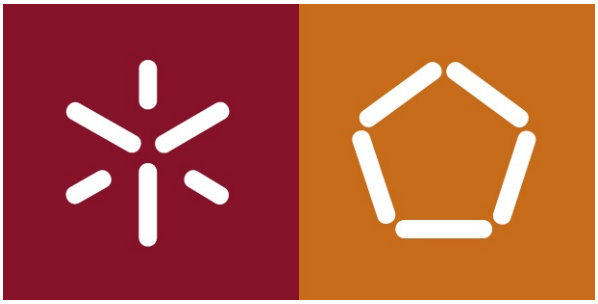

UNIVERSIDADE DO MINHO

ENGENHARIA DE SISTEMAS DE COMPUTAÇÃO
MESTRADO EM ENGENHARIA INFORMÁTICA
2019/2020

Portfólio dos Trabalhos Práticos

Trabalho realizado por: ***Número de Aluno:***
José Pinto A81317
Pedro Barbosa A82068

July 11, 2020

Introdução

Este portfólio contem todos os trabalhos práticos realizadas na Unidade Curricular de Engenharia de Sistemas de Computação.

Os trabalhos refletem o foco da cadeira, abordando não só a análise do desempenho de aplicações mas também o efeito destas nos recursos do sistema operativo.

Os trabalhos foram realizados no *cluster* SeARCH e numa máquina Solaris 11.

I. TP1

O primeiro trabalho consistiu em observar o impacto que diferentes fatores têm na execução de uma aplicação através de diversas ferramentas e também em como processar os valores medidos. As aplicações testadas foram vários *benchmarks* da NAS.

Entre os fatores testados destacam-se:

- Três paradigmas diferentes, sequencial, memória partilhada (OpenMP) e memória distribuída (MPI);
- Máquinas com processadores de diferentes arquiteturas Intel (Ivy Bridge e Nehalem);
- Compiladores (gfortran da Gnu e ifort da Intel) e opções de compilação (-O3, -ipo, etc) diferentes.

Outra componente importante do trabalho foi a utilização de ferramentas e o processamento de resultados. Foram utilizadas ferramentas como o mpstat e o free para medir parâmetros como a utilização dos vários *cores* e a comunicação por rede.

Para processar a informação significativa gerada pelas medições foram desenvolvidos vários *scripts* em Python que apresentam a informação em gráficos.

II. TP2

O TP2 introduziu o DTrace, uma ferramenta de *dynamic tracing*, desenvolvida para ser utilizada no sistema operativo Solaris. Tendo em conta a natureza da ferramenta utilizada, todos os testes foram realizados numa máquina Solaris com a ferramenta previamente instalada.

O objetivo deste projeto era perceber melhor o que era o DTrace e para nos familiarizarmos com as suas funcionalidades de forma a poderem ser aplicadas no projeto seguinte. Para isso foram realizados uma série de exercícios práticos.

III. TP3

Este projeto consistiu na aplicação das funcionalidades do DTrace num projeto desenvolvido no semestre anteriore na cadeira de Paradigmas de Computação Paralela. Foi possível realizar novos testes que anteriormente não foram possíveis devido às limitações das ferramentas utilizadas.

Foram ainda desenvolvidas duas implementações alternativas ao OpenMP, uma em C++ 11 e outra com a utilização de PThreads. Foram feitas as devidas medições das mesmas e comparadas com a implementação original em OpenMP.

IV. TP4

O último trabalho focou-se no Perf, uma ferramenta de análise de desempenho. O estudo da ferramenta incidiu principalmente sobre o tutorial disponibilizado. Com os conhecimentos adquiridos no tutorial, foram analisados diversos algoritmos de ordenação.

No trabalho também foram utilizados *FlameGraphs*, uma forma de visualização que permite facilmente perceber a distribuição do tempo de execução pelas secções do código.

ESC-TP1

José Pinto A81317 - Pedro Barbosa A82068

I. Introdução

Para a grande maioria das linguagens de programação encontram-se disponíveis vários compiladores. Para além da acessibilidade(preço, plataforma, etc) uma questão importante é o desempenho do programa compilado.

