**DARLON VASATA**

# Acesso Paralelo a Arquivos em Sistemas Multiprocessadores

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia Elétrica.

São Paulo
2010

**DARLON VASATA**

# Acesso Paralelo a Arquivos em Sistemas Multiprocessadores

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia Elétrica.

Área de concentração:
Sistemas Digitais

Orientador:
Liria Matsumoto Sato

São Paulo
2010

*I'm not saying right is wrong, it's up to us to make*

Oasis - *The Masterplan*

# Dedicatória

Dedico este trabalho aos meus pais, Bernardo e Bernardete Vasata.

# Agradecimentos

Agradeço a todos aqueles que tiveram paciência. Paciência por me esperar voltar de tão longe para que possamos nos encontrar e também paciência por aguentarem ao meu lado nos momentos de estresse, trabalho e preocupações.

Agradeço à minha namorada Joice, que com todo seu carinho e atenção me acolheu e entendeu durante os momentos difíceis e riu comigo nos momentos de felicidade, mesmo muitas vezes não podendo me alegrar com sua presença física. Você é uma pessoa muito importante para mim, e exerce um papel fundamental na minha vida.

Agradeço aos meus pais Bernardo e Bernardete e a meu irmão Daniel, por todo seu esforço e esmero para me fazer chegar a até o lugar onde estou. Nada seria como é sem eles, sua fé e apoio. Amo todos vocês.

Agradeço também à minha orientadora Liria Matsumoto Sato, pelo seu apoio, dedicação incondicional, motivação e conselhos de imensa sabedoria. As lições que aprendi com você não se restringem apenas ao trabalho, pois servem para toda a vida.

Agradeço a todos os colegas de laboratório do LAHPC da USP, pelo seu apoio e acolhimento, onde serviram como família no lugar daquela que ficou longe. Sem dúvida alguma, os momentos que passei na presença de vocês deixarão muitas saudades e sou eternamente grato a todo o apoio que vocês me proporcionaram.

Agradeço também a aqueles que puderam me ajudar na chegada a São Paulo, fornecendo um espaço dentro de suas casas e sem mesmo esperar algo em troca.

Nada seria como é sem os colegas de treino de Sanshou, pelo seu incentivo e apoio. Obrigado a todos aqueles que dedicaram seu tempo em me ajudar ou mesmo por compartilhar sua alegria. Companheirismo que com certeza não será esquecido.

Agradeço à Capes pelo seu apoio financeiro, cuja contribuição de papel fundamental para a elaboração deste trabalho.

E principalmente, agradeço ao nosso grande Deus, pois foi ele quem proporcionou que todas estas maravilhosas pessoas fizessem parte de minha vida. Tais pessoas e suas ações comigo servem no mínimo, como prova que o Senhor existe.

# Resumo

Este trabalho apresenta um estudo sobre o gargalo no acesso a disco sob arquiteturas com sistemas multiprocessadores e processadores *multicore* e apresenta uma proposta de solução para o problema e sua implementação. A solução desenvolvida é chamada de FPA.

A estratégia desenvolvida no trabalho aproveita os recursos de memória compartilhada da arquitetura e utiliza buffers de dados compartilhados entre as diversas *threads* das aplicações, bem como tenta otimizar o acesso ao disco rígido. Para que seja diminuída a concorrência no acesso a disco, é permitido que apenas uma *thread* realize leituras ou gravações no disco por vez, e nas situações de escrita é feita a sobreposição entre o processamento e o envio dos dados ao disco. As operações de escrita são realizadas por uma *thread* adicional, que é inicializada e gerenciada pela FPA.

Foram executados diversos testes com diferentes números de *threads*, tamanhos de arquivo e quantidade de processamento de cálculos na aplicação. Os resultados apresentaram ganho de desempenho com a FPA em diversas situações, onde um dos fatores mais significativos para o ganho de desempenho é a quantidade de processamento existente na aplicação. De modo geral, as operações de leitura apresentaram ganho de desempenho em relação ao uso de *streams* nas situações onde existe pouco processamento de cálculos na aplicação, e as operações de escrita mostraram vantagem nas situações onde existe uma quantidade de processamento de cálculos maior na aplicação.

Nas operações de escrita é obtido vantagem com a sobreposição entre processamento e escrita de dados. Porém, quando a quantidade de processamento de cálculos é muito alta, o tempo de espera pela gravação em disco torna-se irrisório perante o tempo gasto no processamento dos cálculos, ocasionando perda de desempenho da FPA em relação aos *streams*.

Palavras-chave: Gargalo no acesso a disco. Mapeamento de arquivos em memória.

# Abstract

This work presents a study on disk access bottlenecks for multiprocessor architecture systems and also proposes a specification to solve this problem and its implementation. The developed strategy is called FPA.

The developed strategy makes use of shared memory architecture and uses shared data buffers among several application threads and also tries to optimize disk access time. In order to decrease disk concurrency, is allowed to just one application thread to read/write simultaneously and processing thread operations run parallelly to disk operations. Write operations are performed by an internal thread, which is started and managed by FPA.

Several tests were performed concerning different amounts of application threads instances, file sizes and computing instructions. Results show performance gain with FPA in several situations. The main factor that affects performance is the amount of computing instructions on each application. In general, read operations showed a better performance compared to use of streams in situations where there is few computing instructions in the application, and write operations showed advantages on situations where there are more computing instructions in the application.

Write operations are advantageous using data writing and computing instructions overlap. However, in cases of a large amount of computing instructions compared to disk operations, the time to perform disk operations becomes too few compared to computing time, occurring FPA performance loss compared to streams.

Keywords: Disk access bottleneck. Memory mapped files.

# Sumário

**Lista de Figuras**

**Lista de Tabelas**

**Lista de abreviaturas e siglas**

**Lista de Símbolos**

**1 Introdução** **19**

1.1 Motivação . . . . . 19

1.2 Objetivos . . . . . 20

1.3 Metodologia utilizada . . . . . 21

1.4 Trabalhos relacionados . . . . . 22

1.5 Nota sobre os termos utilizados no trabalho . . . . . 24

1.6 Estrutura do trabalho . . . . . 25

**2 Arquiteturas paralelas com memória compartilhada e acesso a disco** **26**

2.1 Multiprocessadores . . . . . 27

2.2 *Multicore* . . . . . 28

2.3 O problema da concorrência . . . . . 28

2.4 Programação para sistemas multiprocessadores . . . . . 29

2.4.1 *Threads* . . . . . 30

2.5 Os discos e o sistema operacional . . . . . 31

2.6 Rotinas de acesso a disco . . . . . 32

2.6.1 Open . . . . . 32

2.6.2 Mmap . . . . . 33

2.6.3 Fopen . . . . . 34
2.6.4 Ffio . . . . . 36
2.7 Concorrência a dados e sincronização de tarefas . . . . . 36
2.7.1 Semáforos . . . . . 37

**3 Estratégia para otimização no acesso a disco 38**
3.1 Estratégia proposta . . . . . 39
3.2 Implementação . . . . . 41
3.2.1 Mapeamento do arquivo e alocação de buffers . . . . . 42
3.2.2 Rotinas e estruturas . . . . . 43
3.2.3 Compatilidade entre funções . . . . . 50
3.2.4 Seções críticas e sincronização . . . . . 51
3.2.5 Limites de memória e liberação de buffers . . . . . 51
3.2.6 Organização da biblioteca . . . . . 53
3.2.7 Documentação . . . . . 54

**4 Resultados e análises 55**
4.1 Ambiente de execução . . . . . 55
4.2 Testes elaborados . . . . . 57
4.2.1 Informações sobre os testes . . . . . 57
4.2.2 Particionamento uniforme do arquivo . . . . . 62
4.2.3 Particionamento dinâmico do arquivo . . . . . 81
4.2.4 Arquivos distintos por *thread* . . . . . 100
4.2.5 Diferentes parâmetros . . . . . 113
4.2.6 *Stream* compartilhado e *stream* privado . . . . . 115
4.3 Busca em arquivo . . . . . 118
4.4 Leitura de arquivos na aplicação OM . . . . . 121
4.5 Discussão . . . . . 123
4.5.1 Vantagens da FPA . . . . . 124

4.5.2 Desvantagens da FPA . . . . . 125

**5 Conclusões** **126**

5.1 Trabalhos futuros . . . . . 128

5.2 Publicações . . . . . 128

**Referências Bibliográficas** **130**

**Glossário** **134**

**Apêndice A – Documentação da implementação** **136**

# Lista de Figuras


2.1 Multiprocessadores . . . . . 27

2.2 *Multicore* . . . . . 28

3.1 Acesso e mapeamento de arquivos em memória . . . . . 40

3.2 Interfaceamento da FPA com diversas *threads* de aplicações . . . . . 43

3.3 Fluxograma da rotina fpa_start . . . . . 44

3.4 Fluxograma da rotina fpa_open . . . . . 45

3.5 Fluxograma da rotina fpa_seek . . . . . 46

3.6 Fluxograma da rotina fpa_write . . . . . 47

3.7 Fluxograma da rotina fpa_read . . . . . 48

3.8 Fluxograma da rotina fpa_flush . . . . . 49

3.9 Fluxograma da rotina fpa_close . . . . . 50

3.10 Fluxograma da *thread* de escrita de dados em disco . . . . . 52

3.11 Organização da biblioteca FPA . . . . . 53

4.1 Cálculos executados nos testes . . . . . 58

4.2 Medição de tempo com a rotina gettimeofday . . . . . 59

4.3 Teste utilizando particionamento uniforme do arquivo . . . . . 62

4.4 Pseudocódigo da leitura utilizando mapeamento uniforme do arquivo . 63

4.5 Leituras utilizando arquivos de 4 GB . . . . . 64

4.6 Leituras com arquivos de 4 GB e 4 *threads* . . . . . 66

4.7 Leituras utilizando 4 *threads* . . . . . 67

4.8 *Speedup* da leitura utilizando particionamento uniforme do arquivo com arquivos de 4 GB, 4 *threads* e processamento de 5000 iterações . 68

4.9 Eficiência da leitura utilizando particionamento uniforme do arquivo . 68

4.10 Leituras com arquivos de 4 GB na máquina do LAHPC . . . . . 70

4.11 Leituras com arquivos de 4 GB e 4 *threads* na máquina do LAHPC . . 71

4.12 *Speedup* da leitura utilizando particionamento uniforme do arquivo executados na máquina do LAHPC, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 71

4.13 Eficiência da leitura utilizando particionamento uniforme do arquivo na máquina do LAHPC . . . . . 72

4.14 Pseudocódigo da escrita utilizando mapeamento uniforme do arquivo 73

4.15 Escritas de arquivos de 1 GB . . . . . 73

4.16 Escritas de arquivos de 1 GB utilizando 4 *threads* . . . . . 75

4.17 Escritas utilizando mapeamento uniforme dos arquivos, com 4 *threads* 76

4.18 *Speedup* da escrita utilizando particionamento uniforme do arquivo, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 76

4.19 Eficiência da escrita utilizando particionamento uniforme do arquivo, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 77

4.20 Pseudocódigo da leitura e escrita utilizando mapeamento uniforme do arquivo . . . . . 78

4.21 Leitura e escrita com arquivo de 4 GB e processamento de 5000 . . . 79

4.22 Leitura e escrita com arquivo de 4 GB e 4 *threads* . . . . . 79

4.23 *Speedup* da leitura e escrita utilizando mapeamento uniforme do arquivo, com arquivos de 4 GB e processamento de 5000 iterações . . . 80

4.24 Eficiência da leitura e escrita utilizando particionamento uniforme do arquivo, com arquivos de 4 GB e processamento de 5000 iterações . . 81

4.25 Pseudocódigo da leitura utilizando mapeamento dinâmico do arquivo 82

4.26 Leitura utilizando particionamento dinâmico do arquivo com 4 *threads* e arquivo de 4 GB . . . . . 83

4.27 Leitura utilizando arquivos de 4 GB e 4 *threads* . . . . . 85

4.28 Leitura utilizando 4 *threads* . . . . . 85

4.29 Leitura utilizando processamento de 5000 iterações, arquivos de 4 GB e buffer de 16 MB . . . . . 86

4.30 *Speedup* da leitura utilizando pilha de dados, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 87

4.31 Eficiência da leitura utilizando pilha de dados, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 87

4.32 Teste na máquina do LAHPC com leitura utilizando mapeamento dinâmico do arquivo com 4 *threads* e arquivos de 4 GB . . . . . 88

4.33 Teste na máquina do LAHPC com leitura utilizando arquivos de 4 GB e 4 *threads* . . . . . 89

4.34 Testes na máquina do LAHPC com leitura utilizando 4 *threads* . . . . 90

4.35 *Speedup* da leitura utilizando particionamento dinâmico de arquivos na máquina do LAHPC, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 90

4.36 Eficiência da leitura utilizando mapeamento dinâmico do arquivo na máquina do LAHPC, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 91

4.37 Pseudocódigo da escrita utilizando mapeamento dinâmico do arquivo 92

4.38 Escrita utilizando arquivo de 4 GB e processamento de cálculo de 5000 iterações . . . . . 93

4.39 Escrita utilizando arquivo de 4 GB e 4 *threads* . . . . . 93

4.40 Escrita utilizando processamento de 5000 iterações e 4 *threads* . . . . 95

4.41 *Speedup* da escrita utilizando mapeamento dinâmico do arquivo, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 95

4.42 Eficiência da escrita utilizando pilha de dados, com arquivos de 4 GB e processamento de 5000 iterações . . . . . 96

4.43 Pseudocódigo da leitura e escrita utilizando mapeamento dinâmico do arquivo . . . . . 97

4.44 Leitura e escrita utilizando arquivo de 1 GB e processamento de 5000 iterações . . . . . 98

4.45 Leitura e escrita utilizando arquivo de 1 GB e 4 *threads* . . . . . 98

4.46 Leitura e escrita utilizando processamento de 5000 iterações e 4 *threads* 99

4.47 *Speedup* da leitura e escrita utilizando pilha de dados, com arquivos de 1 GB e processamento de 5000 iterações . . . . . 99

4.48 Eficiência da leitura e escrita utilizando pilha de dados, com arquivos de 1 GB e processamento de 5000 iterações . . . . . 100

4.49 Pseudocódigo da leitura utilizando arquivos distintos por *thread* . . . 101

4.50 Leitura utilizando arquivos individuais por *thread* com 4 GB . . . . . 102

4.51 Leitura de arquivos individuais por *thread* com arquivos de 4 GB e 4 *threads* . . . . . 103

4.52 Leitura utilizando 4 *threads* . . . . . 104

4.53 *Speedup* da leitura utilizando arquivos individuais por *thread*, obtido através de execuções com arquivos de 1 GB e processamento de 5000 iterações . . . . . 104

4.54 Eficiência da leitura utilizando arquivos individuais por *thread*, com arquivos de 1 GB e processamento de 5000 iterações . . . . . 105

4.55 Pseudocódigo da escrita utilizando arquivos distintos por *thread* . . . 106

4.56 Escritas utilizando arquivos de 1 GB e processamento de 5000 iterações 106

4.57 Escritas utilizando arquivos de 4 GB e 4 *threads* . . . . . 107

4.58 Escritas utilizando processamento de 5000 iterações e 4 *threads* . . . 107

4.59 *Speedup* da escrita utilizando arquivos individuais, com arquivos de 1 GB e processamento de 5000 iterações . . . . . 108

4.60 Eficiência da escrita utilizando arquivos individuais . . . . . 108

4.61 Pseudocódigo da leitura e escrita utilizando arquivos distintos por *thread* 110

4.62 Leitura e escrita utilizando arquivos de 1 GB e processamento de 5000 iterações . . . . . 111

4.63 Leitura e escrita utilizando arquivos de 1 GB e 4 *threads* . . . . . 111

4.64 Leitura e escrita utilizando processamento de 5000 iterações e 4 *threads* 112

4.65 *Speedup* da leitura e escrita utilizando arquivos individuais, com arquivos de 1 GB e processamento de 5000 iterações . . . . . 112

4.66 Eficiência da leitura e escrita utilizando arquivos individuais, com arquivos de 1 GB e processamento de 5000 iterações . . . . . 113

4.67 Leitura utilizando diferentes tamanhos de buffers e diferentes valores para *LA* . . . . . 114

4.68 Escrita utilizando diferentes tamanhos de buffers e diferentes valores para *LA* . . . . . 115

4.69 Pseudocódigo da escrita de dados utilizando *stream* compartilhado . . 116

4.70 Comparação entre *stream* privado, *stream* compartilhado e FPA em escritas de arquivos de 4 GB e utilizando processamento de 5000 iterações 117

4.71 Comparação entre *stream* privado, *stream* compartilhado e FPA em escritas de arquivos de 4 GB e utilizando 4 *threads* . . . . . 118

4.72 Pseudocódigo da busca em arquivos . . . . . 119

4.73 Busca em arquivos . . . . . 120

4.74 Busca em arquivos na máquina do LAHPC . . . . . 121

4.75 Tempo de execução do *software* de modelagem do projeto OM . . . . 122

4.76 Porcentagem de ganho do uso da FPA em relação aos *streams* no *software* de modelagem do projeto OM . . . . . 123

# Lista de Tabelas

4.1 Configurações das máquinas do *cluster* OM . . . . . 56

4.2 Configurações da máquina do LAHPC . . . . . 57

4.3 Tempo gasto para diversos valores em processamento, no *cluster* OM 59

4.4 Tempo gasto para diversos valores em processamento, na máquina do LAHPC . . . . . 60

# Lista de abreviaturas e siglas

**API** *Application Program Interface*

**CCE** Centro de Computação Eletrônica

**CPU** *Central Processing Unit*

**CMP** *Chip Multiprocessor*

**CRIA** *Centro de Referência de Informação Ambiental*

**Epusp** Escola Politécnica da Universidade São Paulo

**FPA** *File for Parallel Access*

**Ffio** *Flexible File I/O*

**FSB** *Front Side Bus*

**GB** Gigabyte

**GCC** GNU Compiler Collection

**I/O** *Input/Output*

**ID** Identificador da *thread*

**Inpe** Instituto Nacional de Pesquisas Espaciais

**IEEE** Instituto de Engenheiros Eletricistas e Eletrônicos

**Libc** Biblioteca Padrão do C

**ICC** Intel C++ Compiler

**KB** Kilobyte

**LAHPC** *Laboratory of Architecture and High Performance Computing*

**MB** Megabyte

**Mems** *Microelectromechanic System*

**MPI** *Message Passing Interface*

**OM** *Open Modeller*

**Posix** *Portable Operating System Interface for Unix*

**Sata** Serial ATA

**SGI** *Silicon Graphics*

**SO** Sistema Operacional

**TLP** *Thread-level Parallelism*

**USP** Universidade de São Paulo

# Lista de Símbolos

| | |
|---|---|
| $dp$ | desvio padrão |
| $dp_{max}$ | desvio padrão máximo |
| $dp_{min}$ | desvio padrão mínimo |
| $dp_t$ | desvio padrão temporário |
| $LA$ | Limite de Aviso de memória |
| $LM$ | Limite Máximo de memória |
| $ma$ | média aritmética |
| $ma_t$ | média aritmética temporária |

# 1 Introdução

Diversos problemas do mundo real que demandam e processam grandes quantidades de informações possuem cálculo manual inviável e podem ser solucionados pelos computadores atuais, como previsão do tempo, simulações de movimento de fluidos (sangue, água, vento, etc), propagação de doenças, entre outros. Não obstante, tais problemas podem demandar um tempo excessivo de execução, chegando a inviabilizar a aplicação e sua solução.

Mesmo sabendo que durante as últimas décadas a Lei de Moore tem sido confirmada nos sistemas computacionais, onde estes possuem uma capacidade computacional que ultrapassa a capacidade existente nos supercomputadores da geração humana anterior, outros elementos essenciais da computação não acompanharam tal evolução. Apesar das tecnologias com hierarquias de memória cada vez maiores e mais complexas, o armazenamento de informações tem se tornado o principal fator da Lei de Amdahl (PATTERSON; HENNESSY, 2003).

O alavancamento de todo o poder computacional disponível pelos *hardwares* com múltiplos processadores ou processadores *multicore* demanda novas ferramentas e novas formas de pensar da indústria de *software* (SUTTER; LARUS, 2005). Tal *hardware* é melhor aproveitado através da paralelização de código, que consiste em identificar regiões da aplicação que podem ser executadas simultaneamente (MORETTI; BITTENCOURT, 2004).

## 1.1 Motivação

Durante os últimos 20 anos, o aumento do desempenho nos discos rígidos tem sido significativamente menor que o das Unidades Centrais de Processamento (CPU's[1]). De Janeiro de 1996 a Janeiro de 2006 o aumento de desempenho das CPU's foi de 30 vezes, enquanto que os discos rígidos mantiveram uma escala de 1.3 vezes (MATTHEWS et al., 2008).

[1] Do inglês *Central Processing Unit*

Devido à necessidade cada vez maior de processamento, as tecnologias computacionais avançam cada vez mais, onde atualmente possuímos processadores com diversos núcleos de processamento ou mesmo máquinas com diversos processadores. Estes são capazes de executar diversas tarefas simultaneamente, diferente do mecanismo que intercala as execuções das aplicações (*timesharing*), como acontece com os monoprocessadores. Um processador *multicore* contém vários núcleos, enquanto que um computador com vários processadores é classificado como multiprocessador (CULLER; SINGH, 1999). Ambas as tecnologias evoluíram de tal forma que outros dispositivos computacionais não acompanharam seu desenvolvimento.

O fato das arquiteturas com multiprocessadores permitirem que diversas linhas de execução (*threads*) operem simultaneamente, proporciona um melhor desempenho em aplicações com suporte a paralelismo. Porém, como o acesso ao disco é feito utilizando apenas um canal de comunicação, se diversas destas linhas de execução acessarem o disco simultaneamente, os dados obtidos não poderão ser fornecidos a todas as *threads* ao mesmo tempo. Dessa forma para aplicações que processam quantidades massivas de informações, acessam e armazenam seus dados a partir de arquivos, torna-se evidente a existência de um gargalo no acesso ao disco. Outro fator agravante é o mecanismo físico de acesso às informações no disco, onde muitas movimentações do cabeçote podem reduzir significativamente o desempenho de aplicações, se estas necessitarem acessar informações que residam em pontos distantes entre si no disco.

## 1.2 Objetivos

Conhecendo a problemática do gargalo no acesso a disco em sistemas dotados de multiprocessadores, o presente trabalho proporciona um estudo sobre as diversas formas de acesso a disco em aplicações paralelas, apontando suas problemáticas e possíveis soluções. De forma a viabilizar a execução de aplicações com tempo de processamento inviáveis, o atual trabalho especifica e implementa uma biblioteca denominada FPA[2], de forma que esta forneça um mecanismo de auxílio na diminuição do tempo gasto no acesso de leitura e escrita a dados no disco rígido em arquiteturas com sistemas multiprocessadores.

A implementação da FPA consiste em utilizar estratégias de otimização no acesso a disco, como permitir o acesso de apenas uma *thread* por vez e a utilização de buffers para os dados da aplicação. No decorrer do trabalho, foram elaborados diversos testes sobre as formas de acesso a arquivos, bem como uma análise e estudo sobre os

[2]Abreviação do termo em inglês *File library for Parallel Access*

resultados obtidos em tais testes.

## 1.3 Metodologia utilizada

A metodologia do trabalho consistiu em inicialmente realizar uma revisão bibliográfica sobre o tema, o problema e as possíveis soluções. Detalhadamente, a ordem do trabalho realizado aconteceu da seguinte forma:

- Estudo e pesquisa sobre o problema na literatura;
- Estudo sobre as arquiteturas multiprocessadores;
- Pesquisa sobre o tratamento para o problema na literatura;
- Conhecimento sobre as chamadas de sistema para acesso a disco, bem como primitivas básicas, seus parâmetros e configurações;
- Elaboração de aplicativos para testes, com diversas formas de acesso a arquivos;
- Testes com a primitiva `open`;
- Testes com a primitiva `fopen`;
- Testes utilizando o *middleware* `Ffio`;
- Testes utilizando o mapeamento de arquivos em memória com Mmap;
- Testes utilizando mapeamento de arquivos em memória, com acesso controlado ao disco utilizando semáforos;
- Análise e comparação dos resultados dos testes;
- Baseando-se no resultados obtidos, elaboração de uma estratégia de solução eficaz para o problema;
- Implementação da estratégia;
- Testes da implementação, utilizando as rotinas de escrita;
- Testes da implementação, utilizando as rotinas de leitura;
- Comparação dos resultados dos testes com as rotinas da biblioteca `stdio`, da LIBC;
- Tornar a implementação utilizada disponível para a comunidade, de forma que sua utilização seja o mais transparente possível para o usuário.

Dentre as soluções testadas, algumas não tiveram desempenho satisfatório com os parâmetros testados. Entre eles estão o uso da chamada de sistema `open` e o software `ffio`. Estas soluções são apresentadas no trabalho, porém não são utilizadas na implementação da biblioteca FPA.

## 1.4 Trabalhos relacionados

Na literatura são encontrados alguns trabalhos relacionados, como o de Martens e Katchabaw (2006), que propõe uma nova forma de escalonamento de disco e apresenta uma forma de análise dos acessos ao sistema de armazenamento. Sua estratégia consiste em monitorar em tempo real a atividade do disco para futuramente utilizar esses dados e descobrir padrões de acesso, e utilizar um sistema de contabilização para detectar possíveis tendências de acesso realizadas pelos processos existentes no sistema. Após feita uma análise sobre os acessos, seleciona ou otimiza um algoritmo de escalonamento para o sistema com base em heurísticas ou critérios que tornam o algoritmo de escalonamento como o mais recomendado para que um melhor desempenho sobre a situação seja obtido. Seus resultados foram positivos, indicando que sua estratégia possui grande potencial na melhora de desempenho de sistemas.

A pesquisa de Docy, Hong e Potkonjak (1996) apresenta um estudo sobre a evolução da computação e o problema do gargalo do disco, abordando informações sobre a complexidade computacional do acesso ao. Desenvolve uma heurística para atualização de dados e particionamento de disco, de forma que o tempo de espera pelo acesso ao disco seja minimizado. Seus resultados apontaram que a solução desenvolvida reduziu significativamente a movimentação do cabeçote do disco, obtendo ganho de desempenho com as aplicações testadas que utilizam sua implementação.

A preocupação com desempenho de aplicações em sistemas paralelos também é tratada em outros trabalhos. Benson, Long e Pacheco (2003) apresentam o resultado de um estudo experimental sobre aplicações distribuídas com bibliotecas de trocas de mensagens (MPI[3]) e Pthreads[4] utilizando diferentes implementações de rotinas de entrada/saída (I/O[5]), e comparando o desempenho dos resultados obtidos com cada rotina. Seus resultados constataram que utilizando apenas MPI o maior ganho de desempenho é obtido com a utilização de `mmap`[6], enquanto para aplicações não-distribuídas com o uso de Pthreads o maior ganho de desempenho foi atingido com a chamada de sistema `write`.

[3]Do inglês *Message Passing Interface*
[4]Posix *threads*
[5]Do inglês *Input/Output*
[6]*Memory-mapped* I/O

Thakur, Lusk e Gropp (1998) discutem o problema do gargalo no acesso a disco para aplicações paralelas e distribuídas, apontando como sendo este o ponto mais crítico para aplicações envolvendo grandes quantidade de entrada/saída. O trabalho ainda descreve as interfaces de I/O dos sistemas Unix, e discute o quanto estas são inapropriadas para operações de I/O paralelas. Com base nisso apresenta o MPI-IO, que trata-se de um padrão de uma interface de programação desenvolvida com o intuito de obter alto desempenho em operações de I/O em aplicações paralelas.

O trabalho de More et al. (1997) apresenta uma biblioteca para acesso a arquivos em sistemas distribuídos utilizando *threads* para fazer a distinção de fluxo de aplicação e operações de acesso a disco, porém com seu foco voltado para sistemas distribuídos com sistemas de arquivos paralelos. Discute que a vantagem da existência de *threads* que realizam separadamente operações de I/O acontece devido à sobreposição entre o processamento de informações na aplicação e a espera pelo dispositivo de armazenamento em operações de I/O com o disco, enquanto também reduz a sobrecarga do sistema operacional, dado pela escrita e leitura utilizando utilizando maiores quantidades de dados com um menor número de chamadas de sistema envolvendo I/O.

Atuando no nível de sistema de arquivos, Ganger e Kaashoek (1997) desenvolvem e implementam uma nova forma de organização dos *inodes* de arquivos no disco de armazenamento. A idéia principal do trabalho é obter desempenho nas operações de leitura e escrita a arquivos pequenos, obtendo vantagem das operações em que os discos rígidos possuem melhor desempenho (transferência de dados em volume) e evitando as operações onde eles tem menor desempenho (reposicionamento para novas localizações). O trabalho apresenta o *Co-locating Fast File Sistem* (C-FFS), que utiliza as técnicas de *inodes* embarcados e agrupamento explícito. Os *inodes embarcados* possibilitam que os *inodes* de grande parte dos arquivos sejam armazenados no diretório com seu nome correspondente, e o agrupamento explícito permite que blocos de dados de diversos arquivos pequenos sejam alocados adjacentemente. Desta forma a localização espacial do disco é melhor aproveitada, e assim movimentando informações de e para o dispositivo de armazenamento em operações únicas, na maioria dos casos. Seus resultados foram comparados com o sistema de arquivos OpenBSD FFS, e apontaram que o uso do C-FFS reduz o número de chamadas ao disco, com desempenho de 10 a 300 % em algumas aplicações testadas. Apesar de ser um sistema desenvolvido para a obtenção de desempenho com a utilização de arquivos pequenos, o C-FFS não apresentou desempenho inferior utilizando arquivos significativamente grandes.

Em seu trabalho, Dodonov e Guardia (2004) apresentam e implementam uma arquitetura integrada de mecanismos de *cache* e *prefetching* para sistemas de arquivos

distribuídos. Além disso, elabora um algoritmo adaptativo de determinação de padrões de acesso. Seus resultados apontam que com o uso de tais técnicas é possível aumentar o desempenho do sistema, diminuindo a sobrecarga da rede e provendo um mecanismo de otimização para operações de entrada e saída. No trabalho ainda discute a funcionalidade dos mecanismos integrados de *cache* e *prefetching*, avaliando seus algoritmos implementados e a influência da escolha das políticas adequadas no funcionamento de aplicações do usuário.

Pensando na problemática do acesso a disco em servidores multimídia, Rangaswami et al. (2007) apresentam uma arquitetura de acesso a disco utilizando sistemas de armazenamento microeletromecânicos (Mems[7]) (CARLEY; GANGER; NAGLE, 2000) como uma camada intermediária de acesso ao sistema de armazenamento, onde propõe mecanismos de organização de dados e algoritmos de sincronização voltados para aplicações multimídia. Apresenta uma análise análitica de sua hierarquia de armazenamento, onde aponta que pode existir um aumento significativo no desempenho de servidores multimídia com a utilização de equipamentos a um baixo custo.

O trabalho de Ueda, Hirate e Yamana (2009) desenvolve um sistema que trata de utilizar as unidades de processamento ociosas para analisar *logs* de acesso a disco. O intuito da análise é extrair informações de padrões de acesso, e com base nisso otimizar o desempenho das aplicações através do *cache* de arquivos.

## 1.5 Nota sobre os termos utilizados no trabalho

A computação é uma área consideravelmente nova, quando comparada com outras áreas do conhecimento. Devido aos seus grandes avanços em países de idioma inglês, diversos termos utilizados na computação não são convenientes de tradução. Sendo assim, o trabalho possui diversos termos em inglês, devidamente ressaltados em *itálico* e com seus significados descritos no glossario.

Os termos identificados com fonte `typescript` são referentes a aspectos puramente técnicos, como chamadas de sistema, nomes de rotinas, trechos de código, *flags* ou variáveis. Por exemplo, ao falar-se de Mmap, trata-se da técnica de mapeamento de arquivos em memória, enquando `mmap` é referente à chamada de sistema Unix responsável pela realização de tal mapeamento na prática.

O termo "sinais", recorrente principalmente no capítulo que trata da implementação é referente a um mecanismo de aviso para que certas ações sejam tomadas internamente, e não está relacionado a o conceito de sinalização entre processos de sistemas

[7]Do inglês *Microelectromechanic System*

operacionais.

O termo *multicore* é referente ao processador com diversos núcleos[8]. Na literatura em português e espanhol foram encontrados os termos multinúcleo e *multinúcleo*, porém *multicore* foi mantido neste trabalho.

De forma a proporcionar uma visualização melhor dos gráficos no capítulos de resultados, alguns gráficos são apresentados em sub-figuras. Para fins de comodidade de leitura os gráficos são referenciados diretamente na sub-figura, em algumas situações. Por exemplo, ao falar sobre o "gráfico 4.5(g)" é equivalente a dizer "gráfico (g) da figura 4.5".

## 1.6 Estrutura do trabalho

Este trabalho segue a seguinte estrutura:

No capítulo 2 são apresentados os conceitos e ferramentas utilizadas para o desenvolvimento deste documento, bem como uma discussão sobre a escolha de tais ferramentas, suas vantagens e desvantagens.

O capítulo 3 trata de definir a estratégia de solução para o problema do gargalo no acesso a disco em arquiteturas com multiprocessadores, também são mostrados aspectos de implementação e suas peculiaridades.

Os resultados obtidos são mostrados no capítulo 4, onde no decorrer do capítulo é realizada uma análise sobre os resultados obtidos. A seguir, no capítulo 4.5 é realizada uma discussão geral sobre os resultados, os prós e contras da implementação desenvolvida, trabalhos futuros e as publicações obtidas a partir do trabalho.

O capítulo 5 trata da conclusão final do trabalho, com base no resultado obtido. Por último, no apêndice A encontra-se o manual da implementação, desenvolvido para auxílio a futuros usuários e desenvolvedores da biblioteca desenvolvida.

[8]Em inglês, *cores*

# 2 Arquiteturas paralelas com memória compartilhada e acesso a disco

Durante as últimas décadas o desempenho computacional tem crescido exponencialmente, devido ao avanço com tecnologias de implementação, inovações de hardware e otimizações em compiladores (EKMAN; WARG; NILSSON, 2005). Tentar resolver problemas de formas cada vez mais rápidas é uma tendência da evolução da computação, onde busca-se efetuar uma maior quantidade de processamento em um menor espaço de tempo.

Ultimamente o avanço das tecnologias com aplicações multimídia tem resultado em uma grande demanda de processamento aos computadores pessoais (DIEFENDORFF; DUBEY, 1997), fazendo com que necessitassem executar uma quantidade cada vez maior de processamento. Tal demanda impulsionou a indústria tecnológica, chegando a um ponto onde um único processador que acompanhasse a Lei de Moore não fosse mais financeiramente viável em termos de mercado (VACHHARAJANI et al., 2005).

Atualmente possuímos diversas unidades de processamento em uma única máquina, como os processadores *multicore* e os computadores multiprocessadores. Além destes, também são encontrados processadores com mecanismo de suporte à tecnologia *hyperthreading* (BULPIN; PRATT, 2004; KOUFATY; MARR, 2003; TUCK; TULLSEN, 2003), onde é possível que diversas linhas de execução sejam realizadas em um mesmo processador de forma otimizada. No caso do *hyperthreading* é comum os sistemas operacionais indicarem que existem diversos processadores atuando na mesma máquina, porém, tais processadores adicionais são lógicos, e não físicos. Existem replicações de *hardware*, mas não o suficiente para que estes sejam considerados como processadores próprios.

Nos sistemas multiprocessadores, as múltiplas unidades de processamento podem compartilhar diversos componentes da arquitetura (GEER, 2005), como memória, por exemplo. Isto é considerado como uma grande vantagem para os programadores, pois

é possível que os programas sejam executados utilizando o mesmo endereçamento da memória. Desta forma é facilitada a troca de dados entre as múltiplas linhas de execução (AKHTER; ROBERTS, 2006), assim dispensando operações específicas para essa finalidade, como acontece nos sistemas distribuídos.

Além dos computadores multiprocessadores e *multicores*, também estão amplamente difundidas no mercado máquinas utilizando combinações destas tecnologias (KUMAR et al., 2005), fornecendo computadores com diversos processadores, dotados com diversos núcleos. As subseções seguintes abordam tais tecnologias.

## 2.1 Multiprocessadores

A tecnologia de produção de processadores possuía a frequência do *clock* em seu limite, em termos de custo/desempenho (FIELD, 2008; GEPNER; KOWALIK, 2006). Alta frequência requer mais energia e gera mais calor, tornando o aumento de *clock* mais difícil e mais custoso, seja pelo espaço ocupado pelo processador ou na sua refrigeração. Ao invés de produzir processadores mais rápidos para a obtenção de desempenho, optou-se por apenas adicionar outros processadores (BALAKRISHNAN et al., 2005), possibilitando que estes trabalhem em conjunto. A figura 2.1(HUGHES; HUGHES, 2008) ilustra uma arquitetura utilizando dois processadores. Nos multiprocessadores, cada unidade de processamento possui seu próprio *hardware* encapsulado separadamente.

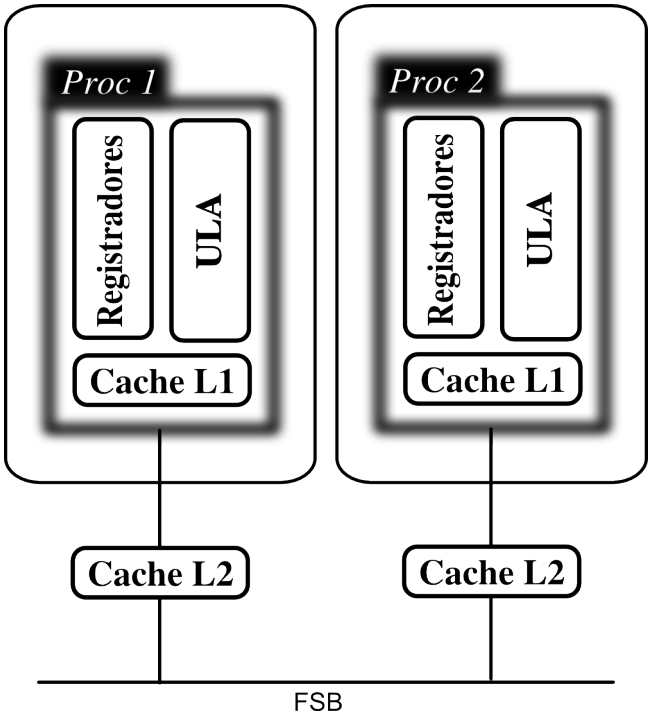



**Figura 2.1:** Multiprocessadores

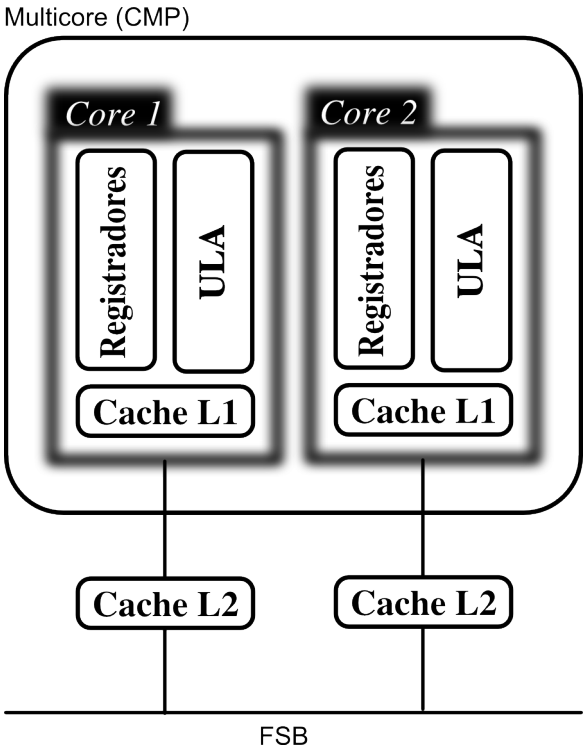



**Figura 2.2:** *Multicore*

## 2.2 *Multicore*

*Multicore* é uma arquitetura computacional que contém diversos processadores em um mesmo *chip*, onde cada processador é chamado de *core* (núcleo). Tal agrupamento tornou-se possível devido ao avanço da capacidade dos *chips*, possibilitando que mais de um processador pudesse compartilhar o espaço de um mesmo *chip* de forma prática (figura 2.2(SPRACKLEN; ABRAHAM, 2005)). Estes designs são conhecidos como Chips Multiprocessadores (CMPs), porque permitem multiprocessamento em um simples *chip*, e *multicore* é apenas um nome popular para CMP (HUGHES; HUGHES, 2008). A idéia de multiprocessamento em um único *chip* não é um conceito novo, pois esta idéia tem sido explorada desde o início dos anos 90. Porém, apenas recentemente, o *multicore* obteve o foco da indústria como sendo o principal método para o aumento do desempenho dos sistemas computacionais.

## 2.3 O problema da concorrência

Dois ou mais eventos são ditos concorrentes se estes ocorrem no mesmo intervalo de tempo. Porém, estes não necessariamente precisam acontecer no mesmo instante, bastando estarem intercalados entre determinados intervalos (HUGHES; HUGHES, 2003). Atualmente os computadores disponíveis dotados com sistema de multiprocessadores dispõem de apenas um canal de comunicação com o sistema de armaze-

namento. Como existem múltiplos trabalhos sendo processados simultaneamente, diversos destes podem acessar o disco em um determinado momento. Torna-se claro que neste instante, o limite físico para o meio de armazenamento torna-se um gargalo, impedindo o computador de atingir um melhor desempenho (UEDA; HIRATE; YAMANA, 2009). Consequentemente, a aplicação necessita esperar que tal acesso seja concluído para avançar à sua próxima instrução. O aspecto importante que aqui deve ser levantado é que durante estes intervalos de tempo de espera a aplicação deverá ficar aguardando pelo recurso, onde o fator aceitável de espera é determinado pela aplicação e o valor do seu resultado.

## 2.4 Programação para sistemas multiprocessadores

O mercado tecnológico proporciona *hardware* capaz de executar diversas tarefas simultaneamente, e cabe aos desenvolvedores de software a criação de aplicativos que aproveitem este *hardware* (GEER, 2007; KALLA; SINHAROY; TENDLER, 2004). A programação para estes sistemas difere da programação sequencial tradicional, pois é necessária a especificação de quais serão as operações executadas em cada unidade do *hardware*.

Os processos e as *threads* já são amplamente difundidos na computação, e comumente utilizados pelos sistemas operacionais (TANENBAUM; WOODHULL, 2000). A utilização destes em sistemas tradicionais monoprocessadores utilizam o mecanismo de *timesharing*, onde a CPU é compartilhada, e alterna seu processamento entre os processos ou *threads* ativos no sistema. Desta forma, o usuário possui a falsa impressão de um sistema que realmente executa diversas tarefas simultaneamente.

Um processo é um programa em execução, incluindo sua atividade em curso representada pelo contador de programa e pelos conteúdos dos registradores do processador. Além disso, um processo geralmente inclui a pilha do processo, que contém dados temporários e uma seção de dados que contém as variáveis globais (SILBERSCHATZ; GALVIN; GAGNE, 2004). Apesar da possibilidade de paralelismo com os processos, as *threads* são o mecanismo mais utilizado para tal, pelo fato destas compartilharem sua seção de código, de dados e outros recursos do sistema operacional com outras *threads*. Desta forma, o foco deste trabalho é mantido sobre as *threads* ao invés dos processos.

### 2.4.1 *Threads*

Uma *thread* é um mecanismo que permite que diferentes sequências de operações sejam executadas simultaneamente de forma independente (LEWIS; BERG, 1996). Estas compreendem um identificador da *thread* (ID), um contador de programa, um conjunto de registradores e uma pilha. Uma determinada aplicação pode ter uma *thread* gerenciando sua interface gráfica, enquanto outra trata de procurar informações pela internet e mais uma cuidando da ortografia do texto, por exemplo. Uma sequencia de instruções em uma *thread* aparece ao processo como sendo um conjunto ordenado ininterrupto de endereços. Em uma máquina com apenas um processador, a execução de diferentes *threads* é feita intercalando estas com o mecanismo de troca de contexto, o qual é realizado pelo sistema operacional (ROBBINS; ROBBINS, 2003). A troca de contexto consiste em salvar o ambiente de execução de uma *thread* ( estados dos registradores, pilha, contador de programa ) para que outra possa ser assumida no lugar, e no futuro a *thread* anterior possa continuar sua execução normalmente do ponto anterior. A grande vantagem sobre o uso de *threads* em sistemas multiprocessadores é o desempenho obtido na execução quase simultânea das *threads* devido à dispensa do mecanismo de troca de contexto, afinal diversos destes podem ser executados em diferentes processadores simultaneamente. De acordo com (TIAN et al., 2003), devido ao advento dos sistemas multiprocessadores e de memória compartilhada, explorar o paralelismo a nível de *thread* (TLP[1]) é uma forma promissora de aumentar o desempenho das aplicações. Segundo (EL-REWINI; ABD-EL-BARR, 2005), em um programa paralelo com memória compartilhada devem existir três construções principais:

a. Criação da tarefa;

b. Comunicação;

c. Sincronização.

Estas construções podem ser obtidas utilizando interfaces de programação de aplicativos (API[2]) de programação para facilitar sua construção. Neste trabalho são citadas e utilizadas as APIs Openmp e Pthreads.

#### Pthreads

Pthreads é um padrão Posix[3] para *threads*, onde este define uma API padrão para a criação e manipulação de *threads* (NICHOLS; BUTTLAR, 1996; LEWIS; BERG, 1996).

[1]Do inglês *Thread-Level Parallelism*
[2]Do inglês *Application Program Interfaces*
[3]Do inglês *Portable Operating System Interface*

As bibliotecas que implementam a Posix *threads* são chamadas Pthreads, sendo muito difundidas no universo Unix e outros sistemas operacionais, como Linux. Para a linguagem de programação C, Pthreads é disponibilizada no arquivo *header* `pthread.h` e uma biblioteca. Inclusive, esta última pode fazer parte de outras bibliotecas, como a `libc` (BARNEY, 2009).

### Openmp

*Open Multi-Processing* (Openmp) é uma API cujo objetivo principal é facilitar a programação paralela *multithread* baseando-se em memória compartilhada. Não se trata de uma nova linguagem de programação, apenas um conjunto de diretivas de compilação que pode ser adicionada para programas sequenciais nas linguagens Fortran, C ou C++. Descreve como os trabalhos devem ser executados em *threads*, aproveitando os diversos núcleos ou processadores da máquina, e como ordenar os acessos aos dados compartilhados entre as *threads* quando necessário (CHAPMAN et al., 2007).

O Openmp consiste de um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente que influenciam na execução do programa. Foi especificado visando algo que seja portável e escalável, fornecendo uma interface de utilização simples e que pudesse ser utilizado para aplicações de pequeno e grande porte.O Openmp usa um modelo *fork/join* na criação e finalização das *threads*, onde existe um fluxo de execução principal e quando especificado, novas *threads* são disparadas para dividir o trabalho. Um recurso interessante do Openmp é que a sincronização entre as *threads* quase sempre ocorre de maneira implícita e automática, simplificando sua utilização (SATO, 2002).

Existem algumas características que diferenciam o Openmp de outras alternativas. Neste não é possível ver como cada *thread* é criada e inicializada, e também não é visível uma função separada contendo o código que cada *thread* executa. Grande parte de suas funcionalidades acontecem de forma transparente, tornando sua utilização bastante simplificada. O modelo Openmp é fortemente baseado em memória compartilhada, porém também existem implementações para outras arquiteturas utilizando troca de mensagens.

## 2.5 Os discos e o sistema operacional

Os discos rígidos são dispositivos físicos de armazenamento e recuperação de dados não-voláteis. São classificados na hierarquia de memórias (STALLINGS, 2002) como dispositivos de armazenamento secundário, pois possibilitam uma grande quantidade

de armazenamento com um menor custo (FERRAZ, 2002). Porém, tem como desvantagem o alto tempo de acesso aos dados, se comparados aos dispositivos de armazenamento primário como a memória, por exemplo.

As aplicações computacionais necessitam realizar operações de I/O por diversas razões. A mais óbvia delas é a gravação dos resultados obtidos pelos programas, e outras razões seriam a inicialização do programa ou então salvar o estado da aplicação para que esta pudesse ser retomada posteriormente (LEVINE; GALBREATH; GROPP, 1993).

