6
Avaliação

A avaliação da biblioteca desenvolvida como parte desse trabalho teve por objetivo não apenas medir os tempos de execução de aplicações de teste, mas também subsidiar decisões de projeto durante o seu desenvolvimento e estimar custos de tempo e recursos necessários ao seu funcionamento. Adicionalmente, permitiu a realização de comparações preliminares com outras bibliotecas e linguagens de programação voltadas à programação concorrente.

Essa avaliação, cujos resultados são apresentados neste capítulo, incluiu as seguintes medidas de desempenho:

Criação de processos Lua – cujo objetivo foi medir os tempos necessários à criação de processos Lua e comparar o desempenho da criação de processos Lua com as funções `luaL_newstate` e `lua_newthread`;

Carga de bibliotecas padrão de Lua – cujo objetivo foi comparar o desempenho da criação de processos Lua ao carregar todas as bibliotecas padrão de Lua em cada estado Lua criado, ao não carregar nenhuma biblioteca e ao carregar apenas as bibliotecas `basic`, `package` e a própria biblioteca para programação concorrente;

Troca de mensagens entre processos Lua – cujo objetivo foi avaliar o desempenho da troca de mensagens entre processos Lua por meio de troca sucessiva de mensagens de tamanhos variados entre dois processos e por meio de troca cíclica de mensagens por processos ordenados em anel;

Reciclagem de processos Lua – cujo objetivo foi estimar o ganho de desempenho que pode ser ocasionado pela utilização da funcionalidade de reciclagem durante a criação sucessiva de processos Lua sem sincronização;

Consumo de memória – cujo objetivo foi medir o consumo de memória resultante da criação de canais de comunicação e processos Lua, bem como derivar uma fórmula para estimar o consumo de memória a partir

das quantidades de trabalhadores, canais de comunicação e processos Lua;

Paralelismo – cujo objetivo foi avaliar o ganho de desempenho ocasionado pela execução, em um microcomputador multiprocessado, de uma aplicação paralela de busca de texto desenvolvida com a biblioteca, bem como comparar a execução de uma versão seqüencial da aplicação com uma versão similar desenvolvida apenas com as bibliotecas padrão de Lua.

A avaliação incluiu ainda as seguintes medidas comparativas:

POSIX Threads (pthreads) – cujo objetivo foi comparar o desempenho, referente aos tempos de criação de fluxos de execução e ao consumo de memória, da biblioteca para programação concorrente em Lua e de uma biblioteca que implementa o padrão POSIX Threads para programação com *multithreading* preemptivo e compartilhamento de memória;

Erlang – cujo objetivo foi comparar o desempenho, referente aos tempos de criação de fluxos de execução, ao consumo de memória e à troca de mensagens, da biblioteca para programação concorrente em Lua e da linguagem de programação Erlang.

Os testes, em sua maioria, foram conduzidos em microcomputador com processador AMD Athlon 64 X2 Dual-core 3600+, 3GB de memória RAM e sistema operacional Linux, distribuição Ubuntu 7.10 - Gutsy Gibbon, com *kernel* `2.6.22-14-generic #1 SMP` e biblioteca Native POSIX Thread Library (NPTL) 2.6.1. Em consonância com a quantidade de núcleos do processador, foram utilizados dois trabalhadores (ou duas *kernel threads*). A ausência de menção explícita a configurações alternativas denota a utilização dessa configuração e quantidade de trabalhadores.

Foi empreendido esforço consciente no sentido de evitar interferências indesejáveis durante a realização dos testes, que foram executados em sua totalidade com contas de usuários não privilegiados. Cada medida foi realizada no mínimo três vezes e os resultados apresentados correspondem às médias aritméticas das medidas. Também são apresentados, sempre que possível, os desvios padrão (σ) correspondentes, ao lado de cada medida.

Todas as medições de tempo foram realizadas com o comando `time`, do GNU Bourne Again Shell (`bash`) do sistema operacional Linux.

6.1 Medidas de Desempenho

6.1.1 Criação de Estados de Processos Lua

A API de Lua para programação em C permite a criação de estados Lua de acordo com duas abordagens distintas. A função `lua_newstate` e a função correlata `luaL_newstate` criam estados Lua totalmente independentes. A função `lua_newthread` cria novos estados Lua a partir de estados Lua já existentes. Os estados Lua criados com a função `lua_newthread` possuem fluxos de execução independentes, mas compartilham objetos globais (como tabelas e variáveis) com os estados a partir dos quais são criados.

O compartilhamento de objetos globais decorrente da utilização da função `lua_newthread` possibilita o aproveitamento de bibliotecas carregadas nos estados Lua a partir dos quais são criados novos estados. Dessa forma, ao utilizá-la para criar estados associados a processos Lua, é possível carregar todas as bibliotecas uma só vez em um único estado Lua, e utilizá-lo como ponto de partida para criação dos novos estados.

A redução na quantidade de cargas de bibliotecas em estados Lua reduz o custo da criação de processos Lua. Por outro lado, o compartilhamento de objetos globais tem outras conseqüências, como a possibilidade de interferência indesejável durante a execução de processos Lua cujos códigos utilizem nomes coincidentes para variáveis globais. Adicionalmente, a existência de memória compartilhada entre processos Lua é contraditória com a estratégia de utilizar a troca de mensagens como recurso exclusivo para comunicação e sincronização e tornaria necessário o desenvolvimento de primitivas para contenção de acessos concorrentes a variáveis globais.

A percepção de que a criação de estados Lua independentes representa a estratégia mais alinhada com os objetivos do modelo proposto e de que os prejuízos associados ao compartilhamento de memória entre estados Lua excedem os benefícios, resultou na utilização exclusiva da função `luaL_newstate` para criar os estados associados aos processos Lua. Entretanto, foi realizado estudo comparativo entre as duas abordagens com o objetivo de avaliar o custo de desempenho ocasionado por essa decisão.

O estudo comparativo foi realizado por meio de testes de criação sucessiva de processos Lua. O código Lua utilizado para realizar os testes é apresentado no Apêndice C. Sua execução compreende a criação de um canal de comunicação para cada processo Lua que será criado, a criação seqüencial de processos Lua que apenas aguardam o recebimento de uma mensagem e o subseqüente

envio de uma mensagem para cada processo Lua. O envio das mensagens após a criação dos processos garante que, pelo menos durante um intervalo de tempo, todos os processos coexistirão. Versões distintas da biblioteca foram compiladas exclusivamente para realização desse teste.

A figura 6.1 apresenta os tempos de execução resultantes da criação sucessiva de quantidades variadas de processos Lua com a utilização das duas estratégias descritas anteriormente: com a função `luaL_newstate`, que produz estados Lua independentes, e com a função `lua_newthread` acompanhada da utilização de um estado Lua, onde são carregadas todas as bibliotecas padrão, para originar os estados associados aos processos Lua.

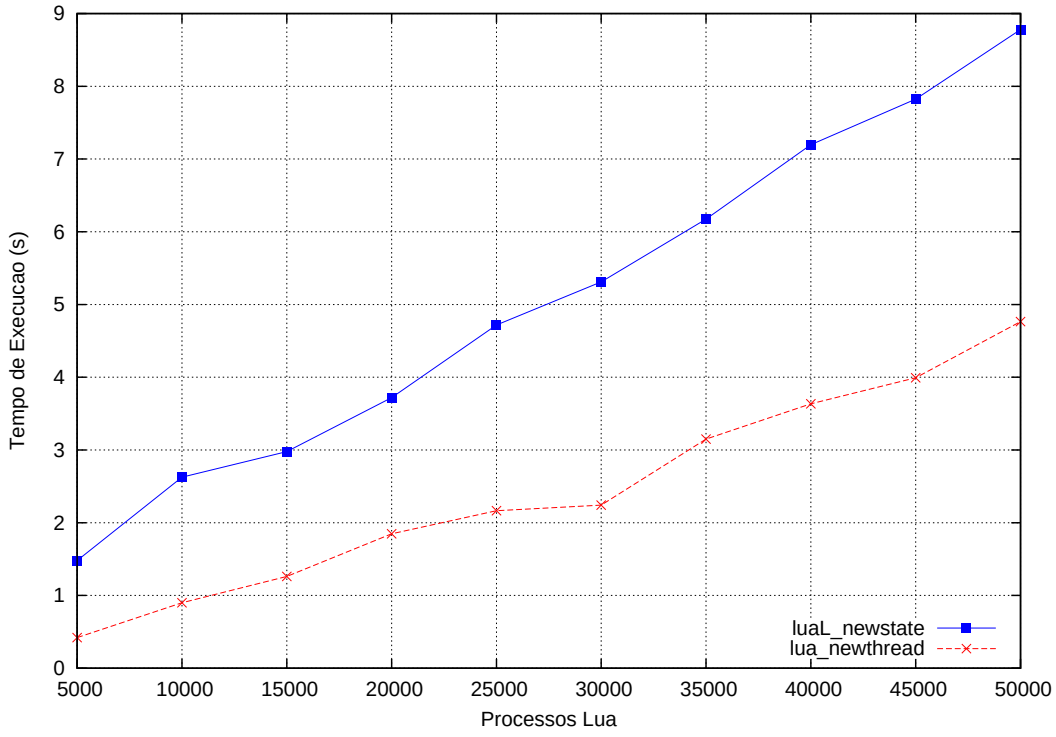

Figura 6.1: Criação sucessiva de processos Lua com estados criados com lua_newthread e luaL_newstate.

