

UNIVERSIDADE DO MINHO  
ESCOLA DE ENGENHARIA  
DEPARTAMENTO DE INFORMÁTICA

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA - 4º ANO  
ENGENHARIA DE SISTEMAS DA COMPUTAÇÃO

---

**Portfólio de Trabalhos Práticos**

---

*Professor:*  
António Pina  
Departamento de Informática  
Computação Paralela e Distribuída

*Aluno:*  
Sérgio Caldas  
A57779

## Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>NAS Parallel Benchmark</b>	<b>4</b>
<b>3</b>	<b>PThreads</b>	<b>18</b>
<b>4</b>	<b>DTrace</b>	<b>23</b>
<b>5</b>	<b>DTrace e IOzone</b>	<b>31</b>
<b>6</b>	<b>Tutorial Perf</b>	<b>38</b>
<b>7</b>	<b>Conclusão</b>	<b>51</b>

# 1 Introdução

A disciplina de Engenharia de Sistemas da Computação (ESC) insere-se no perfil de mestrado de Computação Paralela e Distribuída do 4º Ano do curso do Mestrado Integrado em Engenharia Informática. A disciplina é lecionada ao longo de todo o 2º Semestre, sendo que a metodologia de avaliação desta se centra essencialmente na realização de um teste teórico e no desenvolvimento de trabalhos prático ao longo do semestre, com o objetivo de por em prática a matéria lecionada ao longo das aulas.

A matéria lecionada ao longo do semestre, nesta disciplina, centra-se essencialmente na análise de desempenho e monitorização de ambientes linux. De maneira a podermos fazer esta análise e monitorização, foi-nos apresentado um conjunto de ferramentas capazes de o fazer de uma maneira bastante pormenorizada. As ferramentas em questão são: Dtrace, Perf, strace, truss entre outras, sendo que a mais lecionada foi a ferramenta Dtrace. Esta ferramenta é uma ferramenta que permite o desenvolvimento de scripts, que nos permitem fazer traçados dinâmicos do ambiente de estudo, esta ferramenta é bastante poderosa uma vez que permite o desenvolvimento de scripts focadas naquilo que queremos observar.

Esta disciplina para além de nos preparar para a análise de desempenho e monitorização em ambientes Linux, também nos apresenta um bosto conjunto de conceitos essenciais nos Sistemas de Computação, conceitos como: sistemas distribuidos (grids, clusters, e cloud), programação em memória partilhada com recurso a pthreads, sistemas de ficheiros, CPU's entre outros tópico teóricos.

Neste portfólio são apresentados todos os trabalhos práticos desenvolvidos por mim ao longo do semestre. Trabalhos que tinham como objetivo por em prática toda a matéria referida atrás. Ao longo do semestre foram sugeridos 5 trabalhos práticos:

1. **NAS Parallel Benchmark** - O objetivo deste trabalho era testar e analisar uma gama de aplicações desenvolvidas em vários paradigmas (Sequêncial, Memória Partilhada e Memória Distribuída). No desenvolvimento deste trabalho efetuei uma basta gama de testes, no Cluster Search6, os testes foram feitos para diferentes compiladores, diferentes flags de compilação, diferentes máquinas entre outros fatores. Este trabalho permitiu-me adquirir várias competências tais como a automatização dos testes, análise e comparação de vários testes ao mesmo tempo bem como o tratamento dos dados obtidos.
2. **Programação em Memória Partilhada com Pthreads** - Este trabalho prático tinha como objetivo o desenvolvimento de uma aplicação num paradigma de memória distribuída recorrendo a POSIX Threads. Para isso desenvolvi uma aplicação, em que estava implementada a regra trapezoidal recorrendo a três tipos de exclusão mútua (mutex, semáforos, e busy wait). Este trabalho permitiu-me aperfeiçoar a minha habilidade no desenvolvimento de aplicações num paradigma de memória distribuída para além disso permitiu-me analisar o comportamento dos vários tipos de exclusão mutua bem como qual a melhor técnica a utilizar.
3. **DTrace** - Este trabalho tinha como objetivo exercitar o uso da ferramenta Dtrace, ferramenta essa que como foi referido anteriormente nos permite desenvolver scripts que nos permitem fazer traçados dinâmicos de uma aplicação. Para o desenvolvimento deste trabalho tive de estudar e praticar o uso da linguagem D (linguagem utilizada pelo DTrace). As scripts desenvolvidas fazem um traçado das chamadas ao sistema `openat()` o PID do processo, o UID do utilizador, o GID do grupo o caminho absoluto para o ficheiro que for aberto, a cadeia de caracteres com as flags da chamada ao sistema `openat()` (`O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, `O_CREAT`) e por fim o valor de retorno de chamada ao sistema. Este trabalho permitiu-me ganhar prática no desenvolvimento de scripts em D bem como posteriormente fazer uma análise detalhada do traçado obtido.
4. **DTrace e IOzone** - O desenvolvimento deste trabalho prático teve como objetivo a análise do Benchmark IOZone, benchmark este que foi desenvolvido com o objetivo de fazer uma gama de testes de desempenho de filesystems. Para fazer uma análise detalhada do Benchmark, recorri ao DTrace, e desenvolvi scripts que fazem traçados dinâmicos do comportamento desta aplicação para os testes definidos. Os testes por mim definidos foram testes que me permitiram analisar as operações de `read`, `re-read`, `write` e `re-write` para dois sistemas de ficheiros (ZFS e UFZ). Mais uma vez com este trabalho desenvolvi ainda mais as minhas competências na criação de scripts em D, bem como me permitiu analisar e compreender de uma maneira mais objetiva o comportamento do filesystem em estudo.
5. **Tutorial Perf** - Este trabalho, focou-se essencialmente no seguimento de um tutorial Perf,sugerido pelo professor. O Perf é uma ferramenta de análise de performance desenvolvida para LINUX, esta ferramenta é acessível através da linha de comandos e fornece uma gama de sub-comandos bem como uma basta gama de contadores, tanto de hardware como de software. Estes comandos permitem fazer uma análise estatística

de todo o sistema, quer ao nível do Kernel quer ao nível do utilizador. O tutorial em questão permitiu-me estudar uma forma para detectar pontos quentes de uma aplicação em execução, a utilização de contadores de hardware e ainda traçar perfis de eventos de hardware. Este tutorial permitiu-me desenvolver bastantes competências na utilização da ferramenta Perf e ainda ter uma percepção da potencialidade desta ferramenta.

## 2 NAS Parallel Benchmark

{Página em Branco (Fazer scroll)}

# Introdução ao NAS Parallel Benchmarks (NPB)

## Testes de Referência Versões: Sequencial, Memória Partilhada/Memória Distribuída

### Ambiente de Operação no Cluster Search

Sérgio Caldas  
*Universidade do Minho*  
*Escola de Engenharia*  
*Departamento de Informática*  
*Email: a57779@alunos.uminho.pt*

## Conteúdo

<b>1</b>	<b>Introdução</b>	1
<b>2</b>	<b>Caracterização do Ambiente de Testes</b>	2
2.1	Nodos de Teste . . . . .	2
2.2	Compiladores Utilizados . . . . .	2
<b>3</b>	<b>Caracterização do Benchmark</b>	2
3.1	Escolha de Kernels e Classes de Dados	3
<b>4</b>	<b>Testes Efectuados</b>	3
4.1	Versão Sequencial . . . . .	3
4.2	Versão OMP . . . . .	3
4.3	Versão MPI . . . . .	4
<b>5</b>	<b>Análise de Resultados</b>	4
5.1	Versão Sequencial . . . . .	4
5.2	Versão OMP . . . . .	5
5.3	Versão MPI . . . . .	7
<b>6</b>	<b>Ganhos</b>	8
6.1	Sequencial vs OMP . . . . .	8
6.2	Sequencial vs MPI . . . . .	8
<b>7</b>	<b>Utilitários de Monitorização</b>	8
7.1	Versão Sequencial . . . . .	8
7.2	Versão OMP . . . . .	9
7.3	Versão MPI . . . . .	10
7.4	Análise Geral . . . . .	11
<b>8</b>	<b>Conclusão</b>	11
<b>Referências</b>		11
<b>Apêndice</b>		12

**Resumo**—O *NAS Parallel Benchmark* é um ambiente de testes desenvolvido pela NASA, para medir a performance de super-computadores. Este ambiente de testes é constituído por 5 Kernels (IS, EP, CG, MG, FT) desenvolvidos em C/Fortran em três versões, versão sequencial e versões paralelas (Open-MP e Open-MPI). Para além destes 5 Kernels, este *benchmark* tem um conjunto de classes (S, W, A, B, C, D, E, F) cada uma com diferentes tamanhos de dados. No desenvolvimento deste trabalho tive de escolher 3 desses 5 Kernels e algumas classes, de forma a efectuar uma gama de testes para cada uma das versões, num ambiente de operação cluster, mais precisamente no cluster "Search".

## 1. Introdução

Este trabalho foi realizado no âmbito da disciplina de Engenharia de Sistemas da Computação (ESC), inserida no perfil de Computação Paralela e Distribuída (CPD) do 4º Ano do curso Mestrado Integrado em Engenharia Informática (MIEI) e tem como objectivo analisar a *performance* de um ambiente de testes, neste caso o *Cluster SeARCH*, na execução do *NAS Parallel Benchmark*.

Este *Benchmark* é constituído por 5 Kernels e 3 simulações bem como um conjunto de 8 classes de dados, destes 5 Kernels tive de escolher alguns de forma a efectuar um conjunto de testes, para posteriormente fazer uma análise/comparação detalhada desses mesmos testes, para além da escolha dos Kernels ainda foi necessário escolher um conjunto de classes de dados para os testes.

Os testes consistiram na repetição destes nas diferentes classes de arquitecturas de nós existentes, usando as versões sequenciais e paralelas (memória distribuída - OpenMPI e memória partilhada - OpenMP), nos diferentes compiladores (gnu e intel), diferentes opções de compilação, diferentes tecnologias de comunicação e diferentes dimensões de dados (classes).

A análise e comparação dos resultados obtidos deverá ser feita tendo em consideração medições precisas de tempos de execução, da ocupação da memória, da comutação do tempo de E/S, bem como outras métricas das ferramentas de monitorização.

Para a realização destes testes foi desenvolvido um *Script* que me permitiu fazer a execução dos testes em lotes com base no sistema PBS.

Depois da obtenção dos resultados dos testes, os dados foram tratados com *Shell Scripts* utilizando ferramentas de processamento de linguagens, como o *grep* e *sort*. Posteriormente, depois de filtrados os dados, estes foram processados com *Excel* para geração dos gráficos.

## 2. Caracterização do Ambiente de Testes

O ambiente de testes utilizado no decorrer deste trabalho foi o cluster *SeARCH*, este cluster faz parte do Departamento de Informática da Universidade do Minho, sendo este utilizado por uma vasta comunidade de cientistas/investigadores. O *SeARCH* é constituído por um conjunto de nós com diferentes arquiteturas. Os nós constituintes do *SeARCH* são:

- Arquitectura *Ivy Bridge*
  - 6 Nós 662
  - 2 Nós 652
  - 20 Nós 641
- Arquitectura *Sandy Bridge*
  - 6 Nós 541
- Arquitetura *Nehalem*
  - 2 Nós 432
  - 4 Nós 421
  - 10 Nós 431
- Arquitetura *Penryn*
  - 6 Nós 321
- Arquitetura *AMD Magny-Cours*
  - 2 Nós 262

De notar que todos os detalhes de cada nó, bem como o significado do *rank* (valores do tipo 641, 652, etc. apresentados em cima) podem ser consultados no site do cluster *SeARCH* [3]

### 2.1. Nodos de Teste

As máquinas de teste que escolhi para a execução dos *Kernels* no *Cluster Search*, foram as máquinas 431 e 641 com arquiteturas *Nehalem* e *Ivy Bridge* respectivamente. Na tabela 1 encontra-se as especificação das características da máquina 431 e na tabela 2 as da máquina 641.

### 2.2. Compiladores Utilizados

No *Cluster Search* podemos encontrar uma gama de versões de compiladores da *GNU*, bem como outros compiladores como é o caso do compilador da *Intel*, neste trabalho os compiladores utilizados por mim foram:

System	Máquina 431
# CPUs	2
CPU	Intel® Xeon® X5650
Architecture	Nehalem
# Cores per CPU	6
# Threads per CPU	12
Clock Freq.	2.66 GHz
L1 Cache	192 KB 32 KB por core
L2 Cache	1536 KB 256 KB por core
L3 Cache	12 MB
Inst. Set Ext.	SSE4.2 e AVX
#Memory Channels	3
Memory BW	32 GB/s

Tabela 1. CARACTERIZAÇÃO DA MÁQUINA 431

System	Máquina 641
# CPUs	2
CPU	Intel® Xeon® E5-2650v2
Architecture	Ivy Bridge
# Cores per CPU	8
# Threads per CPU	16
Clock Freq.	2.6 GHz
L1 Cache	256 KB 32 KB por core
L2 Cache	2048 KB 256 KB por core
L3 Cache	20 MB
Inst. Set Ext.	AVX
#Memory Channels	4
Memory BW	59.7 GB/s

Tabela 2. CARACTERIZAÇÃO DA MÁQUINA 641

- gnu/4.9.0 - utilizei este compilador, porque este é o compilador que está predefinido no *Cluster Search*, como tal decidi optar por esta versão;
- gnu/4.9.3 - este compilador foi usado nos meus testes, por ser o compilador mais recente instalado no *Cluster Search*;
- intel/2013.1.117 - este compilador foi usado porque é o único compilador da *Intel* instalado no *Cluster Search* e para além disso utilizei-o para poder comparar os dois compiladores, isto é, comparar entre os compiladores da *Intel* e da *GNU*.

## 3. Caracterização do Benchmark

O *NAS Parallel Benchmark* [4] consiste num conjunto de 5 *Kernels* e 3 simulações desenvolvidos em C e Fortran, bem como um conjunto de 8 classes (cada uma com diferentes tamanhos de dados de *input*). Este *Benchmark* foi desenvolvido com o objetivo de fornecer uma medida das capacidade do hardware e software, para resolver intensivos problemas computacionais de dinâmica de fluídos importantes para a NASA.

Os *Kernels* e as simulações que constituem o *Benchmark* são os seguintes:

- **Kernels:**
  - IS [1] - Ordenação de inteiros, acesso aleatório à memória.
  - EP [5] - Este *Kernel* é embaralhosamente paralelo. Ele gera pares de desvio gausianas

aleatórios de acordo com um esquema específico. Sendo o objetivo estabelecer o ponto de referência para o máximo desempenho de uma determinada plataforma.

- CG [5] - Este *Kernel* utiliza um método de Gradiente Conjugado para calcular uma aproximação para o menor valor próprio de uma matriz grande, esparsa e não estruturada. Este testa cálculos de grelha não estruturados e comunicações usando uma matriz com entradas geradas aleatoriamente.
- MG [5] - Este *Kernel* calcula uma solução para a equação de *Poisson* 3-D através de um método Multigrid ciclo-V.
- FT [5] - Este *Kernel* faz a computação de uma Transformação de *Fourier* 3-D rápida com base no método espectral.

- **Simulações:**

- BT [5] - É uma aplicação de simulação CFD<sup>1</sup> que utiliza um algoritmo implícito para resolver equações de *Navier-Stokes* de 3-Dimenções.
- SP [5] - É uma aplicação de simulação CFD similar a BT, a diferença finita de soluções é baseada na factorização aproximada *Beam-Warming* que dissocia as dimensões x, y e z.
- LU [5] - É uma aplicação de simulação CFD que usa o método SSOR<sup>2</sup> para resolver um sistema *seven-block-diagonal* resultante da discretização de diferenças finitas das equações de *Navier-Stokes* em 3-D.

As classes de dados [1] presentes no *Benchmark* são:

- S - pequena, utilizada para testes rápidos;
- W - tamanho de estação de trabalho;
- A, B, C - Utilizada em testes de problemas *standard*. Aumenta aproximadamente 4X o tamanho de uma classe para a próxima.
- D, E, F - Utilizada em testes de problemas de larga escala. Aumenta aproximadamente 16X o tamanho de uma classe para a próxima.

É possível consultar informação mais detalhada de cada uma das classes, consultando o site do *NAS Parallel Benchmark* [2].

### 3.1. Escolha de *Kernels* e Classes de Dados

Depois de uma análise do *benchmark*, os kernels por mim escolhidos foram os seguintes:

- CG [1] - Gradiente conjugado, Irregular acesso à memória e comunicação irregular;
- EP [1] - Kernel embaralhadamente paralelo;

1. Computação de fluídos dinâmicos
2. Symmetric Successive Over-Relaxation

- IS [1] - Ordenação de inteiros, acesso aleatório à memória.

No que toca a classes de dados para testes, as minhas escolhas foram:

- Classe A [2] - Tem um tamanho de 50 MB ;
- Classe B [2] - Tem um tamanho de 200 MB;
- Classe C [2] - Tem um tamanho de 0.8 GB.

## 4. Testes Efectuados

Como foi dito anteriormente, o objetivo deste trabalho era efectuar uma gama de testes para diferentes *Kernel's* para diferentes Classes de dados, em baixo podem ser consultados quais os testes efectuados por mim, nas diferentes máquinas, para os diferentes *Kernel's*, para as diferentes classes e com diferentes compiladores. Sendo que o X diz que o teste foi realizado e - que não foi realizado.

### 4.1. Versão Sequencial

Na tabela 3 podemos consultar todos os testes efectuados para a versão sequencial na máquina 431 e na tabela 4 todos os testes efectuados para a versão sequencial na máquina 641.

Máquina 431	-O0	-O2	-O3
gnu/4.9.0	X	X	X
gnu/4.9.3	X	X	X
intel/2013.1.117	X	X	X

Tabela 3. TESTES EFECTUADOS PARA VERSÃO SEQUENCIAL NA MÁQUINA 431

Máquina 641	-O0	-O2	-O3
gnu/4.9.0	X	X	X
gnu/4.9.3	X	X	X
intel/2013.1.117	X	X	X

Tabela 4. TESTES EFECTUADOS PARA VERSÃO SEQUENCIAL NA MÁQUINA 641

### 4.2. Versão OMP

Na tabela 5 podemos consultar todos os testes efectuados para a versão OMP na máquina 431 e na tabela 6 todos os testes efectuados para a versão OMP na máquina 641.

Máquina 431	-O0	-O2	-O3
gnu/4.9.0	-	X	X
gnu/4.9.3	-	X	X
intel/2013.1.117	-	X	X

Tabela 5. TESTES EFECTUADOS PARA VERSÃO OMP NA MÁQUINA 431

Máquina 641	-O0	-O2	-O3
gnu/4.9.0	-	X	X
gnu/4.9.3	-	X	X
intel/2013.1.117	-	X	X

Tabela 6. TESTES EFECTUADOS PARA VERSÃO OMP NA MÁQUINA 641

### 4.3. Versão MPI

Na tabela 7 podemos consultar todos os testes efectuados para a versão MPI na máquina 431 e na tabela 8 todos os testes efetuados para a versão MPI na máquina 641. De referir que nestes testes fixei as flags em -O3, sendo que 8, 16 e 32 refere-se ao número de processos para os quais fiz os testes.

Máquina 431	8	16	32
gnu/openmpi.eth/1.8.4	X	X	X
gnu/openmpi.mx/1.8.4	-	-	-
intel/openmpi.eth/1.8.2	-	-	-
intel/openmpi.mx/1.8.2	-	-	-

Tabela 7. TESTES EFECTUADOS PARA VERSÃO MPI NA MÁQUINA 431

Máquina 641	8	16	32
gnu/openmpi.eth/1.8.4	X	X	X
gnu/openmpi.mx/1.8.4	-	-	-
intel/openmpi.eth/1.8.2	X	X	X
intel/openmpi.mx/1.8.2	-	-	-

Tabela 8. TESTES EFECTUADOS PARA VERSÃO MPI NA MÁQUINA 641

## 5. Análise de Resultados

### 5.1. Versão Sequencial

Nos testes para a versão sequencial, fiz uma análise dos tempos e dos MOPS<sup>3</sup> para diferentes compiladores, fixando as flags, isto é, cada gráfico apresenta os tempos para diferentes compiladores com a mesma flag de compilação.

Na figura 1, 2 e 3 encontram-se os gráficos com os tempos para diferentes compiladores, com a mesma flag de compilação para as Máquinas 431.

Através da comparação dos 3 gráficos, á medida que se vai aumentando o nível de optimização, isto é, passado de -O0 para -O2 e por fim para -O3 podemos constatar que o comportamento dos compiladores é semelhante, mesmo com o aumento da Classe de dados. No caso do gráfico da figura 3 para o Kernel CG com a classe de dados B o compilador da Intel mostra-se melhor do que os outros, neste caso o Kernel CG para a classe de dados C, ai o compilador da Intel é pior que os outros dois, isto é o Kernel compilado com o compilador da Intel apresenta tempos de execução superiores aos outros dois. No caso do Kernel EP (Embaraçosamente paralelo) podemos afirmar

3. Milhões de Operações Por Segundo

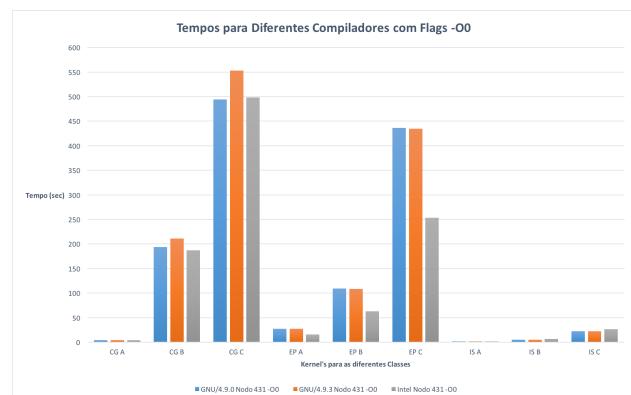


Figura 1. Tempos para Diferentes Compiladores com Flags -O0

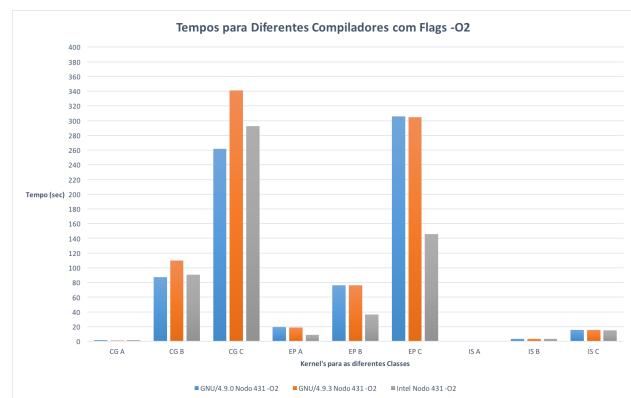


Figura 2. Tempos para Diferentes Compiladores com Flags -O2

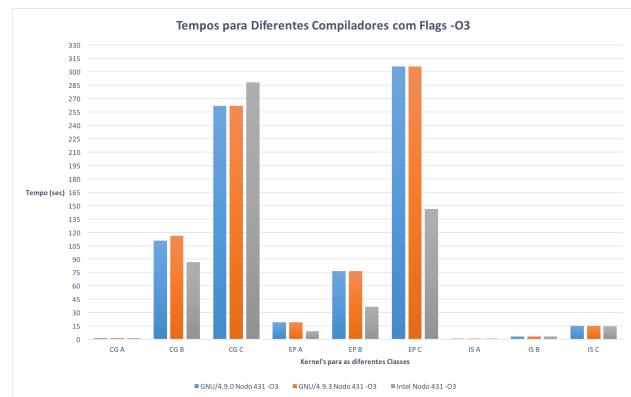


Figura 3. Tempos para Diferentes Compiladores com Flags -O3

que o compilador da Intel é melhor que os outros dois da GNU, pois para qualquer classe de dados o Kernel apresenta sempre tempos melhores que do que quando compilado com os compiladores da GNU.

Nos gráficos das figuras 4, 5 e 6 podemos consultar os gráficos que relacionam os MOPS para os diferentes Kernel's para as diferentes classes de dados, para os diferentes compiladores, com a mesma flag de compilação.

Neste caso, mais uma vez, à medida que se aumenta o nível de compilação, o comportamento dos compiladores é semelhante, sendo que o *Kernel* CG é o que apresenta maior numero de MOPS, no caso do gráfico da figura 6 este *Kernel*, para a classe de dados A o compilador da Intel é ligeiramente menos eficiente do que os outros dois da *GNU*. Para a classe de dados B o compilador mais eficiente é o compilador da *GNU* da versão 4.9.0, tendo este aproximadamente o mesmo valor que o *Kernel* quando compilado com o compilador da *intel*. Na classe de dados C para os dois compiladores da *GNU* o *Kernel* apresenta valores semelhantes sendo estes melhores que o *Kernel* quando compilado com o compilador da *Intel*. A semelhança do que acontece nos tempos, para o *Kernel* EP o compilador da *Intel*, mais uma vez é o mais eficiente, apresentando valores sempre superiores aos outros dois compiladores da *GNU*, quer para a classe de dados A, B e C.