Grande parte dos sistemas operacionais possuem buffers dos dados acessados em disco, de forma a otimizar as operações de I/O. Alguns possuem a característica de serem *write-through*, onde os dados são escritos no disco logo após a chamada de escrita. Outros podem ser *write-back*, onde o sistema operacional informa à aplicação que as informações foram gravadas em disco, porém este apenas agendou uma futura escrita. Os buffers dotados com técnicas de *write-back* geralmente são mais eficientes que buffers *write-through*, porém são mais suscetíveis a falhas, como perda de dados nos casos de uma possível falha no sistema operacional ou queda de energia (WIRZENIUS et al., 2008).

Uma forma de garantir a persistência dos dados em aplicações é assegurar que os buffers do sistema operacional tenham sido liberados e as informações estejam gravadas em disco. Os sistemas Unix disponibilizam a chamada de sistema `sync`, que é uma função bloqueante que retorna apenas após a escrita dos dados em disco. Outras possíveis formas são através de *flags* nas funções de escrita, informando que os dados não devem permanecer em buffers do sistema operacional.

## 2.6 Rotinas de acesso a disco

A seguir são descritas algumas das formas de acesso a disco disponibilizadas em sistemas Linux, bem como suas especificações e algumas peculiaridades.

### 2.6.1 Open

A chamada de sistema Unix `open` é uma requisição ao sistema operacional para o acesso aos dispositivos de I/O, tendo como retorno um número inteiro de valor não-negativo chamado de descritor de arquivo, onde este é o índice do arquivo tratado na tabela de arquivos abertos do sistema operacional. O objetivo do descritor de arquivo é ser utilizado como referência a arquivos em outras futuras chamadas de sistema, como

`lseek`, `read` e `write`. `open` possibilita diversas formas de acesso ao arquivo, como leitura, escrita ou *append*.

A estrutura da chamada de sistema `open` é a seguinte:

```
int open(const char *pathname, int flags, mode_t mode);
```

onde `pathname` corresponde ao nome do arquivo e `mode` indica as permissões do arquivo. `flags` é utilizada para informar ao sistema operacional se o arquivo é de leitura ou escrita, dentre outras configurações possíveis. No contexto deste trabalho, vale ressaltar a *flag* `O_SYNC`, onde esta é responsável por informar ao sistema operacional que as rotinas de leitura e escrita sobre este arquivo não devem ser armazenadas em buffers.

### 2.6.2 Mmap

O `mmap` é uma chamada de sistema Unix que possibilita que seja criada uma correspondência de um-para-um entre um determinado arquivo em disco e uma região de memória. Desta forma, os dados do arquivo são manipulados em memória e o SO torna-se responsável pelo envio e recebimento de dados ao disco, a qual também pode ser feita através de uma ordem explícita do programa.

A estrutura da chamada `mmap` é a seguinte:

```
void *mmap(void *addr, size_t len, int prot, int flags,
int fildes, off_t off);
```

onde `addr` corresponde a um endereço de memória, que é passado ao sistema operacional para que neste ponto da memória seja mapeado o arquivo. Setando este parâmetro como `NULL`, o SO se encarregará de encontrar uma região de memória e mapear o arquivo. `len` indica o tamanho da região em bytes, `flags` são parâmetros que indicam as formas de acesso a memória e configurações de compartilhamento, `fildes` é um descritor de arquivo, retornado pela função `open` , e `off` representa o ponto de partida de onde será mapeado o arquivo.

O arquivo é mapeado em memória de forma que os dados estão sujeitos ao escalonamento de páginas de memória do SO. Desta forma, o tamanho de memória utilizado para o mapeamento do arquivo é múltiplo do tamanho de página utilizado pelo SO. Sendo assim, o valor passado ao parâmetro `off` deve também ser múltiplo do número de páginas. Um valor incorreto neste parâmetro ocasionaria em um erro na chamada da função. Neste ponto, convém ressaltar que o sistema operacional onde todos os testes foram realizados possui um tamanho de página de 4 kilobytes (KB).

Segundo (LOVE, 2007) a utilização do Mmap possui diversas vantagens e desvantagens. Entre as vantagens é dito que um bom desempenho é obtido otimizando as chamadas de sistema `read` e `write`, como:

a. possibilita que diversos processos compartilhem os mesmos dados em memória;

b. apesar de potenciais faltas de página, a leitura e escrita para um arquivo mapeado em memória não apresenta nenhuma chamada de sistema adicional ou *overhead* de troca de contexto;

c. o seek para posições do arquivo é feito apenas com simples operaçoes de ponteiros, descartando o uso da chamada de sistema `lseek`.

O mesmo trabalho também aponta diversos pontos fracos do `mmap`, como:

a. como o mapeamento em memória envolve sempre um número inteiro de páginas, este pode ocasionar casos em que possa ser alocado memória desnecessariamente;

b. existe uma sobrecarga na criação e manutenção do arquivo mapeado e das estruturas de dados dentro do *kernel* do sistema operacional. Desta forma, `mmap` é recomendado para casos onde o arquivo possua um tamanho consideravelmente grande, e dessa forma a memória desperdiçada torna-se irrelevante perante o tamanho do arquivo.

### 2.6.3 Fopen

A biblioteca padrão do C (Libc) é um padrão de rotinas e estruturas da linguagem de programação C, que define rotinas para funções matemáticas, envio de sinais ao SO, acesso a arquivos, entre outros.

A função `fopen` é uma rotina disponibilizada na biblioteca `stdio` , definida pela Libc. Realiza o acesso a arquivos utilizando o conceito de *streams*, ao invés dos descritores de arquivo, diretamente. Um *stream* é uma estrutura de dados intermediária para acesso a arquivos, que realiza o armazenamento em buffers dos dados na própria estrutura, ou seja, utiliza o espaço de usuário para otimização do acesso e posteriormente os envia ao SO. Sua especificação é:

```
FILE *fopen(const char *path, const char *mode);
```

onde o parâmetro `path` corresponde à localização do arquivo e `mode` corresponde à forma de acesso ao arquivo. O retorno da função é um ponteiro para uma estrutura de

descritor de arquivo do tipo `FILE`, onde esta contém informações sobre o arquivo, bem como do seu buffer interno. O tamanho inicial do buffer é definido através da constante `BUFSIZ`, definida na biblioteca `stdio.h`. Porém, é possível alterar a localização e tamanho do buffer através da função `setvbuf`, cuja especificação é:

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

onde `stream` indica o descritor do arquivo, `buf` aponta para o novo buffer, e `mode` indica os modos de armazenamento, como *unbuffered*, *line buffered* ou *fully buffered*. O último parâmetro, `size`, indica o tamanho da região do buffer.

Nos sistemas onde foram efetuados os testes deste trabalho, o tamanho do buffer não foi alterado, mantendo seu valor *default* de 8 KB.

### Limitações e problemas dos *streams*

Os *streams* podem ser utilizados em aplicações *multithread* da mesma forma como são utilizadas em aplicações *single-thread*. Porém estes podem ser utilizados em duas situações diferentes, onde ambas são consideradas neste trabalho. Tais situações são analisadas a seguir:

- *Stream* compartilhado: Antes do início das diversas *threads* da aplicação é aberto o arquivo. Desta forma, o *stream* retornado pela chamada `fopen` é compartilhado pelas diversas *threads*, por onde estas acessam o arquivo. Neste caso existe um único buffer de dados da biblioteca e ponteiros para as posições do arquivo, o qual é compartilhado entre todas as *threads* da aplicação. Este compartilhamento dos ponteiros pode levar a problemas, como movimentar ponteiro do arquivo (`fseek`) e realizar uma escrita (`fwrite`, `fprintf`) na nova posição logo em seguida. Esta situação corre o risco de outra *thread* movimentar o ponteiro do arquivo em um tempo entre a movimentação e a escrita, fazendo com que a primeira *thread* escreva seus dados em uma posicão equivocada do arquivo (LEWIS; BERG, 1996). Para estes casos, a biblioteca `stdio` disponibiliza as funções `flockfile` e `funlockfile`, de forma que quando uma determinada *thread* obtenha o *stream*, outra não possa usá-lo.

- *Stream* privado: Cada *thread* é responsável pela aberura do arquivo. Nesta situação os ponteiros para o arquivo e os buffers não são compartilhados pelas mesmas, pois cada uma possui o seu próprio *stream* retornado pela função `fopen`. Isto acarreta em problemas de atualização e coerência entre os buffers das estruturas, tornando necessário o uso da chamada `fflush` após cada operação de

escrita. Porém, há situações onde nem sempre este problema pode ser contornado, pois as leituras possuem um *read-ahead* para o preenchimento do buffer, que pode ficar desatualizado caso outras *threads* escrevam em tal região do arquivo já localmente armazenada em buffers.

### 2.6.4 Ffio

O *Flexible File I/O* (Ffio) é um sistema proprietário da Silicon Graphics (SGI), que possibilita que seja criada uma camada intermediária, interceptando o tráfego de dados das aplicações e os arquivos armazenados no disco rígido. Tal camada pode ser configurada pelo usuário, como diversos tipos de buffers e *caches*, bem como seus tamanhos e propriedades (REILEY, 1999). Este possibilita que aplicações com um grande tráfego de dados para o disco obtenham um melhor uso do hardware disponivel.

Proporcionando um mecanismo para o aumento do desempenho de I/O de aplicações, o Ffio utiliza de variáveis de ambiente para sua configuração, descartando o conhecimento sobre o código da aplicação. Porém, é aconselhável o conhecimento sobre como a aplicação trabalha, o que pode ajudar a interpretar e otimizar os resultados do Ffio (SGI, 2008).

Dentre as vantagens do Ffio , é possível citar:

- dispensa alteração do código fonte da aplicação;
- os parâmetros de configuração são passados por meio de variáveis de ambiente;
- permite que sejam criados buffers também para aplicações distribuídas, como MPI.

e dos seus pontos fracos, cabe ressaltar:

- é um sistema proprietário, fornecido apenas pelo pacote Propack da SGI;
- disponível apenas para sistemas IRIX e Linux;
- fornecido apenas para máquinas da SGI.

## 2.7 Concorrência a dados e sincronização de tarefas

Em ambientes *multithread* com memória compartilhada, podem acontecer situações em que diversas *threads* acessem os mesmos dados, podendo levá-los a um estado

inválido. Desta forma, torna-se necessário um mecanismo de proteção aos dados compartilhados, utilizando controle de acessos. Uma forma de gerenciar esse controle é com a utilização de semáforos, como feito neste trabalho.

### 2.7.1 Semáforos

Segundo (DOWNEY, 2005), um semáforo é uma variável do tipo inteira que respeita as seguintes propriedades:

- Quando um semáforo é criado, seu valor pode ser inicializado com qualquer valor inteiro. Porém, as únicas operações permitidas neste são o incremento ( adição de 1 ) e o decremento ( subtração de 1 ), não sendo possível ler o valor da variável;
- No momento em que uma *thread* decrementar o semáforo e seu valor for negativo, a *thread* bloqueia a si própria e continua apenas se outra *thread* incrementar o semáforo;
- Se no instante em que uma determinada *thread* incrementar o semáforo existirem outras *threads* bloqueadas, uma das *threads* será desbloqueada.

Dizer que uma *thread* bloqueia a si própria significa dizer que esta informa ao escalonador da CPU que não pode continuar sua execução. Este mecanismo permite que não sejam desperdiçados ciclos de CPU para a execução de possíveis leituras em espera ocupada[4], verificando a possibilidade da *thread* continuar com seu fluxo de execução normal (SILBERSCHATZ; GALVIN; GAGNE, 2004).

[4]Testes repetidos para verificação de uma condição que impede o progresso de um processo, e que só pode ser alterada por outro processo

# 3 Estratégia para otimização no acesso a disco

Para que a eficiência no acesso aos dispositivos de I/O possa ser melhorada, diversos princípios podem ser empregados (SILBERSCHATZ; GALVIN; GAGNE, 2004). Entre eles:

- Utilizar um menor número de cópias dos dados em memória enquanto são transmitidos entre o dispositivo e as aplicações;
- Diminuir a frequencia de interrupções utilizando grandes transferências de dados entre o sistema e o dispositivo;
- Balancear o desempenho da CPU, do subsistema de memória, dos barramentos do sistema e do dispositivo, pois uma sobrecarga em qualquer um destes poderá levar a ociosidade nos demais.

Pensando em quais são os pontos onde deveriam ser implementadas as otimizações surgem os seguintes pontos: na aplicação, no *kernel* do SO ou diretamente no dispositivo, em *hardware*.

A implantação de otimizações na aplicação possui a vantagem do menor custo de desenvolvimento e da flexibilidade, pois possíveis *bugs* na aplicação não levariam a quedas do sistema e durante o desenvolvimento não seria necessária a reinicialização ou recarga dos *drivers* do dispositivo após cada uso do novo código.

Desenvolver um algoritmo no nível do *kernel* é uma tarefa nem sempre simples de ser realizada. Códigos nesta região devem ser cuidadosamente depurados, pois o SO é um sistema grande e complexo, e muito cuidado deve ser tomado para evitar corrupção de dados e quedas do sistema.

Otimizações em *hardware* tem a possibilidade de obter o melhor desempenho, porém o desenvolvimento de otimizações neste nivel é de alto custo e baixa portabilidade. Desta forma, estas otimizações estariam muito restritas apenas à configuração de *hardware* para a qual foi desenvolvida.

As arquiteturas com multiprocessadores disponibilizadas no mercado dispõem de apenas um meio de acesso ao sistema de armazenamento, impondo um limite físico para o tráfego da informação. A solução proposta busca explorar este meio de acesso ao seu limite máximo de tráfego, utilizando os artefatos disponibilizados pela arquitetura. Este trabalho consiste na especificação e implementação da biblioteca FPA, que desenvolve uma técnica de armazenamento temporário de dados e organização dos acessos ao disco rígido. Sendo assim, a aplicação desenvolvida é implementada no nível da aplicação devido ao seu baixo custo e maior portabilidade, permitindo que a aplicação possua uma maior flexibilidade a diversos ambientes computacionais.

## 3.1 Estratégia proposta

Além de proporcionar diversas linhas de processamento simultâneo, uma das principais características das arquiteturas com multiprocessadores é possibilitar o uso de memória compartilhada. Aproveitando este recurso, FPA mantém os dados dos arquivos em uma região de memória compartilhada, e diversas *threads* tem acesso às informações sem que seja necessária uma chamada ao disco a cada momento de uso do arquivo.

O armazenamento temporário de dados possibilita a sobreposição entre o processamento da aplicação e a gravação de dados ao disco, onde a aplicação não precisa aguardar pelo envio dos dados e assim continuar o seu processamento. Posteriormente, quando toda a região temporária estiver cheia, os dados poderão ser enviados ao sistema de armazenamento, nas situações de escrita. Para casos onde existe leitura de dados, toda a região temporária pode ser preenchida de uma única vez. A prática de utilizar operações de entrada e saída com quantidades de dados maiores em menores quantidades ao invés de diversas operações com quantidades pequenas de dados implica em uma sobrecarga menor ao sistema operacional, enquanto também faz um melhor aproveitamento das operações ao disco.

Outra vantagem da utilização de regiões temporárias de dados em memória compartilhada é que uma outra linha de execução pode assumir o gerenciamento dos dados e definir o tempo em que estes devem ser atualizados com o disco, sem que a aplicação principal interfira neste gerenciamento. A vantagem desta situação é que pode ser feita uma organização dos acessos ao disco rígido, permitindo que seja realizado um melhor aproveitamento dos movimentos do cabeçote de leitura e escrita do disco. Para o caso ótimo, isto significa obter a menor movimentação possível do cabeçote durante o acesso às informações.

Aplicando essas vantagens em ambientes com diversas linhas de processamento, torna-se conveniente que existam diversas regiões compartilhadas. Desta forma, *threads* diferentes podem acessar regiões de memória compartilhadas diferentes, e caso sejam acessadas regiões que já estejam em memória previamente, estas poderão utilizar a mesma área de memória.

Como existem diversas áreas compartilhadas em memória e o gerenciamento destas é centralizado por uma única *thread*, o envio das informações ao disco é realizado de forma que o menor tempo de gravação seja obtido. Sendo assim, a *thread* gerenciadora envia as informações ao disco de forma ordenada, para que uma menor movimentação do cabeçote no disco seja realizada.

Os arquivos são mapeados em regiões distintas para buffers de memória, sendo que em cada buffer é mapeada uma determinada região contígua do arquivo. A figura 3.1 representa uma situação em que cada *thread* acessa um buffer do arquivo.

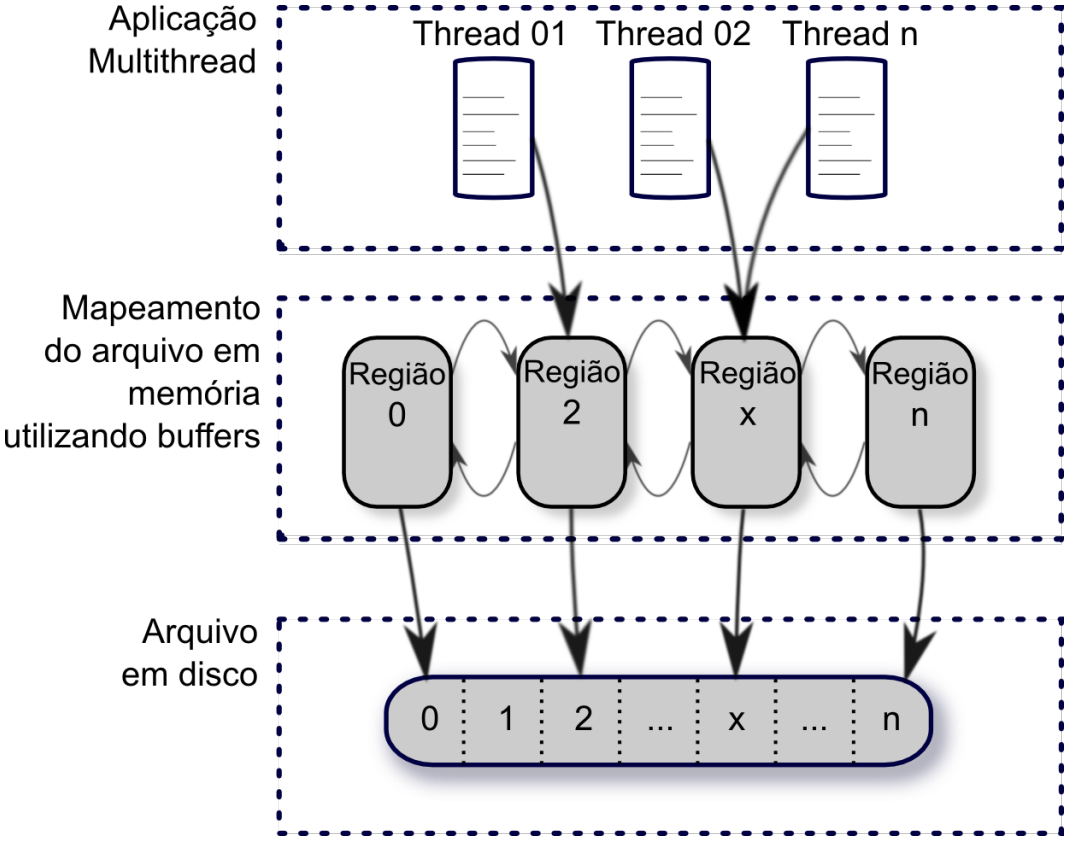



**Figura 3.1:** Acesso e mapeamento de arquivos em memória

Para o caso em que mais de uma *thread* acessa uma mesma região do arquivo simultaneamente, tal região será compartilhada entre as *threads*. Outra vantagem da utilização de buffers para acesso a arquivos, é que estes podem estar organizados em forma de listas. Isto permite que se uma determinada região não estiver mais sendo acessada e um limite arbitrário de uso de memória for alcançado, este buffer poderá

ser excluído da lista e liberado da memória.

O uso de buffers de dados em memória possui a vantagem da velocidade no acesso à informação, pois o acesso à memória é diversas vezes mais rápido que ao disco. Além disso, possibilita que no momento em que um determinado buffer for liberado, seja executada apenas uma chamada de sistema para enviar todos os dados armazenados no buffer ao disco.

Um cuidado a ser tomado é evitar que o disco seja acessado por mais de uma *thread* durante uma mesma requisição de escrita. Testes realizados por (BENSON; LONG; PACHECO, 2003) indicam perda de desempenho quando diversas solicitações são feitas, devido à tentativa do disco de atender todas as requisições, efetuando uma maior quantidade de movimentação do cabeçote de leitura e gravação.

De forma a evitar que existam múltiplos acessos simultâneos ao disco, a estratégia proposta trata de utilizar regiões temporárias em memória para armazenamento de dados e posteriormente envia esses dados ao disco. O envio é realizado de forma a possibilitar uma melhor organização no acesso ao disco e uma menor sobrecarga do sistema operacional.

## 3.2 Implementação

A estratégia FPA busca a solução para o problema da concorrência ao disco sem alterações na arquitetura ou *hardware* adicional. A estratégia é implementada via software, possibilitando que esta possa ser utilizada em quaisquer aplicações construídas utilizando a linguagem de programação C e suporte a Pthreads.

A implementação aplica as estratégias de otimização de acesso a disco, disponibilizando ao usuário uma biblioteca com uma interface semelhante às rotinas tradicionais de acesso a arquivos. A biblioteca fornece rotinas de acesso a arquivos como abertura e fechamento de arquivos, escrita, leitura, movimentação do ponteiro de posição no arquivo e a atualização dos dados no arquivo em disco. Estas funções possibilitam que a otimização de acesso a arquivos seja transparente ao usuário.

Os buffers em memória são implementados em forma de lista, onde cada elemento da lista contém a região com os dados do arquivo, qual região do arquivo aqueles dados representam e quais são as *threads* que utilizam determinado buffer. Os buffers não precisam necessariamente mapear regiões vizinhas do arquivo, bastando mapear apenas aquelas regiões realmente utilizadas. A figura 3.1 ilustra este cenário, onde o primeiro buffer possui os dados da região 0 e o segundo buffer possui os dados da

região 2.

Para evitar que as *threads* escrevam dados no disco diretamente e assim agrave a concorrência ao disco, é criada uma *thread* única para acesso ao disco, como ilustrado na figura 3.2. Todas as operações de escrita ao disco são efetuadas por esta *thread*, que utilizando um controle com semáforos, gerencia múltiplas requisições de acesso ao disco e evita o acesso concorrente.

Objetivando um melhor desempenho durante todo o processo de leitura e escrita, não são feitas cópias dos dados entre os buffers para outras regiões de memória. Todas as *threads* acessam os mesmos endereços de memória. As únicas situações onde são realizadas cópias de informações são aquelas envolvendo os parâmetros de entrada das rotinas ou valores de retorno ao usuário.

### 3.2.1 Mapeamento do arquivo e alocação de buffers

O mapeamento do arquivo em memória pode ser realizado de diferentes formas, como com `mmap` ou `calloc`. Neste trabalho são utilizadas ambas as técnicas, implementadas em diferentes versões da FPA. O objetivo aqui é comparar o desempenho da biblioteca com as diferentes formas, e uma análise comparativa sobre as diferentes técnicas pode ser vista no capítulo 4.

A biblioteca FPA possui três versões implementadas, onde uma destas utiliza `calloc` e as demais utilizam `mmap`. Detalhes das versões implementadas estão descritos a seguir.

#### Versões utilizando Mmap

Quando um novo buffer é criado, sua região de dados é mapeada do arquivo utilizando a chamada de sistema `mmap`. Esta permite que seja mapeada apenas uma região do arquivo, e não apenas o arquivo por completo. Caso exista a partir do usuário uma ordem de enviar os dados ao disco, é executada sobre a região de dados a chamada de sistema `msync`, onde esta trata de enviar as alterações feitas no buffer para o disco. Esta mesma situação ocorre no momento em que um buffer precisa ser liberado e as informações enviadas ao disco.

À medida que um determinado limite de memória é ultrapassado, FPA trata de liberar os buffers não utilizados no momento e enviar seus dados armazenados ao disco. Desta forma, um buffer é liberado por vez, até que a quantidade de memória utilizada pela FPA seja inferior ao limite definido.

Utilizando o `mmap`, existe também uma outra versão na FPA. Esta versão consiste em que durante a liberação dos buffers, sejam liberados todos aqueles buffers que não estão sendo utilizados no momento.

**Versões utilizando alocação de memória**

Ao alocar um novo buffer utilizando a rotina `calloc`, o envio dos dados ao disco deve ser explicitamente realizado pelo usuário ou através de um gerenciamento para tal implementado pela biblioteca. Neste, o sistema operacional não pode intervir e enviar as informações, como acontece com a chamada `mmap`. Sem tal intervenção, existe um maior controle do momento do envio das informações, evitando indesejáveis movimentações do cabeçote de leitura do disco.

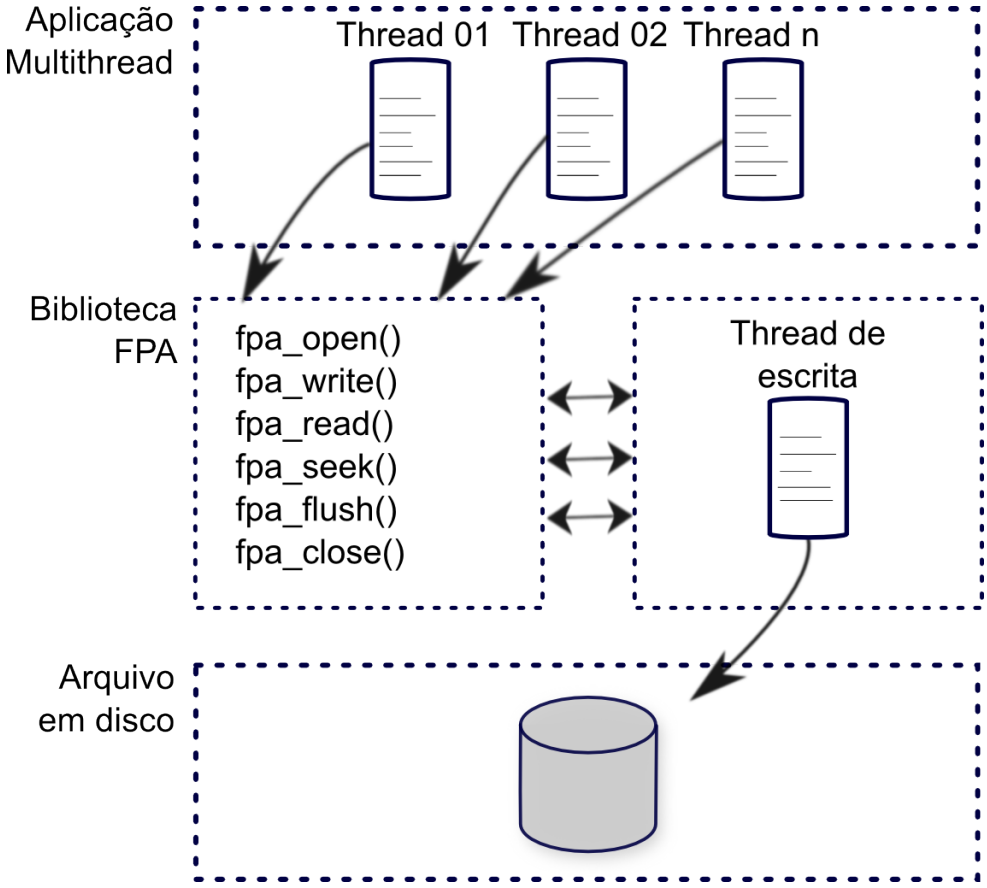



**Figura 3.2:** Interfaceamento da FPA com diversas *threads* de aplicações

### 3.2.2 Rotinas e estruturas

A implementação consiste basicamente em vários conjuntos de listas, e na manipulação das mesmas. As principais são a lista com os arquivos abertos e as listas com os buffers, onde estas últimas contém os dados a serem gravados em disco. Cada elemento da lista de arquivos abertos contém sua própria lista de buffers, referentes ao

mapeamento do arquivo correspondente.

As diversas *threads* acessam o arquivo através de uma estrutura definida como `FPA_FILE` , que é fornecida como retorno da função `fpa_open`. Esta estrutura contém informações sobre qual é o arquivo que está sendo utilizado, qual é a posição onde os dados deverão ser lidos ou escritos, e consequentemente qual buffer contém ou conterá as informações. Apenas por comodidade de nomenclatura, o ponteiro de leitura e escrita é chamado ponteiro de acesso. Em momentos em que apenas o ponteiro de leitura ou escrita especificamente estiver sendo referenciado, este será explicitamente citado.

A biblioteca FPA possibilita que algumas configurações básicas possam ser feitas pelo usuário, como definir o tamanho dos buffers, informar em que momento os dados devem ser enviados ao disco ou então qual deve ser a quantidade máxima de memória a ser utilizada pela biblioteca. Todas estas configurações são realizadas a partir da rotina `fpa_start`, a qual também é responsável pela inicialização da *thread* de acesso a disco e dos semáforos utilizados. O seu funcionamento pode ser visto na figura 3.3.

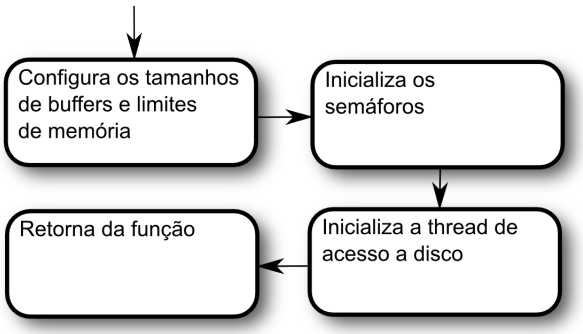



**Figura 3.3:** Fluxograma da rotina fpa_start

A rotina `fpa_open`, além de abrir o arquivo, trata de alocar um elemento na lista de arquivos abertos. O fluxograma do funcionamento da rotina pode ser visto na figura 3.4. O retorno desta rotina é um ponteiro para uma estrutura do tipo `FPA_FILE` .

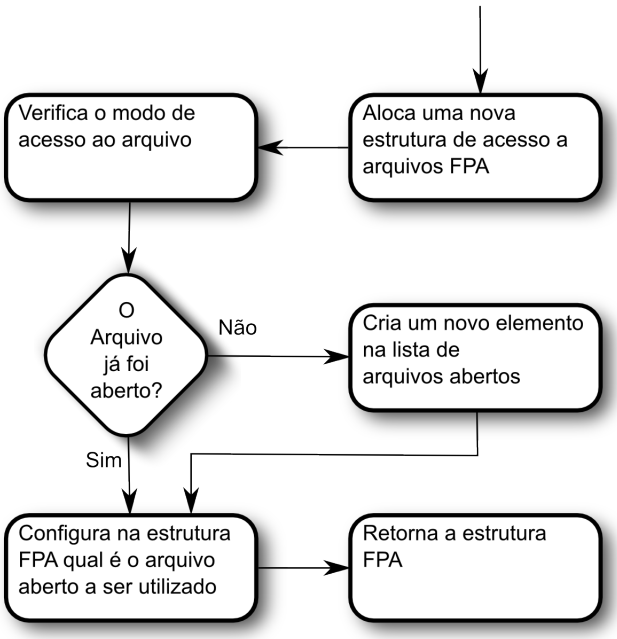



**Figura 3.4:** Fluxograma da rotina fpa_open

Cada elemento da lista de buffers contém uma variável do tipo `int`, informando quantas estruturas `FPA_FILE` contém um ponteiro para aquele buffer (consequentemente, quantas *threads* estão utilizando tal buffer). Através deste contador é possível descobrir se o buffer poderá ser liberado, e suas informações serem enviadas ao arquivo.

A rotina `fpa_seek` trata de movimentar o ponteiro de acesso contido na estrutura `FPA_FILE` . É também responsável pelas chamadas de alocação de novos buffers e também pelo envio de sinais informando que dados devem ser enviados ao disco. Seu funcionamento pode ser acompanhado na figura 3.5.

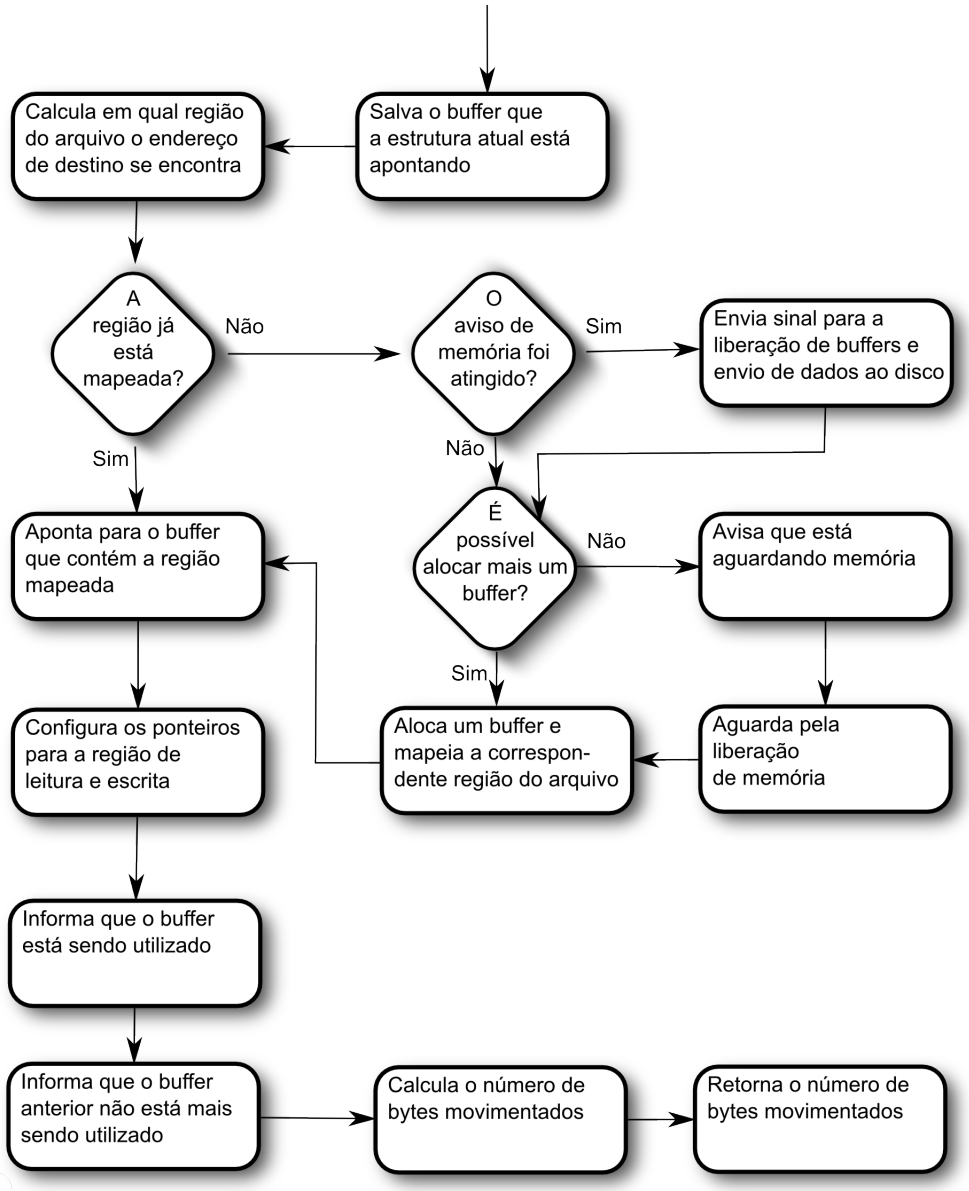



**Figura 3.5:** Fluxograma da rotina fpa_seek

Na FPA, a rotina `fpa_write` consiste em efetuar uma cópia dos dados do usuário para a região de memória do buffer. Levando em conta a possibilidade do usuário escrever mais informações do que o espaço disponível no buffer, a rotina dinamicamente trata de mapear o buffer seguinte e efetuar a cópia das informações para o respectivo espaço. Este mapeamento ocorre chamando recursivamente a própria `fpa_write`, utilizando a função `fpa_seek` para a movimentação dos ponteiros de acesso. O uso da função `fpa_seek` permite que seja efetuado o envio de sinais para que dados sejam enviados ao disco, quando necessário. Uma abstração da sua implementação pode ser vista na figura 3.6.

O valor de retorno da rotina `fpa_write` é o número de bytes escritos nos buffers. A título de implementação, este valor corresponde à quantidade de bytes escritos no

buffer apontado mais a quantidade escrita nas chamadas recursivas.

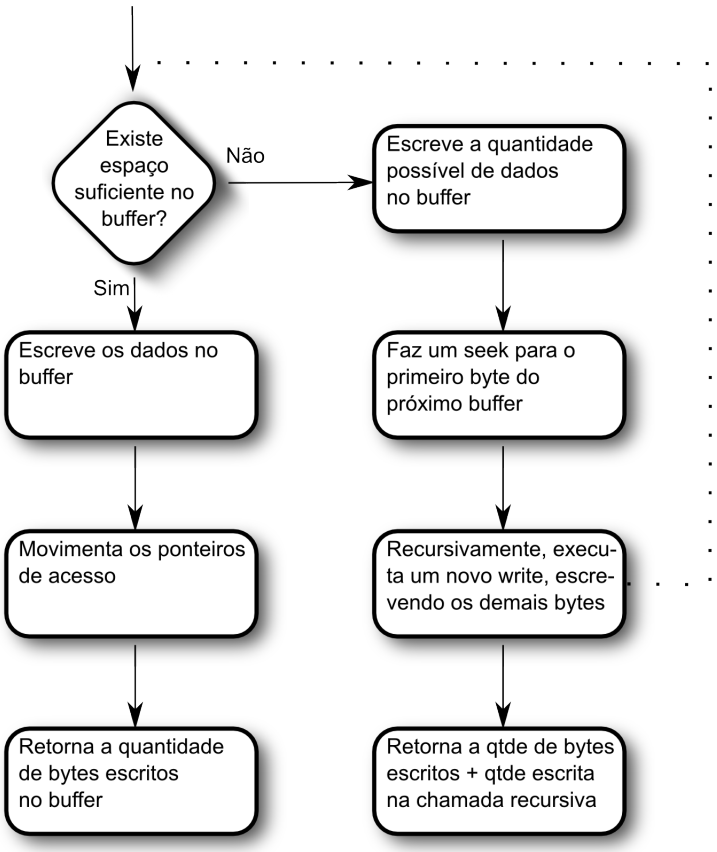



**Figura 3.6:** Fluxograma da rotina fpa_write

Muito semelhante à rotina `fpa_write`, a `fpa_read` consiste em copiar informações na memória. Naturalmente, aqui acontece o inverso, pois a partir dos buffers as informações são passadas para a região de memória informada pelo usuário. O funcionamento desta rotina pode ser visto na figura 3.7.

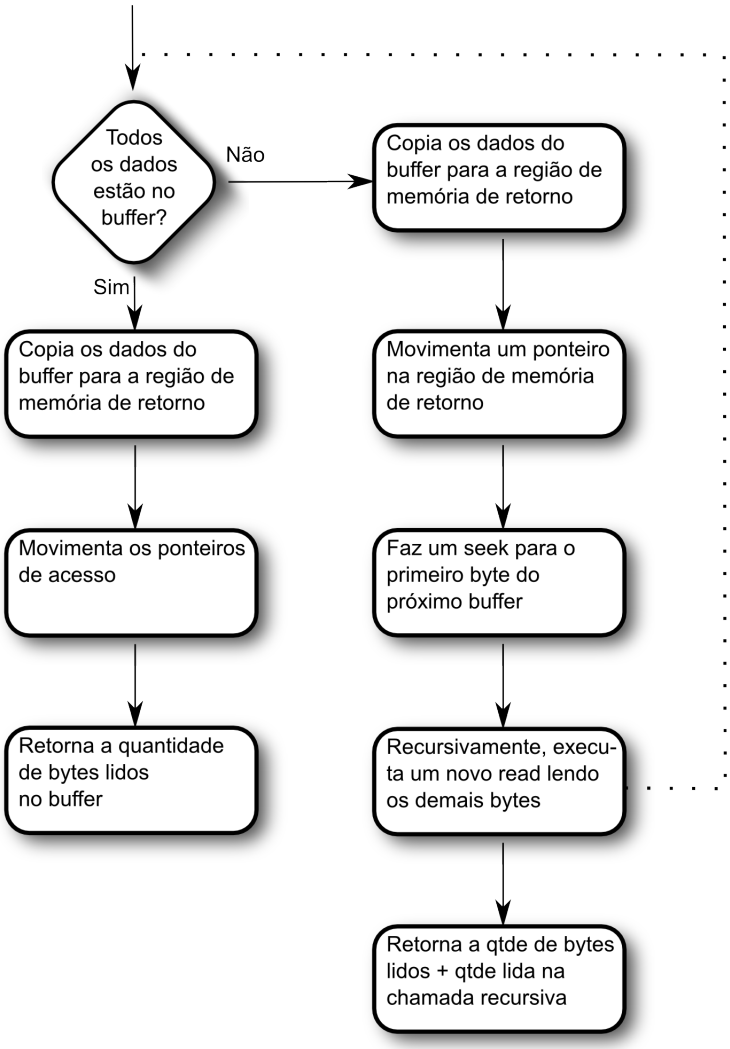



**Figura 3.7:** Fluxograma da rotina fpa_read

As únicas formas de garantir que os dados estejam realmente no disco é fazendo todas as *threads* fecharem o arquivo ou então com a utilização da rotina `fpa_flush`. Esta rotina não trata da liberação de memória, consistindo basicamente em percorrer a lista de buffers e enviar as informações para o disco.

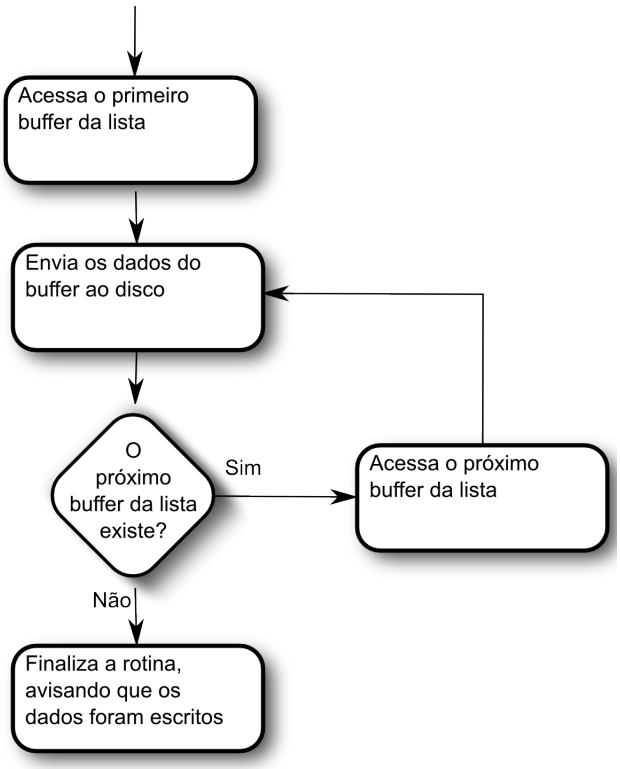



**Figura 3.8:** Fluxograma da rotina fpa_flush

A rotina `fpa_close` deve ser executada por todas as *threads* para que todas informações residentes nos buffers sejam enviadas ao disco. O envio dos dados ao disco é realizado a partir da informação de quantas *threads* estão acessando tal arquivo, que é feito através de um contador. Se para o momento em que uma determinada *thread* fechar o arquivo e o contador informar que esta é a única a utilizar o arquivo, esta *thread* deve ser responsável por enviar os dados residentes nos buffers para o disco e também liberar as respectivas regiões de memória. Na figura 3.9 é possível acompanhar o funcionamento da rotina `fpa_close`.

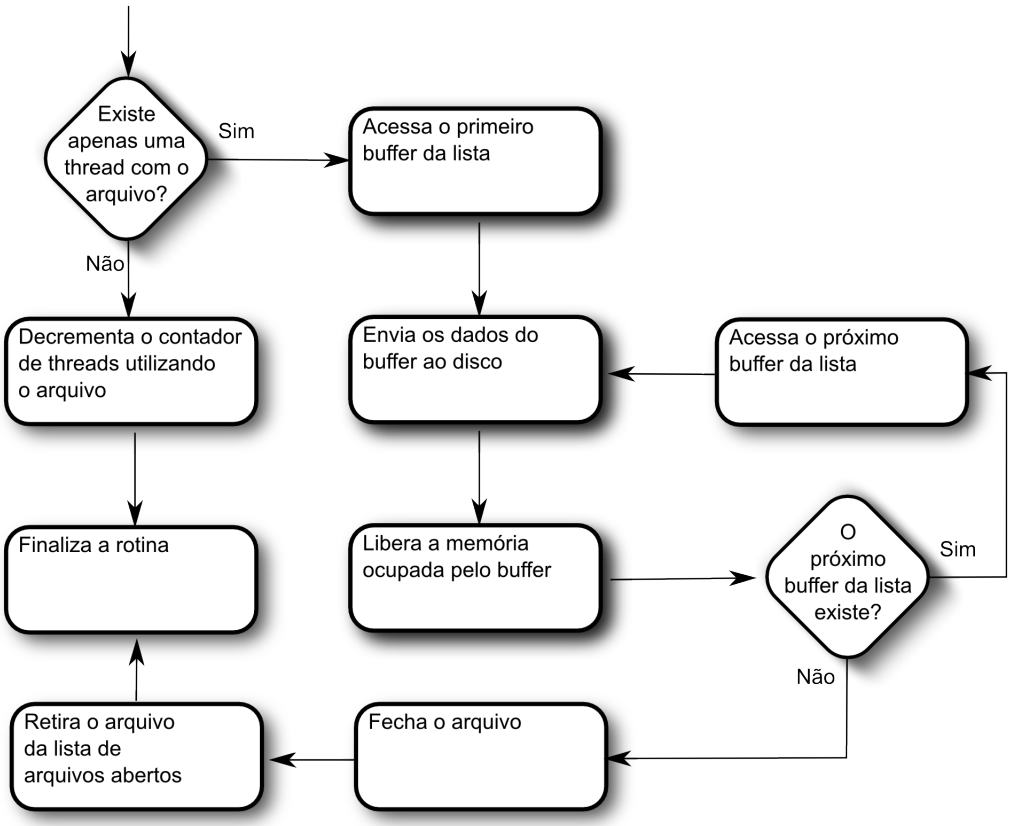



**Figura 3.9:** Fluxograma da rotina fpa_close

A FPA também implementa a rotina `fpa_eof`, onde esta possibilita que seja conhecido se um determinado arquivo chegou ao seu final. Seu funcionamento acontece verificando se o ponteiro para acesso ao arquivo encontra-se no último byte modificado do último buffer. Como as *threads* podem apontar para buffers diferentes, o resultado desta rotina pode não ser o mesmo para todas as *threads* que a executarem.

Para que sejam obtidas informações de posicionamento no arquivo foi desenvolvida a rotina `fpa_tell`, onde consiste em calcular a posição no arquivo onde os dados serão lidos ou escritos, e retornar a respectiva posição em bytes.

### 3.2.3 Compatilidade entre funções

As rotinas fornecidas pela FPA `fpa_fread`, `fpa_write` e `fpa_seek` possuem a ordem de parâmetros equivalentes aos das chamadas de sistema `read`, `write` e `lseek`. De forma a compatibilizar o uso das funções da FPA com rotinas de leitura, escrita e *seek* da biblioteca `stdio`, são também foram implementadas as rotinas `fpa_fseek`, `fpa_fread` e `fpa_fwrite`. O objetivo destas novas funções é manter a ordem dos parâmetros semelhantes às funções `fseek`, `fread` e `fwrite`, respectivamente. Obviamente, o parâmetro referente ao descritor do arquivo nestas funções continua sendo do

tipo `FPA_FILE`, ao invés de um *stream* ou inteiro. Internamente, as mesmas estruturas e funções são utilizadas, pois trata-se apenas de uma diferente interface de acesso às mesmas funções.

