UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

SPROF:
PERFILADOR DE SPEEDUP EM APLICAÇÕES PARALELAS DE MEMÓRIA COMPARTILHADA

MÁRCIO DE OLIVEIRA JALES COSTA

NATAL-RN
2016

MÁRCIO DE OLIVEIRA JALES COSTA

SPROF:
PERFILADOR DE SPEEDUP EM APLICAÇÕES PARALELAS DE MEMÓRIA COMPARTILHADA

Trabalho de conclusão de curso apresentado ao curso de engenharia de computação da Universidade Federal do Rio Grande do Norte como requisito para a obtenção do diploma de graduação e título de Engenheiro de Computação

Orientador: prof. Samuel Xavier de Souza

NATAL-RN
2016

MÁRCIO DE OLIVEIRA JALES COSTA

SPROF: PERFILADOR DE SPEEDUP EM APLICAÇÕES PARALELAS DE MEMÓRIA COMPARTILHADA

Este trabalho de conclusão de curso foi julgado e aprovado para obtenção do diploma de Engenheiro de Computação, no curso de Engenharia de Computação, do Departamento de Engenharia de Computação e Automação, Universidade Federal do Rio Grande do Norte.

Natal, 09 de dezembro de 2016.

BANCA EXAMINADORA

_____

Prof. Luiz Affonso Henderson Guedes de Oliveira (DCA)

_____

Prof. Wellington Silva de Souza (IMD)

Orientador

_____

Prof. Samuel Xavier de Souza (DCA)

JALES, Márcio de O. **Sprof:** perfilador de *speedup* para aplicações paralelas em memória compartilhada, 2016, 76 f. Trabalho de Conclusão de Curso (Graduação) - Curso de Engenharia de Computação, DCA, Universidade Federal do Rio Grande do Norte, Natal, 2016.

RESUMO

Desde o início da era *multicore*, a escalabilidade de um algoritmo paralelo se tornou, sem dúvida, tão importante quanto sua complexidade computacional. A medição precisa dessa escalabilidade frequentemente requer a coleta de informações de execução. Para a otimização do desempenho dessas aplicações, pode ser necessário não apenas a medição do *speedup* do algoritmo como um todo, mas também de cada trecho paralelizado, de forma a identificar potenciais gargalos de desempenho ou pontos que podem ser otimizados. Portanto, este trabalho apresenta *Sprof*, uma ferramenta de perfilamento de *speedup* em aplicações paralelas de memória compartilhada, a partir de múltiplas execuções do programa, para diferentes valores de *threads* para a aplicação alvo. A ferramenta é escrita em C e suporta sistemas baseados em Linux. Os alvos podem ser implementadas em OpenMP pu Pthreads. Usando apenas três tipos simples de chamadas de função, fornecidas por uma biblioteca, o usuário pode instrumentar o código de modo a marcar quantas partes paralelas ele deseje que sejam perfiladas. Além disso, é fornecido um *script* executável que insere tais funções automaticamente em códigos que usam OpenMP, como uma primeira tentativa de tornar o uso da ferramenta o mais automático possível. O trabalho apresenta, também, resultados da execução de Sprof aplicados a 3 aplicações do *PARSEC Benchmark.*

Palavras-chave: Perfilador, *Speedup*, Escalabilidade paralela.

JALES, Márcio de O. **Sprof:** a speedup profiler for shared-memory parallel applications, 2016, 76 p. Bachelor Thesis (Undergraduate) - Computer Engineering course, DCA, Federal University of Rio Grande do Norte, Natal, 2016.

ABSTRACT

Since the beginning of the multicore era, the parallel scalability of an algorithm has become arguably as important as its computational complexity. The accurate measurement of parallel scalability often requires collecting runtime information. Performance optimization of such applications may require not only speedup measurements of an entire parallel algorithm but also from its individual parts in order to identify potential bottlenecks or optimization hotspots. Therefore, this work presents a tool to profile speedup of shared-memory parallel applications across different runs with different number of threads, named “Sprof”. The tool is written in C and supports Linux-based operating systems. Target applications may be implemented in OpenMP or in Pthreads. Using only three types of function calls provided by a library, the user can annotate the code to specify as many parts of the application as he/she wishes to profile. Besides, it is provided a script that annotates OpenMP codes automatically, using these functions, as a first attempt to make the usage of the tool as automatic as possible. The work presents, as well, results of the tool applied to measure scalability of 3 PARSEC Benchmark applications.

Keywords: Profiler, Speedup, Parallel scalability.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

EOF	*End of file*
GNU	*GNU is not Unix*
HPC	*High Performance Computing*
IPC	*Inter-process Communication*
PARSEC	*Princeton Application Repository for Shared-Memory Computers*
PIC	*Position-independent Code*
PID	*Process Identifier*
POSIX	*Portable Operating System Interface*
SO	Sistema operacional
TCP	*Transmission Control Protoco*l
UDP	*Unix Domain Protocol*

1 INTRODUÇÃO

Na precedente era da computação, onde as máquinas costumavam trabalhar com processadores com núcleo único (*single-core*), a velocidade de execução de uma aplicação era definida, primariamente, pela operação de frequência deste processador[1]. A era mais recente, contudo, baseia-se em sistemas com processadores de múltiplos núcleos (*multi-core*). Assim, o número de núcleos deste componente tornou-se tão importante quanto a sua frequência de operação para atingir alto desempenho, mas não só isso. O fluxo de execução de *softwares* foi intensamente alterado pelas novas arquiteturas paralelas, assim como os métodos e métricas de avaliação de desempenho dessas aplicações. Nesse sentido, o paralelismo, enquanto novo paradigma, fez com que novos desafios aparecessem no que diz respeito à coleta de informações sobre a execução de uma aplicação específica.

O ato de inferir características de um programa em execução consiste em uma técnica chamada **perfilamento**[2]. Ferramentas que a implementam se denominam **perfiladores**. De fato, o propósito em coletar dados é avaliar o desempenho de um programa. Portanto os perfiladores são ferramentas fundamentais para tal.

Uma das métricas essenciais para mensurar o desempenho de aplicações paralelas é o ***speedup***. Seu cálculo consiste em um razão entre o tempo de execução de um programa executado com 1 *thread*/processo e o tempo de execução deste com "n" *threads*/processo. Este cálculo pode ser sobre a execução inteira do programa ou apenas sobre uma determinada região. Desse modo, coletar informações sobre *speedup* (e, consequentemente, sobre o tempo) consiste no primeiro passo para identificar os gargalos de desempenho.

Na abordagem tradicional, a aquisição do tempo de execução das aplicações

---

[1] BARROS, Carlos A., JALES, Márcio de O., SILVEIRA, Luís Felipe Q. e XAVIER-DE-SOUZA, Samuel. **Nor faster nor slower tasks, but less energy hungry and parallel**: simulation results. In: Fourth Berkeley Symposium on Energy Efficient Electronic Systems, Berkeley, California, EUA, outubro 1-2, 2015. DOI=

[2] CODINA, Josep M., GILBERT, Enric, MADRILES, Carlos e MARTÍNEZ, Raúl. **Profiling support for runtime managed code**: next generation performance monitoring unit. In: IEEE Computer Architecture Letters, vol. 14, issue 1, p. 62-65. DOI=

necessita o uso de variáveis declaradas no código, que por sua vez devem ser manipuladas e usadas por funções apropriadas. Quanto mais áreas o usuário deseja marcar, mais variáveis ele declarará e mais chamadas às funções de marcação de tempo serão feitas. A aplicação deve, então, ser executada diversas vezes, com vários números de *threads*/processos diferentes, de modo que ele possa traçar um perfil de como ela se comporta com o aumento do processamento paralelo. Em seguida, os resultados deverão ser usados para calcular o *speedup*, organizados para uma fácil leitura posterior e salvos em arquivos, se desejado.

Este trabalho apresenta um meio automatizado de conduzir tais tarefas, para o código seja instrumentado de maneira mais ágil e, assim, focar simplesmente nos resultados. Esta ferramenta é **Sprof** (*Speedup Profiler*)[3], que visa aplicações *multithread* escritas, principalmente, em OpenMP, que, porém, funciona para *Pthreads*. Ela não é apenas uma aplicação, mais um conjunto delas, formado por um binário executável, que é a aplicação perfiladora, um *script* executável e uma biblioteca.

Sprof possui dois objetivos claros: possibilitar uma plataforma de testes para traçar o perfil de *speedup* de uma aplicação paralela *multithread*; e automatizar a instrumentação do código-fonte desta aplicação.

Além disso, há três requisitos importantes com respeito à implementação de Sprof e a como os objetivos serão alcançados: o uso da ferramenta deve ser simples e minimalista; a curva de aprendizagem deve ser alta de modo que o usuário não demore mais do que uma hora para familiarizar-se; e, por ser uma ferramenta que deve medir tempo, ela deve distorcer a execução da aplicação-alvo o mínimo possível.

O executável binário e a biblioteca procuram alcançar o primeiro deles. Por meio de três funções, das quais uma pode ser dispensável, inseridas no código e disponibilizadas pela biblioteca, a aplicação perfiladora enviará as informações necessárias para que a aplicação-alvo execute automaticamente para os números de *threads* e quantidade de testes que o usuário determinar. Ao mesmo tempo, elas possibilitam que a aplicação-alvo envie as informações pertinentes para o perfilador, como o tempo de execução das regiões paralelas marcadas, quais regiões são

[3] Ferramenta disponível através do repositório: <https://github.com/MarcioJales/Sprof>.

essas etc. Com isso, Sprof poderá calcular o *speedup* da aplicação-alvo e organizar essas informações para que o usuário visualize.

Sobre o segundo objetivo, o *script* é responsável por ele. Por meio da análise de texto no código-fonte, ele o altera, fazendo a inserção automática das chamadas das funções em todas as regiões paralelizadas por OpenMP no código. Apesar de não haver implementação desta solução para Pthreads, a inserção manual das funções pelo usuário é possível e o perfilador funcionará normalmente.

Devido à natureza das aplicações paralelas, que geralmente estão atreladas a computação de alto desempenho, o desenvolvimento de ferramentas de perfilamento é movido por uma obsessão por alta eficiência[4]. Isto normalmente requer alta complexidade técnica. Portanto, elas utilizam uma gama de bibliotecas, módulos e demais utilitários para funcionarem.

Ao contrário deste cenário comum, Sprof visa uma rápida curva de aprendizagem. Seu uso não é complexo e não exige conhecimento aprofundado sobre os conceitos que a envolve. O seu objetivo consiste somente em montar um perfil de execução mais geral possível.

O cumprimento destes objetivos serão bem demonstrados apenas com um método de validação convincente, de modo a comprovar a robustez de Sprof. Nesse sentido, o trabalho foi fundamentado em testes em aplicações do mundo real, não didáticas, providas pelo *PARSEC Benchmark*[5].

Este trabalho está organizado da seguinte maneira: a seção 2 apresenta os conceitos e definições relevantes envolvidos no desenvolvimento da ferramenta; a seção 3 fará uma análise do estado da arte, isto é, uma revisão bibliográfica que situa este trabalho no cenário científico da área; a seção 4 esclarecerá a implementação da ferramenta e seu fluxo de execução, mostrando as técnicas utilizadas; a seção 5 mostrará os resultados atingidos, que incluem os testes e comparação dos resultados para fazer uma medição de “intrusão” da ferramenta. Ao final, a seção 6 fará uma série de considerações finais, que, em sua essência,

---

[4] KNÜPFER, Andreas. et al. **Score-P - A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir.** In: Proceedings 5th Parallel Tools Workshop, Dresden, Alemanha, 2012, p. 79-91.

[5] BIENA, Christian. **Benchmarking Modern Multiprocessors.** Princeton, EUA: Princeton University, 2011, 153 f. Tese (Doutorado em filosofia). Curso de Filosofia, Departamento de Ciência da Computação, Princeton University, Princeton, 2011.

analisa quais objetivos e requisitos foram cumpridos, debate as limitações da ferramenta e as relaciona com propostas de melhoramento para um trabalho subsequente.

2 REFERENCIAL TEÓRICO

Antes de tudo, a compreensão dos elementos a serem citados na revisão bibliográfica e na explicação da criação e funcionamento da ferramenta passa por uma série de definições que devem estar claras para o leitor e usuário. Dessa forma. neste capítulo serão apresentados os principais conceitos sobre processamento paralelo, que se caracterizam como a fundamentação teórica do presente trabalho. Além disso, os apêndice B, C e D apresentam definições complementares sobre outros aspectos utilizados na implementação do programa.

2.1 ELEMENTOS DO PARALELISMO

Em contraste com o programa clássico, que é desenhado para executar em processadores com apenas um núcleo, o programa paralelo tem a capacidade de executar em vários núcleos concorrentemente[6]. Para isso, o fluxo de execução deve ser dividido entre várias entidades relativamente independentes. Elas são as *threads* ou os processos. Cada um deve desempenhar certa tarefa sobre dados e, ao final, os resultados de cada processamento devem se juntar para obter o resultado final do cálculo.

2.1.1 *Threads* x Processos

Ao iniciar um programa que deve tirar proveito de vários núcleos, isto pode ser feito a partir da criação de vários **processos**. Eles são entidades no sistema operacional que representam um programa em execução. Cada processo possui sua região de memória privada e, portanto, eles não conseguem acessar os dados usados por outro, convencionalmente. Assim, caso eles sejam usados para paralelizar um código, o principal fator considerado será a **comunicação** entre os processos. São comumente usados em sistemas com **memória distribuída**, quando

[6] PACHECO, Peter S. Why Parallel Programming?. In:_____ **An Introduction to Parallel Programming.** San Francisco, EUA: Morgan Kauffman Publishers, 1. ed., 2011, cap. 1.

cada processador (ou conjunto deles) possui uma memória ligada somente a eles.

Por outro lado, existem as ***threads***. São conhecidas como “processos leves”, pois possuem uma troca de contexto bem mais rápida do que os processos. Ao contrário destes, elas dividem a maior parte das memórias, possuindo apenas a região de *stack* e registradores de estado privados[7]. Dessa forma, são comumente usadas em sistemas com **memória compartilhada**, quando todos os processadores dividem os endereços de memória entre si. A comunicação entre eles é feita de modo implícito. Com este método, o principal problema a ser considerado é, então, o de **sincronização** entre a execução das *threads*, devido a problemas de coerência de memória e condições de corrida. Nos sistemas computacionais, um processo implementa várias *threads*, ou seja, elas existem dentro de contexto de processos e não existem independentemente nos sistemas.

Para ambos os métodos implementados, o fator de **equilíbrio de carga** é essencial, isto é, que cada entidade de processamento paralelo recebe mais ou menos a mesma quantidade de trabalho, de modo que alguma delas não fiquem paradas por muito tempo, enquanto outras trabalham por tempo demais.

2.1.2 Paralelismo através de processos

O principal padrão de paralelismo que utiliza o processos como entidade fundamental é o MPI (*Message Passing Interface*). É implementada nas linguagens C/C++ e Fortran. É definida a partir de uma biblioteca a nível de usuário com uma série de funções que lidarão com toda a manipulação dos processos que executam concorrentemente, incluindo criação, comunicação e encerramento.

Os sistemas Linux[8] também possibilitam, nativamente, a criação de processos a partir de outros programas, a partir da **chamada de sistema** *fork*. Por ser implementada desta forma, ela trabalha diretamente com o *kernel* dos sistemas Linux e, desse modo, só existe neles. *Fork* clona o processo que a chamou. Este recebe a nomenclatura de *processo pai*, enquanto a réplica chama-se *processo filho*. A comunicação entre eles será disponibilizada, também, por chamadas de sistema

[7] PACHECO, Peter S. Parallel Hardware and Parallel Software. in:_____. *op. cit.,* cap. 2.
[8] KERRISK, Michael. **The Linux Man-Pages Project**. Disponível em: <https://www.kernel.org/doc/man-pages/l>. Acesso em: 28 de nov. de 2016.

e as técnicas que a implementam são conhecidas por *Inter-process communication* (*IPC*).

Há inúmeros métodos implementados pelo *kernel* do Linux que prestam este papel. Alguns deles, inclusive, formam uma região de memória compartilhada entre os processos, o que insere a sincronização como um fator a ser considerado. Como o *kernel* não a faz de modo automático, ela deve ser feita pelo usuário em seu código, a partir da utilização de semáforos ou memória mapeada.[9]

Um dos métodos mais simples denomina-se *pipe*. No caso, dois descritores de arquivos são criados, no qual um é caracterizada como a *ponta de escrita* e o outro é a *ponta de leitura*. O fluxo de dados é unidirecional, isto é, da ponta da escrita para a leitura. Na implementação padrão, usa-se a função *write* para a escrita dos dados e *read* para a leitura, funções que serão explicadas mais adiante.

Um segundo método comum são os *sockets*. É uma abordagem majoritariamente usada para comunicação em redes de computadores. Deste modo, é definida a partir dos protocolos de rede TCP e UDP, cada um possuindo suas características próprias.

2.1.3 Paralelismo através de *threads*

Duas principais *APIs* para o uso de *threads* consistem em *Pthreads* (*POSIX Threads*) e *OpenMP*. Ambas são bibliotecas definidas no nível de usuário. Elas funcionam de acordo com o modelo de execução *fork/join*. Todo programa possui uma *thread* inicial, que corresponde à própria imagem do processo a qual ela está associada. A criação de *threads* novas causam uma bifurcação (*fork*) no fluxo de execução do processo. Ao final da execução de cada *thread* criada, elas se juntam novamente ao fluxo original da *thread* inicial (*join*). A figura 1 ilustra o modelo, com a criação de duas *threads*. A linha horizontal representa tempo.