Os resultados para as máquinas 641 são praticamente semelhantes aos resultados das máquinas 431, isto é o comportamento á medida que se aumenta o nível de optimização é praticamente igual. Os seus gráficos podem ser consultados no anexo A

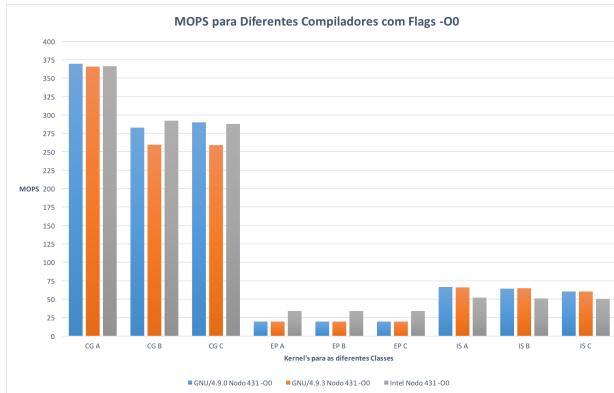


Figura 4. MOPS para Diferentes Compiladores com Flags -O0

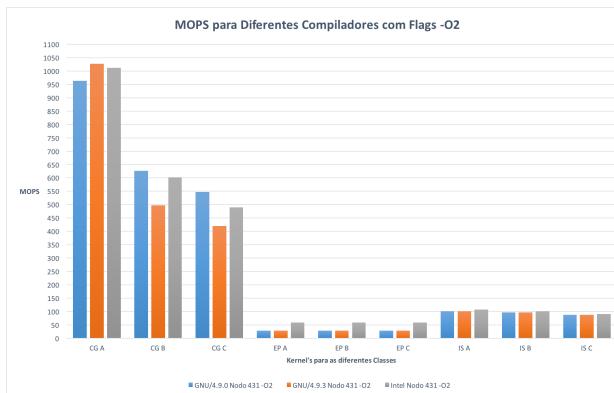


Figura 5. MOPS para Diferentes Compiladores com Flags -O2

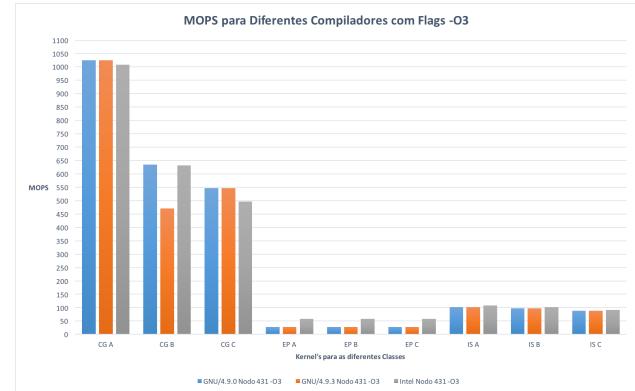


Figura 6. MOPS para Diferentes Compiladores com Flags -O3

## 5.2. Versão OMP

Um dos objetivos do trabalho era executar código em paralelo, neste caso com um paradigma de memória partilhada recorrendo a *OpenMP*, sendo que os testes foram efetuados para 1, 2, 4, 8, 10, 12, 16, 24 e 32 *Threads*. Na análise dos resultados destes testes, procurei comparar os tempos de execução, os MOPS e MOPS/Thread para os diferentes compiladores, fixando as flags de compilação para um *Kernel*. De referir que devido a um grande número de gráficos que foram gerados para a versão OMP, apenas apresento uma parte dos resultados, para isso procurei apresentar os gráficos que achei que representam os melhores resultados que obtive. Como mostra a tabela 5 e a tabela 6 para esta versão apenas efetuei testes com as flags de compilação -O2 e -O3. De notar que os gráficos apresentados dizem respeito aos testes efetuados nas máquinas 431.

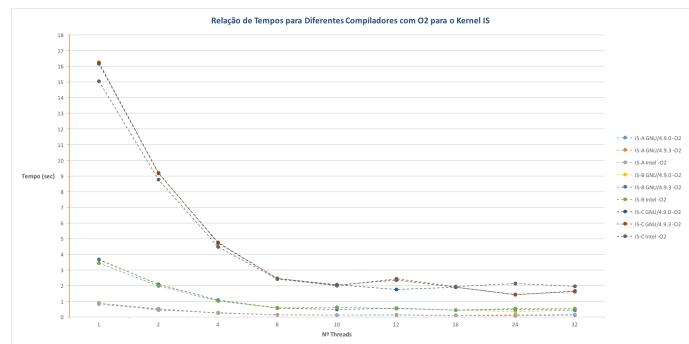


Figura 7. Tempos para Diferentes Compiladores com Flags -O2 para o Kernel IS

Escolhi os gráficos do *Kernel* IS, pois foi o *Kernel* para o qual obtive um tempo de execução inferior. Como podemos verificar pela a análise do gráfico da figura 7 e do gráfico da figura 8 há pouca variação dos tempos quer na versão compilada com a flag -O2, quer com a flag -O3, tanto um gráfico como o outro são praticamente iguais.

No gráfico da figura 7 com flag -O2 o *Kernel* IS atinge o seu menor tempo para o compilador gnu/4.9.0 com 24

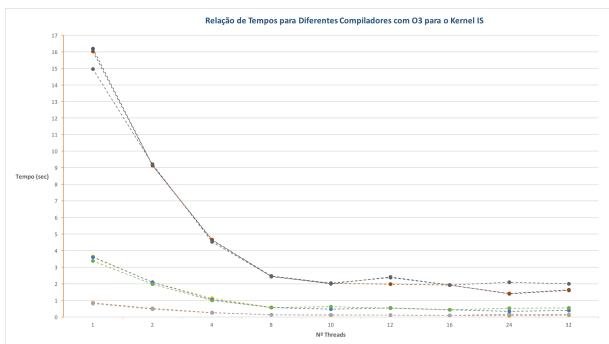


Figura 8. Tempos para Diferentes Compiladores com Flags -O3 para o Kernel IS

*Threads* tanto para a classe de dados A, B e C com tempos de 0.08 segundos, 0.33 segundos e 1.41 segundos respectivamente. Para o compilador gnu/4.9.3 com flag -O2 o *Kernel* atinge o menor tempo para a classe A com 24 *Threads* bem como para a classe C com tempos de 0.08 segundos e 1.44 segundos respectivamente, para a classe B o *Kernel* atinge o menor tempo tanto com 16 *Threads* como com 24 *Threads* com valor de 0.44 segundos. Quanto ao compilador da Intel, com flag -O2 o *Kernel* atinge o seu menor tempo com 16 *Threads*, quer para a classe A, B e C com tempos de 0.1 segundos, 0.43 segundos e 1.93 segundos respectivamente. Podemos assim concluir que para este *Kernel* com flag -O2, independentemente da classe de dados o compilador da Intel é o mais eficiente.

No gráfico da figura 8, com flag -O3, para o compilador gnu/4.9.0 o *Kernel* atinge o seu menor tempo para a classe A com 16 *Threads*, tendo um valor de 0.1 segundos e para as classes de dados B e C atinge o menor tempo com 24 *Threads*, com tempos de 0.34 segundos e 1.4 segundos respectivamente. Para o compilador da gnu/4.9.3, com flag -O3 o *Kernel* atinge o seu menor tempo com 24 *Threads*, quer para a classe A, B e C, com tempos de 0.08 segundos, 0.35 segundos e 1.41 segundos respectivamente. No que toca ao compilador da Intel com flag -O3 o *Kernel* atinge o seu menor tempo com 16 *Threads*, com tempos de 0.1 segundos, 0.43 segundos e 1.92 segundos respectivamente. À semelhança do que acontece com o caso anterior, isto é, quando o *Kernel* é compilado com -O2, aqui o compilador da Intel também mostra ser o mais eficiente, com flag -O3.

Quanto aos MOPS como podemos verificar nos gráficos da figura 9 e no gráfico da figura 10, estes tendem a aumentar até às 24 *Threads*, tanto com flag -O2 como com flag -O3. Em oposição os MOPS/*Thread* tendem a diminuir com o aumento do número de *Threads* tanto com flag -O2 como com flag -O3, como mostra o gráfico da figura 11 e o gráfico da figura 12, o que já era de esperar.

Os gráficos para o *Kernel IS* nas máquinas 641 podem ser consultados no anexo A.

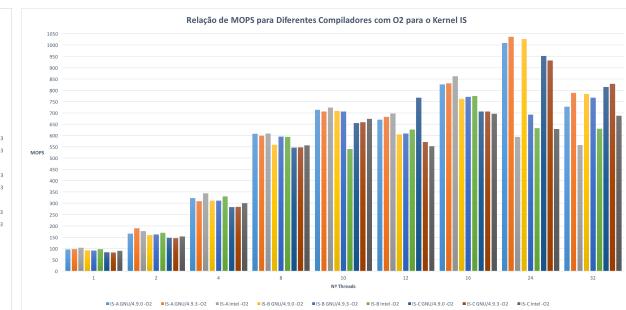


Figura 9. MOPS para Diferentes Compiladores com Flags -O2 para o Kernel IS

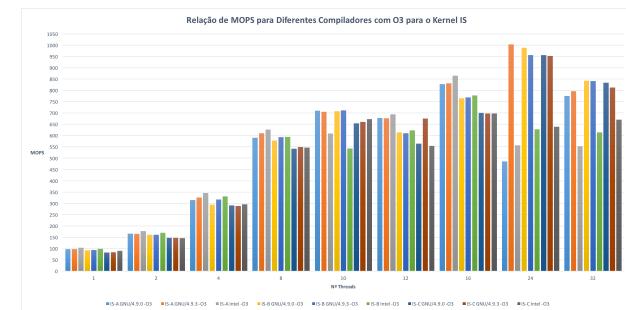


Figura 10. MOPS para Diferentes Compiladores com Flags -O3 para o Kernel IS

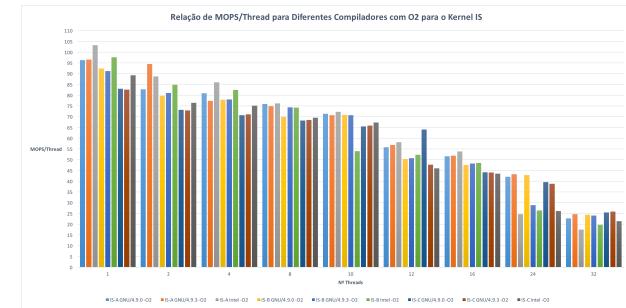


Figura 11. MOPS/Thread para Diferentes Compiladores com Flags -O2 para o Kernel IS

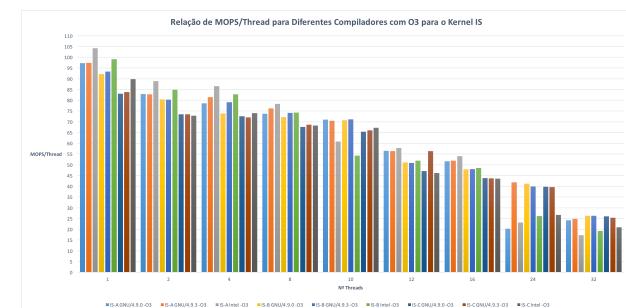


Figura 12. MOPS/Thread para Diferentes Compiladores com Flags -O3 para o Kernel IS

### 5.3. Versão MPI

Para além da versão paralela OMP, era também objetivo deste trabalho efetuar testes com um paradigma de memória distribuída, neste caso MPI. Nesta versão efetuei testes para ethernet quer com compiladores da intel, quer com compiladores da gnu, testei com 8, 16 e 32 processos para cada classe de dados. De notar que os dados apresentados, dizem respeito a testes efetuados nas máquinas 641, apenas com *Ethernet* uma vez que não consegui obter dados para a execução com *Myrinet*.

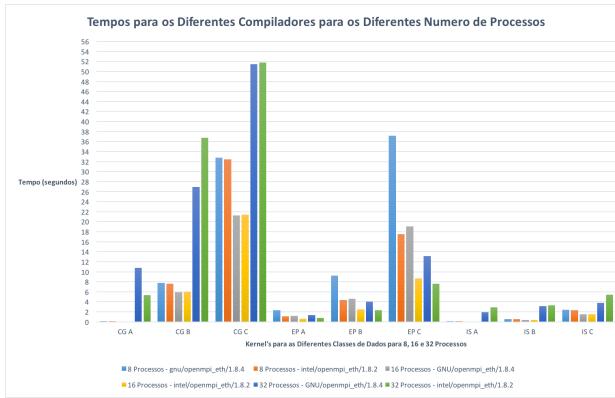


Figura 13. Tempos para os Diferentes Compiladores para os Diferentes Números de processos

No gráfico da figura 13 podemos consultar os tempos de cada *Kernel* para as diferentes classes de dados numa execução com 8 processos, 16 processos e 32 processos. Como podemos analisar o *Kernel* CG atinge o seu menor tempo, independentemente da classe de dados, com 16 processos, quer para o compilador da *GNU*, quer com o compilador da *Intel*.

No que diz respeito ao *Kernel* EP (Embaraçosamente Paralelo), este atinge o seu menor tempo com 32 processos para as classes de dados B e C com o compilador da *Intel* e para a classe A atinge o seu menor tempo com 16 processos mas também com o compilador da *Intel*. Por isso, para este *Kernel* podemos concluir que o compilador da *Intel* é o mais eficiente.

Por fim o *Kernel* IS atinge o seu melhor tempo com 16 processos, qualquer que seja a sua classe de dados, sendo que os tempos de execução entre dois os compiladores, são muito próximos um do outro.

Quanto aos MOPS, ao analisar o gráfico da figura 14, como podemos ver, os MOPS para o *Kernel* CG tendem a diminuir com 8 processos e com 16 processos, com o aumento da classe de dados, sendo que com 32 processos, estes tendem a aumentar, ao aumentar a classe de dados. Pela análise do gráfico podemos concluir que para o *Kernel* CG é mais eficiente executá-lo com 16 processos, uma vez que para as diferentes classes de dados, é com 16 processos que os MOPS são maiores.

No *Kernel* EP os MOPS apresentam aproximadamente os mesmos valores, quer para 8 e 16 processos, como pode

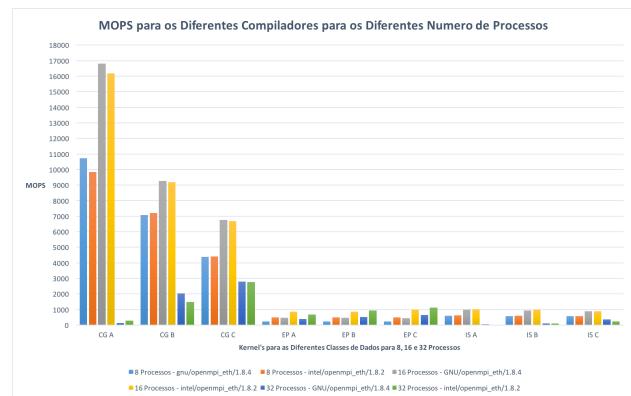


Figura 14. MOPS para os Diferentes Compiladores para os Diferentes Números de processos

ser comprovado pelo gráfico da figura 14, contudo, para 32 processos os MOPS aumentam ligeiramente com o aumento das classes de dados. Neste *Kernel* para as classes de dados B e C, o mais eficiente é executar o *Kernel* com 32 processos com o compilador da *Intel*, sendo que para a classe de dados A, o mais eficiente é executar o *Kernel* com 16 processos, com o compilador da *GNU*.

Por fim no *Kernel* IS os MOPS, mais uma vez tendem a ter aproximadamente os mesmos valores, havendo pouca variação, à medida que se aumenta a classe de dados, contudo é com 16 processos, que a execução deste *Kernel* mostra ser mais eficiente, tanto com o compilador da *GNU* como com o compilador da *Intel*.

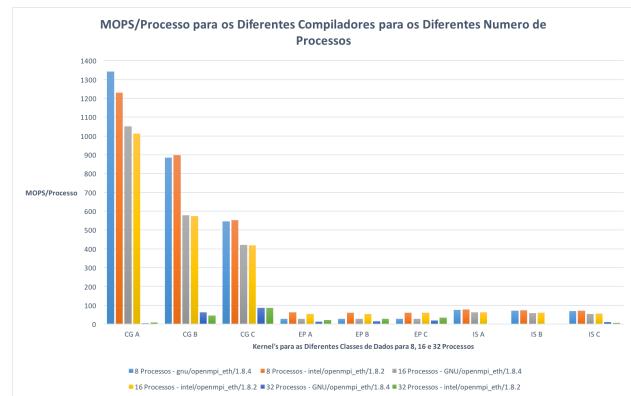


Figura 15. MOPS para os Diferentes Compiladores para os Diferentes Números de processos

O gráfico da figura 15 mostra a relação MOPS/Processo, entre os diferentes *Kernel's* para as diferentes classes de dados, para 8, 16 e 32 processos. Pela análise do gráfico, podemos verificar que para o *Kernel* CG à medida que se aumenta a classe de dados, de A até C os MOPS por processo diminuem, para 8 e 16 processos, sendo que para 32 processos, estes aumentam para os dois compiladores, o que já era de esperar uma vez que o comportamento dos MOPS é o mesmo.

No *Kernel EP* o comportamento dos MOPS por processo é semelhante ao comportamento dos MOPS, isto é, tendem a ter os mesmos valores (aproximadamente), à medida que se aumenta a classe de dados. Para o *Kernel IS*, os MOPS por processo tendem a ter os mesmos valores, à semelhança do que acontece com no gráfico da figura 14.

## 6. Ganhos

No que toca aos ganhos, decidi filtrar um bocado os dados uma vez que, caso não o fizesse o relatório iria ficar muito extenso. Com isto, decidi escolher o *Kernel IS* com as classes de dados A, B e C, compilado com o compilador da *Intel* com flag -O3.

### 6.1. Sequencial vs OMP

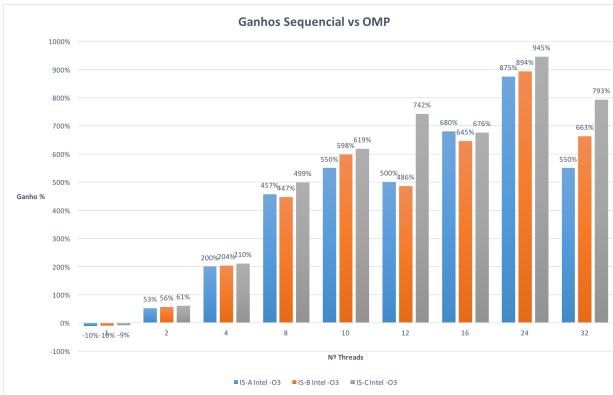


Figura 16. Percentagem de Ganhos Versão Sequencial vs Versão OMP

Como podemos verificar no gráfico da figura 16, os tempos da versão OMP começam mesmo por ter um ganho negativo, isto é, tem um tempo pior que a versão sequencial. Contudo, à medida que vai aumentando o numero de *Threads* os ganhos também vão aumentando, atingindo o seu máximo para 24 *Threads*. Para a classe de dados A, B e C com 24 *Threads* os ganhos são de 875%, 894% e 945% respetivamente, em relação à versão sequencial.

### 6.2. Sequencial vs MPI

Na relação de ganhos entre a versão sequencial e a versão MPI, eu fixei o compilador e as flags, sendo que apresento os ganhos para todas as classes de dados, e para o diferente numero de processos, para o qual efetuei os testes, como pode ser comprovado pela gráfica da figura 17.

Como pode ser analisado pelo gráfico no *Kernel CG*, é com 16 processos que se atinge maior percentagem de ganhos da versão MPI em relação à versão sequencial, tanto para a classe de dados A, como B e como C, apresentando valores de 1189%, 1695% e 1272%. Já no *Kernel EP* a percentagem de ganho da versão MPI em relação à versão sequencial atinge o seu máximo com 32 processos para a

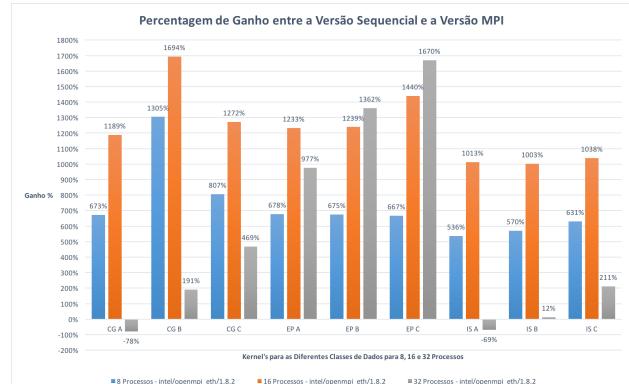


Figura 17. Percentagem de Ganhos Versão Sequencial vs Versão MPI

classe B e C, com valores de 1392% e 1670% e para a classe de dados A atinge o seu máximo com 16 processos com um valor de 1233%.

Por fim, no *Kernel IS* a percentagem de ganho da versão MPI em relação à versão sequencial, atinge o seu máximo com 16 processos, independentemente da classe de dados, apresentando um valor de 1013% para a classe de dados A, 1003% para a classe de dados B e 1038% para a classe de dados C.

De referir que no caso do *Kernel CG* e do *Kernel IS*, com 32 processos para a classe de dados A, a percentagem de ganho chega a ser negativa, isto é, o tempo de execução da versão MPI é pior que o tempo de execução da versão sequencial.

## 7. Utilitários de Monitorização

No desenvolvimento deste trabalho, aquando de cada teste utilizei uma ferramenta de monitorização, no meu caso *dstat*, para examinar o estado do sistema de operação, neste caso o *Cluster Search*. Estas ferramentas permitem-nos visualizar a forma como o sistema é afetado em termos de utilização do *cpu*, da memória e dos discos aquando do aumento gradual da carga de computação.

Como referi, utilizei o comando *dstat* com as seguintes flags:

- c - ativa as estatísticas do *cpu* (*system, user, idle, wait, hardware interrupt* e *software interrupt*);
- d - ativa as estatísticas do disco (*read* e *write*);
- m - ativa as estatísticas de memória (*used, buffers, cache, free*).

Nesta secção irei analizar estas estatísticas para um dado *Kernel*, no meu caso para o *Kernel IS*, para o compilador da *Intel* com flag -O3 com a classe de dados C para cada uma das versões, isto é, para a versão sequencial, versão OMP, nas máquinas 431 e para a e versão MPI nas máquinas 641.

### 7.1. Versão Sequencial

Na versão sequencial apenas apresento os gráficos do *dstat* respetivos ao melhor tempo de execução.

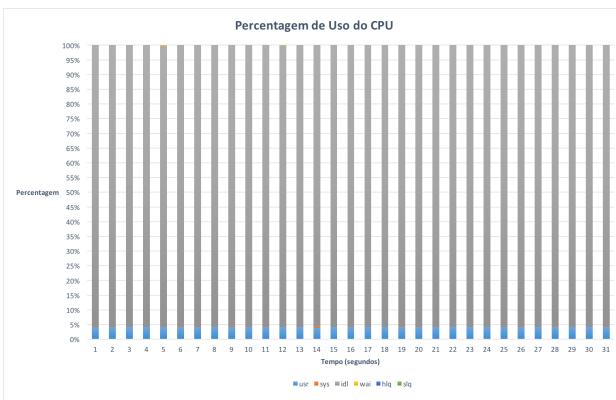


Figura 18. Percentagem de Uso do CPU

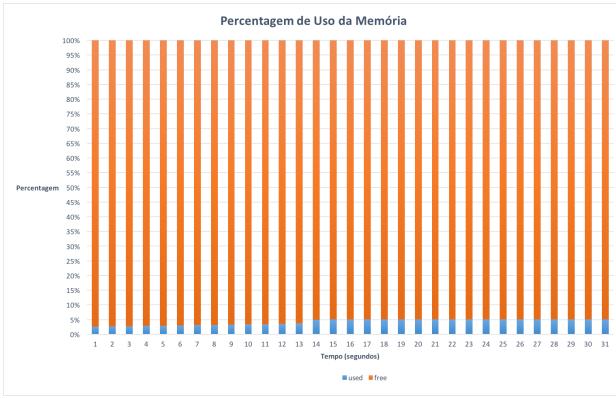


Figura 19. Percentagem de Uso da Memória

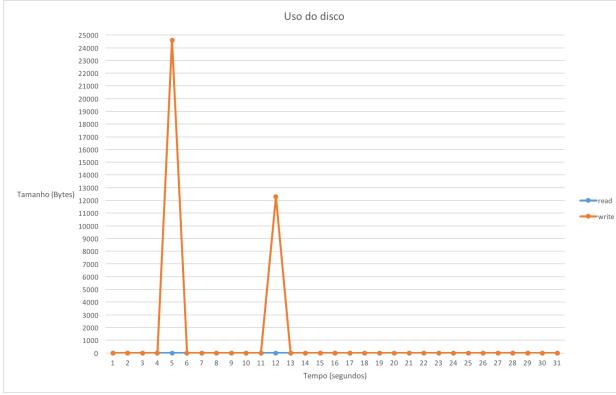


Figura 20. Uso do Disco

No que toca ao uso do *CPU*, para o *Kernel IS* com a classe de dados C, na máquina 431, na versão sequencial, analisando o gráfico da figura 18 podemos verificar que a percentagem do uso para *user* (*usr*), sobe ligeiramente desde o instante inicial até aos 12 segundos, estabilizando a partir dai nos 5%, aproximadamente, do uso do *CPU*. A percentagem de uso do *CPU* para *system* (*sys*) em todo o tempo de execução é sempre inferior a 1%. De referir que o

*CPU*, em todo o instante de tempo de execução se encontra desocupado (*idle*) com uma percentagem de cerca de 95%, daí podemos concluir que o não estamos a tirar o máximo partido do *CPU*, uma vez que a cada instante de tempo, o *Kernel* apenas está a utilizar cerca de 5% deste.

Na percentagem de uso da memória, analisando o gráfico da figura 19, verificamos que desde os 14 e até ao fim da execução o *Kernel* está a utilizar cerca de 5%, aproximadamente, da memória. Sendo que desde o instante inicial, até aos 14 segundos, o *Kernel* vai aumentado o uso da memória ligeiramente. Concluimos assim que o *Kernel* é *CPU Bound* e não *Memory Bound*.

Por fim, referente ao uso do disco, ao analisarmos o gráfico da figura 20, verificamos que em todo o instante de execução não há leituras (*read*) do disco, sendo que no instante 5 segundos e 12 segundos está a ser escrito (*write*) em disco cerca de 25 KB e 12.5 KB respetivamente.

