Acesso Paralelo a Arquivos em Sistemas Multiprocessadores

Darlon Vasata[1], Liria Matsumoto Sato[1]

Escola Politécnica da Universidade de São Paulo (EPUSP)
Avenida Prof. Luciano Gualberto, travessa 3 nº 380
CEP 05508-970 – São Paulo – SP – Brasil
{darlon.vasata,liria.sato}@poli.usp.br

Abstract This paper presents a study about the bottleneck of hard disk access on systems under multiprocessor architectures. It also shows a specification for a solution to the problem and its implementation. The strategy described takes advantage of the shared memory architecture resources and use data buffers between the different threads of the applications. It also tries to reduce the hard disk header movimentation allowing the disk access by only one thread. The implementation maps certain regions of the file on memory and send them to the disk at once. Thus it avoids unnecessary system calls and reduce the operation system's overhead. Tests were executed in different situations variating the file size and considering processing on the applications. The results shown performance gain for applications with considerable processing quantities and applications with have files processed in blocks as a stack.

1 Introdução

Diversos problemas do mundo real que demandam e processam grandes quantidades de informações são inviáveis manualmente 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.

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. Um exemplo disto é o disco rígido, e mesmo 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 [1].

Atualmente há 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 [2].

O alavancamento de todo o poder computacional disponível pelos *hardwares* com multiprocessadores demanda novas ferramentas e novas formas de pensar da indústria de *software* [3]. Tal *hardware* é melhor aproveitado através da paralelização de código, que consiste em identificar regiões da aplicação que podem ser executadas simultaneamente [4].

O fato das arquiteturas com multiprocessadores permitirem que diversas linhas de execução (*threads*) operem simultaneamente, proporciona um melhor desempenho de 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 e acessam e armazenam seus dados a partir de arquivos, torna-se evidente 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 significativamente reduzir o desempenho de aplicações, se estas necessitarem acessar informações que residam em pontos distantes entre si no disco.

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 devido a problemas de acesso a disco, o atual trabalho objetiva especificar e implementar uma biblioteca denominada FPA (*File for Parallel Access*), de forma que esta forneça um mecanismo que auxilie na diminuição do tempo gasto no acesso de leitura ou escrita a dados no disco rígido em arquiteturas com múltiplos processadores.

Na literatura são encontrados alguns trabalhos relacionados, como o de [5] que propõe uma nova forma de escalonamento de disco, onde apresenta uma forma de análise dos acessos ao sistema de armazenamento. Seu objetivo é determinar o melhor algoritmo de escalonamento de disco para o processo, alternando e modificando a forma de escalonamento conforme seja necessário para obter um melhor desempenho no sistema. A pesquisa de [6] apresenta um estudo sobre a evolução da computação e o problema do gargalo do disco, voltando-se para aplicações em tempo real. Inclusive aborda informações sobre a complexidade computacional do acesso e desenvolve uma heurística para atualização de dados e particionamento de disco, de forma que o desempenho da aplicação baseada em disco seja otimizada.

A preocupação com desempenho de aplicações em sistemas paralelos também é tratada em outros trabalhos. Em [7] é apresentado um estudo sobre aplicações distribuídas com MPI e Pthreads, onde estas utilizam diversas formas de escrita de dados nos discos locais. Seu trabalho constatou que utilizando apenas MPI é conseguido um melhor desempenho com o uso de Mmap, enquanto chegou a um

melhor desempenho com Pthreads através da escrita com a chamada de sistema `write`.

Em [8] é apontado o problema do gargalo no acesso a disco para aplicações paralelas e distribuídas, destacando-o como o ponto mais crítico para aplicações envolvendo grandes quantidade de I/O.

O trabalho de [9] apresenta uma biblioteca para acesso a arquivos em sistemas distribuídos utilizando *threads* para fazer a distinção de fluxo de aplicação e gerenciamento de escritas a disco, porém com seu foco voltado para sistemas distribuídos com sistemas de arquivos paralelos.

O presente trabalho está organizado da seguinte forma: a seção 2 apresenta as formas de acesso a arquivos tradicionais e a técnica utilizada no trabalho, enquanto a seção 3 trata da estratégia desenvolvida. Aspectos sobre a implementação da biblioteca são tratados na seção 4 e seus resultados são vistos na seção 5. A seção 6 apresenta as conclusões.

2 Acesso a arquivos

As aplicações nos sistemas Unix podem realizar escrita a disco de duas formas. Em uma, os arquivos podem ser criados/abertos com `fopen`, e o I/O é executado através da biblioteca padrão C (`fread`, `fwrite`). A segunda forma, é a criação/abertura de arquivos com a chamada de sistema `open`, e o I/O é realizado utilizando chamadas diretamente ao sistema de armazenamento (`read`, `write`) [10].

