Proposta de um Modelo para o Processamento de Eventos Concorrentes no ALua

Guilherme Salazar, Bruno Silvestre[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)

{guilhermesilva, brunoos}@inf.ufg.br

Abstract. *Concurrent programming has been used for a long time as a means to make better use of CPU time. Since the use of interruptions to simulate multiprogramming to the support of preemptive multithreading by operating systems, such programming model has been shown effective. However, concurrent programming may be complex and troublesome. The model that will be introduced in this paper aims to explore event-driven concurrency, employing advantages and avoiding disadvantages of the models commonly used.*

Resumo. *Há muito a programação concorrente vem sendo usada como uma forma de melhor aproveitar o tempo de CPU. Desde o emprego de interrupções para simular multiprogramação ao suporte de multithreading preemptiva pelos sistemas operacionais, tal modelo tem se mostrado eficiente. Contudo, a programação concorrente pode ser complexa e problemática. O modelo de programação que será apresentado neste trabalho busca explorar concorrência com orientação a eventos, aproveitando as vantagens e evitando desvantagens de modelos comumente usados.*

1. Introdução

Nos últimos anos a indústria de hardware presenciou o progressivo enfraquecimento da Lei de Moore diante dos limites físicos impostos sobre a evolução dos microprocessadores, em termos de aumento de frequência do relógio. Como uma forma de superar tais limites, e permitir que o desempenho continuasse aumentando, surgiram os processadores multinúcleo. A crescente popularização desses processadores com mais de um núcleo de processamento eleva o interesse por programação concorrente, que passa de um modelo alternativo a uma abordagem imprescindível para o desenvolvimento de aplicações com requisitos de desempenho. Nesse sentido, explorar a programação concorrente significa melhor aproveitar recursos que, de outra maneira, estariam ociosos.

Multithreading preemptiva com memória compartilhada é um modelo largamente adotado em aplicações concorrentes. Apesar de o sistema operacional tratar do escalonamento, livrando o programador de tal tarefa, lidar com a coordenação do acesso a recursos compartilhados não é trivial [Ousterhout 1996]. *Locks* e semáforos são primitivas utilizadas com o intuito de coordenar a computação concorrente. Além da nova classe de *bugs* que surgiu com esse modelo de programação, tais como condições de corrida e inversão de prioridade, as trocas de contexto são computacionalmente intensivas e podem gerar grandes degradações no desempenho.

Como uma alternativa ao modelo baseado puramente em threads, a orientação a eventos vem sendo apontada como uma boa forma de estruturar aplicações

[Dabek et al. 2002, Ousterhout 1996, Lee 2006]. As tarefas nesse modelo são representadas por eventos, que são recebidos e despachados por um *loop* principal. O fluxo de controle retorna ao *loop* principal somente após o tratamento do evento ter sido concluído, o que evita os problemas associados ao uso de threads por permitir que apenas uma tarefa esteja em execução em um dado momento. Como o processamento de cada evento conduz a aplicação a um novo estado, esse modelo é semelhante a uma máquina de estados finitos.

Outros trabalhos vêm investigando o uso de *multithreading* cooperativa para aumentar o isolamento entre os estados dos eventos [von Behren et al. 2003, Fischer et al. 2007, Moura and Ierusalimschy 2009]. *Multithreading* cooperativa se caracteriza por seu modelo simplista de execução, uma vez que apenas uma thread pode estar ativa em um determinado momento. Além disso, a transferência de controle é explícita e o estado da computação salvo entre suspensões, o que dá ao programador o controle de quando ocorre a troca de contexto e ao mesmo tempo a facilidade de não precisar lidar com o salvamento e recuperação do estado da computação. No entanto, vale destacar que *multithreading* cooperativa não tira proveito de máquinas com vários núcleos, o que nos leva de volta à abordagem concorrente, que deve ser tratada de forma a evitar os problemas que motivaram a busca por modelos alternativos.