## 7.2. Versão OMP

Na versão OMP apenas apresento os gráficos do *dstat* respetivos ao melhor tempo de execução para o melhor número de *Threads*, isto é, para o número de *Threads* em que o tempo é menor, neste caso para 24 *Threads*.

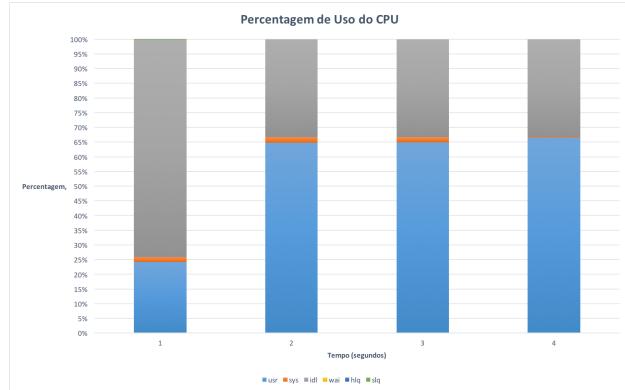


Figura 21. Percentagem de Uso do CPU

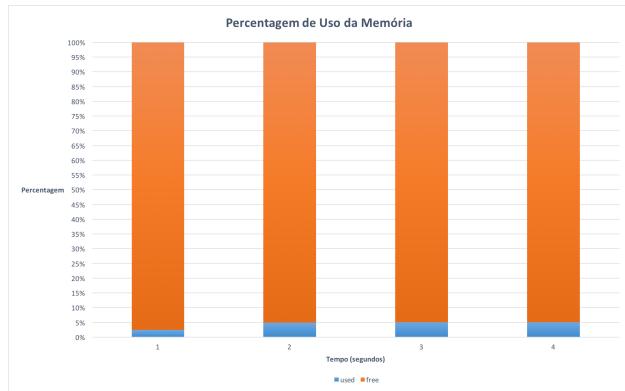


Figura 22. Percentagem de Uso da Memória

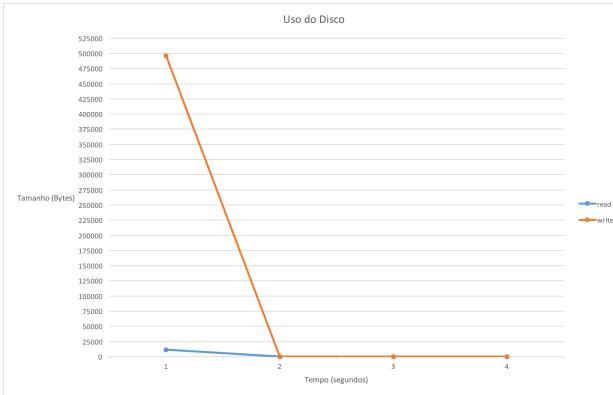


Figura 23. Uso do Disco

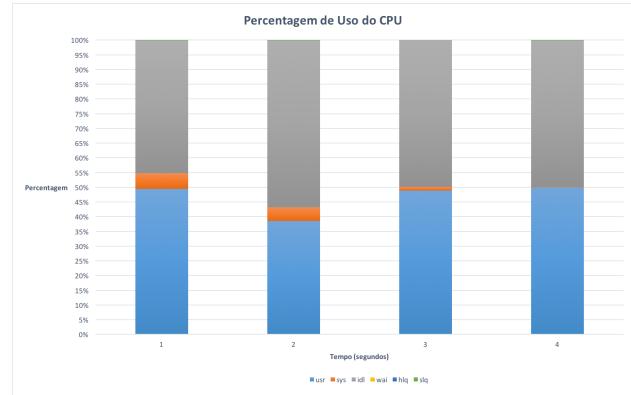


Figura 24. Percentagem de Uso do CPU

Analisando o gráfico da figura 21, referente à percentagem de uso do *CPU*, verificamos que, no primeiro segundo a percentagem de uso do *CPU* para *user (usr)* é de cerca de 25%, a partir do segundo segundo e até final do tempo de execução o uso do *CPU* para *user (usr)* mantém-se nos 65%, aproximadamente. Quanto ao uso do *CPU* para *system (sys)* no primeiro segundo, apresenta um valor inferior a 1% sendo que a partir dai e até final apresenta um valor de aproximadamente 2%. Podemos concluir a partir do gráfico que com o paradigma de memória partilhada conseguimos tirar um maior proveito do *CPU* em relação ao sequencial, visto que apresenta uma percentagem de uso mais elevada.

Referente ao uso da memória e analisando o gráfico da figura 22, verificamos que o *Kernel* no primeiro segundo usa cerca de 2.5% da memória e a partir do segundo segundo, este utiliza cerca de 5% da memória, até ao final da execução. Analisando estes resultados podemos concluir que este *Kernel* é *CPU Bound* e não *Memory Bound*, à semelhança do que acontece na versão sequencial.

Para concluir a versão OMP e analisando o gráfico da figura 23, vemos que no primeiro segundo é escrito (*write*) em disco cerca de 500 KB e lido (*read*) cerca de 11.5 KB. Nos restantes segundos, até final da execução não há mais leituras nem escritas do disco.

### 7.3. Versão MPI

Na versão OMP apenas apresento os gráficos do *dstat* respetivos ao melhor tempo de execução para o melhor número de Processos, isto é, para o número de Processos em que o tempo é menor, neste caso com 16 Processos.

Na versão MPI, quanto ao uso do *CPU* e analisando o gráfico da figura 24, verificamos que no primeiro segundo a percentagem de uso para *user (usr)* é de cerca de 50%, sendo que no segundo segundo, baixa para cerca de 38%, voltando a aumentar nos segundos seguintes, e até final, para cerca de 50%. Quanto ao uso do *CPU* para *system (sys)* no primeiro segundo é de 5%, baixando nos segundos seguintes, até ao instante final, no qual apresenta um valor inferior a 1%. De referir que para esta versão MPI em todos os instantes de

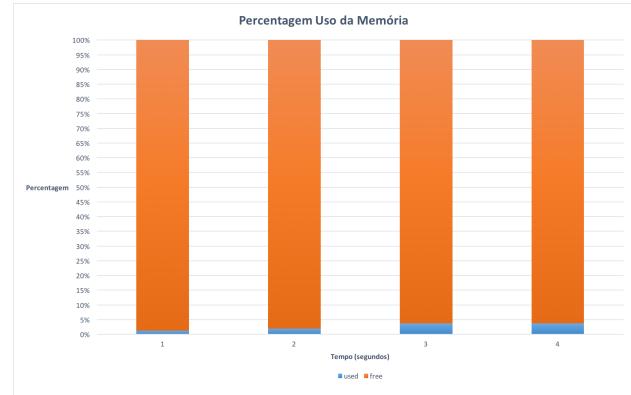


Figura 25. Percentagem de Uso da Memória

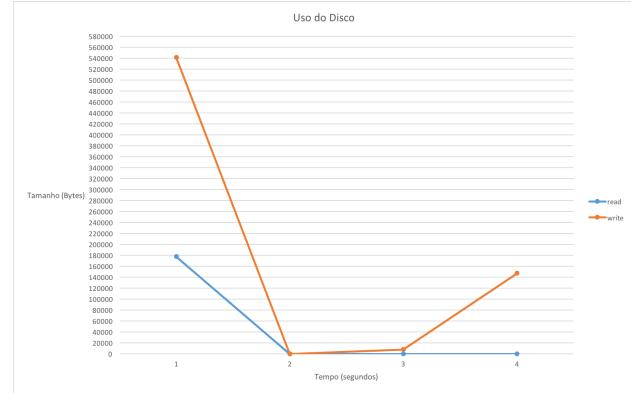


Figura 26. Uso do Disco

tempo o *CPU* tem uma percentagem de desocupação (*idle*) de cerca de 50%.

Nesta versão, a percentagem de uso da memória apresenta padrões semelhantes às versões anteriores (versão sequencial e versão MPI), subindo a cada instante de tempo, mas sempre com valores inferiores a 5%. Mais uma vez podemos concluir que este *Kernel* é *CPU Bound* e não *Memory Bound*.

Na utilização do disco, verificamos através da analise do gráfico da figura 26, que no primeiro segundo é escrito (*write*) em disco cerca de 540 KB e lidos (*read*) cerca de 180 KB. No segundo segundo não é escrito nenhum *Byte*, sendo que a partir deste instante não é lido qualquer *Byte*, até final. No terceiro segundo é escrito cerca de 8 KB no disco e no quarto e ultimo segundo é escrito cerca de 150 KB em disco.

#### 7.4. Análise Geral

Quanto à analise dos gráficos da ferramenta de monitorização *dstat*, podemos concluir que, na versão OMP é utilizado mais *CPU* em relação às outras versões (versão sequencial e versão MPI), sendo que a versão MPI também apresenta valores razoáveis de utilização do *CPU*. Contudo é com o paradigma de memória distribuída (MPI) que se obtém maiores ganhos, como pode ser comprovado na secção 6. Com a análise dos gráficos da percentagem uso de memória, podemos concluir que este *Kernel* é *CPU Bound* e não *Memory Bound*. Por fim, no que diz respeito ao uso da memória podemos concluir que é na versão MPI que é efetuado maior número de operações de escrita e leitura em disco, como pode ser comprovado pelos gráficos de uso de disco.

### 8. Conclusão

Depois de realizado todos os testes, do tratamento dos resultados e por fim a análise destes, posso concluir de uma maneira geral, que na versão sequencial na máquina 431, os tempos de execução são melhores, quando os *Kernel's* são compilados com o compilador da *Intel*, de notar que á medida que se aumenta os níveis de optimização, o intervalo de tempo de cada gráfico, também vai diminuindo, o que já era de esperar, uma vez que estamos a aumentar as optimizações. Quanto ao número de MOPS, também de uma maneira geral o compilador da *Intel* se mostra o mais eficiente, apresentando valores superiores em relação aos outros compiladores. De notar também que à medida que vamos aumentando os níveis de optimização, o intervalo de valores de cada gráfico vai aumentando, sendo que do nível -O2 para -O3 esse aumento não é muito significativo.

Quanto à versão OMP, podemos concluir que para o *Kernel* IS com um nível de optimização -O2 e -O3 e com a classe de dados C (classe de dados maior), que a versão 4.9.0 e a versão 4.9.3 do compilador da *GNU* são os compiladores mais eficientes, uma vez que é com a execução do *Kernel* compilado com estes compiladores que se encontra o menor tempo, sendo esse valor encontrado na execução do *Kernel* com 24 *Threads*. Quanto ao número de MOPS é com a versão do compilador 4.9.3 do compilador da *GNU* que se encontra o valor mais elevado de MOPS tanto para o nível de optimização -O2 e -O3, sendo esse valor encontrado com 24 *Threads* o que nos permite concluir que é este o melhor compilador, para a obtenção de melhores resultados desta métrica. No que toca aos MOPS/*thread* é com o compilador da *Intel* que se encontra o maior numero de MOPS/*Thread*,

sendo este valor encontrado com apenas 1 *Thread*, o que podemos concluir que para a obtenção dos melhores valores desta métrica, é o compilador da *Intel* a melhor escolha. Contudo por uma análise do gráfico dos tempos, podemos concluir que com 1 *Thread* é onde se obtém o pior tempo.

No que toca à versão MPI, podemos concluir que o compilador da *Intel* é o compilador mais eficiente, no que toca à obtenção dos melhores tempos, uma vez que, de uma maneira geral, é com este compilador, numa execução com 16 processos, mapeados por core, que se obtém os melhores tempos. Quanto ao número de MOPS, também de uma maneira geral é o compilador da *Intel* que é o melhor compilador para a obtenção dos melhores valores desta métrica, uma vez que é com este compilador que, de uma maneira geral, se encontra o maior número de MOPS, sendo estes valores encontrados com 16 processos. Por fim, quanto ao número de MOPS/Processo é o compilador da *Intel* que é o melhor, uma vez que é com este, que encontramos os melhores valores para esta métrica, contudo, ao contrário do que acontece com os tempos e com os MOPS, é com 8 processos que encontramos os melhores valores para esta métrica.

Para finalizar a conclusão da analise dos resultados, relativamente aos ganhos, podemos concluir que o paradigma de memória distribuída (MPI) é o que apresenta melhores tempos de execução relativamente ao sequencial. Na versão MPI os tempos, de uma maneira geral, apresentam valores 1000% ou mais inferiores à versão sequencial, enquanto na versão OMP o maior ganho é de 945%.

Referente à realização deste trabalho, efetuei inúmeros testes no *Cluster Search*, para a realização desses testes tive de criar uma forma de automatizar o trabalho, para isso recorri a *Shell Scripts* e ferramentas de escalonamento de tarefas, como o PBS. Depois de efetuados os testes, tive de processar os dados de forma a criar gráficos que exprimissem os meus resultados, para isso usei ferramentas de processamento de linguagens como é o caso do *gawk* e do *grep*, por fim, depois de ter todos os dados organizados, recorri ao excel para a geração dos gráficos.

Este trabalho permitiu-me ambientar de uma forma mais objetiva num ambiente de *clustering*, sendo que passei por algumas dificuldades principalmente na execução dos testes para a versão MPI, dificuldades essas que não foram totalmente ultrapassadas, pelo menos na execução em *Myrinet*. Para alem dessas dificuldades, apesar de ter automatizado grande parte do trabalho com as *scripts*, penso que perdi demasiado tempo na criação dos gráficos, uma vez que os fiz todos à mão e não era esse o objetivo. Contudo, faço um balanço positivo da maior parte do trabalho, sendo que, como trabalho futuro pretendo aperfeiçoar principalmente o meu processo de automatização, de forma a não perder demasiado tempo em algumas partes, bem como conseguir efetuar todos os testes que defini ao inicio.

### Referências

- [1] Nasa advanced supercomputing division. <http://www.nas.nasa.gov/publications/npb.html>

- [2] Problem sizes and parameters in nas parallel benchmarks. [http://www.nas.nasa.gov/publications/npb\\_problem\\_sizes.html](http://www.nas.nasa.gov/publications/npb_problem_sizes.html).
- [3] Services and advanced research computing with htc/hpc clusters. [http://search6.di.uminho.pt/wordpress/?page\\_id=55](http://search6.di.uminho.pt/wordpress/?page_id=55).
- [4] R. F. V. der Wijngaart and M. Frumkin. Nas grid benchmarks version 1.0. *NASA Technical Report NAS-02.005*, 7 2002.
- [5] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. *NAS Technical Report NAS-99-011*, 10 1999.

## Apêndice

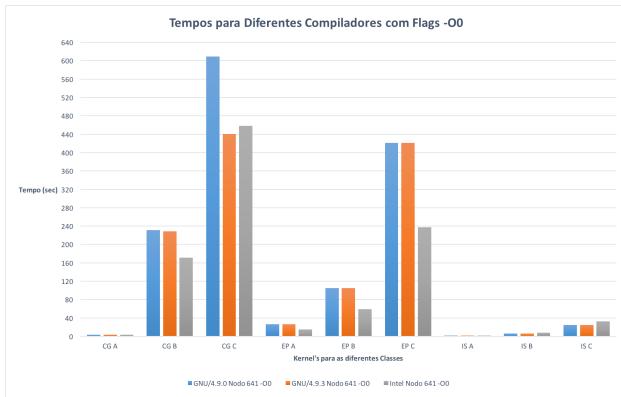


Figura 27. Tempos para Diferentes Compiladores com Flags -O0

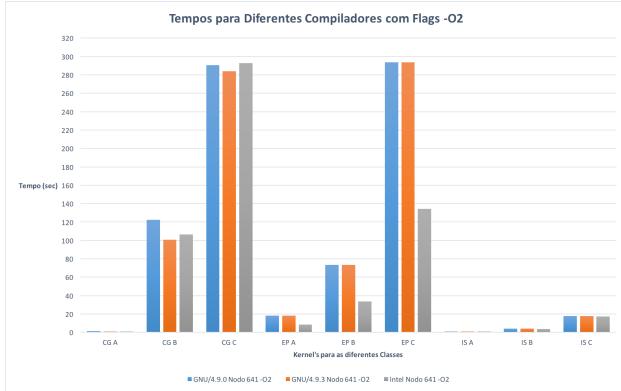


Figura 28. Tempos para Diferentes Compiladores com Flags -O2

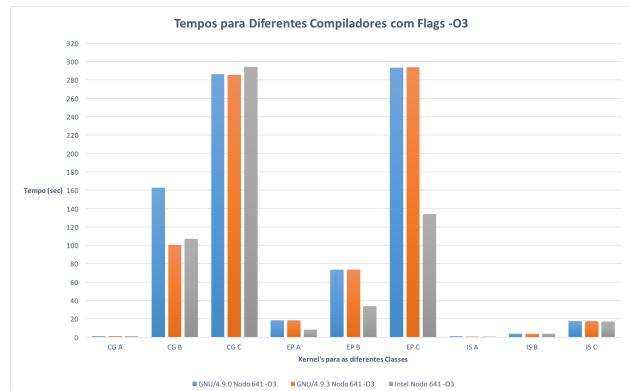


Figura 29. Tempos para Diferentes Compiladores com Flags -O3

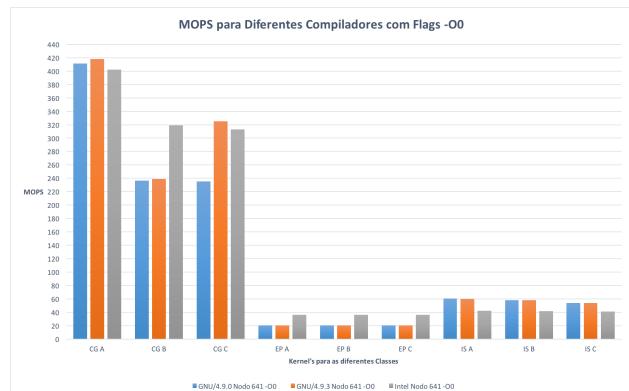


Figura 30. MOPS para Diferentes Compiladores com Flags -O0

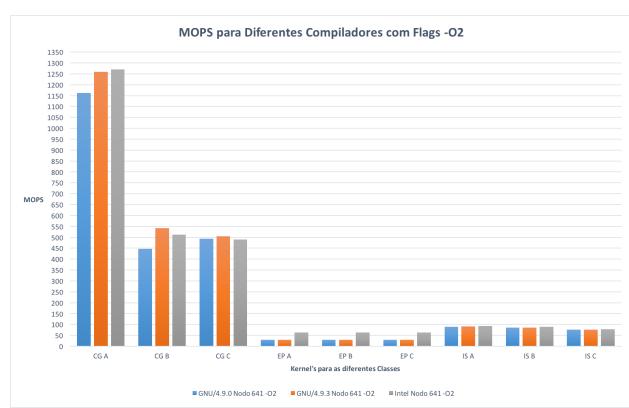


Figura 31. MOPS para Diferentes Compiladores com Flags -O2

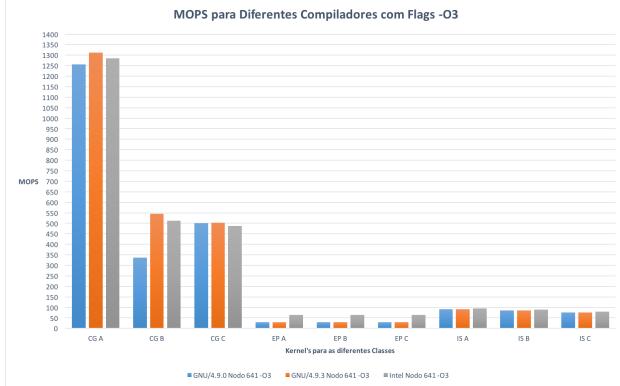


Figura 32. MOPS para Diferentes Compiladores com Flags -O3

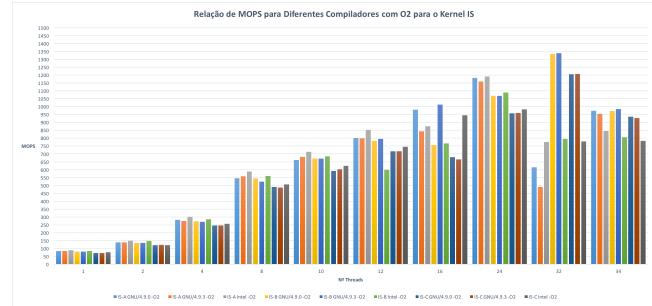


Figura 35. MOPS para Diferentes Compiladores com Flags -O2 para o Kernel IS

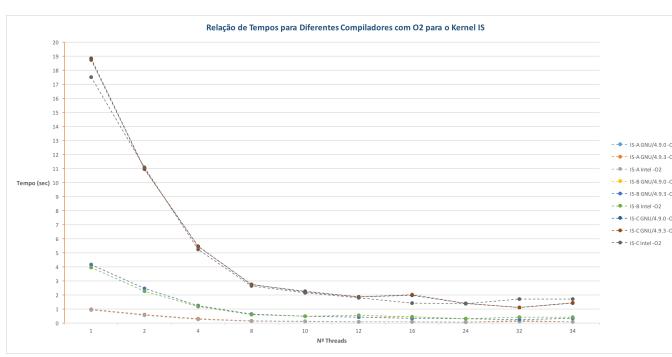


Figura 33. Tempos para Diferentes Compiladores com Flags -O2 para o Kernel IS

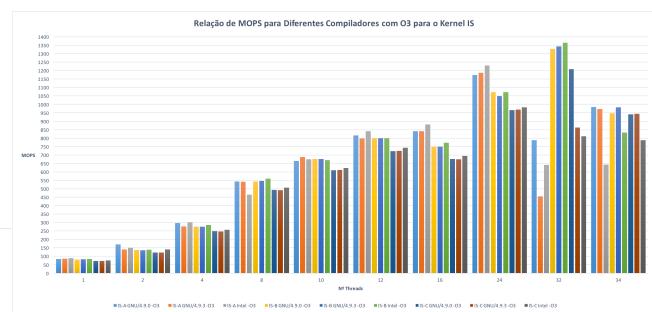


Figura 36. MOPS para Diferentes Compiladores com Flags -O3 para o Kernel IS

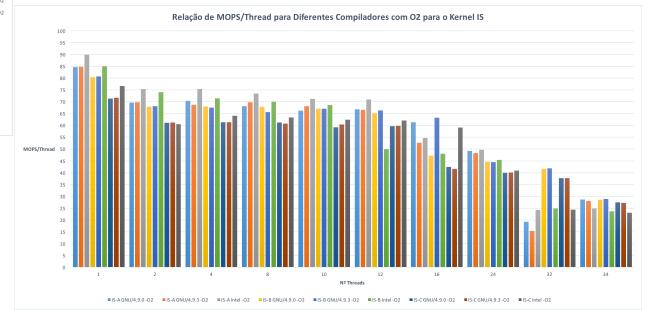


Figura 37. MOPS/Thread para Diferentes Compiladores com Flags -O2 para o Kernel IS

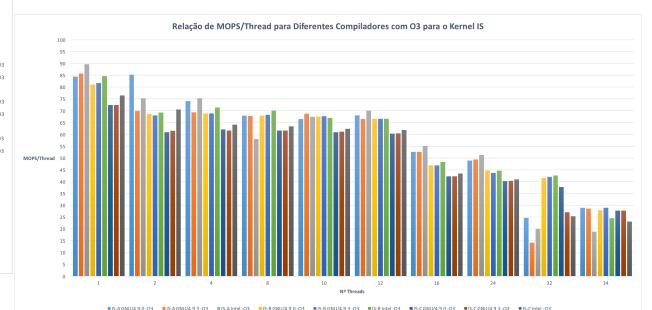


Figura 38. MOPS/Thread para Diferentes Compiladores com Flags -O3 para o Kernel IS

Figura 34. Tempos para Diferentes Compiladores com Flags -O3 para o Kernel IS

### 3 PThreads

{Página em Branco (Fazer scroll)}

# Memória Partilhada: Programação com PThreads

Sérgio Caldas  
*Universidade do Minho*  
*Escola de Engenharia*  
*Departamento de Informática*  
*Email: a57779@alunos.uminho.pt*

## Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Ambiente de testes</b>	<b>1</b>
<b>3</b>	<b>1<sup>a</sup> Parte - Tempo Médio de Criação de um Fio de Execução</b>	<b>1</b>
<b>4</b>	<b>2<sup>a</sup> Parte - Regra Trapezoidal [1]</b>	<b>2</b>
4.1	Caso de Estudo . . . . .	2
4.2	Análise dos Resultados . . . . .	3
4.3	Busy Waiting . . . . .	3
4.4	Semáforos . . . . .	3
4.5	Mutexes . . . . .	3
<b>5</b>	<b>Conclusão</b>	<b>4</b>
<b>Referências</b>		<b>4</b>

**Resumo**—Este trabalho baseia-se essencialmente na programação em memória partilhada, recorrendo a *POSIX Threads*, mais conhecidas como *PThreads*, estas são muito populares no ambiente *UNIX*. O trabalho divide-se em duas partes, na primeira parte medi o tempo médio de criação de uma *Thread* e a segunda parte desenvolvi um algoritmo (*Trapezoidal Rule*) com recurso a *PThreads*.