Além das operações de acesso a disco padrões, com `open` ou `fopen`, os sistemas Unix disponibilizam uma chamada de sistema denominada `mmap`, onde esta possibilita que seja criada uma correspondência de um-para-um entre um arquivo em disco e uma região de memória. O programador pode então acessar o arquivo diretamente pela memória, possibilitando também que tais informações sejam transmitidas ao disco de forma transparente.

Segundo [11] a utilização do Mmap possui diversas vantagens, como:

1. possibilita que diversos processos compartilhem os mesmos dados em memória;
2. 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;
3. o *seek* para posições do arquivo é feito apenas com simples operações de ponteiros, descartando o uso da chamada de sistema `lseek`.

Além destes, o mesmo trabalho também aponta diversos pontos fracos do `mmap`:

1. o mapeamento em memória envolve sempre um número inteiro de páginas, ocorrendo casos em que possa ser alocado memória desnecessariamente;
2. existe um *overhead* na criação e manutenção do arquivo mapeado e das estruturas de dados dentro do *kernel* do sistema operacional. Desta forma, o Mmap é recomendado para casos onde o arquivo possua um tamanho consideravelmente grande, e dessa forma a memória desperdiçada pelo Mmap torna-se irrelevante perante o tamanho do arquivo.

Outra desvantagem do Mmap é seu tratamento com arquivos consideravelmente grandes, com tamanhos que excedem a capacidade da memória do sistema. Nestes casos, Mmap utiliza o mecanismo de memória virtual. Testes indicaram que nessas situações, o uso do Mmap pode levar a um desempenho muito menor do que com simples operações de acesso a disco, como `read` e `write`.

3 Estratégia de otimização no acesso a disco

Além de proporcionar diversas linhas de processamento simultâneo, uma das principais características das arquiteturas com multiprocessadores é possibilitar que as diversas destas utilizem o recurso de memória compartilhada. Este mecanismo proporciona uma maior facilidade na comunicação entre as *threads*, bem como evita que sejam feitas chamadas para comunicação entre as mesmas. Aproveitando este recurso, FPA proporciona que as informações dos arquivos residam nesta região compartilhada, e diversas *threads* tenham acesso à informação sem que seja necessário uma chamada ao disco a cada momento de uso do arquivo.

Os arquivos são mapeados em blocos para um *buffer* de memória, onde cada *buffer* é responsável por um trecho do arquivo. A figura 1 representa esta situação, em que cada *thread* acessa um *buffer* do arquivo.

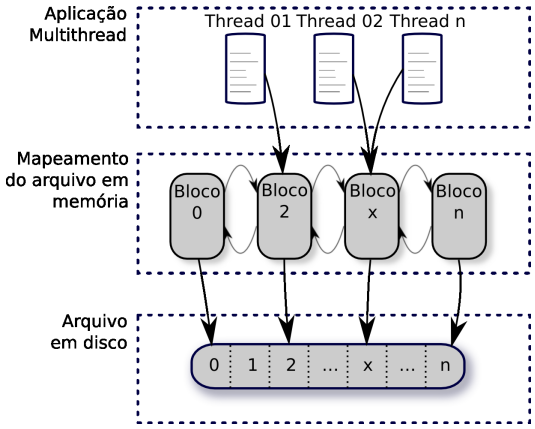


Figura 1. Acesso e mapeamento de arquivos em memória

Para o caso em que mais de uma *thread* acessar uma mesma região (ou bloco) do arquivo, este bloco será compartilhado entre as mesmas. Outra vantagem da utilização de *buffers* para acesso a arquivos, é que estes podem estar organizados em forma de listas. Se um determinado bloco não estiver mais sendo acessado e um limite arbitrário de uso de memória for alcançado, este *buffer* poderá ser excluído da lista e liberado da memória. Assim, será liberado espaço para um outro bloco e evitará que a memória ultrapasse seus limites, descartando a necessidade de uso da memória virtual.

O uso de *buffers* de dados em memória, além da vantagem da velocidade de acesso, possibilita que no momento que um determinado *buffer* for liberado, seja executada uma única chamada de sistema para enviar todo o *buffer* ao disco. Esta prática diminui o *overhead* do sistema operacional, levando a um melhor desempenho do sistema.