Em norma, cada compilador disponibiliza um conjunto de optimizações. A eficácia das optimizações de cada compilador pode variar conforme vários fatores, como a arquitetura do *hardware* e as características do programa.

No caso do Fortran, dois dos compiladores mais usados são o IFORT da Intel e o gFortran da GNU.

A primeira parte deste trabalho lida com estes aspetos e analisa o impacto destes no desempenho dos *benchmarks*.

É importante avaliar métricas para além do tempo de execução. A monitorização da utilização de recursos físicos, desde o processador à rede, permite avaliar se estes estão a ser usados de uma forma óptima e encontrar possíveis gargalos de desempenho.

A segunda parte lida com a utilização de ferramentas para obter e processar estas métricas.

II. Benchmarks

Os *benchmarks* testados são os SP-MZ, BT-MZ e LU-MZ da versão NPB 3.3.1-MZ.

III. SER

A. Flags

As *flags* -xHost (IFORT) e -march=native (gfortran) foram utilizadas com o intuito de optimizar o código para o processador onde foi compilado.[https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-xhost-qxhost][https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html]

A *flag* -ipo, apenas disponivel para o compilador IFORT, permite ao compilador aplicar a *Interprocedural Optimization* (IPO). Esta funcionalidade analisa o programa na totalidade, em vez de um segmento de código específico, permitindo optimizações como *inlining* (substituir chamada da função pelo código da função) e análise de *alias* (determinar se dois apontadores apontam para sítios distintos, o que permite executar instruções fora de ordem). [https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-ipo-qipo] [https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-interprocedural-optimization-ipo]

B. Medições

As seguintes tabelas apresentam o desempenho de cada compilador, com as diferentes *flags*, para os diferentes *benchmarks*. Os resultados obtidos são uma média de 5 execuções obtidas nos nodos 641 e 431 do SeARCH.

SP	**gfortran**			**ifort**		
	-O0	-O3	-O3 -march=native	-O0	-O3	-O3 -xHost -ipo
431	914	139	139	923	114	114
641	619	102	94	783	79	74

TABLE I
Medições feitas em segundos para o *benchmark* SP

BT	**gfortran**			**ifort**		
	-O0	-O3	-O3 -march=native	-O0	-O3	-O3 -xHost -ipo
431	1023	221	223	1052	194	179
641	719	192	158	789	151	123

TABLE II
Medições feitas em segundos para o *benchmark* BT

LU	**gfortran**			**ifort**		
	-O0	-O3	-O3 -march=native	-O0	-O3	-O3 -xHost -ipo
431	888	149	150	1451	147	144
641	648	115	111	1314	144	112

TABLE III
Medições feitas em segundos para o *benchmark* LU

Como é possível verificar através das tabelas, os resultados obtidos com a utilização de *flags* chegam a ser 11,7 vezes mais rápidos, para o caso do compilador IFORT da versão LU-MZ no nodo 641, em relação a versões sem qualquer optimização.

A eficácia das *flags* extra varia conforme o *benchmark*. Para qualquer aplicação é sempre importante testar diferentes combinações uma vez que o desempenho relativo entre estas não é fixo.

Decidimos verificar se a diferença entre -O0 e -O3 é perceptível através das ferramentas de monitorização.

Numa abordagem inicial foi utilizada a ferramenta mpstat para avaliar a utilização do processador. Independentemente da versão, mpstat indica que um dos cores está 100% ocupado com o *benchmark* durante a sua execução. É necessário ter cuidado com a interpretação deste valor. Estar 100% do tempo ocupado com o processo não significa que o processador esteve a efetuar trabalho útil. Uma parte significativa deste tempo é gasta à espera de leituras e escritas da memória, entre outros. Uma ferramenta mais apropriada nesta situação seria o perf. [http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html]

Como as métricas dos *stalled cycles* não são suportadas na arquitetura Ivy Bridge, as medições com o *perf* apresentadas foram efetuadas nos nodos 431.