## 1. Introdução

Este trabalho, desenvolvido no âmbito da disciplina de Engenharia Sistemas de Computação (ESC), inserida no perfil de Computação Paralela e Distribuída (CPD) do curso de MIEI<sup>1</sup>, consiste no desenvolvimento de programas com recurso a *PThreads*. Este trabalho é dividido em duas partes, na primeira parte desenvolvi um pequeno programa (tipo *Hello World*) em que me media o tempo médio da criação de um fio de execução (*Thread*). Na segunda parte desenvolvi o algoritmo da regra trapezoidal, com recurso a *PThreads*, em que media o tempo de execução para um dado intervalo, no algoritmo existe uma variável que irá ser partilhada por todas as *Threads* e para evitar *Data Races* implementei as

1. Mestrado Integrado em Engenharia Informática

System	MacBook Pro (15-inch, Mid 2010)	
# CPUs	1	
CPU	Intel® Core™ i5-520M	
Architecture	Arrandale	
# Cores per CPU	2	
# Threads per CPU	4	
Clock Freq.	2.53 GHz	
L1 Cache	64 KB	32 KB por core
L2 Cache	512 KB	256 KB por core
L3 Cache	3 MB	
Inst. Set Ext.	SSE4.1/4.2	
#Memory Channels	2	
Memory BW	17.1 GB/s	

Tabela 1. CARACTERIZAÇÃO DA MÁQUINA PESSOAL

técnicas que estudamos para forçar a exclusão mutua, mais precisamente *Busy Waiting*, *Mutex* e Semáforos.

Neste relatório encontra-se a análise dos resultados obtidos, tanto para a primeira parte do trabalho, como para a segunda.

## 2. Ambiente de testes

Depois de desenvolvidos os dois algoritmos (relativos à primeira e segunda parte) efetuei testes na minha máquina pessoal e no *Search*, mais precisamente numa máquina 431. A metodologia de testes por mim escolhida foi a de *k-best solution*, esta metodologia baseia-se essencialmente na execução do *Kernel* de *k* vezes escolhendo a melhor solução dessas *k* vezes. O *k* por mim escolhido foi de 5. De referir que foram feitos testes para 1, 2, 4, 8, 16, 32 e 64 *Threads*, sendo que para cada um destes números de *Threads* o *Kernel* foi executado 5 vezes (*k-best solution*).

Na tabela 1 e na tabela 2, encontra-se a caracterização das duas máquinas de teste, isto é, a caracterização da minha máquina e da máquina 431 do *Search*, respetivamente.

## 3. 1<sup>a</sup> Parte - Tempo Médio de Criação de um Fio de Execução

Nesta primeira parte do trabalho, desenvolvi um pequeno algoritmo que cria um determinado número de *Threads*, executa uma função (que não faz nada) por cada *Thread*,

System	Máquina 431
# CPUs	2
CPU	Intel® Xeon® X5650
Architecture	Nehalem
# Cores per CPU	6
# Threads per CPU	12
Clock Freq.	2.66 GHz
L1 Cache	192 KB 32 KB por core
L2 Cache	1536 KB 256 KB por core
L3 Cache	12 MB
Inst. Set Ext.	SSE4.2 e AVX
#Memory Channels	3
Memory BW	32 GB/s

Tabela 2. CARACTERIZAÇÃO DA MÁQUINA 431

depois faz *join* a todas as *Threads* e termina o programa. O objetivo deste algoritmo é calcular o tempo médio de criação de um fio de execução (*Thread*), de forma a poder comparar o comportamento do tempo, quando se aumenta o número de *Threads*.

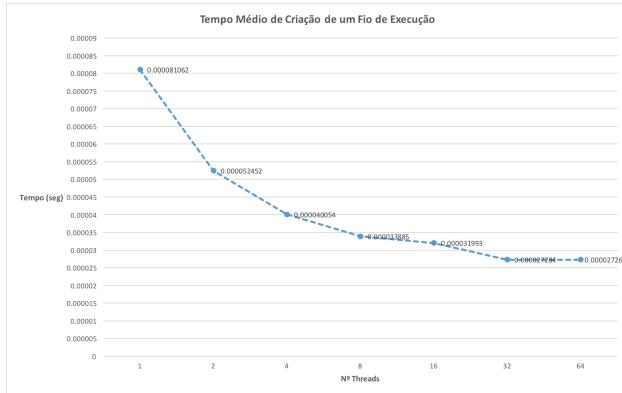


Figura 1. Tempo Médio da Criação de um Fio de Execução

No gráfico da figura 1 podemos encontrar os resultados obtidos na minha máquina. Como podemos ver pela figura, o tempo médio, quando se cria uma única *Thread* é de 0.000081062 segundos, à medida que vamos aumentando o número de *Threads* o tempo médio vai diminuindo, atingindo 0.000027265 segundos com 64 *Threads*, sendo portanto, um ganho de aproximadamente 3.

Relativamente aos resultados do *Search*, o gráfico da figura 2 ilustra os resultados obtidos. A semelhança do que acontece na minha máquina pessoal, os tempos médios diminuem à medida que se aumenta o número de *Threads*, sendo que com 16 *Threads* é quando é atingido o menor tempo médio, subindo ligeiramente a partir dai até às 64 *Threads*.

Em resposta as perguntas efectuadas pelo professor no enunciado, posso dizer que o número de fios de execução (*Threads*) criados, afeta o tempo médio, como podemos constatar pela figura, quanto mais se aumenta o número de *Threads* menor é o tempo médio para a criação de uma *Thread*.

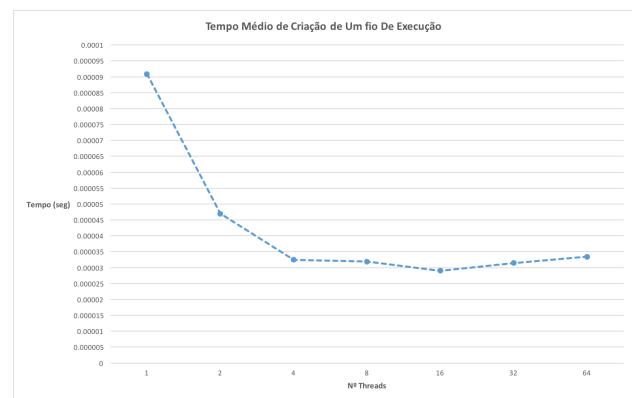


Figura 2. Tempo Médio da Criação de um Fio de Execução

## 4. 2<sup>a</sup> Parte - Regra Trapezoidal [1]

### 4.1. Caso de Estudo

A **Regra Trapezoidal** é uma integração numérica, cujo principal objetivo é aproximar o valor de um integral definido, de uma função, em que não é necessário usar a expressão analítica para a primitiva da função.

Este método usa três fases:

- 1) Decompõe-se o domínio em sub-intervalos;
- 2) Calcula-se uma integração aproximada para cada sub-intervalo;
- 3) Soma de todos os resultados obtidos.

Este método é usado, porque nem todas as funções tem uma primitiva de forma explícita, é difícil avaliar a primitiva de uma função e sobretudo quando não é possível ter uma expressão analítica para o integral, mas conhece-se os seus valores num conjunto de valores de um intervalo.

A formula matemática para a Regra Trapezoidal é dada por:

$$\int_a^b f(x)dx \simeq (b-a) \frac{f(a) + f(b)}{2}$$

Para se perceber melhor a Regra Trapezoidal, suponhamos que temos:

$$x = a$$

$$x = b$$

e

$$x_{n+1} = x_n + h$$

em que

$$h = \frac{(b-a)}{n}$$

em que  $n$  é número de sub-intervalos então a Regra Trapezoidal composta é:

$$\int_a^b f(x)dx \simeq \frac{h}{2} [f(x_1) + 2f(x_2) + 2f(x_3) + \dots + f(x_n)]$$

## 4.2. Análise dos Resultados

O algoritmo usa *PThreads*. Depois de criadas  $x$  *Threads*, cada *Thread* calcula uma aproximação para um sub-intervalo e no fim soma aos outros resultados das outras *Threads*, de referir que é nessa operação que a variável é partilhada pelas *Threads* (região crítica). Na região crítica apliquei as diferentes formas de forçar a exclusão mutua que estudamos (Busy Waiting, mutex e semáforos) de maneira a evitarmos *Data Races*. Depois de desenvolvido o algoritmo, procedi a realização de vários testes, como referi em cima foram feitos 5 execuções para cada número de *Threads* (1, 2, 4, 8, 16, 32 e 64) e extraí o melhor tempo dessas medições. Os meus valores de entrada foram:  $a=0$ ,  $b=1024$ , sendo que  $n$  varia, correspondendo ao número de *Threads*.

O resultado do algoritmo para estes valores é de 357914112. Os resultados obtidos na minha máquina podem ser consultados no gráfico da figura 3.

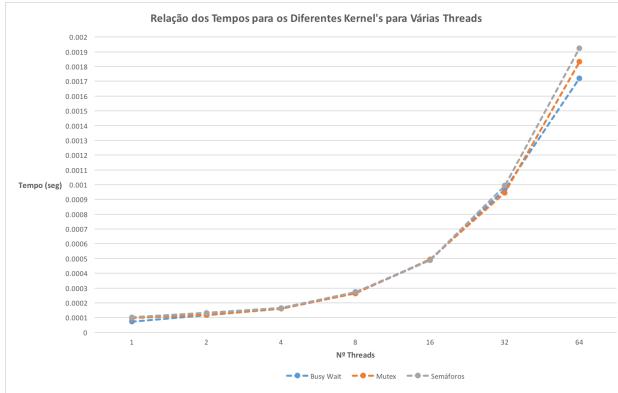


Figura 3. Relação dos Tempos para os Diferentes Kernel's para Várias Threads

Ao analisarmos o gráfico da figura 3, podemos constatar que o comportamento dos métodos de exclusão mutua, à medida que se aumenta o número de *Threads* é praticamente semelhante para os 3. De referir que com um número de *Threads* de 64 o *Busy Waiting* é o mais eficiente seguido do *Mutex* e por fim dos *Semáforos*. Com os resultados que obtive na minha máquina não posso concluir que haja um método que seja melhor que os outros todos, uma vez que há alguma variação, por exemplo, com 32 *Threads* o método *Mutex* é melhor que os outros mas com 64 *Threads* já é o método *Busy Waiting* e com 16 *Threads* já é o método dos *Semáforos* (apesar da diferença ser mesmo muito reduzida em relação aos outros).

Os resultados obtidos na máquina 431 do *Search* estão ilustrados no gráfico da figura 4. Ao analisarmos o gráfico, podemos constatar que o comportamento dos diferentes *Kernel's* é semelhante ao comportamento destes quando executados na minha máquina, ou seja, à medida que se aumenta o número de *Threads* o tempo de execução dos *Kernel's* também tende a aumentar. Mais uma vez, e também à semelhança do que aconteceu na minha máquina, com estes resultados não podemos concluir que um *Kernel* é

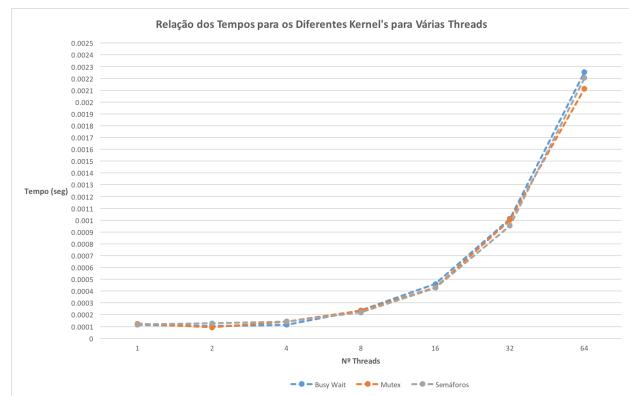


Figura 4. Relação dos Tempos para os Diferentes Kernel's para Várias Threads

melhor que o outro, pois o comportamento dos 3 é muito semelhante. Por exemplo o *Kernel Busy Waiting* é melhor com 4 *Threads*, já o *Kernel Semáforos* é o melhor com 16 e 32 *Threads* e por fim o *Kernel Mutex* é o melhor com 64 *Threads*, posto isto não posso afirmar com certeza qual é o melhor método para forçar a exclusão mutua.

## 4.3. Busy Waiting

- **Vantagens**

- Os algoritmos de *Busy Waiting* são fáceis de implementar em qualquer máquina.

- **Desvantagens**

- Estes algoritmos são da inteira responsabilidade do programador;
- Como existem vários processos á espera, a ocupação do *CPU* acaba por ser um desperdício de recursos. Estes processos poderiam ser bloqueados.

## 4.4. Semáforos

- **Vantagens**

- Em *POSIX* os semáforos podem sincronizar processos com ou sem memória partilhada.

- **Desvantagens**

- Uma má programação pode levar a resultados inesperados;
- Obliga o programador a inserir explicitamente instruções *sem\_wait* ou *sem\_post*.

## 4.5. Mutexes

- **Vantagens**

- Versão simplificada de um semáforo (não precisa de contar);

- Só necessita de 1 bit para representar a variável mutex (livre, ocupado);
- São eficientes e fáceis de implementar;
- Se bem implementados as hipóteses de haver erros é quase 0;
- Há um melhor aproveitamento do processador.

- **Desvantagens**

- São adequados apenas para organizar a exclusão mútua de algum recurso ou parte de código partilhada

## 5. Conclusão

As *PThreads* ou *POSIX Threads*, como já foi dito em cima, são muito utilizadas em ambientes *Unix*, para criação de *Threads*, com estas podemos facilmente paralelizar o código, de forma a tirar o melhor partido dos recursos que temos disponíveis. Como vimos nas análises dos resultados obtidos para a primeira parte deste trabalho quanto mais *Threads* criámos, menor é o seu tempo de criação.

Para se paralelizar um código usando *PThreads*, é necessário ter em atenção as regiões críticas, estas não podem ser acedidas por mais que uma *Thread* ao mesmo tempo, para isso é necessário implementar mecanismos de exclusão mútua (*Busy Waiting*, *Mutex* e Semaforos). Estes 3 mecanismos, como vimos, foram implementados e testados em duas máquinas, a minha máquina pessoal e a máquina 431 do *Search*, das quais obtive os resultados em cima apresentados, resultados esses que não me permitem concluir qual é o mecanismo/método mais eficiente para implementar exclusão mutua.

Apesar de não ter obtido dados muito esclarecedores, penso que a implementação de exclusão mutua será mais eficiente/segura se usarmos semáforos ou *Mutex*, embora o *Mutex* seja apenas adequado para uma parte de código ou algum recurso partilhado. Penso que estes dois métodos são os mais eficientes/seguros pois são os que tem menor margem de erro e como tal a ocorrência de *Data Races* é pouco provável, para além disso, permite-nos tirar um maior partido dos recursos que dispomos.

Na realização deste trabalho não me deparei com grandes dificuldades, uma vez que era um trabalho simples, e curto. Contudo confesso que não estou satisfeito com os meus resultados, isto porque, não era bem estes resultados que estava à espera. Esperava que os tempos de execução dos 3 *Kernel's* fossem mais esclarecedores do que aqueles que obtive, posto isto como trabalho futuro pretendo debruçar-me melhor sobre o algoritmo, apesar de achar que este está bem implementado, mas gostava de perceber o porquê de estes resultados não terem sido os que esperava. Quanto ao restante trabalho faço um balanço bastante positivo.

## Referências

[1] Integração numérica. [https://pt.wikipedia.org/wiki/Integracao\\_numERICA](https://pt.wikipedia.org/wiki/Integracao_numERICA).

## 4 DTrace

{Página em Branco (Fazer scroll)}

# Exercícios sobre a Ferramenta Dtrace

Sérgio Caldas  
*Universidade do Minho*  
*Escola de Engenharia*  
*Departamento de Informática*  
*Email: a57779@alunos.uminho.pt*

## Conteúdo

<b>1</b>	<b>Introdução</b>	1
<b>2</b>	<b>Dtrace</b>	1
<b>3</b>	<b>Exercicio 1</b>	1
3.1	Script . . . . .	2
3.2	Resultados Obtidos . . . . .	2
<b>4</b>	<b>Exercicio 2</b>	3
4.1	Script . . . . .	3
4.2	Resultados Obtidos . . . . .	3
<b>5</b>	<b>Análise de Aplicação Paralela (OpenMP) com Ferramenta Dtrace</b>	4
5.1	Análise dos Tempos para os Diferentes Tipos de Escalonamento . . . . .	5
5.2	Análise do Comportamento das Threads para os Diferentes Tipos de Escalonamento . . . . .	5
<b>6</b>	<b>Conclusão</b>	6

**Resumo**—Este relatório, exprime os resultados obtidos na resolução de exercícios sobre *Dtrace*, no âmbito da disciplina de Engenharia de Sistemas de Computação (ESC), inserida no perfil de Computação Paralela e Distribuída (CPD) do curso de Engenharia Informática. O objetivo deste trabalho é praticar o uso do *Dtrace*, numa máquina *Soláris 11*, para isso foram propostos dois exercícios, sendo que os seus resultados são apresentados ao longo deste relatório.

## 1. Introdução

Como foi dito anteriormente, foram propostos dois exercícios para a praticar o uso do *Dtrace*. O primeiro exercício consiste em desenvolver uma script em *D* que faça um traçado das chamadas ao sistema *open()* (no caso de uma máquina *Soláris 11* *openat()*), imprimindo por linha o nome do ficheiro executável, PID do processo, UID do utilizador e GID do grupo, o caminho absoluto para o ficheiro que for aberto, a cadeia de caracteres com as *flags* da chamada ao sistema *openat()* (*O\_RDONLY*, *O\_WRONLY*,

*O\_RDWR*, *O\_APPEND*, *O\_CREAT*) e por fim o valor de retorno de chamada ao sistema. Este exercício contem um parte opcional, que consiste em modificar a script para sejam apenas detetados os ficheiros com *etc/* no caminho.

O segundo exercício proposto consiste, para todos os processos que estão no sistema, em contar o número de tentativas de abrir ficheiros existentes, o número de tentativas para criar ficheiros e contar o número de tentativas bem-sucedidas. Posteriormente a script deve imprimir, com um período (especificado em segundos) passado como argumento na linha de comandos, a hora e dia atual em formato legível e as estatísticas recolhidas por PID e respetivo nome.

Para além dos exercícios em cima referidos, no final do semestre ainda foi sugerido um exercício extra. Para a execução deste exercício, foi-nos fornecido um código paralelo, que gera um números aleatórios com gama num intervalo dado. Para além desse código foi-nos fornecido também uma *script DTrace* que faz um traçado dinâmico do código referido anteriormente. O objetivo deste trabalho extra passa por analiármos o comportamento das *threads* e da aplicação para os diferentes tipos de escalonamento.

## 2. Dtrace

O *Dtrace* é uma *Framework* para fazer traçados dinâmicos, esta *Framework* é usada para solucionar problemas no *Kernel* e aplicações em produção, em tempo real.

O *Dtrace* pode ser utilizado para se obter uma visão geral da execução do sistema, como a quantidade de memória, o tempo de CPU, os recursos usados por os processos activos. Esta *Framework* permite fazer traçados muito mais rebuscados e detalhados, tais como, por exemplo a lista de processos que tenta aceder a um ficheiro.

Os administradores de sistemas escrevem programas em *D*, ajustando esses programas à informação que querem obter. A linguagem *D*, em termos de estrutura é muito semelhante ao *Awk*. Estes programas em *D*, consistem numa lista de uma ou mais provas e a cada prova está associada uma acção.

## 3. Exercicio 1

Para este exercício, desenvolvi uma script em *D*, script essa que faz um traçado das chamadas ao sistema *openat()*

e imprime a seguinte informação por linha:

- Nome do ficheiro executável;
- PID do Processo;
- UID do Utilizador;
- GID do Grupo;
- Caminho absoluto para o ficheiro que for aberto;
- A cadeia de caracteres com as *flags* da chamada ao sistema openat() (O\_RDONLY, O\_WRONLY, O\_RDWR, O\_APPEND, O\_CREAT);
- O valor de retorno da chamada ao sistema.

Para além de cada um destes tópicos, foi-nos proposto como exercício opcional, modificar a script de forma a só ser detetados os ficheiros com */etc* no caminho.

### 3.1. Script

A *Script* por mim criada contém 2 provas, a primeira deteta à entrada a chamada ao sistema e guarda o *arg1* (que contém o caminho) na variável *self->path* bem como o *arg2* (que contém a *flag*) na variável *self->flags*. Na segunda prova no inicio, contém o predicado */strstr(self->path, "/etc") != NULL/* que permite apenas detectar os ficheiros com */etc* no caminho. Posteriormente nesta prova faço *strjoin* à variável *this->flags\_string* da string *O\_WRONLY* caso a condição *self->flags & O\_WRONLY* se verifique, caso contrário verifica se a condição *self->flags & O\_RDWR* é verdadeira e faz *strjoin* da string *O\_RDWR*, caso a condição seja falsa faz *strjoin* da string *O\_RDONLY*. Ainda nesta acção faço *strjoin* da string *|O\_APPEND* caso a condição *self->flags & O\_APPEND* se verifique, caso contrário faço *strjoin* da string *,*, o mesmo se acontece para a *flag O\_CREAT*.

No fim é imprimido no ecrã o nome do executável o PID do processo, o UID do utilizador, o GID do grupo, o caminho absoluto para o ficheiro que for aberto, a string com o conteúdo da variável *this->flag\_string* e por fim o valor de retorno da chamada ao sistema.

```
/*
inline int O_RDONLY = 0;
inline int O_WRONLY = 1;
inline int O_RDWR = 2;
inline int O_APPEND = 8;
inline int O_CREAT = 256;
*/

this string flag_string;

syscall::openat:entry
{
    self->path = copyinstr(arg1);
    self->flags = arg2;
}

syscall::openat:return
/*strstr(self->path, "/etc") != NULL/
{
    this->flags_string = strjoin(
        self->flags & O_WRONLY
        ? "O_WRONLY"
        : self->flags & O_RDWR
        ? "O_RDWR"
        : "O_RDONLY",
        strjoin(
            self->flags & O_APPEND ? "|O_APPEND" : "",
            self->flags & O_CREAT ? "|O_CREAT" : ""));
}

printf("Executável: %d,%d,%d, \"%s\",%d,%d\n", execname, pid, uid, gid<-
    ,self->path, this->flags_string, arg1);
}
```

### 3.2. Resultados Obtidos

Para testar a script executei alguns comandos de teste, propostos no enunciado, sendo que o resultado de cada comando pode ser consultado nas subsecções em baixo.

- Comando *cat /etc/inittab > /tmp/test* - Como podemos verificar pela tabela da figura 1, ao executarmos o comando *cat /etc/inittab > /tmp/test* na consola, este é logo detetado pelo *Dtrace* (1<sup>a</sup> linha da tabela). Como ao executarmos o comando reencaminhados o *Output* para um novo ficheiro, na coluna das *flags* aparece a *flag O\_WRONLY* que corresponde a leitura do ficheiro *inittab* e a *flag O\_CREAT* que corresponde a criação do ficheiro *test\_sergio*.

Execname	PID	UID	GID	Path	Flags	Valor de Retorno
bash	22177	29231	5000	/tmp/test_sergio	O_WRONLY O_CREAT	4
cat	22177	29231	5000	/var/lib/did.config	O_RDONLY	-1
cat	22177	29231	5000	/lib/libc.so.1	O_RDONLY	3
cat	22177	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	22177	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/etc/resolv.conf	O_RDONLY	10

Figura 1. Resultado Comando *cat /etc/inittab > /tmp/test*

- Comando *cat /etc/inittab >> /tmp/test* - Ao executarmos este comando estamos a fazer uma leitura do ficheiro *inittab*. Além disso este comando acrescenta o *Output* ao ficheiro *test\_sergio* já existente, como tal é de esperar que na coluna referente às *flags* apareçam as *flags O\_WRONLY, O\_APPEND* e a *flag O\_CREAT*, o que se verifica pela análise da 1<sup>a</sup> linha da tabela da figura 2.

Execname	PID	UID	GID	Path	Flags	Valor de Retorno
bash	22182	29231	5000	/tmp/test_sergio	O_WRONLY O_APPEND O_CREAT	
cat	22182	29231	5000	/var/lib/did.config	O_RDONLY	-1
cat	22182	29231	5000	/lib/libc.so.1	O_RDONLY	3
cat	22182	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	22182	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1

Figura 2. Resultado Comando *cat /etc/inittab >> /tmp/test*

- Comando *cat /etc/inittab | tee /tmp/test* - Ao executarmos este comando a saída do comando *cat /etc/inittab* vai ser gravada no ficheiro *test* ao mesmo tempo em que é exibida no ecrã. O resultado obtido pela execução deste comando pode ser consultado na tabela da figura 3.
- Comando *cat /etc/inittab | tee -a /tmp/test* - Esse comando faz exactamente o que faz o comando do tópico anterior, com a diferença que com a *flag -a* este acrescenta a saída do comando *cat /etc/inittab* ao ficheiro *test*. O resultado obtido pela execução

Execname	PID	UID	GID	Path	Flags	Valor de Retorno
cat	22187	29231	5000	/var/lib/d/d.config	O_RDONLY	-1
cat	22187	29231	5000	/lib/libc.so.1	O_RDONLY	3
cat	22187	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	22187	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
tee	22188	29231	5000	/var/lib/d/d.config	O_RDONLY	-1
tee	22188	29231	5000	/lib/libc.so.1	O_RDONLY	3
tee	22188	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
tee	22188	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
tee	22188	29231	5000	/usr/lib/locale/en_US.UTF-8/LC_MESSAGES/SUNW_OSLIB.mo	O_RDONLY	-1
nfsmapid	1351	1	12	/etc/resolv.conf	O_RDONLY	10

Figura 3. Resultado Comando cat /etc/inittab | tee /tmp/test

deste comando pode ser consultado na tabela da figura 4.

Execname	PID	UID	GID	Path	Flags	Valor de Retorno
tee	22197	29231	5000	/var/lib/d/d.config	O_RDONLY	-1
tee	22197	29231	5000	/lib/libc.so.1	O_RDONLY	3
tee	22197	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
tee	22197	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
tee	22197	29231	5000	/usr/lib/locale/en_US.UTF-8/LC_MESSAGES/SUNW_OSLIB.mo	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
cat	22196	29231	5000	/var/lib/d/d.config	O_RDONLY	-1
cat	22196	29231	5000	/lib/libc.so.1	O_RDONLY	3
cat	22196	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	22196	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3

Figura 4. Resultado Comando cat /etc/inittab | tee -a /tmp/test

## 4. Exercício 2

Neste exercício, foi-nos proposto criar uma script que para cada processo em execução no sistema mostra-se as seguintes estatísticas:

- Número de tentativas de abrir ficheiros existentes;
- Número de tentativas para criar ficheiros;
- Número de tentativas bem-sucedidas.