Pthreads é um padrão de sistemas baseado em UNIX[10]. Possui uma granularidade fina, isto é, o comportamento de cada *thread* pode ser detalhadamente personalizado. Nesse sentido, a figura 1 mostra muito bem o

---

[9] MITCHELL, Mark, OLDHAM, Jeffrey e SAMUEL, Alex. Interprocess Communication. In: _____. **Advanced Linux Programming.** Indianapolis, EUA: New Riders, 2001, cap. 5.

[10] PACHECO, Peter S. Shared-Memory Programming with Pthreads. In:_____. *op. cit.*, cap. 4.

comportamento que pode ocorrer neste padrão, cuja criação e término das *threads* pode ocorrer a qualquer momento. Sua biblioteca está disponível para programas em C, mas podem ser usados em C++, com certas alterações.

Figura 1 - Modelo fork/join para a criação de 2 *threads*

Fonte: Pacheco (2011).

Já o padrão *OpenMP*[11], que é construído sobre o padrão *Pthreads*, possui uma granularidade grossa. Em outra palavras, ele paraleliza blocos inteiros. Várias *threads* são criadas e terminam no mesmo momento. Estes blocos são marcados para paralelização por meio de diretivas *pragma*. Apesar de representar uma facilitação enorme para criação e manipulação de *threads*, esse comportamento impede uma personalização mais aprofundada no trabalho que o usuário pode definir para cada uma separadamente.

2.2 DESEMPENHO DE APLICAÇÕES PARALELAS

Como já mencionado, o *speedup* é, talvez, a primeira métrica a ser coletada quando alguém se propõe a analisar o desempenho de aplicações paralelas. Sua equação básica consiste em:

$$S = T_1 / T_n ,$$

na qual "*S*" é o *speedup*, "T_1" é o tempo de execução para 1 *thread*/processo e T_n é o tempo de execução para "n" *threads*/processos. Caso o *speedup* seja igual

[11] OpenMP Architecture Review Board. **OpenMP Application Programming Interface**. Versão 4.5, 2015. Disponível em: <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>. Acesso em: 28 de nov. de 2016.

ao valor “n”, tem-se o chamado ***speedup* linear**. Quando leva-se em consideração o número de “n” no cálculo, tem-se a **eficiência,** dada pela razão entre o *speedup* e “n”:

$$E = S/n .$$

Embora *speedup* linear seja possível, grande parte das aplicações paralelas possuem um limite quanto à sua escalabilidade e não o atinge. Além dos fatores de sincronização, comunicação e equilíbrio de carga, há trechos de código que são intrinsecamente seriais, isto é, eles não podem ser executados paralelamente e isto é o principal fator limitante de escalabilidade de um aplicação de acordo com a **lei de Amdahl**[12]. Ela prevê que a eficiência de um problema paralelo decairá rapidamente, mesmo com aumento de *speedup* para infinitos número de núcleos processadores.

Apesar disso, esta lei, em nenhum momento, levou em consideração o tamanho do problema que será resolvido. Por outro lado, a **lei de Gustafson** o fez. Nesta nova análise da escalabilidade, percebeu-se que, com o progressivo aumento de complexidade dos programas computacionais, regiões paralelizáveis crescem muito mais do que regiões não-paralelizáveis, fazendo com que o *speedup* atinja valores não previstos pela lei de Amdahl e, assim, a eficiência da aplicação não decaia drasticamente.

Dessa forma, quando a escalabilidade é analisada na forma da lei de Amdahl, ela é conhecida por **escalabilidade forte**, enquanto, na forma da lei de Gustafson, é chamada de **escalabilidade fraca**.

[12] MCCOOL, Michael, REINDERS, James e ROBISON, Arch D. Background. In: _____. **Structured Parallel Programming**: Patterns for Efficient Computation. Waltham, EUA: Morgan Kaufmann, 2012, 1. ed, cap. 2.

3 REVISÃO BIBLIOGRÁFICA

A ação de Sprof sobre o código-fonte se resume a instrumentar para perfilar. Por *instrumentar*, entende-se inserir pontos de prova em um programa de modo que, nestes lugares, junte-se informações para analisá-lo, depurá-lo e testá-lo[13]. A partir dessas marcações, origina-se o processo de perfilamento de uma aplicação. Portanto, instrumentar é parte essencial de várias ferramentas que se propõem.

Os três principais métodos de instrumentação ocorrem em termo de: código-fonte, no qual, como sugere o nome, os pontos são inseridos no código escrito; *bytecode,* que, por este motivo, são instrumentações feitas em linguagens de alto nível, como Java e C#; e binário, que consiste na instrumentação de mais baixo nível, diretamente nas instruções de máquina e divide-se em *dinâmica* e *estática*.

Visto que perfilar significa, em diversos casos, usar instrumentação, várias ferramentas desenvolvidas se utilizarão de alguma das três abordagens para extrair informações de execução e gerar arquivos de resultados. Independente de qual abordagem foi utilizada, instrumentar, no escopo de perfilamento, significa monitorar a execução da aplicação que, em vários casos, representa contar eventos, calcular área de cobertura de código, fluxo e contexto de chamada de funções, uso de memória (principal e *cache*) e, claro, adquirir tempo de execução, entre outros.

Ainda com respeito à análise de desempenho das aplicações, grande parte das ferramentas nesta área, além de perfilarem, *rastreiam* (*tracing*) a execução. Pefilar e rastrear andam juntos, porém têm seus nuances. Enquanto o primeiro geralmente agrega informações sobre uma série de ocorrências de determinado evento, o segundo informa sobre cada evento, individualmente.

Por outro lado, perfilar e rastrear a partir de instrumentação significa agir constantemente sobre a aplicação que está sendo perfilada. Em procedimentos pequenos mas altamente requisitados, isto pode causar um aumento no processamento que, ao final, produzirá informações redundantes. Nesse sentido, o

[13] CHITTIMALI, Pavan K., SHAH, Vipul. **GEMS**: A Generic Model Based Source Code Instrumentation Framework. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, Canadá, abril 17-21, 2012, p. 909-914. DOI=10.1109/ICST.2012.195.

perfilamento através de amostragem se mostra adequado nestes cenários.[14]

3.1 FERRAMENTAS DE INSTRUMENTAÇÃO

3.1.1 Instrumentação no código-fonte

Instrumentar diretamente o código é uma estratégia usada por várias linhas de desenvolvimento e pesquisa reconhecidas na área da análise de desempenho de aplicações paralelas.

O conjunto de ferramentas *Scalasca*[15] é uma delas. É um projeto com o objetivo de analisar programas em MPI, OpenMP e programação híbrida (uma combinação de modelos de programação paralela) em sistemas paralelos de larga escala. Elas têm a capacidade de gerar relatórios com várias informações, que incluem métricas de desempenho para caminhos de chamadas de funções individuais e rastreamento de eventos em tempo de execução. Na maioria dos sistemas, a instrumentação pode ser feita com suporte de compiladores e, portanto, automaticamente.

Outra solução importante neste cenário é a *TAU*[16]. Ela procura criar um conceito abstrato e abrangente de modelos de computação paralela, de modo que a ferramenta atinja uma cobertura de análise maior do que as demais. Dentre outras funcionalidades, *TAU* disponibiliza uma *API* de medição, na qual o usuário pode definir onde agir sobre vários níveis de representação do código (fonte, binário de objeto, binário executável *bytecode* etc), dependendo de onde são definidos os eventos a serem analisados.

É evidente que há uma série de outras soluções que implementam instrumentação no código-fonte, como o *Intel Trace Collector*[17], para aplicações em

---

[14] ADHIANTO, Laksono, FRANCO, Michael, LANDRUM, Reed, MELLOR-CRUMMEY, John, TALLENT, Nathan R. **Scalable Fine-grained Call Path Tracing**. In: ICS '11 International Conference on Supercomputing, Tucson, EUA, maio 31- junho 4, 2011, p. 63-74.

[15] GEIMER, Markus, WOLF, Felix, WYLIE, Brian J., ÁBRAHÁM, Erika, BECKER, Daniel, MOHR, Bernd. **The Scalasca Performance Toolset Architecture**. Concurrency and Computation: Practice and Experience 22, 2010, p. 702–719. DOI=10.1002/cpe.1556.

[16] SHENDE, Sameer S., MALONY, Allen D.. **The TAU Parallel Performance System**, SAGE Publications: Knoxville, EUA. In: International Journal of High Performance Computing Applications 20, 2006, p. 287–331.

[17] **Intel Trace Collector 2017**: User and Reference Guide. Disponível em:

MPI, o *ompP*[18], que age em códigos OpenMP, utilizando uma interface de monitoramento disponibilizada pelo próprio padrão e *gprof*[19], um utilitário do GNU e intrinsecamente integrado ao compilador *gcc*.

3.1.2 Instrumentação no binário

A instrumentação binário de código tem *Valgrind*[20] e *Pin*[21] como uns dos mais bem estabelecidos representantes no cenário científico.

Valgrind consiste em uma *framework* que faz análise dinâmica. Em outra palavras, o código de análise será inserido ao código original da aplicação em tempo de execução. Ela é uma aplicação que trabalha sobre o conceito de *shadow values*, que consiste em uma técnica que "esconde" (*shadows*) os valores de todos os registradores e locais de memória com outros valores que os descrevem. Enquanto um poderosa *framework*, *Valgrind* é capaz de fazer uma série de análises na memória, como *branch-prediction* e detecção de erros, além de detectar erros em *threads*.

Esta ferramenta é constantemente confrontada com *Pin*, uma solução que trabalha sobre o mesmo conceito. Por sua vez, ela também consiste em uma instrumentação dinâmica. Além disso, ambas elas agregam uma série de funcionalidades em comum, como *realocação de registradores*, *inlining* (otimização de chamada e retorno de funções), *escalonamento de instruções* etc.

Em contraste com *Pin* e *Valgrind*, *PEBIL*[22] utiliza uma instrumentação estática,

---

<software.intel.com/sites/default/files/managed/17/61/ITC_User_and_Reference_Guide-2017.pdf>. Acesso em: 29 de nov. de 2016.

[18] FÜRLINGER, Karl, GERNDT, Michael. **ompP**: A Profiling Tool for OpenMP. In: 1st International Workshop of OpenMP (IWOMP), Berlin, Heidelberg: Springer-Verlag, 2005, p. 15-23.

[19] KERRISK, Michael. *op. cit.,* seção 1, gprof(1). Disponível em: <http://man7.org/linux/man-pages/man1/gprof.1.html>. Acesso em: 29 de nov. de 2016.

[20] NETHERCOTE, Nicholas e SEWARD, Julina. **Valgrind**: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 PLDI conference, San Diego, EUA, junho 11-13, 2007. ACM: New York, EUA, 2007, vol. 42, p. 89–100.

[21] LUK, Chi K., COHN, Robert, MUTH, Robert, PATIL, Harish, KLAUSER, Artur, LOWNEY, Geoff, WALLACE, Steven, REDDI, Vijah J. e HAZELWOOD, Kim. **Pin**: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Chicago, EUA, junho 12-15, 2005. ACM: New York, EUA, 2005, p. 190–200.

[22] CARRINGTON, Laura, LAURENZANO, Michael A., SNAVELY, Alan e TIKIR, Mustafa M. **PEBIL**: Efficient static binary instrumentation for Linux. In: 2010 IEEE International Symposium on Performance

ou seja, o código de análise é inserido antes que o programa-alvo execute. Isso claramente poupa poder computacional, visto que várias das tarefas relacionadas à análise não ocorrerão em tempo de execução. Este é seu trunfo em comparação às outras citadas, embora lhe falte funcionalidades, como permitir análise de registradores.

Uma das vantagens mais claras da abordagem de instrumentação binária em comparação à instrumentação no código-fonte é ela não precisa dele para que haja análise.

3.2 FERRAMENTAS DE AMOSTRAGEM

Utilizando-se de um conceito para perfilamento e rastreamento totalmente diferente, há o *HPCToolkit*[23][24]. Esta solução trabalha com uma análise pós-execução da aplicação-alvo. Ela recolhe informações a partir de amostras **assíncronas** sobre certos eventos, através do monitoramento de contadores implementados em *hardware*. Ao final, ela apresenta um resultado estatístico dos dados.

Sua análise integra tanto uma análise no binário quanto no código-fonte, de modo a permitir uma avaliação mais conclusiva sobre um contexto de execução. Um dos principais focos desta abordagem é a escalabilidade do rastreamento das chamadas de funções, que é muito prejudicada quando métodos de instrumentação são utilizados.

3.3 ESCOPO DE SPROF

Com base na categorização exposta, Sprof implementa instrumentação a nível de código-fonte. Porém, várias das funcionalidades descritas nas outras ferramentas não existem na proposta deste trabalho. Na verdade, a única

---

Analysis of Systems & Software, White Plains, USA, março 28-30, 2010, p. 175-183. DOI=10.1109/ISPASS.2010.5452024.

[23] ADHIANTO, Laksono, MELLOR-CRUMMEY, John e TALLENT, Nathan R. **Effectively Presenting Call Path Profiles of Application Performance**. In: 2010 39th International Conference on Parallel Processing Workshops San Diego, EUA, setembro 13-16, 2010, p. 179-188. DOI=10.1109/ICPPW.2010.35

[24] ADHIANTO, Laksono, FRANCO, Michael, LANDRUM, Reed, MELLOR-CRUMMEY, John e TALLENT, Nathan R. *op. cit.*

característica de desempenho analisada por Sprof é o *speedup*. Desse modo, o esforço de desenvolvimento foi canalizado para a simplicidade (tanto de uso quanto técnica) em detrimento da elegância. Com isso, foi possível desenvolver uma solução que atinge os dois objetivos descritos, na qual pouquíssima complexidade transparece para o usuário.

4 MATERIAL E MÉTODOS

Antes de iniciar qualquer tipo de atividade de desenvolvimento, deve-se identificar claramente o escopo de uso da ferramenta, de modo a escolher a linguagem de programação e plataforma de desenvolvimento adequadas para a demanda que pretende-se suprir. Nesse sentido, a linguagem C tornou-se a opção a ser escolhida, devido à natureza da maioria das aplicações paralelas no âmbito científico, que são escritas em C/C++. Outro fato em comum dessas aplicações é a execução em ambientes com sistemas operacionais baseados em Linux. Assim, é sensato escolher esses sistemas como alvo para uso de Sprof.

Com isso em mente, o próximo passo consiste em analisar quais são os métodos mais comuns no paradigma paralelo deste escopo. Visto que a ferramenta destina-se ao uso em aplicações *multithread* executadas em processadores, os modelos de *Pthreads* e *OpenMP* foram os escolhidos por serem modelos comuns de implementação de *threads* e devido à familiaridade com tais *APIs*.

Durante o desenvolvimento do perfilador e todos os elementos associados, foi usado o Ubuntu 16.04 Xenial (1), majoritariamente. Outros sistemas operacionais, como o Linux Mint 16.4 Cinammon (2) e o OpenSUSE 42.1 (3) também foram usados. Isto indica que Sprof deve funcionar em diversas distribuições Linux, inclusive em outras não testadas. A compilação dos códigos em C foram feitas com o g++, versão 5.4.0 em (1) e 6.1.0 em (2) e (3), de modo a buscar uma compatibilidade com aplicações-alvo escritas em C++. Caso o usuário queira compilar a ferramenta novamente, basta executar o *script compile.sh*, disponível no diretório raiz de Sprof.

As próximas subseções detalharão vários aspectos importantes para o entender o funcionamento e desenvolvimento da ferramenta. Assim, nos próximos tópicos abordados, serão discutidos a estrutura da ferramenta (seus elementos e como eles se relacionam), as funções que compõem a biblioteca e seu uso (instrumentação do código da aplicação-alvo), o fluxo de execução, tanto do perfilador quanto no *script* de instrumentação, assim como a estrutura dos respectivos arquivos de configuração.

A título de esclarecimento, esta seção não mostrará os testes feitos com a ferramenta. A próxima seção terá esse fim. Isto pode causar certa confusão inicialmente, mas optou-se por essa divisão de conteúdo pois, de acordo com a interpretação feita, os métodos, no caso deste trabalho, consistem no escopo técnico do trabalho. Ou seja, quais métodos de desenvolvimento para as diversas tarefas propostas foram utilizados para alcançar o objetivo, que é a criação da ferramenta. Os testes mostram o resultado deste processo e, portanto, foram considerados para a próxima seção.