Nosso objetivo é investigar concorrência aplicada a um modelo orientado a eventos. Utilizamos o ALua [Ururahy and Rodriguez 1999, ALua 2004] em nosso trabalho, que é uma infraestrutura para construção de aplicações distribuídas orientadas a eventos. Não pretendemos expor o programador ao modelo de memória compartilhada, tendo que prover primitivas de baixo nível (*e.g.*, *locks*), trazendo de volta as dificuldades apontadas por Ousterhout [Ousterhout 1996]. Seguiremos a proposta de que a concorrência, quando provida, deve ser mantida longe do programador, e outros mecanismos de mais alto nível devem ser providos para que tudo seja feito de forma transparente e simples [Ousterhout 1996].

Na seção 2, apresentaremos o ALua em mais detalhes, bem como as modificações realizadas para introduzir um modelo de concorrência juntamente com orientação a eventos. Na seção 3, apresentamos a adaptação de um servidor web para demonstrar o uso e ganhos do modelo proposto. Finalmente, na seção 4, apresentamos as conclusões e trabalhos futuros.

2. ALua e Concorrência

O ALua [Ururahy and Rodriguez 1999, ALua 2004] tem sido nossa infraestrutura de experimentações para estudo e construção de aplicações distribuídas orientadas a eventos. Apesar de ter sido proposto para computação distribuída, sentimos a necessidade e a importância de prover um mecanismo de concorrência para explorar o poder computacional e mesmo a realização de tarefas locais (em cada máquina da rede) de forma concorrente.

Os eventos em ALua são mensagens assíncronas constituídas de trechos de código Lua que são executados no receptor. A primitiva `alua.send` é utilizada para enviar uma mensagem e não há uma correspondente para recebimento. No modelo original, assim que um evento é recebido, ele é processado completamente antes de o próximo ser iniciado, evitando condições de corrida e outros problemas de concorrência. A figura 1 mostra

o modelo original do ALua, com vários processos formando uma aplicação distribuída. Cada processo executa código Lua e os *daemons* são processos especiais da arquitetura que funcionam como porta de entrada para as máquinas da rede. Os daemons são responsáveis por criar dinamicamente novos processos e fazer o roteamento das mensagens.

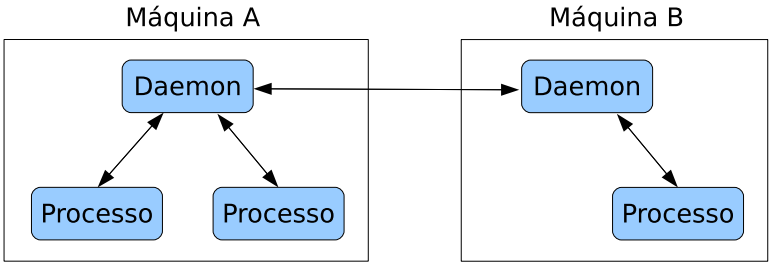

Figura 1. Arquitetura original do ALua.

Alua foi desenvolvido utilizando a linguagem de programação Lua [Ierusalimschy et al.]. Lua é conhecida por sua extensibilidade: podemos, com relativa facilidade, registrar novas funcionalidades escritas em outras linguagens (usualmente C/C++, mas também Fortran, C# e outras). Além disso, Lua pode ser utilizada para estender aplicações ou para conectar componentes de software. Simplicidade e ortogonalidade são duas marcas fundamentais de Lua: apesar de ser uma linguagem pequena e simples, os conceitos presentes na linguagem podem ser combinados e estendidos para se adaptar a diversos tipos de problema. Eficiência também é um ponto forte da linguagem, que figura entre as mais rápidas entre as linguagens dinâmicas [Ierusalimschy 2003].

2.1. Adaptando o ALua para Concorrência

Nossa principal modificação no modelo original do ALua, para permitir a inclusão de concorrência na arquitetura, é no funcionamento dos processos da figura 1. Esses processos são na realidade processos do sistema operacional que rodam uma instância da máquina virtual Lua. Com isso, só podemos rodar um programa Lua (um *loop* de eventos) em cada processo.