LU	-O0	-O3	-O3 -xHost -ipo
unhalted-cycles	4.22e12	4.27e11	4.25e11
stalled-cycles-frontend	1.55e12	1.93e11	1.93e11
stalled-cycles-backend	1.62e11	9.96e10	9.65e10
instructions	9.92e12	5.84e11	5.8e11
CPI	2.35	1.37	1.36

TABLE IV
RESULTADOS DO *perf stat* PARA DIFERENTES *flags* DO IFORT PARA O *benchmark* LU-MZ

BT	-O0	-O3	-O3 -xHost -ipo
unhalted-cycles	3.05e12	5.66e11	5.20e11
stalled-cycles-frontend	1.14e12	1.96e11	1.6711
stalled-cycles-backend	7.10e10	4.35e10	3.75e10
instructions	7.3e12	1.11e12	1.05e12
CPI	2.39	1.95	2.02

TABLE V
RESULTADOS DO *perf stat* PARA DIFERENTES *flags* DO IFORT PARA O *benchmark* BT-MZ

SP	-O0	-O3	-O3 -xHost -ipo
cycles	2.69e12	3.33e11	3.30e11
stalled-cycles-frontend	7.95e11	1.43e11	1.39e11
stalled-cycles-backend	6.36e10	4.84e10	4.78e10
instructions	7.26e12	5.00e11	5.02e11
CPI	2.70	1.50	2.02

TABLE VI
RESULTADOS DO *perf stat* PARA DIFERENTES *flags* DO IFORT PARA O *benchmark* SP-MZ

A utilização destas *flags* leva a que sejam feitas otimizações, como a vetorização, substituição de várias instruções simples por uma instrução SIMD (single instruction multiple data), que provocam uma diminuição no CPI (número de instruções por ciclo), que é frequentemente usado como métrica de desempenho, tornando a métrica algo enganadora neste contexto.

As métricas que melhor indicam os efeitos das optimizações são simples mas importantes. O número de instruções diminuiu significativamente, tal como o número de *stalled cycles*, ciclos em que o processador não efectuou trabalho útil.

Ao utilizar ferramentas para avaliar o desempenho é necessário interpretar correctamente o significado das métricas e o ambiente de execução, de forma a não serem obtidas conclusões erradas.

IV. OMP

Todas as medições efetuadas para a versão OMP foram realizadas utilizando apenas as *flags* que obtiveram melhores resultados nos testes da versão sequencial. Para o compilador gFortran são *-O3 -march=native* e para o IFORT *-O3 -xHost -ipo*. Para além disso, foi ainda necessário adicionar as *flags* -qopenmp e -fopenmp, para os compiladores IFORT e gFortran respetivamente, de forma a que suportassem a paralelização do código.

Assim como na versão sequencial fora utilizados os nodos 641 e 431 do SeARCH. Uma vez que os nodos apresentam processadores com diferentes quantidades de *cores* a quantidade de *threads* máxima foi alterada, ambos realizam testes para 1, 2, 4, 8 e 16 *threads*. No entanto, o nodo 641 termina com 32 *threads* e o 431 apenas com 24.

Apesar de termos tentado utilizar diferentes módulos e *flags* de compilação não conseguimos executar o *benchmark* LU-MZ com mais de uma *thread*. , pelo que o resultado obtido vai ser omitido da discussão no resto da secção assim como não vai estar presente nos gráficos.

Os seguintes gráficos representam os tempos de execução dos *benchmarks* SP-MZ e BT-MZ, nos diferentes nodos, para ambos os compiladores e com quantidades diferentes de *threads*.