4.1 ESTRUTURA GERAL DA FERRAMENTA

A estrutura de Sprof consiste em três elementos básicos: o código-fonte principal, as bibliotecas e o *script* de automatização da instrumentação do código. Mais especificamente, os arquivos que formam a ferramenta como um todo são:

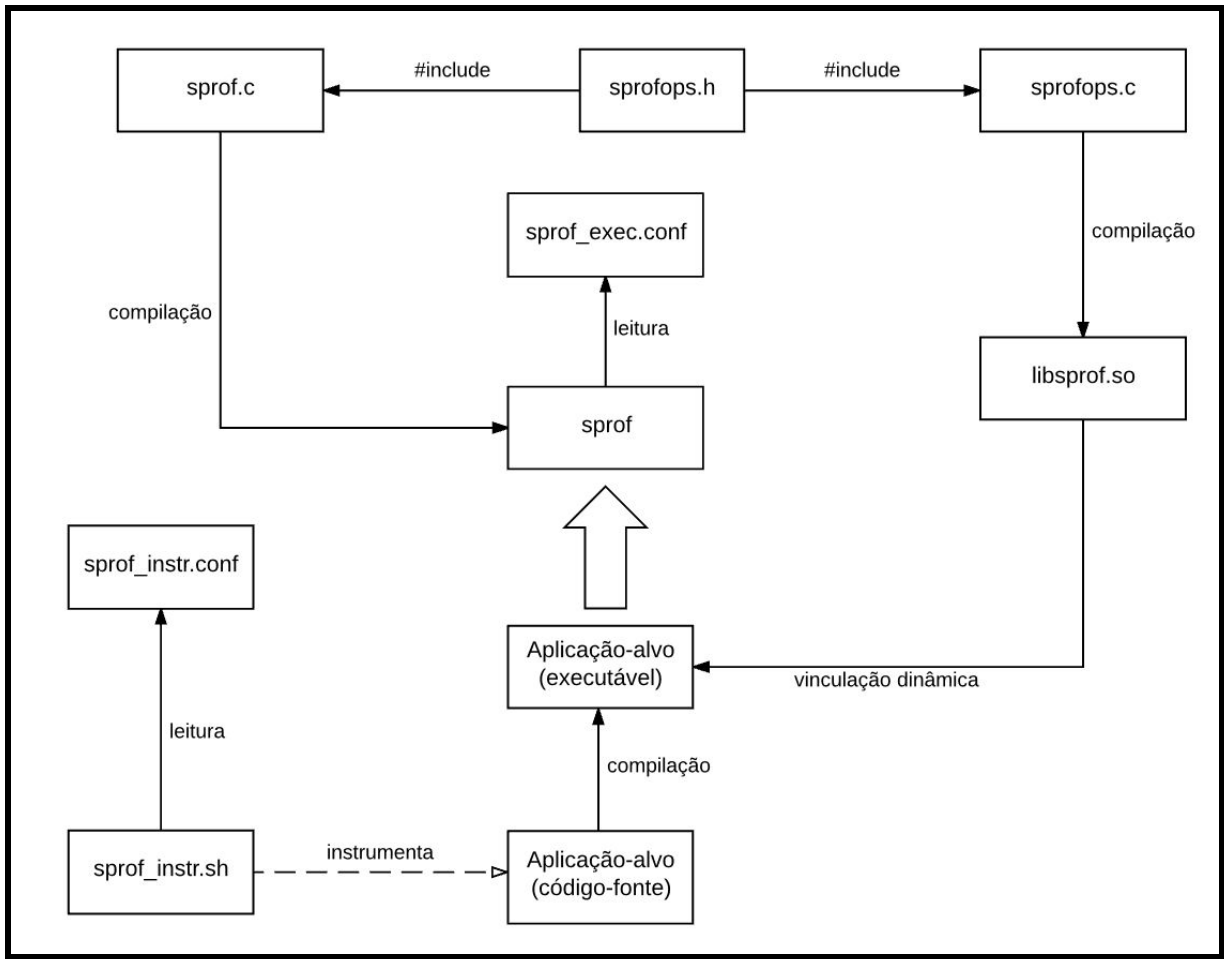
a) dois códigos-fonte (extensão ".c"), **sprof.c** e **sprofops.c**, localizados na pasta *src*;
b) um cabeçalho, chamado **sprofops.h**, localizado na pasta *include*;
c) uma biblioteca dinâmica, **libsprof.so**, localizado na pasta *lib*;
d) um *script* para *bash*, **sprof_instr.sh**, localizado na pasta *etc*;
e) dois arquivos de configuração, **sprof_exec.conf** e **sprof_instr.sh**, ambos contidos na pasta *etc***.**

sprof.c consiste no código-fonte principal, que é compilado e forma o executável do perfilador, **sprof**. A biblioteca **libsprof.so** contém as funções que devem ser usadas para instrumentar o código-fonte das aplicações-alvo, e é formada a partir da compilação de **sprofops.c**. O cabeçalho **sprofops.h** contém os protótipos das funções, além da função que marca o tempo das aplicações. Desse modo, ela é incluída em ambos os códigos-fonte. **sprof_instr.sh** é um *shell script*, arquivo executável que fará a instrumentação automática do código. O uso deste arquivo não é obrigatório, visto que seu resultado final pode ser alcançado igualmente através da inserção manual das funções pertinentes pelo usuário. Por

fim, os dois arquivos de configuração, **sprof_exec.conf** e **sprof_instr.conf** direcionam, respectivamente, a execução do perfilador **sprof** e do *script* **sprof_instr.sh**.

A figura 2 mostra um esquema com as relações entre todos esses arquivos. A linha tracejada indica que a relação entre as duas entidades do gráfico é opcional, ou seja, o uso de sprof_instr.sh não é obrigatório, como já dito anteriormente.

Figura 2 – Estrutura de funcionamento de Sprof.



Fonte: Autor

As próximas subseções detalharão vários aspectos de todos estes arquivos.

4.2 ESTRUTURA E USO DA BIBLIOTECA

Sprof provê, por meio de uma biblioteca, funções que ditam o funcionamento da ferramenta. Elas estão definidas no arquivo sprofops.c e são cinco, das quais três consistem nas principais:

a) **sprof_start** é uma função de marcação indica o início da região paralela que terá seu *speedup* coletado. É uma função que não retorna valor e nem recebe como argumento. Seu uso é obrigatório para o funcionamento correto de Sprof;
b) **sprof_stop** também é uma função de marcação semelhante à função anterior, com a única diferença de marcar o final da região paralela a ser perfilada, em vez do início. Vale salientar que seu uso também é obrigatório, caso deseje-se perfilar a aplicação com sucesso;
c) **sprof_thrnum** é a terceira função mencionada. Diferentemente das duas anteriores, seu uso é dispensado em certos cenários, que serão explicitados adiante. Ela é responsável por passar para a aplicação-alvo o número de *threads* com o qual ela deverá ser executada. Desse modo, um ponteiro para a variável que guarda o valor de *threads* deve ser passado como argumento para esta função.

Estas são as três principais funções que devem ser usadas para a maioria das aquisições. Porém, quando se trata de aplicações escritas com Pthreads, é possível determinar um comportamento individual bem específico para cada *thread*. Pensando nisso, criou-se alternativas para sprof_start e sprof_stop, que são **sprof_pthstart** e **sprof_pthstop**. Com estas funções, é possível perfilar o comportamento individual de cada *thread* criada pela função *pthread_create* e terminada pela função *pthread_join*. Para isso, o identificador da *thread* deve ser passado como argumento para elas. Portanto, elas possuem um argumento do tipo *pthread_t.* Isto se justifica pelo fato de que a criação e término das *threads* podem ser feitas em vários locais, sem uma ordem específica de ocorrência. Dessa forma, as funções de perfilamento precisam saber, de alguma forma, sobre qual *thread* se trata um início e fim de região.

Com o esclarecimento dessas definições, as próximas subseções neste

escopo detalham como elas foram implementadas, como devem ser usadas pelo usuário e como são feitas suas inserções automáticas no código-fonte a partir da ação de sprof_instr.sh, além de outras informações gerais a respeito.

4.2.1 Implementação das funções

Caso o usuário tenha curiosidade em analisar os arquivos *sprofops.c* e *sprofops.h*, ele perceberá que as quatro funções de marcações de início e fim se encontram do modo descrito a seguir:

a) void _sprof_start(int line, const char * filename);
b) void _sprof_stop(int stop_line, const char * filename);
c) void _sprof_pthstart(pthread_t thr, int line, const char * filename);
d) void _sprof_pthstop(pthread_t thr, int stop_line, const char * filename).

Além delas estarem com um subtraço no início do nome das funções, todas elas recebem dois argumentos a mais do que foi especificado. Isto deve-se ao fato de que Sprof identificará as regiões paralelas marcadas pelo arquivo e linhas que elas se encontram. No cabeçalho, é possível ver que há quatro diretivas *define* relacionadas a isto:

a) #define sprof_start() _sprof_start(__LINE__, __FILE__);
b) #define sprof_stop() _sprof_stop(__LINE__, __FILE__);
c) #define sprof_pthstart(thr_id) _sprof_pthstart(thr_id, __LINE__, __FILE__);
d) #define sprof_pthstop(thr_id) _sprof_pthstop(thr_id, __LINE__, __FILE__).

Isto significa que, no momento do pré-processamento da aplicação-alvo, as chamadas das funções serão redefinidas para versões que recebem os argumentos __LINE__ e __FILE__. Estas macros representam, respectivamente, a linha e o arquivo em que elas foram inseridas. Desse modo, como elas estão sendo passadas como argumento das funções, elas armazenam os valores da linha e nome do arquivo em que a função foi chamada. Nada impede que o usuário, ao instrumentar

seu código, chame as funções diretamente com a assinatura completa. Contudo, isto só prejudicaria a curva de aprendizagem da ferramenta. Por isso foi feita as inserções destas diretivas *define* para que o usuário abstraia tal implementação.

4.2.2 Função para marcação do tempo

Todas as funções disponibilizadas ao usuário pela biblioteca fazem a marcação do tempo por meio de uma função local chamada *GET_TIME*. Esta função nada mais é do que uma diretiva *define* que expande esta macro para uma implementação da função *gettimeofday*.

Esta função recebe dois argumentos. O primeiro é uma estrutura que recupera o tempo em duas partes: em segundos e em microsegundos. O segundo também é uma estrutura, que recupera informações sobre fuso horário e, neste trabalho, não é utilizado. Ela retorna 0 em caso de sucesso e -1 em caso de erro.

4.2.3 Inserção das funções pelo usuário

É mandatório que o usuário, ao inserir as funções da biblioteca em um código, posicione-as em locais adequados. Devido ao caráter obrigatório de sprof_start e sprof_stop, comecemos a falar delas.

4.2.3.1 Inserção de sprof_(pth)start e sprof_(pth)stop

Sprof foi uma ferramenta desenvolvida para o uso em aplicações paralelizadas com *Pthreads* e *OpenMP*. Desse modo, os métodos de criação e encerramento das regiões paralelas inerentes às duas *APIs* devem ser levados em consideração no momento da instrumentação. Visto que a região a ser marcada varia de acordo com a vontade do usuário, estas funções podem ser inseridas em quaisquer locais, contanto que não estejam dentro de regiões de paralelismo. Como *sprof_start* marca o início, ela deve ser utilizada sempre antes de sprof_stop. Portanto, para o caso de uso de *Pthreads*, *sprof_start* deve ser inserido em algum momento antes da chamada da função *pthread_create*, e sprof_stop deve estar após

a chamada de *pthread_join*. No caso de *OpenMP*, *sprof_start* seria adequadamente inserido antes da diretiva “pragma omp*”* e *sprof_stop* logo após o fim da região delimitada pela diretiva.

4.2.3.2 Inserção de sprof_thrnum

Como já foi dito, a inserção de *sprof_thrnum* está sujeita a certos cenários. Várias aplicações em ambientes Linux usam argumentos passados por linha de comando com frequência. Deste modo, caso o número de *threads* seja passado para a aplicação-alvo desse jeito, não é necessário o uso desta função, pois sua funcionalidade pode ser atingida através de uma opção de linha de comando permitida por Sprof, que será explicada mais adiante Mesmo assim, se esta opção não for utilizada, será necessário inserir *sprof_thrnum* no código. É aconselhável que, se isto for feito, que seja logo após a linha que, de algum modo, inicializa a variável que carrega o valor de *threads* que será usada no programa eventualmente. Isto também deve ser feito nos programas em que esta variável é *hardcoded*, isto é, definida diretamente no código-fonte.

OpenMP utiliza-se de variáveis de ambiente que controlam a execução da aplicação que o utiliza. Uma dessas variáveis é a OMP_NUM_THREADS. Deste modo, o código não possuirá uma variável explícita que armazena o número de *threads*. Se este for o cenário, o argumento passado para a função deve ser um ponteiro NULL.

4.2.4 Inserção das funções pelo *script*

O *script sprof_instr.sh* possibilita a instrumentação automática de códigos que utilizem OpenMP. Para isso, ele analisa a estrutura de texto nos códigos-fonte e, a partir disso, insere as funções nos locais adequados.

Primeiramente, deve ficar claro que ele não insere as chamadas para a função *sprof_thrnum*. Caso a determinação do número de *threads* a ser usada seja feita diretamente no código, ou seja, impossibilita o uso da opção de linha de comando de Sprof, a tarefa de identificar qual é a variável responsável por

armazenar esse valor é muito complexa. Desse modo, optou-se por não fazer essa inserção automaticamente. O usuário, então, deve inseri-la manualmente.

Portanto, *sprof_instr.sh* responsabiliza-se por inserir as funções *sprof_start* e *sprof_stop* em **todas** as áreas paralelas que ele identificar. Na versão atual, o *script* procura por áreas marcadas por "# pragma omp paralell" e, desse modo, insere uma chamada para *sprof_start* na linha logo acima e sprof_stop na linha logo abaixo do fim desta área (logo abaixo da chave "}" que determina seu fim).

4.3 OS ARQUIVO DE CONFIGURAÇÃO

Os arquivos de configuração disponibilizados são um meio que possibilita ao usuário personalizar a execução do perfilador e do *script* de instrumentação. Cada um possui suas peculiaridades e serão explicados a seguir.

4.3.1 O arquivo sprof_exec.conf

Antes de qualquer execução de uma aplicação-alvo, é necessário passar informações essenciais para o perfilador de modo que ele execute (ele mesmo e a aplicação a ser perfilada) adequadamente. De acordo com o que foi desenvolvido, o perfilador precisa que o usuário o informe sobre 2 elementos básicos: o número de testes que serão feitos e a lista de valores de *threads* com quais ela executará. **sprof_exec.conf** é o arquivo designado para esta finalidade.

A sintaxe do arquivo exige que não haja espaço nas linhas que determinam os valores que o perfilador lerá. Portanto, elas devem estar na forma <*nome da variável*>=<*valor*>. Os comentários são identificados por um "#".

A primeira linha contém a variável *number_of_tests*. Um valor tem de ser atribuído obrigatoriamente a ela. Um *teste* significa que a aplicação-alvo foi executado para todos os valores de *thread* desejados. Isto é, se o usuário determinar que ele quer uma execução com 2, 4 e 8 *threads* (a execução com 1 sempre será feita, pois só assim o *speedup* poderá ser calculado), um *teste* refere-se à execução da aplicação para todos esses valores.

Em seguida, há a variável *list_threads_values*. Ela conterá uma lista com os

valores de *threads* a serem usados (excluindo o 1). A lista deve estar entre chaves e os valores separados por vírgula (e tudo sem espaços, como já mencionado). Um exemplo, então, de sintaxe válida é a declaração *list_threads_values={2,4,7,11,16}*. Ela pode ser vazia (com um par de chaves sem nada no meio ou simplesmente sem as chaves).

As três próximas variáveis atuam em conjunto. São elas: *max_number_threads*, *type_of_step* e *value_of_step*, Elas só precisam ser usadas caso *list_threads_values* não seja especificada e não possuem precedência quanto a ela. Em outra palavras, caso *list_threads_values* seja usada, estas três variáveis não serão lidas.

Voltando às definições das três, *max_number_threads* determina o número máximo de *threads* com qual a aplicação executará. *type_of_step* informa como o número de *threads* será incrementado, a partir do valor 1. Ela só pode assumir dois valores: *power* e *constant*. *Power* é opção para um incremento em progressão geométrica, enquanto *constant* incrementa o valor em progressão aritmética. Por fim, *value_of_step* determina o valor do incremento. A figura 3 mostra como *sprof_exec.conf* se apresenta.

Figura 3 - Arquivo de configuração *sprof_exec.conf*

```
#
# Configuration file of Sprof execution.
#
#

# Number of tests on the target application.

number_of_tests=10

# List of number of threads to be executed.
# The numbers must be comma-separeted, without spaces.
# The execution with 1 thread is mandatory and must not be specified.

list_threads_values={}

# Maximum number of threads to be executed
# Used only if the list_threads_values is not defined

max_number_threads=4

# Step of execution. "type_of_step" may be the values "constant" and "power".
#"value_of_step" is the step value.
# If "constant" is defined, it will occurr the adittion of the "value_of_step" on the number of threads.
# Else, a "value_of_step" power will be applied.
# Used only if the list_threads_values is not defined

type_of_step=power
value_of_step=2
```

Fonte: Autor

Um série de situações hipotéticas ilustram como estes três valores atuam em conjunto:

a) max_number_threads=16, type_of_step=constant e value_of_step=3. A aplicação-alvo executará com 1, 4, 7, 10, 13 e 16 *threads*;
b) max_number_threads=32, type_of_step=power e value_of_step=2. A aplicação-alvo executará com 1, 2, 4, 8, 16 e 32 *threads*;
c) max_number_threads=18, type_of_step=constant e value_of_step=5. A aplicação-alvo executará com 1, 5, 10 e 15 *threads* (Não será executado com 18, já que o valor de incremento excederá o máximo determinado).

4.3.2 O arquivo sprof_instr.conf

A figura 4 exemplifica a estrutura do arquivo:

Figura 4 - Arquivo de configuração *sprof_instr.conf*.

```
#
# Configuration file of automatic source code instrumentation
#
#

# Folder path where is located the source code

DPATH=/home/marciojales/Downloads/parsec-3.0/pkgs/apps/bodytrack/src

# Which file extensions must be parsed. They must be separated by spaces

EXTENSIONS=.cpp,.h
```

Fonte: autor

A configuração para a execução do *script sprof_instr.sh* é mais suscinta. O usuário necessita apenas informar qual é o diretório raiz em que os códigos-fonte da aplicação-alvo se encontram e quais extensões de arquivo devem ser verificadas para instrumentação. Para a primeira finalidade, o nome da variável no arquivo de configuração se chama *DPATH*, enquanto, para a segunda, a variável é *EXTENSIONS*.