Uma importante característica da linguagem Lua é a possibilidade de várias instâncias da máquina virtual Lua serem criadas em um único processo do sistema operacional. Cada uma dessas instâncias é conhecida como *estado Lua*. Outro detalhe é que não há compartilhamento de informações entre os estados Lua, cada um possui suas próprias funções, variáveis e estruturas de controle, permitindo que várias máquinas virtuais coexistam sem que haja interferência mútua.

Sendo assim, no novo modelo proposto para o ALua, a unidade básica de processamento exportada para o desenvolvedor é o estado Lua. A ideia proposta é utilizar vários estados Lua, cada um contendo seu próprio *loop* de eventos, que recebe e despacha cada evento para o devido tratador, retornando ao loop somente após finalizar o processamento do evento. Chamaremos de *processo Lua* a unidade básica composta de um estado Lua e seu *loop* de eventos. A figura 2 mostra a nova arquitetura de processos.

Conforme mostrado na figura 2, o modelo é composto por três unidades fundamentais: processos Lua, threads e processos do sistema operacional. Uma biblioteca foi

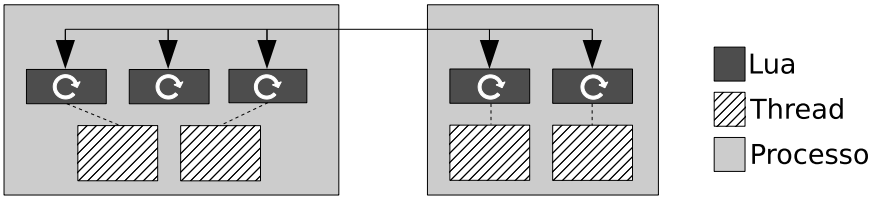

Figura 2. Arquitetura do modelo proposto para orientação a eventos e concorrência.

desenvolvida em C para prover as operações básicas de criação de processos Lua e a comunicação entre eles. Cada processo Lua recebe, ao ser criado, um trecho de código que deve ser executado. Após isso, o processo é colocado em uma fila de prontos para ser executado. A execução dos processos Lua é realizada por um conjunto de threads (*pool de threads*) que são disparadas assim que a biblioteca é carregada no programa principal.

Utilizando vários estados Lua, o problema da interferência na computação em cada estado é evitado, o que nos permite liberar o desenvolvedor do uso direto de mecanismos de baixo nível para coordenação de concorrência, ficando essa tarefa a cargo da infraestrutura subjacente. O programador deve organizar o seu código de tal forma que cada estado Lua realize tarefas concorrentes dentro de um mesmo processo do sistema operacional. De certa forma, isso segue o modelo de organização com threads já conhecido, a diferença é que não há memória compartilhada.

Como não há compartilhamento de informações entre os estados, o conceito de troca de mensagens, que já era adotado para troca de informações entre processos da rede, foi utilizado para possibilitar a troca de informações locais. Adotamos a mesma primitiva `alua.send`, ficando transparente o envio de mensagens para outros processos tanto na rede como localmente. Apesar disso, a infraestrutura utiliza mecanismos internos diferentes para despachar as mensagens de acordo com o destinatário.

Mensagens para destinatários que compartilham o mesmo processo do sistema operacional são despachadas via uma fila, i.e., utilizamos memória compartilhada para enviar essas mensagens. Note que essa tarefa é realizada pela arquitetura e é transparente ao desenvolvedor da aplicação, que só chama a função `alua.send`. Assim, promovemos o isolamento da concorrência. Caso o destinatário esteja em outro processo do sistema operacional, o envio da mensagens se dá da forma original do ALua, neste caso, via canais TCP.

Outra característica que introduzimos no modelo foi o de threads flutuantes. Para que o programa Lua seja executado em um dos estados é preciso que uma thread do sistema operacional seja alocada para ele. Se adotarmos a estratégia de alocar uma thread para cada estado Lua, corremos o risco de desperdício de recursos, pois se não houver eventos a serem tratados, a thread ficará bloqueada. Sendo assim, decidimos adotar uma política de criar uma quantidade menor de threads e que, quando um estado não tem eventos para tratar, a thread é liberada para atender outro estado que tem eventos para tratar. Podemos ver esse efeito na figura 2.