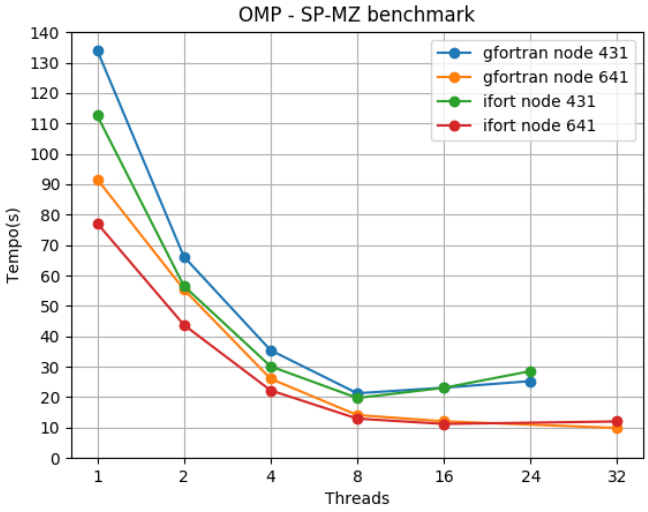

Fig. 1. Tempos de execução do *benchmark* SP-MZ

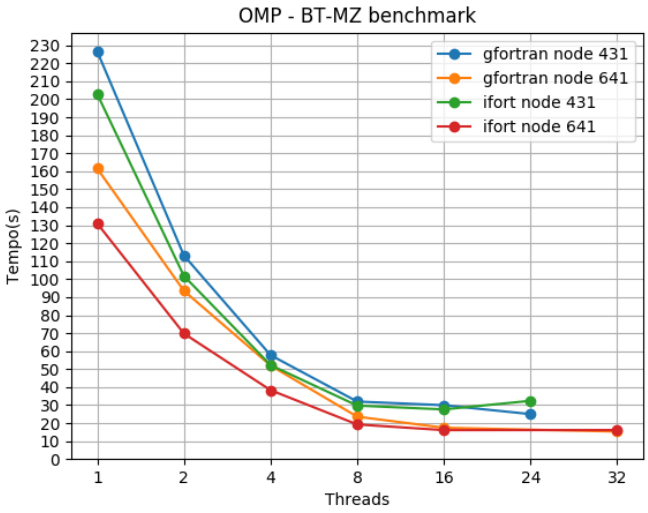

Fig. 2. Tempos de execução do *benchmark* BT-MZ

Como é comum na utilização de OpenMP, os ganhos não escalam idealmente com o número de *threads*.

A melhor combinação varia conforme o *benchmark* utilizado e o nodo.

- Nodo 431 do SP o melhor é o IFORT (8 threads)
- Nodo 641 do SP o melhor é o gFortran (32 threads)
- Nodo 431 do BT o melhor é o gFortran (24 threads)
- Nodo 641 do BT o melhor é o IFORT (32 threads) (apresentam valores bastante semelhantes)

Com esta análise podemos concluir que apesar de muitas vezes ignorado, o compilador é um fator importante na escalabilidade de uma aplicação.

V. MPI+OMP

Na versão híbrida, foram utilizadas as *flags* -O3 para ambos os compiladores e as *flags* -qopenmp e -fopenmp para o compilador IFORT e gFortran respetivamente. Todas as medições foram realizadas em 2 máquinas idênticas em simultâneo, nos nodos 641 e 431 do SeARCH.