Estas estatísticas devem ser imprimidas no ecrã, juntamente com a hora e dia, repetidamente, com um período (em segundos) passado como argumento na linha de comandos.

### 4.1. Script

Esta *Script* é constituída por 3 provas, primeira vai contar o número de tentativas para criar ficheiros, para isso esta prova contém um predicado que testa se a flag da chamada ao sistema *openat* é a flag *O\_CREAT*, caso se verifique este predicado é incrementado um contador através de uma agregação com um array associativo (*@create[execname,pid]*). As outras duas provas, uma conta o numero de tentativas para abrir um ficheiro e a outra o numero de tentativas bem sucedidas, as ações associadas

a estas provas processam-se da mesma forma que foi explicado atrás. No final é imprimido no ecrã o dia e a hora, o nome do executável, o PID, o contador do array *@create*, o contador do array *@open* e o contador do array *@success*, com um período de 2 segundos

```

/*
inline int O_RDONLY = 0;
inline int O_WRONLY = 1;
inline int O_RDWR = 2;
inline int O_APPEND = 8;
inline int O_CREAT = 256;
*/

syscall::openat::entry
/(*arg2 & O_CREAT)==O_CREAT/
{
    @create[execname,pid] = count();
}

syscall::openat*::entry
/(*arg2 & O_CREAT) == 0/ {
    @open[execname, pid] = count();
}

syscall::openat*::return
/ arg1 > 0 / {
    @success[execname, pid] = count();
}

tick-$ls
{
    printf ("%Y\n", walltimestamp);
    printf("%12s %6s %6s %s\n", "EXECNAME", "PID", "CREATE", "OPEN", "←
        SUCCESS");
    printa ("%12s %6d %6d %6d %6d\n", @create, @open, @success);
    trunc(@create);
    trunc(@open);
    trunc(@success);
}

```

### 4.2. Resultados Obtidos

Os resultados obtidos na execução da *script*, referente ao exercício 2, podem ser consultados na figura 5. Estes resultados foram obtidos através da execução do seguinte comando:

```
dtrace -qs exercicio2.d 2
```

Sendo que 2, é o período (segundos) pelo qual os resultados são imprimidos no ecrã. De referir que nem sempre as provas detectam alguma coisa, pode haver períodos em que não há nenhuma atividade que satisfaça as provas definidas.

EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:20				
sshd	23318	0	5	6
bash	23320	0	6	3
nscd	1447	0	12	12
sshd	23319	0	20	19
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:22				
dtrace	23375	0	2	2
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:24				
utmpd	259	0	1	2
ls	23376	0	5	4
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:26				
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:28				
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:30				
bash	23334	0	1	1
nfsmapid	1351	0	2	0
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:32				
automountd	1443	0	1	1
bash	23334	0	1	1
nfsmapid	1351	0	3	1
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:34				
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:36				
bash	23334	0	1	1
nfsmapid	1351	0	4	0
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:38				
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:40				
bash	23334	0	1	1
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:42				
bash	23334	0	1	1
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:44				
ls	23377	0	5	4
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:46				
bash	23334	0	1	1
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:48				
bash	23334	0	2	2
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:50				
rm	23378	0	4	3
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 17:45:52				

## 5. Análise de Aplicação Paralela (OpenMP) com Ferramenta Dtrace

Nesta secção irei analisar, a execução de uma aplicação paralela, fornecida pelo professor, com uma *script* em *Dtrace*. A aplicação em causa é um gerador de números aleatórios, numa gama de um intervalo dado. A aplicação está paralelizada seguindo um paradigma de memória partilhada, mais propriamente com a utilização de pragmas *OpenMP*. O código referente à aplicação fornecida pelo professor pode ser consultado em baixo:

```

/*
 * APina - 2016
 */
/* 
 * compilar com: g++-mp-5 -std=c++11 -Wall -O2 -fopenmp -o ex2_v2 ex2_v2.cxx
 * usar modos de escalonamento : export OMP_SCHEDULE="guided", OMP_SCHEDULE="dynamic"
 */
/*
 * Execu   o:
 * ./ex2_v2 <n.threads><opcional- intervalo>
 */
#include <cstdlib>
#include <iostream>
#include <random>
using namespace std;
#include <omp.h>

#define S 1024*1024
//#define S 100

int main(int argc, char *argv[])
{
    int i, r, a[S], np, nr;
    double T1,T2;

    np = atoi(argv[1]);
    if (argc == 2) nr= 99; else nr= atoi(argv[2]);

    std::random_device d;
    std::default_random_engine el(d());
    // a distribution that takes randomness and produces values in specified range
    std::uniform_int_distribution<int> dist(1,np);

    omp_set_num_threads(np);
    T1 = omp_get_wtime();
#pragma omp parallel for private (r) schedule (runtime)
    for (i=0 ; i < S ; i++) {
        a[i] = 0;
        for (r = dist(el) ; r > 0 ; r -= 20) {
            a[i] += r;
        }
    }
    T2 = omp_get_wtime();
    cout << "flosXecucao =" << np << " Intervalo=" << nr << " tempo -> " << (T2->T1)*le6 << " usecs\n";
}

```

A *script DTrace*, também fornecida pelo professor, permite-nos obter vários tipos de informação relativamente à aplicação em causa. A *script* permite-nos saber quando a *Thread* foi criada e finalizada, em que CPU está a correr (se trocar de CPU também é possível observar essa troca), quando é interrompida e como é interrompida. A *script* em *DTrace* pode ser consultada em baixo:

```

#!/usr/sbin/dtrace -
#pragma D option quiet
BEGIN
{
    baseline = walltimestamp;
    scale = 1000000;
}
sched:::on-cpu
/pid == $target && !self->stamp /
{
    self->stamp = walltimestamp;
    self->lastcpu = curcpu->cpu_id;
    self->lastlgrp = curcpu->cpu_lgrp;
    self->stamp = (walltimestamp - baseline) / scale;
    printf("%d:%d %d TID %d CPU %d created\n",
           self->stamp, 0, tid, curcpu->cpu_id, curcpu->cpu_lgrp);
}

```

```

/* ustack(); */
}

sched:::on-cpu
/pid == $target && self->stamp && self->lastcpu\
  != curcpu->cpu_id/
{
    self->delta = (walltimestamp - self->stamp) / scale;
    self->stamp = walltimestamp;
    self->stamp = (walltimestamp - baseline) / scale;
    printf("%9d:%-9d TID %3d from-CPU %d(%d)",self->stamp,
           self->delta, tid, self->lastcpu, self->lastlgrp);
    printf("to-cpu %d(%d) CPU migration\n",
           curcpu->cpu_id, curcpu->cpu_lgrp);
    self->lastcpu = curcpu->cpu_id;
    self->latgrp = curcpu->cpu_lgrp;
}
sched:::on-cpu
/pid == $target && self->stamp && self->lastcpu\
  == curcpu->cpu_id/
{
    self->delta = (walltimestamp - self->stamp) / scale;
    self->stamp = walltimestamp;
    self->stamp = (walltimestamp - baseline) / scale;
    printf("%9d:%-9d TID %3d CPU %3d(%d) ",self->stamp,
           self->delta, tid, curcpu->cpu_id, curcpu->cpu_lgrp);
    printf("restarted on the same CPU\n");
}
sched:::off-cpu
/pid == $target && self->stamp /
{
    self->delta = (walltimestamp - self->stamp) / scale;
    self->stamp = walltimestamp;
    self->stamp = (walltimestamp - baseline) / scale;
    printf("%9d:%-9d TID %3d CPU %3d(%d) preempted\n",
           self->stamp, self->delta, tid, curcpu->cpu_id,
           curcpu->cpu_lgrp);
}
sched:::sleep
/pid == $target /
{
    self->sobj = (curlwpsinfo->pr_stype == SOBJ_MUTEX ?
      "kernel mutex" : curlwpsinfo->pr_stype == SOBJ_RWLOCK ?
      "kernel RW lock" : curlwpsinfo->pr_stype == SOBJ_CV ?
      "cond var" : curlwpsinfo->pr_stype == SOBJ_SEMA ?
      "kernel semaphore" : curlwpsinfo->pr_stype == SOBJ_USER ?
      "user-level lock" : curlwpsinfo->pr_stype == SOBJ_USER_PI ?
      "user-level PI lock" : curlwpsinfo->pr_stype == SOBJ_SHUTTLE ?
      "shuttle" : "unknown");
    self->delta = (walltimestamp - self->stamp) / scale;
    self->stamp = walltimestamp;
    self->stamp = (walltimestamp - baseline) / scale;
    printf("%9d:%-9d TID %3d sleeping on %s\n",
           self->stamp, self->delta, tid, self->sobj);
/* @sleep(curlwpsinfo->pr_stype, curlwpsinfo->pr_state, ustack())=count(); -->
   */
}
sched:::sleep
/pid == $target && ( curlwpsinfo->pr_stype == SOBJ_CV ||
  curlwpsinfo->pr_stype == SOBJ_USER ||
  curlwpsinfo->pr_stype == SOBJ_USER_PI) /
{
    /* ustack() */
}
sched:::sleep
/pid!=\$pid && 0/
{
    @sleep(execname,curlwpsinfo->pr_stype, curlwpsinfo->pr_state, ustack())=-->
      count();
}

```

O objetivo desta análise passa por analisar o comportamento da aplicação para os diferentes tipos de escalonamento:

- **static** - Este tipo de escalonamento divide o ciclo para *chunks* iguais ou o mais igual possível, caso o numero de iterações do ciclo não seja divisível por o número de *Threads* multiplicado pelo tamanho do *chunk*.
  - **dynamic** - Com este escalonamento é distribuído *chunck's*, com tamanho fixo, pelo numero de *threads* e estes segmentos são organizados numa *queue*. Quando um segmento termina é atribuido outro segmento com o mesmo tamanho. Este tipo de escalonamento tem um *overhead* associado.
  - **guided** - Este tipo de escalonamento é semelhante ao *dynamic*, contudo o tamanho do *chunk* começa grande e diminuindo de forma a melhorar o balanceamento das cargas entre as iterações. Por defeito o tamanho do *chunk* é aproximadamente

*loop\_count/number\_of\_threads.*

Para a execução e teste da aplicação criei uma *shell script* em que testei todos os tipos de escalonamento (*static*, *guided*, *dynamic*) para um numero variável de *Threads* (1, 2, 4, 8, 16, 32 e 64), recolhendo para cada número de *threads* um total de 10 amostras, das quais escolhi a amostra com melhor tempo.

## **5.1. Análise dos Tempos para os Diferentes Tipos de Escalonamento**

No gráfico da figura 6, podemos consultar os diferentes tempos para os diferentes tipos de escalonamento para diferentes números de *threads*.

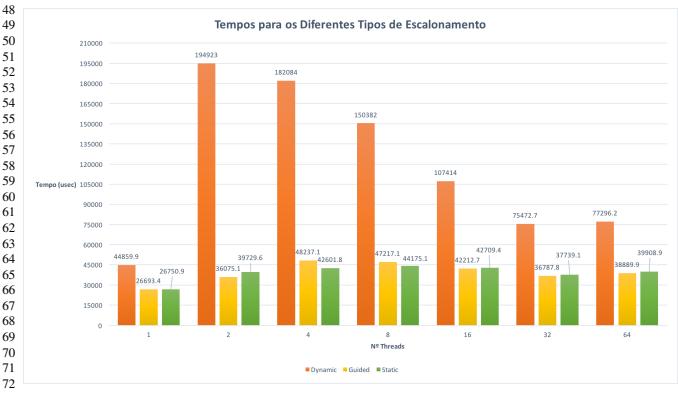


Figura 6. Tempos para os Diferentes Tipos de Escalonamento para Diferentes N° de Threads

Ao analisarmos o gráfico, verificamos que o escalonamento *dynamic* é sempre o que apresenta piores tempos, isto pode ser justificado, pelo facto deste escalonamento ter um *overhead* associado. Este *overhead* está associado ao tempo que é necessário para se fazer as divisões das iterações em blocos e posteriormente organiza-las na *queue*.

Quanto aos outros dois tipos de escalonamento (*guided*, *static*), estes apresentam tempos relativamente iguais à medida que se aumenta o número de *threads*. Como tal não me é possível concluir qual a melhor política de escalonamento em termos de tempo, uma vez que estes tempos são semelhantes para o *static* e para o *guided*.

Relativamente aos resultados obtidos para 64 *threads*, tenho de admitir que os resultados, são surpreendentes, uma vez que quando a aplicação é executada, estamos a usar o dobro das *threads* disponíveis na máquina. Como tal, seria de esperar que os tempos fossem um pouco superiores, uma vez que a máquina está a usar *threads* partilhadas, e como podemos verificar pela a análise do gráfico da figura 6, vemos que os tempos continuam mais ao menos semelhantes de quando a aplicação é executada com 32 *threads*.

Fazendo ainda uma análise ao paralelismo da aplicação, podemos verificar que é com a versão sequencial (*1 thread*) desta que se obtém o melhor tempo, relativamente

á versões com um maior número de *threads*. O que permite concluir que este código não é o melhor código para ser paralelizado uma vez que o melhor tempo é obtido sequencialmente.

## 5.2. Análise do Comportamento das Threads para os Diferentes Tipos de Escalonamento

Para a análise dos resultados, escolhi analisar um excerto do output da *Script Dtrace* da melhor execução, para 32 *threads* para o escalonamento *static* com um intervalo de 100. Escolhi 32 *threads* porque é o número máximo de *threads* suportado pela máquina.

```

0:0          TID 1 CPU 1(1) created
0:0          TID 1 CPU 1(1) created
fiosExecucao=32 Intervalo=99
tempo -> 37739.1 usecs
 9:0          TID 3 CPU 0(1) created
9:1465575350619 TID 3 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 3 sleeping on cond var
9:1465575350619 TID 3 CPU 0(1) preempted
 9:0          TID 7 CPU 0(1) created
9:1465575350619 TID 7 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 7 sleeping on cond var
9:1465575350619 TID 7 CPU 0(1) preempted
 9:0          TID 9 CPU 0(1) created
9:1465575350619 TID 9 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 9 sleeping on cond var
9:1465575350619 TID 9 CPU 0(1) preempted
 9:0          TID 11 CPU 0(1) created
9:1465575350619 TID 11 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 11 sleeping on cond var
9:1465575350619 TID 11 CPU 0(1) preempted
 9:0          TID 13 CPU 0(1) created
9:1465575350619 TID 13 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 13 sleeping on cond var
9:1465575350619 TID 13 CPU 0(1) preempted
 9:0          TID 15 CPU 0(1) created
9:1465575350619 TID 15 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 15 sleeping on cond var
9:1465575350619 TID 15 CPU 0(1) preempted
 9:0          TID 17 CPU 0(1) created
9:1465575350619 TID 17 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 17 sleeping on cond var
9:1465575350619 TID 17 CPU 0(1) preempted
 9:0          TID 19 CPU 0(1) created
9:1465575350619 TID 19 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 19 sleeping on cond var
9:1465575350619 TID 19 CPU 0(1) preempted
 9:0          TID 21 CPU 0(1) created
9:1465575350619 TID 21 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 21 sleeping on cond var
9:1465575350619 TID 21 CPU 0(1) preempted
 9:0          TID 23 CPU 0(1) created
9:1465575350619 TID 23 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 23 sleeping on cond var
9:1465575350619 TID 23 CPU 0(1) preempted
 9:0          TID 25 CPU 0(1) created
9:1465575350619 TID 25 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 25 sleeping on cond var
9:1465575350619 TID 25 CPU 0(1) preempted
 9:0          TID 27 CPU 0(1) created
9:1465575350619 TID 27 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 27 sleeping on cond var
9:1465575350619 TID 27 CPU 0(1) preempted
 9:0          TID 29 CPU 0(1) created
9:1465575350619 TID 29 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 29 sleeping on cond var
9:1465575350619 TID 29 CPU 0(1) preempted
 9:0          TID 31 CPU 0(1) created
9:1465575350619 TID 31 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 31 sleeping on cond var
9:1465575350619 TID 31 CPU 0(1) preempted
 9:0          TID 33 CPU 0(1) created
9:1465575350619 TID 33 CPU 0(1) restarted on the same CPU
9:1465575350619 TID 33 sleeping on cond var
9:1465575350649 TID 7 CPU 0(1) preempted
39:1465575350649 TID 7 sleeping on cond var
39:1465575350649 TID 7 CPU 0(1) preempted
49:1465575350659 TID 7 CPU 0(1) restarted on the same CPU
49:1465575350659 TID 7 sleeping on cond var
49:1465575350659 TID 7 CPU 0(1) preempted
49:1465575350659 TID 7 CPU 0(1) restarted on the same CPU
49:1465575350659 TID 7 sleeping on cond var
49:1465575350659 TID 7 CPU 0(1) preempted
49:1465575350659 TID 1 from-CPU 19(1) to-CPU 0(1) CPU migration
49:1465575350659 TID 1 CPU 0(1) restarted on the same CPU
49:1465575350659 TID 1 sleeping on cond var
49:1465575350659 TID 1 CPU 0(1) preempted
49:1465575350659 TID 1 CPU 0(1) restarted on the same CPU
49:1465575350659 TID 1 sleeping on cond var
49:1465575350659 TID 1 CPU 0(1) preempted
49:1465575350659 TID 1 CPU 0(1) restarted on the same CPU
49:1465575350659 TID 1 sleeping on cond var
49:1465575350659 TID 1 CPU 0(1) preempted
49:1465575350659 TID 1 CPU 0(1) restarted on the same CPU
49:1465575350659 TID 1 sleeping on cond var
49:1465575350659 TID 1 CPU 0(1) preempted

```

Escolhi apenas este excerto do output para o escalonamento *static* para 32 *threads*, uma vez que depois da analise dos outros outputs não verifica grandes diferenças.

Como podemos verificar pelo output em cima, as *threads* são criadas, em diferentes CPU's e à medida que o programa é executado, estas por sua vez podem ser interrompidas (*preempted*), ou seja saem do CPU, depois de interrompidas, as *threads* normalmente recomeçam a sua execução no mesmo CPU (*restarted on the same CPU*).

As *threads* podem ficar adormecidas (*sleeping*), isto acontece porque são forçadas a parar pelas variáveis de condição (*sleeping on 'cond var'* no caso do output apresentado) ficando a esperar que haja uma alteração na variável de condição. Estas também podem ser forçadas a parar por *mutex* e a semáforos.

No output apresentado, também podemos verificar que por várias vezes as *threads* podem migrar de um CPU para outro (*from-CPU 19(1) to-cpu 0(1) CPU migration*).

Como o tipo de escalonamento é *static*, sempre que termina um segmento associado a essa *thread* logo a seguir é atribuído um novo segmento para execução, segmentos esses que têm sempre o mesmo tamanho como já foi referido anteriormente.

Podemos analisar que o aumento do tempo total para a conclusão da aplicação não está directamente associado a tempo "perdido" em *sleep* por, p.e. variáveis de condição ou desafectação forçada. É a própria biblioteca OpenMP que implica uma adição em termos de *overhead* em tempo de computação – neste traçado incluído nas linhas *restarted on the same CPU*, ou seja, CPU-Time. Não nos é possível distinguir o CPU-TIME despendido em porções de código da aplicação ou na biblioteca OpenMP.

## 6. Conclusão

Como podemos verificar, ao longo deste trabalho, tivemos desenvolver duas scripts que nos permitiu traçar/detectar o que nos foi pedido no enunciado. Ao longo do desenvolvimento apercebi-me cada vez mais das capacidades da *Framework Dtrace*, sendo que esta ferramenta nos permite, monitorizar ao pormenor tudo o que acontece num sistema. Para além desta grande vantagem, esta ferramenta também apresenta uma linguagem própria, a linguagem *D*, que como já foi referido anteriormente, é uma linguagem muito parecida (estruturalmente) com o *awk*.

Com esta linguagem, a *Framework* torna-se ainda muito mais "poderosa", uma vez que nos permite aproximar até ao máximo detalhe daquilo que pretendemos analisar no sistema.

Em relação ao trabalho extra, sugerido pelo professor no final do semestre, podemos verificar que com uma simples *script Dtrace* podemos fazer um traçado dinâmico do comportamento de todas as *threads* envolvidas na execução da aplicação paralela, para os diferentes tipos de escalonamento. Com esta *script* temos total controlo do percurso de cada *thread* ao longo da toda a execução. Uma vez mais, com este exemplo, podemos apercebermo-nos da potencia-

lidade e utilidade da ferramenta *DTrace*, quer para versões sequências quer para versões paralelas de código.

Como trabalho futuro, pretendo aperfeiçoar os meus conhecimentos em *Dtrace*, porque, como já disse, para além de ser uma ferramenta poderosa, no que toca a administração de sistemas, também é uma ferramenta muito útil e acessível uma vez que a sua sintaxe não é muito difícil nem a sua estrutura, o que se torna difícil no estudo desta ferramenta é a numerosa informação que se dispõem, bem como as numerosas provas que é permitido se criar com o *Dtrace*.

## 5 DTrace e IOzone

{Página em Branco (Fazer scroll)}

# Benchmarking dos Resultados Obtidos com *IOZone*, e a Framework *Dtrace*

Sérgio Caldas  
*Universidade do Minho*  
*Escola de Engenharia*  
*Departamento de Informática*  
*Email:* a57779@alunos.uminho.pt

## Conteúdo

<b>1</b>	<b>Introdução</b>	1
<b>2</b>	<b>Ferramentas Utilizadas</b>	1
<b>3</b>	<b><i>IOZone Benchmark</i></b>	2
<b>4</b>	<b>Testes/Flags a Realizar/Usar com <i>IOZone</i></b>	2
<b>5</b>	<b>Análise dos Resultados</b>	3
5.1	<i>Write</i> e <i>Rewrite</i> - Testado na <i>Home</i> .	3
5.2	<i>Write</i> e <i>Rewrite</i> - Testado na diretoria <i>diskHitachi</i> . . . . .	4
5.3	<i>Read</i> e <i>Reread</i> - Testado na <i>Home</i> .	4
5.4	<i>Read</i> e <i>Reread</i> - Testado na diretoria <i>diskHitachi</i> . . . . .	5
5.5	Tabela com todos os Resultados Obtidos	5
<b>6</b>	<b>Conclusão</b>	6
<b>Referências</b>		6

**Resumo**—Neste trabalho, desenvolvido no âmbito da disciplina de Engenharia de Sistemas de Computação (ESC), inserida no perfil de Computação Paralela e Distribuída (CPD) do curso de Engenharia Informática, tem como objetivo estudar uma variada gama de testes, constituintes do *IOZone Benchmark*. Este *Benchmark*, é usado para fazer testes de performance de *Filesystems*. Depois de definidos os testes que vou usar, posteriormente, tenho de confirmar esses resultados com a ferramenta *Dtrace*, criando diversas scripts que irão fazer traçados dinâmicos de forma a obter os mesmos resultados que foram obtidos na execução do *IOZone Benchmark*.

## 1. Introdução

*Benchmark* [2] é o ato de executar um conjunto de testes padrão de forma a avaliar o desempenho quer de *Hardware* quer de *Software*. O *Benchmark* é muito usado na área de ciências de computação, sendo que o *Benchmark* se divide em "Passivo" e "Activo" [1]. O *Benchmark* "Passivo", diz respeito à execução de um conjunto de testes padrão, sendo que estes são ignorados até que terminem. O

principal objetivo deste tipo de *Benchmark* é obter dados de referência de forma a poder comprovar/comparar com outros resultados. Com o *Benchmark* "Activo", testa-se a performance do *Benchmark* enquanto este corre, com isto podemos ver o que testa o *Benchmark* bem como perceber a maneira como o *Benchmark* se comporta, aqui os dados são informação.

Com a resolução deste trabalho, é suposto fazer *Benchmark* "Activo" ao *Filesystem* (no meu caso) da máquina *Solaris 11*, recorrendo a diversos testes do *IOZone Benchmark* e à ferramenta *Dtrace*.

Os testes que irei efectuar baseiam-se sobretudo na análise de operações, tais como, *Read*, *write*, *re-read*, *re-write*.

Depois de efetuados os testes em cima referidos, utilizarei a *Framework DTrace*, com a criação de diversas scripts em *D* capazes de calcular os mesmos resultados que foram obtidos com o *IOZone Benchmark*, por forma a poder comparar/comprovar os dois resultados.

## 2. Ferramentas Utilizadas

Na realização deste trabalho, para além do *IOZone Benchmark*, irão ser utilizadas mais duas ferramentas, a ferramenta *DTrace* e a ferramenta *truss* (ferramenta equivalente ao *strace* contudo é usada em *Solaris*).

O *DTrace* é uma *Framework* que permite fazer traçados dinâmicos, esta é usada para solucionar problemas no *Kernel* e aplicações em produção, em tempo real. O *Dtrace* pode ser utilizado para se obter uma visão geral da execução do sistema, como a quantidade de memória, o tempo de CPU, os recursos usados por os processos activos. Esta *Framework* permite fazer traçados muito mais rebuscados e detalhados, tais como, por exemplo a lista de processos que tenta aceder a um ficheiro.

No âmbito deste trabalho a *Framework DTrace*, é utilizada (através do uso *Scripts* em *D*) com o intuito de se obter os mesmos (ou aproximadamente os mesmos) resultados que se obteve com o *IOZone Benchmark*, por forma a confirmar/comparar esses resultados.

A ferramenta/comando *truss*, é uma ferramenta que executa um determinado comando e produz um traçado de todas as chamadas ao sistema que esse comando produz, os sinais que o comando recebe, e as falhas da máquina que ocorrem.