Há menos exigências quanto à sintaxe deste arquivo. O nome das duas variáveis devem estar em início da linhas. Não deve haver espaços nestas linhas. A lista de extensões que devem ser procuradas para instrumentação deve ter seus elementos separados por vírgulas.

4.4 FLUXO DE EXECUÇÃO DO PERFILADOR

Visto que as informações pertinentes do arquivo de configuração foram passadas, já é possível entender o fluxo de Sprof por completo.

O principal elemento da ferramenta consiste no perfilador, executável chamado *sprof*, contido dentro da pasta *bin* e formado a partir da compilação de *sprof.c*. A figura 5 mostra, em pseudocódigo, como este arquivo se caracteriza, enquanto que o apêndice A mostra como é a execução real de Sprof.

Figura 5 – Pseudocódigo do perfilador.

```
main()
{
    LÊ sprof_exec.conf;
    CRIA o arquivo de resultados;
    ANALISA a linha de comando;

    ENQUANTO (número da execução atual < número total de execuções)
    {
        ENQUANTO (número atual de threads de execução < número máximo)
        {
            PRÉ-CONFIGURA a execução do alvo;
            CRIA comunicação inter-processo;
            PRÉ-CONFIGURA a execução do alvo;
            CRIA o processo filho;
            MARCA tempo inicial;
            EXECUTA a aplicação-alvo no processo filho;
            ESPERA a conclusão do processo filho;
            MARCA tempo final;
            ESCREVE no arquivo de resultados;
        }
    }

    FIM;
}
```

Fonte: Autor

Com base nisso, as próximas subseções detalharão cada elemento discriminado neste pseudocódigo.

4.4.1 Executando Sprof

Antes de tudo, o usuário precisa saber de quais modos Sprof pode ser executado. Basicamente, há dois: utilizando a opção "-t" na linha de comando; e não utilizando a opção na linha de comando. De qualquer modo, invoca-se o perfilador apenas pela linha de comando em um terminal.

A opção "-t" será responsável por determinar qual argumento da aplicação-alvo recebe o valor de *threads*, de modo que o usuário não precise instrumentar o código com a função *sprof_thrnum*. Portanto, considerando uma aplicação com nome "app" e que recebe quatro argumentos, dos quais o segundo será o número de *threads* com qual ela executará, têm-se dois cenários possíveis:

a) <caminho para Sprof> -t 2 <caminho para app> <arg1> <arg2> <arg3> <arg4>;
b) <caminho para Sprof> <caminho para app> <arg1> <arg2> <arg3> <arg4>.

No primeiro modo, o código não necessitará instrumentação com *sprof_thrnum*, enquanto no segundo é obrigatório para o funcionamento adequado. Note que, no primeiro caso, <arg2> poderá ser **qualquer coisa** (número, letra, caractere especial etc), pois eles será substituído dentro do perfilador, antes da execução da aplicação-alvo. Já no segundo caso, <arg2> não pode ser qualquer coisa, mas pode receber **qualquer número inteiro**, já que ele será substituído dentro da aplicação-alvo, com o uso de *sprof_thrnum*.

4.4.2 Leitura do arquivo de configuração sprof_exec.conf

O primeiro passo dentro do programa é a leitura da configuração passada pelo usuário. Esta funcionalidade é implantada majoritariamente pela função **exec_conf**. Ela não retorna qualquer valor e recebe quatro ponteiros como

argumentos, dos quais três são para inteiros e o outro é uma *string* (*char *)*.

O primeiro valor receberá o número de execuções que serão realizadas. Portanto, receberá o valor de *number_of_tests*, em sprof_exec.conf. O segundo argumento refere-se ao número máximo de *threads*. Este valor será determinado de modos distintos, dependendo de como os valores de *threads* estão definidos na configuração. Por fim, o terceiro argumento passado é o caminho do executável de Sprof, isto é, "argv[0]".

A primeira medida tomada por *exec_conf* é, evidentemente, abrir o arquivo de configuração. Para isso, é necessário que se conheça onde ele está localizado. Visto que *sprof* encontra-se na pasta *bin* e *sprof_exec.conf* está na pasta *etc*, sabe-se exatamente como achá-lo, Portanto, foi implementada uma função auxiliar, denominada *get_path*, que, a partir do argumento "char * exec_path", determinará o caminho do arquivo de configuração. Esta função auxilia o perfilador na procura tanto deste arquivo quanto da pasta em que os resultados serão escritos (explicado mais adiante). Feito isso, inicia-se sua análise.

Para que as informações sejam extraídas corretamente, o perfilador deve identificar: as regiões de comentário; se alguma variável obrigatória não foi declarada ou atribuída; se os valores atribuídos são válidos; e se a sintaxe, como foi determinada anteriormente, está correta.

A análise consiste, basicamente, em ler o arquivo aberto com a função *fscanf*, atribuir a uma variável e comparar o resultado com padrões pré-determinados, a partir do uso de *strncmp* e *strcmp*. Estes padrões são as variáveis de configuração, usadas com a sintaxe correta.

4.4.2.1 Identificando os comentários

Caso a linha se inicie por um símbolo de comentário (#), nada será feito e a próxima cadeia de caracteres será lida, até que se encontre um início de linha sem "#". Quando isto ocorrer, deve-se, obrigatoriamente, identificar se os caracteres encontrados correspondem a umas das variáveis ou se é qualquer outra coisa. Caso não seja, um erro será retornado e a execução do programa cessará.

4.4.2.2 Identificando as variáveis de configuração

Dentre as cinco variáveis em sprof_exec.conf, três têm seus valores obtidos de forma semelhante: *number_of_tests*, *max_number_threads* e *value_of_step*, pois todas consistem em um único valor inteiro. No caso delas, a posição atual no arquivo é retrocedida em 2 posições, usando-se *fseek*. Apoś, lê-se o próximo caractere, com *fgetc*. Caso ele seja um símbolo de igual (=), então a *string* que vem a seguir será, obrigatoriamente, o valor que deve ser atribuído àquela variável. Este valor é, em seguida, atribuído, a partir de *fscanf,* a uma variável do programa para uso futuro. É importante notar que, caso o usuário atribua um valor que não seja inteiro, o comportamento do perfilador será indeterminado e nenhuma mensagem de erro foi configurada para isso, já que, em sua implementação, ele espera por um valor inteiro.

Já para o caso de *type_of_step*, só há apenas dois valores permitidos (*power* e *constant)* e, por isso, eles devem ser verificados. O esquema com o uso de *fseek* e *fgetc* é o mesmo descrito anteriormente. A diferença vem do fato que, logo após encontrar o caractere "=", é necessário analisar se eles equivalem exatamente às cadeias *power* ou *constant.*

O último caso é o de *list_threads_values*, mais delicado do que os anteriores, pois a sintaxe envolvida é mais restrita. Primeiramente, identifica-se se a chave final "}" está presente na posição final da variável que recebe o resultado da leitura do arquivo, usando-se a função *strlen*. Com isso, usa-se *fseek* e *fgetc* para encontrar o sinal de igual e *fgetc* mais uma vez para identificar se o próximo caractere corresponde a um "{".

4.4.2.4 Leitura dos valores de *threads*

Independente de como os valores de *threads* serão lidos do arquivo, o armazenamento deles no programa será feito do mesmo modo (em uma *array* de inteiros). Devido a natureza distinta entre os dois modos que o usuário pode determiná-los, o tratamento de ambos serão diferentes, consequentemente.

Para o caso da determinação a partir de *list_threads_values*, um esquema

com o uso das funções *strtok* e *sscanf* foi elaborado. Os valores serão identificados utilizando-se a vírgula como delimitador na *string* lida. Isto justifica o motivo da sintaxe exigir vírgulas na separação dos valores. O outro caso refere-se ao conjunto dos valores em *max_number_threads*, *value_of_step* e *type_of_step*. Neste modelo, será feito simplesmente um incremento, a partir do valor "1", multiplicando-se ou adicionando-se o valor anterior pelo passo, de acordo com as configurações passadas.

4.4.3 Criação do arquivo de resultados

Sprof gerará resultados de execução em arquivos em formato texto e com padrão de nome "dia-mês-ano-hora-minuto-segundo" (por exemplo, "17-09-2016-15h56m33s"). Para buscar essas informações de tempo, a função *localtime* é utilizada. Ela retorna uma estrutura (*struct tm*) que contém todas elas. Este padrão é, então, escrito na variável (com *sprintf*) que armazena o nome do arquivo que, por sua vez, será usado na função *fopen* para abrir o arquivo mais adiante.

Mais uma vez, a função local *get_path* é chamada, agora para determinar o diretório em qual o arquivo de resultados será criado. É necessário, então, concatenar (com a função *strcat*) o nome do arquivo ao caminho até este diretório, de modo a formar o caminho completo no sistema.

4.4.4 Análise da linha de comando

Aplicações-alvo que recebam o número de *threads* para execução através da sua linha de comando não precisarão usar a função *sprof_thrnum* da biblioteca. Isso é possível caso o usuário passe a opção "-t" na execução de Sprof, seguido da posição, nos argumentos da aplicação-alvo, em que esse parâmetro é passado.

Para determinar se isso foi feito, o segundo argumento de Sprof (*argv[1]*) é analisado para descobrir se ele corresponde a "-t". Este fato não é importante apenas para identificar a posição em que o número de *threads* será passado para a aplicação-alvo. Para executá-la mais adiante, os argumentos pertinentes serão

passados para um *array* de *strings* auxiliar. Isto é, os argumentos recebido em *argv* serão copiados para outra estrutura semelhante (alocada dinamicamente). Portanto, a presença da opção "-t" é essencial para determinar a lógica usada para fazer essa cópia.

Considerando "-t" está presente, o terceiro argumento (*argv[2]*) contém, portanto, a posição em que o número de *threads* deve ser passado. Este valor será salvo em uma variável de tipo inteiro o momento em que a aplicação-alvo for chamada para execução.

4.4.5 Pré-configurações para a execução da aplicação-alvo

A figura 5 determina que, em dois momentos, uma pré-configuração será feita. Esta ação refere-se à chamada da função local **set_profcfg**. Ela recebe dois inteiros. O primeiro refere-se ao valor que será passado ao ambiente do processo filho (a aplicação-alvo), a segunda indica de que tipo é este valor: SET_THREADS ou SET_PIPE. A primeira opção diz que o valor passado no primeiro argumento será o número de *threads* com o qual a aplicação-alvo deverá executar. A segunda diz respeito ao descritor de arquivo do lado de escrita da *pipe*, cujo uso é necessário na aplicação-alvo.

Dentro da função, o que será feito é a criação de **variáveis de ambiente** que armazenarão o valor passado no primeiro argumento. Portanto, a função *snprintf* foi usada para transformar esse valor em uma *string*. Esta é passada, então, na função *setenv* de modo a atribuir os valores às variáveis de ambiente. Desse modo, como um processo filho herda toda a configuração de ambiente do pai, estas variáveis estarão disponíveis para a aplicação-alvo. Com SET_THREADS, permite-se sua execução com um novo valor de *threads* e com SET_PIPE, o envio dos resultados para Sprof, que está com a ponta de leitura da *pipe*, é possível.

4.4.6 Criação da IPC e execução da aplicação-alvo

Antes da pré-configuração com SET_PIPE, é óbvio que a comunicação inter-processo deve ser criada antes disso. Nesta implementação, decidiu-se pelo

uso de *pipes*, devido, principalmente, à sua simplicidade. Como há uma baixa demanda por troca de informações, não haveria necessidade de um solução como *sockets* e o uso de memória compartilhada, como já discutido, não se mostra adequado para IPC.

A criação das *pipes* ocorre com a chamada da função *pipe.* Ela recebe um *array* de inteiros de tamanho dois e retorna um descritor de arquivos para cada posição. A primeira posição será a ponta da leitura, que ficará com Sprof. A segunda posição será a ponta da escrita, que será enviada à aplicação-alvo, por meio de *set_profcfg*.

Em seguida, o processo filho será criado com a chamada de *fork.* No escopo do processo pai, ela retornará o *PID* do novo processo criado, enquanto neste (o filho), o retorno será o valor zero. Assim, é possível, a partir de uma estrutura de condição, diferenciar códigos que devem ser executados por um processo ou o outro.

Logo após, o tempo de início é marcado, através da função local *GET_TIME*, que ainda será explicada adiante.

No escopo do processo filho, a função *execv* será chamada de modo a substituir toda a imagem do processo atual (um clone se Sprof, processo pai) por um novo processo a ser executado. Este será justamente a aplicação-alvo. Aqui, o uso da opção "-t" na linha de comando terá influência. Caso ela tenha sido usada, o argumento pertinente, no *array* auxiliar criado, terá o valor de *threads* a ser executado atualmente atribuído. O primeiro argumento em *execv* será o caminho para o executável (a aplicação-alvo), enquanto o segundo deve ser um *array* de *strings,* correspondente aos argumentos da aplicação, terminado com um ponteiro para *NULL*.

Enquanto isso, o processo pai, por meio da função *waitpid*, esperará a conclusão do processo filho para marcar o tempo final, também com *GET_TIME*, e prosseguir com a exibição dos resultados. Enquanto ele espera, receberá os valores enviados pelo processo filho, através da função *read,* passando, como argumentos, o descritor de arquivos da *pipe* de leitura e o ponteiro para a estrutura que armazenará os dados.

4.4.7 O lado do processo filho

Antes da exibição dos resultados, é essencial fazer um desvio na explicação do fluxo de execução para explicar tudo que será feito no processo filho e é pertinente a Sprof.

Como já mencionado, o processo pai ficará parado após a chamada de *waitpid*. Neste tempo, a aplicação-alvo executa normalmente até o momento em que ela encontra uma função da biblioteca de Sprof.

4.4.7.1 Execução de *sprof_thrnum*

Caso a função sprof_thrnum esteja presente, a atualização do valor de *threads* que a aplicação-alvo usará será feito durante sua execução. O único trabalho que esta função terá é recuperar o valor de *threads* que o processo pai, usando *set_profcfg*, passou através de uma variável de ambiente. Isto será feito chamando-se a função *getenv* e passar esse resultado para a variável recebida pelo argumento de *sprof_thrnum*. Caso o valor passado seja *NULL*, considera-se que, para casos de aplicações-alvo em OpenMP, a variável de ambiente OMP_NUM_THREADS está sendo a fonte do número de *threads.*

4.4.7.2 Execução de *sprof_start* e *sprof_stop*

Em *sprof_start* é onde se define de qual região marcada se trata. Primeiramente, caso a função esteja sendo chamada pela primeira vez, ela chamará uma função local de nome *setconfig*, cujo papel é recuperar o valor da *pipe* que foi passada para o processo filho por uma variável de ambiente.

Voltando a *sprof_start*, esta função recebe a informação sobre qual linha, no código-fonte da aplicação-alvo, se encontra sua chamada. Com isso, ela atribui um número para aquela região e salva o valor da linha de início, pois estas informações serão escritas nos arquivos de resultado. Após atribuído, marca-se o tempo de início com a função *GET_TIME*. Caso *sprof_start* esteja sendo chamada novamente no mesmo local, então ela simplesmente marca o tempo de início, já que ela é capaz de

identificar que um número já foi atribuído àquela região.

Em *sprof_stop*, primeiramente chama-se *GET_TIME* para marcar o tempo final da região. Com o número que foi atribuído a esta em *sprof_start*, é possível identificar a correspondência do tempo final marcado com algum tempo inicial armazenado naquela função. Faz-se, então, o cálculo do tempo total decorrido para a região paralela. *sprof_stop* é responsável, também, por enviar as informações coletadas e pertinentes para escrita nos resultados ao processo pai, que são: tempo, linha de início da região, linha de fim da região, número atribuído a esta e o nome do arquivo em que ela está. Tal tarefa é feita criando-se uma estrutura (*struct*) que armazenará todos esses valores e passando-a para a função *write.*

4.4.7.3 Execução de *sprof_pthstart* e *sprof_pthstop*

Ambas as funções trabalharão de modo semelhante à *sprof_start* e *sprof_stop*. A lógica utilizada nestas considera que uma chamada de *sprof_stop* virá obrigatoriamente após uma de *sprof_start*. Porém, tal fato não é necessariamente verdadeiro caso esteja-se instrumentando *threads* individuais com Pthreads. Assim, *sprof_pthstart* armazena, em um *array*, o identificador da *thread* que a chamou. Em *sprof_stop*, com o valor do identificador da *thread* recebida, é verificado se há um identificador correspondente no *array* para, então, calcular o tempo decorrido corretamente.

4.4.8 Escrita no arquivo de resultados

Uma função local foi designada para receber os resultados da aplicação-alvo e imprimí-los no arquivo de saída correspondente. Ela se chama *time_information* e recebe oito argumentos. Eles são, respectivamente, um vetor que armazena o valor de tempo de cada região instrumentada para 1 *thread,* o número de *threads* na execução atual, tempos de fim e início de execução da aplicação-alvo, a variável do tipo FILE que corresponde ao arquivo de resultados*,* o número de teste atual, os argumentos da linha de comando da aplicação-alvo e a quantidade total de argumentos de Sprof.

Nesta função, o cálculo do *speedup* é realizado e várias manipulações gerais são feitas com o intuito de formatar o arquivo de resultados com todas as informações pertinentes. A figura 6 mostra um exemplo de organização do arquivo.