Um cuidado a ser tomado é evitar que o disco seja acessado por mais de uma *thread* durante uma mesma requisição de leitura ou escrita. Testes realizados por [7] indicam perda de desempenho enquanto diversas solicitações são feitas, devido à tentativa do disco atender todas as requisições e efetuar uma maior quantidade de movimentação do cabeçote de leitura e gravação.

4 Implementação

A biblioteca FPA busca a solução para o problema da concorrência sem alterações na arquitetura ou *hardware* adicional. Toda a estratégia é abordada via *software*, possibilitando que esta possa ser utilizada em quaisquer aplicações implementadas utilizando a linguagem de programação C, ambiente Linux e desenvolvimento *multithtread*.

A implementação visa utilizar algumas estratégias de otimização de acesso a disco, fornecendo ao usuário da biblioteca uma interface semelhante a rotinas comuns de acesso a arquivos.

Os *buffers* em memória são implementados em forma de lista, onde cada elemento da lista contêm a região com os dados do arquivo, qual região do arquivo aqueles dados representam e quais são as *threads* que utilizam aquele *buffer*. Tais *buffers* não precisam necessariamente mapear regiões contíguas do arquivo, bastando apenas aquelas regiões realmente utilizadas. A figura 1 ilustra este cenário, onde o primeiro *buffer* possui os dados do bloco 0 e o segundo *buffer* possui os dados do bloco 2.

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 serem enviadas ao disco.

Evitando que as *threads* acessem o disco diretamente e assim agravando a concorrência ao disco, é criada uma *thread* única para acesso ao disco, como ilustrado na figura 2. Todas as chamadas ao disco são efetuadas por esta *thread*, que utilizando um controle com semáforos bloqueia múltiplas requisições de acesso ao sistema de armazenamento. Objetivando um melhor desempenho durante todo o processo de leitura e escrita, não são feitas cópias dos dados para outras regiões de memória. Todas as *threads* acessam os mesmos endereços de memória.

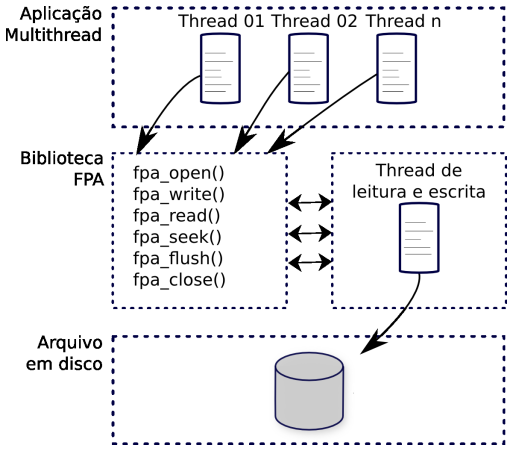


Figura 2. Acesso e mapeamento de arquivos em memória

5 Resultados e Análise

Os testes efetuados foram executados sobre uma máquina com as configurações definidas na tabela 1:

Processador	Dual Quadcore Intel (R) Xeon(R) CPU
Clock	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 9.1 20070215
Ferramenta de Paralelismo	OpenMp

Tabela 1. Configurações do ambiente de testes

Vários testes foram executados, com o intuito de encontrar os casos e situações onde o uso da FPA obtenha o melhor desempenho. Tais testes envolvem diferentes tamanhos de arquivo, número de *threads*, processamento da aplicação e método de particionamento do arquivo. Os testes efetuados realizam comparações entre o uso da FPA e o acesso utilizando Fopen. Os tempos utilizados nos gráficos foram obtidos através de 10 execuções de cada teste, onde foram descartados os 2 maiores e menores tempos. O valor médio dos 6 tempos restantes é utilizada nos gráficos.

A implementação da FPA é realizada utilizando duas versões. Uma delas conta com a alocação dos *buffers* e o respectivo mapeamento do arquivo utilizando a chamada de sistema `mmap`, enquanto a outra versão realiza a alocação dos *buffers* com a chamada `calloc`. Tais versões são denominadas FPA Mmap e FPA Calloc, respectivamente.

A principal diferença entre as versões testadas da FPA é que em FPA Mmap, o sistema operacional conhece qual é a região do arquivo mapeada em memória, e pode vir a enviar dados ao disco de acordo com sua necessidade. Tal atitude pode ocasionar a escrita em disco em momentos indesejáveis, podendo levar a excessivas movimentações do cabeçote do disco. Já em FPA Calloc, tal interferência do sistema operacional não acontece, e o envio dos dados ao disco é realizado apenas em momentos definidos pela biblioteca FPA.