Como pode ser observado, os resultados comprovam o menor custo de criação de processos Lua com a função `lua_newthread`, em decorrência da carga singular das bibliotecas padrão. Os resultados também indicam que o aumento do tempo de execução ocasionado pela criação de mais processos Lua é aproximadamente linear.

6.1.2 Carga de Bibiliotecas Padrão em Estados de Processos Lua

A carga de bibliotecas em estados Lua, como mencionado na seção 5.1, é realizada de forma seletiva e comedida. Apenas as bibliotecas padrão `basic`

e `package`, além da própria biblioteca para programação concorrente, são carregadas nos estados, o que reduz o custo de criação de processos Lua.

O custo de carregar as bibliotecas em estados Lua e seu impacto à criação de processos Lua podem ser quantificados por meio de testes de criação sucessiva de processos Lua. O código Lua utilizado para realizar esse teste é o mesmo utilizado para realizar os testes comparativos entre a criação de estados com as funções `lua_newthread` e `luaL_newstate`, apresentado no Apêndice C. Mais uma vez, versões distintas da biblioteca foram compiladas exclusivamente para realização do teste.

As figuras 6.2 e 6.3 apresentam, respectivamente, os tempos de execução resultantes da criação sucessiva de quantidades variadas de processos Lua com a carga de todas as bibliotecas padrão nos estados e sem a carga de bibliotecas padrão nos estados. Os tempos de execução resultantes da criação padrão de processos Lua, que contempla apenas a carga das bibliotecas padrão `basic` e `package`, são apresentados na figura 6.1.

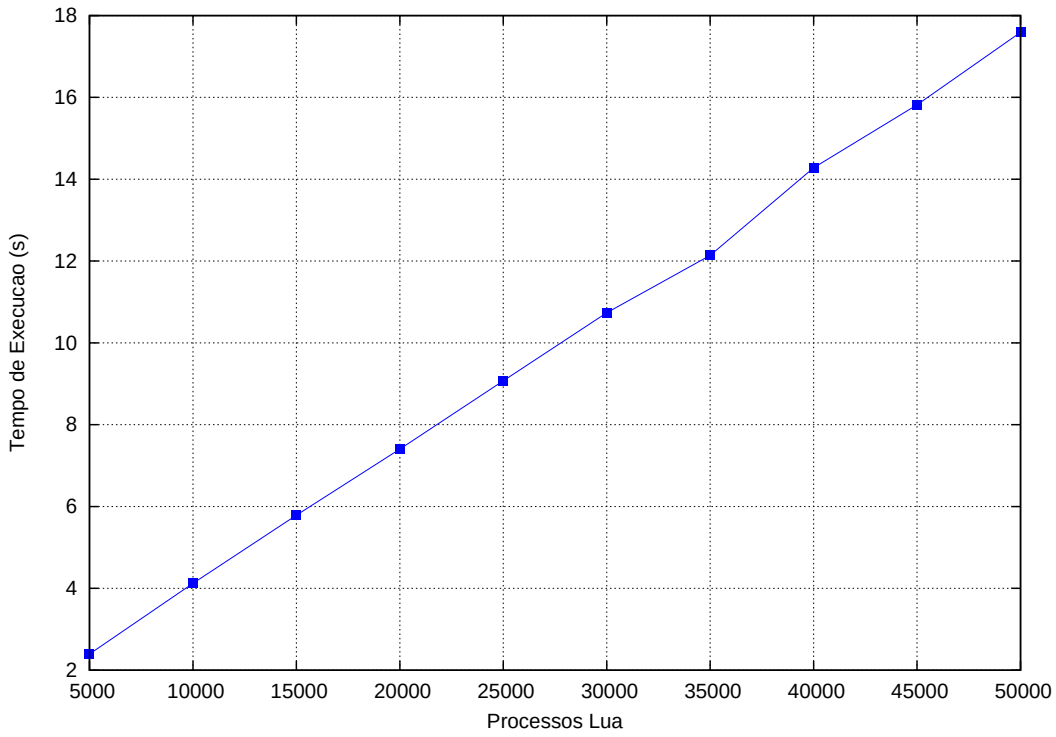

Figura 6.2: Criação sucessiva de processos Lua com carga de todas as bibliotecas padrão nos estados.

Os resultados demonstram que a preocupação com a carga de bibliotecas nos estados é procedente e apontam um aumento de cerca de quatro vezes entre os tempos de execução sem a carga de bibliotecas padrão e os tempos de execução com a carga de todas as bibliotecas padrão. Adicionalmente,

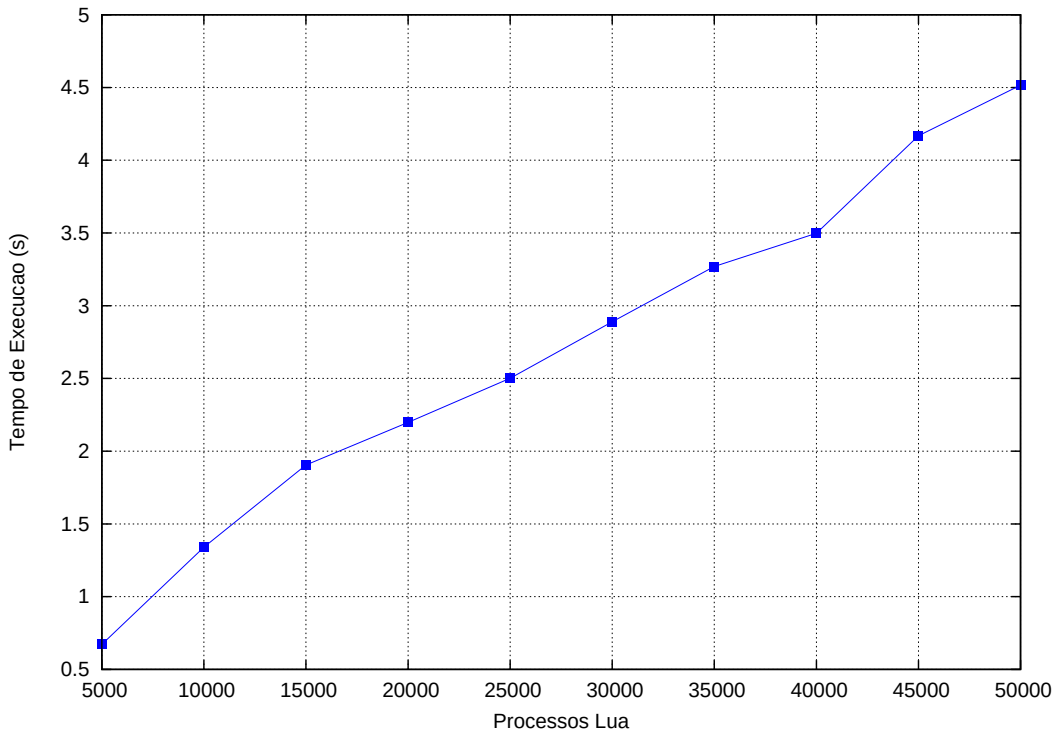

Figura 6.3: Criação sucessiva de processos Lua sem carga de bibliotecas padrão nos estados.

os resultados reforçam o caráter linear do aumento do tempo de execução ocasionado pela criação de mais processos Lua.

6.1.3 Troca de Mensagens entre Processos Lua

A troca de mensagens é o principal recurso oferecido pelo modelo proposto para comunicação e sincronização de processos Lua. O custo de envio e recebimento de mensagens foi quantitativamente avaliado por meio de testes de envio e recebimento sucessivos de mensagens de tamanhos variados.