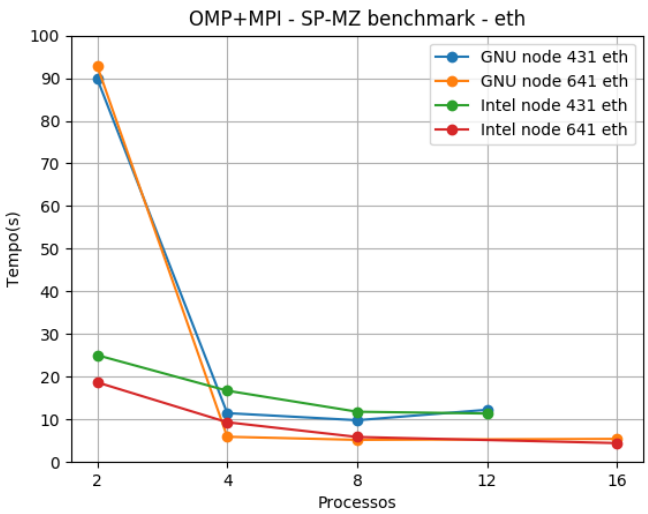

Fig. 3. Tempos de execução do *benchmark* SP-MZ

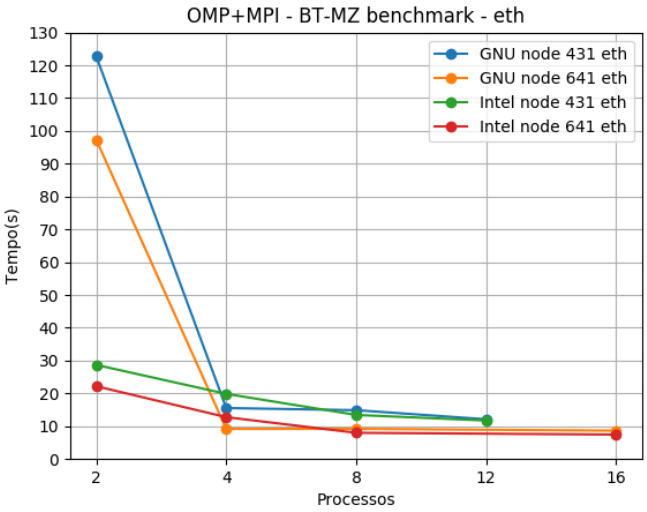

Fig. 4. Tempos de execução do *benchmark* BT-MZ

É possível verificar que a execução com o OpenMPI da GNU apresentou resultados anómalos quando eram utilizados 2 processos. No entanto, para 4 ou mais processos os tempos de execução são conforme o esperado. A primeira ação tomada foi a repetição dos testes, porém a anomalia manteve-se. Decidimos então utilizar o -mpstat para perceber melhor o que se passava.

Os gráficos seguintes representam a média da utilização dos *cores* por aplicações do utilizador durante a execução do *benchmark*. Os valores são referentes a apenas uma das máquinas utilizadas.

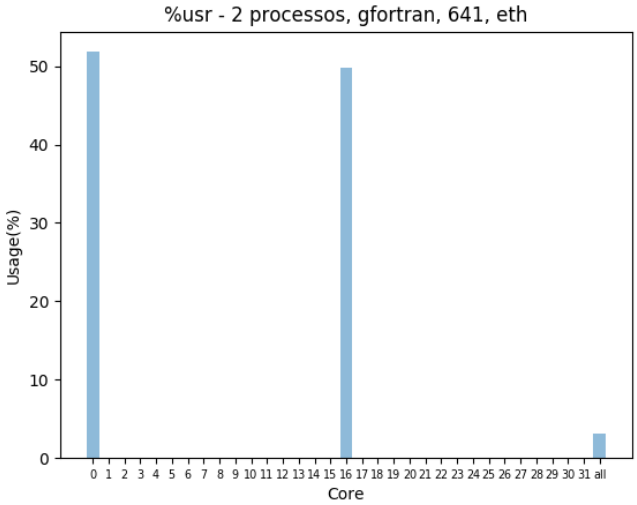

Fig. 5. Utilização dos *cores* para 2 processos (1 por máquina)

Em cada máquina apenas eram usados 2 cores, mesmo com o valor correto da variável de ambiente OMP_NUM_THREADS.

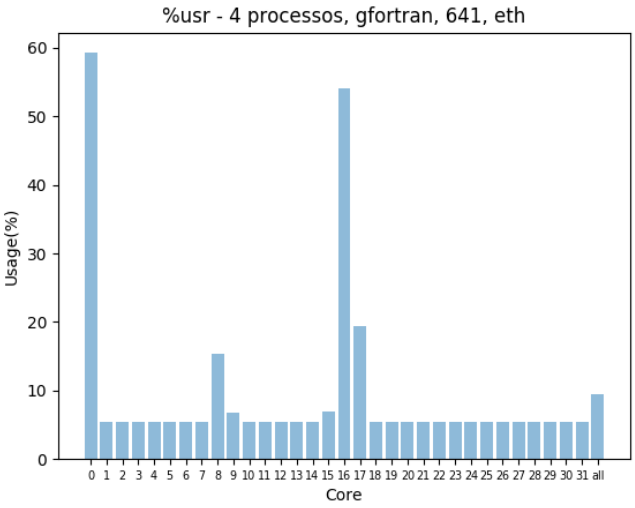

Fig. 6. Utilização dos *cores* para 4 processos (2 por máquina)

