, bj)
  local n = getn(ai)
  local cij = infinity
  for k=1,n do
    cij = min(cij,
              sum(ai[k], bj[k]))
  end
  return cij
end
```

```
function finish (a)
  if m < n then   -- apenas para a versão paralela.
    d = multiply(a, transpose(a), n)
    m = 2 * m
  else
    show_graph(a)
  end
end
```

Figura 3.7: Caminhos mais curtos - código específico da aplicação.

as linhas necessárias da primeira matriz, mas todos eles recebem a segunda matriz inteira.

Cada um dos processos executa a função `node` repetidamente (a qual eles obtêm do processo mestre) para calcular as linhas da matriz final. Um processo envia cada linha para o mestre assim que ele termina seu cálculo. Como a comunicação em ALua é assíncrona, o mestre possui uma função de *callback* chamada `finish` para sinalizar a aplicação que ela já obteve o resultado final.

Na multiplicação de matrizes, a função `finish` simplesmente exibe a matriz resultante. Entretanto, ao calcular todos os pares de caminhos mais curtos, torna-se necessário fazer multiplicações sucessivas até o resultado final ser obtido. Portanto,

```
function multiply(a, b, dim)
  local bstr = alua.tostring(b)
  n = dim
  for k=1,nmax do    -- nmax é o número de nós
    -- frow é a primeira linha a ser calculada no nó k.
    -- lrow é a última linha a ser calculada no nó k.
    -- Ambos dependem de qmin e qmax (ver o texto).
    msg = format([[column=%s row={}]], bstr)
    for i=frow,lrow do
      msg = msg .. format(" row[%d]=%s",i, alua.tostring(a[i]))
    end
    msg = msg .. format(" n=%d node(%d, %d)", n, frow, lrow)
    alua.send(Procs[k], msg)
  end
  master_i = 0
  master_c = {}
end

function master_receive(row, i)
  master_c[i] = row
  master_i = master_i + 1
  if master_i == n then     -- o resultado está pronto.
    finish(master_c)
  end
end