No primeiro teste, cujo código Lua é apresentado no Apêndice D, são criados dois processos Lua e um canal de comunicação. Em seguida, um processo Lua realiza o envio seqüencial de uma quantidade definida de mensagens cujo conteúdo é lido a partir de um arquivo. A tabela 6.1 apresenta os tempos de execução referentes ao envio sucessivo de quantidades variadas de mensagens de tamanhos distintos.

No segundo teste, cujo código Lua é apresentado no Apêndice E, são criados processos Lua e canais de comunicação individuais para cada processo. Em seguida, uma mensagem simples é enviada ao primeiro processo Lua, que envia uma mensagem com o mesmo conteúdo para o segundo processo Lua e assim sucessivamente até que a mensagem retorne ao primeiro processo. O ciclo

Tamanho da Mensagem (bytes)	Envios/ Recebimentos	Tempo de Execução (s)	σ
10	10	0,009	0,001
	100	0,009	0,003
	1.000	0,012	0,002
104	10	0,009	0,001
	100	0,008	0,001
	1.000	0,011	0,001
1.048	10	0,008	0,000
	100	0,008	0,002
	1.000	0,019	0,006
10.485	10	0,012	0,002
	100	0,016	0,004
	1.000	0,053	0,003
104.857	10	0,040	0,002
	100	0,072	0,001
	1.000	0,400	0,002
1.048.576	10	0,332	0,005
	100	0,687	0,002
	1.000	4,267	0,005
10.485.760	10	2,732	0,359
	100	3,876	0,068
	1.000	24,190	0,233

Tabela 6.1: Tempos de execução para envio e recebimento de mensagens de tamanhos variados.

é repetido por um número definido de vezes, até que a mensagem do último ciclo é recebida pelo primeiro processo Lua. Essa estrutura cíclica compõe um anel de troca de mensagens. A tabela 6.2 apresenta os tempos de execução referentes à execução do teste com quantidades variadas de processos Lua e ciclos de troca de mensagens.

Os resultados indicam que, conforme esperado, o aumento no tempo de execução é aproximadamente linear à medida que a quantidade de processos Lua aumenta.

Processos Lua	Ciclos de Troca de Mensagens	Tempo de Execução (s)	σ
10	10	0,014	0,001
	100	0,021	0,007
	1.000	0,112	0,045
	10.000	1,333	0,028
100	10	0,064	0,001
	100	0,188	0,002
	1.000	1,425	0,017
	10.000	14,078	0,276
1.000	10	0,581	0,009
	100	2,177	0,026
	1.000	15,892	4,497
	10.000	153,956	44,827
10.000	10	4,143	0,080
	100	12,432	0,047
	1.000	107,315	1,642
	10.000	1.064,548	29,746

Tabela 6.2: Tempos de execução de anel de troca de mensagens.

6.1.4 Reciclagem de Processos Lua

Conforme mencionado na seção 5.1, uma das funcionalidades contempladas pela biblioteca é a reciclagem de processos Lua, cujo objetivo é permitir o reaproveitamento de estados Lua e conseqüentemente reduzir o ônus de criação de processos Lua. O ganho de desempenho associado à reciclagem de processos Lua foi quantificado por meio de testes de criação sucessiva de processos Lua não-sincronizados.

O código Lua utilizado para realizar esse teste é apresentado no Apêndice F. Seu funcionamento é similar ao do código utilizado anteriormente para efetuar testes de criação sucessiva de processos Lua. A diferença fundamental está na ausência de sincronização entre os processos criados. Os processos, criados seqüencialmente, apenas imprimem uma mensagem na tela e terminam a sua execução. A simplicidade do código dos processos criados reduz o tempo de execução individual e conseqüentemente permite avaliar melhor a influência da reciclagem.

A tabela 6.3 apresenta os tempos de execução referentes à criação de

quantidades variadas de processos Lua com limites variados de reciclagem e a contagem de processos cuja criação resultou no reaproveitamento de estado Lua existente. O limite de reciclagem simplesmente define o tamanho máximo da fila de estados Lua que podem ser reaproveitados. Assim, um limite de reciclagem de zero indica que nenhum estado será reaproveitado, enquanto um limite de n indica que a qualquer instante existirão no máximo n estados prontos para serem reaproveitados na criação de novos processos Lua. Durante a realização dos testes a saída padrão (`stdout`) foi redirecionada para o dispositivo nulo (`/dev/null`). Uma versão específica da biblioteca foi compilada exclusivamente para permitir a exibição da contagem de reciclagens.

Os resultados não deixam dúvidas com relação ao potencial de ganho de desempenho representado pela reciclagem de processos Lua. Os tempos de execução referentes à criação de processos sem reciclagem apresentam redução de cerca de cinco vezes em relação aos tempos para criação com reciclagem de apenas dez processos. É possível observar ainda que a partir de um determinado limite de processos recicláveis, os tempos de execução não diminuem e, em alguns casos, até mesmo aumentam. Esse comportamento ressalta a importância de escolher criteriosamente o limite de processos recicláveis, de forma que o custo da reciclagem não inviabilize ganho de desempenho.

6.1.5 Consumo de Memória

O consumo de memória é uma característica importante de qualquer biblioteca voltada à programação concorrente, uma vez que está diretamente associado à escalabilidade. É desejável que o consumo de memória ocasionado pela criação de novos fluxos de execução seja reduzido e que seja possível estimar antecipadamente o consumo de memória ocasionado pela criação de quantidades variadas de fluxos de execução.

Uma avaliação quantitativa do consumo de memória ocasionado pela criação de canais de comunicação e processos Lua foi realizada por meio de testes de criação sucessiva de quantidades variadas de canais e processos. O código Lua utilizado para realizar os testes, apresentado no Apêndice G, é bastante similar ao código utilizado anteriormente para realizar os testes de criação de estados Lua. No entanto, dessa vez, retardos foram introduzidos no código para permitir a realização de medidas de consumo de memória em pontos específicos. As medidas foram realizadas com o comando `pmap`, voltado ao mapeamento do consumo de **memória virtual** de processos do sistema operacional, que integra o pacote de utilitários `procps` do sistema operacional Linux.

Limite de Processos Lua Recicláveis	Processos Lua	Tempo de Execução (s)	σ	Reciclagens	σ
0	10	0,008	0,000	0	0
	100	0,028	0,001	0	0
	1.000	0,189	0,001	0	0
	10.000	1,774	0,068	0	0
	100.000	9,711	0,112	0	0
1	10	0,008	0,000	0	0
	100	0,019	0,001	42	8
	1.000	0,077	0,009	752	51
	10.000	0,733	0,293	7.527	2.079
	100.000	5,182	0,215	65.391	888
10	10	0,007	0,000	0	0
	100	0,017	0,001	57	0
	1.000	0,039	0,001	974	14
	10.000	0,308	0,009	9.959	13
	100.000	1,975	0,037	99.869	29
100	10	0,008	0,001	0	0
	100	0,012	0,002	74	15
	1.000	0,042	0,003	960	12
	10.000	0,337	0,016	9.909	9
	100.000	2,791	0,212	99.936	41
1.000	10	0,008	0,001	0	0
	100	0,014	0,000	59	1
	1.000	0,038	0,001	978	8
	10.000	0,355	0,036	9.917	27
	100.000	2,460	0,180	99.819	146

Tabela 6.3: Tempos de execução para disparar processos Lua com reciclagem de processos e quantidade efetiva de reciclagens.

Os resultados, apresentados na tabela 6.4, representam as medidas do consumo de memória em três momentos distintos da execução do código: após carregar a biblioteca, criar um novo trabalhador e disparar o processo Lua principal; após criar os canais de comunicação individuais dos processos Lua; e após criar os processos Lua.