Figura 6 - Exemplo de um arquivo de resultados.

```
-----> Execution number 1 for ./targets/bodytrack:

   --> Result for 1 threads, application ./targets/bodytrack, arguments: /home/marcio/sequenceB_261, 4, 100, 6000, 12, 3, 1,

       Parallel execution time of the region 1, lines 108 to 120 on file ParticleFilterOMP.h: 3.718798 seconds
       Speedup for the parallel region 1: 1.000000

       Parallel execution time of the region 2, lines 44 to 58 on file ParticleFilterOMP.h: 4.320774 seconds
       Speedup for the parallel region 2: 1.000000

       Parallel execution time of the region 3, lines 73 to 87 on file ParticleFilterOMP.h: 4.240621 seconds
       Speedup for the parallel region 3: 1.000000

       Parallel execution time of the region 4, lines 68 to 76 on file ParticleFilterOMP.h: 485.738586 seconds
       Speedup for the parallel region 4: 1.000000

       Parallel execution time of the region 5, lines 114 to 120 on file ParticleFilterOMP.h: 43.485641 seconds
       Speedup for the parallel region 5: 1.000000

       Total time of execution: 559.486284 seconds
       Speedup for the entire application: 1.000000

   --> Result for 2 threads, application ./targets/bodytrack, arguments: /home/marcio/sequenceB_261, 4, 100, 6000, 12, 3, 1,

       Parallel execution time of the region 1, lines 108 to 120 on file ParticleFilterOMP.h: 1.888891 seconds
       Speedup for the parallel region 1: 1.968773

       Parallel execution time of the region 2, lines 44 to 58 on file ParticleFilterOMP.h: 2.183974 seconds
       Speedup for the parallel region 2: 1.978400

       Parallel execution time of the region 3, lines 73 to 87 on file ParticleFilterOMP.h: 2.140480 seconds
       Speedup for the parallel region 3: 1.981154

       Parallel execution time of the region 4, lines 68 to 76 on file ParticleFilterOMP.h: 247.850769 seconds
       Speedup for the parallel region 4: 1.959803
```

Fonte: Autor.

4.5 FLUXO DE EXECUÇÃO DO *SCRIPT* DE INSTRUMENTAÇÃO

O *script sprof_instr.sh* consiste em uma primeira tentativa de instrumentação automática da aplicação-alvo, a nível de código-fonte. Desse modo, esse executável visa atingir o objetivo secundário da ferramenta.

Semelhante ao que foi feito para o fluxo do perfilador, a figura 7 mostra um pseudocódigo também para o fluxo de *sprof_instr.sh*.

O apêndice A mostra ilustrações do funcionamento real do script.

4.5.1 Executando sprof_instr.sh

A execução do *script* ocorre de maneira simples. Há dois tipos de casos contemplados pelo executável: instrumentação de código OpenMP; e limpeza de instrumentação. Com respeito ao segundo, esta funcionalidade foi inserida de modo que, caso o usuário queira retirar tudo relacionado a Sprof do seu código, isto possa ser desempenhado de modo rápido.

Portanto, os dois modos de uso do *script* se resume a:

a) <caminho para sprof_instr.sh> openmp;
b) <caminho para sprof_instr.sh> clean.

Figura 7 – Pseudocódigo do *script* de instrumentação.

```
LÊ sprof_instr.conf;
ENQUANTO houver extensões
{
    PESQUISA quantos arquivos há com certa extensão
    ENQUANTO houver arquivos
    {
        IDENTIFICA nome do arquivo;
        CASO instrumentar openmp
        {
            IDENTIFICA áreas paralelizadas;
            SE existirem áreas
            {
                INCLUI o cabeçalho de Sprof;
                ENQUANTO houver regiões
                {
                    IDENTIFICA os comentários;
                    INCLUI sprof_start();
                    INCLUI sprof_stop();
                }
            }

        }
        CASO limpeza da instrumentação
        {
            APAGA o cabeçalho e as chamadas das funções;
        }
    }
}
```

Fonte: Autor

4.5.2 Leitura de sprof_instr.conf

A leitura do arquivo de configuração consiste em determinar os valores dados para as variáveis *DPATH* e *EXTENSIONS*. É usado, desse modo, o comando *perl* com um *pipe* para o comando *cut*. *Perl* identificará, respectivamente, as linhas com os padrões "DPATH" e "EXTENSIONS" que iniciem uma linha. *Cut* receberá o resultado disso e selecionará apenas o que vem depois do sinal de igual.

No caso de "EXTENSIONS", este procedimento deverá ser feito novamente ao final do laço de repetição que verifica se ainda há extensões, para que todas elas sejam verificadas.

4.5.3 Pesquisa dos arquivos e identificação de seus nomes

Para descobrir todos os arquivos na pasta do código-fonte que possuem as extensões determinadas pelo usuário, um esquema com o comando *find* precedendo uma *pipe* com o comando *wc* foi utilizado. O primeiro comando encontrará e listará todos os arquivos, enquanto o segundo fará a contagem de quantos foram encontrados.

Aproveitando o resultado do comando *find*, a identificação dos nomes de cada um, dentro do laço de repetição que verifica a existência de arquivos, será feita com o comando *sed.* Este selecionará uma linha a cada iteração para fazer a análise de cada arquivo.

4.5.4 Análise dos arquivos

Os arquivos serão analisados de modos distintos de acordo com a opção que foi dada ao executá-lo.

4.5.4.1 Identificando as regiões paralelas

A primeira medida tomada é identificar quantas diretivas "pragma omp parallel" existem no código. Isso é feito usando uma cascata de *pipes* de comando. Primeiro, um *grep* é executado de modo a encontrar o padrão "#" como primeiro

caractere de alguma linha (espaços são desconsiderados). Com este resultado, identifica-se quais deles dão início a um "pragma omp parallel", a partir do uso de outro *grep*. Por fim, conta-se os resultado com o comando *wc*.

4.5.4.2 Identificando as regiões de comentário

Se houver regiões, primeiramente o cabeçalho *sprofops.h* será incluído logo na primeira linha do arquivo, com o comando *sed*. Em seguida, será verificado se elas estão dentro de regiões comentadas com "/* ... */". A função *id_comments* fará essa verificação. Ela criará um arquivo temporário que identifica as regiões comentadas na forma "início-fim", onde "início" é o número da linha de início do comentário e "fim" é o número da linha de término.

Sua execução começa criando dois arquivos temporários. O primeiro armazena as linhas que começam por "/*" (desconsiderando espaços) ou que terminam com "*/", descobertas por meio do comando *grep*. A partir deste arquivo, identifica-se todas as linhas que, ao mesmo tempo, comecem por "/*" e terminem por "*/". O segundo arquivo temporário marcará justamente esta linhas, a partir do uso de dois *grep* em sequência. Da maneira que o *script* foi implementado, não haverá nenhum "pragma omp parallel" identificado nestas linhas. Portanto, elas não figurarão no arquivo temporário que identifica as regiões comentadas.

Nesse sentido, com as informações do segundo arquivo, as linhas inválidas serão deletadas no primeiro. Usando o comando *cat* seguido de um *wc*, verifica-se o tamanho deste arquivo. Por meio de um *sed* seguido de um *cut* em cada arquivo, eles serão comparados linha a linha. Caso as linhas sejam a mesma, significa que ela pode ser deletada no primeiro arquivo.

Ao final, o primeiro arquivo temporário terá todas as linhas que começam por "/*" ou terminam por "*/" exclusivamente. Pode-se, então, formar o arquivo com os intervalos "início-fim".

Para isso, será usado, novamente, um comando *sed* seguido de um *cut* para cada linha. Cada par de linhas será analisado em conjunto, pois, se uma linha identifica um "/*", a próxima linha necessariamente identificará um "*/" e, juntas

formarão a região de comentário. O comando para cada par estará embutido em um comando *echo,* de modo que seja possível escrever o resultado direto no arquivo.

4.5.4.3 Inserindo sprof_start()

A inserção de *sprof_start* ocorrerá na linha acima de "pragma omp parallel" e será feita pel a função *set_start* no *script*. Os dois comando *grep* foram executados exatamente como descrito na subseção 4.5.4.1. Do modo que eles foram usados, o resultado não trará apenas as linhas correspondentes, mas também o número dessas linhas. Essa informação será selecionado com um *cut* e, em sequência, um *sed* selecionará cada linha individualmente.

A próximo passo é confrontar essa linha com a informação salva no arquivo de identificação de comentários. Caso seu número esteja em um intervalo discriminado neste arquivo, essa linha será marcada com inválida e a inserção de *sprof_start* não será feita. Um função chamada *is_valid* fará este trabalho.

Nela, o tamanho do arquivo de comentários é consultado com um *cat* seguido por um *wc*. Enquanto houver linhas a serem verificadas, o comando *sed* será usado para recuperar os valores de início e fim de linhas. Se o número da linha a ser instrumentada é maior do que a linha de início e menor do que a linha de fim, então ela é inválida e *sprof_start* não será inserido. Uma *flag* sinalizará isso para que, no momento da inserção *sprof_stop*, não o faça também. Caso a linha não seja inválida, *sprof_start* será inserida com o comando *sed*.

4.5.4.4 Inserindo sprof_stop()

Caso a linha calculada com *is_valid* seja válida, então haverá a inserção de *sprof_stop*. Para isso, é preciso saber qual chave "}" representa o fim da área paralelizada. Visto que a chave não estará em qualquer lugar antes da região paralela, um *sed* seleciona apenas as linhas do início da região paralela até o fim do arquivo e um *grep* em sequência determina quais dessas linhas possuem um "{". O mesmo esquema é feito novamente, porém com um *grep* para selecionar as linhas

com "}". Dois arquivos temporários são criados para guardar essas informações, respectivamente.

A partir disso, o arquivo com as informações das chaves "}" é concatenado ao arquivo de chaves "{" com o comando *cat*. Como ocorrência cada ocorrência da chave acompanha o número da linha dela, elas serão ordenadas numericamente com o comando *sort*.

A primeira linha do arquivo temporário das chaves sempre corresponderá à chave "{" que delimita o início da área paralelizada. Desse modo, será feito uma verificação das chaves a partir da segunda linha do arquivo. Caso, nesta linha, a chave consista em um "}", significa que ela é o fim da região paralela e, portanto, pode ter "sprof_stop" inserido logo abaixo. Caso contrário, significa que é simplesmente uma região com chaves dentro da área paralelizada. Sabe-se, então, que a próxima chave "}" encontrada delimitará o fim desta região, e não da área paralelizada. Contadores farão o controle disso.

4.5.4.5 Remoção da instrumentação

A opção *clean* passada no momento da execução do *script* determina que a instrumentação deve ser removida do código. Isto será feito pra função *cleaning*. Nela, deleta-se, com o comando *sed,* a presença das *strings* "sprof_start()", "sprof_stop()" e a linha que contém o cabeçalho de *sprofops.h*.

5 TESTES, RESULTADOS E DISCUSSÕES

Os testes foram desempenhados em um servidor com 4 AMD Opterons, cada um com 16 núcleos. Portanto, dando um possibilidade de execução escalável com 64 *threads.* A escolha de aplicações-alvo adequadas demonstrarão grande parte das capacidades de Sprof. Elas devem ser programas consolidados, um consenso no meio científico. Por tais motivos, o *PARSEC Benchmark* foi escolhido para confrontar a implementação da ferramenta. Três aplicações disponíveis por ele foram perfiladas. São elas: *bodytrack, blackscholes* e *freqmine*.

O *PARSEC Benchmark* (*The Princeton Repository for Shared-Memory Computers*) consiste em um conjunto de aplicações que são grandes e diversas o suficiente para serem representativas no que diz respeito aos processadores modernos[25]. Seus programas apresentam paralelização usando-se OpenMP e Pthreads, os quais podem ser escolhidos no momento da compilação.

Desse modo, é necessário, antes de tudo, compilar as aplicações escolhidas. A subseção a seguir apresenta como se deu este processo.

5.1 O PROCESSO DE COMPILAÇÃO DAS APLICAÇÕES

PARSEC apresenta um *script* executável que permite, de maneira fácil, escolher uma aplicação do seu pacote para compilação, passando várias opções que, dentre outra possibilidades, permite escolher qual *API* será usada para paralelização do código.

Visto que a aplicação-alvo será executada pelo perfilador, antes de formar os executáveis dela, precede-se a inserção das funções de marcação de tempo da bilbioteca se Sprof. Este procedimento será explicado mais adiante.

Considerando que toda a instrumentação foi feita, o início do processo de compilação necessita que o compilador saiba onde se encontram as bibliotecas dinâmicas que serão usadas. A primeira solução consiste em transferi-las para uma pasta padrão de bibliotecas no sistema Linux. Porém, a solução implantada neste

[25] BIENA, C. *Op. cit.,* cap. 2.

trabalho consistiu em alterar arquivos de configuração do *PARSEC* que controlam as opções de compilação das aplicações. Cada pasta de programa (localizadas na pasta *pkgs*), possui uma pasta chamada *parsec*, que contém esses arquivos. Para programas que serão compilados em Pthreads ou OpenMP, os arquivos pertinentes corresponderão, respectivamente, a *gcc-pthreads.bldconf* e *gcc-openmp.bldconf*. Portanto, eles devem ser alterados para conter as seguintes linhas, ao final deles:

a) LDFLAGS="${LDFLAGS} -L<caminho para a biblioteca>";
b) LIBS="${LIBS} -lsprof.so".

A primeira linha insere o diretório especificado nos diretórios de procura do linkeditor (*linker*). A segunda linha diz ao linkeditor que a biblioteca fará parte da execução e assim, referencia os símbolos das funções dela, que serão resolvidos no momento da que a aplicação for chamada.

Além disso, é necessário informar ao *loader* o local da biblioteca, pois apenas assim ele poderá, no momento da montagem, achá-la para criar o contexto de execução da aplicação. Isto foi feito criando-se a variável LD_LIBRARY_PATH, a partir do comando *export*, isto é, "export LD_LIBRARY_PATH=<caminho para a biblioteca>".

Finalmente, pode-se compilar os códigos. *PARSEC* permite uma compilação rápida das aplicações desejadas através do *script parsecmgmt*, dentro da pasta *bin* no diretório raiz do *PARSEC*. Dessa forma, executou-se a linha de comando, para cada aplicação: *./parsecmgmt -a build -p <nome da aplicação> -c gcc-openmp/pthreads*. A *flag* "-p" informa qual o programa será compilado. Portanto, assumiu a forma *blackscholes*, *bodytrack* ou *freqmine*. A *flag* "-c" determina qual método de paralelização será utilizado e, consequentemente, qual arquivo de configuração será lido. Portanto, programas em OpenMP receberão *gcc-openmp* e programas em Pthreads receberão *gcc-pthreads*.

Para a exibição da linha de comando dos testes, os caminhos para as aplicações e demais arquivos foram omitidos, por motivos de clareza. Além disso, todos os arquivos de entrada são dos pacotes *input_native.tar* presentes nas pastas *inputs* das respectivas aplicações.

Para as execuções de teste, foi usada uma progressão aritmética de razão 2, de 1 a 64 *threads*. Portanto, o arquivo *sprof_exec.conf* permite duas configurações equivalentes:

a) number_of_tests=1, list_threads_values={2,4,8,16,32,64};
b) ou number_of_tests=1, max_number_threads=64, type_of_step=power, value_of_step=2.

5.2 TESTES COM *BLACKSCHOLES*

Blackscholes é uma aplicação que calcula, analiticamente, preços para um portfólio de opções de investimento no mercado europeu através de uma equação diferencial parcial de Black-Scholes[26], dada pela equação:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

A aplicação dá a opção de ser compilada para OpenMP e Pthreads. Para os testes deste trabalho, foi escolhido Pthreads. Toda a instrumentação foi feita no código-fonte *blackscholes.c*.

Ela deve receber três argumentos. O primeiro será o número de *threads*, o segundo é um arquivo de entrada sobre qual os cálculos serão feitos. O terceiro consiste no arquivo de saída, com os preços calculados. Perceba, que o uso da função *sprof_thrnum* não é necessário, pois o primeiro argumento de *blackscholes* recebe o número de *threads*. Consequentemente, a opção "-t" de Sprof pode ser utilizada.

Por ser um programa em Pthreads, não foi possível a execução de *sprof_instr.sh* para instrumentar o código automaticamente. Portanto, a inserção das funções de marcação estão passíveis à análise do código em *blackscholes.c*.

À procura da região onde as *threads* são criadas e terminadas, encontrou-se apenas uma, onde uma laço sobre o número de *threads* (variável *nThreads*), que as

[26] BIENA, Christian, KUMAR, Sanjeev, LI, Kai e SINGH, Jaswinder P. **The PARSEC Benchmark Suite:** Characterization and Architectural Implications. In: 17th International Conference on Parallel Architectures and Compilation Techniques, Toronto, Canadá, outubro 25-29, 2008, p. 72-81. DOI=

cria com a chamada de *CREATE_WITH_ARG*. O término de todas elas ocorre com a chamada de WAIT_FOR_END. Portanto, como não é preciso marcar o tempo de *threads* individualmente, não a necessidade de usar as funções *sprof_pthstart* e *sprof_pthstop*. Portanto, o conjunto de funções *sprof_start* e *sprof_stop* foi adequadamente escolhido. As funções ficaram nas linhas 440 e 446 em *blackscholes.c*. A figura 8 mostra como ficou essa instrumentação.