Para os resultados apresentados nas figuras 3, 5(a) e 5(b), o formato de escrita das diversas *threads* no arquivo é realizado utilizando o particionamento do arquivo em tamanhos iguais. Desta forma, cada *thread* é responsável pela escrita sequencial em uma determinada parte do arquivo.

O resultado apresentado na figura 3 mostra a simples utilização das diferentes técnicas, para a escrita de arquivos de 1 gigabyte. Nesta situação, a aplicação com Fopen obteve o melhor desempenho. A partir disto, percebe-se que os cálculos realizados para o gerenciamento dos *buffers* interferem no desempenho da aplicação que utiliza FPA.

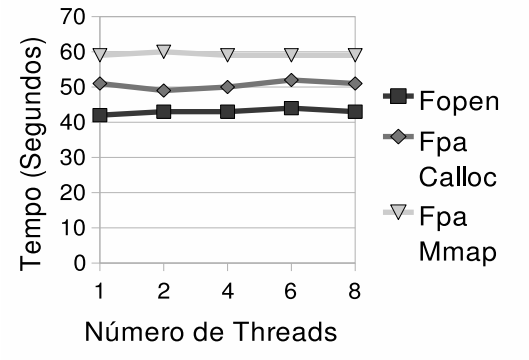


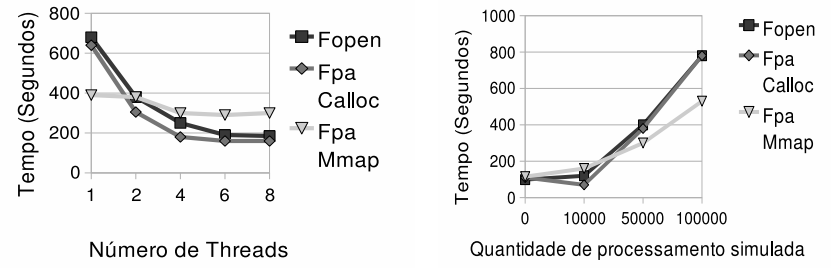
Figura 3. Teste sem processamento adicional com arquivo de 1 gigabyte

Outra situação analisada é quando a aplicação possui uma quantidade considerável de processamento. A simulação de tal processamento consiste na inserção do código mostrado na figura 4 nas aplicações, e o parâmetro `processamento` é variado, realizando uma diferente quantidade de cálculos. Todos os programas são compilados dispensando otimizações do compilador, impedindo que diferentes tempos de execução sejam realizados entre as aplicações.

```
1  for( j=0; j < 1024/nthreads; j++ ){
2          for( i =0; i < 1024 * gigabytes; i++ ){
3                  for( x =1; x <= processamento; x++ )
4                          X = X*x/1023;
5                  Y=X;
6                  //Rotina de escrita testada
7          }
8  }
```

Figura 4. Cálculos adicionais executados nos testes

Utilizando processamento na aplicação, FPA obteve melhor desempenho em relação à Fopen. Tal desempenho acontece devido à utilização da *thread* responsável pela escrita das informações, pois assim as diversas *threads* da aplicação podem apenas escrever os dados do arquivo nos *buffers* da biblioteca e continuar seu processamento normal. Tal resultado pode ser visto na figura 5(a), onde nesta é utilizado em `processamento` o valor de 10000 e é realizado a escrita de um arquivo de 4 gigabytes.



(a) Arquivos de 4 gigabytes e processamento de 10000

(b) Arquivos de 2 gigabytes e 4 *threads*, variando a quantidade de processamento da aplicação

Figura 5. Testes com diferentes números de *threads* e diferentes quantidades de processamento

Outro aspecto analisado é o comportamento da aplicação para os diversos valores de simulação de processamento, levando em conta o tamanho do arquivo. A relevância de tais testes deve-se a que a *thread* de acesso pode escrever as informações durante o processamento da aplicação, influenciando no desempenho desta, assim como a quantidade de informações escritas também possui relação direta com tal resultado. A figura 5(b) mostra os tempos gastos para testes utilizando 4 *threads* para a escrita de arquivos de 2 gigabytes e diversos valores de simulação de processamento. É possível perceber que para casos em que a

aplicação possui uma quantidade de processamento considerável, o uso da FPA torna-se vantajoso.

Outro teste realizado considera os blocos do arquivo como uma pilha. Desta forma, cada *thread* escreve no primeiro bloco não escrito do arquivo e quando tal escrita for terminada, escreve no próximo bloco disponível e assim sucessivamente, até que o arquivo esteja escrito por completo. O resultado de tal teste com arquivo de 12 gigabytes e `processamento` de 10000 pode ser observado na figura 6. Neste cenário, cada bloco de arquivo considerado possui um tamanho de 16 megabytes.