Apenas para embasar essa política, nos sistema operacionais Linux de hoje, a cada thread criada é alocado um espaço de memória em torno de 8 a 10 megabytes para a pilha de execução. Um estado Lua ocupa em torno de uma centena de kilobytes [Skyrme 2007].

Com isso é interessante ter um grande número de estados Lua sendo atendidos por um conjunto menor de threads.

Após a modificação no modelo do processo, realizamos a integração com a distribuição. Com a adaptação da arquitetura do ALua, os processos de sistema operacional do modelo original, que continham um programa Lua, passam a poder conter vários *processos Lua*. Vale destacar que a comunicação entre os processos Lua é realizada pela fila de mensagens, enquanto a comunicação entre os processos de sistema operacional é realizada por meio do canal TCP. Para encaminhar a mensagem corretamente, cada processo Lua contém um identificador que indica se o evento deve ser enviado por meio da fila de eventos ou deve ser encaminhado ao daemon para envio por meio do canal TCP. A figura 3 mostra a arquitetura do ALua após a adaptação.

Para facilitar a integração com o modelo distribuído, adotamos um processo Lua especial conhecido *despachante*. Ele é responsável por receber e encaminhar as mensagens aos respectivos destinatários dentro de um processo do sistema operacional. Podemos fazer uma analogia dele com o daemon, sendo esse responsável por fazer a mesma tarefa, mas nas máquinas da rede. O despachante deve verificar constantemente se há mensagens no canal TCP.

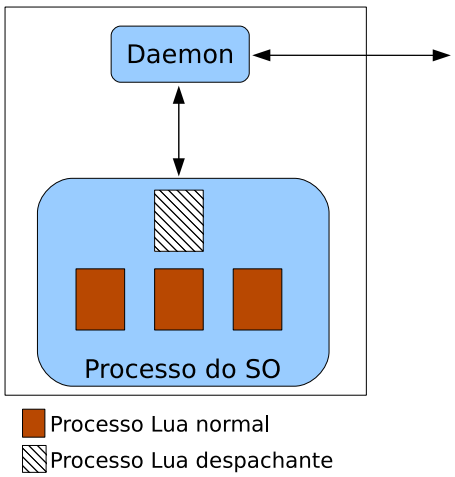

Figura 3. Arquitetura do ALua após a integração da concorrência e distribuição.

3. Estudo de Caso

Para exercitar as características do modelo proposto, adaptamos o servidor web Xavante [Xavante 2004], desenvolvido em Lua, para uso de processos Lua no processamento de requisições. Neste estudo de caso, não estamos interessados em testar as características da computação distribuída do ALua, mas apenas nossa adaptação ao modelo para concorrência.

Na implementação original do Xavante, cada requisição HTTP é tratada por meio de multithreading cooperativa, oferecida por Lua através das co-rotinas. A arquitetura foi adaptada para o modelo de bolsa de tarefas (*bag of tasks*), com um processo Lua mestre esperando por requisições HTTP e outros processos Lua (processos trabalhadores) responsáveis por atender requisições (papel das co-rotinas no modelo original). Como

mostra a figura 4, o servidor é executado em um único processo do sistema operacional, com um número limitado de threads executando os trabalhadores.

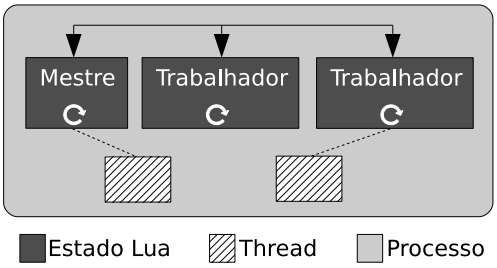

Figura 4. Arquitetura do Xavante após a adaptação para o modelo concorrente.