### 3.2.4 Seções críticas e sincronização

O gerenciamento de listas em um ambiente *multithread* possui diversas informações e operações que devem ser cuidadosamente controladas, como qual é o primeiro elemento da lista e a inserção de um novo elemento, por exemplo. Levando isto em consideração para a implementação da biblioteca FPA, qualquer operação envolvendo alguma das listas utilizadas deve ser considerada como uma seção crítica. O controle das listas é feito utilizando semáforos, assim impossibilitando que mais de uma *thread* opere simultaneamente sobre as listas.

Além das seções críticas, os semáforos também são utilizados para sincronização de operações. Na FPA, parte da estratégia de solução é impedir o acesso de mais de uma *thread* ao disco durante o mesmo intervalo de tempo.

Os sinais, referenciados nas figuras da seção 3.2.2 também são feitos com a utilização de semáforos. Estes significam a liberação do semáforo em questão, como o aviso para a liberação de buffers, por exemplo. Neste caso, a liberação do semáforo permitirá que a *thread* responsável pelo envio das informações ao disco execute tais operações.

### 3.2.5 Limites de memória e liberação de buffers

Os limites de memória são indispensáveis para a implementação, pois através destes são determinados os momentos em que as informações devem ser enviadas ao disco. Um contador de buffers alocados indica quanto de memória está sendo utilizada, e o valor neste contador é utilizado para verificar se foi excedido algum dos limites de memória configurados no sistema.

Na implementação da FPA, existem dois limites de memória: o limite de aviso de memória (*LA*) e o limite máximo (*LM*). O aviso de memória é um limite que indica que caso a quantidade de memória utilizada pela biblioteca seja ultrapassada, buffers devem ser liberados. Porém, este limite ainda permite que seja alocada mais memória para outros buffers conforme necessário. Já o limite máximo de memória indica que não devem existir mais alocações de memória.

O aviso de memória sempre deve possuir um valor inferior ao máximo de memória.

Isto deve-se a que o aviso de memória é responsável pelo sinal de envio de dados ao disco, o que não acontece com o limite máximo. Desta forma, sempre que um limite máximo de memória for atingido, antes terá sido atingido um limite de aviso de memória.

A configuração destes limites e tamanho de buffer pode ser feita através da rotina `fpa_start` . Caso nenhum valor seja informado, os valores *default* são assumidos.

O recebimento dos sinais de envio de informações é feito pela *thread* responsável pelo acesso ao disco. A figura 3.10 apresenta tal mecanismo de tratamento dos sinais, bem como o envio dos dados ao disco.

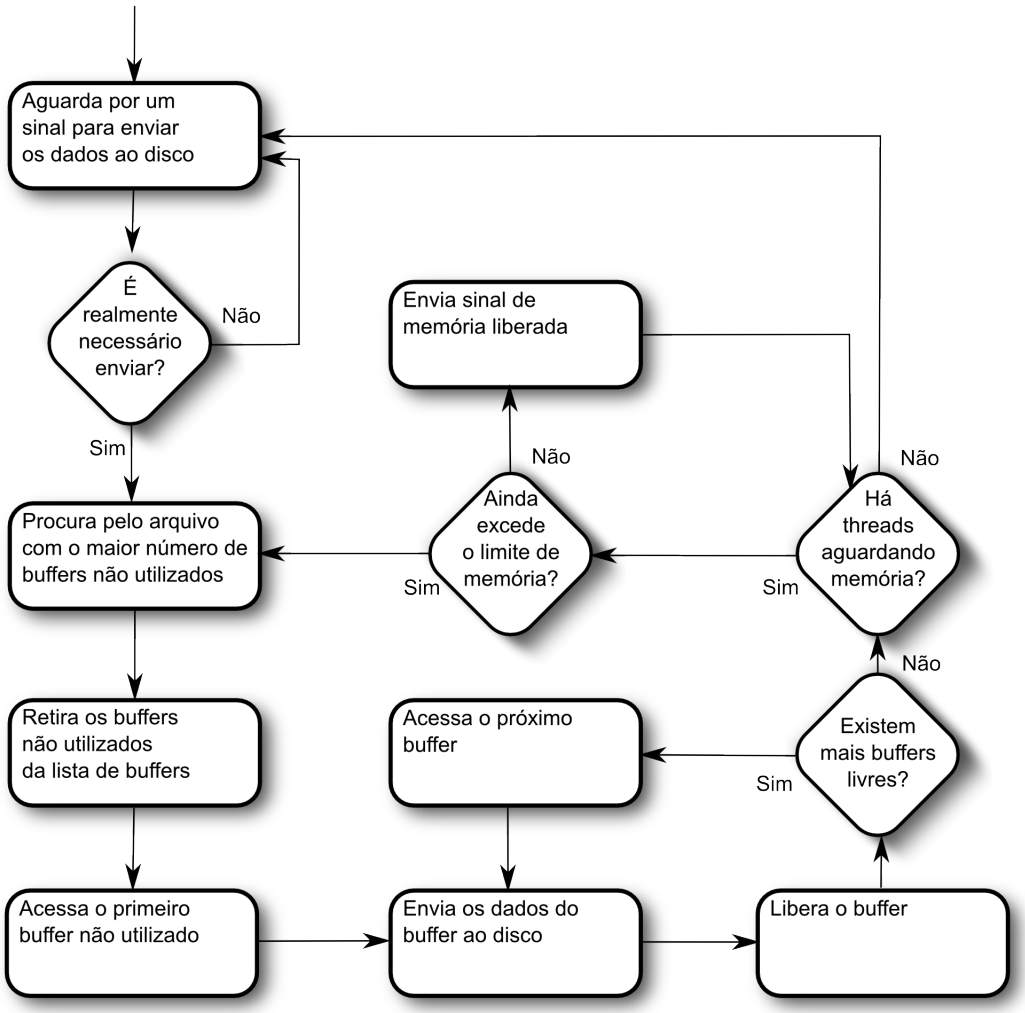



**Figura 3.10:** Fluxograma da *thread* de escrita de dados em disco

Durante as operações de leitura o acesso ao disco é realizado antes do buffer ser utilizado, e nestas situações a *thread* que gerencia o acesso ao disco não pode auxiliar, afinal esta age de acordo com a quantidade de memória alocada e então envia os dados ao disco.

Durante as operações de leitura cada *thread* é responsável pela busca dos seus

dados no disco e o preenchimento dos respectivos buffers. Para que não exista múltiplas chamadas de leitura simultaneamente ao disco a leitura é serializada, permitindo a leitura de 1 buffer por vez.

### 3.2.6 Organização da biblioteca

De forma a organizar melhor a estrutura da implementação, a biblioteca FPA é composta em duas partes, como pode ser conferido na figura 3.11

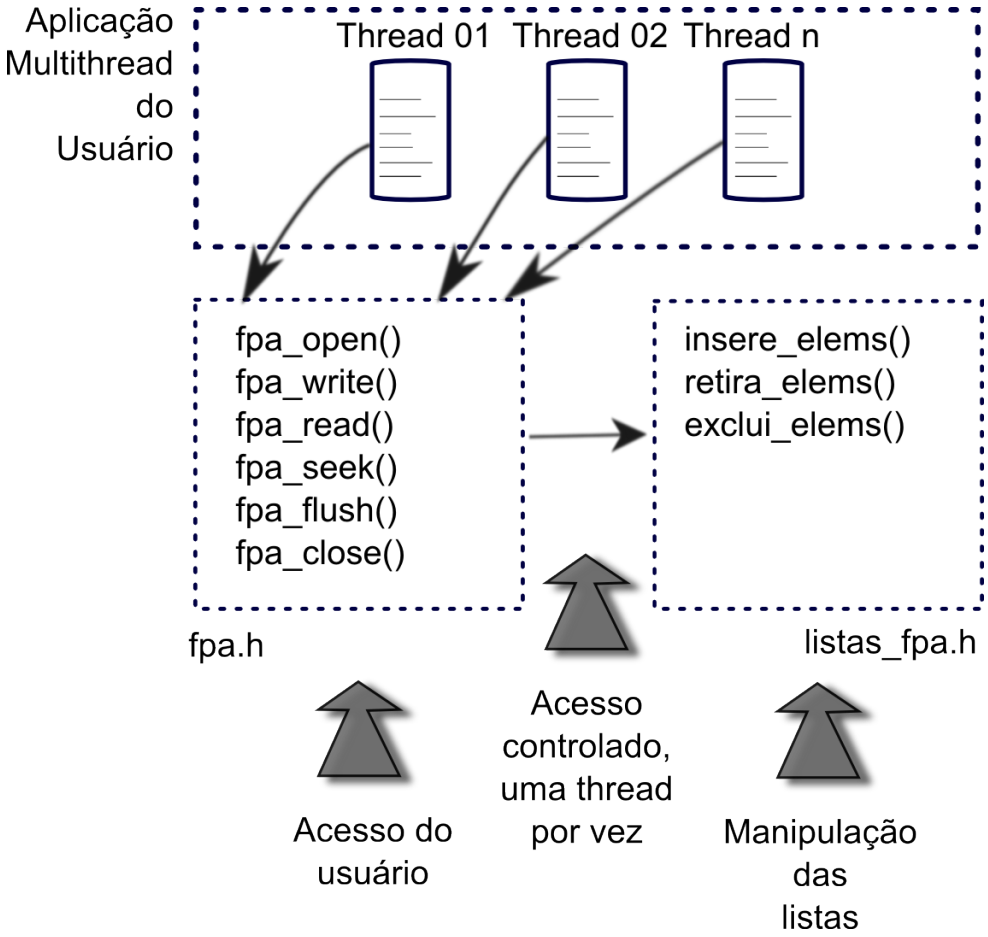



**Figura 3.11:** Organização da biblioteca FPA

O interfaceamento com o usuário é realizado através da biblioteca `fpa`, com o *header* `fpa.h`. Nesta, estão contidas as declarações das rotinas e estruturas de acesso a disco FPA. Já a codificação da biblioteca em si, está contida no arquivo `fpa.c`, com os devidos gerenciamentos de buffers e a *thread* de escrita a disco.

Outra biblioteca também é desenvolvida, a `listas_fpa`. Nesta são desenvolvidas todas as operações que manipulam as listas de arquivos abertos e buffers, necessários pelas rotinas implementadas em `fpa.c`. Como as alterações em listas são seções críticas, todas as chamadas de rotinas de `listas_fpa` devem ser protegidas por semáforos. Esta organização proporciona um melhor controle sobre os semáforos utilizados, dado

que todos serão parte apenas de `fpa.c`.

### 3.2.7 Documentação

Uma questão importante, seja pelo ponto de vista do usuário ou do desenvolvedor, é a documentação. A documentação deve conter especificações e informações sobre os parâmetros de entrada e o funcionamento da biblioteca, para que o máximo proveito da biblioteca seja alcançado. A documentação sobre a implementação realizada pode ser conferida junto ao apendice A. Como o manual é um documento à parte ele possui sua própria numeração de capítulos e páginas, sendo independente do restante deste trabalho.

# 4 Resultados e análises

Os resultados atuais obtidos a partir da implementação proposta neste trabalho são apresentados e discutidos neste capítulo. As seções seguintes apresentam o ambiente onde os testes foram realizados, seu formato e os resultados obtidos.

A seção 4.2 é referente a testes elaborados neste trabalho, onde estes possuem o intuito de analisar o comportamento das aplicações desenvolvidas sob diversas condições conhecidas. As seções 4.3 e 4.4 correspondem à implantação da FPA em aplicações já existentes, apresentando um breve estudo de caso sobre cada situação e seus resultados.

## 4.1 Ambiente de execução

Os testes da biblioteca FPA foram executados no *cluster Open Modeller* (OM), disponível no edifício do Centro de Computação Eletrônica (CCE) da Universidade de São Paulo (USP). O *cluster* OM é composto por 10 máquinas homogêneas, onde cada nó possui as configurações definidas na tabela 4.1. Como os testes envolvem paralelismo em nível de *threads* e não envolvem processamento distribuído, cada teste é executado isoladamente em um único nó do *cluster*.

**Tabela 4.1:** Configurações das máquinas do *cluster* OM

| Item | Valor |
|---|---|
| Processador | Dual Quadcore Intel (R) Xeon(R) |
| *Clock* da CPU | 2.0 Gigahertz |
| Arquitetura | 64 bits |
| Memória física | 8 Gigabytes |
| Memória virtual | 10 Gigabytes, mapeado a partir de arquivo |
| Sistema Operacional | SUSE Linux Enterprise Server 10 (x86_64) |
| Versão de Kernel | 2.6.16.21 – 0.8 |
| Tamanho da página de memória | 4 Kilobytes |
| Sistema de arquivos | XFS |
| Compilador | ICC - Intel(R) C/C++ Compiler |
| Versão do compilador | 9.120070215 |
| Ferramenta de Paralelismo | Openmp |
| Padrão de conexão do disco rígido | Serial ATA (Sata) |

Adicionalmente, foram realizados outros testes em uma máquina do laboratório de arquitetura de computadores e computação de alto desempenho (LAHPC[1]) da Escola Politénica da Usp (Epusp). O objetivo dos testes em outra máquina é comparar os resultados do uso da FPA sob diferentes configurações de *hardware* e verificar seu comportamento. As configurações da máquina podem ser vistas na tabela 4.2.

[1]Do inglês *Laboratory of Architecture and High Performance Computing*

**Tabela 4.2:** Configurações da máquina do LAHPC

| Item | Valor |
|---|---|
| Processador | Intel(R) Core(TM) i7 |
| *Clock* da CPU | 2.67 Gigahertz |
| Arquitetura | 64 bits |
| Memória física | 6 Gigabytes |
| Memória virtual | 2 Gigabytes |
| Sistema Operacional | openSUSE 11.1 (x86_64) |
| Versão de Kernel | 2.6.31.5-lcs-taskaff |
| Tamanho da página de memória | 4 Kilobytes |
| Sistema de arquivos | EXT3 |
| Compilador | Gnu *Compiler Collection* (GCC) |
| Versão do compilador | 4.3.2 |
| Ferramenta de Paralelismo | Openmp |
| Padrão de conexão do disco rígido | Serial ATA (Sata) |

Todos os testes resultados apresentados neste trabalho sem especificação explícita do ambiente onde foram executados, referem-se às execuções no *cluster* OM. Os gráficos que apresentam resultados obtidos através das execuções na máquina do LAHPC possuem indicações de que foram realizadas em tal máquina.

## 4.2 Testes elaborados

Os testes apresentados nesta seção tratam da execução das implementações desenvolvidas no trabalho sob circunstâncias definidas, e dado isso, espera-se um resultado compatível com tais circunstâncias. Além da análise do comportamento estes resultados propiciam, estes também servem como comprovação de que as aplicações realizam o resultado final esperado, uma vez que a entrada e saída é monitorada e a validade dos resultados é conferida.

### 4.2.1 Informações sobre os testes

Nesta seção é descrito como são realizados os testes, como é feita a medição dos tempos de execução, os parâmetros passados para as execuções e os pontos a serem observados nos resultados.

### Simulação de processamento

Na execução dos testes, um fator levado em consideração é o processamento envolvido nas aplicações. Desta forma, foram efetuados cálculos que realizam processamento, onde a quantidade calculada é passada como parâmetro para o teste. A figura 4.1 mostra tais cálculos, que foram executados para cada KB lido ou escrito pelas aplicações.

Neste trabalho, optou-se por efetuar cálculos ao invés da utilização de chamadas de espera, como `sleep` ou `usleep` . Estas funções podem causar ociosidade nas CPU's e a *thread* de escrita em disco pode aproveitar este tempo e ocupar a CPU para realizar suas execuções. Esta não é uma situação desejável, pois procura-se simular uma situação real de processamento e efetuar as análises baseando-se inclusive nesta concorrência pela CPU da *thread* de escrita da FPA e conhecer seu impacto no ganho de desempenho da aplicação. Inclusive, para evitar possíveis otimizações dos compiladores e garantir que todos os cálculos sejam realmente executados, aos compiladores das aplicações de testes é passada a opção `-O0`.

```
1 for( x =1; x <= processamento; x++ )
2 {
3         X = X*x/1023;
4 }
5 Y=X;
6
7 //Rotinas de leitura e/ou escrita
```

**Figura 4.1:** Cálculos executados nos testes

### Medição de tempo

As tabelas 4.3 e 4.4 apresentam o tempo necessário para que os cálculos indicados na figura 4.1 sejam executados. A quantidade de iterações no laço é definida pela variável `processamento`, onde este valor é alterado de acordo com a quantidade de processamento desejada na aplicação. Nas tabelas, a coluna "Tempo de processamento para 1 ciclo"é referente ao tempo gasto em uma iteração de cálculo, ou seja, o tempo tomado na execução do código apresentado na figura 4.1. Já a coluna "Tempo total de processamento"mostra o tempo utilizado em computar a execução do cálculo adicional para um arquivo de 1 GB com uma *thread*, sem as respectivas chamadas de rotinas de acesso a disco e utilizando na variável `processamento` o respectivo valor apresentado na coluna "Número de iterações". Ou seja, como o processamento é realizado a cada 1 KB de dados acessados, o cálculo será realizado $1024 * 1024$ vezes.

Os tempos de execução apresentados nos gráficos são obtidos através da chamada `gettimeofday`, cuja forma de execução e tomada de tempo é apresentada na figura 4.2.

```
1  struct timeval tv1, tv2;
2  double t1, t2;
3
4  gettimeofday(&tv1, NULL);
5
6  //Codigo da aplicacao, com o acesso ao disco
7
8  gettimeofday(&tv2, NULL);
9  t1 = (double)(tv1.tv_sec) + (double)(tv1.tv_usec)/ 1000000.00;
10 t2 = (double)(tv2.tv_sec) + (double)(tv2.tv_usec)/ 1000000.00;
11
12 printf("Tempo Total: %lf\n", (t2 - t1));
```

**Figura 4.2:** Medição de tempo com a rotina gettimeofday

**Tabela 4.3:** Tempo gasto para diversos valores em processamento, no *cluster* OM

| Número de iterações | Tempo de processamento para 1 ciclo completo - executado a cada 1 KB (segundos) | Tempo total de processamento - para arquivos de 1 GB (segundos) |
|---|---|---|
| 0 | 1,00E-06 | 0,00 |
| 10 | 2,00E-06 | 0,18 |
| 100 | 3,00E-06 | 2,04 |
| 500 | 1,00E-05 | 10,24 |
| 1000 | 2,00E-05 | 20,50 |
| 5000 | 9,80E-05 | 102,56 |
| 10000 | 2,14E-04 | 205,22 |
| 50000 | 9,79E-04 | 1027,33 |

**Tabela 4.4:** Tempo gasto para diversos valores em processamento, na máquina do LAHPC

| Número de iterações | Tempo de processamento para 1 ciclo completo - executado a cada 1 KB (segundos) | Tempo total de processamento - para arquivos de 1 GB (segundos) |
|---|---|---|
| 0 | 1,00E-06 | 0,01 |
| 10 | 2,00E-06 | 0,14 |
| 100 | 3,00E-06 | 1,51 |
| 500 | 1,40E-05 | 7,61 |
| 1000 | 2,80E-05 | 15,22 |
| 5000 | 1,32E-04 | 75,97 |
| 10000 | 2,64E-04 | 154,30 |
| 50000 | 1,37E-03 | 761,96 |

Convém destacar que, como os valores mostrados na coluna "Tempo total de processamento"das tabelas 4.3 e 4.4 são referentes a um teste sequencial, e em aplicações paralelas estes tempos são divididos pelo número de *threads* existentes na aplicação.

Para evitar que sejam realizadas medições errôneas no tempo de execução dos testes devido à interferência do SO, antes de cada execução das aplicações são executados a chamada de sistema `sync` e o comando `sysctl -w vm.drop_caches=3`. A chamada de sistema envia todos os dados armazenados em buffer ao disco, enquanto o segundo comando invalida os dados armazenados em *cache* e com isso faz que todas as operações que envolvam leitura de dados em arquivos busquem seus dados no disco.

### Metodologia de escolha do tempo de execução

A escolha do tempo de execução é realizada de tal forma que o valor final represente uma situação comum de execução da aplicação em questão. Para tal, cada teste é executado diversas vezes e uma metodologia de escolha é aplicada. Tal metodologia é apresentada a seguir.

a. Cada teste é executado 6 vezes, formando um conjunto de amostras inicial;

b. Temporariamente, são calculados a média aritmética $ma_t$ e o desvio padrão $dp_t$ das amostras iniciais;

c. Descarta-se os valores discrepantes das amostras, formando um novo conjunto de amostras com os valores que pertencem ao intervalo $[ma_t - dp_t, ma_t + dp_t]$;

d. Por último, são calculados a média aritmética *ma* e o desvio padrão $dp$ da nova amostra;

e. Toma-se *ma* como tempo utilizado no teste;

f. Toma-se $dp$ como o desvio padrão dos tempos do conjunto de teste.

Para cada conjunto de testes executado são apresentados os desvios padrões máximo ($dp_{max}$) e mínimo ($dp_{min}$), onde estes correspondem ao maior e menor valor dos desvios padrões de cada conjunto. Estes valores são mostrados no canto inferior esquerdo de cada gráfico resultado de tais execuções.

### Gráficos e rótulos de dados

Os testes realizados envolvem diversos aspectos a serem observados graficamente. Entre eles, estão a quantidade de *threads* utilizadas, a quantidade de processamento simulada e o tamanho do arquivo considerado.

As aplicações testadas foram rotuladas a partir da primitiva básica peculiar de cada aplicação. As aplicações desenvolvidas para a FPA foram testadas com o uso de `calloc` e foram comparadas com uma aplicação utilizando `streams`, que preserva o mesmo formato de execução dos testes. Tais aplicações, são apresentadas nos testes como "FPA", e "Streams", respectivamente. Ocasionalmente foram executados alguns testes utilizando a primitiva `mmap` , porém em tais situações a rotulação das aplicações executadas é explícita.

### Parâmetros dos testes

Para os testes realizados, as configurações da FPA assumiram os valores de 2 GB para o parâmetro *LM*, 128 MB para o parâmetro *LA* e 32 MB para cada buffer.

Os arquivos de entrada para os testes de leitura consistem em arquivos texto de tamanho definido pelo teste, podendo ser de 1, 2 ou 4 GB. Analogamente, para os testes de escrita são aplicadas as mesmas configurações, porém nestas operações os arquivos são a saída da aplicação.

### Operações analisadas

As análises dos testes tratam de observar o comportamento e desempenho das aplicações durante operações de leitura, escrita e também em ambas as operações. As operações de leitura consistem apenas na leitura das informações do disco, enquanto

na escrita, um dado é gerado em memória e posteriormente enviado ao disco. Já para os testes de leitura e escrita, a informação lida é modificada em memória, e logo após enviada novamente para o dispositivo de armazenamento.

Os resultados de cada execução são apresentados através de gráficos, onde estes podem ser apresentados com diferente número de *threads*, quantidade de processamento na aplicação ou tamanho do arquivo de entrada/saída. Com base nestes dados também são apresentados os gráficos de *speedup* e eficiência, onde o tempo de execução sequencial utilizado no cálculo do *speedup* é o tempo da execução sequencial utilizando *streams*.

Como descrito na seção 3.2, a implementação da FPA utiliza uma *thread* apenas para escrita dos dados em disco. Tendo isto, todas as execuções utilizando FPA contam com esta *thread* adicional, e diversos gráficos apresentados nas seções seguintes apresentam o número de *threads* utilizadas na implementação testada. Em tais gráficos, o número de *threads* não inclui a *thread* adicional da FPA, sendo referente apenas a aquelas criadas pela aplicação, e não pela biblioteca.

### 4.2.2 Particionamento uniforme do arquivo

O teste com particionamento uniforme do arquivo possui o intuito de verificar o desempenho da aplicação sob uma situação de acesso a disco onde as *threads* não compartilham buffers de dados, porém todas acessam o mesmo arquivo. Os testes são efetuados de forma que cada *thread* realiza o acesso sequencial sem sobreposição de regiões no arquivo, como indicado na figura 4.3. As posições onde cada *thread* acessa o arquivo são estáticas e definidas pelo ID da *thread*, uma vez que o arquivo é dividido pelo número de *threads* utilizado no teste. Por exemplo, na execução de um teste envolvendo um arquivo de 4 GB e utilizando 8 *threads*, a primeira *thread* é responsável pelo primeiro $1/4$ do arquivo, a segunda *thread* pelo segundo $1/4$ e assim sucessivamente.

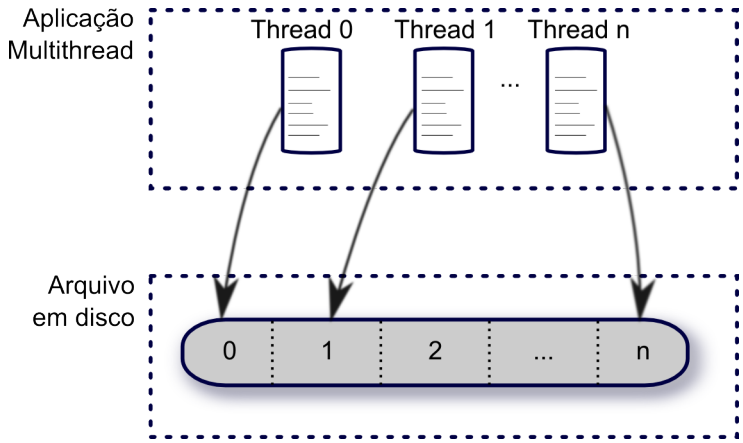



**Figura 4.3:** Teste utilizando particionamento uniforme do arquivo

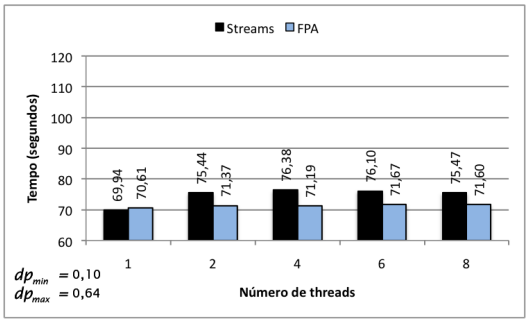
### Leitura

Os testes de leitura são realizados através de operações de leitura de 1 KB de dados por vez do arquivo, até que toda a região responsável pela *thread* tenha sido atendida.

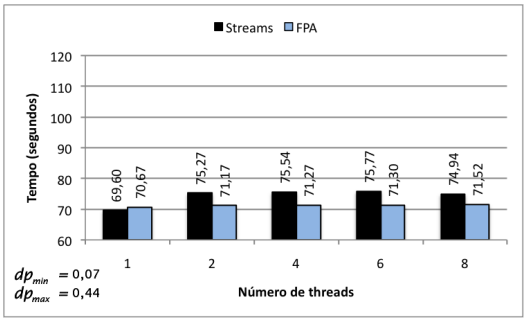
O objetivo destes testes é fornecer dados para que uma análise do comportamento das operações de leitura seja realizado. O pseudocódigo apresentado na figura 4.4 mostra como é realizado o teste, bem como os pontos onde existem o paralelismo e o cálculo da aplicação.

```
1  INÍCIO DA SEÇÃO PARALELA OPENMP
2          arquivo = open(arquivo_entrada);
3          seek( 1024 * 1024 * gigabytes / nthreads * thread_id );
4
5          for( j=0; j< 1024/nthreads; j++ )
6          {
7                  for( i=0; i<1024*gigabytes; i++ )
8                  {
9                          //Leitura de 1KB do disco
10                         read(dados);
11
12                         //cálculos da aplicação
13                         calcula(processamento);
14                 }
15         }
16 FIM DA SEÇÃO PARALELA OPENMP
```

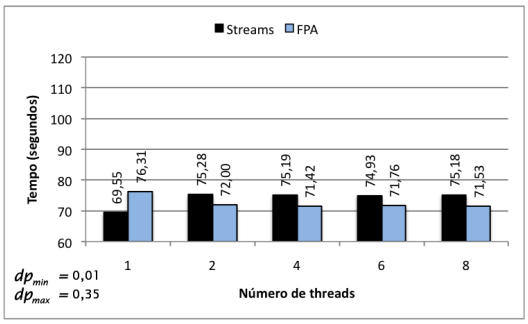
**Figura 4.4:** Pseudocódigo da leitura utilizando mapeamento uniforme do arquivo



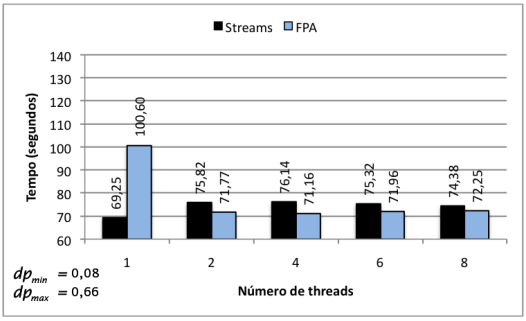

(a) Processamento de 0 iterações



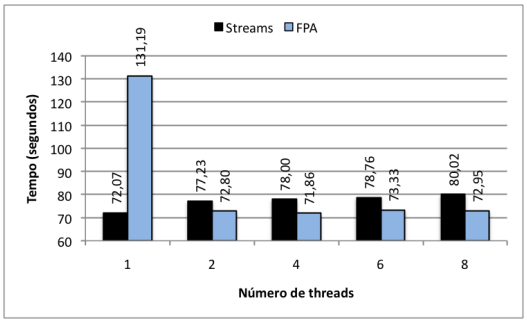

(b) Processamento de 10 iterações



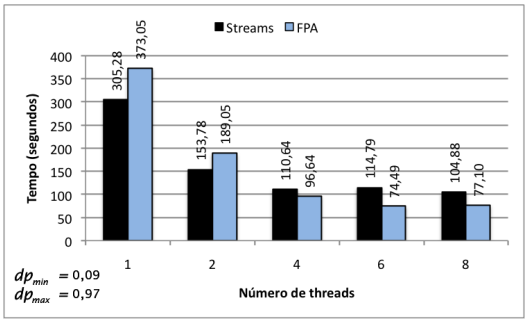

(c) Processamento de 100 iterações



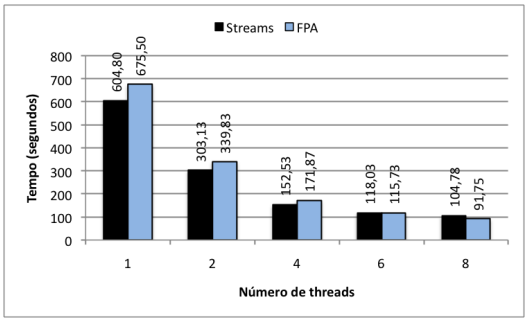

(d) Processamento de 500 iterações



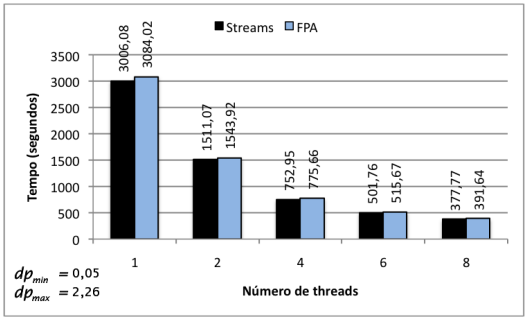

(e) Processamento de 1000 iterações




(f) Processamento de 5000 iterações




(g) Processamento de 10000 iterações




(h) Processamento de 50000 iterações

**Figura 4.5:** Leituras utilizando arquivos de 4 GB

Nos gráficos apresentados na figura 4.5 é possível visualizar que na maioria dos casos testados a FPA possui ganho de desempenho, em relação aos *streams*. Este ganho deve-se principalmente ao preenchimento completo dos buffers, onde isto implica em um menor número de chamadas de sistema requisitando operações de I/O e assim

resultando em uma sobrecarga menor ao sistema operacional.

A utilização da FPA toma vantagem nas execuções onde existe o paralelismo, pois utilizando diversas *threads* é possível aproveitar a a sobreposição entre processamento da aplicação e acesso a disco. Já em execuções sequenciais este aproveitamento não acontece, e o gerenciamento dos buffers da FPA torna-se custoso para a aplicação, ocasionando em um tempo de execução final maior. Como pode ser visto no teste sequencial do gráfico 4.5(a), onde não existe processamento de cálculo na aplicação, a diferença de tempo entre as diferentes execuções é mínima, podendo serem considerados tempos iguais. À medida que o processamento de cálculo aumenta, torna-se visível que a FPA possui um ganho de desempenho inferior aos *streams* em operações sequenciais.

Observando os gráficos da figura 4.5 é possível notar que à medida que a quantidade de processamento de cálculo presente na aplicação aumenta, o ganho de desempenho da FPA em relação ao uso de *streams* decresce. Tal comportamento acontece devido a que, com uma quantidade de processamento maior, a implementação utilizando *streams* consegue efetuar um aproveitamento da relação entre o processamento e acesso a disco de forma mais eficiente.

Há um ganho de desempenho da FPA em relação aos *streams* quando são utilizadas pequenas quantidades de processamento de cálculo na aplicação. Isto acontece devido à leitura do buffer completo, que possibilita que os dados já estejam em memória nas leituras seguintes. Esta leitura realizando o preenchimento completo do buffer acarreta em que inicialmente aconteça uma serialização das leituras pelas *threads*, e após isso ocorra um balanceamento entre o processamento da aplicação e o acesso das informações no disco. Esta situação acontece diferente com os *streams*, pois com quantidades pequenas de processamento não há sobreposição do processamento da aplicação e do envio dos dados ao disco. A figura 4.6 mostra os resultados com diversos valores de processamento, e este comportamento pode ser melhor compreendido.

Em situações onde a aplicação possui um maior processamento de cálculo, o *stream* apresenta-se como uma técnica mais vantajosa. Ambas as implementações realizam a sobreposição entre o processamento e o acesso a disco, porém com a FPA existe uma desvantagem nessa situação. Como no início da aplicação todas as *threads* realizam a requisição de leitura ao mesmo tempo e as operações de leitura na FPA acontecem com 1 buffer de cada vez, existe a sequencialização das leituras entre as *threads*. Desta forma, a última *thread* que executar a leitura do disco terá de aguardar até que todas as demais *threads* tenham lido seus dados e preenchido seus buffers. Esta espera da *thread* implica em um tempo desperdiçado que poderia estar sendo apro-

veitado para realizar o processamento da aplicação. Desta forma, é mais vantajoso realizar leituras de pequenas quantidades de dados e em seguida realizar os cálculos do que ler uma grande quantidade de informações.

Para aplicações com 2 threads, o ponto onde passa a existir desvantagem com uso da FPA é dado entre 1000 e 5000 iterações (gráficos 4.5(e) e 4.5(f)), para aplicações com 4 *threads* é dado entre 5000 e 10000 (gráficos 4.5(f), 4.5(g) e 4.6) e para aplicações com 6 *threads* é dado entre 10000 e 50000 (gráficos 4.5(g) e 4.5(h)).

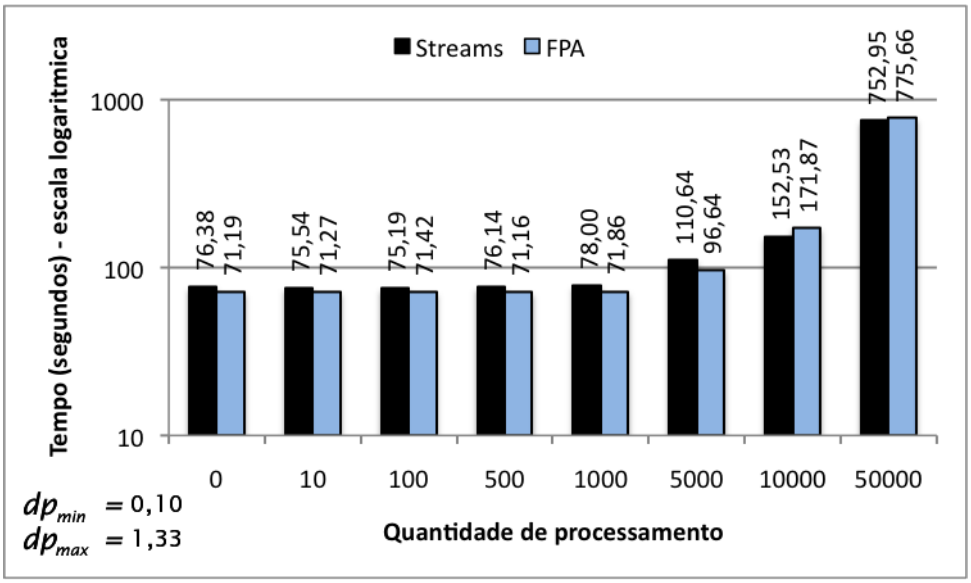
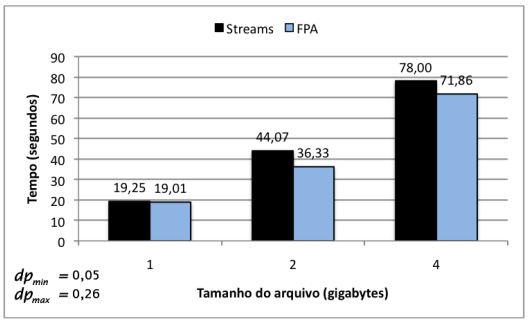
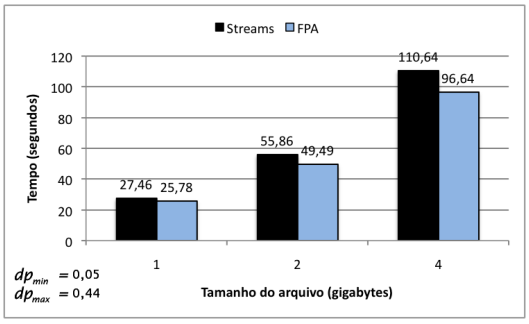


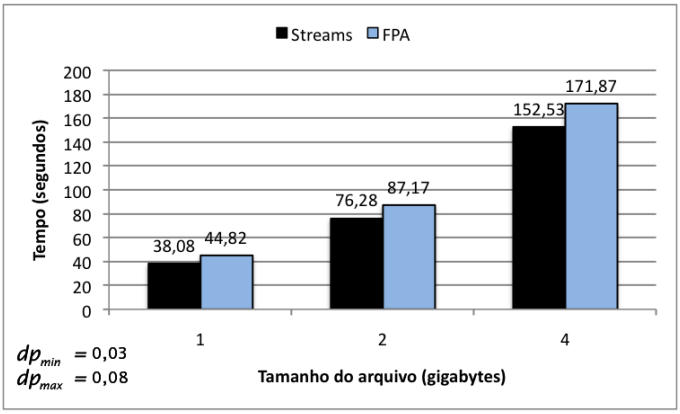

**Figura 4.6:** Leituras com arquivos de 4 GB e 4 *threads*




(a) Processamento de 1000 iterações




(b) Processamento de 5000 iterações




(c) Processamento de 10000 iterações

**Figura 4.7:** Leituras utilizando 4 *threads*

Analisando os gráficos da figura 4.7 é possível visualizar que o ganho de desempenho da FPA em relação aos *streams* é independente do tamanho do arquivo. O fator determinante para o desempenho da FPA é a quantidade de processamento presente na aplicação. O ganho obtido permanece praticamente constante ao utilizar um mesmo número de *threads* e quantidade de processamento. Como o acesso ao arquivo é realizado no momento da criação do buffer e não é necessário enviar informações de volta ao disco, a *thread* que gerencia a liberação de buffers apenas libera a memória utilizada pelo buffer, que é uma operação rápida, se comparada às operações de acesso a disco. Desta forma, mesmo com o aumento de liberações de buffers quando aumenta o tamanho do arquivo, o seu custo é praticamente insignificante e não influencia no desempenho. Com isto, a memória utilizada pela FPA chega ao máximo de 1 buffer além do limite *LA*, mantendo o uso de memória pela biblioteca baixo e diminuindo a probabilidade de que a memória virtual do SO seja utilizada pela aplicação.

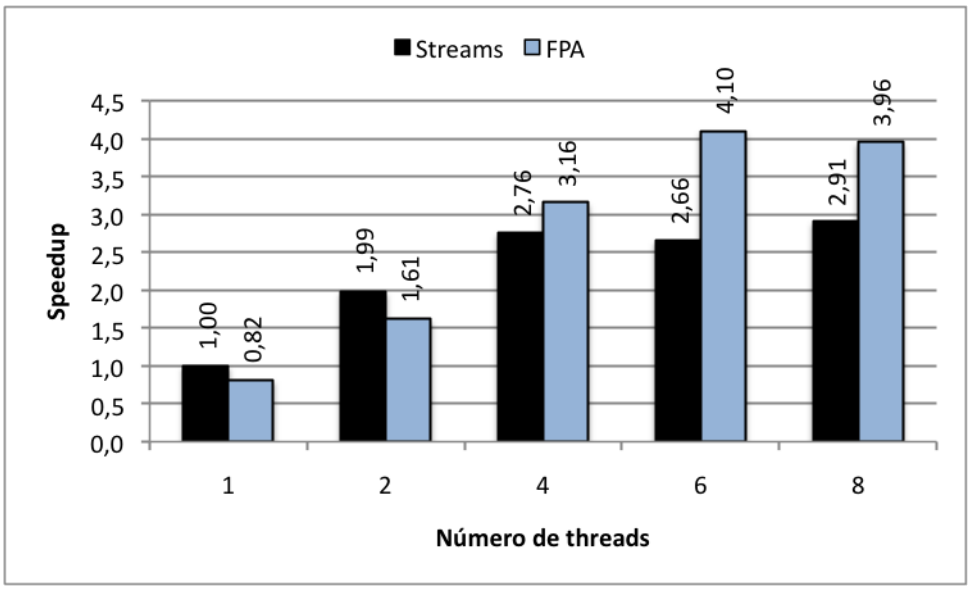



**Figura 4.8:** *Speedup* da leitura utilizando particionamento uniforme do arquivo com arquivos de 4 GB, 4 *threads* e processamento de 5000 iterações

De acordo com o gráfico da figura 4.8, é possível visualizar que a FPA é vantajosa em aplicações com número de *threads* superior a 2. Para 2 *threads* ainda não existe ganho de desempenho, pois o tempo gasto devido à concorrência ao disco da FPA ainda não é suficiente para compensar o custo da utilização dos buffers. A mesma situação pode ser visualizada com 1 thread, onde não existe concorrência no acesso ao disco.

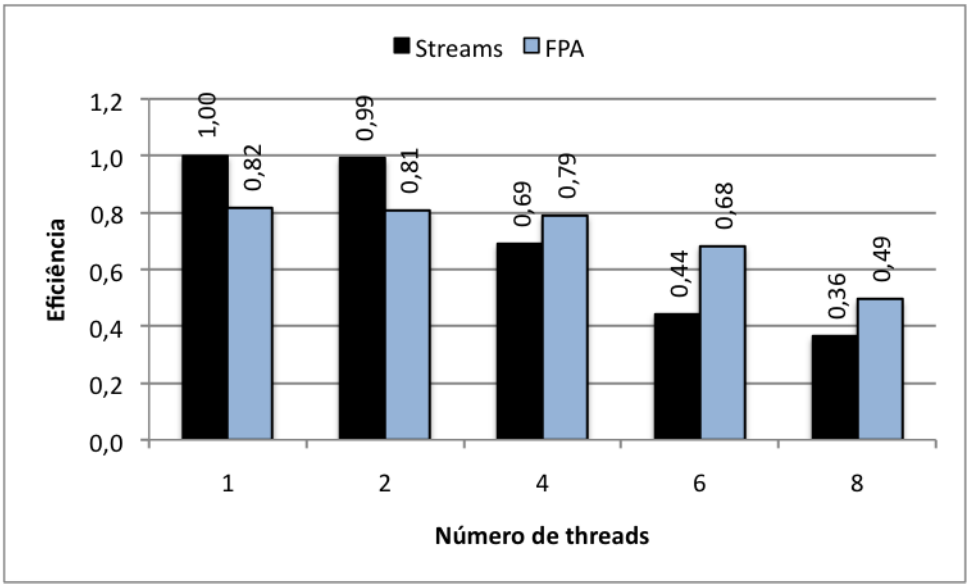
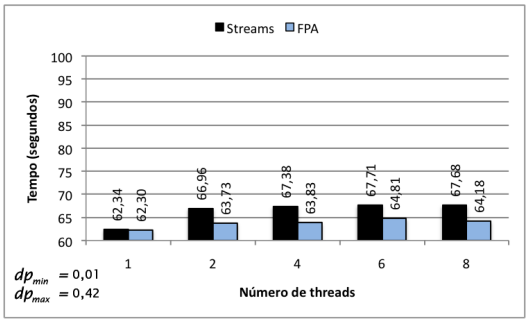


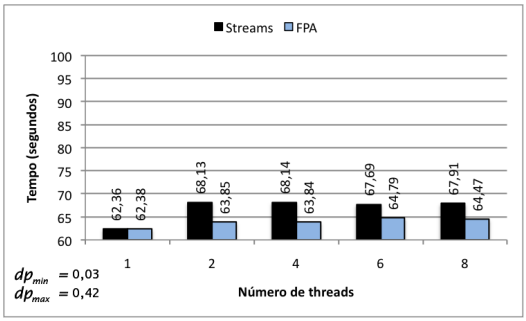

**Figura 4.9:** Eficiência da leitura utilizando particionamento uniforme do arquivo

O ganho de desempenho visualizado no gráfico de *speedup* (figura 4.8) é refletido no gráfico da eficiência (figura 4.9). Neste nota-se que apesar da aplicação com FPA não ser altamente eficiente, esta ainda é superior à aplicação com *streams*. Porém, convém ressaltar que na eficiência é refletido também o método de paralelismo das

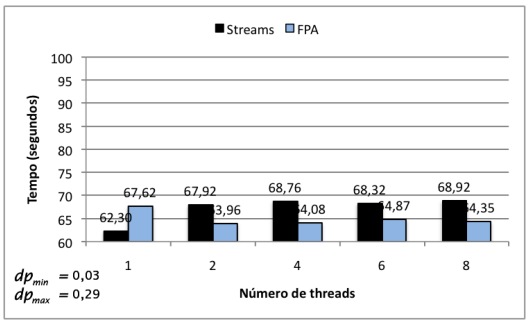
tarefas nas aplicações e o método de acesso a disco é apenas parte de todo esse processo. Outro ponto analisado é que o *speedup* dos testes utilizando 8 *threads* é inferior ao *speedup* utilizando 6 threads, apesar de existirem 8 unidades de processamento na máquina de teste. Este resultado deve-se a que a *thread* da FPA responsável pelo gerenciamento dos buffers ocupa espaço na CPU, assim lugar das *threads* da aplicação. Desta forma, os cálculos da aplicação levam um tempo maior para serem realizados, resultando em um tempo de execução da aplicação maior.