A utilização de dois trabalhadores corresponde à criação de duas *kernel threads*, o que ocasiona o consumo inicial de pelo menos 16.384kB em decorrência da alocação de 8.192kB para criação de cada pilha virtual. O consumo de memória ocasionado pela criação de novos trabalhadores (C_{trab}) pode ser estimado pela fórmula 6-1, onde c_{pilha} é a quantidade de memória reservada para a pilha de uma *kernel thread* (8.192kB, por exemplo) e n_{trab} é a quantidade de trabalhadores.

$$C_{trab} \approx c_{pilha} \times n_{trab} \tag{6-1}$$

Estimativa, em kilobytes, do consumo básico de memória foi calculada com base nos resultados apresentados na tabela 6.4 e na quantidade de memória reservada para as pilhas de cada *kernel thread*. A estimativa, apresentada na fórmula 6-2, representa o consumo de memória após a inicialização da biblioteca e a criação de um processo Lua, excluindo o consumo ocasionado pela criação das pilhas das *kernel threads*.

$$C_{base} \approx 33.941 - (2 \times 8.192) = 17.557 \tag{6-2}$$

Estimativas, em kilobytes, do consumo de memória ocasionado pela criação de canais de comunicação (C_{canal}) e pela criação de processos Lua ($C_{luaproc}$) foram calculadas com base nos resultados apresentados na tabela 6.4 e são apresentadas nas fórmulas 6-3 e 6-4, respectivamente. As variáveis n_{canais} e $n_{luaprocs}$ representam as quantidades de canais e processos Lua, respectivamente.

$$C_{canal} \approx 0,37 \times n_{canais} \tag{6-3}$$

$$C_{luaproc} \approx 17,48 \times n_{luaprocs} \tag{6-4}$$

Dessa forma, o consumo total de memória, em kilobytes, pode ser estimado pela fórmula 6-5. As variáveis C_{base} e c_{pilha} possuem relação intrínseca com o sistema operacional, em particular com as bibliotecas GNU C (`glibc`) e POSIX Threads, portanto propositalmente não foram quantificadas.

$$C_{total} \approx C_{base} + (c_{pilha} \times n_{trab}) + (0,37 \times n_{canais}) + (17,48 \times n_{luaprocs}) \tag{6-5}$$

A precisão da fórmula proposta para estimar o consumo total de memória pode ser avaliada mediante comparação entre os consumos estimados e os consumos reais, medidos durante a realização dos testes. A comparação é apresentada na tabela 6.5.

Os resultados da comparação indicam que a fórmula oferece uma boa aproximação para estimar o consumo real de memória. No melhor caso a

Processos Lua	Consumo de Memória Inicial (kB)	σ	Consumo de Memória Após Criação dos Canais (kB)	σ	Consumo de Memória Após Criação dos Processos (kB)	σ
5.000	33.941	2	35.869	2	152.577	988
10.000	33.941	2	37.721	2	220.541	2
15.000	33.941	2	39.305	2	317.472	6.935
20.000	33.941	2	41.537	2	450.798	4.5131
25.000	33.942	2	42.990	2	502.258	1.990
30.000	33.941	2	44.441	2	606.518	23.584
35.000	33.940	0	47.560	0	723.549	62.674
40.000	33.942	2	49.014	2	787.749	13.141
45.000	33.940	0	50.464	0	918.421	42.649
50.000	33.941	2	52.049	2	996.636	53.035

Tabela 6.4: Consumos de memória durante a criação sucessiva de processos Lua.

Processos Lua	Consumo de Memória Real (kB)	Consumo de Memória Estimado (kB)	Variação Absoluta (kB)	Grau de Acerto (Estimado / Real)
5.000	152.577	123.191	29.386	0,807
10.000	220.541	212.441	8.100	0,963
15.000	317.472	301.691	15.781	0,950
20.000	450.798	390.941	59.857	0,867
25.000	502.258	480.191	22.067	0,956
30.000	606.518	569.441	37.077	0,939
35.000	723.549	658.691	64.858	0,910
40.000	787.749	747.941	39.808	0,949
45.000	918.421	837.191	81.230	0,912
50.000	996.636	926.441	70.195	0,930

Tabela 6.5: Comparação entre os consumos de memória estimados e reais.

estimativa correspondeu a cerca de 96% do consumo real, enquanto no pior caso a estimativa correspondeu a cerca de 80% do consumo real.

6.1.6 Exploração de Paralelismo

A utilização de mais de um trabalhador em ambientes multiprocessados possibilita a execução paralela de processos Lua. A paralelização e os ganhos de desempenho decorrentes foram objeto de avaliação por meio da realização de testes com uma aplicação de busca de cadeias de caracteres em arquivos texto (ASCII).

Essa aplicação foi motivada por uma demanda real para reduzir o tempo necessário à busca de URLs em registros de acesso à Internet realizados através de um *proxy* Microsoft Internet Security and Acceleration Server (ISA). Foram desenvolvidas duas versões da aplicação: uma que utiliza a biblioteca para programação concorrente, voltada à execução paralela, e outra que utiliza apenas as bibliotecas básicas de Lua, voltada à execução seqüencial.

A versão que utiliza a biblioteca para programação concorrente com processos Lua é dividida em três módulos, apresentados no Apêndice H. O primeiro deles, apresentado na seção H.1.1, é responsável pela inicialização da aplicação: criar os trabalhadores e os canais de comunicação, instanciar um processo Lua mestre (coordenador) e processos Lua trabalhadores, bem como enviar ao mestre a quantidade de processos Lua trabalhadores disponíveis, o nome do arquivo com as cadeias de caracteres a serem buscadas e os nomes dos arquivos onde as buscas serão realizadas.

O segundo módulo, apresentado na seção H.1.2, é responsável por coordenar a distribuição de tarefas e centralizar a recepção de resultados. É sua atribuição realizar a leitura do arquivo com as cadeias de caracteres, enviar as cadeias lidas para os trabalhores, enviar os nomes dos arquivos que devem ser inspecionados para os processos Lua trabalhadores, receber os resultados por eles enviados e notificá-los quando todos os arquivos tiverem sido processados.

O terceiro módulo, apresentado na seção H.1.3, representa o processo Lua trabalhador, que é responsável por inspecionar os arquivos texto em busca das cadeias lidas e informar eventuais casamentos de padrão para o processo Lua mestre. Cada processo Lua trabalhador recebe um nome de arquivo distinto e processa o arquivo na íntegra antes de enviar os resultados.

A execução de ambas as versões da aplicação descrita nessa seção foi excepcionalmente realizada em microcomputador com quatro processadores AMD Opteron Dual-core 2.2GHz, 32GB de memória RAM e sistema operacional Linux, distribuição CentOS 5.1, com *kernel* `2.6.18-53.1.6.el5xen #1 SMP` e biblioteca Native POSIX Thread Library (NPTL) 2.5.

Inicialmente, na execução da versão voltada à execução paralela, foram

utilizados seis trabalhadores visando fomentar o paralelismo e reduzir a concorrência na execução de um processo Lua mestre e cinco processos Lua trabalhadores. Em seguida, com a mesma versão, foi utilizado apenas um trabalhador para permitir comparação mais equilibrada com a versão que utiliza somente as bibliotecas padrão de Lua. O arquivo de padrões foi o mesmo em todos os testes e continha 25 linhas. Nos testes com múltiplos arquivos, foram utilizadas cópias do mesmo arquivo, que é composto por 6.605.423 linhas e 2.147.483.849 bytes (cerca de 2GB). Os resultados encontram-se na figura 6.4.

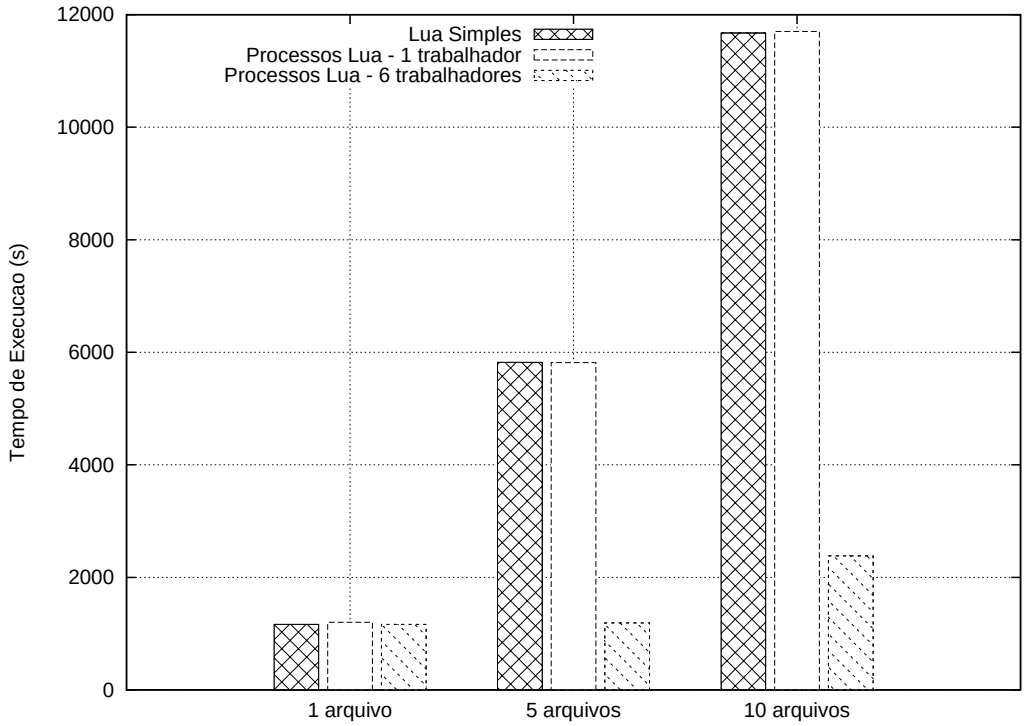

Figura 6.4: Tempos de execução para a busca de cadeias de caracteres.