No entanto com 2 processos em cada máquina(4 no total), todos os cores lógicos eram usados.

Ao contrário da versão da GNU, a versão do OpenMPI da Intel com um processo por máquina apresentava resultados normais. Por uma questão de curiosidade decidimos também observar a utilização dos *cores*.

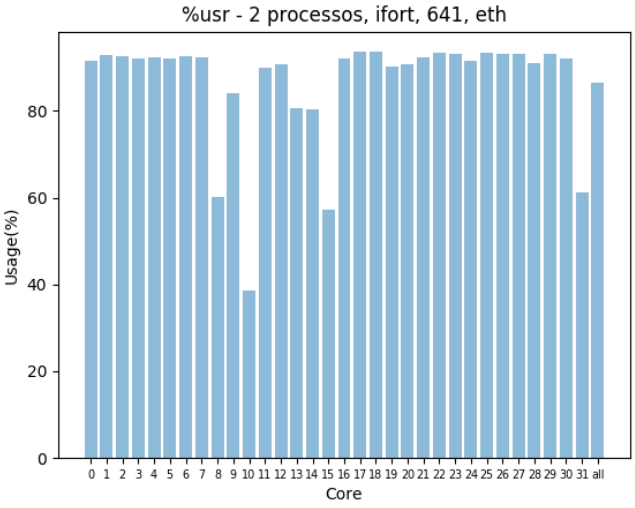

Fig. 7. Utilização dos *cores* para 2 processos (1 por máquina)

A versão da Intel utiliza todos os *cores*, o que nos leva a suspeitar da existência de alguma incompatibilidade entre o *benchmark* e a versão da GNU usada.

Inicialmente, os testes que utilizam ethernet como forma de comunicação entre máquinas foram realizados recorrendo às versões mais recentes de OpenMPI para ambos os compiladores, openmpi_eth/2.0.0 para o gFortran e openmpi_eth/1.8.2 para o IFORT.

Para a versão que utilizava *myrinet* foram utilizadas as versões openmpi_mx/1.8.4 para ambos os compiladores. No entanto, após executados os testes podemos observar que os valores obtidos eram piores que a versão *ethernet* do compilador GNU. Uma das principais causas poderia ser a versão mais desatualizada dos módulos da *myrinet*, pelo que decidimos utilizar a versão openmpi_eth/1.8.4 para o compilador da GNU na versão *ethernet*.

Após novas medições, a versão openmpi_eth/1.8.4 não apresenta melhores resultados que a versão myrinet. No entanto a diferença é pouco significativa, por isso achamos que o problema pode não ter ficado completamente resolvido.

Para além dos módulos de MPI também é necessário importar o módulo gcc/5.3.0 para o compilador da GNU e o módulo intel/2019 para o compilador da Intel, tanto na versão *ethernet* como na *myrinet*.

Os testes realizados foram corridos com 2, 4, 8 e 16 processos para os nodos 641 e 2, 4, 8, 12 processos para os nodos 431. O número de *threads* utilizado varia consoante o número de processos, de forma a utilizar todos os recursos das máquinas. Para cada uma das execuções, este valor era calculado através da divisão do número máximo de *cores lógicos* pelo número de processos na mesma máquina, por exemplo, para os nodos 641 (com 32 *cores* lógicos) e 4 processos totais (2 por máquina), vão ser utilizados 16 *threads* por máquina.

Foram utilizadas as *flags* -report-bindings e I_MPI_DEBUG 4 para verificar se o mapeamento estava de acordo com as expectativas.