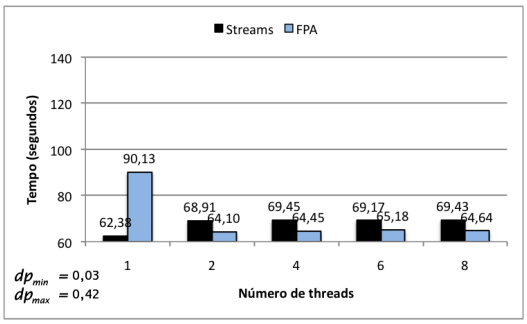

(a) Processamento de 0 iterações



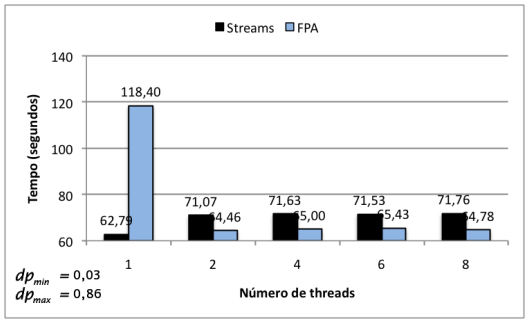

(b) Processamento de 10 iterações



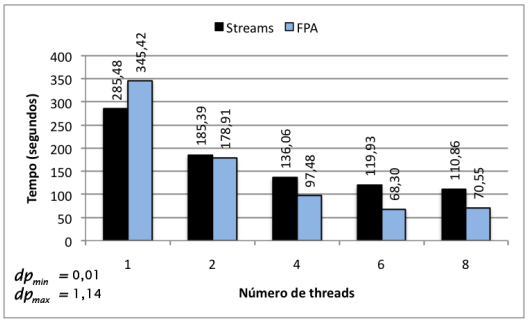

(c) Processamento de 100 iterações



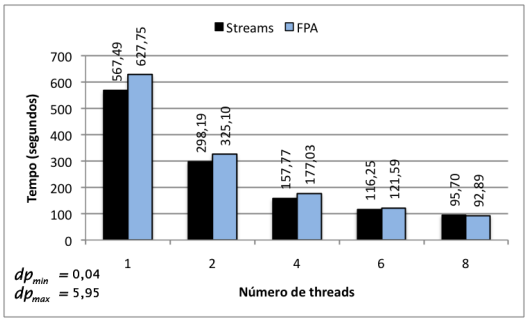

(d) Processamento de 500 iterações



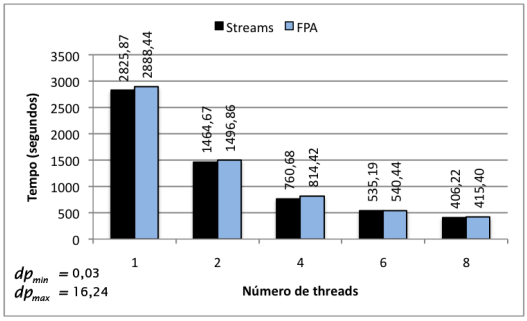

(e) Processamento de 1000 iterações




(f) Processamento de 5000 iterações




(g) Processamento de 10000 iterações




(h) Processamento de 50000 iterações

**Figura 4.10:** Leituras com arquivos de 4 GB na máquina do LAHPC

Nos gráficos apresentados na figura 4.10 percebe-se que o comportamento dos resultados obtidos é semelhante ao dos testes executados na máquina OM. Apesar de serem máquinas diferentes com configurações de *hardware* distintas, a execução dos testes sob os mesmos parâmetros proporciona resultados semelhantes. A mesma

conclusão pode ser obtida ao observar os gráficos das figuras 4.11, 4.12 e 4.13.

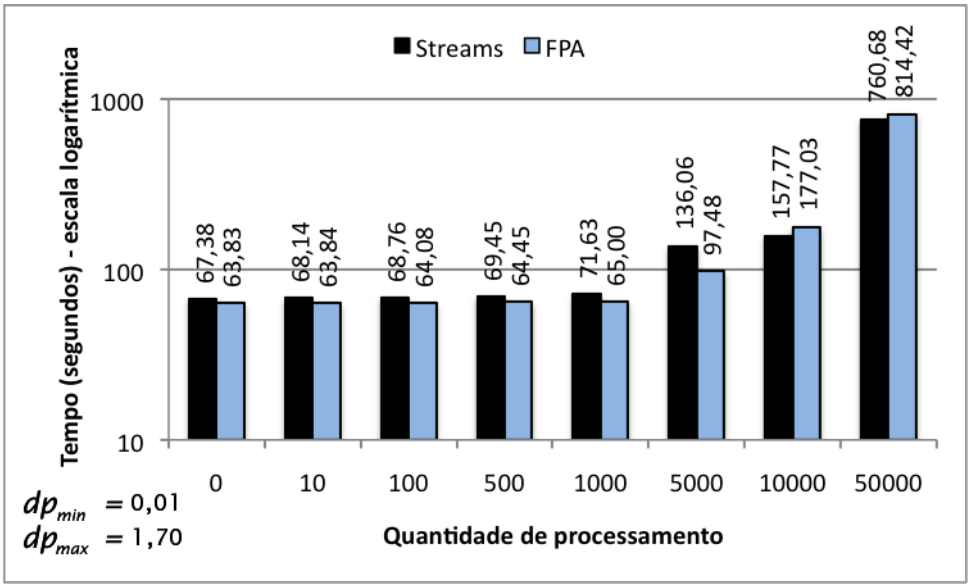



**Figura 4.11:** Leituras com arquivos de 4 GB e 4 *threads* na máquina do LAHPC

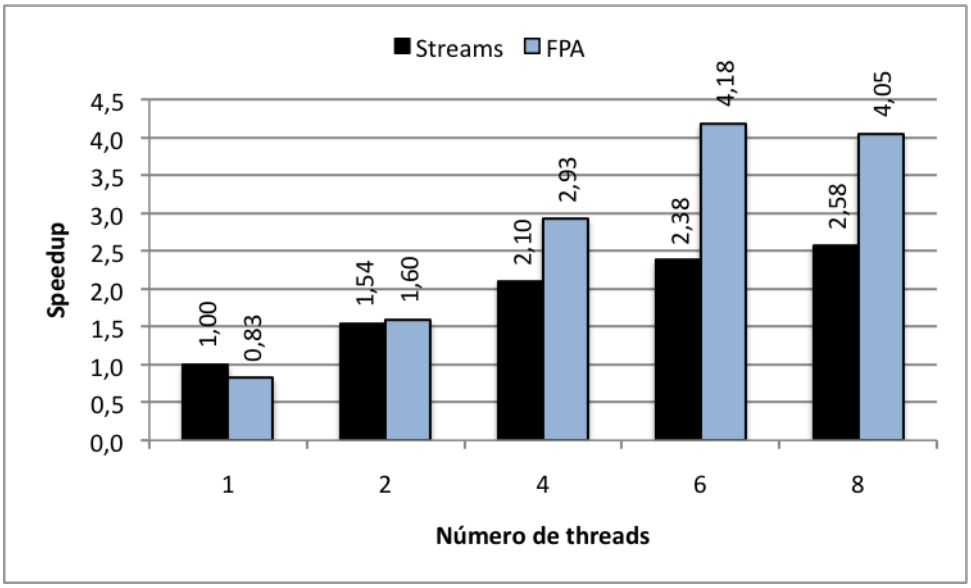



**Figura 4.12:** *Speedup* da leitura utilizando particionamento uniforme do arquivo executados na máquina do LAHPC, com arquivos de 4 GB e processamento de 5000 iterações

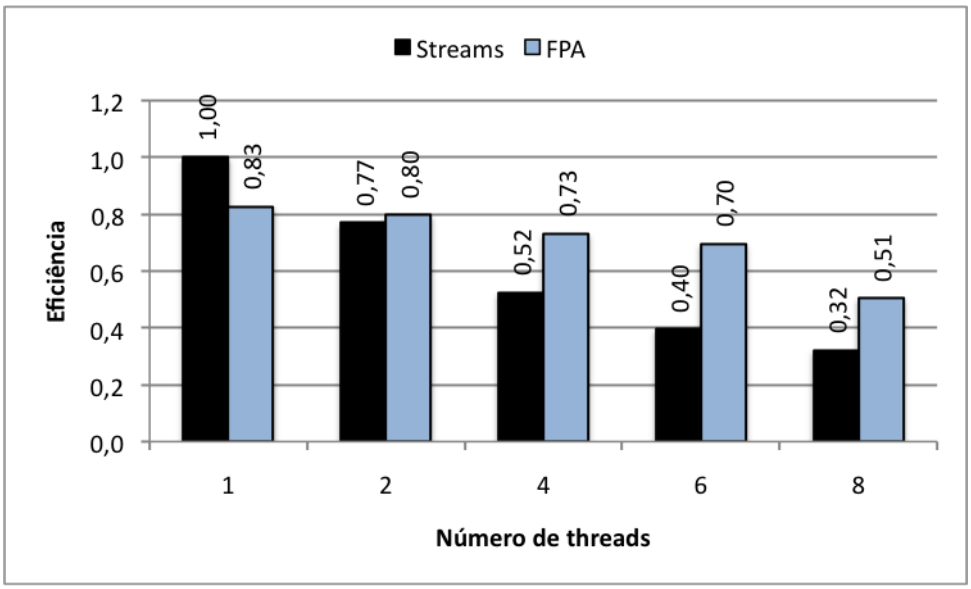



**Figura 4.13:** Eficiência da leitura utilizando particionamento uniforme do arquivo na máquina do LAHPC

### Escrita

O objetivo do teste é analisar o comportamento da FPA em operações de escrita de dados, comparando os resultados obtidos com entre as diferentes técnicas. Para as operações de escrita com particionamento uniforme do arquivo foram realizados testes com diferentes versões da biblioteca FPA, utilizando `calloc` e `mmap`. A versão com `mmap` conta com 2 variações:

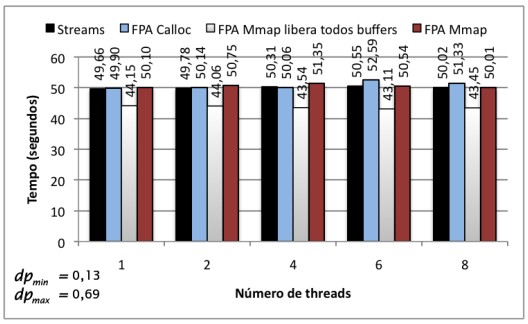
- Uma versão que envia ao disco todos os buffers que não estão sendo utilizados no momento, ou seja, que não possuem nenhuma *thread* com ponteiros de escrita ou leitura sobre a região de dados do buffer. As modificações desta versão foram realizadas apenas na *thread* de gerenciamento de buffers, onde as demais rotinas permanecem iguais;

- Uma versão que envia um único buffer não utilizado ao disco, à medida que o limite de memória *LA* for ultrapassado. Caso a quantidade de memória utilizada continue sendo maior que *LA*, um novo buffer será liberado e assim sucessivamente. Esta liberação de buffers acontece exatamente da mesma forma que a implementação utilizando `calloc`.

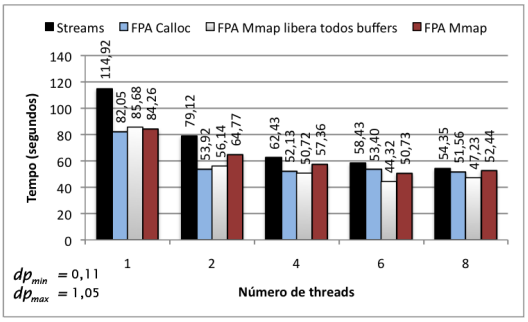
Tais versões da FPA foram rotuladas como "FPA calloc", "FPA mmap libera todos os buffers"e "FPA mmap", respectivamente. O pseudocódigo de como o teste é executado pode ser visto na figura 4.14.

```
1  INÍCIO DA SEÇÃO PARALELA OPENMP
2          arquivo = open(arquivo_entrada);
3          seek( 1024 * 1024 * gigabytes / nthreads * thread_id );
4
5          for( j=0; j< 1024/nthreads; j++ )
6          {
7                  for( i=0; i<1024*gigabytes; i++ )
8                  {
9                          //cálculos da aplicação
10                         calculos(processamento);
11
12                         //Escrita de 1KB no disco
13                         write(dados);
14                 }
15         }
16 FIM DA SEÇÃO PARALELA OPENMP
```

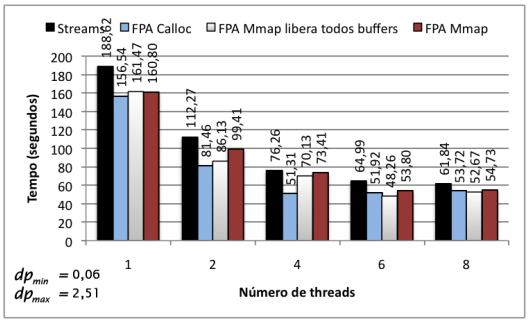
**Figura 4.14:** Pseudocódigo da escrita utilizando mapeamento uniforme do arquivo



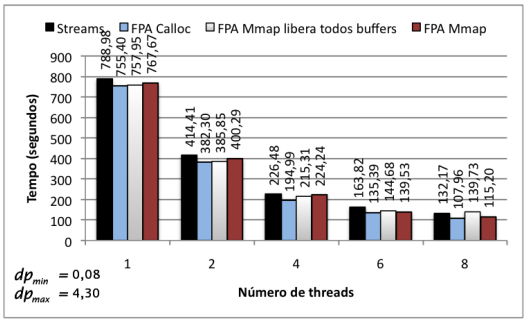

(a) Processamento de 0 iterações




(b) Processamento de 5000 iterações




(c) Processamento de 10000 iterações




(d) Processamento de 50000 iterações

**Figura 4.15:** Escritas de arquivos de 1 GB

Nas aplicações Fpa calloc e FPA mmap são sempre mantidos em memória os últimos 4 buffers, pois esta é a quantidade de memória permitida pelo limite *LA* de dados em memória sem que haja necessidade de serem enviados ao disco. A vantagem de manter estes dados armazenados é que possibilita que os dados recém escritos possam ser reutilizados por outras *threads*, caso queiram fazer alterações ou leituras nessas regiões do arquivo. Por outro lado, a desvantagem é que os dados que se encontram armazenados serão enviados ao disco no final da execução, fazendo a aplicação esperar pelo envio das informações ao disco. Ao contrário disso, a implementação utilizando FPA Mmap que libera todos os buffers consiste em enviar ao disco os dados armazenados em todos os buffers que não estão sendo utilizados, onde os dados são enviados ao disco assim que a quantidade de memória total utilizada pelos buffers for maior que o limite *LA* de 128 MB, como acontece nas demais implementações da FPA.

Nos gráficos da figura 4.15 visualiza-se que os resultados obtidos com FPA Mmap possuem ganho de desempenho inferior aos obtidos com o uso da FPA Mmap que libera todos os buffers livres para todas as situações testadas. Isto se deve à quantidade maior de envio de dados ao disco, o que aumenta a probabilidade de concorrência de acesso ao disco, como também o custo total dos acessos.

Comparando os gráficos da figura 4.15 é possível visualizar que na ausência de processamento na aplicação, a FPA calloc não é uma estratégia vantajosa. Este resultado acontece devido ao envio de dados ainda não armazenados ao disco após o término dos cálculos da aplicação. Como com o Mmap o SO pode sincronizar com o disco dados dos buffers durante a execução, parte da informação armazenada será enviada ainda durante o processamento dos cálculos da aplicação, levando a uma espera menor pelo envio dos dados ao disco no final da execução da aplicação que utiliza Mmap. O Mmap e a sincronização de dados com o disco são tratados no capítulo 2.6.2, da página 33.

Para a quantidade de processamento 10000 (4.15(c)) existe vantagem com a FPA Calloc inclusive para aplicações que utilizam uma única *thread*. Este ganho de desempenho acontece não somente pela sobreposição processamento/envio de dados, mas também pela diminuição da sobrecarga de I/O ao SO devido à diminuição de chamadas de sistema.

Através da análise dos testes utilizando mapeamento uniforme foi concluído que a versão principal da FPA a ser continuada e utilizada nos testes seria a versão FPA Calloc, devido ao seu ganho de desempenho em relação ao FPA mmap. Já com a versão FPA Mmap libera todos os buffers livres, foi obtido ganho de desempenho em relação à FPA Calloc, porém esta não permite a reutilização dos buffers, e caso qualquer dado

recente queira ser acessado novamente este terá que ser buscado no disco novamente. Desta forma, o FPA Mmap libera todos os buffers livres também não está no foco do trabalho, porém ambas as versões estão integradas à implementação da FPA, podendo serem acessadas simplesmente através de parâmetros na inicialização da FPA.

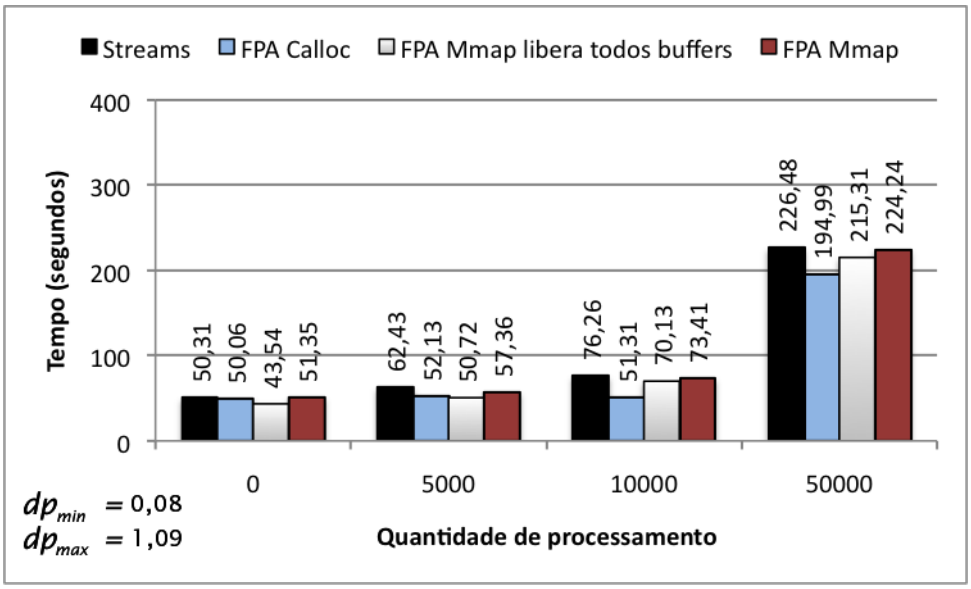
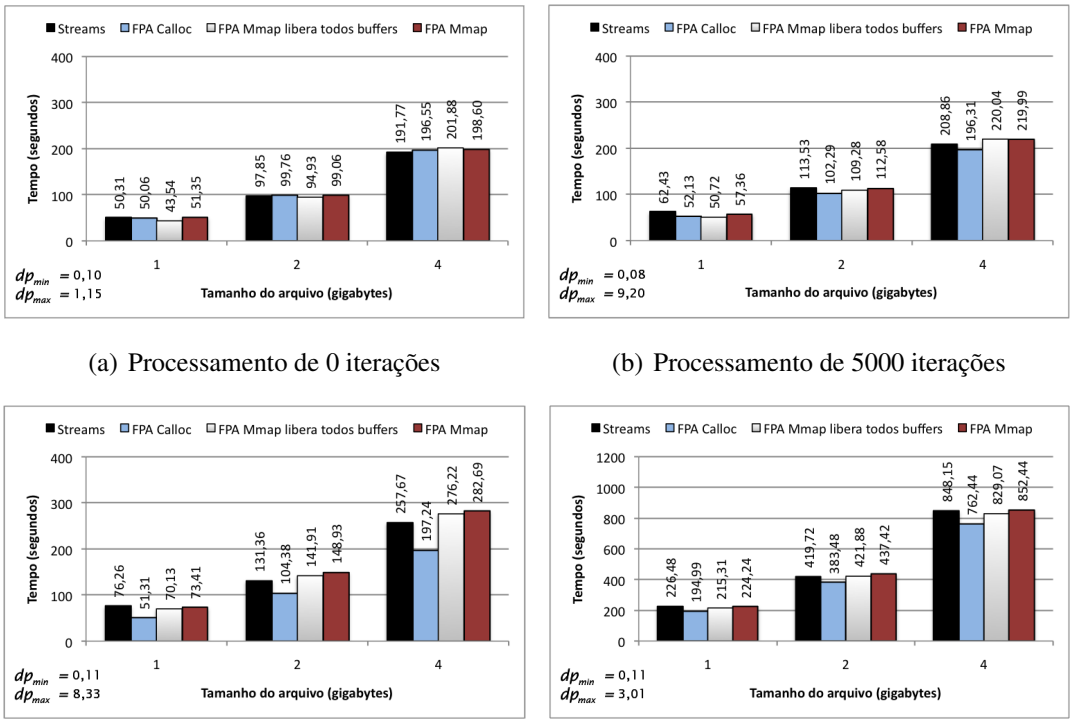



**Figura 4.16:** Escritas de arquivos de 1 GB utilizando 4 *threads*

Através da figura 4.16 é possível visualizar que a quantidade de processamento de cálculo da aplicação impacta diretamente no desempenho da biblioteca. Esta situação ocorre devido à sobreposição de processamento/envio de dados, onde a utilização de streams prevalece como a mais vantajosa em aplicações sem ou com baixo nível de processamento.




(a) Processamento de 0 iterações

(b) Processamento de 5000 iterações

(c) Processamento de 10000 iterações

(d) Processamento de 50000 iterações

**Figura 4.17:** Escritas utilizando mapeamento uniforme dos arquivos, com 4 *threads*

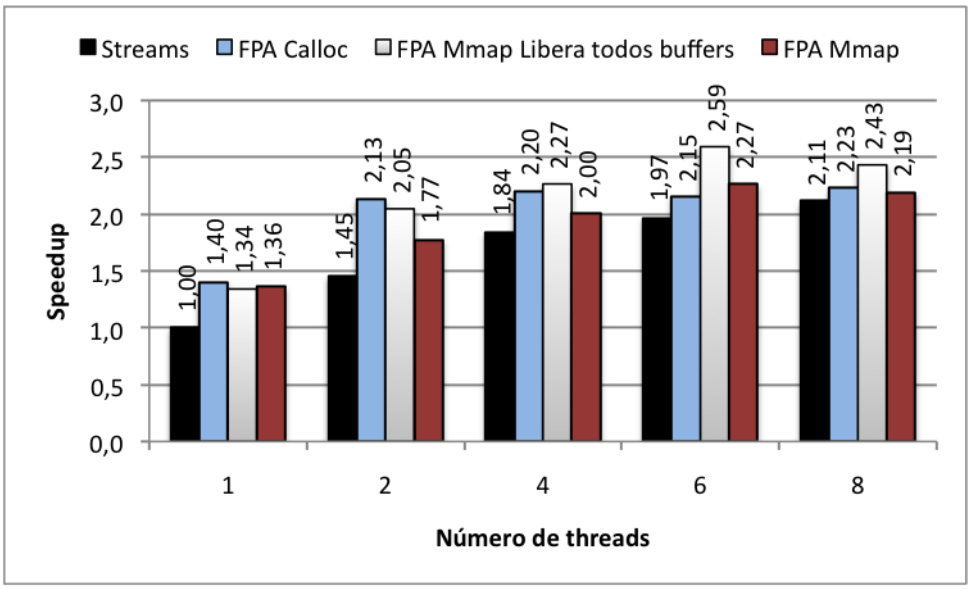



**Figura 4.18:** *Speedup* da escrita utilizando particionamento uniforme do arquivo, com arquivos de 4 GB e processamento de 5000 iterações

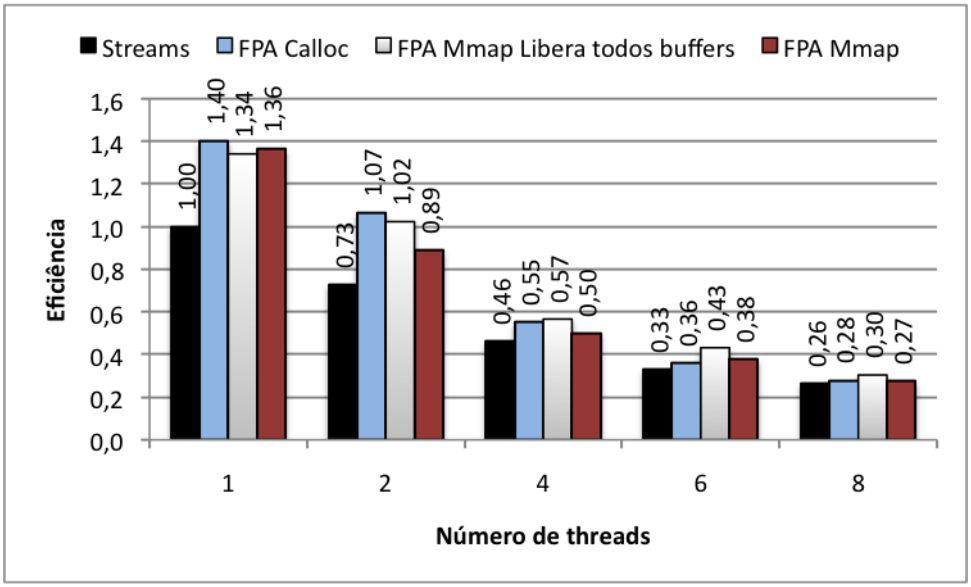



**Figura 4.19:** Eficiência da escrita utilizando particionamento uniforme do arquivo, com arquivos de 4 GB e processamento de 5000 iterações

### Leitura e escrita

A forma como o teste que realiza leitura e escrita é executado pode ser vista na figura 4.20, onde o processamento de cálculo acontece antes das operações de leitura e escrita.

```
1  INÍCIO DA SEÇÃO PARALELA OPENMP
2          arquivo = open(arquivo_entrada);
3          seek( 1024 * 1024 * gigabytes / nthreads * thread_id );
4
5          for( j=0; j< 1024/nthreads; j++ )
6          {
7                  for( i=0; i<1024*gigabytes; i++ )
8                  {
9                          //Leitura de 1KB do disco
10                         read(dados);
11
12                         //cálculos da aplicação
13                         calculo( processamento );
14
15                         //alteração dos dados lidos
16                         altera(dados);
17
18                         seek( -1KB );
19
20                         //escrita de 1KB no disco
21                         write(dados);
22                 }
23         }
24 FIM DA SEÇÃO PARALELA
```

**Figura 4.20:** Pseudocódigo da leitura e escrita utilizando mapeamento uniforme do arquivo

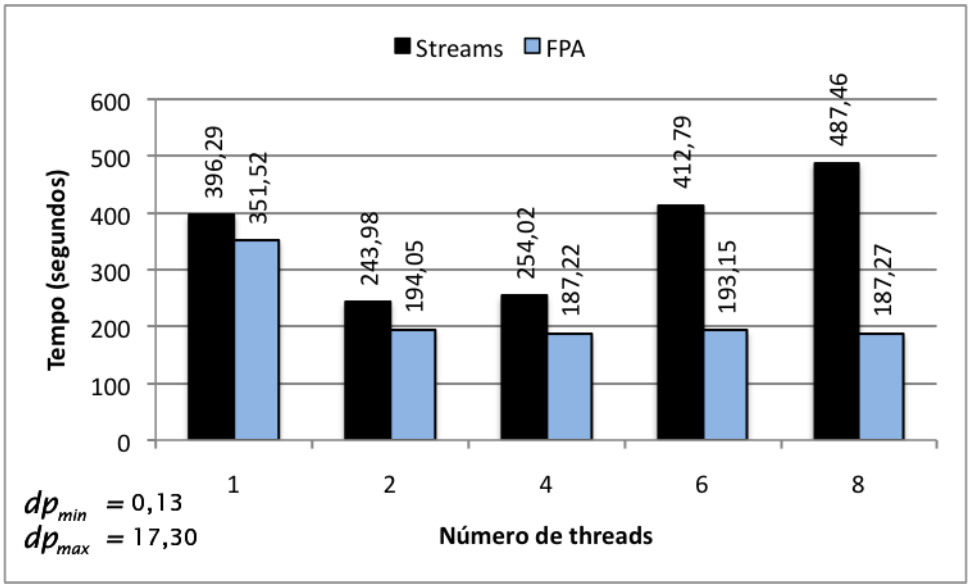



**Figura 4.21:** Leitura e escrita com arquivo de 4 GB e processamento de 5000

Nos testes envolvendo leitura e escrita ocorre um melhor aproveitamento da concorrência ao disco pela FPA, pois existem ambas operações de leitura e escrita, e a biblioteca possui vantagens em ambos os tipos de acesso.

A divisão do tempo total de processamento da aplicação é aproveitada pelos *streams* até os testes utilizando 2 *threads*. A partir desse ponto a concorrência ao disco aumenta a ponto de inviabilizar a aplicação paralela, pois com 6 e 8 *threads* o tempo de execução é maior do que o tempo gasto na aplicação sequencial, como pode ser visto no gráfico da figura 4.21. Este resultado é refletido no *speedup* (figura 4.23), onde possui valores inferiores a 1.

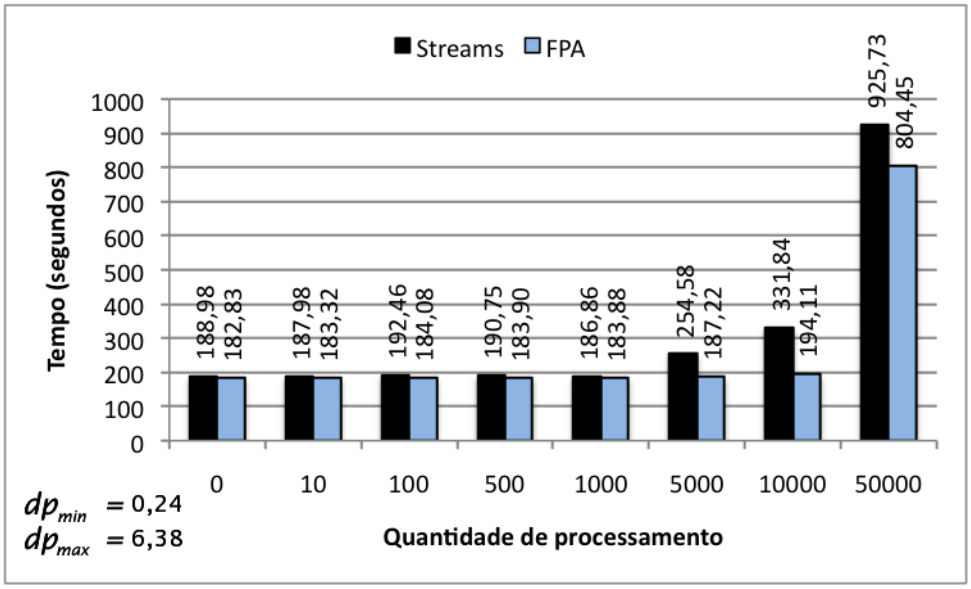



**Figura 4.22:** Leitura e escrita com arquivo de 4 GB e 4 *threads*

A sobreposição entre o processamento e o acesso a disco também acontece. Porém,

nestes casos existem as otimizações em operações de leitura e escrita, e nas situações onde as operações de leitura não apresentam ganho de desempenho, as operações de escrita são vantajosas, e vice-versa. Desta forma, o resultado final de ambas é vantajoso, inclusive para o caso sequencial.

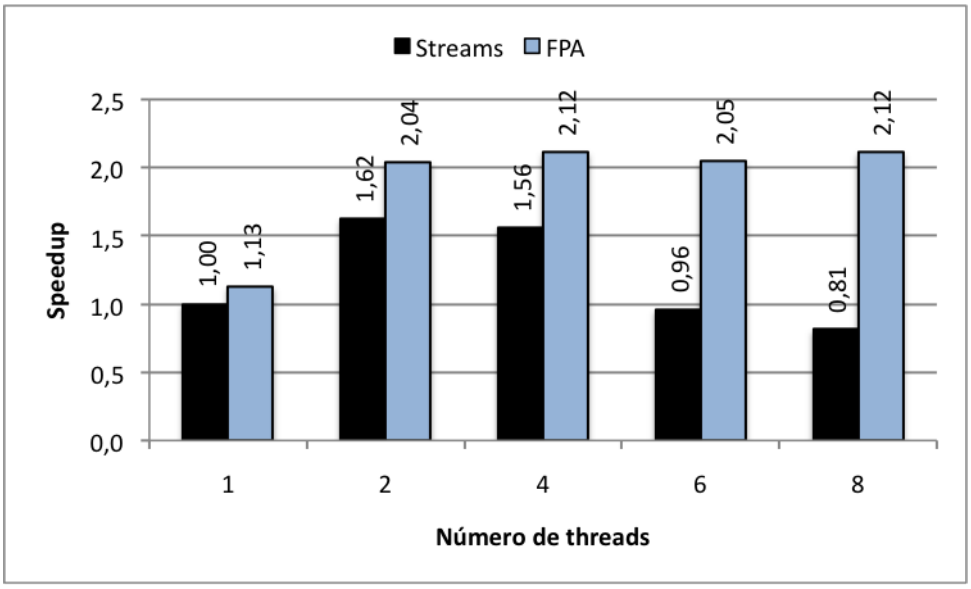



**Figura 4.23:** *Speedup* da leitura e escrita utilizando mapeamento uniforme do arquivo, com arquivos de 4 GB e processamento de 5000 iterações

Na figura 4.23 é possível visualizar o ganho de desempenho da FPA em relação aos *streams* utilizando leitura e escrita de dados. Como a leitura de dados do disco é feita utilizando 32 MB, existe a vantagem da menor sobrecarga ao sistema operacional. Esta vantagem acontece novamente no momento da escrita, quando o buffer é enviado por completo ao disco. Dado que a *thread* que lê os dados no disco não realiza operações de *seek* para outras posições no arquivo, os dados permanecem no buffer em memória, onde serão alterados pelas rotinas de escrita da mesma *thread*.

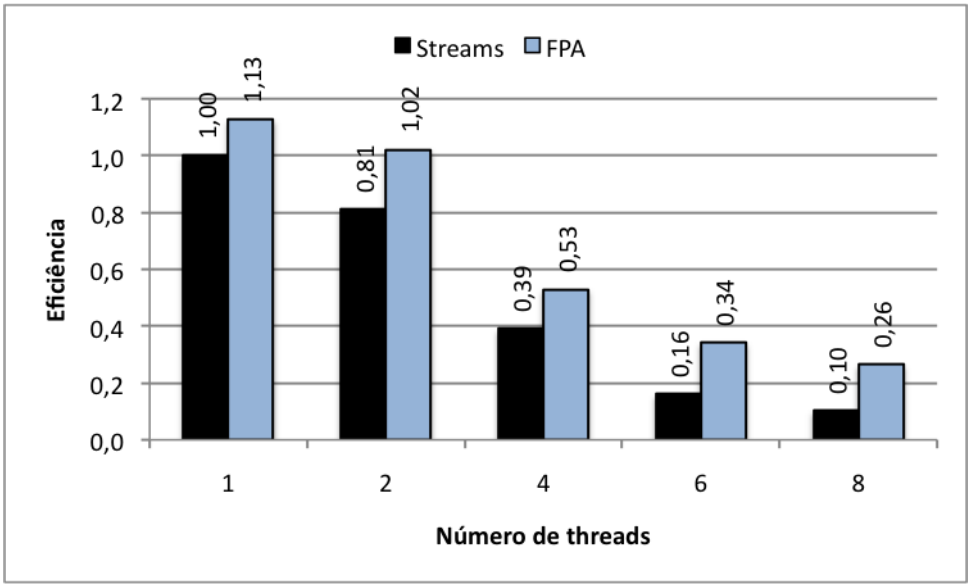



**Figura 4.24:** Eficiência da leitura e escrita utilizando particionamento uniforme do arquivo, com arquivos de 4 GB e processamento de 5000 iterações

### 4.2.3 Particionamento dinâmico do arquivo

Este teste é realizado utilizando uma divisão dinâmica do arquivo entre as *threads*, onde cada *thread* é responsável pela leitura e/ou escrita de 16 MB no arquivo. A especificação das regiões que cada *thread* será responsável é realizada em formato de fila, de forma que quando cada *thread* terminar suas operações na região definida, a próxima região de 16 MB da fila é selecionada. O teste termina quando não houverem mais regiões para serem acessadas no arquivo.

A partir do momento em que uma *thread* assume uma determinada região, são realizadas operações de leitura/escrita múltiplas de 1 KB até que todos os 16 MB tenham sido atendidos.

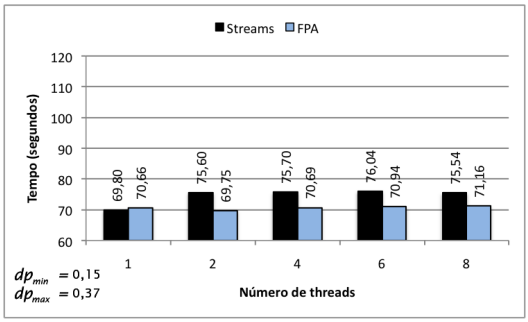
O objetivo do teste é analisar o desempenho da FPA em situações em que mais de uma *thread* acessa o mesmo buffer, afinal a região que cada *thread* acessa no disco é menor que o tamanho dos buffers da biblioteca.

#### Leitura

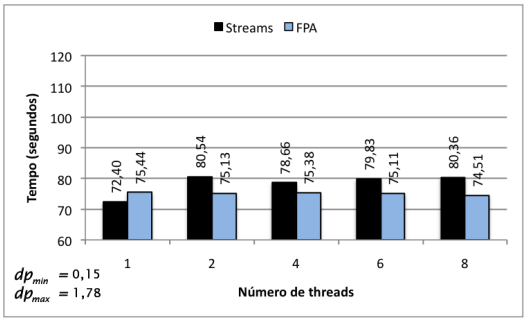
Uma ilustração de como é realizado os testes de leitura utilizando o mapeamento dinâmico do arquivo pode ser vista na figura 4.25. Nesta, é possível visualizar em detalhes como é realizada a escolha de cada região pelas *threads*, bem como o acesso realizado.

```
1  contador_regioes=0;
2  numero_regioes = tamanho total do arquivo / 16 MB;
3  INÍCIO DA SEÇÃO PARALELA OPENMP
4          arquivo = open(arquivo_entrada);
5          regiao=0;
6          while( 1 )
7          {
8                  //início da seção crítica
9                  regiao = contador_regioes;
10                 contador_regioes++;
11                 //fim da seção crítica
12                 if( região >= numero_regioes )
13                         break;
14                 seek(arquivo, regiao * 16 MB );
15                 for( i=0; i < 16 * 1024; i++ )
16                 {
17                         //Leitura de 1 KB
18                         read(dados);
19                         //Realiza os cálculos
20                         calculos(dados);
21                 }
22         }
23 FIM DA SEÇÃO PARALELA OPENMP
```

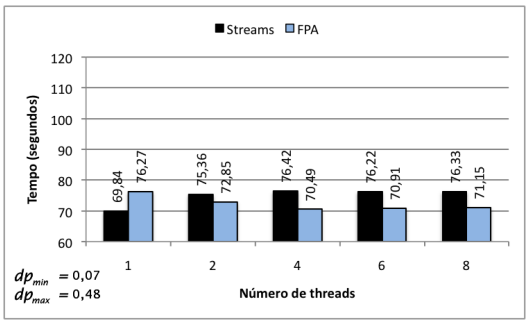
**Figura 4.25:** Pseudocódigo da leitura utilizando mapeamento dinâmico do arquivo



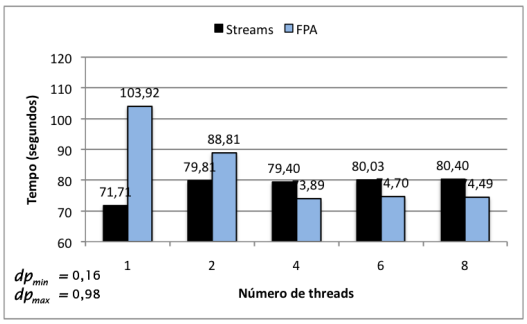

(a) Processamento de 0 iterações



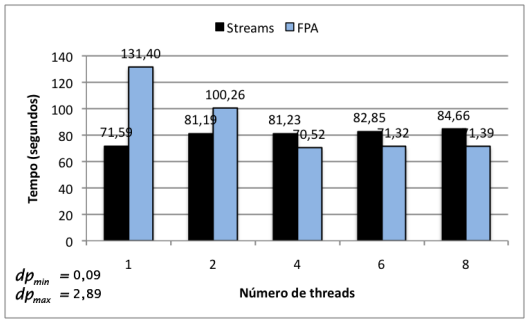

(b) Processamento de 10 iterações



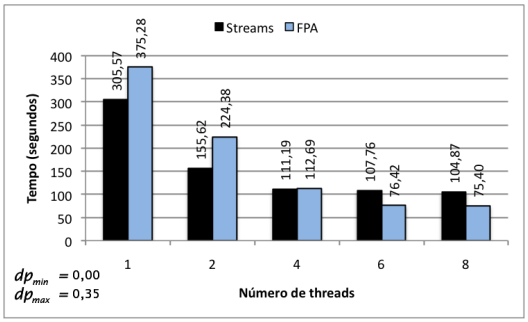

(c) Processamento de 100 iterações



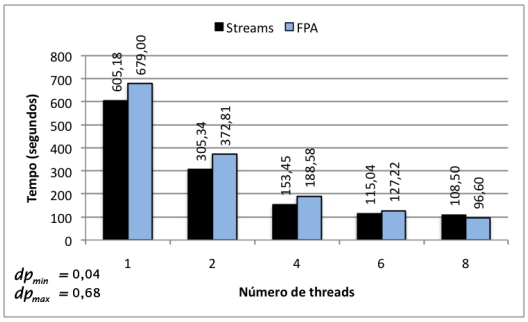

(d) Processamento de 500 iterações



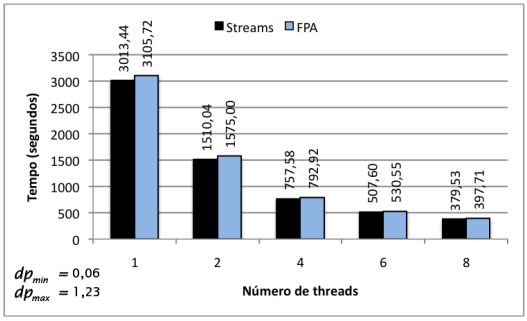

(e) Processamento de 1000 iterações




(f) Processamento de 5000 iterações




(g) Processamento de 10000 iterações




(h) Processamento de 50000 iterações

**Figura 4.26:** Leitura utilizando particionamento dinâmico do arquivo com 4 *threads* e arquivo de 4 GB

O comportamento da aplicação apresentado nos resultados da figura 4.26 é semelhante aos resultados obtidos na leitura com particionamento uniforme (figura 4.5), com algumas pequenas variações.

Nestes testes, 2 *threads* podem compartilhar o mesmo buffer, uma vez que as regiões a serem acessadas correspondem a dados sequencialmente organizados no arquivo. Esta característica, durante operações de leitura, pode levar a uma espera adicional das *threads* pela leitura das informações. Dado que em operações de leitura o acesso ao disco é realizado antes do uso do buffer, é natural que as *threads* tenham que esperar pela informação. Porém, como nos testes realizados o tamanho do buffer é de 32 MB e o tamanho das regiões de leitura do arquivo por cada *thread* é de 16 MB, em cada buffer são armazenados os dados referentes a 2 regiões do arquivo.

Suponha que a *thread* $t_0$ seja responsável pela leitura da região $a$ do arquivo e a *thread* $t_1$ pela região $b$, que no arquivo é consecutiva à região $a$. Na primeira leitura de $t_0$ todo o buffer será preenchido, com uma leitura de 32 MB, que correspondem às regiões $a$ e $b$. A *thread* $t_0$ terá que aguardar pela leitura de 32 MB, mesmo sendo responsável apenas pelos primeiro 16 MB do buffer (região $a$). Esta espera adicional pelo preenchimento do buffer implica que $t_0$ não poderá realizar o processamento da aplicação neste intervalo de tempo. Como $t_1$ requisitou dados que estão no mesmo buffer em um tempo muito pequeno logo após $t_0$ realizar a operação de leitura (Este tempo entre as leituras é pequeno, dado que o desempilhamento pela escolha da região do arquivo é uma operação rápida), $t_1$ também terá de esperar pela leitura de 32 MB (regiões $a$ e $b$) e não poderá realizar o processamento da aplicação neste intervalo de tempo. Já que o tempo de preenchimento do buffer é sempre praticamente constante[2], o impacto decorrente deste tempo de espera implica em que o desempenho obtido pela FPA comece a ser desvantajoso em aplicações, que efetuem um pouco menos de processamento, do que sem o compartilhamento de buffers. Esta situação pode ser conferida analisando os gráficos de leitura dos testes utilizando arquivo com particionamento dinâmico (figuras 4.26 e 4.27) e a leitura de arquivo com particionamento uniforme, onde não acontece o compartilhamento de buffers (figuras 4.5 e 4.6). Observando tais gráficos é possível ver que o ganho obtido pelo uso da FPA com 2 *threads* acontece com quantidades de processamento de cálculo inferiores a 500 iterações, pois na figura 4.26(d) já existe essa desvantagem. Para 4 *threads*, essa vantagem acontece com processamentos inferiores a 5000 iterações, como pode ser conferido nas figuras 4.26(f) e 4.27. Já para arquivos que não compartilham buffers, a quantidade de processamento de cálculo máxima onde existe ganho de desempenho com com a FPA encontra-se entre 1000 e 5000 iterações para 2 *threads* (gráficos 4.5(e) e 4.5(f)) e entre 5000 e 10000 iterações para 4 *threads* (gráficos 4.5(g), 4.5(h) e 4.6). Os motivos desse

[2]Praticamente constante neste caso indica uma variação de tempo muito baixa, pois o tempo das operações de leitura varia devido ao escalonamento do SO e pela busca dos dados no disco. Porém, como nesta situação o tempo de espera é sempre o tempo de uma leitura de 32 MB, este tempo é considerado como sendo constante.

gradativo aumento do número de *threads* com FPA sem desempenho conforme a quantidade de processamento de cálculo da aplicação é resultante do ganho do mapeamento de buffers sobre a concorrência ao disco, como já discutido na seção 4.2.2.

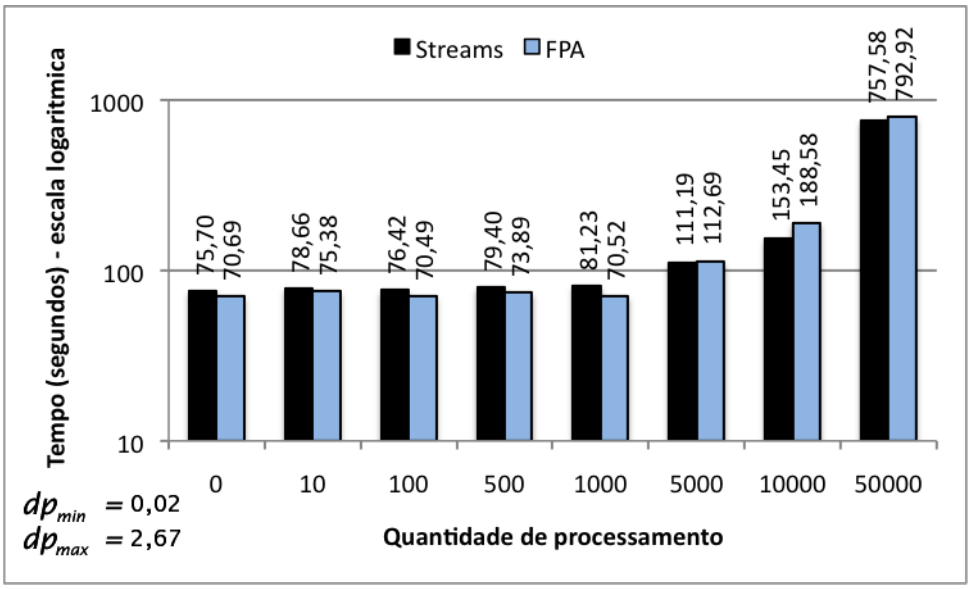
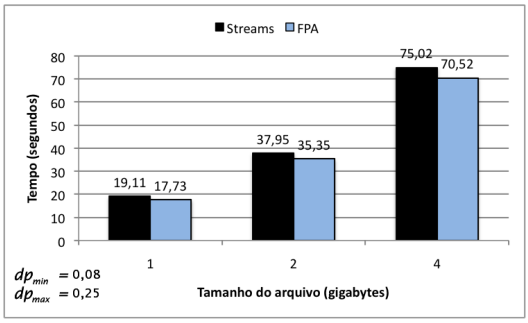
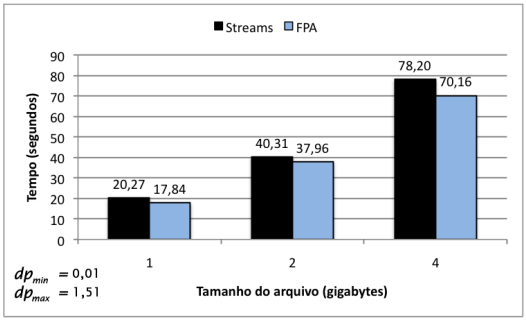


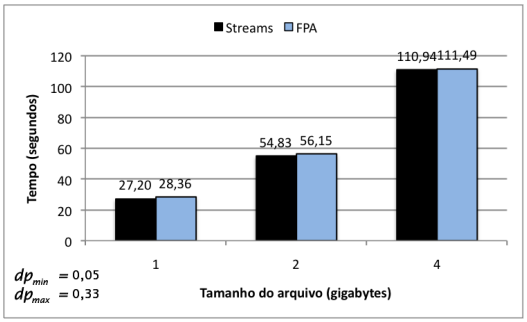

**Figura 4.27:** Leitura utilizando arquivos de 4 GB e 4 *threads*



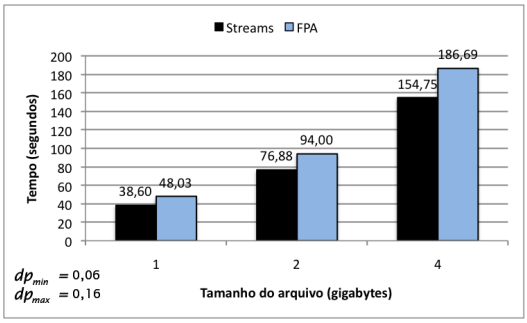

(a) Processamento de 0 iterações




(b) Processamento de 1000 iterações




(c) Processamento de 5000 iterações




(d) Processamento de 10000 iterações

**Figura 4.28:** Leitura utilizando 4 *threads*

Para uma melhor análise da espera pelo preenchimento completo do buffer foi realizado um teste adicional envolvendo uma variação nos parâmetros de testes. O

novo teste consiste em utilizar 16 MB como tamanho do buffer da FPA, e manter as demais configurações sem modificações.