A compilação da aplicação foi feita com o comando no terminal: *./parsecmgmt -a build -p blackscholes -c gcc-pthreads*. Com o executável, Sprof foi invocado na forma: *./sprof -t 1 blackscholes 4 in_10M.txt prices.txt.* O arquivo *in_10M.txt* é a entrada, *prices.txt* será a saída e "4" é o número de *threads*, que poderia ser qualquer valor, já que não será usado.

Figura 8 - Instrumentação em *blackscholes.c*.

```
sprof_start();
for(i=0; i<nThreads; i++) {
    tids[i]=i;
    CREATE_WITH_ARG(bs_thread, &tids[i]);
}
WAIT_FOR_END(nThreads);
sprof_stop();
```

Fonte: Autor.

5.3 TESTES COM *BODYTRACK*

Bodytrack é uma aplicação de visão computacional que rastreia um corpo humano, com múltiplas câmeras, através uma sequência de imagens[27].

Em contraste com *blackscholes*, este programa foi paralelizado com OpenMP, possibilitando, assim, o uso do *script* de instrumentação. Com isso, o arquivo *sprof_instr.conf* foi usado, com o caminho para a pasta raiz do código fonte de *bodytrack* na variável *DPATH* e as extensões definidas para instrumentação foram ".h" e ".cpp".

Feita a execução do *script*, cinco regiões foram corretamente instrumentadas. Duas delas ficaram no arquivo *ParticleFilterOMP.h*, nos pares de linhas 68 com 76

[27] *Idem.*

(região 4) e 114 com 120 (região 5). As outras três ficaram no arquivo *TrackingModelOMP.cpp*, nos pares de linhas 108 com 120 (região 1), 44 com 58 (região 2) e, por fim, 73 com 87 (região 3).

Assim como na aplicação anterior, *bodytrack* também recebe o número de *threads* por linha de comando e, assim, não exige o uso de *sprof_thrnum*.

A compilação da aplicação foi feita com o comando no terminal: *./parsecmgmt -a build -p bodytrack -c gcc-openmp*. Em seguida, Sprof foi executado: *./sprof -t 7 bodytrack sequenceB_261 4 100 6000 12 3 4 1.* Vale salientar o significado de alguns desses argumentos. *sequenceB_261* é a pasta que contém as imagens a serem analisadas. “100” é a quantidade de imagens que serão utilizadas. “3” é o valor para determinar que a aplicação foi compilada em OpenMP. “4” é o número de *threads*, que equivale ao sétimo argumento e, por isso, tem-se “-t 7”. A explicação dos demais foge do escopo deste trabalho e foram baseados em exemplos de manuais do *PARSEC*.

5.4 TESTES COM *FREQMINE*

Freqmine é um programa da área de mineração de dados que emprega uma versão baseada em vetor do algoritmo *FP-growth* (*Frequent Pattern Growth*) para mineração de conjunto de itens frequentes (*Frequent Itemset Mining*)[28].

Para este último teste, OpenMP foi escolhido novamente, possibilitando mais um teste da instrumentação automática de código-fonte. As extensões definidas foram as mesmas que em *bodytrack*, mas, obviamente, a variável *DPATH* teve de ser ajustada para a pasta do código-fonte de *freqmine*. A execução dele gerou a instrumentação de sete regiões, todas contidas no arquivo *fp_tree.cpp*. Portanto, as regiões instrumentadas foram:

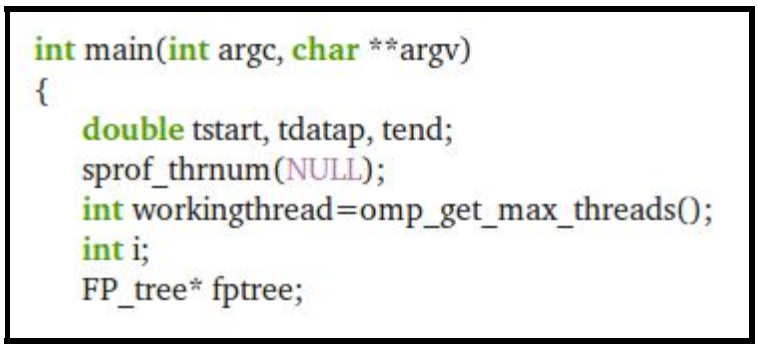
a) Região 1 nas linhas 888 a 946;
b) Região 2 nas linhas 541 a 636;
c) Região 3 nas linhas 730 a 756;
d) Região 4 nas linhas 1078 a 1186;

[28] *Idem*.

e) Região 5 nas linhas 1197 a 1205;
f) Região 6 nas linhas 213 a 289;
g) Região 7 nas linhas 1377 a 1461.

Porém, *freqmine* não recebe valor de *thread* pela linha de comando, exigindo, assim, a inserção da função *sprof_thrnum*. Constatou-se, a partir da análise do código-fonte, que não há nenhuma variável declarada que armazene um possível número de *threads*. O programa se utiliza da variável de ambiente OMP_ NUM_THREADS para esta finalidade, devido ao uso de funções como *omp_set_num_threads* e *omp_get_num_threads*[29]. Dessa forma, *sprof_thrnum* deve ser inserido antes da primeira chamada à *omp_get_num_threads*. A figura 9 mostra, então, o local mais adequado encontrado, em um trecho localizado no arquivo *fpmax.cpp*.

Figura 9 - Inserção de *sprof_thrnum* em *fpmax.cpp*.



```
int main(int argc, char **argv)
{
    double tstart, tdatap, tend;
    sprof_thrnum(NULL);
    int workingthread=omp_get_max_threads();
    int i;
    FP_tree* fptree;
```

Fonte: Autor.

Feito isso, compilou-se a aplicação com o comando: *./parsecmgmt -a build -p freqmine -c gcc-openmp*. Em seguida, Sprof foi executado: *./sprof freqmine webdocs_250k.dat 14000.* O primeiro argumento é o arquivo de entrada, enquanto o segundo representa o tamanho do problema e, para este arquivo de entrada, deve ser maior do que 11000[30]. Nota-se que, desta vez, a opção "-t" não está presente no comando de Sprof.

[29] OpenMP Architecture Review Board. *op. cit.*
[30] BIENA, Christian, KUMAR, Sanjeev, LI, Kai e SINGH, Jaswinder P. *op. cit.*

5.5 RESULTADOS DOS TESTES

As tabelas que seguem mostram os resultados para 1 teste nas três aplicações. O número dentro dos parêntesis indicam a qual região aquela coluna corresponde.

A tabela 1 contém os resultados de tempo e *speedup* considerando a aplicação em sua totalidade, enquanto as tabelas 2 e 3 indicam os resultados para cada região específica. Os valores foram arredondados para 3 casa decimais, com os tempos em segundos. a letra “S” representa o *speedup*.

Tabela 1 - Tempos totais e *speedups* para as aplicações.

	Blackscholes	*Bodytrack*	*Freqmine*
1 *thread*	501,448 s S = 1,000	559,482 s S = 1,000	188,090 s S = 1,000
2 *threads*	277,952 s S = 1.804	294,442 s S = 1,900	99,929 s S = 1,882
4 *threads*	166,025 s S = 3,020	156,550 s S = 3,578	54,985 s S = 3,420
8 *threads*	109,301 s S = 3,020	91,201 s S = 6,134	32,531 s S = 5,781
16 *threads*	80,887 s S = 6,199	54,815 s S = 10,206	22,903 s S = 8,212
32 *threads*	68,272 s S = 7,344	39,922 s S = 14,014	18,309 s S = 10,272
64 *threads*	60,688 s S = 8,262	31,270 s S = 17,8920	17,381 s S = 10,821

Fonte: Autor

Um dos principais aspectos de Sprof fica claro a partir destes resultados: a ferramenta ajuda a entender o comportamento de uma aplicação desconhecida. A presença de várias regiões paralelas levanta a dúvida para um usuário que analise seu código-fonte: “qual a influência de cada uma delas na execução? Qual o nível de paralelismo que apresentam?”.

Tabela 2 - Tempos e *speedups* de cada região em *blackscholes* e *bodytrack*.

	Blackscholes	*Bodytrack(1)*	*Bodytrack(2)*	*Bodytrack(3)*	*Bodytrack(4)*	*Bodytrack(5)*
1 *thread*	448,754 s S = 1,000	3,719 s S = 1,000	4,321 s S = 1,000	4,241 s S = 1,000	485,738 s S = 1,000	43,486 s S = 1,000
2 *threads*	224,741 s S = 1,996	1,889 s S = 1,968	2,184 s S = 1,978	2,140 s S = 1,981	247,851 s S = 1,960	22,041 s S = 1,900
4 *threads*	112,657 s S = 3,983	0,988 s S = 3,764	1,097 s S = 3,937	1,075 s S = 3,945	123,698 s S = 3,927	11,254 s S = 3,574
8 *threads*	56,437 s S = 7,951	0,538 s S = 6,915	0,558 s S = 7,736	0,546 s S = 7,770	64,954 s S = 7,478	6,036 s S = 7,204
16 *threads*	28,337 s S = 15,836	0,355 s S = 10,479	0,279 s S = 15,500	0,278 s S = 15,255	32,087 s S = 15,138	3,184 s S = 13,655
32 *threads*	14,535 s S = 30,873	0,321 s S = 11,562	0,169 s S = 25,493	0,163 s S = 26,143	18,539 s S = 26,200	1,996 s S = 21,784
64 *threads*	7,964 s S = 56,343	0,385 s S = 9,657	0,164 s S = 26,236	0,149 s S = 28,388	9,927 s S = 48,926	1,628 s S = 26,711

Fonte: Autor

Tabela 3 - Tempos e *speedups* de cada região em *freqmine*.

	Freqmine(1)	*Freqmine(2)*	*Freqmine(3)*	*Freqmine(4)*	*Freqmine(5)*	*Freqmine(6)*	*Freqmine(7)*
1 *thread*	NS[31]	1,468 s S = 1,000	NS	0,363 s S = 1,000	NS	0,495 s S = 1,000	177,450 s S = 1,000
2 *threads*	NS	0,694 s S = 2,113	NS	0,152 s S = 2,385	NS	0,361 s S = 1,373	89,486 s S = 1,983
4 *threads*	NS	0,441 s S = 3,331	NS	0,079 s S = 4,567	NS	0,184 s S = 2,691	45,257 s S = 3,921
8 *threads*	NS	0,287 s S = 5,119	NS	0,052 s S = 7,021	NS	0,106 s S = 4,649	23,055 s S = 7,696
16 *threads*	NS	0,234 s S = 6,263	NS	0,058 s S = 6,228	NS	0,074 s S = 6,674	12,937 s S = 13,715
32 *threads*	NS	0,187 s S = 7,842	NS	0,040 s S = 9,109	NS	0,056 s S = 8,885	8,724 s S = 20,339
64 *threads*	NS	0,204 s S = 7,186	NS	0,069 s S = 5,261	NS	0,035 s S = 13,939	7,445 s S = 23,834

Fonte: Autor

[31] NS = Pouca significância. Os valores encontrados foram insuficientes para uma boa precisão com as três casas decimais estipuladas.

Nesse sentido, *freqmine* consiste na melhor ilustração deste fato. Sprof mostrou que as regiões 1 e 5 possuem tempos de execução baixíssimos e, portanto, não desempenham qualquer papel importante com respeito ao *speedup* e tempo de execução do programa. Ao final, apenas a região 7 demonstrou um comportamento escalável. Conclusão semelhante pode ser tomada sobre *bodytrack*, na qual as regiões 4 e 5 se mostraram importantes para uma análise do paralelismo envolvido na aplicação.

Os dados estão em conformidade com vários aspectos da literatura especializada. Percebe-se, por exemplo, capacidade de paralelismo limitada nas regiões com baixo tempo de execução. Isto deve-se ao fato de que, devido ao tempo gasto com comunicação entre as *threads*, isto se sobreponha ao cálculo paralelo em si.

Outro fator observado foi que o *speedup* das aplicações completas são menores do que para cada parte individual. Evidentemente, a aplicação completa está repleta de trechos em serial, muito mais do que os trechos paralelizados internamente possuem, e a lei de Amdahl prevê muito bem que eles representam um gargalo muito grande para a escalabilidade das aplicações. Baseando-se na lei de Gustafson, seria possível obter resultados melhores caso aumentássemos o tamanho dos problemas. Porém, os maiores fatores possíveis disponibilizados pelo *PARSEC* foram usados para esta finalidade, além de tal análise foge do escopo deste trabalho.

Embora estes resultados pareçam promissores, um dos requisitos (talvez o mais importante) deve ser levado em consideração antes de qualquer conclusão sobre a efetividade da ferramenta: o quão “intrusiva” ela é? Os tempos adquiridos com Sprof destoam muito em relação àqueles adquiridos sem ele?

5.6 MEDIÇÃO DE INTRUSÃO

A validação completa de Sprof passa, necessariamente, pela medição do tempo de execução das aplicações sem seu uso. Desta forma, um teste de “intrusão” foi feito para todas as elas.

Cada um foi executada usando-se o programa *time*[32]. Esta aplicação, disponível em sistema Linux, basicamente mostra três tipos de tempo. O primeiro (*real*) é o tempo total gasto na execução. O segundo (*user*) refere-se ao tempo no escopo do usuário. A terceira (*sys*), enfim, marca o tempo para o escopo do sistema. Para este trabalho, o interesse está no tempo *real*. Este será comparado, então, ao tempo total marcado por Sprof para os mesmo parâmetros de execução. Com valores de média e desvio padrão, tem-se uma métrica que mostra a intrusão da ferramenta quantitativamente.

Para cada uma das três, foram desempenhados 100 testes, com 64 *threads* para *blackscholes* e 32 *threads* para as outras duas, primeiro com Sprof e depois com *time*. As linhas de comando usadas foram iguais às usadas nos testes gerais, com exceção ao número de imagens usadas para *bodytrack* que passou de “100” para “20”, devido à grande quantidade de testes realizada.

A tabelas 4 mostra os resultados numéricos, com média e desvio-padrão calculados, enquanto as figuras 10, 11 e 12 mostram os resultados graficamente.

Tabela 4 - Média (M) e desvio-padrão (S.D) dos tempos de execução

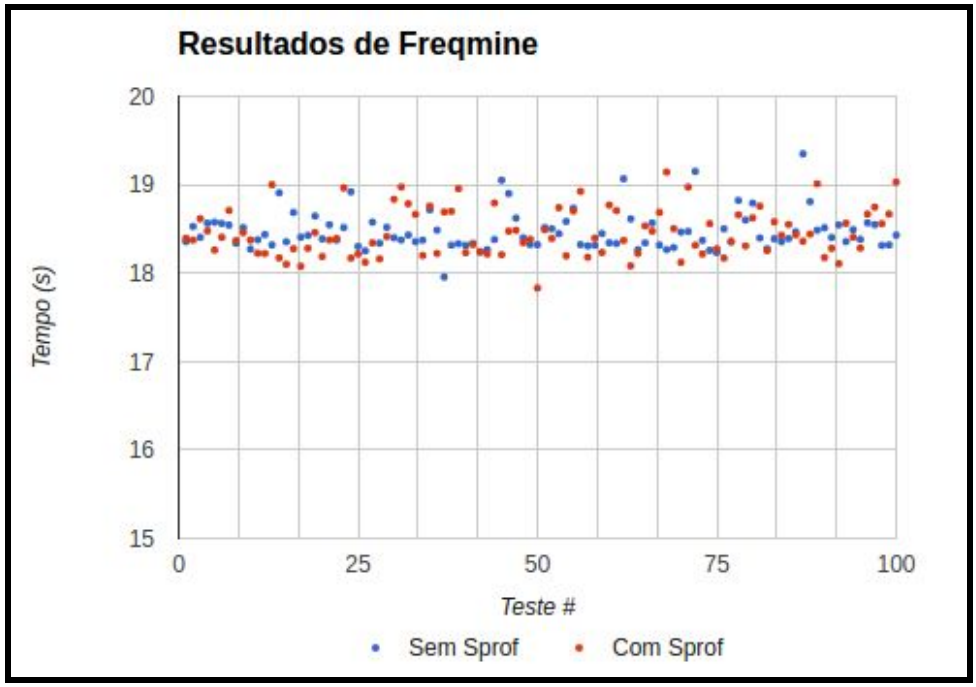
	Blackscholes	*Bodytrack*	*Freqmine*
Com Sprof	M = 60,642 s S.D = 0,472	M = 8,285 s S.D = 0,070	M = 18,456 s S.D = 0,264
Sem Sprof	M = 60,749 s S.D = 0,337 s	M = 8,304 s S.D = 0,071	M = 18,472 s S.D = 0,212

Fonte: Autor

Para analisar estes resultados, consideremos a hipótese de que os dois grupos **não** possuem uma diferença estatisticamente significante entre eles. A partir disso, pode-se usar algum método estatístico de teste de hipótese.