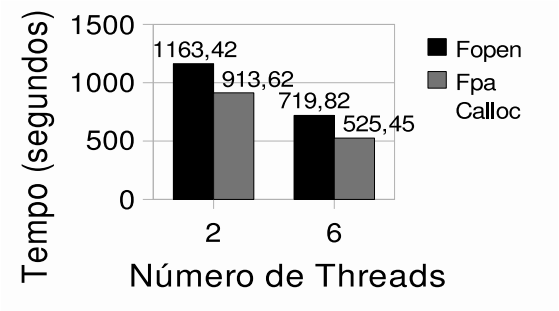


Figura 6. Testes por pilha de blocos, com arquivo de 12 gigabytes e processamento de 10000

Uma situação percebida, é que quanto maior o número de *threads*, menor é a diferença de tempo entre as aplicações. Tal situação acontece devido à paralelização do processamento simulado. Tais gráficos também mostram que a aplicação utilizando Fopen possui um desempenho inferior à FPA. Tal resultado é decorrente da maior movimentação do cabeçote de escrita no disco pela aplicação com Fopen, devido ao acesso concorrente ao disco pelas diversas *threads*. Esta situação não acontece em FPA, pois esta possui o controle dessa situação devido à *thread* de escrita em disco.

6 Conclusão

Através dos resultados torna-se possível compreender 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 tal solução possui um custo computacional com o gerenciamento dos *buffers* das informações e em alguns casos a solução proposta não garante desempenhos melhores do que com soluções utilizando abordagens tradicionais, como o uso da biblioteca `fopen`.

A presente implementação ainda encontra-se em estágio de desenvolvimento, de forma que testes realizando leituras de arquivos ainda serão desenvolvidos.

Com base nos testes atuais verificou-se que a estratégia é vantajosa para alguns casos, como em implementações que possuem quantidades de processamento consideráveis. Isto deve-se ao fato de que a existência de uma *thread* responsável pelo envio dos dados ao disco possibilita às aplicações transferirem seus dados para regiões de memória e posteriormente serem enviados ao disco. A transferência de informações em memória é diversas vezes mais rápida que a transferência ao disco, permitindo às aplicações continuarem seu processamento enquanto a *thread* da biblioteca FPA envia as informações ao disco.

Além do custo computacional de gerenciamento dos *buffers*, existe o custo pela *thread* de escrita. Desta forma, o desempenho obtido nos testes onde o número de *threads* era equivalente ao número de processadores disponíveis não obteve grande vantagem, pois nestes casos a *thread* de escrita passa a ocupar CPU das *threads* da aplicação.

A estratégia mostra-se vantajosa para os casos em que existe uma quantia considerável de processamento pela aplicação, e o arquivo escrito em disco é grande o suficiente para que os *buffers* do sistema operacional não suportem o arquivo por completo, gerando assim para o Fopen uma escrita desordenada das informações no disco.

Referências

1. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann (2003)
2. Culler, E.D., Singh, J.P.: Parallel Computer Architecture - A Hardware/Software Approach. Morgan Kaufmann Publishers, Inc. (1999)
3. Sutter, H., Larus, J.: Software and the concurrency revolution. Queue **3**(7) (2005) 54–62
4. Moretti, C., Bittencourt, T.: Aspectos Gerais da Computação Paralela e do Sistema PVM (2004)
5. Martens, D., Katchabaw, M.: Disk access analysis for system performance optimization. 5th WSEAS International Conference on Applied Computer Science (2006)
6. Docy, S., Hong, I., Potkonjak, M.: Throughput optimization in disk-based real-time applicationspecific systems. In: System Synthesis, 1996. Proceedings., 9th International Symposium on. (1996) 133–138
7. Benson, G., Long, K., Pacheco, P.: The Performance of Parallel Disk Write Methods for Linux Multiprocessor Nodes. Lecture notes in computer science (2003) 71–80
8. Thakur, R., Lusk, E., Gropp, W.: I/O in parallel applications: The weakest link. International Journal of High Performance Computing Applications **12**(4) (1998) 389
9. More, S., Choudhary, A., Foster, I., Xu, M.: MTIO. A multi-threaded parallel I/O system. In: Parallel Processing Symposium, 1997. Proceedings., 11th International. (1997) 368–373
10. Newman, H.: Application Performance and I/O. SW Expert. (April 2001)
11. Love, R.: Linux system programming. O'Reilly Media, Inc. (2007)