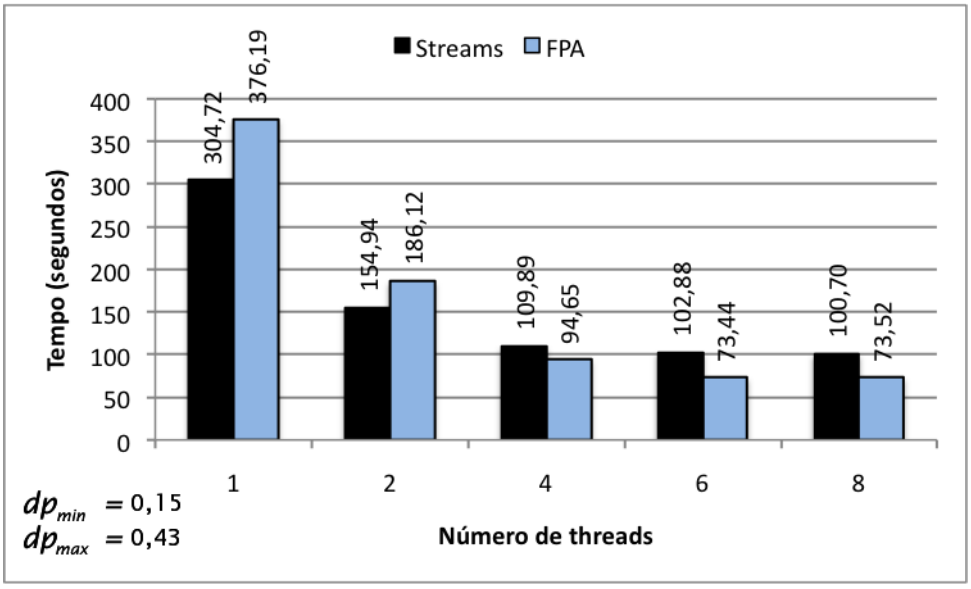



**Figura 4.29:** Leitura utilizando processamento de 5000 iterações, arquivos de 4 GB e buffer de 16 MB

O uso de buffers de memória de tamanho igual ao da região mapeada implica em que cada *thread* tenha seu próprio buffer, descartando o compartilhamento de buffers. Desta forma, cada *thread* terá de aguardar pela leitura de 16 MB do disco e não mais 32 MB, como anteriormente. Isto resulta em que cada *thread* espere apenas pelos seus dados e após isso volte para o processamento da aplicação. Sendo assim, o comportamento deste teste é semelhante ao dos testes com particionamento uniforme do arquivo. Tal resultado pode ser observado comparando os gráficos 4.5(f) e 4.29, onde são praticamente equivalentes e diferem do teste onde existem buffers compartilhados (gráfico 4.26(f)).

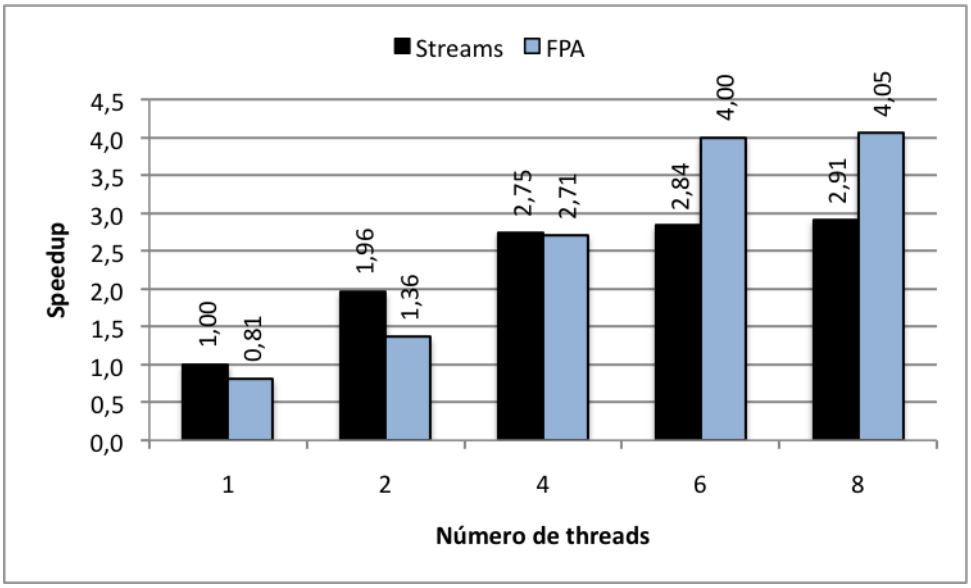



**Figura 4.30:** *Speedup* da leitura utilizando pilha de dados, com arquivos de 4 GB e processamento de 5000 iterações

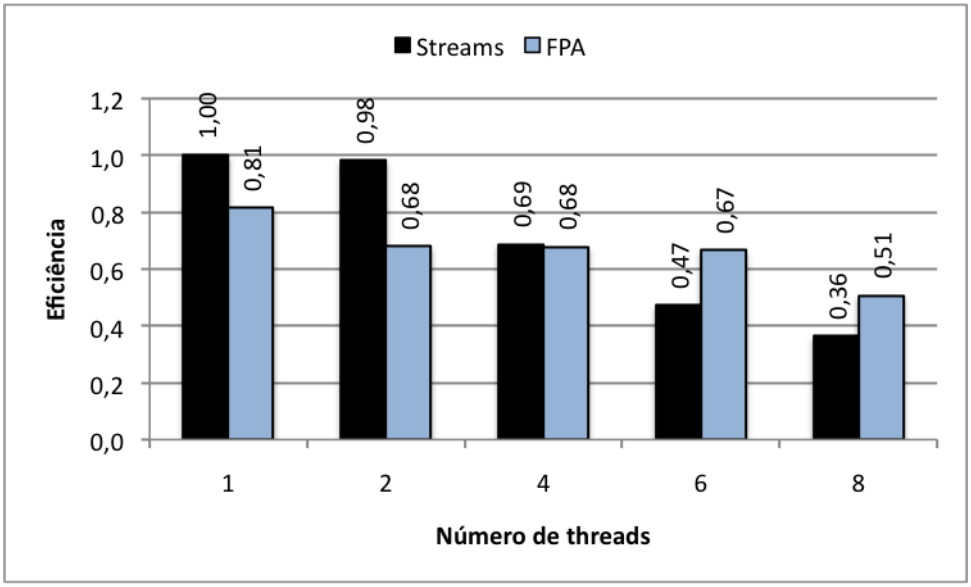
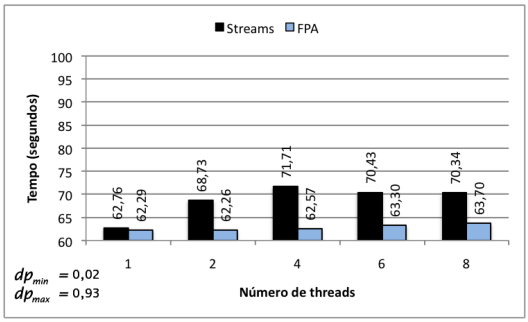
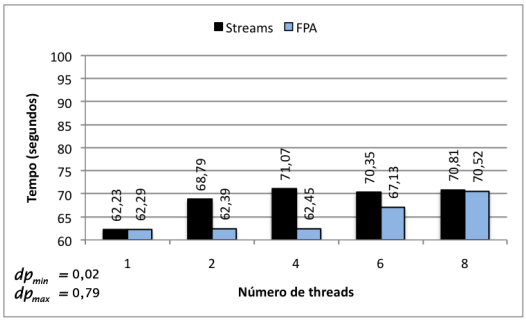



**Figura 4.31:** Eficiência da leitura utilizando pilha de dados, com arquivos de 4 GB e processamento de 5000 iterações

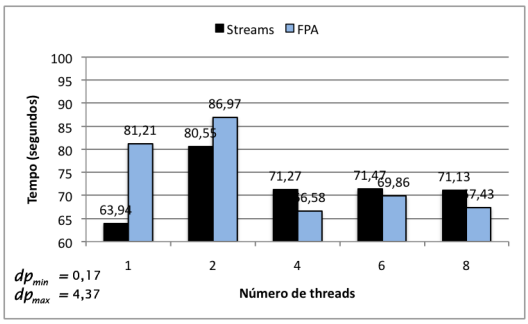
Assim como nos testes de leitura utilizando particionamento uniforme do arquivo, o *speedup* obtido mostra que o uso da biblioteca é vantajoso para aplicações paralelas (figura 4.30). Para o caso de 4 *threads* e processamento de 50000 iterações, a vantagem da FPA aparece com o uso de um número maior que 4 threads, como já analisado anteriormente. Esta mesma situação também é refletida na eficiência (figura 4.31) da aplicação, apresentando comportamento semelhante.



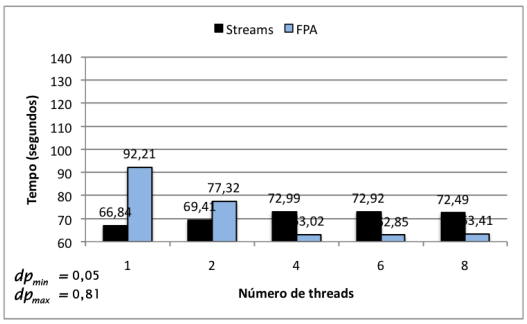

(a) Processamento de 0 iterações



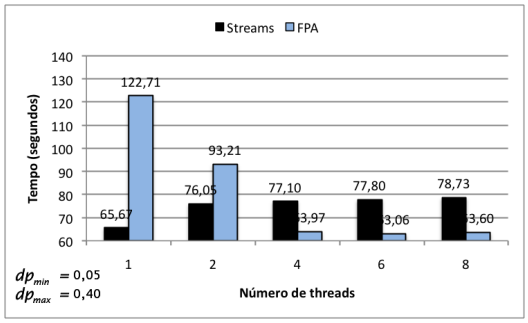

(b) Processamento de 10 iterações



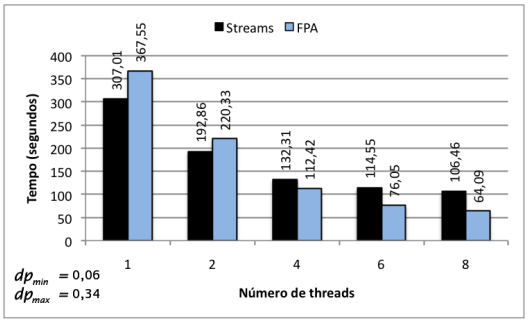

(c) Processamento de 100 iterações



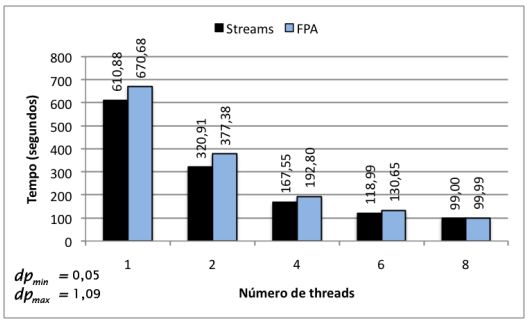

(d) Processamento de 500 iterações



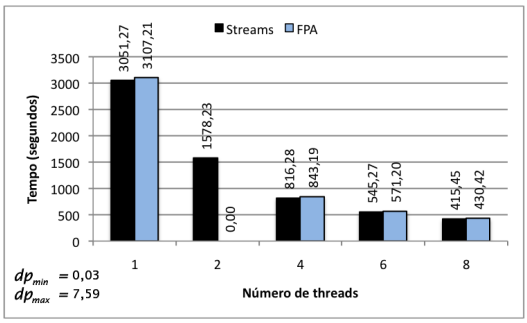

(e) Processamento de 1000 iterações




(f) Processamento de 5000 iterações




(g) Processamento de 10000 iterações




(h) Processamento de 50000 iterações

**Figura 4.32:** Teste na máquina do LAHPC com leitura utilizando mapeamento dinâmico do arquivo com 4 *threads* e arquivos de 4 GB

Os resultados dos testes de leitura com particionamento dinâmico do arquivo mostraram comportamento semelhante aos resultados dos testes executados no cluster OM.

Além de algumas diferenças nos tempos de execução decorrentes das diferentes

configurações do *hardware* das máquinas, os testes diferem em que na máquina do LAHPC um desempenho maior que na OM é alcançado à medida que a quantidade de processamento aumenta.

Para 4 *threads*, a utilização da FPA é vantajosa até um valor de processamento intermediário entre 5000 e 10000 iterações, como pode ser visto nas figuras 4.32(f), 4.32(g) e 4.27, e não mais entre 1000 e 5000 iterações, como nos resultados obtidos no cluster OM. Tal diferença deve-se ao maior *clock* da CPU, fazendo com que a diferença de tempo entre o processamento da aplicação e o acesso a disco seja menor, levando a que o processamento executado possua um menor impacto sobre o resultado final.

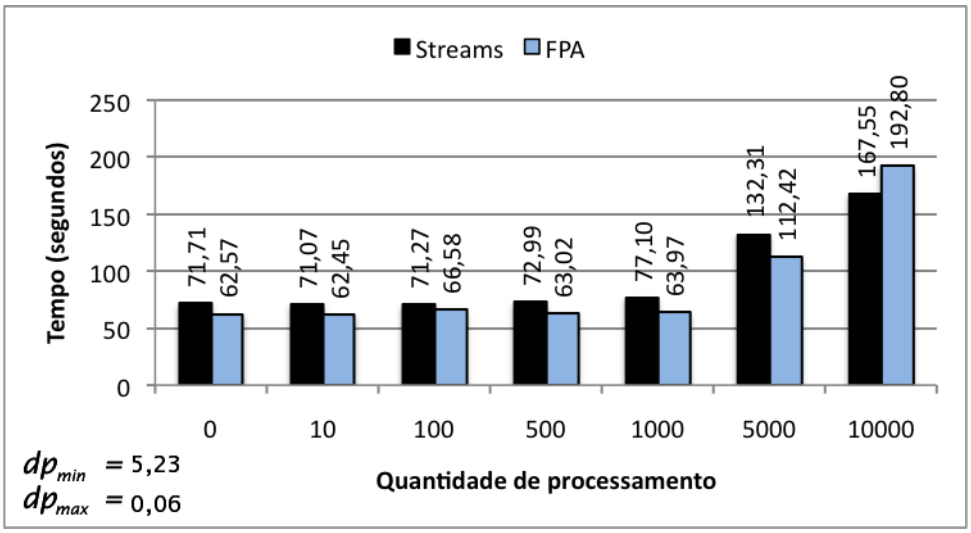
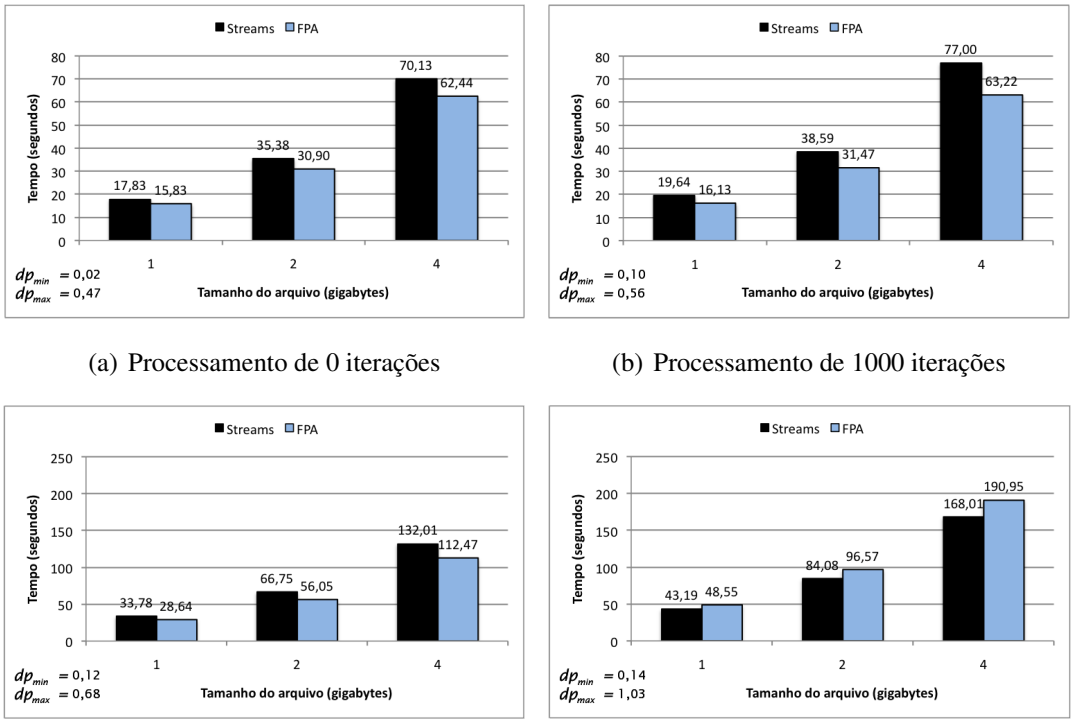



**Figura 4.33:** Teste na máquina do LAHPC com leitura utilizando arquivos de 4 GB e 4 *threads*




(a) Processamento de 0 iterações

(b) Processamento de 1000 iterações

(c) Processamento de 5000 iterações

(d) Processamento de 10000 iterações

**Figura 4.34:** Testes na máquina do LAHPC com leitura utilizando 4 *threads*

De acordo com a figura 4.34, é possível visualizar que o desempenho da FPA novamente independe do tamanho do arquivo. Isto deve-se a que arquivos maiores apenas utilizam quantidades de dados maiores, porém como o acesso é realizado da mesma forma, o ganho de desempenho permanece o mesmo para diferentes tamanhos de arquivo.

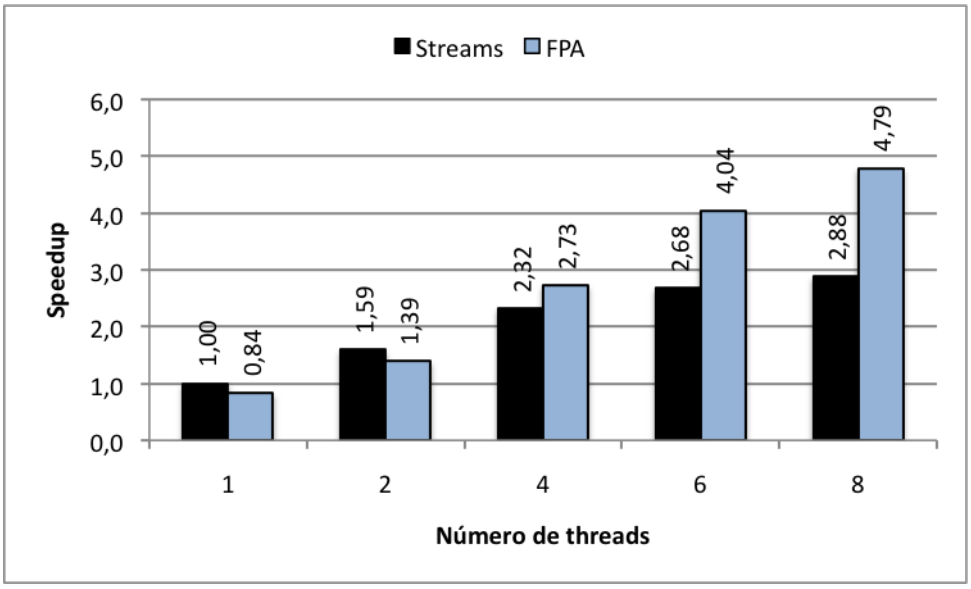



**Figura 4.35:** *Speedup* da leitura utilizando particionamento dinâmico de arquivos na máquina do LAHPC, com arquivos de 4 GB e processamento de 5000 iterações

Os fatores discutidos anteriormente sobre a quantidade de processamento necessária à aplicação para que a FPA obtenha desempenho tem seu resultado refletido no *speedup* e eficiência (figuras 4.35 e 4.36), onde para aplicações envolvendo paralelismo o uso da FPA torna-se vantajoso.

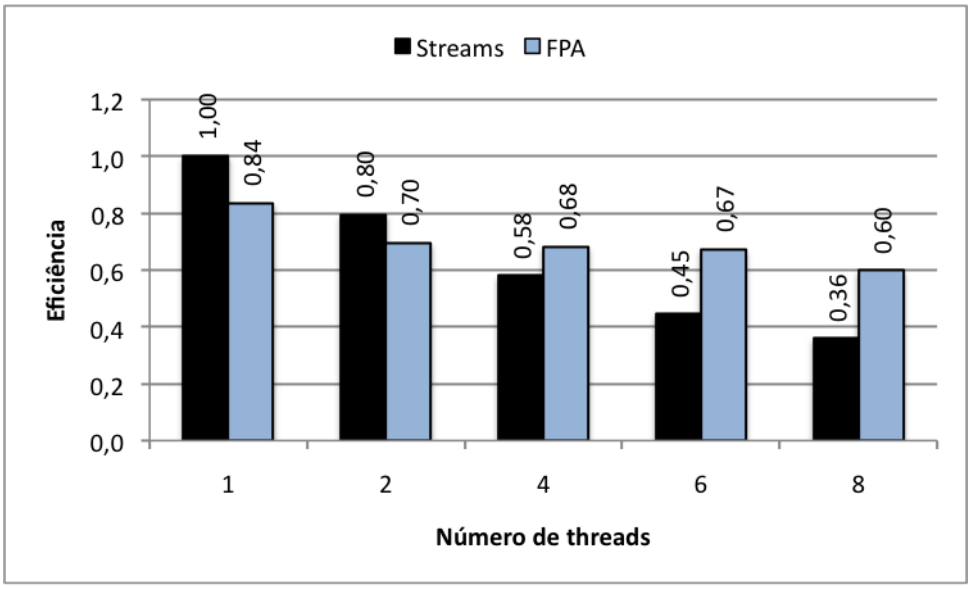



**Figura 4.36:** Eficiência da leitura utilizando mapeamento dinâmico do arquivo na máquina do LAHPC, com arquivos de 4 GB e processamento de 5000 iterações

### Escrita

Em situações de escrita o acesso ao disco acontece após o uso do buffer, fazendo com que o ganho de desempenho da FPA ocorra inclusive em aplicações com uma única *thread*, como pode ser visto na figura 4.38. Tal vantagem é decorrente da menor sobrecarga ao SO por utilizar um menor número de chamadas de sistema, pois o buffer é enviado por completo ao disco através de uma única operação de escrita. O pseudocódigo de como o teste é realizado pode ser visto na figura 4.37.

```
1  contador_regioes=0;
2  numero_regioes = tamanho total do arquivo / 16 MB;
3  INÍCIO DA SEÇÃO PARALELA OPENMP
4          arquivo = open(arquivo_entrada);
5          regiao=0;
6          while( 1 )
7          {
8                  //início da seção crítica
9                  regiao = contador_regioes;
10                 contador_regioes++;
11                 //fim da seção crítica
12                 if( região >= numero_regioes )
13                         break;
14                 seek(arquivo, regiao * 16 MB );
15                 for( i=0; i < 16 * 1024; i++ )
16                 {
17                         //Realiza os cálculos
18                         calcula(processamento);
19                         //Escrita de 1 KB
20                         write(processamento);
21                 }
22         }
23 FIM DA SEÇÃO PARALELA OPENMP
```

**Figura 4.37:** Pseudocódigo da escrita utilizando mapeamento dinâmico do arquivo

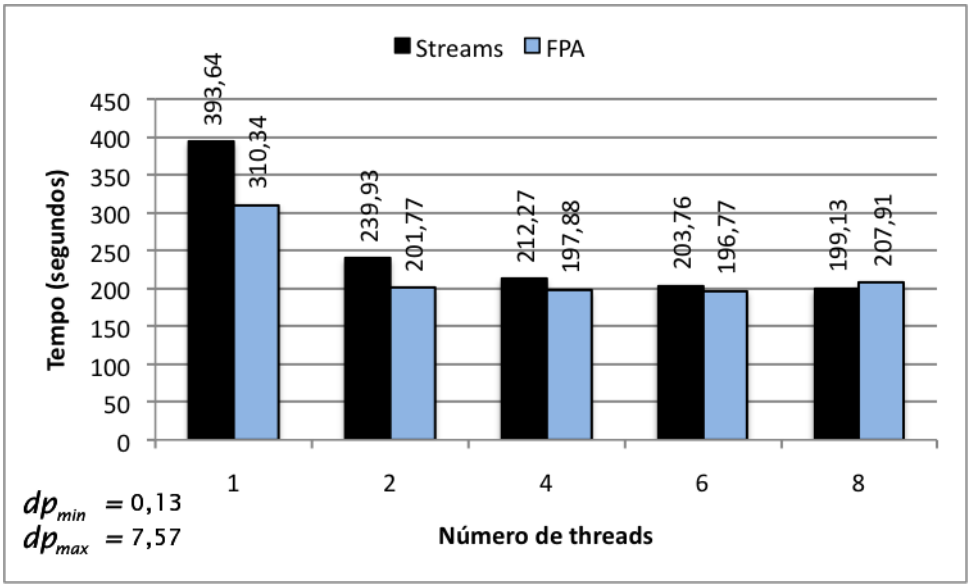



**Figura 4.38:** Escrita utilizando arquivo de 4 GB e processamento de cálculo de 5000 iterações

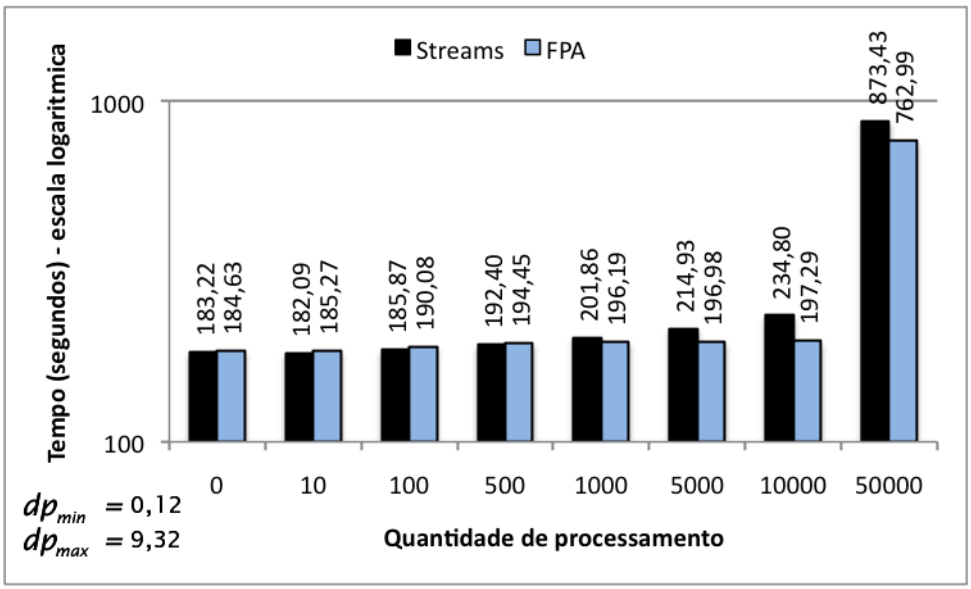



**Figura 4.39:** Escrita utilizando arquivo de 4 GB e 4 *threads*

Analisando a figura 4.39, é possível visualizar que a FPA possui ganho de desempenho em relação aos *streams* em situações onde existe alto processamento de cálculo na aplicação e o contrário acontece nas situações onde não existe processamento. Esta situação decorre do custo do gerenciamento dos buffers e o envio dos dados armazenados ao final da aplicação, e tal custo deve ser compensado pela sobreposição entre o processamento da aplicação e o envio dos dados ao disco. O custo de gerenciamento de buffers é dado pelo tempo gasto em alocações de memória e a movimentação dos buffers nas listas de buffers, afinal as listas na FPA são seções críticas, e alterações nestas devem ser serializadas. Já o tempo gasto pelo envio de dados ao disco no final da aplicação deve-se a uma espera adicional pelo envio dos dados, até que os 128 MB

sejam enviados ao disco. No caso dos testes realizados, esta vantagem inicia-se com uma quantidade de processamento de cálculo entre 500 e 1000 iterações, como visto na figura 4.39. Neste ponto o desempenho da aplicação com FPA começa a tornar-se vantajoso sobre a aplicação utilizando *streams*, devido a que o ganho de desempenho obtido pela sobreposição de processamento e envio de dados começa a compensar o tempo gasto no gerenciamento dos buffers.

Nas situações onde a quantidade de processamento de cálculo é pequeno, para os *streams* haverá sobreposição de processamento e acesso ao disco, mas também haverá períodos de espera devido à concorrência entre as *threads* e o acesso ao disco para gravação dos dados quando o buffer interno do *stream* de cada *thread* estiver cheio. Como no uso de *streams* o SO utiliza buffers para otimização no acesso, o tempo gasto nas operações de escrita é pequeno e acontecem gravações de dados ao disco durante a execução da aplicação e também ao final da mesma.

Para a FPA a execução da aplicação com pouco processamento ocorre de forma rápida, devido a que os dados das operações de escrita são armazenados em buffers. Porém, como o a execução é mais rápida que o envio de dados ao disco a quantidade de buffers alocados chega ao seu máximo de memória, limitado pela quantidade $LM$. Com isso a aplicação terá de aguardar que buffers sejam liberados para que outros novos sejam criados e possa continuar com sua execução. Desta forma, existirá espera no decorrer da aplicação e ao final da execução haverão 2 GB de dados a serem enviados para o disco. Sendo assim, quando o processamento na aplicação é pequeno o uso dos *streams* torna-se uma técnica vantajosa.

Quando existe uma grande quantidade de processamento de cálculo existe um ganho de desempenho com o uso da FPA, como pode ser visto na figura 4.39. A FPA realiza a sobreposição entre o processamento e a gravação de dados no disco durante a execução da aplicação, e seu tempo total de execução é referente ao tempo de processamento dos cálculos em cada *thread* e a gravação dos buffers no final da aplicação. Estes últimos buffers são aqueles mantidos em memória, pois a quantidade total de memória ocupada por estes é inferior ao limite de memória $LA$, indicando que durante a execução da aplicação não devem ser enviados ao disco. Como nos testes realizados o valor de $LA$ é de 128 MB, isto indica que ao final da execução da aplicação restam 4 buffers a serem gravados no disco. Para os *streams* a execução de cada aplicação refere-se ao processamento dos cálculos da aplicação e as escritas no disco, que ocorre de forma intercalada. O armazamento temporário dos dados pelo SO possibilita que o custo destas operações de escrita seja reduzido, mas não eliminado. Ao final da execução ainda serão enviados dados que resultam dos buffers do *stream* de cada *thread* e do SO, porém a espera pela escrita no decorrer da execução faz com que exista uma

perda de desempenho no uso dos *streams*.

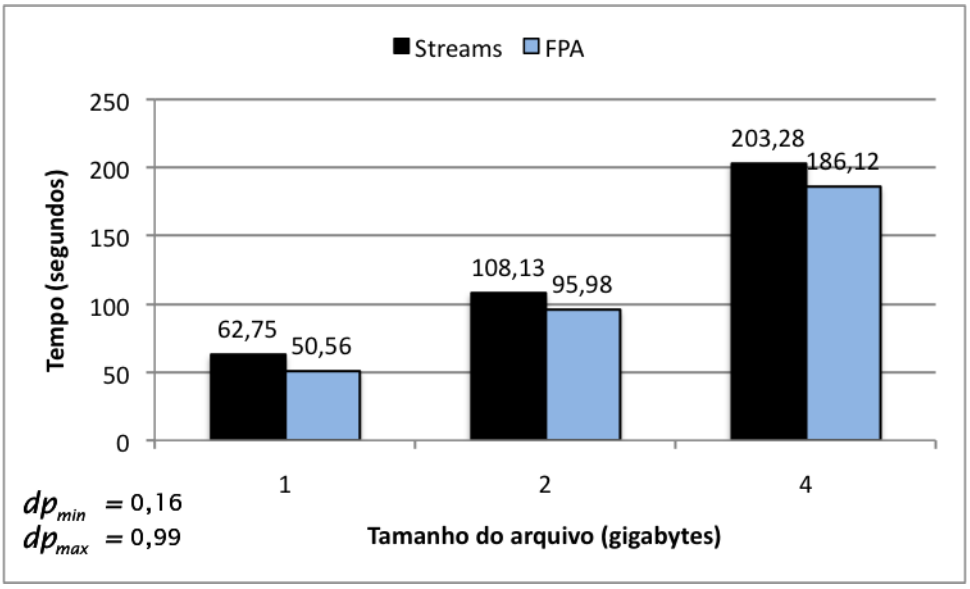



**Figura 4.40:** Escrita utilizando processamento de 5000 iterações e 4 *threads*

Assim como nos testes de leitura, o uso da FPA em operações de escrita não é influenciado pelo tamanho do arquivo. Tal resultado pode ser conferido através do gráfico 4.40.

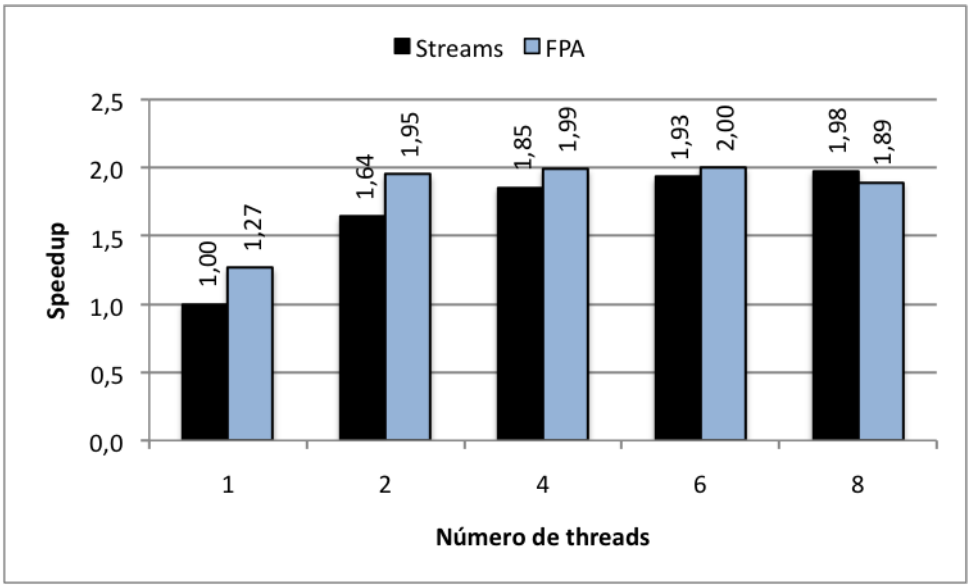



**Figura 4.41:** *Speedup* da escrita utilizando mapeamento dinâmico do arquivo, com arquivos de 4 GB e processamento de 5000 iterações

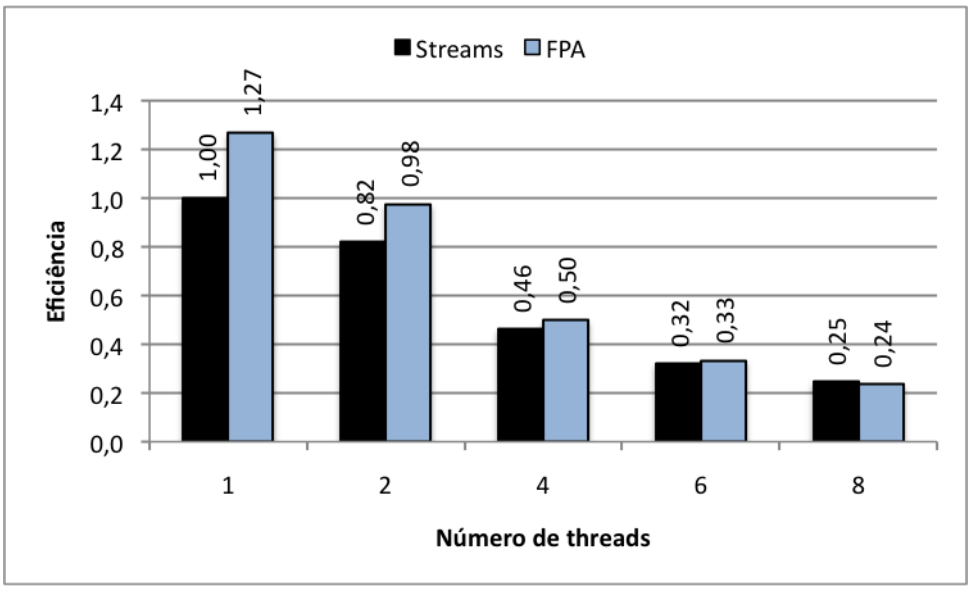



**Figura 4.42:** Eficiência da escrita utilizando pilha de dados, com arquivos de 4 GB e processamento de 5000 iterações

### Leitura e escrita

Através das figuras 4.44, 4.45 e 4.46 é possível analisar o comportamento da FPA com ambas operações de leitura e escrita, atuando na mesma aplicação. O método como estes testes são realizados pode ser visto no pseudocódigo apresentado na figura 4.43. O objetivo do teste é comparar o comportamento e o ganho de desempenho da FPA em relação aos *streams*.

```
1  contador_regioes=0;
2  numero_regioes = tamanho total do arquivo / 16 MB;
3  INÍCIO DA SEÇÃO PARALELA OPENMP
4          arquivo = open(arquivo_entrada);
5          regiao=0;
6          while( 1 )
7          {
8                  //início da seção crítica
9                  regiao = contador_regioes;
10                 contador_regioes++;
11                 //fim da seção crítica
12                 if( região >= numero_regioes )
13                         break;
14                 seek(arquivo, regiao * 16 MB );
15                 for( i=0; i < 16 * 1024; i++ )
16                 {
17                         //Leitura de 1 KB
18                         read(dados);
19                         //Realiza os cálculos
20                         calculos(processamento);
21                         //Modificação dos dados lidos
22                         altera(dados);
23                         seek( arquivo, -1 KB );
24                         //Escrita dos dados modificados
25                         write(dados);
26                 }
27         }
28 FIM DA SEÇÃO PARALELA OPENMP
```

**Figura 4.43:** Pseudocódigo da leitura e escrita utilizando mapeamento dinâmico do arquivo

Como pode ser visto nos gráficos 4.44 e 4.45 o uso da FPA apresenta ganho de desempenho em relação aos *streams* em todas as situações testadas. Assim como nos testes de leitura, no uso da FPA acontece uma espera maior pelo dado do disco, pois cada *thread* é responsável por 16 MB e é necessário esperar para ler 32 MB. Esta leitura de 32 MB corresponde ao preenchimento do buffer, que possivelmente será compartilhado entre 2 *threads*. Porém nestas situações as operações de escrita possuem

ganho de desempenho, através da vantagem do envio do buffer completo ao disco, pois na FPA as operações de escrita são operações rápidas para a aplicação, afinal as informações são mantidas e memória e posteriormente serão enviadas ao disco. Sendo assim, a espera pelo uso do buffer na operação de leitura é compensada e torna-se uma operação vantajosa. Desta forma, o uso da FPA possui ganho de desempenho em relação aos *streams* para operações que envolvam leitura e escrita, inclusive em operações envolvendo apenas 1 *thread* ou com ausência de processamento na aplicação.

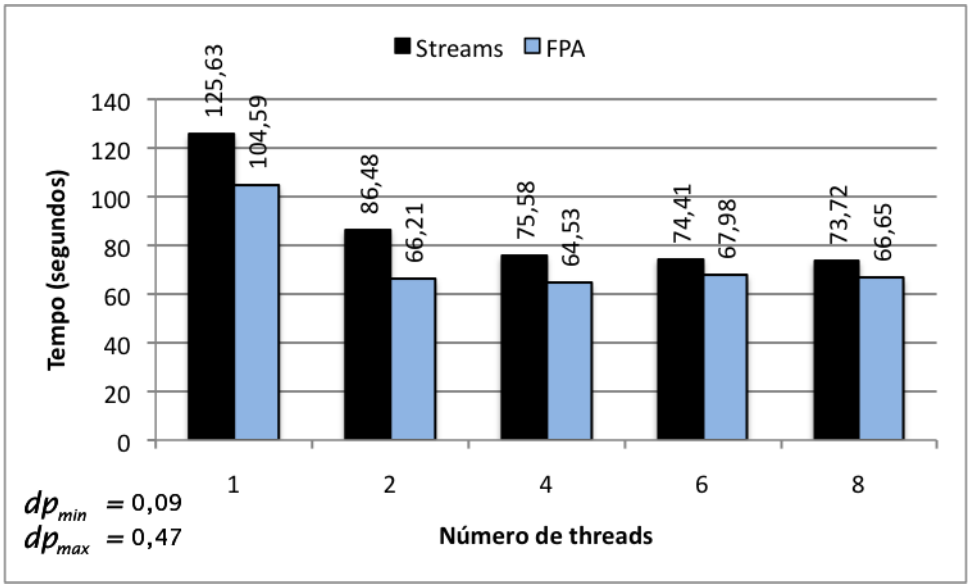



**Figura 4.44:** Leitura e escrita utilizando arquivo de 1 GB e processamento de 5000 iterações

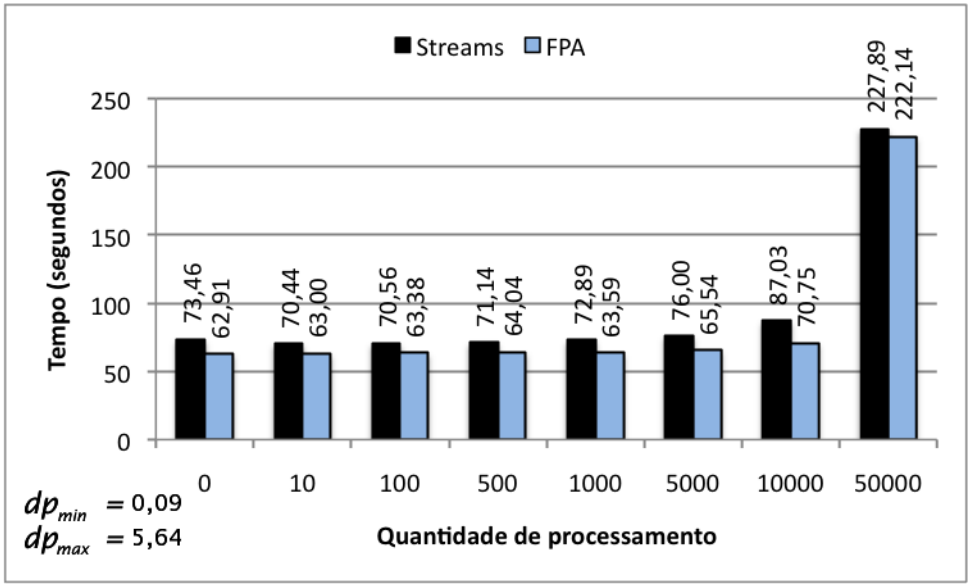



**Figura 4.45:** Leitura e escrita utilizando arquivo de 1 GB e 4 *threads*

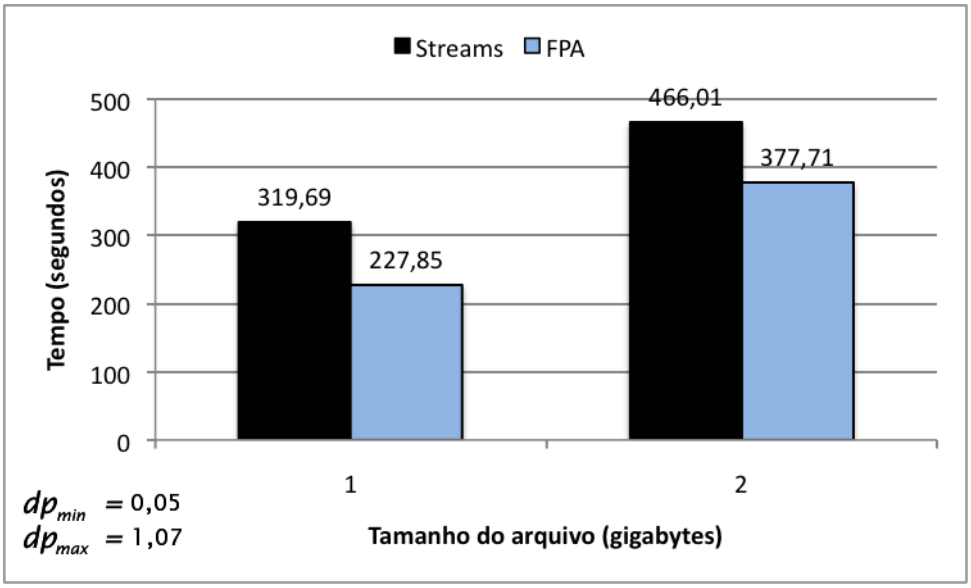



**Figura 4.46:** Leitura e escrita utilizando processamento de 5000 iterações e 4 *threads*

Após a aplicação ler os 128 MB de dados, alterá-los e executar operações de escrita haverá concorrência pelo disco também entre as operações de leitura e escrita, e não apenas pelas leituras das *threads*. A partir desse ponto a *thread* adicional de escrita da FPA começa a enviar os dados ao disco e a liberar buffers em memória. Com isso ao início da aplicação é obtida uma vantagem com o armazenamento em memória, e ao final da execução com a escrita dos dados armazenados em disco, pois não acontece concorrência nesses pontos.