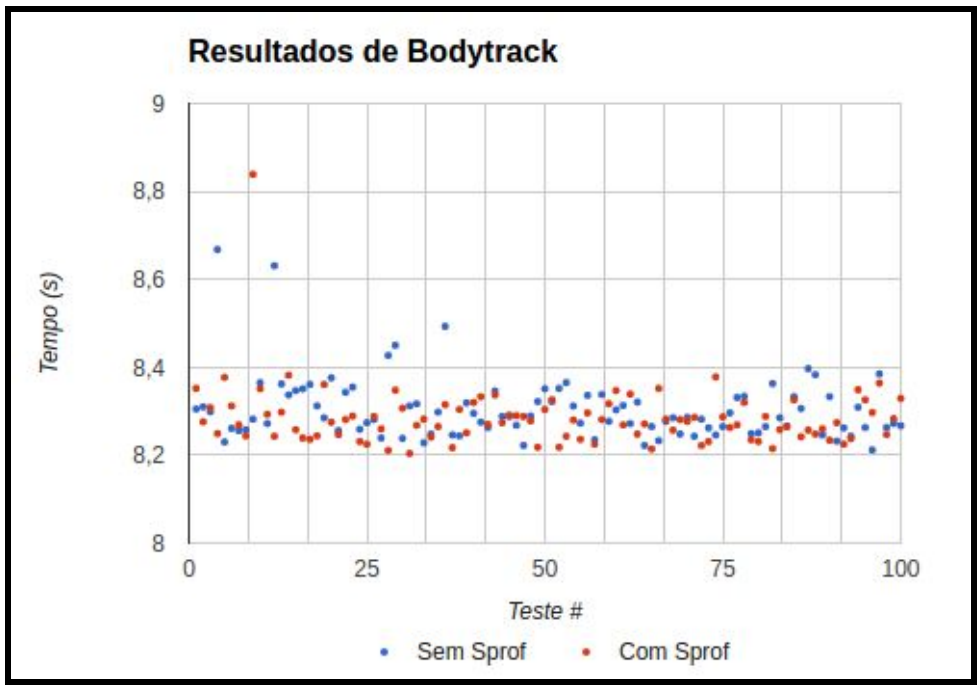
[32] NEWELL, Gary. **Run a Command And Return Time Statistics Using time Command**. Disponível em: <https://www.lifewire.com/command-return-time-command-4054237>. Acesso em: 29 de nov. de 2016.

Figura 10 - Resultados de 100 testes para 32 *threads* com *Freqmine.*



Fonte: Autor

Figura 11 - Resultados de 100 testes para 32 *threads* com *Freqmine.*



Fonte: Autor

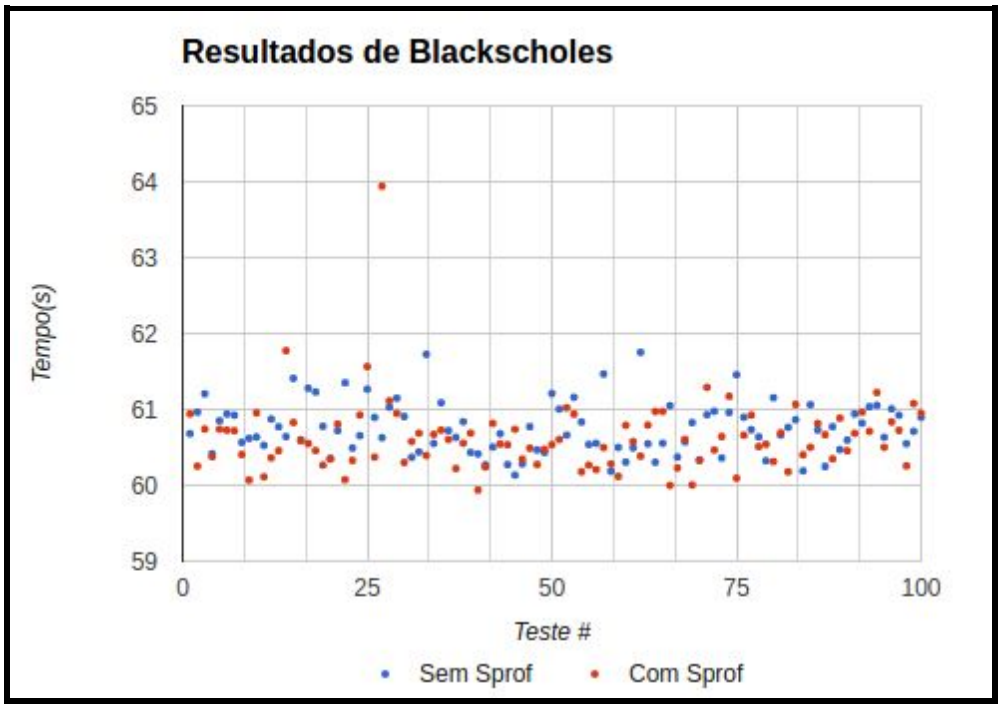
Por meio de uma **análise de variância** (*one-way ANOVA*)[33] [34], é possível

[33] **One-Way ANOVA**. Disponível em: <https://statistics.laerd.com/statistical-guides/one-way-anova-statistical-guide.php>. Acesso em: 29 de

extrair essas informações. Com seu cálculo, foram obtidos os seguintes resultados:

a) F = 3,404 e p = 0,067 para *blackscholes*;
b) F = 3,631 e p = 0,058 para *bodytrack*;
c) F = 0.223 e p = 0,637 para *freqmine*.

Figura 12 - Resultados de 100 testes para 64 *threads* com *Blackscholes.*



Fonte: Autor

Se tomarmos o nível de significância padrão ($\alpha = 0,05$), conclui-se que todas as aplicações **não rejeitam** a hipótese, pois os respectivos valores *p* são sempre maiores do que α, consequentemente, os valores de *F* são menores do que o valor crítico para o grau de liberdade do problema. Assim, os dados **não possuem diferença estatisticamente significante**, de acordo com o método *ANOVA*, o que valida um dos requisitos que Sprof deve possuir: não ser intrusivo.

nov. de 2016.
[34] **Analysis of Variance (ANOVA) Calculator - One-Way ANOVA from Summary Data.** Disponível em: <http://www.danielsoper.com/statcalc/calculator.aspx?id=43>. Acesso em: 29 de nov. de 2016.

6 CONCLUSÕES

Sprof é uma ferramenta muito simples com intuito de traçar um perfil de *speedup* de uma aplicação paralela que se utiliza do modelo de *threads* de modo automático. O usuário deve instrumentar o código, marcando o início e fim da região a ser perfilada, com funções disponibilizadas por uma biblioteca. Sua utilidade baseia-se na importância em saber o tempo de execução e *speedup* que, no modelo de computação paralela, consistem nas informações primordiais para analisar o desempenho, a alto nível, de um programa .

Nesse sentido, o desenvolvimento da ferramenta estabeleceu dois objetivos: ser uma plataforma automática de testes, na qual a aplicação-alvo deve ser executada diversas vezes, para vários valores de *threads*; permitir uma instrumentação mais automática possível, de modo que o risco de erro nos resultados por ação do usuário seja reduzido.

Ao longo do trabalho, foram mostradas várias facetas da ferramenta que, com os testes desempenhados, respaldam estes objetivos. Com a ação do perfilador *sprof* e o uso das funções da biblioteca *libsprof.so*, possibilita-se que a aplicação-alvo execute automaticamente, com vários cenários de testes diferentes. A instrumentação automática também é realizada, com sucesso satisfatório, pelo *script sprof_instr.sh*.

A facilidade na instrumentação das aplicações do *PARSEC Benchmark* e a simplicidade na manipulação de *sprof* e *sprof_instr.sh* mostraram que os requisitos de desenvolvimento da ferramenta também foram cumpridos. Dessa forma, o usuário tem à sua disposição uma ferramenta de fácil aprendizado e que é muito objetiva nas suas funcionalidades e informações que produz. Mais importante ainda: *Sprof* se mostrou estatisticamente não intrusivo e, portanto, pode ser usado sem temer alterações no tempo de execução e *speedup* das aplicações-alvo.

Embora os resultados sejam promissores, há diversas limitações, das quais várias existem devido justamente à simplicidade proposta para seu desenvolvimento. Entre elas, podemos citar:

a) Falta de uma instrumentação automática para programas em Pthreads, devido a alta granularidade que este modelo apresenta;
b) As áreas em códigos OpenMP que podem ser instrumentadas devem estar contidas entre chaves;
c) Não há suporte à instrumentação através do compilador, funcionalidade comum entre as ferramentas que instrumenta código-fonte;
d) Os resultados são escritos em arquivos texto. Geralmente, as ferramentas de perfilamento utilizam telas gráficas e bem desenhadas para apresentar os resultados.
e) Há um executável separado, com uma linguagem naturalmente diferente, para a instrumentação automática. Tal funcionalidade seria mais elegante se estivesse inserida no código-fonte de *Sprof*;
f) a estrutura de arquivos é rígida e, portanto, os arquivos de configuração devem estar na mesma pasta do *script.*

Portanto, a solução para estas limitações faz parte do trabalho futuro de desenvolvimento de *Sprof*. Além disso, considera-se, também, a possibilidade de uso de instrumentação binária para a finalidade de perfilamento e adição de outras funcionalidades, de modo que ela se aproxime das soluções renomadas no cenário científico.

REFERÊNCIAS

ADHIANTO, Laksono, FRANCO, Michael, LANDRUM, Reed, MELLOR-CRUMMEY, John, TALLENT, Nathan R. **Scalable Fine-grained Call Path Tracing**. In: ICS '11 International Conference on Supercomputing, Tucson, EUA, maio 31- junho 4, 2011.

_____, MELLOR-CRUMMEY, John e TALLENT, Nathan R. **Effectively Presenting Call Path Profiles of Application Performance**. In: 2010 39th International Conference on Parallel Processing Workshops San Diego, EUA, setembro 13-16, 2010. DOI=10.1109/ICPPW.2010.35.

Analysis of Variance (ANOVA) Calculator - One-Way ANOVA from Summary Data. Disponível em: <http://www.danielsoper.com/statcalc/calculator.aspx?id=43>. Acesso em: 29 de nov. de 2016.

BARROS, Carlos A., JALES, Márcio de O., SILVEIRA, Luís Felipe Q. e XAVIER-DE-SOUZA, Samuel. **Nor faster nor slower tasks, but less energy hungry and parallel**: simulation results. In: Fourth Berkeley Symposium on Energy Efficient Electronic Systems, Berkeley, California, EUA, outubro 1-2, 2015.

BIENA, Christian. **Benchmarking Modern Multiprocessors.** Princeton, EUA: Princeton University, 2011, 153 f. Tese (Doutorado em filosofia). Curso de Filosofia, Departamento de Ciência da Computação, Princeton University, Princeton, 2011.

_____, KUMAR, Sanjeev, LI, Kai e SINGH, Jaswinder P. **The PARSEC Benchmark Suite:** Characterization and Architectural Implications. In: 17th International Conference on Parallel Architectures and Compilation Techniques, Toronto, Canadá, outubro 25-29, 2008.

CHITTIMALI, Pavan K., SHAH, Vipul. **GEMS**: A Generic Model Based Source Code Instrumentation Framework. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, Canadá, abril 17-21, 2012. DOI=10.1109/ICST.2012.195.

CODINA, Josep M., GILBERT, Enric, MADRILES, Carlos e MARTÍNEZ, Raúl. **Profiling support for runtime managed code**: next generation performance monitoring unit. In: IEEE Computer Architecture Letters, vol. 14, issue 1.

Free Software Foundation. **GNU Grep 2.26**. Disponível em: <http://www.gnu.org/software/grep/manual/grep.html#Copying>. Acesso em: 18 de nov. de 2016.

GEIMER, Markus, WOLF, Felix, WYLIE, Brian J., ÁBRAHÁM, Erika, BECKER, Daniel, MOHR, Bernd. **The Scalasca Performance Toolset Architecture**. Concurrency and Computation: Practice and Experience 22, 2010. DOI=10.1002/cpe.1556.

Intel Trace Collector 2017: User and Reference Guide. Disponível em:

<software.intel.com/sites/default/files/managed/17/61/ITC_User_and_Reference_Guide-2017.pdf>. Acesso em: 29 de nov. de 2016.

KERRISK, Michael. **The Linux Man-Pages Project**. Disponível em: <https://www.kernel.org/doc/man-pages/l>. Acesso em: 28 de nov. de 2016.

KNÜPFER, Andreas. et al. **Score-P - A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir.** In: Proceedings 5th Parallel Tools Workshop, Dresden, Alemanha, 2012.

MCCOOL, Michael, REINDERS, James e ROBISON, Arch D. Background. In: _____. **Structured Parallel Programming**: Patterns for Efficient Computation. Waltham, EUA: Morgan Kaufmann, 2012, 1. ed, cap. 2.

MITCHELL, Mark, OLDHAM, Jeffrey e SAMUEL, Alex. Interprocess Communication. In: _____. **Advanced Linux Programming.** Indianapolis, EUA: New Riders, 2001.

NEWELL, Gary. **Run a Command And Return Time Statistics Using time Command**. Disponível em: <https://www.lifewire.com/command-return-time-command-4054237>. Acesso em: 29 de nov. de 2016.

One-Way ANOVA. Disponível em: <https://statistics.laerd.com/statistical-guides/one-way-anova-statistical-guide.php>. Acesso em: 29 de nov. de 2016.

OpenMP Architecture Review Board. **OpenMP Application Programming Interface**. Versão 4.5, 2015. Disponível em: <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>. Acesso em: 28 de nov. de 2016.

PACHECO, Peter S. Why Parallel Programming?. In:_____ **An Introduction to Parallel Programming.** San Francisco, EUA: Morgan Kauffman Publishers, 1. ed., 2011.

PERMENT, Eric. **The Sed FAQ**: Frequently Asked Questions about sed, the stream editor. Disponível em <http://sed.sourceforge.net/sedfaq.html>. Acesso em: 18 de nov. de 2016.

SHENDE, Sameer S., MALONY, Allen D.. **The TAU Parallel Performance System**, SAGE Publications: Knoxville, EUA. In: International Journal of High Performance Computing Applications 20, 2006.

STEVANOVIC, Milan. The Impact of Reusing Concept. In:_____. **Advanced C and C++ Compiling.** California, EUA: Apress Berkeley, 2014.

The Open Group. **Shell and Utilities**: Detailed Toc. Disponível em: <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html>. Acesso em: 26 de nov. de 2016.

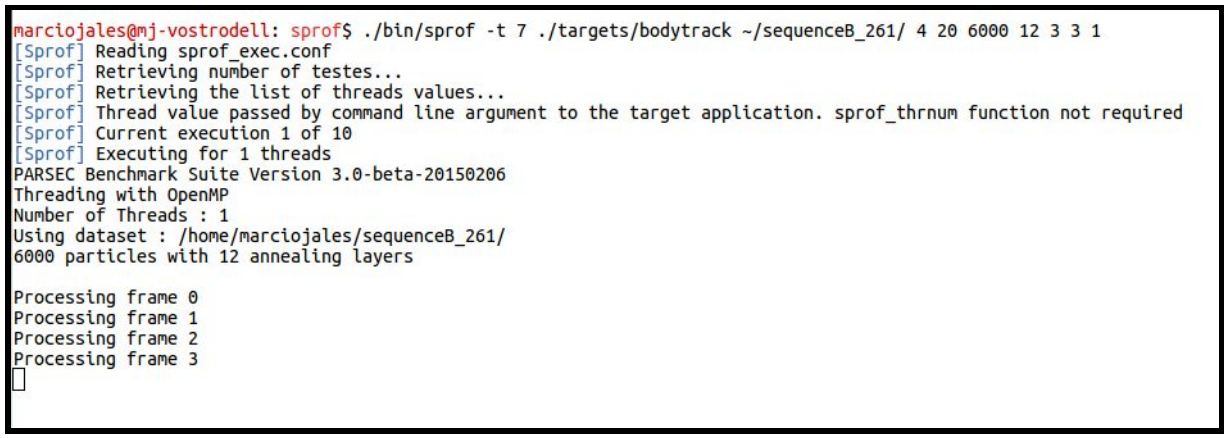
TRACY, Robb H. **LPI Linux Essentials Certification All-in-One Exam Guide**. New York, EUA: McGraw-Hill, 2013. Cap. 5.

WALL, Larry. **Perl version 5.24.0 documentation.** Disponível em <http://perldoc.perl.org/perl.pdf>. Acesso em: 20 de nov. de 2016.

APÊNDICE A - ILUSTRAÇÕES DE FUNCIONAMENTO

As figuras abaixo mostram um pouco do funcionamento de Sprof nos testes desempenhados na aplicação *bodytrack*.

Figura 13 - Execução de *bodytrack.*

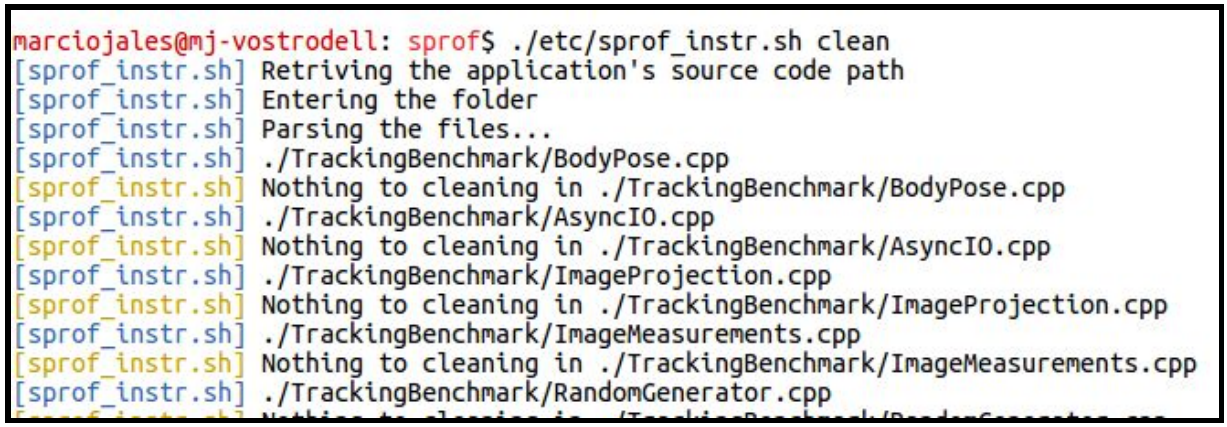


Fonte: Autor.