Os resultados indicam, conforme esperado, que a exploração do paralelismo em máquinas multiprocessadas resulta na redução proporcional do tempo de execução. Como pode ser observado, ao utilizar Lua simples ou apenas um trabalhador, o tempo de execução aumenta linearmente com a quantidade de arquivos. Por outro lado, ao utilizar seis trabalhadores, o tempo de execução para busca em cinco arquivos é similar ao tempo de execução para busca em um arquivo, devido à paralelização da execução (um trabalhador executa o mestre e os outros cinco realizam a busca nos arquivos). Ainda com seis trabalhadores, é possível observar que a busca em dez arquivos resulta, como esperado, no aumento linear, em cerca de duas vezes, do tempo de execução. Os resultados indicam também que a diferença nos tempos de execução das versões que utilizam Lua simples e apenas um trabalhador são irrisórias, o que exalta o bom desempenho da biblioteca.

Ainda que, de maneira geral, a utilização de múltiplos trabalhadores em ambientes multiprocessados permita ganhos de desempenho, compete à aplicação explorar o paralelismo adequadamente. É necessário que a definição da quantidade de trabalhadores seja criteriosa e esteja em sintonia com a estrutura da aplicação. A utilização de mais de um trabalhor, por si só, não apenas não assegura ganhos, como pode ocasionar degradação do desempenho em decorrência do aumento de trocas de contexto, bloqueios e consumo de recursos de processamento.

A degradação do desempenho resultante do aumento arbitrário da quantidade de trabalhadores foi observada durante a realização de testes com a aplicação para troca cíclica (anel) de mensagens, cujo código é apresentado no Apêndice E, e quantidades variadas de trabalhadores. Essa aplicação foi escolhida por ser inerentemente serial - somente um processo Lua pode prosseguir enquanto os demais aguardam o recebimento de mensagens. Os resultados referentes aos tempos de execução, no ambiente padrão onde foram realizados os demais testes, de 1.000 ciclos de troca de mensagem por 1.000 processos Lua, com quantidades variadas de trabalhadores, são apresentados na tabela 6.6.

Tempo de Execução (s) 1 Trabalhador	σ	Tempo de Execução (s) 2 Trabalhadores	σ	Tempo de Execução (s) 3 Trabalhadores	σ
7,049	0,247	17,813	0,122	16,685	1,408

Tabela 6.6: Tempos de execução de um anel de troca de mensagens 1.000 x 1.000 com quantidades variadas de trabalhadores.

6.2 Comparativo com POSIX Threads

A biblioteca Native POSIX Thread Library (NPTL), sucessora da biblioteca LinuxThreads, é a principal biblioteca utilizada em distribuições atuais do sistema operacional Linux para programação concorrente com *multithreading* preemptivo e memória compartilhada. Essa característica, aliada à intenção de oferecer alternativa superior à sua utilização exclusiva, estimulou a realização de uma avaliação comparativa entre a NPTL e a biblioteca desenvolvida para implementar o modelo proposto para programação concorrente em Lua.

No entanto, como já elaborado no capítulo 2, a NPTL apresenta limitações de escalabilidade, o que prejudica a realização de testes nas mesmas condições utilizadas para testar a biblioteca desenvolvida. No microcomputador utilizado para realizar os testes, na prática, não é possível criar mais do

que cerca de 20.000 *kernel threads* simultâneas com a NPTL sem esbarrar em limites impostos pelo sistema operacional. Em comparação, como observado anteriormente, quantidades superiores de fluxos de execução simultâneos de código Lua podem ser criados com o auxílio da biblioteca desenvolvida.

Adicionalmente, a NPTL não provê primitivas para a comunicação entre *threads* por troca de mensagens. Dessa forma, a realização de quaisquer testes que envolvessem a comunicação entre fluxos de execução demandaria a implementação de mecanismos de comunicação entre *threads* ou a utilização de memória compartilhada aliada a recursos da biblioteca para contenção de acesso a variáveis compartilhadas. A diferença fundamental na forma de comunicação entre fluxos de execução torna pouco proveitosa qualquer tentativa de comparação referente a esse aspecto.

Portanto, face às restrições de escalabilidade e à distinção na estrutura de comunicação entre fluxos de execução, optou-se por realizar apenas dois testes simples para avaliar os tempos de execução referentes à criação sucessiva de quantidades variadas de *threads* e o consumo de memória ocasionado pela criação de *threads*. O código-fonte utilizado para realizar os testes é apresentado nos Apêndices I e J, respectivamente.

No teste de criação sucessiva de quantidades variadas de *threads*, nenhuma sincronização adicional foi utilizada, além do necessário para impedir que a execução do processo do sistema operacional fosse encerrada antes da conclusão da execução de todas as *threads*. As *threads* são criadas no estado *detached* mediante utilização do atributo `PTHREAD_CREATE_DETACHED`, o que possibilita a liberação dos recursos alocados às *threads* tão logo a execução seja concluída e proporciona algum ganho de escalabilidade. Os tempos de execução são apresentados na figura 6.5.

Para facilitar a comparação com a biblioteca desenvolvida para programação concorrente em Lua, a figura 6.6 apresenta os tempos de execução para criação sucessiva de quantidades análogas de processos Lua, sem sincronização entre os processos. O código Lua utilizado para realizar esse teste é o mesmo utilizado para o teste de reciclagens, apresentado no Apêndice F; no entanto, o limite de processos recicláveis foi fixado em zero, ou seja, nenhum processo foi reciclado.

Os resultados do teste demonstram que, apesar das restrições de escalabilidade apontadas, a criação de *kernel threads* com a NPTL pode ser realizada em tempos menores do que os observados na criação de processos Lua. No entanto, ao comparar diretamente os tempos de execução, é preciso considerar que o código usado para realizar os testes com a NPTL foi compilado, enquanto o código usado para realizar os testes com Lua foi interpretado. Adi-

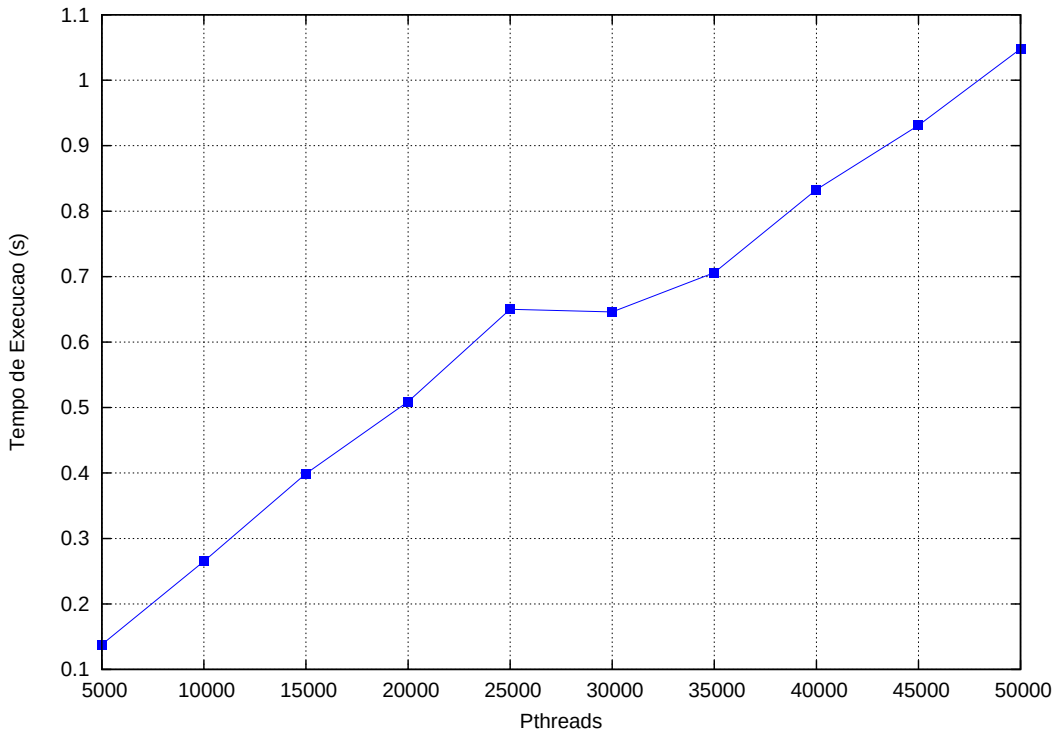

Figura 6.5: Criação sucessiva de Pthreads.