VI. Utilitários de Monitorização

Para processar os dados das ferramentas foram desenvolvidos vários scripts em Python que agregam as estatísticas da execução do programa. Para além disso, os scripts foram utilizados para visualizar graficamente valores como médias, máximos, e variações ao longo do tempo. Apesar de ser possível visualizar graficamente todas as métricas recolhidas, nas subsecções seguintes apenas são apresentadas as que consideramos mais pertinentes.

Devido ao tamanho reduzido da classe A, apenas foi possível realizar medições para a versão sequencial. Tanto na versão OMP como na MPI, o tempo de execução é de tal maneira reduzido que torna irrelevante qualquer valor obtido com as ferramentas utilizadas. Todos os testes nesta fase utilizaram o compilador da Intel com as flags de optimização mencionadas, e foram executados no nodo 641.

A. CPU

Para obter estatísticas de utilização do CPU foi utilizado o comando mpstat -P ALL 1. A flag -P ALL permite a obtenção de informação sobre todos os cores lógicos da máquina. [https://linux.die.net/man/1/mpstat]

No caso dos benchmarks serial, apenas um core é utilizado e a sua utilização é sempre próxima dos 100%. usr, independentemente da classe do problema. Verificou-se que ocasionalmente a execução migra entre cores.

No caso das versões OMP e híbrida, as classes maiores correspondem a alguns aumentos da utilização global dos cores

Os gráficos seguintes correspondem à média da percentagem de utilização correspondente a aplicações *user level*.

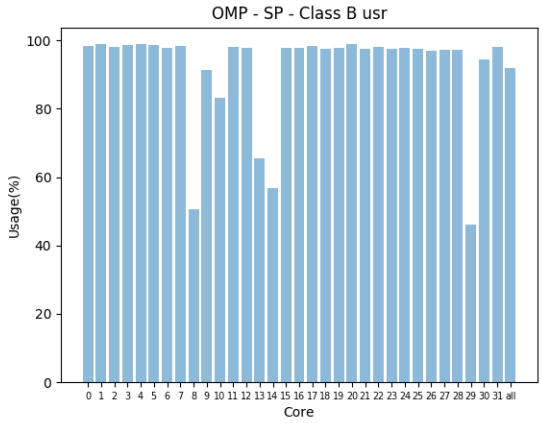

Fig. 8. Utilização dos cores lógicos pelo benchmark SP-MZ da versão OMP da classe B

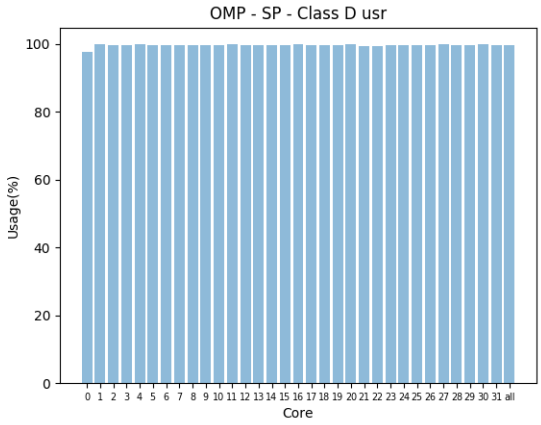

Fig. 10. Utilização dos cores lógicos pelo benchmark SP-MZ da versão OMP da classe D

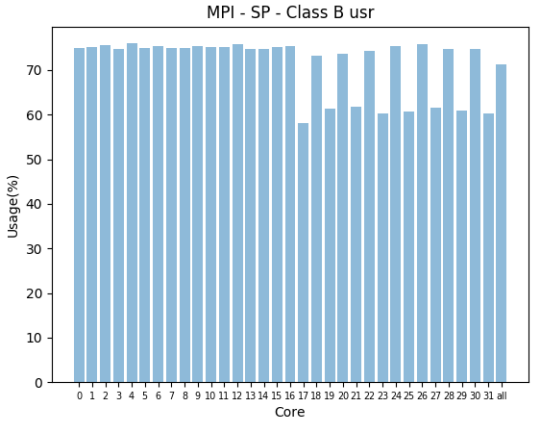

Fig. 11. Utilização dos cores lógicos pelo benchmark SP-MZ da versão MPI+OMP da classe B