Figura 14 - Trecho da instrumentação automática no código de *bodytrack.*

Fonte: Autor.

Figura 15 - Trecho da eliminação de instrumentação no código de *bodytrack*.



Fonte: Autor.

APÊNDICE B - MANIPULAÇÃO DE ARQUIVOS EM C/LINUX

A linguagem C e os sistemas Linux possuem uma série de bibliotecas padrão que possibilitam a leitura e escrita de dados tanto em memória quanto em disco. Elas disponibilizam uma imensa quantidade de funções que permitirão manipular tanto arquivos de texto quanto binários, os dois tipos encontrados em sistemas Linux.

B.1 FUNÇÕES PARA ABRIR E FECHAR ARQUIVOS

Antes de qualquer tipo de ação dentro dos arquivos, a linguagem C exige que o desenvolvedor, antes de tudo, abra o arquivo. Ao final, é altamente recomendado, enquanto boa prática, que eles sejam fechados. Como é de se esperar, estas tarefas são feitas através de funções.

Para a finalidade de abrir arquivos, *open, fopen* e *fdopen* são algumas das principais funções, enquanto, para encerrá-los, *close* e *fclose* figuram entre as principais. As funções *open* e *close* são chamadas de sistemas*,* enquanto *fopen, fdopen* e *fclose* são implementações da biblioteca C padrão e, assim, não se restringem a um sistema específico.

A função *open* retorna um descritor de arquivo (ou -1 em caso de erro) e recebe três argumentos: O primeiro é uma cadeia de caracteres (*string*) com o caminho do arquivo a ser aberto: o segundo são *flags* que indicam quais características o arquivo aberto terá; e o terceiro refere-se aos modos que algumas destas *flags* agirão. Portanto, ele será usado apenas quando a *flag* no segundo argumento exigir.

Já *fopen* e *fdopen* retornam um objeto *FILE* (ou *NULL* em caso de erro) e ambas recebem dois argumentos. O segundo argumento é igual e consiste no modo com que o arquivo deve ser aberto (escrita, leitura ou concatenação). A diferença entre as duas está no primeiro argumento. *Fopen* recebe o caminho para o arquivo, enquanto *fdopen* recebe um descritor de arquivo.

No que diz respeito ao fechamento dos arquivos, *close* é o equivalente a *open* e *fclose* a *fopen* e *fdopen.* Ambas recebem apenas um argumento que, para *close,* é

o descritor de arquivo e, para *fclose*, o objeto *FILE* e retornam 0 em caso de sucesso. Erro em *close* é indicado com retorno -1, enquanto para *fclose* com retorno *EOF* (*end of file).*

B.2 FUNÇÕES PARA ESCRITA E LEITURA EM ARQUIVOS

Há, basicamente, dois grandes grupos de funções para escrita e leitura em arquivos: com e sem formatação. O primeiro tipo são àquelas funções que, em sua manipulação, permitem escrever e ler dados de vários formatos (inteiros, ponto flutuante, caracteres etc). Já o segundo não permite tal variação na formatação e, basicamente, escreve e recupera os dados apenas como caracteres.

B.2.1 Funções com formatação

Todas elas recebem, obrigatoriamente, uma cadeia de caracteres que representa a informação que será escrita/lida, com os devidos formatos para os dados. Para cada variável que será escrita/lida no arquivo, deve-se adicionar um argumento para cada uma, ao final da lista.

Entre as principais funções para escrita, estão *printf, fprintf, dprintf*, *sprintf* e *snprintf*. Todas elas retornam um valor inteiro correspondente ao número de caracteres escritos no arquivo em caso de sucesso e um valor negativo em caso de erro.

A diferença entre elas está na quantidade de argumentos obrigatoriamente recebidos. *Printf* escreve na saída padrão, portanto, a *string* a ser escrita será o primeiro argumento. Nos casos de *fprintf, dprintf, sprintf* e *snprintf*, o primeiro argumento delas devem ser, respectivamente, um objeto *FILE*, um descritor de arquivo e um ponteiro para caracteres, no últimos dois casos. Isto é, *sprintf* e *snprintf* não escrevem para arquivos, mas sim para um *strings*. *snprintf* tem um argumento adicional, que será o segundo e representa o número máximo de caracteres que serão escritos.

Com respeito à leitura, as três principais funções são *scanf*, *fscanf* e *sscanf*. Seus retornos consistem no número de itens que foram lidos do arquivo com

sucesso (cada item corresponde a um formato definido). *EOF* é retornado em caso de erro.

scanf pode ser comparada a *printf*, pois lê da entrada padrão. *fscanf* deve receber, no primeiro argumento, um objeto *FILE*. Portanto, é um equivalente de leitura à *fprintf*. A terceira, *sscanf*, se assemelha, então, à *sprintf*, pois recebe um ponteiro para caracteres no primeiro argumento.

B.2.2 Funções sem formatação

No lado das funções que não usam formatação, a escrita é feita comumente com as funções *putc*, *fputc, puts* e *fputs*, enquanto a leitura é associada às funções *getc, fgetc* e *fgets*.

Para a finalidade de escrita, *puts* recebe uma *string* no seu único argumento e imprime na saída padrão. Perceba que um possível equivalente *gets* existe, porém está depreciado. *putc* e *fputc* são equivalentes a *getc* e *fgetc*, inclusive retornam da mesma forma. Recebem um caractere no primeiro argumento que será escrito em um arquivo passado através de um objeto *FILE* no segundo argumento. Por fim, *fputs* fará semelhante às duas funções anteriores, com a diferença que o primeiro argumento corresponde a uma *string*. Tanto ela quanto *puts* um valor inteiro não negativo em caso de sucesso e *EOF* em caso de erro.

Para a leitura, tanto *getc* quanto *fgetc* recebem apenas um argumento, um objeto *FILE*, com apenas uma diferença de implementação entre elas que foge ao escopo. Elas leem o caractere que está na próxima posição do arquivo aberto e o retornam na forma de uma variável inteira. Em caso de erro, *EOF* é retornado. Por outro lado, *fgets* lê uma *string* e recebe três argumentos, respectivamente: a variável do tipo ponteiro para *char* que receberá a leitura, o número de caracteres que devem ser lidos, e o objeto *FILE* que representa o arquivo. Ela retorna a *string* do primeiro argumento em caso de sucesso e *NULL* em caso de erro ou se não há mais nada para ler.

B.2.3 As funções *read* e *write*

As funções *read* e *write* não levam em conta o fator formatação. Em suas implementações, elas não manipulam *strings* formatadas com os tipos das variáveis. Elas trabalham diretamente com a leitura e escrita de *bytes*.

write recebe três argumentos. O primeiro é o descritor de arquivos a ser escrito. O segundo é um ponteiro sem tipo (*void*), de onde vêm os *bytes* que serão escritos. O último argumentos diz a quantidade de *bytes* que será lida. *read* possui o mesmo protótipo de *write*, com a diferença que o segundo argumento será o endereço da variável que receberá os *bytes* lidos. Ambas retornam o número de *bytes* lidos/escritos e “-1” em caso de erro.

B.3 FUNÇÕES DE POSICIONAMENTO EM ARQUIVOS

Ao abrir um arquivo e manipulá-lo, é possível mudar o posicionamento em que a próxima informação será lida/escrita. Três funções principais, entre outras, prestam esse papel: *fseek, rewind* e *ftell*.

A primeira possui três argumentos e posiciona o arquivo em qualquer lugar. Para isso, passa-se, no terceiro argumento, um posicionamento inicial, que pode ser SEEK_SET para especificar o início do arquivo, SEEK_CUR para dizer que a posição deve ser a atual ou SEEK_END, que posiciona ao final do arquivo.

A partir desse “pré-posicionamento”, o segundo argumento indica o *offset*, em *bytes*, ou seja, o deslocamento do ponto de leitura/escrita a partir da posição definida anteriormente. O arquivo deve ser passado como um objeto *FILE* no primeiro argumento. Ela retorna 0 em caso de sucesso e -1 em caso de erro.

Rewind é uma função que é simplesmente equivalente a *fseek* com *offset* zero e SEEK_SET, ou seja, leva a posição para o início do arquivo. Recebe o arquivo como argumento e não possui retorno.

Por fim, *ftell* não é um função que manipula o posicionamento. Ela simplesmente retorna qual é este valor atualmente. Em caso de erro, -1 é retornado.

APÊNDICE C - BIBLIOTECAS

Em programação e, mais especificamente, se tratando de linguagens compiladas, uma *biblioteca* consiste em um arquivo em formato binário que tem como objetivo final a viabilização do conceito de reuso de código. Ela possui uma série de definições recursos (comumente funções e variáveis, além de outros elementos) que podem ser "anexados" ao programa do usuário, com o intuito de formar um executável. Com isto, o usuário não precisa criar todos os recursos necessários para seu programa, mas sim, reaproveitar as implementações que possuem o mesmo intuito, feitas por outros desenvolvedores.

Na linguagem C/C++, o primeiro conceito de biblioteca (atrelado ao conceito de reuso de código) criado foi o de **biblioteca estática**. Com o surgimento do conceito de sistemas operacionais *multitasking*, o conceito de **biblioteca dinâmica** veio à tona[35].

C.1 BIBLIOTECA ESTÁTICA

Este tipo consiste simplesmente em um conjunto de **arquivos objeto**, que formam um binário ao final. Deste modo, usa-se um compilador para criar os arquivos objetos. Com o resultado, basta apenas usar uma aplicação que junte estes arquivos em um pacote. No Linux, utiliza-se comumente a aplicação **ar** (archiver tool) para esta finalidade. O pacote resultante deve estar com extensão "**.a**". Comparado a um binário executável, a diferença entre sua natureza e de uma biblioteca estática é bem perceptível, pois esta nem sequer passou pela fase da vinculação (*linking*, em inglês).

Após o processo de formação do pacote de biblioteca, ela deve ser passada, no momento da compilação, para a aplicação que irá usá-la. Assim, o executável final conterá todos os arquivos objeto, tanto o da aplicação quanto o da biblioteca. Note que, caso haja alguma alteração na biblioteca, as aplicações deverão ser recompiladas com a nova versão dela.

---

[35] STEVANOVIC, Milan. The Impact of Reusing Concept. In:_____. **Advanced C and C++ Compiling.** California, EUA: Apress Berkeley, 2014, cap. 4.

C.2 BIBLIOTECA DINÂMICA

Apesar da nomenclatura "biblioteca" para ambos os casos sugerir certa aproximação, a biblioteca dinâmica tecnicamente se aproxima muito de um binário executável do que o faz a biblioteca estática[36]. Isso se deve ao fato de que, para sua criação, a fase da vinculação é necessária, em contraste com a biblioteca estática. Basicamente, o principal fator que difere um binário executável de uma biblioteca dinâmica é que esta não possui um **ponto de entrada** de programa, isto é, uma rotina que inicialize sua execução.

Várias funções no âmbito C/C++ são requisitadas no desenvolvimento de diversas aplicações (a biblioteca padrão C, por exemplo). Desse modo, percebeu-se que era desnecessário que todos os executáveis desses programas sempre contivessem o código dessas bibliotecas. Caso o usuário deseje, por exemplo, implementar a funcionalidade de impressão de documentos na sua aplicação, há uma biblioteca que permite sua manipulação. Se ela for vinculada estaticamente, toda alteração nesta biblioteca significa que todas as aplicações que a usam precisarão ser recompiladas. Criou-se, então, uma forma de referenciar tais arquivos **em tempo de execução**, de modo que eles não precisem estar presente no executável, mas no sistema operacional. Formou-se o conceito de biblioteca dinâmica. Com isso, considerando o cenário da impressão de documentos, caso haja alguma alteração dos arquivos da biblioteca, os executáveis que dependem dela ficarão possivelmente intactos, sem necessidade (ou necessidade mínima) de alteração.

Bibliotecas dinâmicas possuem extensão "**.so**" e, em ambientes Linux, são geralmente criadas também pelas ferramentas que normalmente também são responsáveis pela compilação (GCC, G++, por exemplo).

---

[36] STEVANOVIC, Milan. *Op. cit.,* cap. 4.

APÊNDICE D - LINHA DE COMANDO NO LINUX

Sprof se utiliza constantemente dos conceitos de linha de comando, tanto na aplicação perfiladora quanto no *script* de inserção da instrumentação. Em computação, mais especificamente nos sistemas Linux, quando nos referimos à linha de comando, estamos frequentemente falando do ambiente *shell,* que consiste na interface de linha de comando, acessada, geralmente, através de um *terminal*.

A *shell* nada mais é do que um programa que recebe e interpreta comandos, efetuados pelo usuário, e os passa para o *kernel* do sistema operacional, de forma que ele o trate e execute alguma ação[37].

D.1 *SHELL SCRIPTS*

A *shell* se comporta de forma interativa. A cada comando dado pelo usuário, ela o executa imediatamente e retorna o resultado. Porém, como em qualquer linguagem interpretada, é possível criar *scripts* que contenham uma série de comandos e instruções e, assim, sejam todos executados em sequência. Possuem extensão **.sh**.

Neles, é possível fazer diversas tarefas, inclusive àquelas comuns às linguagens compiladas: executar programas instalados na máquina, executar comandos internos ao programa *shell*, usar variáveis, criar laços de repetição, expressões condicionais e funções etc.

D.2 FERRAMENTAS DE MANIPULAÇÃO DE ARQUIVOS E TEXTO

Os sistemas operacionais baseados em Linux geralmente possuem uma gama enorme de ferramentas que podem ser instaladas. Elas monitoram redes, alteram arquivos binários, compilam códigos, fazem manipulação de texto etc. Todas elas serão executadas de modo muito similar pela *shell*. Basta passar o caminho do arquivo executável no sistema, podendo acrescentar parâmetros de linha de

[37] TRACY, Robb H. **LPI Linux Essentials Certification All-in-One Exam Guide**. New York, EUA: McGraw-Hill, 2013. Cap. 5.

comando a este arquivo, os quais personalizarão sua execução.

D.2.1 Manipulação de arquivos

Um grupo de aplicações importante e vastamente utilizado por inúmeros desenvolvedores é o de manipulação de arquivos. Estes programas, como padrão, recebem um arquivo texto de entrada e operam alguma ação sobre ele de modo a extrair informações sem alterá-lo. Dentre os principais utilitários para manipular arquivos, estão os comando g*rep*[38]*, cut, sort, cat* e *wc*[39]*.*

Grep é uma ferramenta que, dada uma lista de padrões de texto, procura pelas linhas que os contém, dentro de um arquivo de entrada. Caso ela ache-os, ele copia a linha para o terminal (saída padrão) ou produz algum arquivo de saída, de acordo com o que o usuário ordenou.

Cut tem como função principal cortar certas partes de uma linha de texto, que podem ser identificadas pelo número de *bytes*, caracteres ou campos separados por um delimitador.

Por fim, os comando *sort*, *cat* e *wc* (sigla para *word count*) fazem, respectivamente, ordenação de arquivo (por letra ou numérica), concatenação, isto é, junção do conteúdo dos arquivos, um logo após o outro, e contagens, como o número de linhas, palavras ou caracteres do arquivo.

D.2.2 Manipulação de texto

Um outro grupo de aplicações mais específico do que o anterior é o de manipulação de textos. Eles também recebem um arquivo texto de entrada e devolvem um arquivo texto de saída, alterado de acordo com os parâmetros passados pelo usuário. Os comando *sed, perl* e *awk* são alguns vastamente utilizados.

---

[38] Free Software Foundation. **GNU Grep 2.26**. Disponível em: <http://www.gnu.org/software/grep/manual/grep.html#Copying>. Acesso em: 18 de nov. de 2016.
[39] The Open Group. **Shell and Utilities**: Detailed Toc. Disponível em: <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html>. Acesso em: 26 de nov. de 2016.

Sed (stream editor) é uma aplicação usada para desempenhar transformações de texto em um fluxo de entrada. Este fluxo pode ser de um arquivo ou da saída de outro comando (como o *grep*). Além disso, o usuário deve mandar instruções de edição, que serão aplicadas sobre o fluxo de entrada e podem ser implementados em forma de *script*. Nesse sentido, *sed* funciona como um filtro de texto, deletando, alterando e inserindo caracteres, palavras e até linhas inteiras de texto[40].

Já *perl* (Practical Extraction and Report Language) é, na verdade, uma linguagem de programação de propósito geral. Apesar disso, ela foi originalmente pensada para analisar arquivos de texto, extrair informações dele e imprimir em forma de relatório. Desse modo, esse fim ainda está fortemente presente no uso da linguagem[41].

Nos sistemas Linux, é possível encontrar a aplicação *perl*, que interpreta arquivos (*scripts*) escritos nessa linguagem. Esta aplicação permite que um programa curto em *perl* possa ser executado diretamente na linha de comando, isto é, sem a necessidade de um *script* para ser interpretado.

[40] PERMENT, Eric. **The Sed FAQ**: Frequently Asked Questions about sed, the stream editor. Disponível em <http://sed.sourceforge.net/sedfaq.html>. Acesso em: 18 de nov. de 2016.
[41] WALL, Larry. **Perl version 5.24.0 documentation.** Disponível em <http://perldoc.perl.org/perl.pdf>. Acesso em: 20 de nov. de 2016.