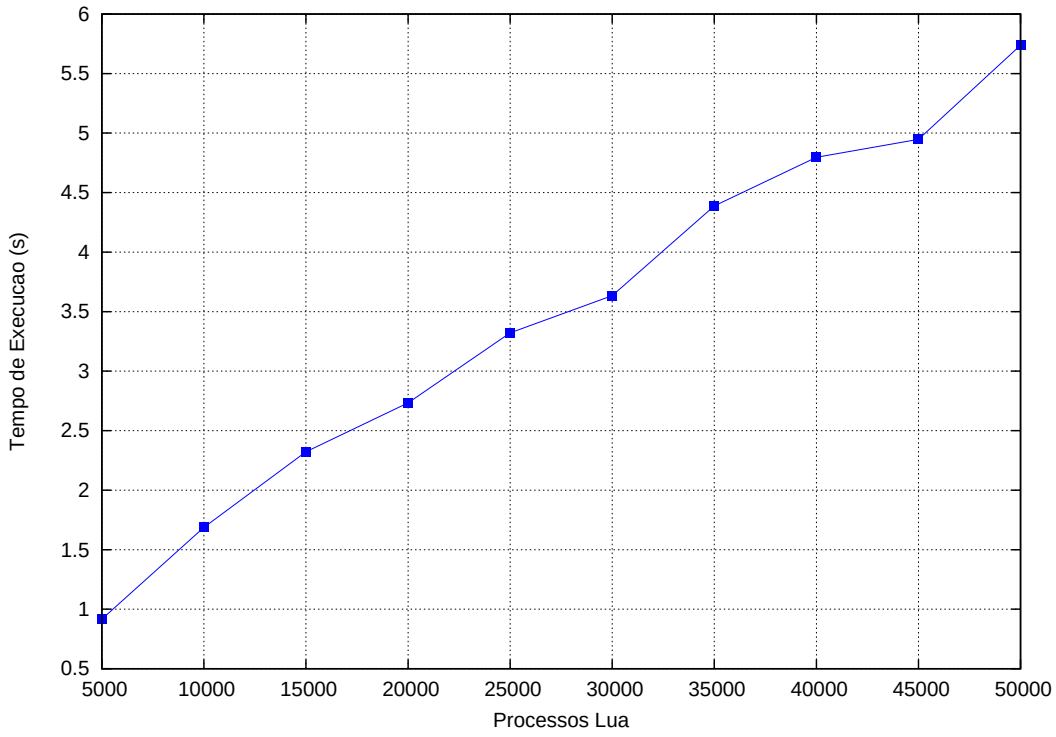

Figura 6.6: Criação sucessiva de processos Lua sem sincronização entre processos.

cionalmente, a NPTL possui maior integração com o *kernel*, uma vez que é destinada à gestão de *kernel threads*.

Na avaliação do consumo de memória ocasionado pela criação de *threads*

foi adotada abordagem distinta. Recursos de sincronização intrínsecos à biblioteca NPTL foram utilizados para assegurar a coexistência das *threads* criadas durante a realização das medidas. Em seguida, foram introduzidos retardos (com a função `usleep`) imediatamente após o início da execução e imediatamente após a criação de todas as *threads*. Os consumos de memória, medidos em ambos os pontos com o comando `pmap`, são apresentados na tabela 6.7.

Pthreads	Consumo de Memória Inicial (kB)	σ	Consumo de Memória Após Criação das Threads (kB)	σ
5.000	6.002	2	40.989.426	2
10.000	6.042	2	81.970.918	2
15.000	6.084	4	122.952.280	4
20.000	5.994	2	163.933.774	2

Tabela 6.7: Consumos de memória durante a criação sucessiva de Pthreads.

Os resultados confirmam que o consumo de memória ocasionado pela criação de *threads* com a NPTL é bastante elevado, em particular quando comparado ao consumo de memória da biblioteca para programação concorrente em Lua (tabela 6.4), e claramente representa um obstáculo à sua escalabilidade.

6.3 Comparativo com Erlang

A linguagem de programação Erlang, entre os trabalhos apresentados no capítulo 3, representa uma alternativa madura para programação concorrente e é reconhecida por proporcionar boa escalabilidade. Dessa forma, caracteriza bom termo de comparação de desempenho para a biblioteca desenvolvida para implementar o modelo proposto para programação concorrente em Lua.

Ainda que o código-fonte de Erlang esteja publicamente disponível e tenha sido brevemente inspecionado, uma análise minuciosa não foi realizada por não estar contemplada no escopo do trabalho. Os testes com Erlang tiveram por objetivo apenas apresentar uma avaliação comparativa preliminar e, por mais que tenham sido empreendidos esforços para homogeneizar as condições dos testes, não é possível garantir a equivalência sintática entre os códigos Lua e Erlang utilizados. Em particular, por se tratar de linguagem de programação funcional, Erlang demanda a utilização de um padrão de desenvolvimento que difere do padrão normalmente utilizado no desenvolvimento em Lua.

A execução de programas desenvolvidos em Erlang pode ser realizada por meio da utilização do interpretador (`escript`) ou mediante compilação

(`erlc`) e subseqüente utilização da *shell* do emulador de Erlang (`erl`). O interpretador não permite a passagem de parâmetros para configuração do ambiente de execução, o que representa um limitante em alguns casos. Tanto o interpretador, como o emulador em sua configuração padrão, não utilizam simultaneamente mais de um núcleo do processador. Entretanto, é possível alterar esse comportamento no emulador mediante passagem de parâmetros na linha de comando.

Três testes foram escolhidos para compor o comparativo: a criação sucessiva de processos Erlang (avaliada pelo tempo de execução e pelo consumo de memória), a execução de ciclos de trocas seqüenciais (anel) de mensagens e o envio e recebimento de mensagens de tamanhos variados. Cada um deles foi executado de três formas distintas: com o interpretador (`escript`), com o emulador (`erl`) em sua configuração padrão e com o emulador configurado para utilizar dois núcleos do processador (opção `-smp enable`). Os códigos Erlang destinados à interpretação e à compilação são, forçosamente, diferentes.

O teste de criação sucessiva de processos Erlang, cujo código-fonte é apresentado no Apêndice K, é caracterizado pela criação seqüencial de quantidades variadas de processos Erlang que simplesmente aguardam o recebimento de uma mensagem. Após a criação de todos os processos, o processo principal envia mensagens individuais a cada processo criado e conclui a execução. Os tempos de execução da versão interpretada (restrita pelo limite padrão de criação de 32.768 processos) e da versão compilada, executada mediante alteração do limite padrão de criação de processos para 65.536, estão na figura 6.7.

Os resultados demonstram que, independentemente da metodologia de execução, o aumento do tempo de execução é praticamente linear à medida que a quantidade de processos Erlang aumenta. Na execução de código interpretado, é possível observar tempos um pouco inferiores aos observados na criação padrão de processos Lua (figura 6.1) e um pouco superiores aos observados na criação de processos Lua sem a carga de bibliotecas padrão (figura 6.3). Na execução de código compilado, por outro lado, os tempos de criação de processos Erlang são consideravelmente inferiores aos observados na criação de processos Lua. No entanto, é preciso considerar que o código Lua usado nos testes foi interpretado.

Aproveitando o ensejo do teste de criação sucessiva de processos Erlang, foi realizada medida de consumo de memória ocasionada pela criação dos processos. O mesmo código-fonte utilizado para medir os tempos de execução da criação de processos foi aproveitado para esse teste, que ocasionou a inserção de retardos (com a função `timer:sleep`) imediatamente após o início da execução do módulo principal e imediatamente após a criação dos processos.