Cada linha do *output* retorna a falha ou o nome do sinal ou o nome da chamada ao sistema com os argumentos e valores de retorno.

No âmbito deste trabalho, a ferramenta *truss* é usada de forma a nos ajudar a compreender o comportamento de toda a aplicação que estamos a executar (neste caso o *IOZone Benchmark*), ver quais as chamadas ao sistema utilizadas pela a aplicação, bem como os sinais e falhas que esta produz. Basicamente esta ferramenta é usada apenas como forma de estudar o comportamento do *Benchmark*.

### 3. IOZone Benchmark

O *IOZone* é uma ferramenta de *Benchmark* para *Filesystem's*, com este *benchmark* é possível gerar e medir uma grande variedade de operações, esta ferramenta foi adaptada para diversos sistemas, correndo em diversos sistemas operativos.

O *IOZone* é util para determinar uma basta gama de análises sobre um *Filesystem* de um determinada plataforma de um determinado vendedor. Este *Benchmark* testa o desempenho de ficheiros I/O para as seguintes operações:

- read;
- write;
- re-read;
- re-write;
- read backwards;
- read strided;
- fread;
- fwrite;
- random read/write;
- pread/pwrite variants;
- aio\_read;
- aio\_write;
- mmap.

Com o uso deste *Benchmark*, obtemos uma grande gama de fatores de desempenho relativos ao *Filesystem*, com isto o cliente consegue ver os pontos fortes e fracos de uma plataforma e de um sistema operativo e tomar uma decisão mais equilibrada quanto a sua escolha.

### 4. Testes/Flags a Realizar/Usar com IOZone

Para correr o *IOZone Benchmark*, executa-se o seguinte comando, juntamente, com as *flags* que desejarmos.

```
/opt/csw/bin/iozone
```

Nesta secção, irei apresentar todos os testes que pretendo realizar com o *Benchmark IOZone* [3], bem como as *Flags* que pretendo usar na execução do *Benchmark*. Todos os resultados do *IOZone Benchmark* são em Kbytes/sec.

As *Flags* que irei usar são as seguintes:

- **-I** - Esta *flag* usa *DIRECT I/O* para todas as operações de ficheiros. Diz ao *Filesystem*, para todas

as operações ignorar o *buffer cache* e ir diretamente ao disco;

- **-a** - Esta *flag* é usada para ativar o modo automático, basicamente, esta flag ativa todos os testes disponíveis do *Benchmark*;
- **-r#** - Esta *flag* é usada para definir o tamanho do registo, podemos usar **-r#k** (tamanho em *Kbytes*), **-r#m** (tamanho em *Mbytes*), **-r#g** (tamanho em *Gbytes*);
- **-s#** - Esta *flag* é usada para definir o tamanho do ficheiro a testar, podemos usar **-s#k** (tamanho em *Kbytes*), **-s#m** (tamanho em *Mbytes*), **-s#g** (tamanho em *Gbytes*);
- **-t#** - Executa o *Benchmark* com *throughput mode*. Esta opção permite ao utilizador especificar quantas *Threads* ou Processos pretende ativar durante os testes;

No que toca aos testes, apresento em baixo todos os que pretendo realizar:

- **Write e Rewrite** - para este teste é usado a *flag* **-i0**. O comando completo pode ser consultado em baixo para um ficheiro de 10 Mb com um *Record Size* de 64 Kb:

```
/opt/csw/bin/iozone -i0 -r64k -s10m
```

- **Write** - mede a *performance* de escrever um ficheiro novo, quando um ficheiro é escrito, não é só os dados que são escritos mas também "meta-dados". Estes "meta-dados" contém informação que permite controlar onde os dados estão armazenados nas unidades físicas de armazenamento. Estes meta-dados contém informação da diretoria e do espaço alocado entre outros dados;
- **Rewrite** - Este teste mede a *performance* de escrever um ficheiro já existente. Quando um ficheiro existente é escrito, o trabalho necessário é menor uma vez que os "meta-dados" já existem. É normal a *performance* de rescrever um ficheiro já existente ser maior que a *performance* de escrever um ficheiro novo.

O output produzido pelo comando apresentado em cima foi:

```
Record Size 64 kB
File size set to 10240 kB
Command line used: /opt/csw/bin/iozone -i0 -r64k -s10m
Output is in KBytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 kBytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.

kB      reclen write   rewrite
10240     64    119047   119346
```

Como podemos verificar pelo *output* em cima, o *Record Size* é de 64 kB e o tamanho do ficheiro é de 10240 kB ou seja 10 Mb. Quanto aos resultados

obtidos vemos que as operações de escrita num novo ficheiro (*write*) foram feitas a uma velocidade de 119047 kBytes/sec e as operações de escrita num ficheiro já existente (*Rewrite*) foram feitas a uma velocidade de 119346 kBytes/sec.

- **Read e Reread** - para este teste é usado a flag **-il**. O comando completo pode ser consultado em baixo para um ficheiro de 10 Mb com um *Record Size* de 64 Kb:

```
/opt/csw/bin/iozone -il -r64k -s10m
```

- **Read** - Este teste mede a *performance* de leitura de um ficheiro já existente;
- **Reread** - Este teste mede a *performance* de leitura de um ficheiro que foi lido recentemente. É normal a *performance* ser maior uma vez que o sistema operativo mantém em cache os dados dos arquivos que foram lidos recentemente. Esta *cache* pode ser usada para satisfazer as leituras e melhorar a *performance*.

O output produzido pelo comando apresentado em cima foi:

```
Record Size 64 kB
File size set to 10240 kB
Command line used: /opt/csw/bin/iozone -il -r64k -s10m
Output is in kBytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 kBBytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.

kB    reclen     write   rewrite   read   reread
10240      64       118673   119714   2637083   3101102
```

Como podemos verificar pelo *output* em cima, o *Record Size* é de 64 kB e o tamanho do ficheiro é de 10240 kB ou seja 10 Mb. Quanto aos resultados obtidos vemos que as operações de leitura de um ficheiro já existente (*read*) foram feitas a uma velocidade de 2637083 kBytes/sec e as operações de escrita de um ficheiro lido recentemente (*Reread*) foram feitas a uma velocidade de 3101102 kBytes/sec. Como podemos verificar a velocidade das operações *Reread* é mais elevada, o que já era de se esperar, visto que o sistema operativo mantém em *cache* os dados de arquivos lidos recentemente o que melhora a performance.

## 5. Análise dos Resultados

Nesta secção irei apresentar os resultados que obtive com a execução do *Benchmark IOZone* para diferentes testes, bem como o *output* produzido pelas *scripts* que desenvolvi em *D*. Os testes foram realizados por etapas, a primeira etapa testei as *Syscall's write e Rewrite* e na segunda etapa testei *Syscall's read e reread*.

### 5.1. Write e Rewrite - Testado na Home

Para testar a *Syscall write (rewrite)* na *home*, usa um sistema de ficheiros *ZFS*, executei o seguinte comando:

```
/opt/csw/bin/iozone -i0 -s10m
```

sendo que o seu *output* foi o seguinte:

```
1  File size set to 10240 kB
2  Command line used: /opt/csw/bin/iozone -i0 -s10m
3  Output is in kBytes/sec
4  Time Resolution = 0.000001 seconds.
5  Processor cache size set to 1024 kB.
6  Processor cache line size set to 32 bytes.
7  File stride size set to 17 * record size.

8      kB    reclen     write   rewrite
9      10240      4       179488   391934
10
```

A *script* em *D* por mim desenvolvida e que posteriormente foi executada com *Framework Dtrace* pode ser consultada em baixo:

```
1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4
5  syscall::open*:entry
6  /execname=="iozone" & uid==29231/
7  {
8     self->path = copyinstr(arg1);
9
10
11    syscall::open*:return
12    /self->path=="iozone.tmp"/
13    {
14        total_time=0;
15        size = 0;
16        flag = 1;
17    }
18
19    syscall::write:entry
20    /flag == 1/
21    {
22        self->start_time = timestamp;
23        self->w_size = (arg2/1024);
24        size = size + self->w_size;
25    }
26
27    syscall::write:return
28    /self->start_time > 0 & self->w_size > 0/
29    {
30        self->stop_time = timestamp;
31        self->elapsed = self->stop_time - self->start_time;
32        total_time = total_time + self->elapsed;
33    }
34
35
36    syscall::close*:entry
37    /self->path=="iozone.tmp"/
38    {
39        printf("%-12s %s\n", "SIZE", "ELAPSED TIME");
40        printf("%-12d %d\n",size,total_time);
41        flag = 0;
42    }
43
```

Depois de executada a *script* em cima apresentada, esta apresentou o seguinte *output*:

```
1  SIZE      ELAPSED TIME
2  0          0
3  SIZE      ELAPSED TIME
4  0          0
5  SIZE      ELAPSED TIME
6  10240    50946670
7  SIZE      ELAPSED TIME
8  10240    50946670
9  SIZE      ELAPSED TIME
10 10240   22982712
```

Ao analisarmos estes resultados podemos constatar que os valores se encontram num intervalo aceitável relativamente aos resultados do *Benchmark IOZone*. Ora vejamos, o tamanho (*SIZE*) encontra-se já em *Kbytes* e o tempo (*ELAPSED TIME*) encontra-se em nanosegundos,

por isso necessito converter para segundos, para isso fiz  $50946670 * 10^{-9} = 0.05094667$ , por fim para se obter o resultado pretendido ( $Kb/sec$ ) fiz:  $10240/0.05094667 = 200994.49091 Kb/sec$ . Como podemos verificar este valor encontra-se entre um intervalo aceitável ( $179488 \leq 200994.49091 \leq 391934$ ).

Com isto podemos concluir que o *Benchmark IOZone* inicia a contagem do tempo sempre que executa a *syscall open* e termina sempre que executa a *syscall close*. A contagem dos *Bytes* escritos é feita sempre à entrada da *syscall write*.

De referir que este valor obtido através da *script* em *D* contém um erro relativo ao tamanho, uma vez que a medida que se aumenta o tamanho dos ficheiros, os valores resultantes apresentam valores com uma margem de erro maior.

## 5.2. Write e Rewrite - Testado na diretoria *diskHitachi*

Para testar o *Benchmark IOZone* no disco HDD (*diskHitachi*) tive de correr o seguinte comando, de referir que tive de usar a *flag -f* para redirecionar o ficheiro temporário para o disco referido.

```
/opt/csw/bin/iozone -f /diskHitachi/a57779/iozone.tmp -i0 -s10m
```

o qual teve o seguinte *output*:

```
File size set to 10240 kB
Command line used: /opt/csw/bin/iozone -f /diskHitachi/a57779/iozone.←
tmp -i0 -s10m
Output is in kBytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 kBytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.

kB reclen write rewrite
10240 4 126864 340835
```

Em simultâneo foi executada a seguinte *Script* com o dtrace, de notar também que no predicado em que verifico o *path* (*self->path*) também tive de por a diretoria do disco em causa.

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

syscall::open*:entry
/execname=="iozone" & uid==29231/
{
    self->path = copyinstr(arg1);
}

syscall::open*:return
/self->path=="/diskHitachi/a57779/iozone.tmp"/
{
    total_time=0;
    size = 0;
    flag = 1;
}

syscall::write:entry
/flag == 1/
{
    self->start_time = timestamp;
    self->w_size = (arg2/1024);
    size = size + self->w_size;
}

syscall::write:return
/self->start_time > 0 & self->w_size > 0/
{
    self->stop_time = timestamp;
```

```
        self->elapsed = self->stop_time - self->start_time;
        total_time = total_time + self->elapsed;
    }

syscall::close*:entry
/self->path=="/diskHitachi/a57779/iozone.tmp"/
{
    printf("%-12s %s\n", "SIZE", "ELAPSED TIME");
    printf("%-12d %d\n", size, total_time);
    flag = 0;
}
```

O *Output* da execução da *script* com a *Framework Dtrace* foi o seguinte:

1	SIZE	ELAPSED TIME
2	0	0
3	SIZE	ELAPSED TIME
4	0	0
5	SIZE	ELAPSED TIME
6	10240	74204464
7	SIZE	ELAPSED TIME
8	10240	74204464
9	SIZE	ELAPSED TIME
10	10240	26921551

Ao analisarmos os dados da mesma maneira que analisei em cima, isto é fazendo os mesmos cálculos que efectuei em cima, obtive os seguintes resultados:

- Tempo:  $74204464 * 10^{-9} = 0.074204464sec$
- Kb/sec:  $10240/0.047833799 = 137997.08869Kb/sec$

Como podemos verificar o valor resultante é um valor que se encontra dentro de um intervalo aceitável, comparado com os resultados do *Benchmark IOZone*, isto é  $126864 \leq 137997.08869 \leq 340835$ .

Os discos SSD em termos de escrita têm um desempenho idêntico a um disco HDD, dai as operações de escrita do disco HDD (*diskHitachi*) serem semelhantes às do disco SSD (teste *Home*).

## 5.3. Read e Reread - Testado na *Home*

Para testar a *Syscall read (reread)*, executei o seguinte comando:

```
/opt/csw/bin/iozone -i0 -il -s10m
```

sendo que o seu output foi o seguinte:

```
1
2
3 File size set to 10240 kB
4 Command line used: /opt/csw/bin/iozone -i0 -il -s10m
5 Output is in kBytes/sec
6 Time Resolution = 0.000001 seconds.
7 Processor cache size set to 1024 kBytes.
8 Processor cache line size set to 32 bytes.
9 File stride size set to 17 * record size.

10
11 kB reclen write rewrite read reread
12 10240 4 428200 347270 1223574 1247709
13
14
15
16
17
18
19
20
21
22
23 #!/usr/sbin/dtrace -s
24
25 #pragma D option quiet
26
27 syscall::open*:entry
28 /execname=="iozone" & uid==29231/
29 {
    self->path = copyinstr(arg1);
```

Quanto a *script* em *D* por mim desenvolvida, para o traçado da *syscall read*, que posteriormente foi executada com *Framework Dtrace* pode ser consultada em baixo:

```

    self->flag = arg2;
}

syscall::open*:return
/self->path=="iozone.tmp" /
{
    self->start_time = timestamp;
    total_time=0;
    size = 0;
    flag = 1;
    total_time_io = 0;
}

syscall::read:return
/self->start_time> 0/
{
    self->r_size = (arg0/1024);
    size = size + self->r_size;
}

syscall::close*:entry
/self->path=="iozone.tmp" /
{
    self->stop_time = timestamp;
    self->lapsed = self->stop_time - self->start_time;
    total_time = total_time + self->lapsed;
    printf("%-12s %s\n", "SIZE", "ELAPSED TIME");
    printf("%-12d %d\n", size, total_time);
    flag = 0;
}

```

```

9      Command line used: /opt/csw/bin/iozone -f /diskHitachi/a57779/iozone.←
10     tmp -i0 -i1 -s10m
11     Output is in kB/sec
12     Time Resolution = 0.000001 seconds.
13     Processor cache size set to 1024 kBbytes.
14     Processor cache line size set to 32 bytes.
15     File stride size set to 17 * record size.
16
17          kB  recien   write  rewrite   read   reread
18          10240      4  117760  603490  1356985  1521537
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
826
827
828
829
829
830
831
832
833
834
835
836
836
837
838
839
839
840
841
842
843
844
845
845
846
847
848
848
849
849
850
851
852
853
854
855
856
856
857
858
859
859
860
861
862
863
864
865
865
866
867
868
868
869
869
870
871
872
873
874
875
875
876
877
878
878
879
879
880
881
882
883
884
885
885
886
887
887
888
889
889
890
891
892
893
893
894
895
895
896
897
897
898
898
899
899
900
901
902
903
903
904
905
905
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1
```

## 5.5. Tabela com todos os Resultados Obtidos

Apresento na tabela 1 todos os resultados obtidos para todos os testes realizados neste trabalho, quer para o *Benchmark IOZone* quer para a *Ferramenta Dtrace*.

		Write	Rewrite	Read	Reread
SSD (Teste Home (ZFS))	IOZone	179488 Kb/sec	391934 Kb/sec	1223574 Kb/sec	1247709 Kb/sec
	Dtrace	200994.49091 Kb/sec		1219830.5098 Kb/sec	
HDD (Teste /diskHitachi (UFS))	IOZone	126864 Kb/sec	340835 Kb/sec	1356985 Kb/sec	1521537 Kb/sec
	Dtrace	137997.08869 Kb/sec		1318182.0463 Kb/sec	

Tabela 1. RESULTADOS OBTIDOS PARA IOZONE, DTRACE PARA DISCOS SSD E HDD PARA AS CHAMADAS AO SISTEMA WRITE (REWRITE), READ (REREAD)

Ao analisar a tabela 1 posso concluir que os resultados obtidos para um disco SSD (teste *Home*), para um sistema de ficheiros ZFS, para operações de escrita (*write (rewrite)*) são resultados, ligeiramente mais altos do que aqueles que foram obtidos para um disco HDD (teste */diskHitachi*), com um sistema de ficheiros UFS.

Em termos de operações de leitura (*read (reread)*) tanto para o disco SSD como para o disco HDD, os resultados são semelhantes, sendo que para o disco HDD os resultados são um pouco mais altos, resultados estes que vão um pouco contra o que esperava, uma vez que os discos SSD tem um melhor desempenho do que os discos HDD, como tal tenho algumas dificuldades em explicar este facto, uma vez que isto acontece tanto na execução do *Benchmark IOZone* como com as *scripts Dtrace*. De referir que na operação de re-leitura (*Rewrite*), os resultados apresentados apresentam um melhor desempenho relativamente à operação de leitura, isto deve-se ao facto de o Sistema Operativos manter em cache dados de ficheiros que foram lidos recentemente, como tal o acesso a esses dados é mais rápido.

Nas operações de re-escrita, acontece exatamente o mesmo, isto é, esta apresenta um melhor desempenho comparativamente à operação de escrita, isto porque, a primeira vez que se escreve num ficheiro, para além dos dados também são escritos meta-dados, o que não acontece na operação de re-escrita, uma vez que não é necessário escrever os meta-dados de novo.

## 6. Conclusão

Numa primeira abordagem a este trabalho, decidi explorar bem o *IOZone Benchmark*, realizando vários testes, com várias *flags*, por forma a perceber bem o funcionamento deste *Benchmark*. Para além da realização destes testes, ainda usei a ferramenta *truss* de forma a perceber qual o comportamento desta aplicação, isto é, que chamadas ao sistema utiliza, quantas vezes são feitas, quantas operações de escrita/leitura são feitas, entre outras análises.

Nesta fase do trabalho a maior dificuldade com que me deparei, foi essencialmente com a análise dos resultados resultantes do comando *truss*, uma vez que, quando executava o comando *truss*, para o *IOZone*, para um determinado tamanho de ficheiro, com um determinado *Record Size*, estava à espera de um resultado (no meu caso, um determinado numero de operações de escrita) e na verdade o resultado

era um que não estava à espera, situação que para já não consigo perceber bem.

Na segunda parte deste trabalho desenvolvi *scripts* em *D*, que depois de usadas com a *Framework Dtrace*, produziram resultados semelhantes aos resultados obtidos com o *Benchmark IOZone*. Depois de obtidos esses resultados

comparei os dois e pude concluir que, estes são semelhantes.

Nesta fase a maior dificuldade com que me deparei foi em perceber como iria fazer as provas na *script D* de forma a obter os resultados que desejava, para ultrapassar essa dificuldade tive de analisar melhor o *output* do comando *truss*, por forma a perceber ainda melhor o comportamento do *Benchmark*.

Como trabalho futuro, gostava de melhorar as *scripts* que desenvolvi de maneira a conseguir obter resultados mais precisos. Também gostava de desenvolver algumas *scripts*, que na minha opinião, também seriam relevantes para este trabalho.

## Referências

- [1] Active benchmarking. <http://www.brendangregg.com/activebenchmarking.html>.
- [2] Benchmark (computação). <https://pt.wikipedia.org/wiki/Benchmark>.
- [3] W. D. Norcott. Iozone filesystem benchmark.

## 6 Tutorial Perf

{Página em Branco (Fazer scroll)}

# ***Profiling de Software/Hardware com Perf***

Sérgio Caldas  
*Universidade do Minho*  
*Escola de Engenharia*  
*Departamento de Informática*  
*Email: a57779@alunos.uminho.pt*

## **Conteúdo**

<b>1</b>	<b>Introdução</b>	1
<b>2</b>	<b>Perf</b>	1
<b>3</b>	<b>Caracterização do Ambiente de Testes</b>	2
<b>4</b>	<b>Parte 1 - Procura dos pontos quentes de uma aplicação em execução</b>	2
4.1	Tempo de Execução e Contagem de Eventos . . . . .	3
4.2	Procura de <i>Hotspots</i> . . . . .	3
4.3	Profiling com o <i>Perf</i> . . . . .	5
<b>5</b>	<b>Parte 2 - Contagem de eventos de Hardware</b>	5
5.1	Código Fonte do Programa . . . . .	5
5.2	Análise do Desempenho . . . . .	6
5.3	Rácio e Taxas . . . . .	6
<b>6</b>	<b>Parte 3 - Perfis de Eventos de Hardware</b>	7
6.1	Modo Contagem: <i>naive_large</i> vs <i>interchange_large</i> . . . . .	7
6.2	Visualização dos Perfis Baseados em Eventos . . . . .	8
<b>7</b>	<b>Modo Amostragem: <i>naive_large</i> vs <i>interchange_large</i></b>	10
<b>8</b>	<b>Flame Graphs</b>	10
8.1	Geração de <i>Flame Graphs</i> . . . . .	10
8.2	Flame Graph das 4 Aplicações Usadas neste Trabalho . . . . .	11
<b>9</b>	<b>Conclusão</b>	11
<b>Referências</b>		12

**Resumo**—Este artigo, representa o relatório do trabalho prático nº5, desenvolvido no âmbito da disciplina de Engenharia de Sistemas de Computação (ESC), inserida no perfil de Computação Paralela e Distribuída (CPD) do curso de Engenharia Informática. O objetivo deste trabalho é seguiremos um tutorial [2] providenciado pelo professor, com o intuito de iniciarmos e praticarmos a utilização da ferramenta *Perf*.

Para a execução deste trabalho, para além do tutorial, também nos foi facultado um código (*naive.c*), este código efectua a multiplicação de matrizes. Com este código e juntamente com a ferramenta *Perf*, procedi ao *profiling* do mesmo, utilizando diferentes comandos desta ferramenta. Com estes comandos é possível fazermos *profiling*, quer com contadores de *software*, quer com contadores de *hardware*

## **1. Introdução**

Este trabalho está dividido em 3 partes, assim como o tutorial providenciado, a primeira parte diz respeito à deteção de *hot spots* da execução da aplicação produzida pelo código facultado, esta primeira parte, cobre os comandos e opções básicas da ferramenta *Perf* assim como os seus eventos de desempenho de software mais básicos.

A segunda parte introduz os eventos de desempenho de *hardware*, sendo que o tutorial faz uma demonstração de como realizar medições dos eventos de *hardware* em torno de toda a aplicação. Nesta segunda parte para além do código referido atrás é também utilizada uma versão optimizada desse mesmo código (*interchange.c*).

Na terceira e ultima parte deste tutorial, utilizo amostras de eventos de desempenho de *hardware* para identificar e analisar "*hotspots*" nos programas que são testados. Nesta parte são testados dois programas o *naive\_large*, que corresponde ao programa *naive* referido atrás mas com uma maior dimensão do tamanho das matrizes e o programa *interchange\_large* que corresponde a uma versão com uma maior dimensão do tamanho das matrizes para o programa optimizado referido atrás.

Por fim, neste trabalho, também procedemos a geração de *FlameGraphs*, para cada uma das aplicações. Estes gráficos foram obtidos através dos dados recolhidos com o *perf* e tratados com *scripts* específicas, posteriormente.

Neste relatório, apresento todos os meus resultados obtidos na realização e no acompanhamento do tutorial referido, bem como a análise desses mesmos resultados.

## **2. Perf**

O *Perf* é uma ferramenta de análise de *performance* desenvolvida para *LINUX*, esta ferramenta é acessível através

da linha de comandos e fornece uma gama de sub-comandos bem como uma basta gama de contadores, tanto de *hardware* como de *software*. Estes comandos permitem fazer uma análise estatística de todo o sistema, quer ao nível do *Kernel* quer ao nível do utilizador.

Esta ferramenta para além dos referidos contadores, providênciia também *Tracepoints* e provas dinâmicas como por exemplo *kprobes* ou *uprobes*.

O *Perf* apresenta um conjunto de comandos principais (os mais utilizados), esses comandos encontam-se listados e explicados em baixo:

- *perf stat* - este comando permite fazer uma recolha estatística dos principais eventos do *Perf*, se quisermos selecionar apenas um sub-conjunto desses principais eventos apenas temos de adicionar ao comando a *flag* -e juntamente com o nome dos eventos desejados, como por exemplo *perf stat -e cpu-clock*. O comando referido é um comando mais leve em relação aos restantes comandos do *Perf*.
- *perf record* - este comando faz uma captura/gravação dos dados dos contadores especificados no ficheiro *perf.data*, para posteriormente serem tratados pelo comando *perf report*. O comando *perf record* à semelhança do comando do ponto anterior também permite seleccionarmos quais os eventos que queremos recolher informação, para isso só temos de adicionar ao comando a *flag* -e juntamente com o nome dos eventos desejados, como por exemplo *perf record -e cpu-clock,faults*.
- *perf report* - com este comando é possível consultarmos e analisarmos os dados guardados no ficheiro *perf.data*. Este comando à semelhança dos outros também permite uma gama de opções/*flags*. Para a seleção da *interface do utilizador* podemos usar os seguintes opções:
  - -tui, esta opção permite seleccionar uma *interface* baseada na linha de comandos. Esta opção suporta uma navegação *interativa*.
  - -stdio, esta opção imprime o *output* do *profile* capturado no *standard output*.
  - -gtk esta opção seleciona a *GTK interface*.

Para além dos comandos atrás referidos, o *Perf* ainda contem um conjunto de contadores pré-definidos tais como:

- *cpu-clock*
- *task-clock*
- *page-faults OR faults*
- *context-switches OR cs*
- *cpu-migrations OR migrations*
- *minor-faults*
- *major-faults*
- *alignment-faults*
- *emulation-faults*

A versão do *Perf* utilizada na realização deste trabalho é a versão 4.0.0.

### 3. Caracterização do Ambiente de Testes

A máquina utilizada para se realizar este trabalho, foi um nó do *Cluster search* mais especificamente o nó 431. Na tabela 1 encontra-se a especificação desse mesmo nó.

System	Máquina 431
# CPUs	2
CPU	Intel® Xeon® X5650
Architecture	Nehalem
# Cores per CPU	6
# Threads per CPU	12
Clock Freq.	2.66 GHz
L1 Cache	192 KB 32 KB por core
L2 Cache	1536 KB 256 KB por core
L3 Cache	12 MB
Inst. Set Ext.	SSE4.2 e AVX
#Memory Channels	3
Memory BW	32 GB/s

Tabela 1. CARACTERIZAÇÃO DA MÁQUINA 431

Para a compilação dos programas referidos anteriormente foi carregado neste nó o modulo com o gnu/4.9.0, para além desta versão todos os programas foram compilados com a *flag* -O2 -ggdb -g -c.

### 4. Parte 1 - Procura dos pontos quentes de uma aplicação em execução

A primeira parte deste trabalho, foca-se essencialmente na procura de pontos quentes de uma aplicação em execução, para isso iniciei o tutorial exatamente pela sua primeira parte.

Inicialmente comecei por testar alguns comandos básicos do *Perf*, comandos como,

```
perf --help
```

este comando apresenta uma lista com todos os comandos todos os comandos mais utilizados do *Perf* como pode ser consultado em baixo.

The most commonly used perf commands are:	1
annotate      Read perf.data (created by perf record) and display ↔	2
annotated code	3
archive      Create archive with object files with build-ids found in ↔	4
perf.data file	5
bench        General framework for benchmark suites	6
buildid-cache    Manage build-id cache.	7
buildid-list    List the buildids in a perf.data file	8
diff        Read perf.data files and display the differential profile	9
evlist      List the event names in a perf.data file	10
inject      Filter to augment the events stream with additional ↔	11
information	12
kmem       Tool to trace/measure kernel memory(slab) properties	13
kvm        Tool to trace/measure kvm guest os	14
list       List all symbolic event types	15
lock       Analyze lock events	16
mem        Profile memory accesses	17
record     Run a command and record its profile into perf.data	18
report     Read perf.data (created by perf record) and display the ↔	19
profile	20
sched      Tool to trace/measure scheduler properties (latencies)	21
script     Read perf.data (created by perf record) and display trace ↔	22
output	23
stat       Run a command and gather performance counter statistics	24
test       Runs sanity tests.	
timechart   Tool to visualize total system behavior during a workload	
top        System profiling tool.	
trace      strace inspired tool	
probe     Define new dynamic tracepoints	

Se quisermos obter mais informação relativamente aos comandos apresentados em cima basta-nos executar o seguinte comando:

```
perf help COMMAND
```

ou

```
perf COMMAND --help
```

Como foi referido anteriormente, o *Perf* suporta, quer eventos de *Software* quer eventos de *Hardware*. Para termos acesso à lista de eventos disponíveis na máquina, só temos de executar o comando em baixo exemplificado.