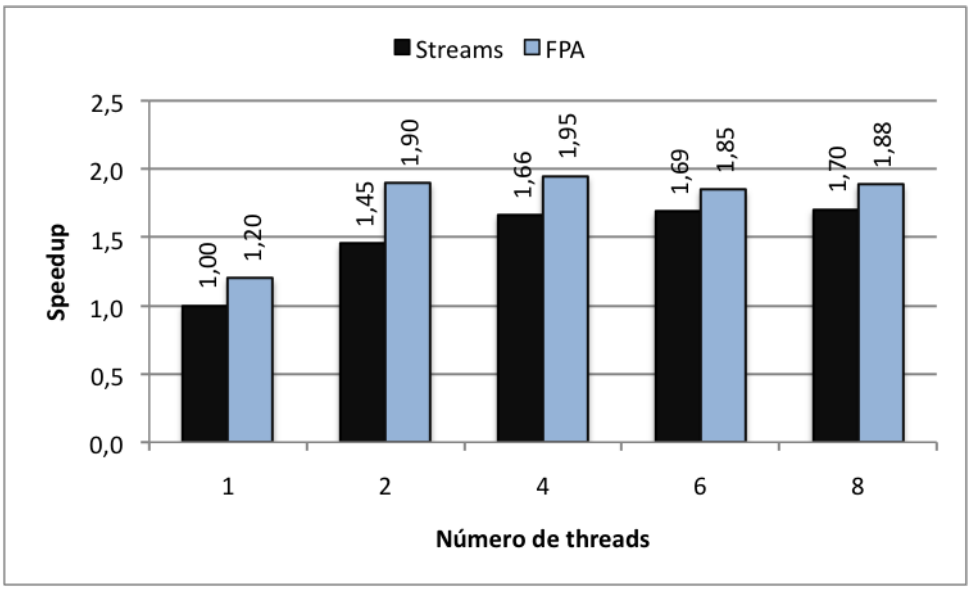



**Figura 4.47:** *Speedup* da leitura e escrita utilizando pilha de dados, com arquivos de 1 GB e processamento de 5000 iterações

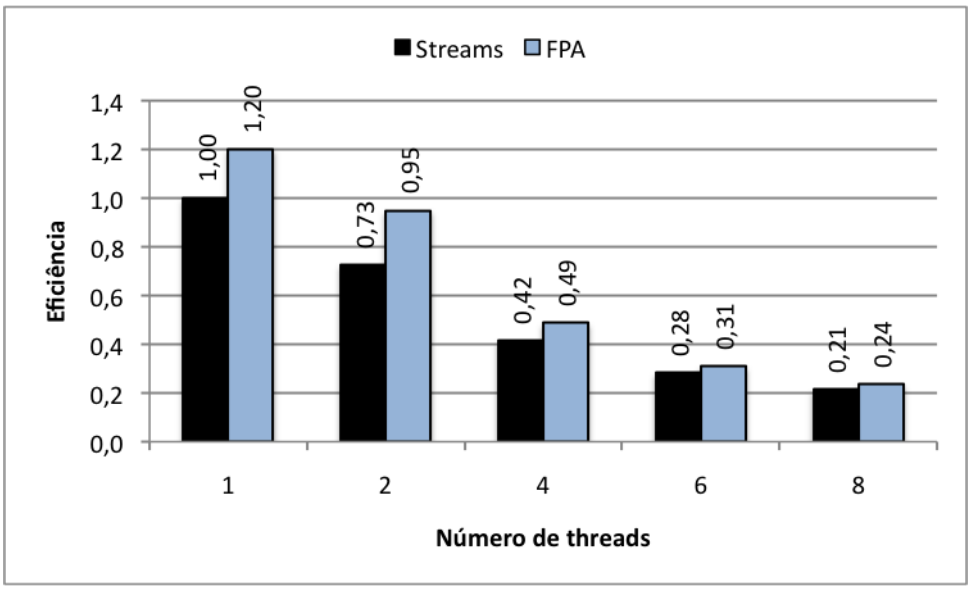



**Figura 4.48:** Eficiência da leitura e escrita utilizando pilha de dados, com arquivos de 1 GB e processamento de 5000 iterações

### 4.2.4 Arquivos distintos por *thread*

O objetivo do teste é analisar o desempenho da FPA utilizando diversos arquivos simultaneamente, onde cada *thread* é responsável pelo acesso a um arquivo diferente. Esta é uma situação de concorrência ao disco onde cada *thread* acessa sequencialmente um arquivo inteiro.

Diferente dos demais testes, onde o arquivo de entrada era dividido pelo número de *threads* da aplicação, cada arquivo aqui é lido/escrito inteiramente por uma única linha de execução. Por exemplo, para um teste utilizando arquivos de 4 GB e 2 *threads* serão lidos/escritos um total 8 GB de dados.

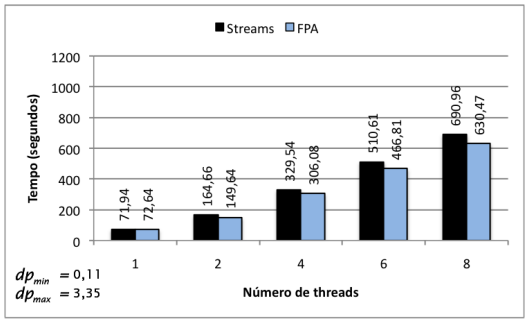
#### Leitura

O objetivo deste teste é observar o comportamento da FPA e dos *streams* em situações com leituras sequenciais de cada *thread*, onde cada uma destas acessa dados de um arquivo distinto. A forma como os testes são realizados pode ser vista no pseudocódigo da figura 4.49.

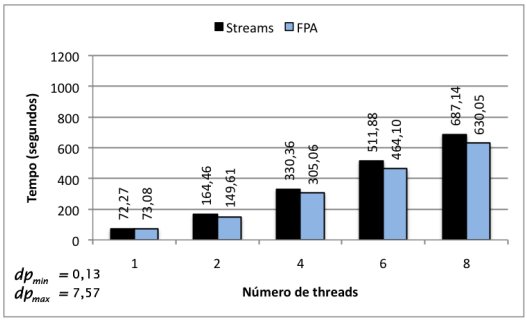
Como o processamento da aplicação é realizado a cada 1 KB de dados lidos e cada *thread* é responsável por um arquivo completo, não existe o paralelismo entre o processamento da aplicação. Desta forma, os resultados destes testes apresentam um comportamento inverso se comparados aos testes com particionamento uniforme e dinâmico do arquivo. À medida que o número de *threads* aumenta o tempo de execução da aplicação também aumenta, e não mais diminui, como nos testes anteriores.

```
1  INÍCIO SEÇÃO PARALELA OPENMP
2          arquivo = open(arquivo_entrada);
3
4          for( i=0; i < tamanho do arquivo / 1 KB; i++ )
5          {
6                  //Leitura de 1 KB
7                  read(dados):
8
9                  //Realiza os cálculos
10                 calculos(processamento);
11         }
12 FIM DA SEÇÃO PARALELA OPENMP
```

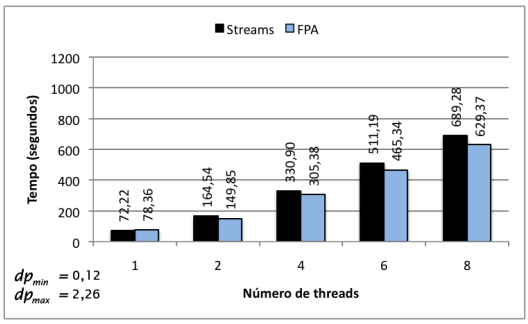
**Figura 4.49:** Pseudocódigo da leitura utilizando arquivos distintos por *thread*



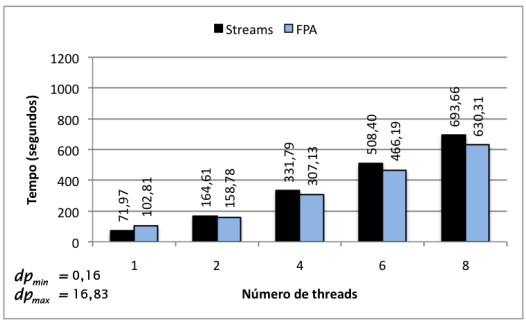

(a) Processamento de 0 iterações



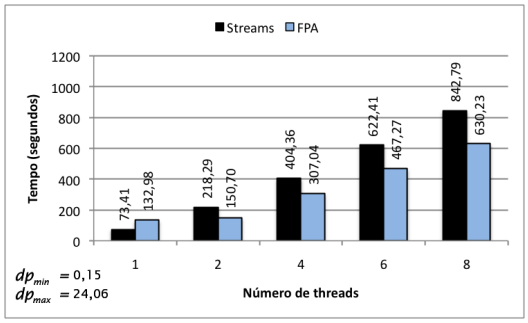

(b) Processamento de 10 iterações



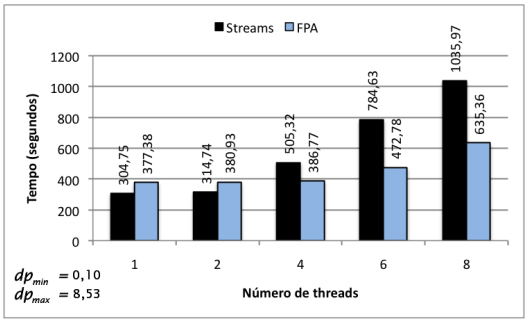

(c) Processamento de 100 iterações



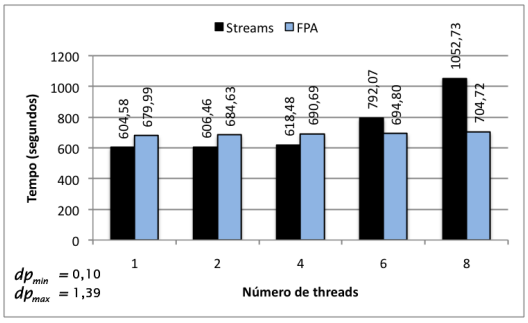

(d) Processamento de 500 iterações



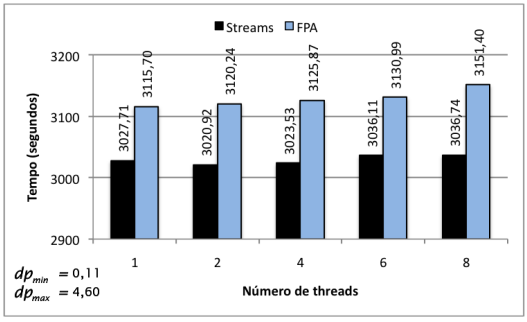

(e) Processamento de 1000 iterações




(f) Processamento de 5000 iterações




(g) Processamento de 10000 iterações




(h) Processamento de 50000 iterações

**Figura 4.50:** Leitura utilizando arquivos individuais por *thread* com 4 GB

O fato de não utilizar o compartilhamento do arquivo entre as *threads*, implica que não existem buffers que serão compartilhados. Sendo assim, os resultados destes testes de leitura possuem comportamento equivalente ao dos testes utilizando particionamento uniforme do arquivo, e as mesmas análises podem ser aplicadas.

Semelhante aos testes da seção 4.2.2, os pontos máximos onde existe ganho de desempenho com FPA em relação a *streams* para 2, 4 e 8 *threads* são os mesmos de acordo com o processamento da aplicação, como pode ser visto nos gráficos 4.50(f), 4.50(g), 4.50(h) e 4.51. O principal diferencial é o tempo de execução da aplicação, devido ao tamanho maior dos arquivos. Porém, como pode ser visto na figura 4.52, o tamanho dos arquivos não afeta no ganho de desempenho da aplicação, mantendo o mesmo comportamento dos testes utilizando particionamento uniforme do arquivo.

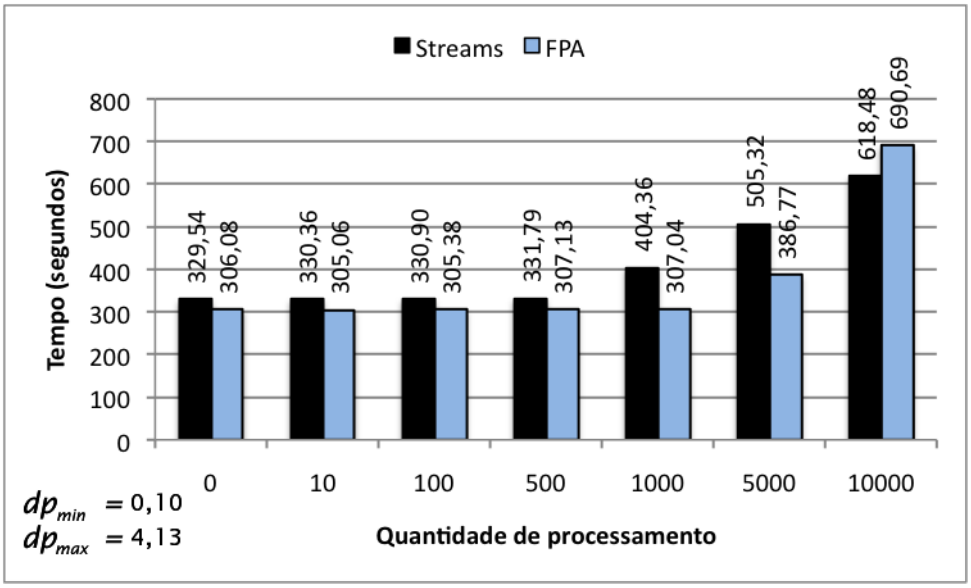
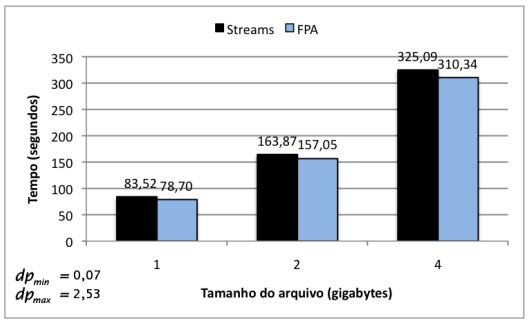
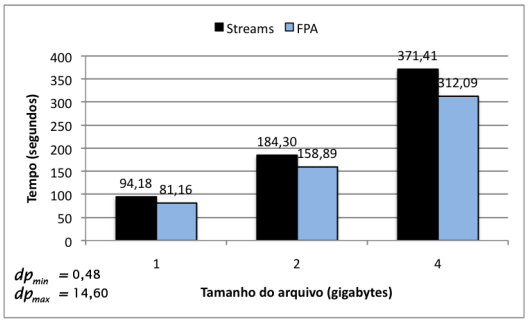


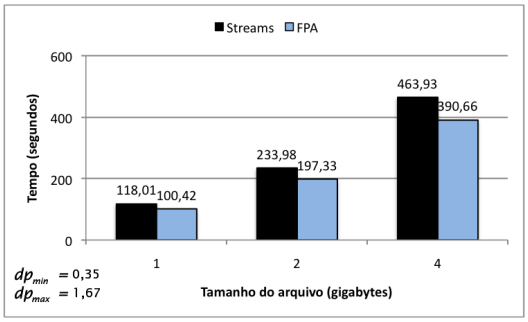

**Figura 4.51:** Leitura de arquivos individuais por *thread* com arquivos de 4 GB e 4 *threads*



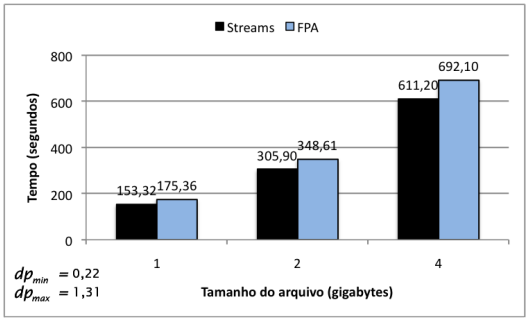

(a) Processamento de 0 iterações




(b) Processamento de 1000 iterações




(c) Processamento de 5000 iterações




(d) Processamento de 10000 iterações

**Figura 4.52:** Leitura utilizando 4 *threads*

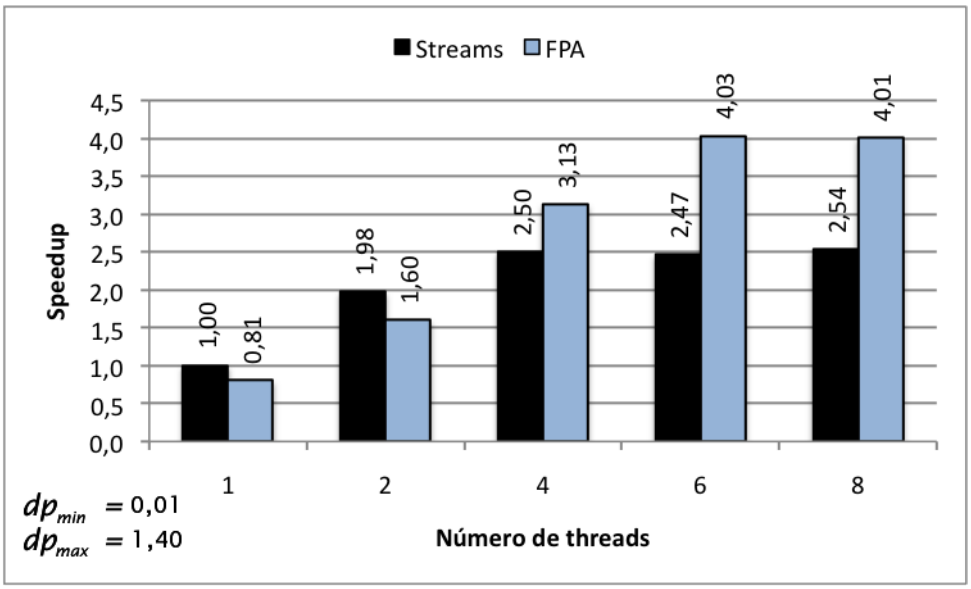



**Figura 4.53:** *Speedup* da leitura utilizando arquivos individuais por *thread*, obtido através de execuções com arquivos de 1 GB e processamento de 5000 iterações

O *speedup* visualizado na figura 4.53 é calculado pela divisão do tempo de execução da versão sequencial pela versão paralela com *streams* e FPA, sendo que na versão sequencial é realizada a leitura sequencial de todos os arquivos lidos pelas *threads* da versão paralela, ou seja, o cálculo do *speedup* para 4 *threads*, cada uma lendo 1 arquivo de 1 GB, e considerando o tempo de execução com leitura sequencial dos mesmos 4 arquivos de 1 GB.

Para um número de *threads* maior que 2 há um ganho de desempenho da FPA em relação aos *streams*, como pode ser visto no gráfico 4.53. Tal desempenho acontece neste ponto devido ao ganho obtido com a sobreposição entre o processamento e a leitura de dados dados. Já para as execuções com um número de *threads* pequeno, a concorrência ao disco não é grande o suficiente para que exista um ganho de desempenho significativo. Esta mesma situação é novamente refletida na eficiência da aplicação, apresentada na figura 4.54.

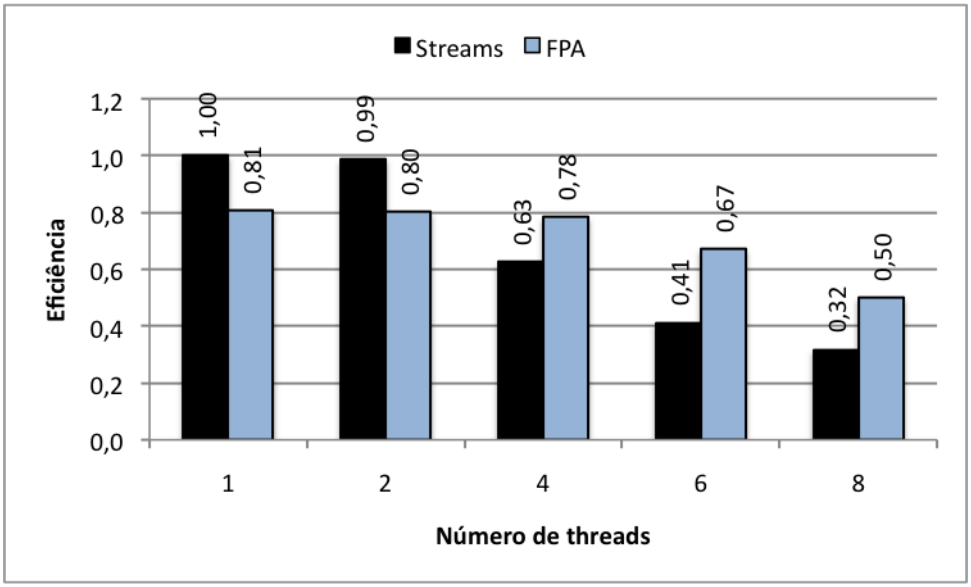



**Figura 4.54:** Eficiência da leitura utilizando arquivos individuais por *thread*, com arquivos de 1 GB e processamento de 5000 iterações

### Escrita

Para os testes utilizando escrita de arquivos individuais por *thread*, cada linha de execução abre o seu próprio arquivo e inicia a escrita de dados. O objetivo do teste é comparar o comportamento da FPA e seu ganho de desempenho relativo aos *streams* nas situações de escrita em arquivos distintos. O pseudocódigo apresentado na figura 4.55 indica como o teste é realizado.

```
1  INÍCIO DA SEÇÃO PARALELA OPENMP
2          arquivo = open(arquivo_entrada);
3
4          for( i=0; i < tamanho do arquivo / 1 KB; i++ )
5          {
6                  //Realiza os cálculos
7                  calcula(processamento);
8
9                  //Escrita de 1 KB
10                 write(dados);
11         }
12 FIM DA SEÇÃO PARALELA OPENMP
```

**Figura 4.55:** Pseudocódigo da escrita utilizando arquivos distintos por *thread*

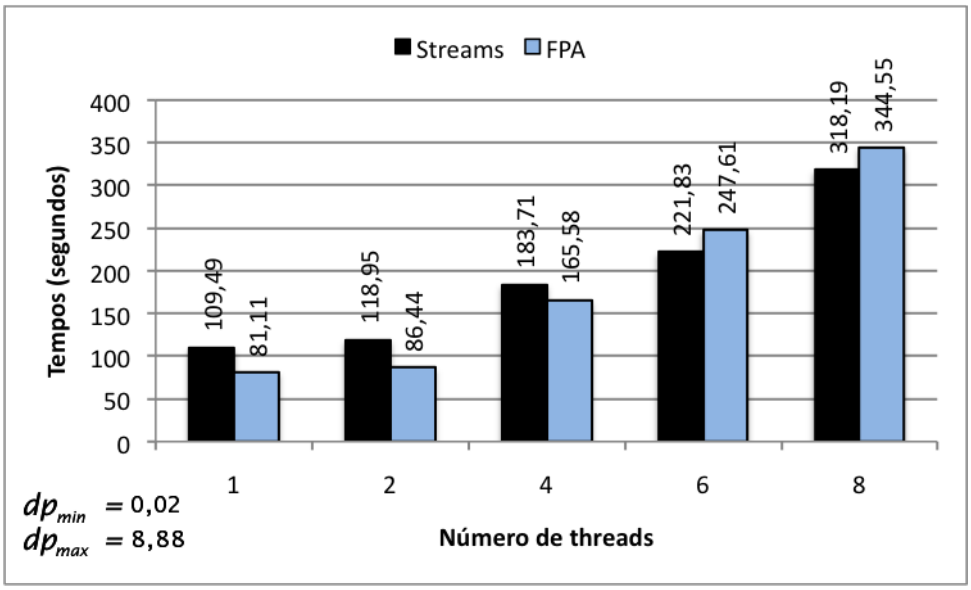



**Figura 4.56:** Escritas utilizando arquivos de 1 GB e processamento de 5000 iterações

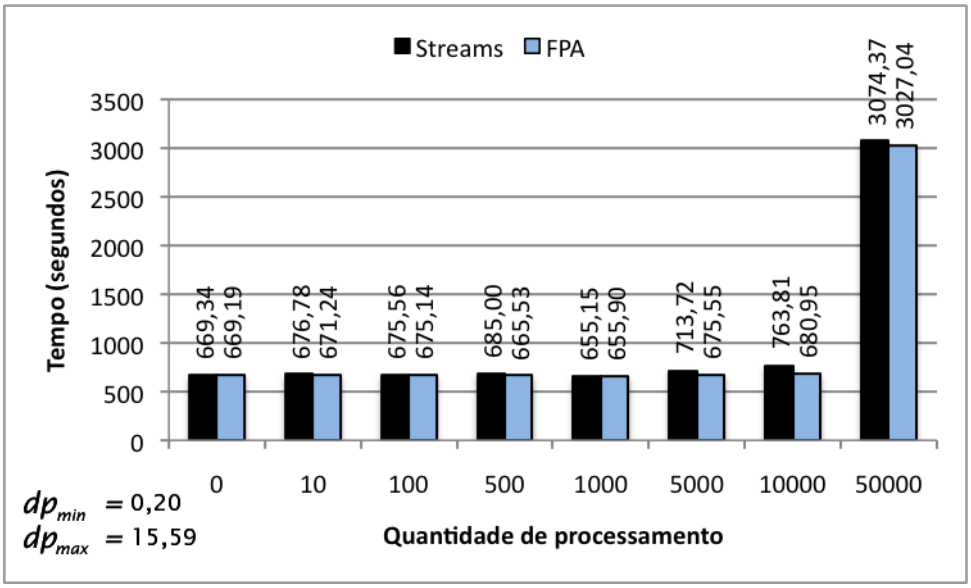



**Figura 4.57:** Escritas utilizando arquivos de 4 GB e 4 *threads*

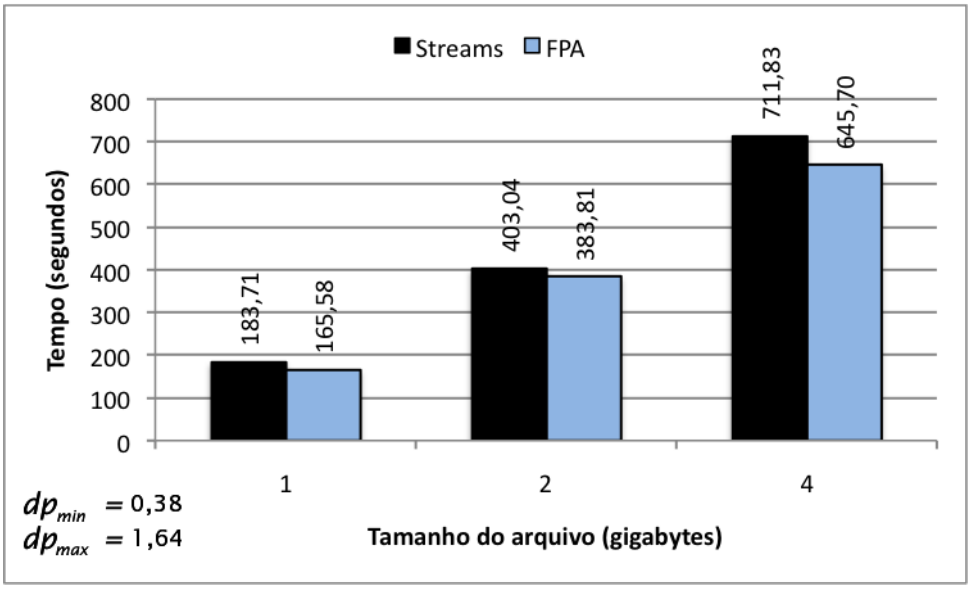



**Figura 4.58:** Escritas utilizando processamento de 5000 iterações e 4 *threads*

Analogamente aos testes de leitura utilizando arquivos distintos por *thread*, o cálculo do *speedup*, apresentado na figura 4.59, para a execução com *n threads* é realizado com referência à aplicação sequencial escrevendo os mesmos *n* arquivos.

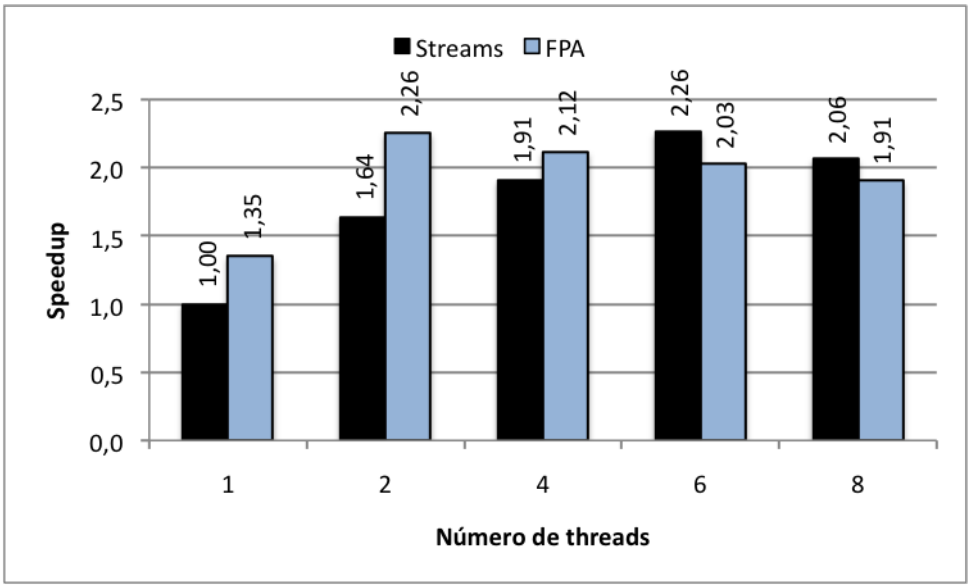



**Figura 4.59:** *Speedup* da escrita utilizando arquivos individuais, com arquivos de 1 GB e processamento de 5000 iterações

Com as escritas de arquivos independentes em cada *thread*, existe ganho de desempenho com a FPA devido à sobreposição do processamento e o envio de dados. Contudo, com o aumento do número de *threads* a velocidade com que os buffers são criados e ocupados aumenta, elevando o número de ocorrências em que o limite máximo *LM* é atingido. Isto aumenta o número de bloqueios de processamento de cálculo, devido à espera do término de escrita de um buffer, liberando espaço em memória.

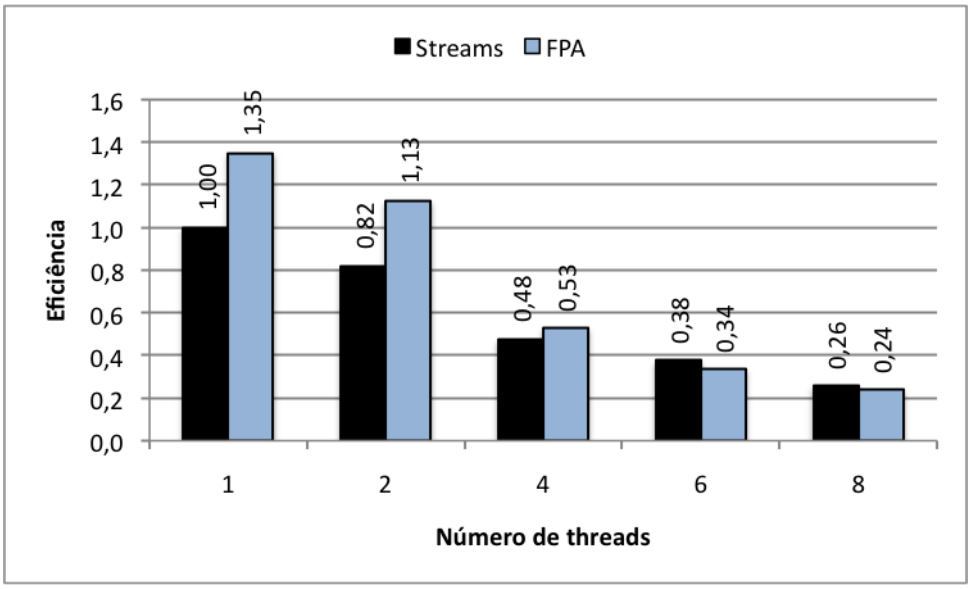



**Figura 4.60:** Eficiência da escrita utilizando arquivos individuais

**Leitura e escrita**

Durante a leitura e escrita utilizando arquivos distintos por *thread*, cada linha de execução é responsável pela abertura seu próprio arquivo. Em cada iteração cada *thread* lê 1 KB de dados, alterando-os na memória e a seguir sincroniza com todas as demais *threads*. Após a sincronização a *thread* $t_0$ soma a cada dado, por ela alterado, nos respectivos dados alterados pelas demais *threads*, sobrescrevendo o resultado no seu arquivo. Sendo assim, a *thread* $t_0$ é a única que realiza a operação de escrita.

```
1  INÍCIO DA SEÇÃO PARALELA OPENMP
2          arquivo = open(arquivo_entrada_thread_id);
3
4          for( i=0; i < tamanho do arquivo / 1 KB; i++ )
5          {
6                  //Leitura de 1 KB, e coloca os dados em
7                  //uma região de memória compartilhada
8                  read(dados);
9
10                 //Realiza os cálculos
11                 calculos(processamento);
12
13                 //Modifica os dados lidos
14                 altera(dados);
15
16                 //Sincroniza as threads
17                 barrier();
18
19                 if( thread_id == 0 )
20                 {
21                         //soma os dados modificados
22                         //de todas as threads
23                         soma(dados);
24
25                         seek( arquivo, -1 KB );
26
27                         //escrita dos dados somados
28                         write(dados);
29                 }
30         }
31 FIM DA SEÇÃO PARALELA OPENMP
```

**Figura 4.61:** Pseudocódigo da leitura e escrita utilizando arquivos distintos por *thread*

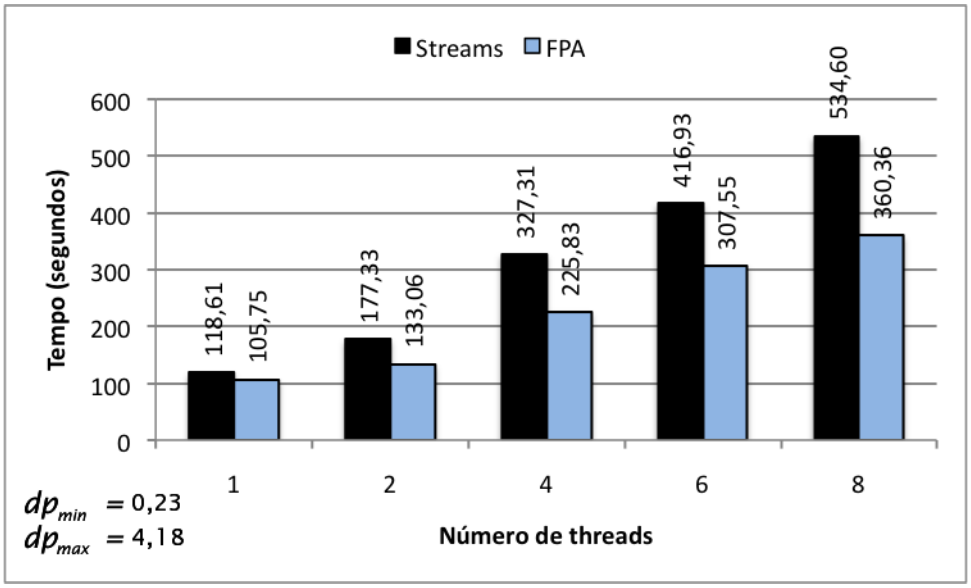



**Figura 4.62:** Leitura e escrita utilizando arquivos de 1 GB e processamento de 5000 iterações

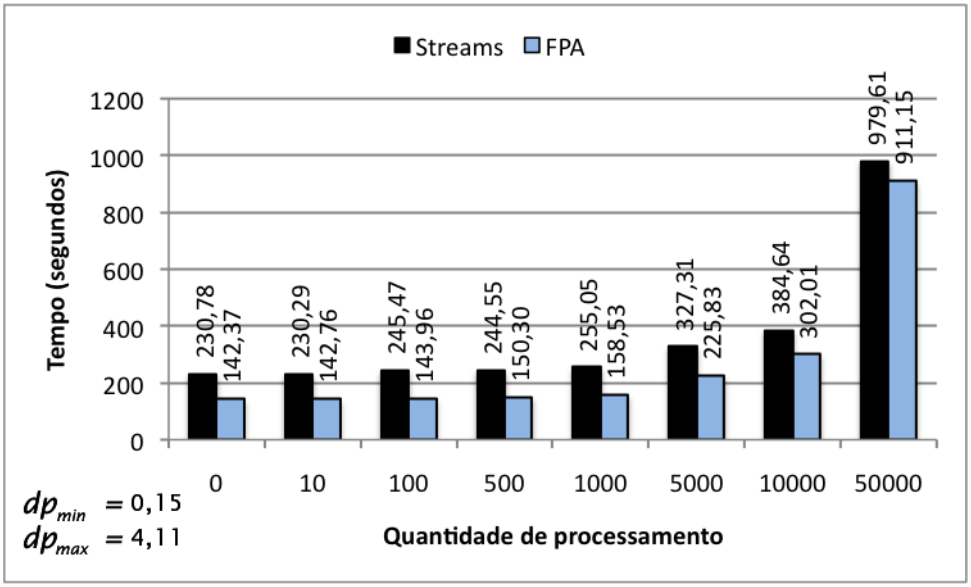



**Figura 4.63:** Leitura e escrita utilizando arquivos de 1 GB e 4 *threads*

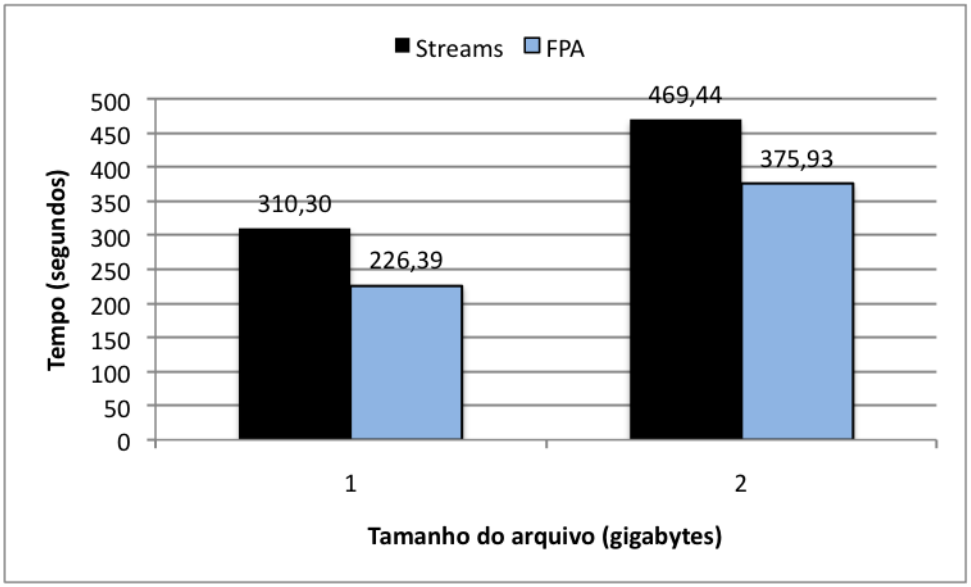



**Figura 4.64:** Leitura e escrita utilizando processamento de 5000 iterações e 4 *threads*

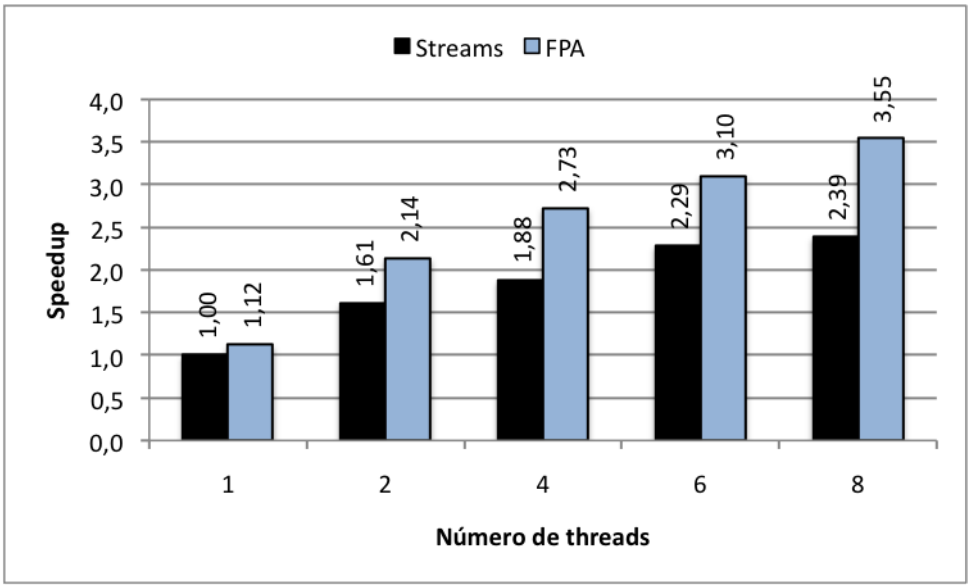



**Figura 4.65:** *Speedup* da leitura e escrita utilizando arquivos individuais, com arquivos de 1 GB e processamento de 5000 iterações

Como pode ser verificado no gráfico do *speedup* na figura 4.65, existe ganho de desempenho na FPA sobre os *streams* para todas as situações testadas. Isto deve-se ao maior tempo de espera para o acesso ao disco com os *streams* devido a concorrência causada pela leitura e escrita, e neste ponto o uso da FPA é vantajoso. Tal vantagem é devido ao armazenamento dos dados nos buffers, pois podem ser enviados ao disco posteriormente e a espera para a escrita será dada pela *thread* adicional da FPA, e não pelas *threads* da aplicação. Este resultado também é refletido na eficiência da aplicação, vista no gráfico 4.66.

O ganho de desempenho da FPA nos testes envolvendo leitura e escrita de arquivos

também pode ser explicado através de uma comparação do gráfico de *speedup* da figura 4.65 com os *speedups* das operações de leitura e escrita, vistos nas figuras 4.53 e 4.59. Nestes é possível observar que o ganho de desempenho da FPA nas operações de leitura acontece com o uso de 4 a 8 *threads*, e com as escritas o ganho acontece com o uso de 1 a 4 *threads* na aplicação. Como neste teste são efetuadas ambas as operações, é obtido vantagem na leitura e escrita, apresentando vantagem em todos os casos testados, como visto no *speedup* da figura 4.65.

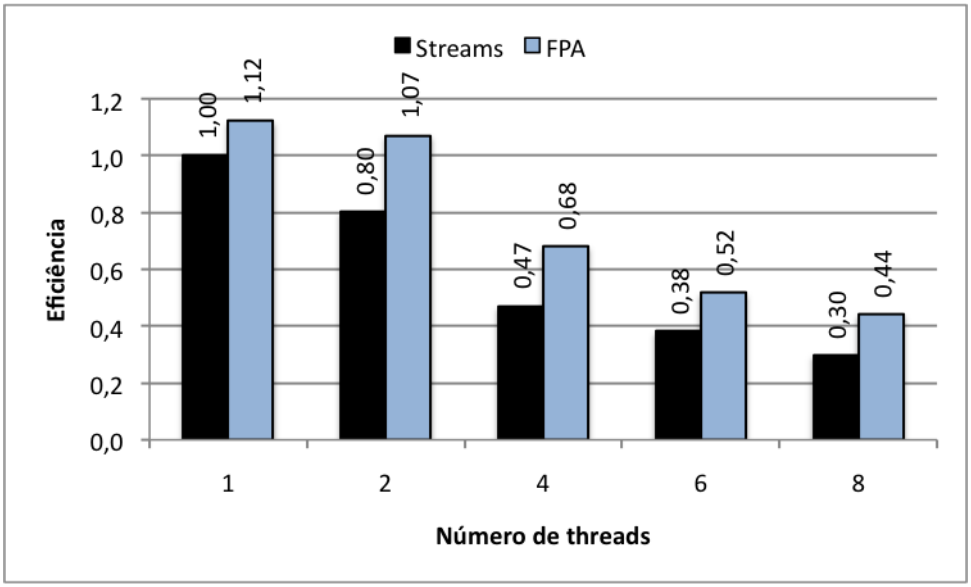



**Figura 4.66:** Eficiência da leitura e escrita utilizando arquivos individuais, com arquivos de 1 GB e processamento de 5000 iterações

### 4.2.5 Diferentes parâmetros

De forma a conhecer o comportamento da FPA sob diversos tamanhos de buffers e diferentes limites de aviso de memória foram realizados testes especificando diferentes valores para estes parâmetros.

O teste realizado consistiu em executar leituras e escritas de dados utilizando o particionamento dinâmico do arquivo, utilizando processamento de 5000 iterações, 4 *threads* e arquivos de 4 GB. Para o particionamento dinâmico, o tamanho de cada região mapeada é de 16 MB, sendo o mesmo valor testado na seção 4.2.3.

O desempenho de cada configuração de buffers e *LA* é expresso através de uma matriz de cores, onde a tonalidade mais escura indica um menor tempo de execução e consequentemente um melhor desempenho. Os resultados dos testes podem ser vistos no gráficos apresentados nas figuras 4.67 e 4.68.

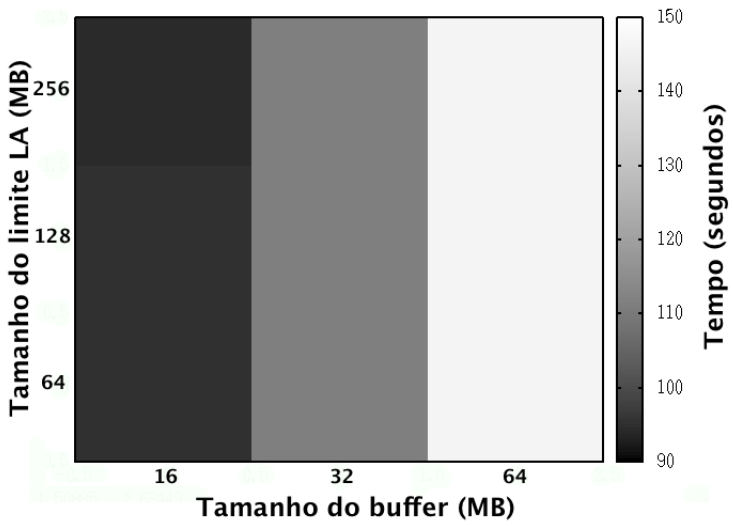



**Figura 4.67:** Leitura utilizando diferentes tamanhos de buffers e diferentes valores para *LA*

Durante as operações de leitura, o tamanho dos buffers exerce influência direta no desempenho final da aplicação, onde na figura 4.67 é apontado como o melhor desempenho a configuração que utiliza buffers de 16. Esta configuração é vantajosa porque no teste realizado o tamanho de cada região mapeada é de 16 MB e assim não existe a espera das *threads* pelo preenchimento completo do buffer. O limite *LA* não exerce grande influência no resultado final, pois durante as operações de leitura a concorrência ao disco é o principal limitante de desempenho, e não as manipulações em memória. Esta situação também ocorre porque não existe o reaproveitamento dos buffers em memória, e com isso o menor número destes já é suficiente para a execução da aplicação.

As esperas pelo preenchimento do buffer implicam em um tempo de execução de 44% para o melhor caso em relação ao pior, mostrando que a diferença de desempenho entre as diferentes configurações é significativa.

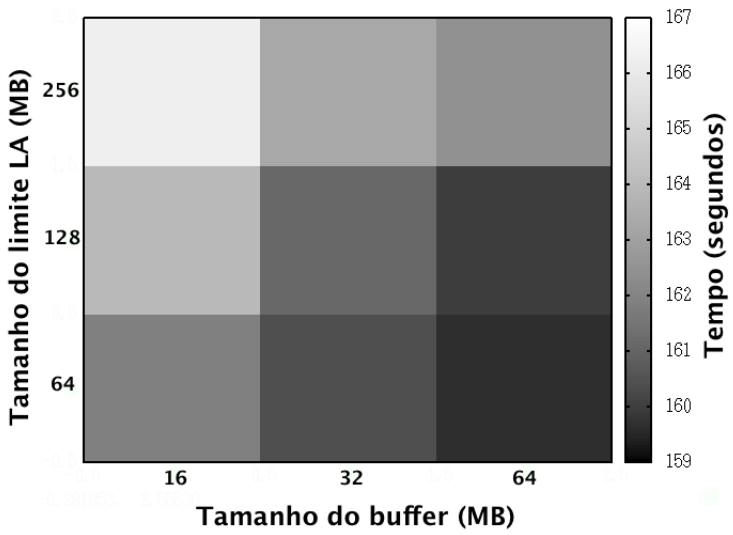



**Figura 4.68:** Escrita utilizando diferentes tamanhos de buffers e diferentes valores para *LA*

Diferente das operações de leitura, o parâmetro *LA* influencia diretamente no resultado final. Quanto menos informações forem deixadas para serem enviadas ao final da execução da aplicação melhor será seu desempenho nesta situação. Analisando o gráfico da figura 4.68 nota-se que o melhor desempenho obtido acontece com 64 MB para *LA*, porque este é o menor valor testado e faz com que ao final da aplicação uma menor quantidade de dados seja necessária enviar ao disco.

Quanto ao tamanho dos buffers, foram testados os valores 16, 32 e 64 MB, onde os resultados indicaram que o maior ganho de desempenho é obtido com o maior buffer, pois os dados são enviados ao disco de uma única vez e não é necessário que sejam feitas diversas operações de manipulação na lista de buffers. De acordo com os resultados obtidos, a melhor configuração nas operações de escrita possui um tempo de execução $4,8\%$ menor que o tempo obtido no pior caso.

Observando os resultados e tomando as configurações utilizadas nos testes deste trabalho, percebe-se que os parâmetros adotados, de 128 MB para *LA* e 32 MB para tamanho de buffers, situa-se em um ponto intermediário entre os melhores desempenhos para operações de leitura e escrita. Desta forma, estes valores podem ser considerados como um bom ponto de referência para execuções de propósito geral, onde não são conhecidas quais as operações acontecem em maior quantidade.

### 4.2.6 *Stream* compartilhado e *stream* privado

Como apontado na seção 2.6.3, o uso dos *streams* pode ser privado ou compartilhado entre as diversas *threads* da aplicação.