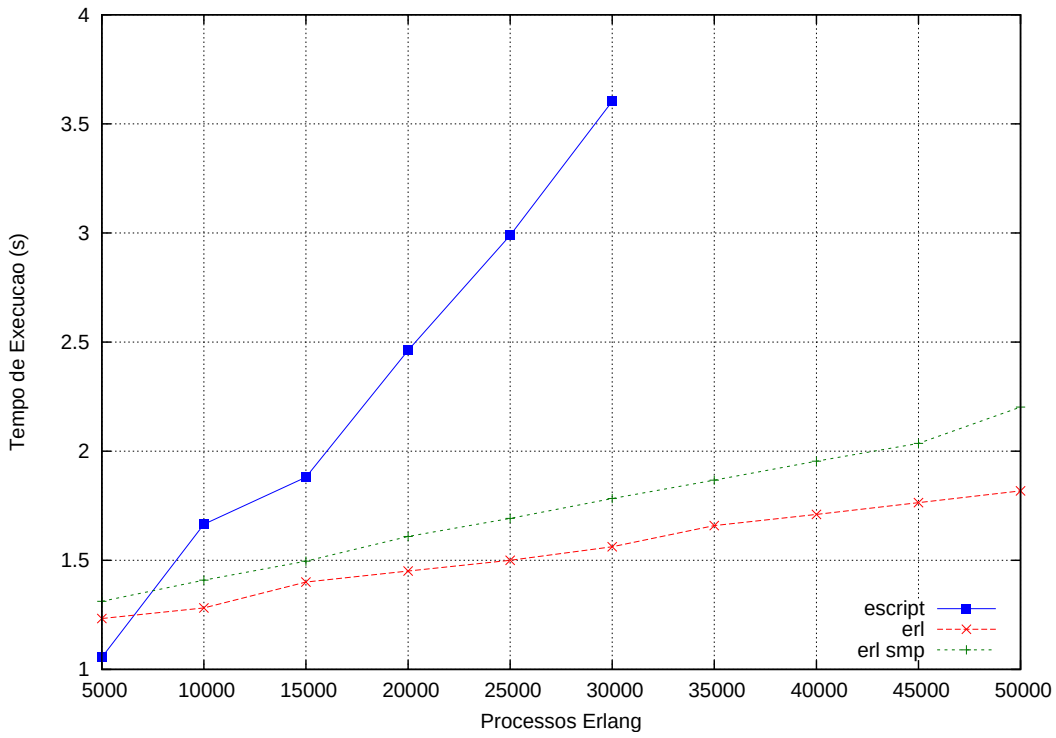

Figura 6.7: Criação sucessiva de processos Erlang.

O consumo de memória em ambos os pontos foi medido a partir do sistema operacional com o comando `pmap`. Os resultados são apresentados nas tabelas 6.8 (versão interpretada), 6.9 e 6.10 (código compilado). Como Erlang não trabalha com canais de comunicação, pois as mensagens são endereçadas diretamente para os processos, é possível considerar que a medida do consumo de memória após a criação dos processos é equivalente à medida do consumo de memória após a criação dos canais e dos processos, na biblioteca para programação concorrente em Lua.

Processos Erlang	Consumo de Memória Inicial (kB)	σ	Consumo de Memória Após Criação dos Processos (kB)	σ
5.000	26.149	2	72.465	1.436
10.000	26.150	2	116.854	2
15.000	26.149	2	161.485	2
20.000	26.149	2	203.633	2
25.000	26.149	2	253.817	2
30.000	26.150	2	298.594	2

Tabela 6.8: Consumos de memória durante a criação sucessiva de processos em Erlang interpretado (escript).

Processos Erlang	Consumo de Memória Inicial (kB)	σ	Consumo de Memória Após Criação dos Processos (kB)	σ
5.000	24.098	2	39.478	2
10.000	24.100	0	49.940	0
15.000	24.098	2	62.990	2
20.000	24.097	2	74.297	2
25.000	24.096	0	86.632	0
30.000	24.097	2	98.969	2
35.000	25.126	2	112.838	2
40.000	25.126	2	125.174	2
45.000	25.124	0	137.508	0
50.000	25.126	2	149.846	2

Tabela 6.9: Consumos de memória durante a criação sucessiva de processos em Erlang compilado (erl).

Processos Erlang	Consumo de Memória Inicial (kB)	σ	Consumo de Memória Após Criação dos Processos (kB)	σ
5.000	57.148	0	70.484	0
10.000	57.149	2	84.017	2
15.000	57.150	2	97.070	2
20.000	57.152	0	108.380	0
25.000	57.150	2	120.714	2
30.000	57.149	2	133.049	2
35.000	58.177	2	146.917	2
40.000	58.177	2	159.253	2
45.000	58.178	2	171.590	2
50.000	58.176	0	183.924	0

Tabela 6.10: Consumos de memória durante a criação sucessiva de processos em Erlang compilado (erl) com suporte a multiprocessamento habilitado.

Os resultados apontam consumo de memória cerca de duas vezes menor, no caso de código Erlang interpretado, e cerca de cinco a seis vezes menor, no caso de código Erlang compilado, em comparação com o consumo de memória

observado durante a criação de processos Lua (tabela 6.4). É possível observar ainda que o crescimento do consumo de memória ocasionado pela criação de novos processos é aproximadamente linear.

O teste de envio e recebimento de mensagens de tamanhos variados, cujo código-fonte é apresentado no Apêndice L, é caracterizado pela criação de dois processos Erlang: um destinado a enviar mensagens e o outro a recebê-las. O conteúdo das mensagens é lido a partir de um arquivo pelo processo responsável pelo envio de mensagens. Um seqüência de envio de mensagens com o mesmo conteúdo é realizada e, após a sua conclusão, o processo responsável pelo recebimento das mensagens notifica o processo principal do final das transmissões para que a execução possa ser concluída. Os tempos de execução da versão interpretada e da versão compilada são apresentados, respectivamente, nas tabelas 6.11 e 6.12.

Os resultados apresentam tempos de execução superiores aos observados na transmissão de mensagens da biblioteca para programação concorrente em Lua (figura 6.1) para mensagens de tamanho igual ou inferior a 104.587 bytes. Mensagens de tamanho superior a 104.587 bytes apresentam tempos de transmissão inferiores em Erlang. A execução de código Erlang interpretado resultou em tempos de execução inferiores aos tempos, praticamente constantes, observados na execução de código Erlang compilado.

O teste de execução sucessiva de trocas seqüenciais (anel) de mensagens, cujo código-fonte é apresentado no Apêndice M, é caracterizado pela criação de uma determinada quantidade de processos Erlang que, essencialmente, recebem e repassam mensagens seqüencialmente. O processo principal é responsável por enviar algumas informações necessárias aos processos criados, introduzir a primeira mensagem no ciclo e aguardar a conclusão de todas as transmissões. Os tempos de execução da versão interpretada e da versão compilada são apresentados, respectivamente, nas tabelas 6.13 e 6.14.

Apesar do aparente bom desempenho generalizado, chama a atenção o tempo de execução, na versão interpretada, superior a uma hora para o anel de mensagens composto por 10.000 processos Erlang e 1.000 ciclos de trocas de mensagens. O tempo de execução, na versão interpretada, do anel de mensagens composto por 10.000 processos Erlang e 10.000 ciclos de trocas de mensagens não é apresentado pois, após decorrida mais de uma hora, é exibida uma mensagem de falta de memória e a execução é abortada.

Investigação preliminar foi conduzida com o objetivo de determinar a causa da demora na execução e determinar a procedência da mensagem de erro observada. Os resultados dessa investigação apontaram que, de fato, a causa mais provável para a demora na execução foi o elevado consumo de memória

Tamanho da Mensagem (bytes)	Envios/ Recebimentos	Tempo de Execução (s)	σ
10	10	0,245	0,111
	100	0,229	0,080
	1.000	0,675	0,211
104	10	0,229	0,081
	100	0,229	0,080
	1.000	0,671	0,204
1.048	10	0,229	0,080
	100	0,229	0,080
	1.000	0,613	0,203
10.485	10	0,178	0,006
	100	0,178	0,005
	1.000	0,532	0,021
104.857	10	0,178	0,006
	100	0,180	0,005
	1.000	0,559	0,012
1.048.576	10	0,192	0,009
	100	0,188	0,002
	1.000	0,575	0,008
10.485.760	10	0,294	0,122
	100	0,226	0,002
	1.000	0,614	0,005

Tabela 6.11: Tempos de execução para envio e recebimento de mensagens de tamanhos variados em Erlang interpretado (escript).