```
perf list
```

Ao executarmos esse comando é-nos apresentado uma lista com todos os eventos disponíveis na máquina. Como podemos ver na lista em baixo.

```
cpu-cycles OR cycles
instructions
cache-references
cache-misses
branch-instructions OR branches
branch-misses
stalled-cycles-frontend OR idle-cycles-frontend
stalled-cycles-backend OR idle-cycles-backend

cpu-clock
task-clock
page-faults OR faults
context-switches OR cs
cpu-migrations OR migrations
minor-faults
major-faults
alignment-faults
emulation-faults

L1-dcache-loads
L1-dcache-load-misses
L1-dcache-stores
L1-dcache-store-misses
L1-dcache-prefetches
L1-icache-loads
L1-icache-load-misses
LLC-loads
LLC-load-misses
LLC-stores
LLC-store-misses
LLC-prefetches
LLC-prefetch-misses
dTLB-loads
dTLB-load-misses
dTLB-stores
dTLB-store-misses
iTLB-loads
iTLB-load-misses
branch-loads
branch-load-misses
```

[Hardware event]  
[Software event]

## 4.1. Tempo de Execução e Contagem de Eventos

Depois de uma familiarização mais básica com a ferramenta *Perf*, passei então à análise da aplicação em questão. De referir mais uma vez que o código utilizado nesta primeira parte do tutorial foi o código *naive.c*.

Para se fazer uma boa análise da aplicação em execução, temos de ter uma referência, isto é, uma base por onde nos podemos guiar. Então o primeiro paço deste tutorial, é fazer uma recolha estatística da aplicação em execução, para isso o evento *cpu-clock* é o evento que precisamos, este evento dá-nos o número de *cpu-clock* em milissegundos, bem como o tempo gasto na execução. O comando utilizado para fazer esta recolha estatística foi o seguinte:

```
perf stat -e cpu-clock ./naive
```

sendo que o seu *output* foi o seguinte:

```
1 Performance counter stats for ./naive :
2           185.696104      cpu-clock (msec)
3           0.187093191   seconds time elapsed
4
5
```

Para além do evento *cpu-clock* podemos medir mais do que um evento, para isso só temos de executar o comando com a flag *-e* juntamente com o sub-conjunto de eventos desejados. Como mostra o exemplo seguinte:

```
perf stat -e cpu-clock,faults ./naive
```

O *output* do comando em cima exemplificado foi:

```
1 Performance counter stats for ./naive :
2           182.586308      cpu-clock (msec)
3             845          faults
4
5           0.184159812   seconds time elapsed
6
```

As regiões do código que gastam a maior parte do tempo de execução, são chamadas de *hotspots*, estas regiões são as melhores regiões para se fazer alterações no código, de forma a optimizá-lo, pois com um pequeno esforço podemos ter grandes ganhos.

Depois de medirmos os *cpu-clocks* e as *page-faults*, perguntámos-nos se estas eventos indicam um problema de desempenho ou não?! O número 845 de *page-faults* é demasiado?! Para respondermos a estas questões é necessário termos um conhecimento aprofundado da estrutura do código bem como das suas estruturas de dados. Pelo menos temos de saber se a aplicação é *Memory Bound* ou *CPU Bound*, isto é se perde demasiado tempo nos acessos à memória ou se efectua demasiado trabalho computacional respetivamente.

Com o *Perf* podemos fazer um profiling mais detalhado sobre a aplicação de forma a identificarmos esses problemas, para posteriormente serem optimizados e por fim efectuar novas medições de forma a serem comparadas com as medições que foram feitas no inicio deste tutorial. Só depois dessa comparação e dessa nova análise é que saberemos se tivemos algum *speed-up* ou não.

## 4.2. Procura de Hotspots

Com a ajuda do *Perf* é possível localizarmos *hotspots* para isso temos de fazer um profiling da aplicação em questão, para isso executamos o comando *perf record* que faz uma recolha de dados de perfil e guarda no ficheiro *perf.data*. O comando executado para se fazer esta recolha foi o seguinte comando:

```
perf record -e cpu-clock,faults ./naive
```

Neste caso específico o *Perf* faz uma recolha de dados de perfil para dois eventos: *cpu-clock* e *page-faults*.

Depois destes dados serem recolhidos, estes são tratados com o comando *perf report* este comando mostra-nos toda a informação guardada no ficheiro *perf.data*. O comando utilizado para a análise dos dados contidos no ficheiro *perf.data* foi o seguinte:

```
perf report --stdio --sort comm,dso
```

sendo que o seu *output* foi:

```
# =====
# captured on: Tue May 24 00:12:26 2016
# hostname : compute_431-9.local
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 4.0.0
# arch : x86_64
# nrcpus online : 24
# nrcpus avail : 24
# cpudesc : Intel(R) Xeon(R) CPU E5649 @ 2.53GHz
# cpuid : GenuineIntel,6,44,2
# total memory : 49551752 kB
# cmdline : /share/jade/SOFT/perf/perf record -e cpu-clock,faults ./naive
# event : name = cpu-clock, type = 1, config = 0x0, config1 = 0x0, config2 = 0x0, exl_usr = 0
# event : name = faults, type = 1, config = 0x2, config1 = 0x0, config2 = 0x0, exl_usr = 0,
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: cpu = 4, tracepoint = 2, software = 1
# =====
#
# Samples: 738 of event cpu-clock
# Event count (approx.): 738
#
# Overhead Command Shared Object Symbol
# ..... .
#
95.66% naive naive [...] multiply_matrices
1.49% naive naive [...] initialize_matrices
0.14% naive naive [...] rand@plt
#
# Samples: 17 of event faults
# Event count (approx.): 1245
#
# Overhead Command Shared Object Symbol
# ..... .
#
65.62% naive naive [...] initialize_matrices
```

```
# total memory : 49551752 kB
# cmdline : /share/jade/SOFT/perf/perf record -e cpu-clock,faults ./naive
# event : name = cpu-clock, type = 1, config = 0x0, config1 = 0x0, config2 = 0x0, exl_usr = 0
# event : name = faults, type = 1, config = 0x2, config1 = 0x0, config2 = 0x0, exl_usr = 0,
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: cpu = 4, tracepoint = 2, software = 1
# =====
#
# Samples: 738 of event cpu-clock
# Event count (approx.): 738
#
# Overhead Command Shared Object Symbol
# ..... .
#
95.66% naive naive [...] multiply_matrices
1.49% naive naive [...] initialize_matrices
0.14% naive naive [...] rand@plt
#
# Samples: 17 of event faults
# Event count (approx.): 1245
#
# Overhead Command Shared Object Symbol
# ..... .
#
65.62% naive naive [...] initialize_matrices
```

Ao analisarmos o *output* em cima, já podemos ter uma ideia mais pormenorizada das funções onde é gasto a maior parte do tempo de execução e onde há um maior numero de *page-faults* na aplicação. Como podemos ver, 95.66% do tempo de execução da aplicação *naive* é gasto na função *multiply\_matrices*, e a função *initialize\_matrices* é a função que apresenta um maior numero de *page-faults* na aplicação. Com estes dados já estamos mais perto do *hotspot* que procuramos, contudo ainda é possível aprofundarmos mais esta nossa pesquisa, para isso usamos o comando *perf annotate* com a flag *-dsos* e a flag *-symbol*. Para termos ainda uma analise mais detalhada, de forma a encontrarmos o *hotspot*, executarmos o seguinte comando:

```
perf annotate --stdio --dsos=naive --symbol=multiply_matrices
```

sendo que o seu *output* é o seguinte:

Percent	Source code & Disassembly of naive for cpu-clock
:	:
:	Disassembly of section .text:
:	000000000400810 <multiply_matrices>:
:	multiply_matrices():
:	}
:	}
:	void multiply_matrices()
:	{
0.00 :	400810: pxor %xmm2,%xmm2
0.00 :	400814: mov \$0x7e97c0,%edi
0.00 :	400819: mov %rdi,%r8
0.00 :	40081c: xor %esi,%esi
0.00 :	40081e: sub \$0x7e97c0,%r8
0.00 :	400825: nopl (%rax)
0.00 :	400828: lea 0x6f5580(%rsi),%rax
0.00 :	40082f: lea 0x7e97c0(%rsi),%rcx
0.00 :	400836: mov %rdi,%rdx
0.00 :	400839: movaps %xmm2,%xmm1
0.00 :	40083c: nopl 0x(%rax)
:	:
24.65 :	for (i = 0 ; i < MSIZE ; i++) {
1 0.28 :	for (j = 0 ; j < MSIZE ; j++) {
2 0.00 :	float sum = 0.0 ;
3 20.11 :	for (k = 0 ; k < MSIZE ; k++) {
4 :	sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
5 :	400840: movss (%rdx),%xmm0
6 :	400844: add \$0x7d0,%rax
7 :	40084a: mulss -0x7d0(%rax),%xmm0
8 :	400852: add \$0x4,%rdx
9 :	int i, j, k ;
10 22.38 :	for (i = 0 ; i < MSIZE ; i++) {
11 0.28 :	for (j = 0 ; j < MSIZE ; j++) {
12 0.00 :	float sum = 0.0 ;
13 20.11 :	for (k = 0 ; k < MSIZE ; k++) {
14 0.00 :	400856: cmp %rcx,%rax

Como podemos ver, pela a análise dos dados em cima apresentados, podemos verificar que 97.29% do tempo de execução é atribuída à aplicação *naive*. Quanto ao número de *page-faults* podemos verificar que 65.62% é atribuída também á execução da aplicação *naive*. Com esta informação sabemos que é a aplicação que está a ter um maior tempo de execução bem como um maior numero de *page-faults*, contudo precisamos de ser mais minuciosos no nosso *profiling*. Para isso executarmos o comadno *perf report* com a flag *-dsos*. Com esta flag restringimos o *output* ao objecto partilhado dinamicamente neste caso o programa *naive*.

Como tal executamos o seguinte comando:

```
perf report --stdio --dsos=naive,libc-2.13.so
```

sendo que o seu *output* é:

```
# =====
# captured on: Tue May 24 00:12:26 2016
# hostname : compute-431-9.local
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 4.0.0
# arch : x86_64
# nrcpus online : 24
# nrcpus avail : 24
# cpudesc : Intel(R) Xeon(R) CPU E5649 @ 2.53GHz
# cpuid : GenuineIntel,6,44,2
```

```

0.00 :           sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
400859:    addss  %xmm0,%xmm1
int i, j, k;

for (i = 0 ; i < MSIZE ; i++) {
    for (j = 0 ; j < MSIZE ; j++) {
        float sum = 0.0 ;
        for (k = 0 ; k < MSIZE ; k++) {
            32.44:   jne    400840 <multiply_matrices+0x30>
            sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
        }
        matrix_r[i][j] = sum ;
    }
}
0.00 :    movss  %xmm1,0x601340(%r8,%rsi,1)
400869:    add    $0x4,%rsi
void multiply_matrices()
{
    int i, j, k;

    for (i = 0 ; i < MSIZE ; i++) {
        for (j = 0 ; j < MSIZE ; j++) {
            0.00:   cmp    $0x7d0,%rsi
            0.00:   jne    400828 <multiply_matrices+0x18>
            0.00:   add    $0x7d0,%rdi
            400876: repz  retq
        }
    }
}
void multiply_matrices()
{
    int i, j, k;

    for (i = 0 ; i < MSIZE ; i++) {
        0.00:   cmp    $0x8dd0,0,%rdi
        0.00:   jne    400819 <multiply_matrices+0x9>
        0.00:   repz  retq
    }
}

```

Ao analisarmos o *output* apresentado vemos qual a instrução que gasta um maior tempo de execução, com cerca de 32.44%. A instrução em causa é a seguinte instrução:

```
32.44 : 40085d: jne 400840 <multiply_matrices+0x30>
```

esta instrução corresponde ao seguinte excerto de código em C:

```
sum = sum + (matrix_a[i][k] * matrix_b[k][j]);
```

Ou seja a maior parte do tempo gasto na execução da aplicação é gasta no cálculo da multiplicação das duas matrizes e na adição do seu resultado à variável *sum*. Com isto encontramos o *hotspot*, sendo que o a parte mais *hottest* da função *multiply\_matrices* é o ciclo mais interno. Este ciclo corresponde ao excerto de código em C apresentado em cima.

### 4.3. Profiling com o Perf

O *Perf*, usa amostras estatísticas para recolher informação. O evento *cpu-clock*, usa o tempo de relógio do *LINUX* fazendo uma recolha das amostras em intervalos de tempo pré-definido. Quando esse tempo passa o *Perf* provoca uma interrupção, e determina o que o *CPU* está a fazer nesse momento da interrupção e recolhe a informação desejada.

Como o *Perf*, usa amostras estatísticas é necessário recolher um número de amostras, que sejam capazes de exprimir corretamente o que se passa na máquina no momento da recolha. O *Perf* usa um intervalo de tempo pré-definido para fazer a recolha de informação, contudo é possível mudar esse intervalo de forma a se recolher um numero maior ou menor de amostras. O numero de amostras mínimo varia conforme a máquina onde se está a fazer a recolha de informação, podendo aumentar ou diminuir de uma máquina para a outra.

O *Perf* faz a recolha da informação da informação e guarda essa informação em *Buffers* chamados *Samples*,

posteriormente essa informação é eventualmente guardada no ficheiro *perf.data* referido anteriormente.

O comando que pode ser usado para se alterar o intervalo de tempo de recolha da informação é o seguinte:

```
54  perf record -e cpu-clock --freq=8000 ./naive
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
```

Como podemos ver é só adicionar a flag *-freq*, esta flag altera a frequência de amostragem, variando de máquina para máquina, como foi referido anteriormente ou conforme o utilizador desejar.

## 5. Parte 2 - Contagem de eventos de Hardware

O objetivo da segunda parte do tutorial, é usar e introduzir o uso de eventos de desempenho de *Hardware* em torno de toda a aplicação. Durante este tutorial é usado o mesmo programa que foi usado na primeira parte deste tutorial (*naive.c*), para além desse programa é usado um segundo (*interchange.c*) que contem umas optimizações relativamente ao primeiro, contudo este programa faz o mesmo que o primeiro, ou seja é um programa que exemplifica a multiplicação de matrizes.

### 5.1. Código Fonte do Programa

Como foi referido em cima, o programa *naive.c* sofreu uma optimização no código, dando origem ao programa *interchanche.c*. A optimização referida aconteceu na função *multiply\_matrices*, sendo que a função original é a seguinte:

```

void multiply_matrices()
{
    int i, j, k;

    for (i = 0 ; i < MSIZE ; i++) {
        for (j = 0 ; j < MSIZE ; j++) {
            float sum = 0.0 ;
            for (k = 0 ; k < MSIZE ; k++) {
                sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
            }
            matrix_r[i][j] = sum ;
        }
    }
}

```

A optimização feita neste excerto de código, foi trocar a ordem dos ciclos, nomeadamente o ciclo mais interior. Com isto o programa acesa à memória de forma sequencial para os elementos da matriz a. Com esta troca tiramos partido da localidade espacial dos dados, tirando um melhor partido da *cache*, isto porque quando o programa vai ler à memória guarda em *cache* bloco de dados, mas como os dados estão a ser acedidos de forma sequencial. Como os dados estão organizados de forma sequencial o acesso é feito mais rápido.

A alteração feita à função *multiply\_matrices* pode ser vista no excerto de código em baixo.

```

void multiply_matrices()
{
    void multiply_matrices()
{
    int i, j, k;

    // Loop nest interchange algorithm
    for (i = 0 ; i < MSIZE ; i++) {
}
}

```

```

for (k = 0 ; k < MSIZE ; k++) {
    for (j = 0 ; j < MSIZE ; j++) {
        matrix_r[i][j] = matrix_r[i][j] +
            (matrix_a[i][k] * matrix_b[k][j]) ;
    }
}

```

Depois desta alteração corri de novo o *perf stat* de forma a ver se obtive algum *speed-up* com a optimização feita. O comando usado foi:

```
perf stat -e cpu-clock,instructions ./interchange
```

sendo o resultado obtido foi

```

Performance counter stats for ./interchange :
      140.088408      cpu-clock (msec)
      907314368      instructions
      0.141760903 seconds time elapsed

```

Analisando o tempo gasto no primeiro código e neste segundo código podemos ver que temos um *speedup* de 1.3071 ou seja é um *speedup* de aproximadamente 30% relativamente ao primeiro código. Com isto podemos verificar que a optimização feita teve algum efeito positivo no desempenho do programa.

## 5.2. Análise do Desempenho

Posteriormente, neste tutorial procedi a medição do desempenho obtido com as alterações feitas, para isso corri o comando em baixo apresentado, tanto para o programa *naive.c* como para o programa *interchange.c*.

```
perf stat -e cpu-cycles,instructions,cache-references,cache-misses,branch-  
instructions,branch-misses,bus-cycles,L1-dcache-loads,L1-dcache-load-  
misses,L1-dcache-stores,L1-dcache-store-misses,LLC-loads,LLC-load-  
misses,LLC-stores,LLC-store-misses,dTLB-load-misses,dTLB-store-misses,  
iTLB-load-misses,branch-loads,branch-load-misses ./interchange
```

Com o resultado obtido por o comando referido para os dois programas construi a tabela 2

Ao fazermos uma análise à tabela 2, podemos verificar que a optimização feita no código fonte teve algum efeito no desempenho do programa, como foi referido anteriormente. Esta técnica de optimização chamada de *Loop Nested*, como já era de esperar provocou um menor número de *cpu-cycles* no código optimizado *interchange* em relação ao código não optimizado *naive*, isto acontece porque a cada ciclo é feito um maior numero de trabalho/cálculo do que na versão não optimizada. Como tal são necessários menos ciclos para se obter o mesmo resultado que na versão não optimizada.

No que toca ao primeiro nível da *cache*, os dois programas (optimizado e não optimizado) têm praticamente os mesmos valores para os *loads*, quanto aos *misses*, o programa optimizado apresenta um número muito mais reduzido do que o programa não optimizado, isto acontece, devido ao facto da técnica utilizada na optimização tirar um maior partido da localidade dos dados, estando estes armazenados de forma sequencial fazendo com que ocorram menos *misses*.

9  
10  
11  
12  
13  
14  
15  
16

Event Name	Naive	Interchanged
cpu-cycles	547997864	385408603
instructions	931779569	873670026
cache-references	8320994	379814
cache-misses	27905	18545
branch-instructions	133776058	126115011
branch-misses	277424	256916
bus-cycles	0	0
L1-dcache-loads	240142105	243739176
L1-dcache-load-misses	54162900	7343843
L1-dcache-stores	9683369	125326167
L1-dcache-store-misses	290076	98819
LLC-loads	7231132	274505
LLC-load-misses	3056	4625
LLC-stores	259152	203242
LLC-store-misses	27378	14222
dTLB-load-misses	5896	9885
dTLB-store-misses	722	445
iTLB-load-misses	633	0
branch-loads	131571446	123282031
branch-load-misses	6534783	5270872

Tabela 2. EVENTOS DE HARDWARE: NAIVE VS INTERCHANGE

Para o ultimo nível de *cache*, o comportamento dos dados apresentados relativamente aos *misses*, deixou-me surpreso de uma certa forma, pois estava à espera de um número mais reduzido para a versão optimizada do que para a versão não optimizada, e como podemos ver pela a análise da tabela 2 o que acontece é o contrário, ou seja existe um maior numero de *misses* para a versão optimizada do que para a versão não optimizada, fenómeno que não consigo explicar. No que toca aos resultados de *loads*, estes apresentam um numero mais reduzido para a versão optimizada, o que já era de se esperar uma vez que são carregados um maior numero de dados de cada vez para os níveis mais superiores de *cache*.

Mais uma vez posso dizer que as optimizações feitas tiveram um impacto positivo no desempenho do programa, mostrando melhorias no acesso aos dados, no numero de ciclos e consequentemente o tempo de execução do programa apresentado um *speedup* em relação à versão não optimizada.

## 5.3. Rácio e Taxas

Depois de efetuado a análise de desempenho dos algoritmos, procedi ao cálculo dos rácio e taxas para as duas versões dos programas. Estas medidas permite-nos ter uma melhor percepção do que realmente acontece de um programa par o outro, as formulas utilizadas para os cálculos destas medidas foram:

- Instructions per cycles = instructions / cycles
- L1 cache miss ratio = L1-dcache-load-misses/L1-dcache-loads
- L1 cache miss rate PTI<sup>1</sup> = L1-dcache-load-misses / (instructions / 1000)
- Data TLB miss ratio = dTLB-load-misses / cache-references

1. Per Thousand Instructions

- Data TLB miss rate PTI = dTLB-load-misses / (instructions / 1000)
- Branch mispredict ratio = branch-misses / branch-instructions
- Branch mispredict rate PTI = branch-misses / (instructions / 1000)

Depois de efetuados estes cálculos para os dois programas construi a tabela 3 que exprime esses mesmo cálculos, quer para a versão não optimizada (*naive*) quer para versão optimizada (*interchange*).

RATIO or RATE	NAIVE	INTERCHANGE
Elapsed time (seconds)	0.189498719	0.144975696
Instructions per cycle	1.70 IPC	2.27 IPC
L1 cache miss ratio	0.2255	0.030
L1 cache miss rate PTI	58.13	7.88
Data TLB miss ratio	0.00071	0.026
Data TLB miss rate PTI	0.0063	0.011
Branch mispredict ratio	0.0021	0.00204
Branch mispredict rate PTI	0.29774	0.2940

Tabela 3. RÁCIO E TAXAS: NAIVE VS INTERCHANGE

Como já tinha referido anteriormente existe um *speedup* da versão optimizada em relação a versão não optimizada. Analisando os tempos apresentados na tabela 3 e calculando o *speedup* podemos comprovar isso mesmo.

$$\frac{0.189498719}{0.144975696} = 1.3071068064 \quad (1)$$

Verificando o numero de instruções por ciclo, vemos que a versão optimizada apresenta um maior numero de instruções por ciclo do que a versão não optimizada, o que já era de se esperar. Isto acontece porque são executadas mais instruções por ciclo, sendo que não são necessários um maior numero de ciclos, como na versão não optimizada, para se obter o mesmo resultado. É feito um maior trabalho, ou seja há um maior numero de instruções, na versão optimizada do que na versão não optimizada.

Quanto ao primeiro nível de cache o rácio de *misses* é menor para a versão optimizada do que para a versão a versão não optimizada. No que toca a taxa de *misses* por milhar de instruções também esses valores são mais reduzidos para a versão optimizada do que para a versão não optimizada.

No que toca aos *Data TLB misses* o rácio para a versão não optimizada apresenta valores inferiores do que na versão optimizada, o mesmo acontece para a taxa por milhar dos *Data TLB misses*. Estes valores mais uma vez apanharam-me de surpresa, uma vez que estava à espera de valores inferiores para a versão optimizada do que para a versão não optimizada o que não acontece como se pode verificar. Mais uma vez não consigo explicar o porque destes resultados.

## 6. Parte 3 - Perfis de Eventos de Hardware

A terceira e ultima parte deste tutorial, é feita uma análise de um perfil mais completo dos eventos de *Hardware*, de uma certa forma, é feita uma revisão em que engloba as duas partes anteriores do tutorial.

A técnica de *profiling* usada nesta parte do tutorial, é a medição de desempenho com base em amostragem. Esta é uma técnica de medição estatística, em que o *Perf* faz uma seleção de amostras e guarda-as no ficheiro *perf.data*, depois de feita esta seleção as amostras individuais são agregadas durante o processamento dos dados as estatísticas finais dão-nos uma perspectiva interior do desempenho e comportamento do programa.

O método mais usado para a seleção de amostras é a utilização de um período de amostragem fixo, que basicamente é o numero de eventos que ocorrem entre amostras. Cada evento que é recolhido tem o seu próprio período.

Nesta ultima parte do tutorial, alterei o tamanho das matrizes quer para a versão não optimizada (*Naive*) quer para a versão optimizada (*Interchange*). O tamanho das matrizes inicialmente era  $500 \times 500$ , uma vez que temos 3 matrizes e cada *float* ocupa 4 Bytes então:

$$\frac{((500 \times 500) \times 3) \times 4}{1024^2} = 2.8610229492 \quad (2)$$

Como podemos ver pela equação 2 as 3 matrizes usadas no código de multiplicação de matrizes ocupam 2.9 MBytes aproximadamente como tal cabem todas no nível 3 da *cache*.

Ao alterar o tamanho das matrizes, fiz um aumento para 2048, ou seja  $2048 \times 2048$ , quer para a versão não optimizada quer para a versão não optimizada.

$$\frac{((2048 \times 2048) \times 3) \times 4}{1024^2} = 48 \quad (3)$$

Ao analisarmos a equação 3 verificamos que as 3 matrizes com este tamanho ocupam 48 MBytes com isto garante que as matrizes não cabem na *cache*, ficando armazenadas em memória. Com esta alteração foram criados dois novos códigos, um não optimizado ao qual chamei *naive\_large.c* e um optimizado ao qual chamei *interchange\_large.c*.

Depois de realizada estas alterações, ambos os códigos foram compilados com o mesmo compilador que nas partes anteriores deste tutorial, bem como as mesmas *flags*.

### 6.1. Modo Contagem: *naive\_large* vs *interchange\_large*

A tabela 4 e a tabela 5 apresentam os resultados obtidos da execução do *perf* em modo de contagem, de uma maneira sintetizada e de fácil análise, sendo que estas tabelas representam a contagem recolhida bem como os rácios e taxas, respetivamente.

O comando do *perf* executado para a aplicação *naive\_large* foi:

```
perf record -c 100000 -e cpu-cycles,instructions,cache-references,cache-misses<-->,LLC-loads,LLC-load-misses,dTLB-load-misses,branches,branch-misses ./naive_large
```

e para a aplicação *interchange\_large*

```
perf record -c 100000 -e cpu-cycles,instructions,cache-references,cache-misses<-->,LLC-loads,LLC-load-misses,dTLB-load-misses,branches,branch-misses ./interchange_large
```

A flag `-c 100000` especifica um período fixo de amostragem de `100000 cpu-cycles`.

Event Name	Naive Large	Interchange Large
Elapsed Time	103.3436	10.5054
cpu-cycles	117011200000	17004500000
instructions	40648400000	39952000000
cache-references	5709100000	26100000
cache-misses	4898800000	22100000
LLC-loads	5781400000	25800000
LLC-load-misses	4930900000	22800000
dTLB-load-misses	1700000	100000
branches	3914700000	3786900000
branch-misses	2200000	1800000

Tabela 4. MODO CONTAGEM: INTERCHANGE LARGE VS NAIVE LARGE

Ao analisarmos a tabela 4 em cima apresentada, podemos verificar que em termos de tempo, a versão optimizada do programa (*interchange\_large*) foi mais rápida apresentando um *speedup* de aproximadamente 10 como pode ser comprovado pela equação 4. Podemos verificar também que com o aumento do tamanho dos dados os tempos de execução de ambas as aplicações também aumentou.

$$\frac{103.3436}{10.5054} = 9.8371884935 \quad (4)$$

No que toca a *cache misses* e *LLC-load-misses* podemos verificar que para a versão optimizada existe um menor numero de *misses* para ambos os eventos. Isto acontece pois o programa optimizado tira um melhor partido da localidade dos dados.

Analizando agora a tabela 5, que diz respeito aos rácios e taxas, podemos verificar que em termos de instruções por ciclo a versão optimizada do programa apresenta valores superiores relativamente à versão não optimizada. Isto acontece porque o programa optimizado executa um maior número de instruções por ciclo do que a versão não optimizada.

Quanto à análise da taxa *Cache miss rate PTI* e da taxa *LLC load miss rate PTI* verificamos que mais uma vez, é a versão optimizada que apresenta menores taxas relativamente à versão não optimizada. Isto acontece, devido à localidade dos dados, como já foi falado atrás.

EVENT NAME	NAIVE LARGE	INTERCHANGE LARGE
IPC	0.3474	2.3495
Cache miss ratio	0.85	0.86
Cache miss rate PTI	120.516	0.55
LLC load miss ratio	0.85	0.88
LLC load miss rate PTI	121.31	0.57
dTLB load miss rate PTI	0.042	0.0025
Branch mispredict ratio	0.00056	0.00048
Branch mispred rate PTI	0.054	0.045

Tabela 5. MODO CONTAGEM: RATES AND RATIOS (NAIVE LARGE VS INTERCHANGE LARGE)

Relativamente aos restantes rácios e taxas, ambas as aplicações apresentam valores semelhantes. O que já era de se esperar pois a optimização que foi feita, apenas tirou partido da localidade dos dados e como tal tira um maior partido da *cache*, que por sua vez faz menos acessos à memória principal. Como consequência disto, o tempo de execução da aplicação também diminui.

## 6.2. Visualização dos Perfis Baseados em Eventos

Nesta parte do tutorial, apenas executei o comando para consultar os dados recolhidos com *perf record* de forma a conseguir construir as tabelas 4, 5, 6 e 7. Sendo que o comando executado quer para a aplicação *naive\_large* quer para aplicação *interchange\_large*, foi:

```
perf report -n --no-source --stdio --percent-limit 0.1
```

o resultado obtido para a aplicação *naive\_large* foi:

```

# Samples: IM of event    cpu-cycles
# Event count (approx.): 117011200000
# Overhead      Samples   Command      Shared Object      Symbol
# .....          .....   .....          .....          .....   ↔
#
#      99.38%     1162876  naive_large  naive_large      [.] ←
#           multiply_matrices
#      0.17%       1986   naive_large  [kernel.kallsyms] [k] 0←
#           xfffffff81096e3e

# Samples: 406K of event  instructions
# Event count (approx.): 40648400000
# Overhead      Samples   Command      Shared Object      Symbol
# .....          .....   .....          .....          .....   ↔
#
#      99.12%     402902   naive_large  naive_large      [.] ←
#           multiply_matrices
#      0.24%       986    naive_large  naive_large      [.] ←
#           initialize_matrices
#      0.12%       486    naive_large  libc-2.12.so   [.] __random

# Samples: 57K of event  cache-references
# Event count (approx.): 5709100000
# Overhead      Samples   Command      Shared Object      Symbol
# .....          .....   .....          .....          .....   ↔
#
#      99.87%     57017   naive_large  naive_large      [.] ←
#           multiply_matrices

# Samples: 48K of event  cache-misses
# Event count (approx.): 4898800000
# Overhead      Samples   Command      Shared Object      Symbol
# .....          .....   .....          .....          .....   ↔
#
#      99.97%     48971   naive_large  naive_large      [.] ←
#           multiply_matrices

# Samples: 57K of event  LLC-loads
# Event count (approx.): 5781400000
# Overhead      Samples   Command      Shared Object      Symbol
# .....          .....   .....          .....          .....   ↔
#
#      99.95%     57786   naive_large  naive_large      [.] ←
#           multiply_matrices

# Samples: 49K of event  LLC-load-misses
# Event count (approx.): 4930900000
# Overhead      Samples   Command      Shared Object      Symbol
# .....          .....   .....          .....          .....   ↔
#
#      99.97%     49296   naive_large  naive_large      [.] ←
#           multiply_matrices

# Samples: 17 of event  dTLB-load-misses
# Event count (approx.): 1700000
# Overhead      Samples   Command      Shared Object      Symbol
# .....          .....   .....          .....          .....   ↔
#
#      58.82%      10    naive_large  naive_large      [.] ←
#           multiply_matrices
#      5.88%       1    naive_large  [kernel.kallsyms] [k] 0←
#           xfffffff8104f488
#      5.88%       1    naive_large  [kernel.kallsyms] [k] 0←
#           xfffffff81057e50

```

68	0.77%	2	interchange_lar	[kernel.kallsyms]	[k] 0↔
69	0.77%	2	interchange_lar	[kernel.kallsyms]	[k] 0↔
70	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
71	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
72	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
73	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
74	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
75	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
76	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
77	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
78	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
79	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
80	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
81	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
82	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
83	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
84	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
85	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
86	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
87	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
88	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
89	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
90	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
91	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
92	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
93	0.38%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
		xfffffff814fea42			60
					61
#	Samples: 221 of event cache-misses				62
#	Event count (approx.): 22100000				63
#	Overhead	Samples	Command	Shared Object	Symbol
#	.....	.....	.....	.....	↔
1	#				68
2	53.39%	118	interchange_lar	interchange_large	[.] 0↔
3	x000000000000083d				69
4	18.55%	41	interchange_lar	interchange_large	[.] 0↔
5	x0000000000000830				70
6	12.22%	27	interchange_lar	interchange_large	[.] 0↔
7	x0000000000000846				71
8	9.05%	20	interchange_lar	interchange_large	[.] 0↔
9	x0000000000000839				72
10	3.17%	7	interchange_lar	[kernel.kallsyms]	[k] 0↔
11	xfffffff8127d387				73
12	3.17%	7	interchange_lar	interchange_large	[.] 0↔
13	x000000000000084f				74
14	0.45%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
15	xfffffff81126b85				75
16					76
17	#	Samples: 258 of event LLC-loads			77
18	#	Event count (approx.): 25800000			78
19	#	Overhead	Samples	Command	Symbol
20	#	.....	.....	.....	↔
21	#				83
22	46.90%	121	interchange_lar	interchange_large	[.] 0↔
23	x000000000000083d				84
24	20.54%	53	interchange_lar	interchange_large	[.] 0↔
25	x0000000000000846				85
26	11.63%	30	interchange_lar	interchange_large	[.] 0↔
27	x0000000000000830				86
28	9.30%	24	interchange_lar	interchange_large	[.] 0↔
29	x000000000000084f				87
30	7.75%	20	interchange_lar	interchange_large	[.] 0↔
31	x0000000000000839				88
32	0.78%	2	interchange_lar	[kernel.kallsyms]	[k] 0↔
33	xfffffff8109538c				89
34	0.39%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
35	xfffffff8105483c				90
36	0.39%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
37	xfffffff810580d3				91
38	0.39%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
39	xfffffff81073e74				92
40	0.39%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
41	xfffffff81074786				93
42	0.39%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
43	xfffffff8107e7a6				94
44	0.39%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
45	xfffffff8107e844				95
46	0.39%	1	interchange_lar	[kernel.kallsyms]	[k] 0↔
47	xfffffff8109cf25				96
48	0.39%	1	interchange_lar	interchange_large	[.] 0↔
49	x0000000000000824				97
50					98
51	#	Samples: 228 of event LLC-load-misses			99
52	#	Event count (approx.): 22800000			100
53	#	Overhead	Samples	Command	Symbol
54	#	.....	.....	.....	↔
55	#				102

```

53.07%      121 interchange_lar interchange_large [.] 0↔
18.86%       43 interchange_lar interchange_large [.] 0↔
x000000000000083d
16.67%       38 interchange_lar interchange_large [.] 0↔
x0000000000000846
8.33%        19 interchange_lar interchange_large [.] 0↔
x0000000000000830
3.07%         7 interchange_lar interchange_large [.] 0↔
x000000000000084f

# Samples: 1 of event dTLB-load-misses
# Event count (approx.): 100000
#
# Overhead    Samples  Command      Shared Object      Symbol
# .....          .....   .....          .....   .....   ↔
#
# 100.00%      1 interchange_lar [kernel.kallsyms] [k] 0↔
xfffffff810585c5

# Samples: 37K of event branches
# Event count (approx.): 3786900000
#
# Overhead    Samples  Command      Shared Object      Symbol
# .....          .....   .....          .....   .....   ↔
#
# 68.14%     25803 interchange_lar interchange_large [.] 0↔
x000000000000084f
14.46%      5474 interchange_lar interchange_large [.] 0↔
x0000000000000846
13.06%      4945 interchange_lar interchange_large [.] 0↔
x0000000000000830
2.19%       828 interchange_lar interchange_large [.] 0↔
x000000000000083d
1.76%       665 interchange_lar interchange_large [.] 0↔
x0000000000000839
0.11%        41 interchange_lar interchange_large [.] 0↔
x0000000000000824

# Samples: 18 of event branch-misses
# Event count (approx.): 1800000
#
# Overhead    Samples  Command      Shared Object      Symbol
# .....          .....   .....          .....   .....   ↔
#
# 66.67%      12 interchange_lar interchange_large [.] 0↔
x000000000000083d
22.22%       4 interchange_lar interchange_large [.] 0↔
x0000000000000846
5.56%        1 interchange_lar interchange_large [.] 0↔
x0000000000000824
5.56%        1 interchange_lar interchange_large [.] 0↔
x0000000000000839

```

## 7. Modo Amostragem: *naive\_large* vs *interchange\_large*

Nesta fase, decidi fazer o mesmo tipo de recolha de informação que é falada no tutorial, para isso construi a tabela 6 e a tabela 7. A primeira corresponde à contagem das amostras para os eventos de *hardware* que se encontram na tabela, sendo que a segunda tabela corresponde aos rácios e taxas calculados a partir da primeira tabela. Estas tabelas foram obtidas através da análise dos dados em cima apresentados.

EVENT NAME	NAIVE LARGE	INTERCHANGE LARGE
Elapsed Time	103.3436	10.5054
cpu-cycles	1M amostras	170K amostras
instructions	406K amostras	399K amostras
cache-references	57K amostras	261 amostras
cache-misses	48K amostras	221 amostras
LLC-loads	57K amostras	258 amostras
LLC-load-misses	49K amostras	228 amostras
dTLB-load-miss	17 amostras	1 amostras
branches	39K amostras	37k amostras
branch-miss	22 amostras	18 amostras

Tabela 6. MODO AMOSTRAS: NAIVE LARGE VS INTERCHANGE LARGE

106 Ao analisarmos a tabela 6, podemos verificar que para  
107 a versão optimizada do código *interchange\_large*, com-  
108 parativamente com a versão não optimizada do código  
109 *naive\_large* é recolhido um menor número de amostras para  
110 os eventos expressos na mesma tabela.

111 De uma certa forma já estava a esperar destes resultados,  
112 uma vez que a versão optimizada do código executa num  
113 tempo cerca de 10 vezes inferior em relação à versão não  
114 optimizada, como tal se executa em menor tempo também  
115 faz uma recolha de amostras também é menor.

EVENT NAME	NAIVE LARGE	INTERCHANGE LARGE
IPC	0.406	2.35
Cache miss ratio	0.84	0.86
Cache miss rate PTI	118.23	0.55
LLC load miss ratio	0.86	0.88
LLC load miss rate PTI	120.68	0.57
dTLB load miss rate PTI	0.042	0.0025
Branch mispredict ratio	0.00056	0.00049
Branch mispred rate PTI	0.054	0.045

129 Tabela 7. MODO AMOSTRAS: RATES E RATIOS (NAIVE LARGE VS  
130 INTERCHANGE LARGE)

131  
132 Por fim, ao analisarmos a tabela 7, verificamos que o  
133 comportamento e até mesmo o valor dos rácios e das taxas  
134 é semelhante aos da tabela 5, obtida no modo contagem.  
135

136 À semelhança do que acontece na tabela 6, já estava a  
137 esperar dos resultados obtidos nesta tabela. Com isto quero  
138 dizer que esperava uma semelhança entre os valores dos  
139 rácios e taxas, quer para o modo de amostragem quer para o  
140 modo de contagem, isto porque as aplicações não mudaram  
141 e apenas foi feita uma *perf record* para cada uma das  
142 aplicações.

145

## 8. Flame Graphs

Os *Flame Graphs* [1] são gráficos que nos permitem visualizar perfis de *software*. Nestes gráficos são apresentados os métodos, permitindo a qualquer pessoa visualizar de forma rápida quais os métodos que são consomem maior tempo de CPU.

No eixo dos XX é representado a população da pilha de perfil, sendo que no eixo dos YY é representado a profundidade do padrão. Cada retângulo representa a *stack frame*. De notar que as cores deste tipo de gráfico são escolhidas aleatoriamente não tendo qualquer significado.

### 8.1. Geração de *Flame Graphs*

Este tipo de gráficos são obtidos através de *scripts* que vão tratar os dados recolhidos pelo *perf* e armazenados no ficheiro *perf.data*.

A sequência de comandos em baixo apresentada, mostra um exemplo de como obtive os meus *Flame Graphs*.

```
perf record --ag -F 99 ./large_naive
```

```
perf script | /share/jade/SOFT/FlameGraph/stackcollapse-perf.pl > out.perf→
folded
```

```
cat out.perf-folded | /share/jade/SOFT/FlameGraph/flamegraph.pl > ←
large_naive_flame_graph.svg
```

## 8.2. Flame Graph das 4 Aplicações Usadas neste Trabalho

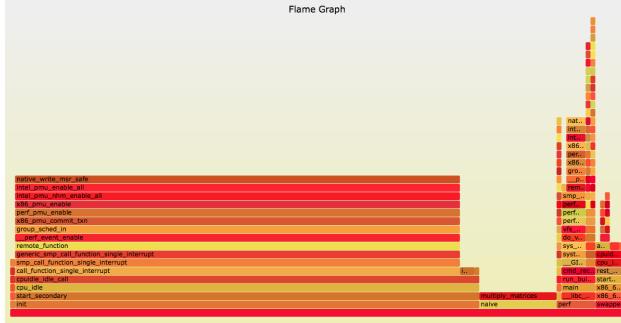


Figura 1. Flame Graph da Aplicação *naive*

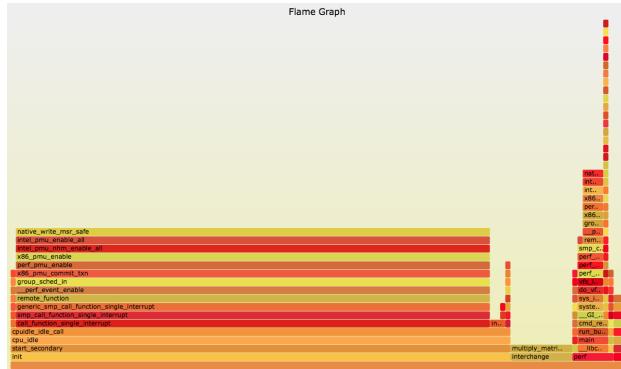


Figura 2. Flame Graph da Aplicação *interchange*

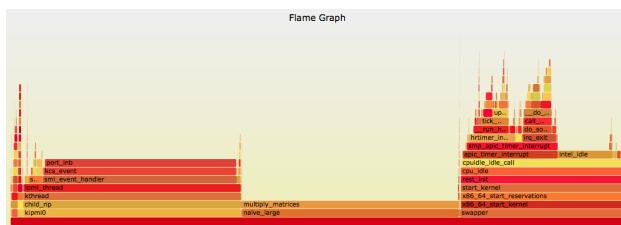


Figura 3. Flame Graph da Aplicação *naive\_large*

## 9. Conclusão

Como foi referido anteriormente, o desenvolvimento deste trabalho tinha como objetivo introduzirmos a ferramenta *perf* e praticarmos a sua utilização. Para isso seguimos o tutorial fornecido pelo professor, tutorial esse que estava dividido em 3 partes. Ao longo destas 3 partes o

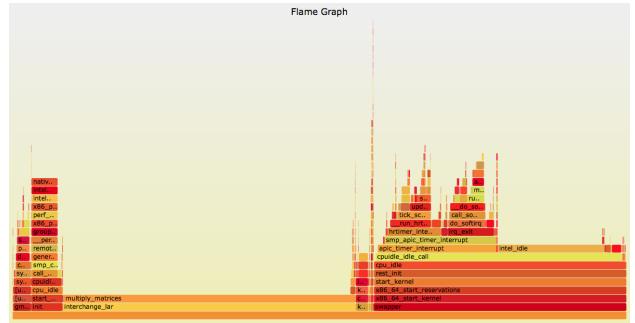


Figura 4. Flame Graph da Aplicação *interchange\_large*

tutorial sugeria-nos comandos do *perf* para utilizarmos e experimentarmos, graças a este tutorial posso dizer que já tenho um certo conhecimento acerca da utilização da ferramenta *perf*.

Na primeira, deste trabalho foi-nos introduzido o *perf*, apresentando-nos alguns comandos básicos da ferramenta para recolha de informação e leitura/tratamento da mesma, bem como nos ajudou a encontrar *hotspots* numa aplicação. Na segunda parte foi-nos apresentados alguns contadores e fizemos análise dos resultados obtidos para esses contadores. Na ultima e terceira parte, fiz uma análise completa de desempenho para eventos de hardware. Em termos de dificuldades encontradas ao longo do desenvolvimento deste trabalho posso dizer que não foram muitas, ou mesmo quase nenhuma.

A maior dificuldade por mim sentida, basicamente foi em termos de análise de alguns resultados, contudo penso que com alguma pesquisa e esforço consegui superar essa dificuldade, acabando por analisar os resultados obtidos.

No que toca à ferramenta *perf*, posso concluir que é uma ferramenta bastante útil, que nos permite fazer uma análise pormenorizada de uma aplicação, permitindo-nos encontrar, por exemplo *hotspots*, que posteriormente podem ser optimizados para obtermos um maior desempenho da aplicação. Para além de considerar uma ferramenta bastante útil, considero que o *perf* é bastante prático e fácil de usar, características que a tornam ainda mais interessante.

Quanto à parte dos *Flame Graphs*, posso concluir que é uma técnica bastante prática e fácil de usar/obter. Estes gráficos permitem-nos visualizar o perfil do *software* em análise, permitindo-nos ver quais os métodos que consomem mais tempo de CPU, o que torna esta técnica interessante para outro tipo de análise de *software*.

Globalmente, faço uma apreciação bastante positiva deste trabalho, penso que os objetivos foram todos cumpridos e o conhecimento adquirido com o desenvolvimento do trabalho também foi o esperado. Em termos de trabalho futuro, posso dizer que a ferramenta *perf* vai ser uma ferramenta que vou ter em conta sempre que necessitar de analisar algum código, por isso penso que vai ser bastante utilizada da minha parte daqui para a frente.

## Referências

- [1] Site de flame graphs. <http://www.brendangregg.com/flamegraphs.html>.
- [2] Site do tutorial. <http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/>.

## 7 Conclusão

Como podemos verificar ao longo do semestre foram desenvolvidos os 5 trabalhos apresentados anteriormente, uns com um grau de dificuldade maior que outros, mas de uma maneira geral penso que o objetivo de todos eles foi cumprido. Todas as dificuldades encontradas, com alguma pesquisa e um pouco de análise da bibliografia disponibilizada, foram superadas.

Relativamente à disciplina em si, penso que a disciplina é extremamente útil, e permitiu-me ganhar competências práticas e teóricas no que toca à análise e monitorização de aplicações. Para além desta componente prática que adquiri também posso dizer que fiquei com um conhecimento muito mais abrangente, a nível teórico do funcionamento de um sistema de computação. Todos estes conhecimentos adquiridos ao longo do semestre, irão-me permitir de agora em diante olhar de uma maneira diferente no momento em que tiver de desenvolver software.

Ao longo do semestre apercebemos-nos que a matéria abrangida pela disciplina é muito extensa, sendo que nesta apenas foram lecionados alguns tópicos. Apesar disso, temos uma valia, relativamente aos assuntos que foram abordados na disciplina, que é o facto de existir uma basta gama de referências bibliográficas nas quais nos podemos basear no futuro. Posso dizer com toda a certeza, que mesmo depois de finalizadas as atividades letivas da disciplina, os assuntos abordados nesta, vão continuar a ser objeto de estudo ao longo do meu percurso académico e profissional, uma vez que todos os tópicos aqui abordados podem ser aplicados em qualquer área das ciências da computação.

Para finalizar, posso concluir que a disciplina foi das disciplinas mais interessantes e úteis que já tive no meu percurso académico, sendo que, em forma de autoavaliação, posso concluir que o meu desempenho ao longo do semestre foi bastante positivo. Quanto ao conhecimento adquirido nesta UC, posso também concluir que, como referi anteriormente, foi bastante útil e interessante e de uma forma geral penso que esse conhecimento foi adquirido da forma que era suposto. Contudo reconheço que em alguns dos trabalho práticos desenvolvidos por mim poderia ter aprofundado ainda mais a pesquisa e o meu conhecimento.