A aplicação de diferentes técnicas de armazenamento temporário de dados e acesso a disco faz com que o uso do *stream* privado ou compartilhado apresente diferentes comportamentos perante as diversas condições de concorrência a disco e da relação processamento/envio de dados.

```
1  arquivo = open(arquivo_entrada);
2  contador_regioes=0;
3  numero_regioes = tamanho total do arquivo / 16 MB;
4  INÍCIO DA SEÇÃO PARALELA OPENMP
5          posicao=0;
6          regiao=0;
7          while( 1 )
8          {
9                  //início da seção crítica
10                 regiao = contador_regioes;
11                 contador_regioes++;
12                 //fim da seção crítica
13                 if( região >= numero_regioes )
14                         break;
15                 posicao = regiao * 16 MB;
16                 for( i=0; i < 16 * 1024; i++ )
17                 {
18                         //Realiza os cálculos
19                         calculos(processamento);
20                         lockfile(arquivo);
21                         seek(posicao);
22                         //Escrita de 1 KB
23                         write(arquivo);
24                         unlockfile(arquivo);
25                         posicao += 1 KB;
26                 }
27         }
28 FIM DA SEÇÃO PARALELA
```

**Figura 4.69:** Pseudocódigo da escrita de dados utilizando *stream* compartilhado

O teste é realizado utilizando escritas a disco pelo método de particionamento dinâmico do arquivo, como executado na seção 4.2.3. Porém, o método de escrita utilizando *streams* compartilhados difere dos demais métodos. O pseudocódigo de

como este é realizado é apresentado na figura 4.69.

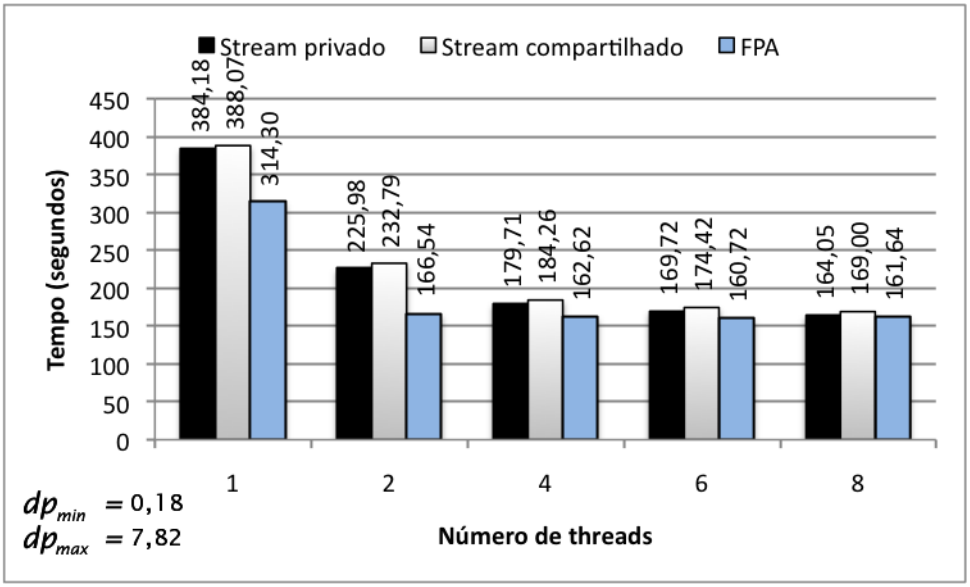



**Figura 4.70:** Comparação entre *stream* privado, *stream* compartilhado e FPA em escritas de arquivos de 4 GB e utilizando processamento de 5000 iterações

A medida que a concorrência ao disco aumenta, o uso do *stream* compartilhado se torna vantajoso, podendo inclusive ter ganho de desempenho em relação à FPA em algumas situações.

O uso simultâneo de múltiplos *streams* privados por múltiplas *threads* gera a concorrência ao disco, a qual aumenta com o acréscimo de *threads* na aplicação. Nas situações durante a execução que esta concorrência ocorre a serialização do processamento, devido à execução exclusiva do acesso ao disco.

Com o *stream* compartilhado, é necessário o uso de um semáforo, garantindo a exclusividade do seu acesso. Isto implica na serialização de todo o processo de escrita, e não apenas no acesso ao disco. Desta forma, quando se tem um certo nível de conrrência, como visto na figura 4.70, o desempenho do *stream* privado é melhor do que com stream compartilhado. Contudo, com o uso dos *streams* compartilhados é obrigatório o uso do semáforo para os casos onde há leitura e escrita feitas por *threads* distintas sobre os mesmos dados nas mesmas posições do arquivo. Isto é necessário para evitar que existam problemas de inconsistência de dados no *buffer* interno do *stream*, como mostrado na seção 2.6.3 na página 35.

Conforme já discutido na seção 4.2.3, a FPA apresentou melhor desempenho em relação ao uso do *stream* privado. Visualiza-se também na figura 4.70 o melhor desempenho da FPA para quaisquer número de *threads*, exceto para o caso de 8 *threads*.

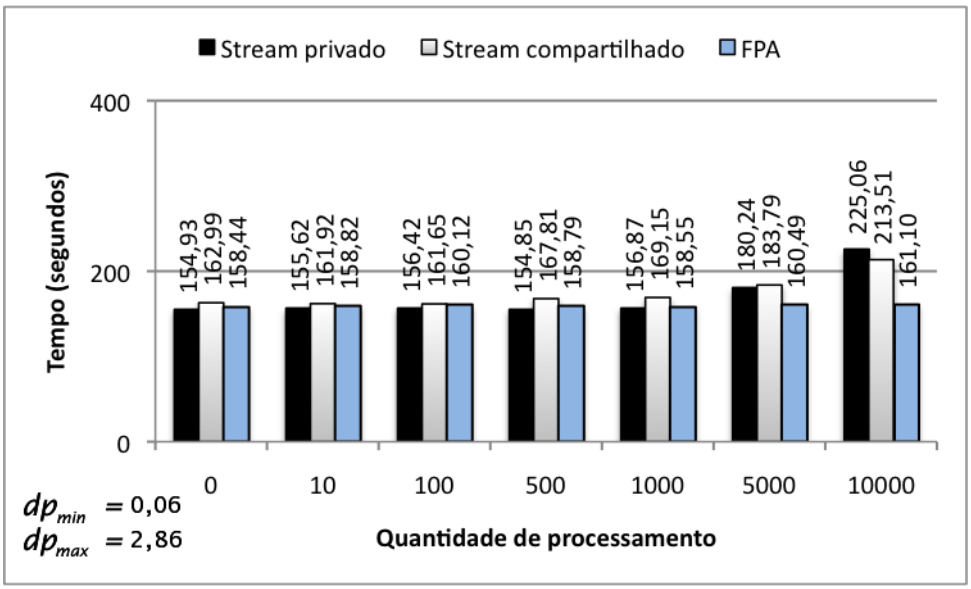



**Figura 4.71:** Comparação entre *stream* privado, *stream* compartilhado e FPA em escritas de arquivos de 4 GB e utilizando 4 *threads*

## 4.3 Busca em arquivo

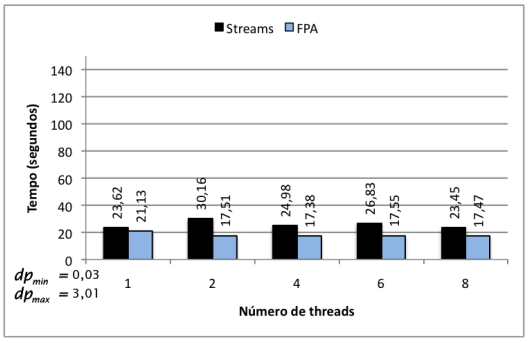
Os testes utilizando busca em arquivos consistem em descobrir o desempenho da FPA sob uma aplicação real, e não um modelo sintético de aplicação como os testes realizados na seção anterior. Foram realizadas buscas em ambas máquinas de testes: em uma máquina do *cluster* OM e na máquina do LAHPC.

O teste de busca em arquivo consiste em contar as ocorrências de um determinado número inteiro em um arquivo de entrada, e apresentar esta quantidade como saída para o usuário. As entradas consistem em arquivos texto com números inteiros aleatórios, que variam de 0 a 65535.

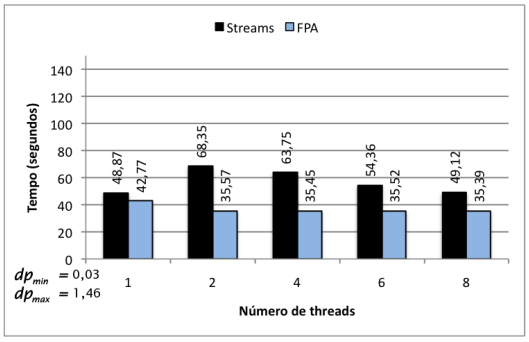
```
1  INÍCIO DA SEÇÃO PARALELA OPENMP
2          contador[thread_id]=0;
3          arquivo = open(arquivo_entrada);
4
5          seek( 1024 * 1024 * gigabytes / nthreads * thread_id );
6
7          for( j=0; j< 1024/nthreads; j++ )
8          {
9                  for( i=0; i< 1024*1024*gigabytes; i++ )
10                 {
11                         //Leitura de 1 inteiro do disco
12                         read(numero);
13
14                         //verifica se o número lido é igual
15                         //ao número procurado e incrementa
16                         //o contador
17                         if(numero == numero_procurado)
18                                 contador++;
19                 }
20         }
21         //sincroniza as threads
22         barrier();
23         //soma os os valores dos contadores de todas as threads
24         if( thread_id == 0 )
25                 for( i=0; i < numero_threads; i++ )
26                         contador[0]+=contador[thread_id];
27 FIM DA SEÇÃO PARALELA OPENMP
```

**Figura 4.72:** Pseudocódigo da busca em arquivos

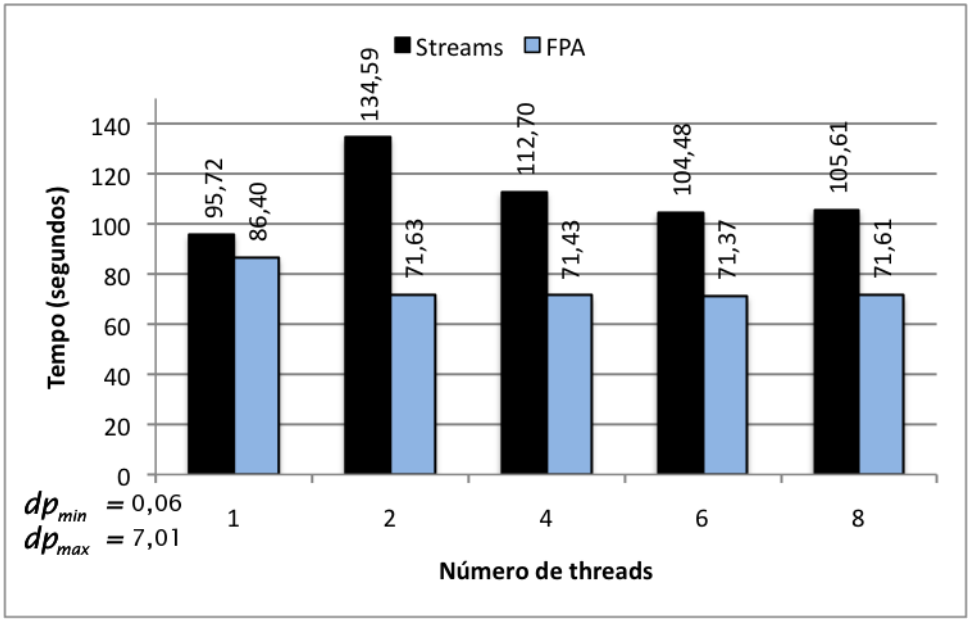
O particionamento entre as diversas *threads* é realizado de forma uniforme no arquivo, dividindo-o pelo número de *threads* na aplicação. Desta forma, cada *thread* é responsável pela busca em uma região de tamanho $1/n$ no arquivo, onde $n$ é o número total de *threads*. O método como o teste é realizado é apresentado no pseudocódigo da figura 4.72.




(a) Arquivos de 1 GB




(b) Arquivos de 2 GB



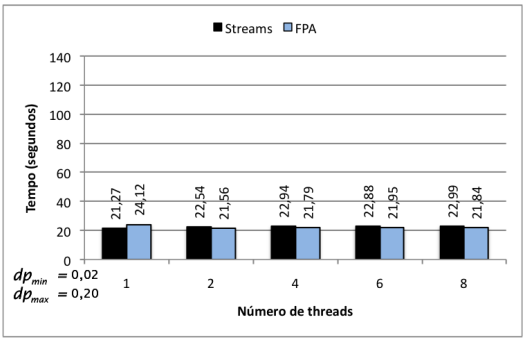

(c) Arquivos de 4 GB

**Figura 4.73:** Busca em arquivos

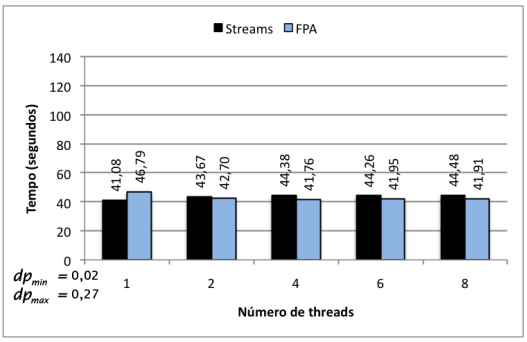
Nos gráficos da figura 4.73 verifica-se que existe ganho de desempenho com a implementação que utiliza FPA e também que seu comportamento mantem-se constante, independente do tamanho do arquivo de entrada. As principais vantagens do uso da FPA encontram-se no preenchimento completo do buffer, o que realiza uma leitura maior dos dados do arquivo e acarreta em uma menor movimentação do cabeçote de leitura do disco no momento em que o acesso físico é realizado, e no aproveitamento da sobreposição entre processamento e acesso a disco. Nota-se também que a partir de 2 *threads* o tempo de execução da FPA mantem-se praticamente constante, pois provavelmente o melhor aproveitamento possível das otimizações no acesso ao disco já tenha sido alcançado neste ponto.

O aproveitamento da aplicação paralela utilizando *streams* não acontece, pois para arquivos de 2 e 4 GB a aplicação sequencial é a mais rápida, e para arquivos de 1 GB, apenas a aplicação com 8 *threads* possui tempo de execução menor, porém este é muito

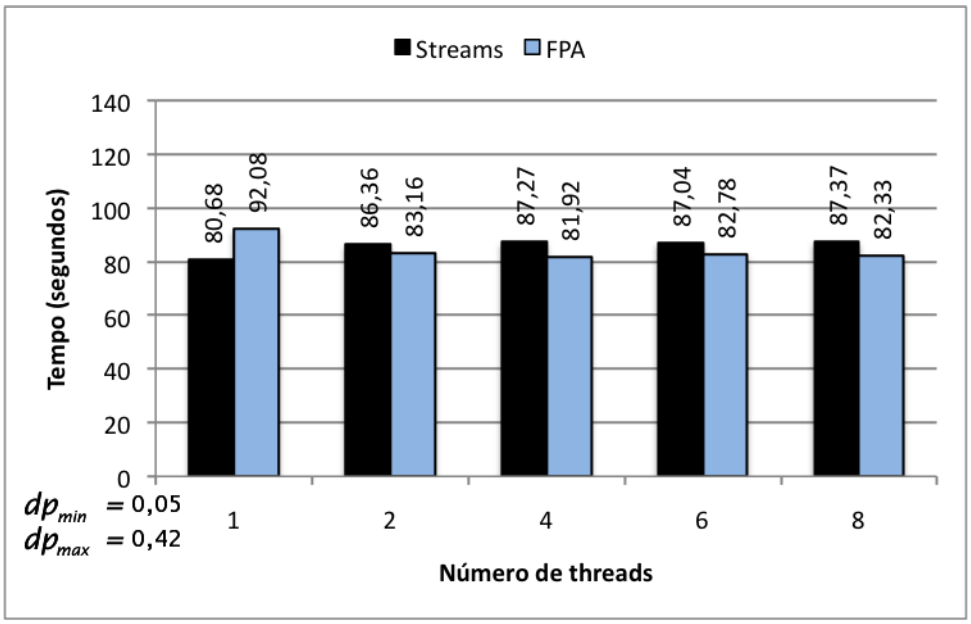
pequeno e não pode ser considerado como vantajoso para a situação.




(a) Arquivos de 1 GB




(b) Arquivos de 2 GB




(c) Arquivos de 4 GB

**Figura 4.74:** Busca em arquivos na máquina do LAHPC

## 4.4 Leitura de arquivos na aplicação OM

Um estudo de caso sobre o uso da FPA também foi feito implantando a biblioteca junto ao *software* de modelagem do projeto *Open Modeller*. O *Open Modeller* é um sistema de modelagem de propagação de espécies desenvolvido em conjunto pelo Centro de Referência de Informação Ambiental (Cria), Instituto Nacional de Pesquisas Espaciais (Inpe) e pela Epusp.

O software de modelagem é uma aplicação sequencial bastante custosa, com um tempo médio de execução de 30 minutos para a entrada utilizada. Tal entrada é um mapa considerado pequeno para as proporções do projeto, sendo de amplo uso nos testes e execuções dos experimentos realizados.

Como o projeto possui um *cluster* dispondo de diversas unidades de processamento por máquina, uma possibilidade de aproveitamento da aplicação no cluster é executar diversos *jobs* simultaneamente em um mesmo nó do *cluster*.

A entrada de dados da aplicação é composta por 350 arquivos, com tamanhos que variam de 4 KB a 50 MB, sendo em sua maioria arquivos consideravelmente pequenos. O experimento completo consiste em ler ao total 650 MB de dados do disco, e uma pequena quantidade de dados de escrita.

O *software* de modelagem utilizada a biblioteca de abstração de dados espaciais (Gdal[3] para acesso aods dados no disco. Desta forma,a FPA foi implantada no Gdal, e não no software de modelagem diretamente. A aplicação utiliza o Gdal apenas para a leitura de dados, implicando na não utilização da FPA nas operações de escrita.

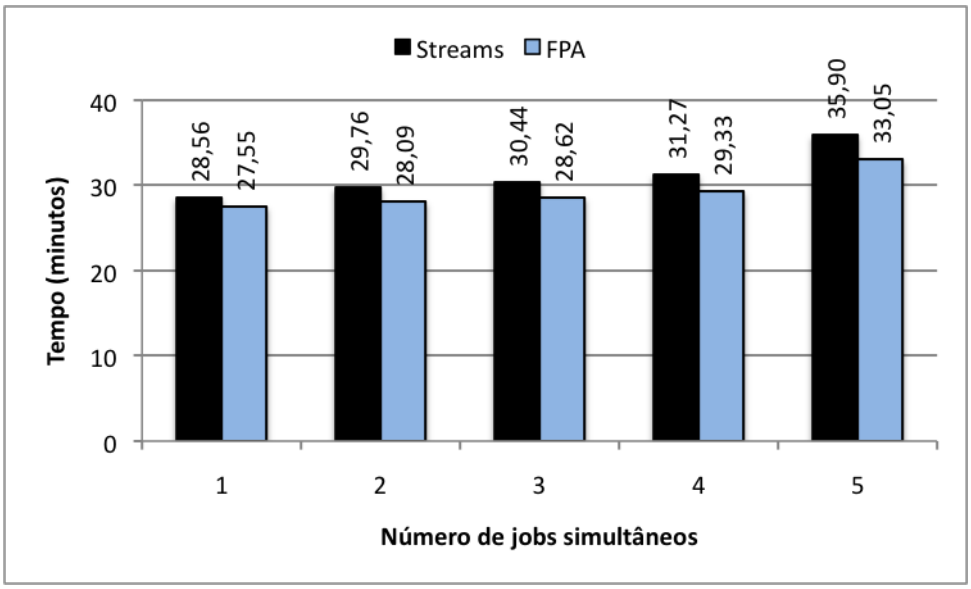



**Figura 4.75:** Tempo de execução do *software* de modelagem do projeto OM

Os testes foram realizados com as seguintes configuração para a FPA: 8 MB como tamanho de cada buffer, 128 MB para o limite *LA* e 2 GB para o limite *LM*.

Como os *jobs* são instâncias independentes, não existe compartilhamento dos buffers entre as aplicações. O gráfico da figura 4.75 mostra o tempo de execução da aplicação utilizando de 1 a 5 *jobs* simultâneos. É possível também observar que a leitura utilizando o preenchimento completo do buffer de uma única vez é vantajoso, mesmo quando estes não são compartilhados.

Como esta é uma situação onde envolve apenas leitura, todos os *jobs* lêem os mesmos arquivos, e o SO armazena temporariamente os dados de entrada, cada *job* aproveita tais dados armazenados temporariamente. Este armazenamento possibilita que a

[3]Do inglês *Geospatial Data Abstraction Library*

concorrência ao disco não aumente tanto à medida que o número de *jobs* simultâneos cresce.

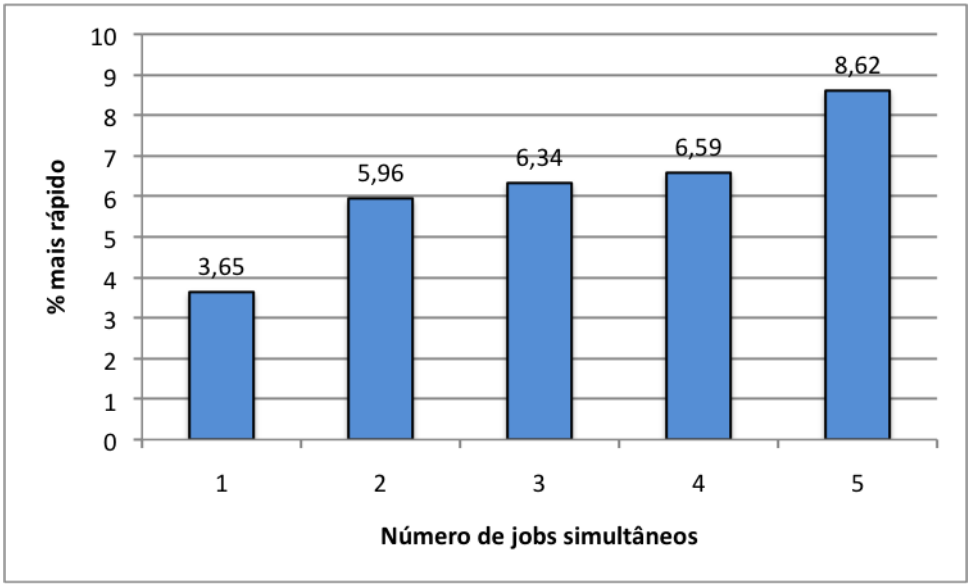



**Figura 4.76:** Porcentagem de ganho do uso da FPA em relação aos *streams* no *software* de modelagem do projeto OM

Mesmo com o armazenamento em buffers do SO com os *streams*, na leitura o uso da FPA toma vantagem em relação aos *streams*. O gráfico da figura 4.76 indica a porcentagem de tempo em que a execução da aplicação com FPA é mais rápida que a execução com *streams*, indicando ganho de desempenho com a FPA em todas as situações testadas.

## 4.5 Discussão

No decorrer do trabalho foram desenvolvidas diversas versões da biblioteca FPA, como as implementações com Mmap ou Calloc. O desenvolvimento dos testes do trabalho iniciou-se com a escrita de dados utilizando particionamento uniforme do arquivo, onde foi realizada uma análise e com base nisso decidido qual das versões implementadas seria a principal e utilizada nos demais testes.

Com base nos resultados dos testes com particionamento uniforme do arquivo decidiu-se que a versão oficial da biblioteca seria a implementação utilizando `calloc`, pois o desempenho da aplicação utilizando `mmap` foi inferior a esta.

A implementação que trata de enviar todos os buffers livres do arquivo possui vantagens principalmente nos casos onde não existe processamento na aplicação, e em grande parte das demais situações não apresenta desempenho em relação à implementação utilizando `calloc`. Outra desvantagem dessa implementação é a não possibi-

lidade de reutilização dos buffers, e sendo assim, esta também esteve fora do foco principal do trabalho.

Apesar da implementação utilizando `calloc` ser o foco do trabalho, as demais técnicas também estão implementadas e podem ser utilizadas de forma transparente através de especificações nos parâmetros da rotina de inicialização da FPA, a `fpa_start`.

### 4.5.1 Vantagens da FPA

As situações onde o uso da FPA torna-se vantajoso é dependente da quantidade de *threads* existentes na aplicação e também da quantidade de processamento de cálculos realizados. Esta relação entre processamento e concorrência ao disco possui influência direta no desempenho da biblioteca.

Durante as operações de leitura com múltiplas *threads*, o uso da FPA é vantajoso para as situações onde existe um baixo processamento de cálculos por parte da aplicação, como visto nos testes de leitura realizados.

Na escrita os resultados apresentaram um comportamento diferente, devido ao acesso a disco ser realizado após o uso do buffer. Isto possibilita com que a aplicação realize o seu processamento sem que seja necessário esperar pela escrita dos dados ao disco, e o envio das informações pode ser otimizado através do uso de uma única *thread* acessando o disco.

Observando os resultados dos testes envolvendo operações de leitura e escrita, a FPA mostrou-se ser uma alternativa vantajosa. A concorrência no acesso a disco é amplamente explorada, apresentando desempenho na maioria dos casos testados. As situações onde o uso da FPA não obteve desempenho foram onde existe um processamento excessivo por parte da aplicação, fazendo com que o tempo gasto no acesso a disco torne-se insignificante e assim o tempo gasto no gerenciamento dos buffers aparece como um custo de processamento adicional à aplicação.

O desempenho da biblioteca desenvolvida independe do tamanho do arquivo tratado. Foram realizados testes envolvendo arquivos de 1, 2 e 4 GB, onde a vantagem obtida no acesso manteve-se proporcional em todas as situações. Esta análise leva a concluir que o desempenho obtido durante o acesso a disco é uma característica constante, e que arquivos maiores ou menores implicam apenas em diferentes quantidades de acessos ao disco.

O tamanho dos buffers utilizados FPA também é outro fator influente no desempenho da aplicação, onde buffers menores tendem a um melhor desempenho em operações de leitura, caso exista compartilhamento dos buffers entre as *threads* da aplicação.

Já em operações de escrita acontece um melhor aproveitamento com buffers maiores, porém deve-se levar em conta que quanto maior é o buffer a quantidade de dados a serem enviados no final da execução da aplicação também é maior. Um fator de amortização para este problema é o limite *LA*, pois um valor que permita manter apenas 1 buffer que não esteja sendo utilizado em memória por vez faz com que os dados sejam enviados ao disco durante o decorrer da aplicação, deixando assim uma menor quantidade de informações para serem enviadas ao disco no final da aplicação. Como nas operações de leitura o acesso ao disco é feito antes da utilização do buffer, resta apenas a liberação da memória ocupada por estes após o seu uso. Dado que o tempo gasto durante a operação de liberação de memória é desprezível perante as operações envolvendo I/O, o limite *LA* não exerce grande influência nas operações de leitura.

### 4.5.2 Desvantagens da FPA

O mapeamento de arquivos em memória possui algumas desvantagens. Por exemplo, caso queira-se ler apenas o primeiro byte de um arquivo, o buffer completo terá que ser preenchido e logo após isso é que o primeiro byte do buffer será repassado à aplicação. Esta situação implica que mais informações terão de ser trazidas do disco, o que acontece de forma desnecessária. Em situações de escrita acontece algo semelhante, pois o buffer completo será enviado ao disco, mesmo que sejam escritos apenas alguns bytes. Uma possível solução para esse caso seria a implementação de um controle de alteração nos buffers, que poderia ser feito atrávés de listas ligadas.

# 5 Conclusões

Através dos resultados apresentados e discutidos no capítulo 4, conclui-se que o problema do gargalo no acesso a disco em arquiteturas *multicore* pode ser atenuado com a implementação da estratégia FPA. Porém, tornou-se visível que a estratégia possui um custo computacional com o gerenciamento dos buffers das informações, e em alguns casos, a solução proposta não proporciona ganho de desempenho em relação a soluções utilizando abordagens comuns, como com o uso dos *streams*.

A estratégia definida para otimização no acesso a disco apresentou ser uma alternativa vantajosa para obtenção de desempenho em operações de acesso a disco. Os resultados obtidos com os testes da implementação indicaram ganho de desempenho da FPA em relação ao uso dos *streams*, da LIBC.

O uso da FPA é vantajoso para diversas situações, onde um dos fatores que influenciam no resultado final é a quantidade de processamento existente na aplicação. Para situações com pouco processamento, a leitura utilizando FPA apresenta ganho de desempenho em relação aos *streams*. Já nas operações de escrita, o ganho de desempenho acontece com uma maior quantidade de processamento na aplicação. Porém estas vantagens acontecem até certo ponto, pois com uma grande quantidade de processamento o tempo gasto na espera pela leitura ou escrita da informação no disco torna-se insignificante, e é com base nessa espera que acontece a vantagem de uso da FPA.

O número de *threads* presente na aplicação influencia no comportamento da FPA, afinal com um maior número de *threads* a concorrência ao disco tende a aumentar. Porém, como a FPA utiliza uma *thread* adicional para envio dos dados ao disco, caso todas as unidades de processamento estejam ocupadas, esta *thread* da FPA pode ocasionar concorrência pela CPU com as *threads* da aplicação, podendo levar a uma perda de desempenho nessas situações.

Os testes realizados apontaram que o mapeamento com uso de rotinas de alocação de memória como `calloc` possuem ganho de desempenho em relação ao mapeamento utilizando `mmap`. A desvantagem do Mmap acontece devido à sincronização de dados com o disco pelo SO, o que pode ocorrer em momentos que aumentam a concorrência

ao disco e consequentemente pioram o desempenho da aplicação.

Durante as operações de leitura, a *thread* que realizar a chamada da rotina terá de aguardar pelo preenchimento completo do buffer, o que pode ser desvantajoso em algumas situações. Na maioria dos casos testados, esta é uma técnica vantajosa devido à menor sobrecarga do SO, que decorre do uso de uma única chamada de sistema. Porém, nas situações onde o ganho de tempo obtido pela menor sobrecarga do SO é irrelevante perante o processamento da aplicação, o preenchimento completo do buffer não apresenta ser uma alternativa vantajosa.

Foram realizados testes de leitura e escrita utilizando arquivos de 1, 2 e 4 GB, onde os resultados indicaram que o desempenho da FPA independe do tamanho dos arquivos.

O mapeamento de arquivos utilizando buffers em memória trata de enviar dados ao disco durante o processamento da aplicação. Com isto a aplicação pode continuar seu processamento e os dados posteriormente serão enviados ao disco de forma otimizada, e a concorrência ao disco também é reduzida, pois a escrita será realizada por uma única *thread*. Porém, a desvantagem do uso dos buffers em memória nas operações de escrita acontece com grandes quantidades de processamento na aplicação e os dados serão enviados apenas ao final da execução, consistindo em uma espera adicional desvantajosa.

O tamanho dos buffers implica diretamente no desempenho da FPA. Foram realizados testes com buffers de 16, 32 e 64 MB e os resultados apontaram que durante as operações de leitura o uso de buffers menores apresenta um ganho de desempenho em relação aos demais. Já para as operações de escrita, o ganho de desempenho prevalece com o maior buffer. Mas isto implica em uma espera maior para escrita em disco nestas situações, pois acontece um envio maior de dados ao disco no final da aplicação. Esta situação pode ser amenizada utilizando um limite de aviso *LA* menor na FPA, indicando que os dados devem ser enviados ao disco com antecedência.

A implantação da biblioteca FPA nas operações de leitura do *software* de modelagem do projeto OM apresentou ganho de desempenho em relação aos *streams*, mesmo sendo uma aplicação sequencial.

O trabalho contribui com a comunidade científica pela estratégia desenvolvida para a obtenção de desempenho em aplicações que realizam acesso a disco, que foi comprovada através da implementação.

## 5.1 Trabalhos futuros

Após a análise dos resultados foram observados diversos pontos fracos, onde acredita-se que melhorias e otimização nestes pontos podem contribuir para um melhor desempenho da FPA. Sendo assim, a seguir são apresentadas diversas recomendações, servindo como sugestão e ponto de partida para possíveis continuações do trabalho.

- Implementação de listas de controle de alterações de dados: com o uso destas listas torna-se possível que em operações que envolvam escrita sejam enviados ao disco apenas os dados alterados, e não sempre o buffer completo. Com estas listas também torna-se possível uma implementação com alocação de memória utilizando `malloc`, uma vez que esta possui desempenho superior ao `calloc`;
- Utilizar o reaproveitamento de buffers, de forma que estes não sejam desalocados da memória após o envio dos dados ao disco. Com isto acredita-se que poderá ser reduzida a sobrecarga do SO, podendo aumentar o desempenho das aplicações que fazem uso da FPA;
- Implementação de uma versão onde as operações de leitura devolvem os dados necessários à aplicação antes do preenchimento do buffer completo, caso a quantidade de dados requisitada seja menor do que o tamanho do buffer;
- Realizar mais testes utilizando a biblioteca em aplicações *I/O bound*;
- Desenvolver um pré-compilador que realize a substituição das chamadas de leitura e escrita tradicionais pelas rotinas FPA em aplicações de forma automática, podendo assim utilizar a biblioteca desenvolvida de uma forma mais transparente em aplicações já existentes.

## 5.2 Publicações

Com o resultado obtido na escrita de arquivos utilizando mapeamento dividido, foi escrito e publicado um artigo na XXXV Conferência Latino Americana de Informática (CLEI 2009) ((VASATA; SATO, 2009)), e os resultados obtidos da utilização da FPA no *software* de modelagem do *Open Modeller* contribuíram para a publicação de outro artigo no *8th International Information and Telecommunication Technologies Symposium* (I2TS 2009) ((STANZANI et al., 2009)). A aceitação destes artigos comprova a contribuição do trabalho desenvolvido, exposto neste documento.

As publicações derivadas deste trabalho que foram realizadas até o presente momento foram feitas com uma pequena parcela dos resultados apresentados, permitindo que futuras publicações ainda sejam realizadas aproveitando os demais resultados.

# Referências Bibliográficas


AKHTER, S.; ROBERTS, J. *Multi-Core Programming: Increasing performance through software multi-threading*. Santa Clara, CA, USA: Intel Press, 2006. 336 p.

BALAKRISHNAN, S. et al. The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 33, n. 2, p. 506–517, 2005. ISSN 0163-5964.

BARNEY, B. *POSIX Threads Programming*. 2009. Online by Lawrence Livermore National Laboratory. Accessado em 05/04/2009. Disponível em: <"https://computing-.llnl.gov/tutorials/pthreads">.

BENSON, G.; LONG, K.; PACHECO, P. The Performance of Parallel Disk Write Methods for Linux Multiprocessor Nodes. *Lecture notes in computer science*, Springer, 2130 Fulton Street, San Francisco, CA 94117-1080, USA, p. 71–80, 2003. ISSN 1611-3349.

BULPIN, J.; PRATT, I. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In: CITESEER. *Workshop on Duplicating, Deconstructing, and Debunking (WDDD04)*. J J Thomson Avenue, Cambridge, UK, CB3 0FD., 2004. p. 53–64.

CARLEY, L. R.; GANGER, G. R.; NAGLE, D. F. Mems-based integrated-circuit mass-storage systems. *Commun. ACM*, ACM, New York, NY, USA, v. 43, n. 11, p. 72–80, 2000. ISSN 0001-0782.

CHAPMAN, B. et al. *Using OpenMP: portable shared memory parallel programming*. Cambridge: The MIT Press, 2007.

CULLER, E. D.; SINGH, J. P. *Parallel Computer Architecture - A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers, Inc., 1999. 1025 p.

DIEFENDORFF, K.; DUBEY, P. How multimedia workloads will change processor design. *Computer*, IEEE, v. 30, n. 9, p. 43–45, 1997.

DOCY, S.; HONG, I.; POTKONJAK, M. Throughput optimization in disk-based real-time application specific systems. In: IEEE COMPUTER SOCIETY. *System Synthesis, 1996. Proceedings., 9th International Symposium on*. Washington, DC, USA, 1996. p. 133–138.

DODONOV, E.; GUARDIA, H. Um mecanismo integrado de cache e prefetching para sistemas de entrada e saída de alto desempenho. *Dissertação de mestrado*, Ufscar, São Carlos, SP, Brasil, 2004.

DOWNEY, A. *The little book of semaphores*. [S.l.]: Green Tea Press, 2005.

EKMAN, M.; WARG, F.; NILSSON, J. An in-depth look at computer performance growth. *ACM SIGARCH Computer Architecture News*, ACM, New York, NY, USA, v. 33, n. 1, p. 144–147, 2005.

EL-REWINI, H.; ABD-EL-BARR, M. *Advanced computer architecture and parallel processing*. USA: Wiley-Interscience, 2005. ISBN 0-471-46740-5.

FERRAZ, I. *Programação com Arquivos, 1ª edição*. Barueri, SP, Brasil: Manole, 2002. 345 p. ISBN 8520414893.

FIELD, D. *The Impact of Multi-core Processors on Application Performance*. 2008.

GANGER, G. R.; KAASHOEK, M. F. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. USENIX Association, Berkeley, CA, USA, p. 1–1, 1997.

GEER, D. Chip makers turn to multicore processors. *Computer*, IEEE, v. 38, n. 5, p. 11–13, 2005. ISSN 0018-9162.

GEER, D. For Programmers, Multicore Chips Mean Multiple Challenges. *Computer*, v. 40, n. 9, p. 17–19, 2007.

GEPNER, P.; KOWALIK, M. Multi-core processors: New way to achieve high system performance. In: *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*. Los Alamitos, CA, USA: [s.n.], 2006. p. 9–13. ISBN 0-7695-2554-7.

HUGHES, C.; HUGHES, T. *Parallel and Distributed Programming Using C++*. [S.l.]: Prentice Hall Professional Technical Reference, 2003. 720 p. ISBN 0-13-101376-9.

HUGHES, C.; HUGHES, T. *Professional Multicore Programming: Design and Implementation for C++ Developers*. [S.l.]: Wiley Publishing, Inc., 2008.

KALLA, R.; SINHAROY, B.; TENDLER, J. IBM Power5 chip: A dual-core multithreaded processor. *IEEE micro*, v. 24, n. 2, p. 40–47, 2004.

KOUFATY, D.; MARR, D. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, v. 23, n. 2, p. 56–65, 2003.

KUMAR, R. et al. Heterogeneous chip multiprocessors. *Computer*, v. 38, n. 11, p. 32–38, 2005.

LEVINE, D.; GALBREATH, N.; GROPP, W. Applications-driven parallel i/o. In: *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1993. p. 462–471. ISBN 0-8186-4340-4.

LEWIS, B.; BERG, D. *PThreads Primer - A Guide to Multithreaded Programming*. One Lake Street, Upper Saddle River, NJ: Prentice-Hall PTR, 1996. 370 p. ISBN 0-13-443698-9.

LOVE, R. *Linux system programming*. Sebastopol, CA: O'Reilly Media, Inc., 2007. 400 p. ISBN 978-0-596-00958-8.

MARTENS, D.; KATCHABAW, M. Disk access analysis for system performance optimization. *5th WSEAS International Conference on Applied Computer Science*, 2006.

MATTHEWS, J. et al. Intel® Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *ACM Transactions on Storage (TOS)*, ACM, New York, NY, USA, v. 4, n. 2, p. 4, 2008.

MORE, S. et al. MTIO. A multi-threaded parallel I/O system. In: *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1997. p. 368–373.

MORETTI, C.; BITTENCOURT, T. *Aspectos Gerais da Computação Paralela e do Sistema PVM*. São Paulo, SP, Brasil, 2004. 11 p.

NICHOLS, B.; BUTTLAR, D. *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly Media, Inc., 1996. 288 p. ISBN 1-56592-115-1.

PATTERSON, D.; HENNESSY, J. *Computer Architecture: A Quantitative Approach*. 3rd. ed. [S.l.]: Morgan Kaufmann, 2003. 833 p. ISBN 1-55860-724-2.

RANGASWAMI, R. et al. Building mems-based storage systems for streaming media. *Trans. Storage*, ACM, New York, NY, USA, v. 3, n. 2, p. 6, 2007. ISSN 1553-3077.

REILEY, W. Efficient parallel out-of-core implementation of the Cholesky factorization. *University of Texas at Austin, Austin, TX*, Citeseer, p. 32, 1999.

ROBBINS, A.; ROBBINS, S. *Unix Systems Programming: Communication, Concurrency, and Threads*. Upper Saddle River, New Jersey, USA: Pearson Education Inc., 2003. 912 p. ISBN 0-13-042411-0.

SATO, M. Openmp: parallel programming api for shared memory multiprocessors and on-chip multiprocessors. In: *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*. New York, NY, USA: ACM, 2002. p. 109–111. ISBN 1-58113-576-9.

SGI. *Linux Application Tuning Guide*. 2008. Online by SGI Inc. Accessado em 05/04/2009. Disponível em: <"http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc-.cgi?coll=linux&db=bks&fname=/SGI_Developer/LX_AppTune/ch07.html">.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Fundamentos de sistemas operacionais*. 6ta. ed. Rio de Janeiro, RJ, Brasil: LTC - Livros técnicos e científicos editora S.A., 2004. 600 p. ISBN 85-216-1414-4.

SPRACKLEN, L.; ABRAHAM, S. Chip multithreading: Opportunities and challenges. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005. p. 248–252.

STALLINGS, W. *Arquitetura e Organização de Computadores*. São Paulo, SP, Brasil: Prentice Hall, 2002. 792 p. ISBN 8587918532.

STANZANI, S. L. et al. A job scheduler based on application profile for openmodeller. *8th International Information and Telecommunication Technologies Symposium*, Florianópolis, SC, Brazil, p. 8, 2009.

SUTTER, H.; LARUS, J. Software and the concurrency revolution. *Queue*, ACM, New York, NY, USA, v. 3, n. 7, p. 62, 2005.

TANENBAUM, A.; WOODHULL, A. *Sistemas Operacionais, Projeto e Implementação*. 2da. ed. Porto Alegre, RS, Brasil: Bookman, 2000. 759 p.

THAKUR, R.; LUSK, E.; GROPP, W. I/O in parallel applications: The weakest link. *International Journal of High Performance Computing Applications*, Argonne, IL, USA, v. 12, n. 4, p. 389–395, 1998.

TIAN, X. et al. Compiler and runtime support for running OpenMP programs on Pentium-and Itanium-architectures. In: IEEE COMPUTER SOCIETY. *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. [S.l.], 2003. p. 47.

TUCK, N.; TULLSEN, D. M. Initial observations of the simultaneous multithreading Pentium 4 processor. In: CITESEER. *Proceedings of 12th Intl Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2003. p. 26. ISBN 0-7695-2021-9.

UEDA, T.; HIRATE, Y.; YAMANA, H. Exploiting idle cpu cores to improve file access performance. In: *ICUIMC '09: Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*. New York, NY, USA: ACM, 2009. p. 529–535. ISBN 978-1-60558-405-8.

VACHHARAJANI, N. et al. Chip multi-processor scalability for single-threaded applications. *SIGARCH Computer Architecture News*, ACM, New York, NY, USA, v. 33, n. 4, p. 44–53, 2005.

VASATA, D.; SATO, L. M. Acesso paralelo a arquivos em sistemas multiprocessadores. *XXXV Conferência Latino Americana de Informática*, Pelotas, RS, Brasil, p. 10, 2009.

WIRZENIUS, L. et al. *Linux System Administrator's Guide*. 2008. Online by About.com. Accessado em 05/04/2009. Disponível em: <"http://linux.about.com/od-/lsa_guide/a/gdelsa44.htm">.

# Glossário

*append* Adicionar informações em um arquivo a partir de seu final.

buffer Região de memória temporária.

*bug* Falha durante a execução de um determinado *software*.

*cache* Dispositivo de acesso rápido, interno a um sistema, que serve de intermediário entre um operador de um processo e o dispositivo de armazenamento ao qual esse operador acede.

*chip* Circuito eletrônico miniaturizado, utilizado amplamente em equipamentos eletrônicos.

*clock* Sinal usado para coordenar as ações de dois ou mais circuitos eletrônicos.

*cluster* Conjunto de computadores, muitas vezes construído a partir de computadores convencionais ligados em rede. Tais computadores comunicam-se através do sistema, trabalhando como se fossem uma única máquina de grande porte.

*default* Valor padrão fornecido automaticamente pelo sistema, quando não fornecido pelo usuário.

*core* Cada um dos processadores contidos em uma mesma pastilha com vários processadores encapsulados.

*driver* Software utilizado para que o sistema operacional comunique-se com um determinado dispositivo em *hardware*.

*flag* Valor informando uma determinada característica.

*flush* Operação que torna informações temporárias residentes em buffers como permanentes, também referenciando à ação de limpar o *buffer*.

*header* Arquivo texto contendo informações e formato de chamadas de rotinas implementadas em uma biblioteca da linguagem C.

IRIX Sistema operacional baseado no Unix com o BSD desenvolvido pela Silicon Graphics para funcionar em computadores com arquitetura MIPS de 32 e 64 bits em servidores e estações de trabalho.

*job* Sinônimo de tarefa, dentro do contexto computacional.

*kernel* Núcleo de um sistema operacional, responsável pelo gerenciamento da memória, multi-processamento, arquivos e periféricos.