Os testes foram realizados na provisão de conteúdo dinâmico, i.e., uma página HTML é gerada dinamicamente por um script Lua contendo números de 1 a 5.000. Isso simula a necessidade de processamento na provisão do conteúdo. Em cada teste utilizamos 400 processos Lua como trabalhadores e variamos o número de threads (entre 2 e 16 threads) para avaliar o comportamento do servidor web.

Utilizamos como servidor uma máquina com quatro núcleos de processamento (cada um com 2,66 GHz), 4GB de memória RAM, executando Linux com kernel 32 bits versão 2.6.26. Como clientes, para a geração de tráfego, utilizamos 10 máquinas com dois núcleos de processamento (cada um com 2,66 GHz), 1 GB de RAM, executando Linux com kernel 32 bits versão 2.6.26.

Os clientes geravam um conjunto de requisições simultâneas (100, 400, 700 e 1.000) utilizando a ferramenta ApacheBench (versão 2.3). O tempo médio de resposta do servidor para cada cenário é mostrado na tabela 1.

Requisições	Original	2 Ths	4 Ths	8 Ths	16 Ths	32 Ths
100	24,91	9,31	4,44	4,90	4,74	4,95
400	103,81	32,72	20,30	20,80	20,58	20,52
700	175,51	82,44	33,59	34,53	35,15	35,31
1.000	246,08	89,60	52,53	53,90	53,53	53,76

Tabela 1. Tempos médios, em segundos, da distribuição de conteúdo dinâmico no Xavante com concorrência.

Como pode ser observado na tabela, as melhoras nos tempos variaram de 2,68 a 5,16 vezes, nas implementações com 2 e 4 threads, respectivamente, em comparação com a implementação original. Isso deixa claro o ganho de concorrência proporcionado pelo modelo, pois aproveita os núcleos disponíveis. No entanto, podemos notar que a partir de 4 threads não houve ganhos significativos (perdas podem ser observadas). Atribuímos esse comportamento ao fato da máquina servidora possuir apenas quatro núcleos. Dessa forma, o aumento no número de threads não traz um ganho real de processamento paralelo e até promove a disputa e alta troca de contexto nos núcleos de processamento.

4. Conclusão

Neste trabalho apresentamos uma adaptação do modelo de orientação a eventos provido pelo ALua para inclusão de concorrência. Essa adaptação manteve a abstração fornecida

pelo ALua, mantendo assim a ideia de que os eventos são processados ininterruptamente, mas agora com a possibilidade de ter tratamento de eventos em paralelo, o que nos permite explorar melhor os núcleos de processamento, aumentando o desempenho de certas aplicações. Com isso, introduzimos o conceito de *processo Lua* que corresponde a um *estado Lua* juntamente com uma thread que o está executando. A comunicação entre os processos Lua se dá por meio de troca de mensagens, não havendo compartilhamento de memória visível ao programador. Em [Silvestre 2009] são apresentados mais detalhes sobre testes realizados e outras aplicações do modelo proposto.

Apesar de a programação concorrente com compartilhamento de memória estar amplamente difundida, e em certos casos ser o modelo ideal, a coordenação se torna um desafio na medida em que a aplicação cresce. O modelo apresentado é uma alternativa a ser considerada, visto que além de evitar grande parte dos problemas associados a *multithreading* preemptiva, pode também propiciar ganho de desempenho.

A linguagem Erlang é conhecida pelo seu modelo de processos leves, o que garante alta escalabilidade aos programas, pois a criação de um grande número de processos não se torna um problema quanto esses são leves – comparativamente, processos e threads do sistema operacional podem ser tidos como pesados. Apesar de a implementação em Lua consumir mais memória do que a de Erlang [Skyrme 2007], consome bem menos do que o sistema operacional. Dessa forma, uma combinação de threads e estados Lua resulta em um modelo mais flexível e que melhor aproveita os recursos do sistema.