-- Esta string é enviada para os processos escravos quando
-- eles são inicializados.
node_code=[[
  function node(r, s)        -- 1 <= r <= s <= n
    for i=r,s do
      c = {}
      for j=1,n do
        c[j] = f(row[i], column[j])
      end
      msg = format("master_receive(%s,%d)", alua.tostring(c), i)
        alua.send("master", msg)
    end
  end
]]
```

Figura 3.8: Paradigma de multiplicação de tuplas: esqueleto paralelo.

a função `finish` chama a função `multiply` novamente após cada iteração (menos a final) e chama a função `show_graph` para exibir a matriz de caminhos mais curtos quando o resultado final é obtido.

Como dissemos anteriormente, o interessante desta abordagem é a possibilidade de trocar apenas o esqueleto do programa para trocar o paradigma.

3.4.3 Um Time de Agentes Assíncronos

Outro exemplo interessante da flexibilidade do ALua é a implementação de um Time de Agentes Assíncronos (*A-Team*) [46] para o problema de recobrimento de conjuntos [47]. O programa completo é bastante complexo, por isso nesta seção vamos apresentar apenas um esboço dele.

Um A-Team é composto por *agentes* (entidades autônomas trabalhando de forma cooperativa) e memórias compartilhadas, criando super agentes com um fluxo de dados cíclico e sem controle de fluxo. As principais características da arquitetura do A-Team são:

- agentes autônomos
  Um agente autônomo faz sua própria escolha quando seleciona seus dados de entrada e política de alocação de recursos. Uma vez que agentes autônomos são completamente independentes um do outro, novos agentes podem ser adicionados ou removidos do sistema sem que os outros agentes ou um gerente sejam notificados.
- comunicação assíncrona
  Agentes podem ler e escrever dados nas memórias compartilhadas sem nenhum tipo de sincronização entre eles. Junto com sua autonomia, esta característica permite que os agentes trabalhem em paralelo, em tempo integral.
- fluxo de dados cíclico
  Agentes podem recuperar, modificar e armazenar dados em memórias compartilhadas continuamente. Esse fluxo de dados cíclico permite iteração e *feedback* contínuos entre os agentes.

Experiências anteriores com este paradigma indicam que a cooperação entre agentes tende a gerar sinergia, ou seja, o resultado produzido por eles, quando visto como um todo, pode ser melhor que a soma dos resultados obtidos independentemente (existe uma chance melhor de se encontrar uma solução perto da

ótima) [46, 47]. Estas experiências também sugerem que um A-Team é uma organização escalável, i.e., seu desempenho fica melhor quando novos componentes (como agentes ou memórias) são introduzidos no sistema.

Nós nos baseamos em uma implementação anterior do A-Teams escrita em C, que usa o pacote de comunicação DPSK+P [48], baseado em uma arquitetura de objetos compartilhados. O sistema (descrito em [47]) é composto por dois *servidores* que agem como repositórios de soluções, representando a memória compartilhada do A-Team, dois *agentes de inicialização* que inicializam o repositório, e nove *agentes trabalhadores* que implementam diferentes algoritmos de refinamento para as soluções armazenadas nos repositórios.

Dada uma instância específica do problema de recobrimento de conjuntos, o padrão típico de comportamento de um agente A-Team é requisitar uma solução do repositório de soluções, refinar esta solução, enviar a nova solução de volta para o repositório, e recomeçar. Existem dois repositórios que armazenam as soluções geradas. Ambos os repositórios apenas atendem a requisições dos agentes. Dependendo dos dados de entrada, esta aplicação pode continuar sua execução por alguns dias antes de atingir um resultado interessante.

Com base nesta implementação, substituímos o pacote de comunicação DPSK+P pelo ALua. O sistema original era composto por 15 programas, com 34 módulos, onde apenas 18 eram diretamente relacionados a atividades de comunicação. Reescrevemos estes 18 módulos para usar ALua, e mantivemos os demais módulos inalterados. Nossa nova implementação mantém um console Lua disponível, de modo que possamos entrar novos comandos no sistema a qualquer momento.

O uso do ALua nos permite reconfigurar o sistema, consultar o repositório ou até mesmo redefinir funções dinamicamente. Por exemplo, o código a seguir mostra como podemos redefinir o comportamento da função `alua.send` em tempo de execução, para instrumentar a aplicação, criando um arquivo de *log* das mensagens de comunicação.

```
old_send = alua.send

function new_send(to, msg)
   -- escreve arquivo de log
   write(alua.mytid .. " enviando mensagem para " .. to)
   msg = format("write(alua.mytid..' recebendo mensagem de %s');",
                alua.mytid) .. msg
   old_send(to, msg)
end
```

```
alua.send = new_send
```

Após esta redefinição, a função `alua.send` irá executar o código

```
write(alua.mytid .. " enviando mensagem para " .. to)
```

toda vez que enviar alguma mensagem, e irá instruir o receptor a executar

```
write(alua.mytid .. ' recebendo mensagem de <SENDER'S TID>')
```

ao receber a mensagem.

Se executarmos o código acima em apenas um agente, apenas suas mensagens serão monitoradas. Para acompanhar todas as mensagens do sistema, podemos enviar o código acima para todos os agentes. Para restaurar a função original, apenas enviamos a mensagem `alua.send = old_send` para cada agente.

Este exemplo ilustra como a natureza interpretada do ALua pode ser útil para controlar aplicações de longa duração interativamente. A aplicação A-Team pode levar horas para terminar e por isso é extremamente interessante para o programador ser capaz de alterar seu comportamento sem ter que esperar por uma nova execução. O programador ALua pode usar um console interativo para injetar código de monitoramento, como discutido acima, ou para redefinir o comportamento de um ou mais agentes, permitindo que partes do programa sejam redefinidas de acordo com resultados parciais observados.