*loop* Conjunto de instruções de um programa executado repetidamente.

| | |
|---|---|
| *middleware* | Camada de *software* que não consiste necessariamente em uma aplicação, e tem o objetivo de realizar ou facilitar a interligação entre diversos componentes de um *software* ou conjunto destes. |
| MPI | Padrão para comunicação de dados em computação paralela. |
| nó | Computador integrante de um *cluster*. |
| *overhead* | Sobrecarga do sistema operacional. |
| Posix | Conjunto de normas definida pelo IEEE, com o objetivo de garantir a portabilidade de um sistema operacional que atenda às normas Posix para outro sistema Posix. |

# APÊNDICE A – Documentação da implementação

# FPA - File for Parallel Access

1.0

Gerado por Doxygen 1.6.1

Mon Dec 28 22:36:59 2009

# Sumário


**1 Índice das Estruturas de Dados** **1**

1.1 Estruturas de Dados . . . . . 1

**2 Índice dos Arquivos** **3**

2.1 Lista de Arquivos . . . . . 3

**3 Estruturas** **5**

3.1 Referência da Estrutura ARQUIVO_ABERTO . . . . . 5

3.2 Referência da Estrutura FPA_BUFFER . . . . . 6

3.3 Referência da Estrutura LISTA_ARQUIVOS . . . . . 7

3.4 Referência da Estrutura ST_FPA_FILE . . . . . 8

**4 Arquivos** **9**

4.1 Referência do Arquivo fpa.c . . . . . 9

4.1.1 Descrição Detalhada . . . . . 12

4.1.2 Funções . . . . . 12

4.1.2.1 fpa_close . . . . . 12

4.1.2.2 fpa_eof . . . . . 12

4.1.2.3 fpa_flush . . . . . 12

4.1.2.4 fpa_fread . . . . . 13

4.1.2.5 fpa_fseek . . . . . 13

4.1.2.6 fpa_fwrite . . . . . 13

4.1.2.7 fpa_lseek . . . . . 14

4.1.2.8 fpa_open . . . . . 14

4.1.2.9 fpa_read . . . . . 14

4.1.2.10 fpa_start . . . . . 15

4.1.2.11 fpa_tell . . . . . 15
4.1.2.12 fpa_write . . . . . 15
4.1.2.13 thread_write . . . . . 16
4.2 Referência do Arquivo fpa.h . . . . . 17
4.2.1 Descrição Detalhada . . . . . 18
4.2.2 Definições e macros . . . . . 19
4.2.2.1 FPA_SEEK_CUR . . . . . 19
4.2.2.2 FPA_SEEK_END . . . . . 19
4.2.2.3 FPA_SEEK_SET . . . . . 19
4.2.3 Funções . . . . . 19
4.2.3.1 fpa_close . . . . . 19
4.2.3.2 fpa_eof . . . . . 19
4.2.3.3 fpa_flush . . . . . 20
4.2.3.4 fpa_fread . . . . . 20
4.2.3.5 fpa_fseek . . . . . 20
4.2.3.6 fpa_fwrite . . . . . 21
4.2.3.7 fpa_lseek . . . . . 21
4.2.3.8 fpa_open . . . . . 21
4.2.3.9 fpa_read . . . . . 22
4.2.3.10 fpa_start . . . . . 22
4.2.3.11 fpa_tell . . . . . 22
4.2.3.12 fpa_write . . . . . 23
4.3 Referência do Arquivo listas.c . . . . . 24
4.3.1 Descrição Detalhada . . . . . 25
4.3.2 Funções . . . . . 25
4.3.2.1 busca_arquivo_liberar_buffer . . . . . 25
4.3.2.2 lista_arquivos_buscar . . . . . 25
4.3.2.3 lista_arquivos_excluir . . . . . 26
4.3.2.4 lista_arquivos_flush . . . . . 26
4.3.2.5 lista_arquivos_incluir_read . . . . . 26
4.3.2.6 lista_arquivos_incluir_write . . . . . 26
4.3.2.7 lista_buffers_aloca_read . . . . . 27
4.3.2.8 lista_buffers_busca_aloca_write . . . . . 27
4.3.2.9 lista_buffers_busca_read . . . . . 27

4.3.2.10 retira_buffer_arquivo . . . . . 28
4.3.2.11 retira_todos_os_buffers_livres . . . . . 28
4.4 Referência do Arquivo listas.h . . . . . 29
4.4.1 Descrição Detalhada . . . . . 31
4.4.2 Funções . . . . . 31
4.4.2.1 busca_arquivo_liberar_buffer . . . . . 31
4.4.2.2 lista_arquivos_buscar . . . . . 31
4.4.2.3 lista_arquivos_excluir . . . . . 31
4.4.2.4 lista_arquivos_flush . . . . . 32
4.4.2.5 lista_arquivos_incluir_read . . . . . 32
4.4.2.6 lista_arquivos_incluir_write . . . . . 32
4.4.2.7 lista_buffers_aloca_read . . . . . 32
4.4.2.8 lista_buffers_busca_aloca_read . . . . . 33
4.4.2.9 lista_buffers_busca_aloca_write . . . . . 33
4.4.2.10 lista_buffers_busca_read . . . . . 33
4.4.2.11 retira_buffer_arquivo . . . . . 34
4.4.2.12 retira_todos_os_buffers_livres . . . . . 34

# Capítulo 1

# Índice das Estruturas de Dados

## 1.1 Estruturas de Dados

Aqui estão as estruturas de dados e suas respectivas descrições:


ARQUIVO_ABERTO . . . . . 5
FPA_BUFFER . . . . . 6
LISTA_ARQUIVOS . . . . . 7
ST_FPA_FILE . . . . . 8

# Capítulo 2

# Índice dos Arquivos

## 2.1 Lista de Arquivos

Esta é a lista de todos os arquivos documentados e suas respectivas descrições:

fpa.c (Arquivo com a implementação das rotinas da biblioteca FPA ) . . . . . 9
fpa.h (Cabeçalhos das rotinas da biblioteca FPA ) . . . . . 17
listas.c (Arquivo com a implementação das listas internas da biblioteca FPA ) 24
listas.h (Cabeçalhos das rotinas de manipulação de listas da biblioteca FPA ) . 29

# Capítulo 3

# Estruturas

## 3.1 Referência da Estrutura ARQUIVO_ABERTO

### Campos de Dados

- char **nome_arquivo** [PATH_MAX+1]
- int **fd**
- FILE ∗ **file**
- struct FPA_BUFFER ∗ **buffers**
- struct ARQUIVO_ABERTO ∗ **anterior**
- struct ARQUIVO_ABERTO ∗ **proximo**
- unsigned long int **n_buffers_alocados**
- unsigned long int **n_threads_utilizando**

A documentação para esta estrutura foi gerada a partir do seguinte arquivo:

- fpa.h

## 3.2 Referência da Estrutura FPA_BUFFER

### Campos de Dados

- unsigned long long **bloco**
- char * **dados**
- char * **end_inicio**
- char * **end_fim**
- off_t **pos_arquivo_inicio**
- off_t **pos_arquivo_fim**
- struct FPA_BUFFER * **anterior**
- struct FPA_BUFFER * **proximo**
- int **usando**
- char **alterado**
- char **cheio**

A documentação para esta estrutura foi gerada a partir do seguinte arquivo:

- fpa.h

## 3.3 Referência da Estrutura LISTA_ARQUIVOS

### Campos de Dados

- struct ARQUIVO_ABERTO * **primeiro**
- struct ARQUIVO_ABERTO * **ultimo**
- int **n**

A documentação para esta estrutura foi gerada a partir do seguinte arquivo:

- fpa.h

## 3.4 Referência da Estrutura ST_FPA_FILE

### Campos de Dados

- char **nome_arquivo** [PATH_MAX+1]
- char **modo_acesso** [5]
- struct FPA_BUFFER ∗ **buffer_r**
- struct FPA_BUFFER ∗ **buffer_w**
- char ∗ **end_r**
- char ∗ **end_w**
- struct ARQUIVO_ABERTO ∗ **arquivo**

A documentação para esta estrutura foi gerada a partir do seguinte arquivo:

- fpa.h

# Capítulo 4

# Arquivos

## 4.1 Referência do Arquivo fpa.c

Arquivo com a implementação das rotinas da biblioteca FPA. `#include "listas.h"`

`#include "fpa.h"`

`#include <unistd.h>`

### Definições e Macros

- #define **_XOPEN_SOURCE** 500

### Funções

- void ∗ thread_write (void ∗arg)

  *Thread de escrita.*

- struct RET_SEEK **fpa_seek** (FPA_FILE ∗fpa_file, long long int fpa_offset, int fpa_whence)
- struct RET_SEEK **fpa_seek_read** (FPA_FILE ∗fpa_file, long long int fpa_-offset, int fpa_whence)
- struct RET_SEEK **fpa_seek_write** (FPA_FILE ∗fpa_file, long long int fpa_-offset, int fpa_whence)
- struct RET_SEEK **fpa_seek_read_write** (FPA_FILE ∗fpa_file, long long int fpa_offset, int fpa_whence)
- struct RET_SEEK **fpa_seek_read_write_separado** (FPA_FILE ∗fpa_file, long long int fpa_offset, int fpa_whence)
- void fpa_start (unsigned long long aviso_memoria, char tecnica)

  *Inicialização da FPA. Inicializa os semáforos e também a thread de escrita.*

- FPA_FILE * fpa_open (char nome_arquivo[PATH_MAX+1], char modo_acesso[8])

  *Abre um arquivo utilizando FPA.*

- int fpa_close (FPA_FILE *fpa_file)

  *Fecha um arquivo utilizando FPA.*

- ssize_t fpa_fwrite (const void *end_dados, size_t tam_item, size_t n_items, FPA_FILE *fpa_file)

  *rotina de escrita, semelhante à fwrite*

- ssize_t fpa_write (FPA_FILE *fpa_file, const void *end_dados, size_t qtde_dados)

  *rotina de escrita*

- int fpa_fseek (FPA_FILE *fpa_file, long long int fpa_offset, int fpa_whence)

  *Rotina de seek no arquivo.*

- off_t fpa_lseek (FPA_FILE *fpa_file, long long int fpa_offset, int fpa_whence)

  *Rotina de seek no arquivo.*

- size_t fpa_fread (const void *end_dados, size_t tam_item, size_t n_items, FPA_FILE *fpa_file)

  *Rotina de leitura de dados.*

- ssize_t fpa_read (FPA_FILE *fpa_file, const void *end_dados, size_t qtde_dados)

  *Rotina de leitura de dados.*

- void **fpa_imprime** (char *palavra)
- int fpa_flush (FPA_FILE *fpa_file)

  *Envia os dados ao disco, porém sem liberar a memória dos buffers.*

- long fpa_tell (FPA_FILE *fpa_file)

  *Retorna a posição do ponteiro de leitura ou escrita no arquivo.*

- int fpa_eof (FPA_FILE *fpa_file)

## Variáveis

- char _tecnica = 'c'

  *Técnica utilizada para o mapeamento dos buffers. 'c' para o uso de calloc e 'm' para o uso de mmap. Calloc é o valor default.*

- char _modo = '1'

  *Modo de liberação dos buffers. '1' para liberação de 1 buffer por vez e 't' para liberação de todos os buffers não utilizados de uma única vez.*

- int fpa_started = 0

  *Flag indicando se as variáveis utilizadas na biblioteca já foram configuradas.*

- unsigned long _tamanho_buffer = 33554432

  *Tamanho de cada buffer, em bytes. Default: 32 MB.*

- unsigned long _aviso_memoria = 104857600

  *Quantidade de buffers mantidos em memória, sem necessidade de enviar dados ao disco. Default: 128 MB.*

- unsigned long _maximo_memoria = 2147483648

  *Máximo de memória alocada pelos buffers. Caso este valor seja atingido e mais buffers forem necessários, os novos buffers deverão esperar que outros buffers sejam liberados. Default: 2 GB.*

- unsigned long _memoria_usada = 0

  *Contador da quantidade de memória utilizada pelos buffers. Valor inicial 0, e este valor sempre é múltiplo do tamanho de buffers.*

- unsigned long chamadas = 0

  *Contador de requisições para novos buffers, quando o máximo de memória for atingido.*

- unsigned long **liberados** = 0
- struct LISTA_ARQUIVOS lista_arquivos

  *Lista de arquivos abertos.*

- sem_t sem_listas

  *Semáforo para controlar o acesso à lista de buffers.*

- sem_t sem_aviso_memoria

  *Semáforo para controlar quando os dados devem ser enviados ao disco. A thread de escrita aguarda dormindo por este semáforo.*

- sem_t sem_espera_memoria

  *Semáforo indicando que a memória máxima foi atingida, e os buffers devem esperar para serem alocados.*

- sem_t sem_acesso_disco

  *Semáforo para controlar o acesso ao disco.*

- unsigned long long int esperando_memoria

  *Contador da quantidade de buffers estão esperando para serem alocados.*

- pthread_t _thread_write

  *Thread de escrita de dados ao disco e de liberação de buffers.*

### 4.1.1 Descrição Detalhada

Arquivo com a implementação das rotinas da biblioteca FPA. Implementação das rotinas da biblioteca FPA. As principais rotinas são as de leitura, seek e escrita.

### 4.1.2 Funções

#### 4.1.2.1 int fpa_close (FPA_FILE ∗ *fpa_file*)

Fecha um arquivo utilizando FPA. O arquivo fechado terá todos os seus dados bufferizados enviados ao disco e também a memória ocupada pelos buffers liberada.

**Parâmetros:**

***fpa_file*** estrutura de acesso ao arquivo FPA_FILE.

**Retorna:**

0 para ok e -1 para casos de erro.

#### 4.1.2.2 int fpa_eof (FPA_FILE ∗ *fpa_file*)

Verifica se o final do arquivo foi atingido

**Parâmetros:**

***fpa_file*** arquivo FPA_FILE

**Retorna:**

0 caso não seja o final do arquivo e 1 caso contrário

#### 4.1.2.3 int fpa_flush (FPA_FILE ∗ *fpa_file*)

Envia os dados ao disco, porém sem liberar a memória dos buffers. Envia todos os dados armazenados em buffers ao disco. Os dados permanecem em memória, e os buffers não são desalocdos.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE aberto.

**Retorna:**

0 para ok e -1 caso ocorram problemas.

#### 4.1.2.4 size_t fpa_fread (const void * *end_dados*, size_t *tam_item*, size_t *n_items*, FPA_FILE * *fpa_file*)

Rotina de leitura de dados. Leitura de dados do arquivo, com parâmetros compatíveis à rotina fread, da biblioteca stdio da libc

**Parâmetros:**

***end_dados*** Ponteiro para a região de memória onde os dados serão armazenados.

***tam_item*** Tamanho da estrutura a ser lida (em bytes).

***n_items*** Quantidade de estruturas a serem lidas.

***fpa_file*** Estrutura FPA_FILE com o arquivo aberto.

**Retorna:**

Número de estruturas lidas

#### 4.1.2.5 int fpa_fseek (FPA_FILE * *fpa_file*, long long int *fpa_offset*, int *fpa_whence*)

Rotina de seek no arquivo. Realiza a movimentação do ponteiro de leitura ou escrita no arquivo. O nome fseek é mantido por questões de compatibilidade com a função fseek da biblioteca stdio, disponível na libc.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE aberto.

***fpa_offset*** Posição no arquivo, em bytes.

***fpa_whence*** Posicão de referência do offset, podendo assumir os valores FPA_-SEEK_SET (início do arquivo - 0), FPA_SEEK_CUR (posição atual - 1) ou FPA_SEEK_END (final do arquivo - 2 ).

**Retorna:**

número de bytes movimentados

#### 4.1.2.6 ssize_t fpa_fwrite (const void * *end_dados*, size_t *tam_item*, size_t *n_items*, FPA_FILE * *fpa_file*)

rotina de escrita, semelhante à fwrite Esta rotina realiza a mesma função que a rotina fpa_write, porém têm seus parâmetros ajustados por questões de compatibilidade de parâmetros da rotina fwrite, da biblioteca stdio disponível na libc.

**Parâmetros:**

***end_dados*** Ponteiro para a região de memória com as informações a serem enviadas ao disco

***tam_item*** Tamanho da estrutura de dados apontada

***n_items*** Quantidade de estruturas a serem enviadas

***fpa_file*** Estrutura com arquivo aberto FPA_FILE

#### 4.1.2.7 off_t fpa_lseek (FPA_FILE ∗ *fpa_file*, long long int *fpa_offset*, int *fpa_whence*)

Rotina de seek no arquivo. Realiza a movimentação do ponteiro de leitura ou escrita no arquivo.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE aberto.

***fpa_offset*** Posição no arquivo, em bytes.

***fpa_whence*** Posicão de referência do offset, podendo assumir os valores FPA_-SEEK_SET (início do arquivo - 0), FPA_SEEK_CUR (posição atual - 1) ou FPA_SEEK_END (final do arquivo - 2 ).

**Retorna:**

número de bytes movimentados

#### 4.1.2.8 FPA_FILE∗ fpa_open (char *nome_arquivo*[PATH_MAX+1], char *modo_acesso*[8])

Abre um arquivo utilizando FPA. Abre um arquivo utilizando a biblioteca FPA

**Parâmetros:**

***nome_arquivo*** Nome do arquivo a ser aberto

***modo_acesso*** Tipo de acesso ao arquivo, como leitura, escrita ou ambos. Os modos válidos são r, r+, w, w+, rb, r+b, rb+, w+, wb+, w+b e rw.

**Retorna:**

ponteiro para uma estrutura FPA_FILE, utilizada para acessar o arquivo.

#### 4.1.2.9 ssize_t fpa_read (FPA_FILE ∗ *fpa_file*, const void ∗ *end_dados*, size_t *qtde_dados*)

Rotina de leitura de dados. Leitura de dados do arquivo.

**Parâmetros:**

***end_dados*** Ponteiro para a região de memória onde os dados serão armazenados.

***tam_item*** Tamanho da estrutura a ser lida (em bytes).

***n_items*** Quantidade de estruturas a serem lidas.

***fpa_file*** Estrutura FPA_FILE com o arquivo aberto.

**Retorna:**

Número de bytes lidos

#### 4.1.2.10 void fpa_start (unsigned long long *aviso_memoria*, char *tecnica*)

Inicialização da FPA. Inicializa os semáforos e também a thread de escrita. Inicialização da FPA. Inicializa os semáforos e também a thread de escrita. Esta rotina não é thread-safe, devido à inicialização dos semáforos.

**Parâmetros:**

***aviso_memoria*** Quantidade de memória máxima utilizada pelos buffers sem que seja necessário enviar informações ao disco.

***tecnica*** Tipo de alocação de memória. 'm' para mmap e 'c' para calloc.

#### 4.1.2.11 long fpa_tell (FPA_FILE ∗ *fpa_file*)

Retorna a posição do ponteiro de leitura ou escrita no arquivo. Rotina de seek no arquivo.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE

**Retorna:**

posição no arquivo (em bytes)

#### 4.1.2.12 ssize_t fpa_write (FPA_FILE ∗ *fpa_file*, const void ∗ *end_dados*, size_t *qtde_dados*)

rotina de escrita Escreve dados nos buffers

**Parâmetros:**

***fpa_file*** Estrutura com arquivo aberto FPA_FILE

***end_dados*** Ponteiro para a região de memória com as informações a serem enviadas ao disco

***qtde_dados*** Número de bytes a serem escritos

#### 4.1.2.13 void ∗ thread_write (void ∗ *arg*)

Thread de escrita. Thread responsável pelo envio das informações ao disco e também da liberação da memória ocupada pelos buffers.

## 4.2 Referência do Arquivo fpa.h

Cabeçalhos das rotinas da biblioteca FPA. `#include <stdio.h>`

```
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
```

### Estruturas de Dados

- struct FPA_BUFFER
- struct ARQUIVO_ABERTO
- struct LISTA_ARQUIVOS
- struct ST_FPA_FILE

### Definições e Macros

- #define FPA_SEEK_SET 0

  *Início do arquivo.*

- #define FPA_SEEK_CUR 1

  *Posição atual no arquivo.*

- #define FPA_SEEK_END 2

  *Final do arquivo.*

### Definições de Tipos

- typedef struct ST_FPA_FILE FPA_FILE

  *Estrutura FPA_FILE, com as informações de acesso ao arquivo e ponteiros para os buffers de dados.*

## Funções

- void fpa_start (unsigned long long aviso_memoria, char tecnica)

  *Inicialização da FPA. Inicializa os semáforos e também a thread de escrita.*

- FPA_FILE * fpa_open (char nome_arq[PATH_MAX+1], char modo_acesso[2])

  *Abre um arquivo utilizando FPA.*

- int fpa_close (FPA_FILE *fpa_file)

  *Fecha um arquivo utilizando FPA.*

- ssize_t fpa_write (FPA_FILE *fpa_file, const void *end_dados, size_t qtde_dados)

  *rotina de escrita*

- ssize_t fpa_fwrite (const void *end_dados, size_t tam_item, size_t n_items, FPA_FILE *fpa_file)

  *rotina de escrita, semelhante à fwrite*

- ssize_t fpa_read (FPA_FILE *fpa_file, const void *end_dados, size_t qtde_dados)

  *Rotina de leitura de dados.*

- size_t fpa_fread (const void *end_dados, size_t tam_item, size_t n_items, FPA_FILE *fpa_file)

  *Rotina de leitura de dados.*

- off_t fpa_lseek (FPA_FILE *fpa_file, long long int fpa_offset, int fpa_whence)

  *Rotina de seek no arquivo.*

- int fpa_fseek (FPA_FILE *fpa_file, long long int fpa_offset, int fpa_whence)

  *Rotina de seek no arquivo.*

- long fpa_tell (FPA_FILE *fpa_file)

  *Rotina de seek no arquivo.*

- int fpa_flush (FPA_FILE *fpa_file)

  *Envia os dados ao disco, porém sem liberar a memória dos buffers.*

- int fpa_eof (FPA_FILE *fpa_file)

### 4.2.1 Descrição Detalhada

Cabeçalhos das rotinas da biblioteca FPA.

### 4.2.2 Definições e macros

#### 4.2.2.1 #define FPA_SEEK_CUR 1

Posição atual no arquivo. Define utilizado na rotina de seek, no parâmetro whence. Indica a posição atual do arquivo

#### 4.2.2.2 #define FPA_SEEK_END 2

Final do arquivo. Define utilizado na rotina de seek, no parâmetro whence. Indica a posição inicial do arquivo

#### 4.2.2.3 #define FPA_SEEK_SET 0

Início do arquivo. Define utilizado na rotina de seek, no parâmetro whence. Indica a posição inicial do arquivo

### 4.2.3 Funções

#### 4.2.3.1 int fpa_close (FPA_FILE ∗ *fpa_file*)

Fecha um arquivo utilizando FPA. O arquivo fechado terá todos os seus dados bufferizados enviados ao disco e também a memória ocupada pelos buffers liberada.

**Parâmetros:**

***fpa_file*** estrutura de acesso ao arquivo FPA_FILE.

**Retorna:**

0 para ok e -1 para casos de erro.

#### 4.2.3.2 int fpa_eof (FPA_FILE ∗ *fpa_file*)

Verifica se o final do arquivo foi atingido

**Parâmetros:**

***fpa_file*** arquivo FPA_FILE

**Retorna:**

0 caso não seja o final do arquivo e 1 caso contrário

#### 4.2.3.3 int fpa_flush (FPA_FILE ∗ *fpa_file*)

Envia os dados ao disco, porém sem liberar a memória dos buffers. Envia todos os dados armazenados em buffers ao disco. Os dados permanecem em memória, e os buffers não são desalocdos.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE aberto.

**Retorna:**

0 para ok e -1 caso ocorram problemas.

#### 4.2.3.4 size_t fpa_fread (const void ∗ *end_dados*, size_t *tam_item*, size_t *n_items*, FPA_FILE ∗ *fpa_file*)

Rotina de leitura de dados. Leitura de dados do arquivo, com parâmetros compatíveis à rotina fread, da biblioteca stdio da libc

**Parâmetros:**

***end_dados*** Ponteiro para a região de memória onde os dados serão armazenados.

***tam_item*** Tamanho da estrutura a ser lida (em bytes).

***n_items*** Quantidade de estruturas a serem lidas.

***fpa_file*** Estrutura FPA_FILE com o arquivo aberto.

**Retorna:**

Número de estruturas lidas

#### 4.2.3.5 int fpa_fseek (FPA_FILE ∗ *fpa_file*, long long int *fpa_offset*, int *fpa_whence*)

Rotina de seek no arquivo. Realiza a movimentação do ponteiro de leitura ou escrita no arquivo. O nome fseek é mantido por questões de compatibilidade com a função fseek da biblioteca stdio, disponível na libc.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE aberto.

***fpa_offset*** Posição no arquivo, em bytes.

***fpa_whence*** Posicão de referência do offset, podendo assumir os valores FPA_-SEEK_SET (início do arquivo - 0), FPA_SEEK_CUR (posição atual - 1) ou FPA_SEEK_END (final do arquivo - 2 ).

**Retorna:**

número de bytes movimentados

#### 4.2.3.6 ssize_t fpa_fwrite (const void * *end_dados*, size_t *tam_item*, size_t *n_items*, FPA_FILE * *fpa_file*)

rotina de escrita, semelhante à fwrite Esta rotina realiza a mesma função que a rotina fpa_write, porém têm seus parâmetros ajustados por questões de compatibilidade de parâmetros da rotina fwrite, da biblioteca stdio disponível na libc.

**Parâmetros:**

***end_dados*** Ponteiro para a região de memória com as informações a serem enviadas ao disco

***tam_item*** Tamanho da estrutura de dados apontada

***n_items*** Quantidade de estruturas a serem enviadas

***fpa_file*** Estrutura com arquivo aberto FPA_FILE

#### 4.2.3.7 off_t fpa_lseek (FPA_FILE * *fpa_file*, long long int *fpa_offset*, int *fpa_whence*)

Rotina de seek no arquivo. Realiza a movimentação do ponteiro de leitura ou escrita no arquivo.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE aberto.

***fpa_offset*** Posição no arquivo, em bytes.

***fpa_whence*** Posicão de referência do offset, podendo assumir os valores FPA_SEEK_SET (início do arquivo - 0), FPA_SEEK_CUR (posição atual - 1) ou FPA_SEEK_END (final do arquivo - 2 ).

**Retorna:**

número de bytes movimentados

#### 4.2.3.8 FPA_FILE* fpa_open (char *nome_arq*[PATH_MAX+1], char *modo_acesso*[2])

Abre um arquivo utilizando FPA. Abre um arquivo utilizando a biblioteca FPA

**Parâmetros:**

***nome_arquivo*** Nome do arquivo a ser aberto

***modo_acesso*** Tipo de acesso ao arquivo, como leitura, escrita ou ambos. Os modos válidos são r, r+, w, w+, rb, r+b, rb+, w+, wb+, w+b e rw.

**Retorna:**

ponteiro para uma estrutura FPA_FILE, utilizada para acessar o arquivo.

#### 4.2.3.9 ssize_t fpa_read (FPA_FILE ∗ *fpa_file*, const void ∗ *end_dados*, size_t *qtde_dados*)

Rotina de leitura de dados. Leitura de dados do arquivo.

**Parâmetros:**

***end_dados*** Ponteiro para a região de memória onde os dados serão armazenados.

***tam_item*** Tamanho da estrutura a ser lida (em bytes).

***n_items*** Quantidade de estruturas a serem lidas.

***fpa_file*** Estrutura FPA_FILE com o arquivo aberto.

**Retorna:**

Número de bytes lidos

#### 4.2.3.10 void fpa_start (unsigned long long *aviso_memoria*, char *tecnica*)

Inicialização da FPA. Inicializa os semáforos e também a thread de escrita. Inicialização da FPA. Inicializa os semáforos e também a thread de escrita. Esta rotina não é thread-safe, devido à inicialização dos semáforos.

**Parâmetros:**

***aviso_memoria*** Quantidade de memória máxima utilizada pelos buffers sem que seja necessário enviar informações ao disco.

***tecnica*** Tipo de alocação de memória. 'm' para mmap e 'c' para calloc.

#### 4.2.3.11 long fpa_tell (FPA_FILE ∗ *fpa_file*)

Rotina de seek no arquivo. Realiza a movimentação do ponteiro de leitura ou escrita no arquivo.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE aberto.

***fpa_offset*** Posição no arquivo, em bytes.

***fpa_whence*** Posicão de referência do offset, podendo assumir os valores FPA_-SEEK_SET (início do arquivo - 0), FPA_SEEK_CUR (posição atual - 1) ou FPA_SEEK_END (final do arquivo - 2 ).

**Retorna:**

número de bytes movimentados

Rotina de seek no arquivo.

**Parâmetros:**

***fpa_file*** Arquivo FPA_FILE

**Retorna:**

posição no arquivo (em bytes)

#### 4.2.3.12 ssize_t fpa_write (FPA_FILE * *fpa_file*, const void * *end_dados*, size_t *qtde_dados*)

rotina de escrita Escreve dados nos buffers

**Parâmetros:**

***fpa_file*** Estrutura com arquivo aberto FPA_FILE

***end_dados*** Ponteiro para a região de memória com as informações a serem enviadas ao disco

***qtde_dados*** Número de bytes a serem escritos

## 4.3 Referência do Arquivo listas.c

Arquivo com a implementação das listas internas da biblioteca FPA. `#include "listas.h"`

### Definições e Macros

- #define **_XOPEN_SOURCE** 500

### Funções

- void **imprime_lista_buffers** (struct ARQUIVO_ABERTO *arquivo, char *nome)
- struct ARQUIVO_ABERTO * lista_arquivos_buscar (char nome_arquivo[PATH_MAX+1], char modo)

  *Procura por um arquivo na lista de arquivos abertos.*

- struct ARQUIVO_ABERTO * **lista_arquivos_incluir_read_write** (char nome_arquivo[PATH_MAX+1])
- struct ARQUIVO_ABERTO * lista_arquivos_incluir_write (char nome_arquivo[PATH_MAX+1])

  *Inclui um novo arquivo para escrita na lista de arquivo abertos.*

- struct ARQUIVO_ABERTO * lista_arquivos_incluir_read (char nome_arquivo[PATH_MAX+1])

  *Inclui um novo arquivo para leitura na lista de arquivo abertos.*

- int lista_arquivos_excluir (struct ARQUIVO_ABERTO *arquivo)

  *Envia as informações armazenadas nos buffers do arquivo ao disco, libera a memória utilizada nos buffers do arquivo e exclui o arquivo da lista de arquivos abertos.*

- struct FPA_BUFFER * lista_buffers_busca_aloca_write (FPA_FILE *fpa_file, unsigned long long bloco)

  *Procura por um determinado buffer em uma lista de buffers de um arquivo. Caso o buffer não exista, este será alocado e colocado na lista. O buffer é alocado como sendo de leitura e escrita.*

- struct FPA_BUFFER * lista_buffers_busca_read (FPA_FILE *fpa_file, unsigned long long bloco)

  *Procura por um determinado buffer em uma lista de buffers de um arquivo.*

- struct FPA_BUFFER * lista_buffers_aloca_read (FPA_FILE *fpa_file, unsigned long long bloco)

  *Aloca um novo buffer de apenas leitura na lista de buffers.*

- struct ARQUIVO_ABERTO * busca_arquivo_liberar_buffer (void)

*Procura um arquivo que tenha buffers para serem liberados. Caso exista mais de um arquivo com buffers não utilizados, é retornado aquele que tem o maior número de buffer não utilizados no momento.*

- struct FPA_BUFFER * retira_buffer_arquivo (struct ARQUIVO_ABERTO *arquivo)

  *Retira um buffer da lista de buffers do arquivo. Retira um buffer da lista de buffers do arquivo. A lista de buffers do arquivo é atualizada, já sem o buffer.*

- struct FPA_BUFFER * retira_todos_os_buffers_livres (struct ARQUIVO_-ABERTO *arquivo)

  *Retira todos os buffers não utilizados de um arquivo de sua lista de buffers. Retira todos os buffers não utilizados de um arquivo da sua lista de buffers. A lista de buffers do arquivo é atualizada, já sem os buffers.*

- int lista_arquivos_flush (struct ARQUIVO_ABERTO *arquivo)

  *atualiza os dados no disco com as informações armazenadas nos buffers.*

## 4.3.1 Descrição Detalhada

Arquivo com a implementação das listas internas da biblioteca FPA. Implementação das rotinas de manipulação das listas de buffers e lista de arquivos abertos da fpa.

## 4.3.2 Funções

### 4.3.2.1 struct ARQUIVO_ABERTO* busca_arquivo_liberar_buffer (void) `[read]`

Procura um arquivo que tenha buffers para serem liberados. Caso exista mais de um arquivo com buffers não utilizados, é retornado aquele que tem o maior número de buffer não utilizados no momento.

**Retorna:**

Ponteiro para um elemento na lista de arquivos abertos.

### 4.3.2.2 struct ARQUIVO_ABERTO* lista_arquivos_buscar (char *nome_arquivo*[PATH_MAX+1], char *modo*) `[read]`

Procura por um arquivo na lista de arquivos abertos.

**Parâmetros:**

***nome_arquivo*** Nome do arquivo

***modo*** Modo de acesso. 'r' para leitura, 'w' para escrita e 'b' para leitura e escrita

**Retorna:**

Ponteiro para a estrutura do arquivo aberto. Caso o arquivo não exista, NULL é retornado.

#### 4.3.2.3 int lista_arquivos_excluir (struct ARQUIVO_ABERTO ∗ *arquivo*)

Envia as informações armazenadas nos buffers do arquivo ao disco, libera a memória utilizada nos buffers do arquivo e exclui o arquivo da lista de arquivos abertos.

**Parâmetros:**

***Estrutura*** de arquivo aberto.

**Retorna:**

0 para ok, e -1 caso ocorra algum erro no processo.

#### 4.3.2.4 int lista_arquivos_flush (struct ARQUIVO_ABERTO ∗ *arquivo*)

atualiza os dados no disco com as informações armazenadas nos buffers.

**Parâmetros:**

***arquivo*** Estrutura do arquivo aberto.

**Retorna:**

0 para ok, e -1 caso ocorra algum problema no processo.

#### 4.3.2.5 struct ARQUIVO_ABERTO∗ lista_arquivos_incluir_read (char *nome_arquivo*[PATH_MAX+1]) `[read]`

Inclui um novo arquivo para leitura na lista de arquivo abertos.

**Parâmetros:**

***nome_arquivo*** Nome do arquivo.

**Retorna:**

Ponteiro para a estrutura do novo arquivo.

#### 4.3.2.6 struct ARQUIVO_ABERTO∗ lista_arquivos_incluir_write (char *nome_arquivo*[PATH_MAX+1]) `[read]`

Inclui um novo arquivo para escrita na lista de arquivo abertos.

**Parâmetros:**

***nome_arquivo*** Nome do arquivo.

**Retorna:**

Ponteiro para a estrutura do novo arquivo.

#### 4.3.2.7 struct FPA_BUFFER∗ lista_buffers_aloca_read (FPA_FILE ∗ *fpa_file*, unsigned long long *bloco*) `[read]`

Aloca um novo buffer de apenas leitura na lista de buffers.

**Parâmetros:**

***fpa_file*** Arquivo aberto.

***bloco*** Região do arquivo, referente ao buffer alocado.

**Retorna:**

Ponteiro para o buffer, e -1 caso ocorra algum erro no processo.

#### 4.3.2.8 struct FPA_BUFFER∗ lista_buffers_busca_aloca_write (FPA_FILE ∗ *fpa_file*, unsigned long long *bloco*) `[read]`

Procura por um determinado buffer em uma lista de buffers de um arquivo. Caso o buffer não exista, este será alocado e colocado na lista. O buffer é alocado como sendo de leitura e escrita.

**Parâmetros:**

***fpa_file*** Arquivo aberto.

***bloco*** Região do arquivo, referente ao buffer alocado.

**Retorna:**

Ponteiro para o buffer, e -1 caso ocorra algum erro no processo.

#### 4.3.2.9 struct FPA_BUFFER∗ lista_buffers_busca_read (FPA_FILE ∗ *fpa_file*, unsigned long long *bloco*) `[read]`

Procura por um determinado buffer em uma lista de buffers de um arquivo.

**Parâmetros:**

***fpa_file*** Arquivo aberto.

***bloco*** Região do arquivo, referente ao buffer alocado. return Ponteiro para o buffer, e NULL caso o buffer não exista.

#### 4.3.2.10 struct FPA_BUFFER∗ retira_buffer_arquivo (struct ARQUIVO_ABERTO ∗ *arquivo*) `[read]`

Retira um buffer da lista de buffers do arquivo. Retira um buffer da lista de buffers do arquivo. A lista de buffers do arquivo é atualizada, já sem o buffer.

**Parâmetros:**

***Arquivo*** Estrutura de arquivo FPA_FILE.

**Retorna:**

Ponteiro para a buffer liberado.

#### 4.3.2.11 struct FPA_BUFFER∗ retira_todos_os_buffers_livres (struct ARQUIVO_ABERTO ∗ *arquivo*) `[read]`

Retira todos os buffers não utilizados de um arquivo de sua lista de buffers. Retira todos os buffers não utilizados de um arquivo da sua lista de buffers. A lista de buffers do arquivo é atualizada, já sem os buffers.

**Parâmetros:**

***Arquivo*** Estrutura de arquivo FPA_FILE.

**Retorna:**

Ponteiro para uma lista de buffers, contendo os buffers liberados.

## 4.4 Referência do Arquivo listas.h

Cabeçalhos das rotinas de manipulação de listas da biblioteca FPA. `#include <limits.h>`

`#include "fpa.h"`

### Funções

- struct ARQUIVO_ABERTO * lista_arquivos_buscar (char nome_arquivo[PATH_MAX+1], char modo)

  *Procura por um arquivo na lista de arquivos abertos.*

- struct ARQUIVO_ABERTO * lista_arquivos_incluir_write (char nome_arquivo[PATH_MAX+1])

  *Inclui um novo arquivo para escrita na lista de arquivo abertos.*

- struct ARQUIVO_ABERTO * lista_arquivos_incluir_read (char nome_arquivo[PATH_MAX+1])

  *Inclui um novo arquivo para leitura na lista de arquivo abertos.*

- int lista_arquivos_excluir (struct ARQUIVO_ABERTO *arquivo)

  *Envia as informações armazenadas nos buffers do arquivo ao disco, libera a memória utilizada nos buffers do arquivo e exclui o arquivo da lista de arquivos abertos.*

- int lista_arquivos_flush (struct ARQUIVO_ABERTO *arquivo)

  *atualiza os dados no disco com as informações armazenadas nos buffers.*

- struct FPA_BUFFER * lista_buffers_busca_aloca_write (FPA_FILE *fpa_file, unsigned long long bloco)

  *Procura por um determinado buffer em uma lista de buffers de um arquivo. Caso o buffer não exista, este será alocado e colocado na lista. O buffer é alocado como sendo de leitura e escrita.*

- struct FPA_BUFFER * lista_buffers_busca_read (FPA_FILE *fpa_file, unsigned long long bloco)

  *Procura por um determinado buffer em uma lista de buffers de um arquivo.*

- struct FPA_BUFFER * lista_buffers_aloca_read (FPA_FILE *fpa_file, unsigned long long bloco)

  *Aloca um novo buffer de apenas leitura na lista de buffers.*

- struct FPA_BUFFER * lista_buffers_busca_aloca_read (FPA_FILE *fpa_file, unsigned long long bloco)

  *Procura por um determinado buffer em uma lista de buffers de um arquivo. Caso o buffer não exista, este será alocado e colocado na lista. O buffer é alocado como sendo de apenas leitura.*

- struct ARQUIVO_ABERTO ∗ busca_arquivo_liberar_buffer (void)

  *Procura um arquivo que tenha buffers para serem liberados. Caso exista mais de um arquivo com buffers não utilizados, é retornado aquele que tem o maior número de buffer não utilizados no momento.*

- struct FPA_BUFFER ∗ retira_buffer_arquivo (struct ARQUIVO_ABERTO ∗arquivo)

  *Retira um buffer da lista de buffers do arquivo. Retira um buffer da lista de buffers do arquivo. A lista de buffers do arquivo é atualizada, já sem o buffer.*

- struct FPA_BUFFER ∗ retira_todos_os_buffers_livres (struct ARQUIVO_-ABERTO ∗arquivo)

  *Retira todos os buffers não utilizados de um arquivo de sua lista de buffers. Retira todos os buffers não utilizados de um arquivo da sua lista de buffers. A lista de buffers do arquivo é atualizada, já sem os buffers.*

## Variáveis

- char _modo

  *Modo de liberação dos buffers. '1' para liberação de 1 buffer por vez e 't' para liberação de todos os buffers não utilizados de uma única vez.*

- char _tecnica

  *Técnica utilizada para o mapeamento dos buffers. 'c' para o uso de calloc e 'm' para o uso de mmap. Calloc é o valor default.*

- struct LISTA_ARQUIVOS lista_arquivos

  *Lista de arquivos abertos.*

- unsigned long _tamanho_buffer

  *Tamanho de cada buffer, em bytes. Default: 32 MB.*

- unsigned long _memoria_usada

  *Contador da quantidade de memória utilizada pelos buffers. Valor inicial 0, e este valor sempre é múltiplo do tamanho de buffers.*

- sem_t sem_acesso_disco

  *Semáforo para controlar o acesso ao disco.*

- sem_t sem_aviso_memoria

  *Semáforo para controlar quando os dados devem ser enviados ao disco. A thread de escrita aguarda dormindo por este semáforo.*

- unsigned long _aviso_memoria

  *Quantidade de buffers mantidos em memória, sem necessidade de enviar dados ao disco. Default: 128 MB.*

- unsigned long chamadas

  *Contador de requisições para novos buffers, quando o máximo de memória for atingido.*

### 4.4.1 Descrição Detalhada

Cabeçalhos das rotinas de manipulação de listas da biblioteca FPA.

### 4.4.2 Funções

#### 4.4.2.1 struct ARQUIVO_ABERTO∗ busca_arquivo_liberar_buffer (void) `[read]`

Procura um arquivo que tenha buffers para serem liberados. Caso exista mais de um arquivo com buffers não utilizados, é retornado aquele que tem o maior número de buffer não utilizados no momento.

**Retorna:**

Ponteiro para um elemento na lista de arquivos abertos.

#### 4.4.2.2 struct ARQUIVO_ABERTO∗ lista_arquivos_buscar (char *nome_arquivo*[PATH_MAX+1], char *modo*) `[read]`

Procura por um arquivo na lista de arquivos abertos.

**Parâmetros:**

***nome_arquivo*** Nome do arquivo

***modo*** Modo de acesso. 'r' para leitura, 'w' para escrita e 'b' para leitura e escrita

**Retorna:**

Ponteiro para a estrutura do arquivo aberto. Caso o arquivo não exista, NULL é retornado.

#### 4.4.2.3 int lista_arquivos_excluir (struct ARQUIVO_ABERTO ∗ *arquivo*)

Envia as informações armazenadas nos buffers do arquivo ao disco, libera a memória utilizada nos buffers do arquivo e exclui o arquivo da lista de arquivos abertos.

**Parâmetros:**

***Estrutura*** de arquivo aberto.

**Retorna:**

0 para ok, e -1 caso ocorra algum erro no processo.

#### 4.4.2.4 int lista_arquivos_flush (struct ARQUIVO_ABERTO ∗ *arquivo*)

atualiza os dados no disco com as informações armazenadas nos buffers.

**Parâmetros:**

***arquivo*** Estrutura do arquivo aberto.

**Retorna:**

0 para ok, e -1 caso ocorra algum problema no processo.

#### 4.4.2.5 struct ARQUIVO_ABERTO∗ lista_arquivos_incluir_read (char *nome_arquivo*[PATH_MAX+1]) `[read]`

Inclui um novo arquivo para leitura na lista de arquivo abertos.

**Parâmetros:**

***nome_arquivo*** Nome do arquivo.

**Retorna:**

Ponteiro para a estrutura do novo arquivo.

#### 4.4.2.6 struct ARQUIVO_ABERTO∗ lista_arquivos_incluir_write (char *nome_arquivo*[PATH_MAX+1]) `[read]`

Inclui um novo arquivo para escrita na lista de arquivo abertos.

**Parâmetros:**

***nome_arquivo*** Nome do arquivo.

**Retorna:**

Ponteiro para a estrutura do novo arquivo.

#### 4.4.2.7 struct FPA_BUFFER∗ lista_buffers_aloca_read (FPA_FILE ∗ *fpa_file*, unsigned long long *bloco*) `[read]`

Aloca um novo buffer de apenas leitura na lista de buffers.

**Parâmetros:**

*fpa_file* Arquivo aberto.

*bloco* Região do arquivo, referente ao buffer alocado.

**Retorna:**

Ponteiro para o buffer, e -1 caso ocorra algum erro no processo.

#### 4.4.2.8 struct FPA_BUFFER* lista_buffers_busca_aloca_read (FPA_FILE * *fpa_file*, unsigned long long *bloco*) `[read]`

Procura por um determinado buffer em uma lista de buffers de um arquivo. Caso o buffer não exista, este será alocado e colocado na lista. O buffer é alocado como sendo de apenas leitura.

**Parâmetros:**

*fpa_file* Arquivo aberto.

*bloco* Região do arquivo, referente ao buffer alocado.

**Retorna:**

Ponteiro para o buffer, e -1 caso ocorra algum erro no processo.

#### 4.4.2.9 struct FPA_BUFFER* lista_buffers_busca_aloca_write (FPA_FILE * *fpa_file*, unsigned long long *bloco*) `[read]`

Procura por um determinado buffer em uma lista de buffers de um arquivo. Caso o buffer não exista, este será alocado e colocado na lista. O buffer é alocado como sendo de leitura e escrita.

**Parâmetros:**

*fpa_file* Arquivo aberto.

*bloco* Região do arquivo, referente ao buffer alocado.

**Retorna:**

Ponteiro para o buffer, e -1 caso ocorra algum erro no processo.

#### 4.4.2.10 struct FPA_BUFFER* lista_buffers_busca_read (FPA_FILE * *fpa_file*, unsigned long long *bloco*) `[read]`

Procura por um determinado buffer em uma lista de buffers de um arquivo.

**Parâmetros:**

*fpa_file* Arquivo aberto.

*bloco* Região do arquivo, referente ao buffer alocado. return Ponteiro para o buffer, e NULL caso o buffer não exista.

#### 4.4.2.11 struct FPA_BUFFER∗ retira_buffer_arquivo (struct ARQUIVO_ABERTO ∗ *arquivo*) `[read]`

Retira um buffer da lista de buffers do arquivo. Retira um buffer da lista de buffers do arquivo. A lista de buffers do arquivo é atualizada, já sem o buffer.

**Parâmetros:**

***Arquivo*** Estrutura de arquivo FPA_FILE.

**Retorna:**

Ponteiro para a buffer liberado.

#### 4.4.2.12 struct FPA_BUFFER∗ retira_todos_os_buffers_livres (struct ARQUIVO_ABERTO ∗ *arquivo*) `[read]`

Retira todos os buffers não utilizados de um arquivo de sua lista de buffers. Retira todos os buffers não utilizados de um arquivo da sua lista de buffers. A lista de buffers do arquivo é atualizada, já sem os buffers.

**Parâmetros:**

***Arquivo*** Estrutura de arquivo FPA_FILE.

**Retorna:**

Ponteiro para uma lista de buffers, contendo os buffers liberados.

# Índice Remissivo


ARQUIVO_ABERTO, 5

busca_arquivo_liberar_buffer
- listas.c, 25
- listas.h, 31

fpa.c, 9
- fpa_close, 12
- fpa_eof, 12
- fpa_flush, 12
- fpa_fread, 12
- fpa_fseek, 13
- fpa_fwrite, 13
- fpa_lseek, 14
- fpa_open, 14
- fpa_read, 14
- fpa_start, 15
- fpa_tell, 15
- fpa_write, 15
- thread_write, 15

fpa.h, 17
- fpa_close, 19
- fpa_eof, 19
- fpa_flush, 19
- fpa_fread, 20
- fpa_fseek, 20
- fpa_fwrite, 20
- fpa_lseek, 21
- fpa_open, 21
- fpa_read, 21
- FPA_SEEK_CUR, 19
- FPA_SEEK_END, 19
- FPA_SEEK_SET, 19
- fpa_start, 22
- fpa_tell, 22
- fpa_write, 23

FPA_BUFFER, 6

fpa_close
- fpa.c, 12
- fpa.h, 19

fpa_eof
- fpa.c, 12
- fpa.h, 19

fpa_flush
- fpa.c, 12
- fpa.h, 19

fpa_fread
- fpa.c, 12
- fpa.h, 20

fpa_fseek
- fpa.c, 13
- fpa.h, 20

fpa_fwrite
- fpa.c, 13
- fpa.h, 20

fpa_lseek
- fpa.c, 14
- fpa.h, 21

fpa_open
- fpa.c, 14
- fpa.h, 21

fpa_read
- fpa.c, 14
- fpa.h, 21

FPA_SEEK_CUR
- fpa.h, 19

FPA_SEEK_END
- fpa.h, 19

FPA_SEEK_SET
- fpa.h, 19

fpa_start
- fpa.c, 15
- fpa.h, 22

fpa_tell
- fpa.c, 15
- fpa.h, 22

fpa_write
- fpa.c, 15
- fpa.h, 23

LISTA_ARQUIVOS, 7

lista_arquivos_buscar
listas.c, 25
listas.h, 31
lista_arquivos_excluir
listas.c, 26
listas.h, 31
lista_arquivos_flush
listas.c, 26
listas.h, 32
lista_arquivos_incluir_read
listas.c, 26
listas.h, 32
lista_arquivos_incluir_write
listas.c, 26
listas.h, 32
lista_buffers_aloca_read
listas.c, 27
listas.h, 32
lista_buffers_busca_aloca_read
listas.h, 33
lista_buffers_busca_aloca_write
listas.c, 27
listas.h, 33
lista_buffers_busca_read
listas.c, 27
listas.h, 33
listas.c, 24
busca_arquivo_liberar_buffer, 25
lista_arquivos_buscar, 25
lista_arquivos_excluir, 26
lista_arquivos_flush, 26
lista_arquivos_incluir_read, 26
lista_arquivos_incluir_write, 26
lista_buffers_aloca_read, 27
lista_buffers_busca_aloca_write, 27
lista_buffers_busca_read, 27
retira_buffer_arquivo, 27
retira_todos_os_buffers_livres, 28
listas.h, 29
busca_arquivo_liberar_buffer, 31
lista_arquivos_buscar, 31
lista_arquivos_excluir, 31
lista_arquivos_flush, 32
lista_arquivos_incluir_read, 32
lista_arquivos_incluir_write, 32
lista_buffers_aloca_read, 32
lista_buffers_busca_aloca_read, 33
lista_buffers_busca_aloca_write, 33
lista_buffers_busca_read, 33
retira_buffer_arquivo, 33
retira_todos_os_buffers_livres, 34

retira_buffer_arquivo
listas.c, 27
listas.h, 33
retira_todos_os_buffers_livres
listas.c, 28
listas.h, 34

ST_FPA_FILE, 8

thread_write
fpa.c, 15