Outros trabalhos utilizaram a ideia de aumentar o isolamento entre unidades de execução por meio de estados Lua. Dois deles são as bibliotecas Lanes [Kauppi 2007] e luaproc [Skyrme 2007]. Em Lanes, a cada novo estado Lua criado, uma nova thread é disparada para executar o código que foi carregado no estado. Essa correspondência forte entre estado Lua e thread pode levar a um grande consumo de recurso por parte da aplicação. No caso do luaproc, há um pool de threads, como realizado neste trabalho, mas a comunicação entre os estados Lua se dá de forma síncrona e por meio de canais de comunicação nomeados. Dependendo da necessidade da aplicação, chamadas de comunicação síncrona (i.e., bloqueantes) podem não ser adequadas. Tanto o lanes como o luaproc não foram desenvolvidos para dar suporte à criação de aplicações distribuídas, o que também introduz complicadores com relação à coordenação da aplicação. Esse assunto de coordenação e sincronização de aplicações distribuídas orientadas a eventos é discutido em [Silvestre et al. 2008].

Como trabalho futuro, estamos interessados em estudos de políticas de escalonamento que possam melhorar o desempenho com relação à associação das threads com os estados Lua. Atualmente há um único conjunto de threads atendendo os estados Lua, numa política FIFO. Nesse sentido, estamos iniciando um projeto de iniciação científica, modalidade PIBIC/PIVIC, para atacar esse assunto. Esperamos analisar a gerência automática das threads oferecidas, utilizando por exemplo a biblioteca libdispatch [Apple 2011] (criada pela Apple e presente também no sistema operacional FreeBSD), com alguma política de gerência manual que use diversos pools de threads.

Referências

ALua (2004). ALua: asynchronous distributed programming in Lua. `http://alua.inf.puc-rio.br`. Acesso em: Mar 2011.

Apple (2011). Concurrency programming guide. http://developer.apple.com/library/mac/\#documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html\#//apple_ref/doc/uid/TP40008091.

Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., and Morris, R. (2002). Event-driven programming for robust software. In *ACM Special Interest Group on Operating Systems (SIGPOS)*, pages 186–189. ACM Press.

Fischer, J., Majumdar, R., and Millstein, T. (2007). Tasks: language support for event-driven programming. In *ACM Special Interest Group on Programming Languages (SIGPLAN)*, pages 134–143, New York, NY, USA. ACM Press.

Ierusalimschy, R. (2003). *Programming in Lua*. Lua.org.

Ierusalimschy, R., Figueiredo, L., and Celes, W. The programming language Lua. http://www.tecgraf.puc-rio.br/lua/.

Kauppi, A. (2007). Lua Lanes: Multithreading in Lua. http://luaforge.net/projects/lanes/. Acesso em: Jun 2009.

Lee, E. (2006). The problem with threads. *IEEE Computer*, 39(5):33–42.

Moura, A. and Ierusalimschy, R. (2009). Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31.

Ousterhout, J. (1996). Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference.

Silvestre, B. (2009). *Modelos de Concorrência e Coordenação para o Desenvolvimento de Aplicações Orientadas a Eventos em Lua*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro.

Silvestre, B., Rossetto, S., Rodriguez, N., and Briot, J.-P. (2008). Flexibility and coordination in event-based, loosely-coupled, distributed systems. Monografia em Ciência da Computação, 09/08, Depto de Informática, PUC-Rio.

Skyrme, A. (2007). Um modelo alternativo para programação concorrente em Lua. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro.

Ururahy, C. and Rodriguez, N. (1999). ALua: An event-driven communication mechanism for parallel and distributed programming. In *Parallel and Distributed Computing and Systems*, Fort Lauderdale, Florida.

von Behren, R., Condit, J., Zhou, F., Necula, G., and Brewer, E. (2003). Capriccio: scalable threads for internet services. In *19th ACM Symposium on Operating Systems Principles*, pages 268–281. ACM Press.

Xavante (2004). Xavante web server. http://www.keplerproject.org/xavante/. Acesso em: Jun 2009.