(superior a 3GB, o total disponível no microcomputador) que ocasionou a paginação excessiva e conseqüente degradação do desempenho. Dessa forma, o mais provável é que a execução do anel composto por 10.000 processos Erlang e 10.000 ciclos de trocas de mensagens tenha sido abortada também em decorrência de consumo excessivo de memória.

6.4 Considerações

Os resultados da avaliação oferecem uma visão geral do desempenho da biblioteca, por meio das medidas realizadas para explorar as suas principais

Tamanho da Mensagem (bytes)	Envios/ Recebimentos	Tempo de Execução (s) Simples	σ	Tempo de Execução (s) SMP	σ
10	10	1,207	0,055	1,243	0,000
	100	1,200	0,056	1,242	0,001
	1.000	1,234	0,003	1,247	0,002
104	10	1,231	0,002	1,243	0,002
	100	1,231	0,002	1,242	0,000
	1.000	1,233	0,002	1,247	0,002
1.048	10	1,232	0,002	1,242	0,000
	100	1,232	0,002	1,242	0,000
	1.000	1,234	0,000	1,249	0,003
10.485	10	1,233	0,002	1,243	0,002
	100	1,233	0,002	1,242	0,000
	1.000	1,235	0,002	1,246	0,001
104.857	10	1,233	0,002	1,242	0,001
	100	1,232	0,003	1,242	0,000
	1.000	1,233	0,002	1,246	0,000
1.048.576	10	1,238	0,001	1,246	0,000
	100	1,238	0,000	1,246	0,000
	1.000	1,238	0,001	1,250	0,000
10.485.760	10	1,275	0,004	1,284	0,002
	100	1,274	0,001	1,286	0,001
	1.000	1,276	0,002	1,290	0,000

Tabela 6.12: Tempos de execução para envio e recebimento de mensagens de tamanhos variados em Erlang compilado (erl).

funcionalidades, e um breve comparativo com o desempenho de outras alternativas para programação concorrente. Esses resultados, de maneira geral, demonstraram que a biblioteca desenvolvida para programação concorrente em Lua oferece escalabilidade e desempenho satisfatórios.

Os testes de criação de processos Lua e de carga de bibliotecas padrão nos estados Lua associados aos processos permitiram determinar o impacto de duas características fundamentais da biblioteca: a criação de estados por meio de chamadas à função `luaL_newstate` (em oposição à função `lua_newthread`) e a carga seletiva de bibliotecas padrão de Lua nos estados. Em ambos os testes

Processos Erlang	Ciclos de Troca de Mensagens	Tempo de Execução (s)	σ
10	10	0,378	0,322
	100	0,258	0,100
	1.000	0,637	0,247
	10.000	3,746	0,267
100	10	0,281	0,106
	100	0,688	0,259
	1.000	3,917	0,213
	10.000	37,914	1,093
1.000	10	0,671	0,004
	100	3,832	0,079
	1.000	35,854	0,712
	10.000	508,824	10,561
10.000	10	86,533	2,568
	100	111,400	3,032
	1.000	7.270,296	154,951
	10.000	-	-

Tabela 6.13: Tempos de execução de anel de troca de mensagens em Erlang interpretado (escript).

os resultados demonstraram que o tempo necessário à criação de processos Lua cresce aproximadamente linearmente com a quantidade de processos.

Os testes com a reciclagem de estados Lua durante a criação de processos demonstraram resultados promissores. Ainda que esses testes tenham sido realizados em condições ideais para explorar a funcionalidade de reciclagem, o ganho de desempenho observado chegou a resultar em redução de cinco vezes no tempo total de execução, o que exalta a importância dessa funcionalidade.

A comparação entre o desempenho da criação de processos Lua e o desempenho da criação de *threads* com a biblioteca Native POSIX Threads Library (NPTL), que implementa o padrão POSIX Threads, como esperado, demonstrou tempos de execução inferiores para a NPTL. O desempenho superior resulta, ao menos em parte, da utilização de código compilado e da integração com o *kernel* para criar e gerenciar as *threads*. A comparação com Erlang, também como esperado, demonstrou tempos de execução inferiores para Erlang, com diferença mais acentuada para código Erlang compilado e

Processos Erlang	Ciclos de Troca de Mensagens	Tempo de Execução (s) Simples	σ	Tempo de Execução (s) SMP	σ
10	10	1,167	0,052	1,206	0,068
	100	1,226	0,001	1,238	0,000
	1.000	1,230	0,001	1,259	0,002
	10.000	1,275	0,002	1,478	0,001
100	10	1,227	0,003	1,244	0,001
	100	1,231	0,002	1,268	0,002
	1.000	1,287	0,006	1,503	0,008
	10.000	1,817	0,034	3,298	0,178
1.000	10	1,248	0,002	1,176	0,005
	100	1,380	0,006	1,740	0,010
	1.000	2,586	0,049	4,990	0,459
	10.000	10,776	0,059	30,610	0,713
10.000	10	1,290	0,058	1,842	0,217
	100	2,666	0,119	4,827	0,355
	1.000	10,604	0,340	26,865	0,641
	10.000	96,935	1,865	167,830	0,515

Tabela 6.14: Tempos de execução de anel de troca de mensagens em Erlang compilado (erl).

menos acentuada para código Erlang interpretado. A habilitação de suporte a multiprocessamento (SMP) em Erlang resultou em pequena degradação de desempenho.

Os testes de trocas de mensagens entre processos Lua demonstraram bom desempenho, em particular para a transmissão de mensagens pequenas, cujo conteúdo não exceda cerca de 10kB. A comparação com Erlang, interpretado e compilado, confirmou que os tempos para a transmissão de mensagens desse tipo foram consideravelmente inferiores nos testes com a biblioteca para programação concorrente em Lua, que apresentou desempenho superior para mensagens com conteúdo de até cerca de 100kB. O bom desempenho para a transmissão de mensagens pequenas é um resultado importante pois é bastante comum utilizá-las em sistemas cuja comunicação depende da troca de mensagens, em particular para fins de sincronização.

Ainda com relação à troca de mensagens, os resultados dos testes com

Erlang demonstram tempos de execução praticamente constantes para a transmissão de mensagens de tamanhos variados, enquanto que os resultados dos testes com a biblioteca para programação concorrente em Lua demonstram tempos crescentes. Os tempos constantes sugerem que Erlang utiliza alguma operação que pode ser executada em O(1) para efetivamente transmitir as mensagens. O mais provável é que essa operação seja apenas uma cópia de ponteiros, de modo que os dados enviados por um processo permanecem no mesmo espaço de memória e apenas o ponteiro para esse espaço é repassado ao processo receptor. É provável, portanto, que a implementação da troca de mensagens na biblioteca para programação concorrente em Lua ainda possa ser otimizada mediante análise criteriosa de arquiteturas conhecidas para troca de mensagens (23).

As medidas de consumo de memória reiteraram a boa escalabilidade da biblioteca e evidenciaram a possibilidade de criar até 50.000 processos Lua com consumo inferior a 1GB, bem como o crescimento linear do consumo de acordo, principalmente, com a quantidade de processos. Adicionalmente, as medidas permitiram derivar uma fórmula que possibilita estimar, com boa aproximação, o consumo de memória a partir das quantidades de trabalhadores, canais de comunicação e processos Lua. A comparação com o consumo de memória da biblioteca NPTL foi bastante favorável à biblioteca para programação concorrente em Lua, haja vista que a criação de apenas 5.000 *threads* ocasiona um consumo de cerca de 40GB de memória virtual. A comparação com o consumo de memória de Erlang, por outro lado, evidenciou consumo superior por parte da biblioteca para programação concorrente em Lua.

O teste de exploração do paralelismo, finalmente, demonstrou o potencial da biblioteca de explorar ambientes multiprocessados para execução paralela de processos Lua. Conforme esperado, a execução paralela de uma aplicação de busca de texto apresentou desempenho proporcionalmente superior à execução seqüencial da mesma aplicação. Adicionalmente, a comparação do tempo de execução seqüencial da aplicação com o tempo de execução de uma aplicação similar desenvolvida apenas com as bibliotecas padrão de Lua resultou em diferença irrisória, o que reiterou o bom desempenho da biblioteca.