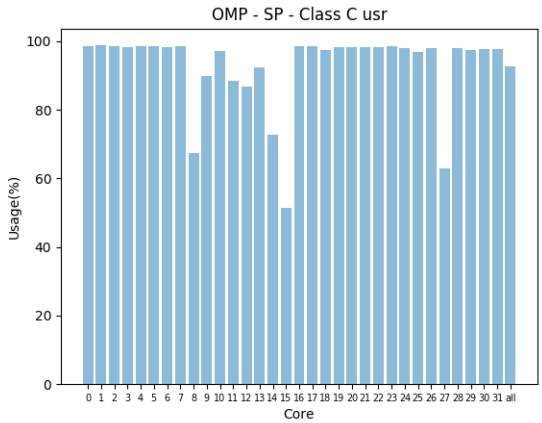

Fig. 9. Utilização dos cores lógicos pelo benchmark SP-MZ da versão OMP da classe C

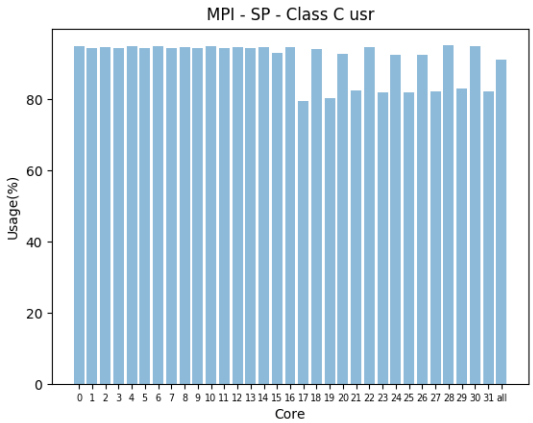

Fig. 12. Utilização dos cores lógicos pelo benchmark SP-MZ da versão MPI+OMP da classe C

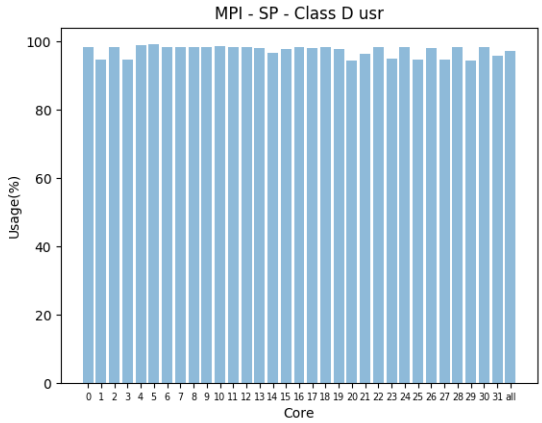

Fig. 13. Utilização dos cores lógicos pelo benchmark SP-MZ da versão MPI+OMP da classe D

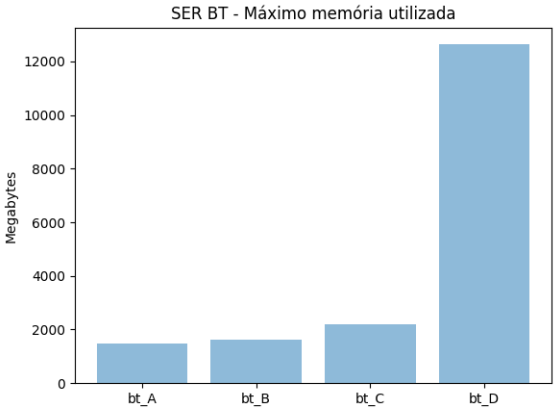

Fig. 14. Memória utilizada pela versão SER do benchmark BT-MZ para as diferentes classes

B. *Memória*

As medições foram feitas através do uso da ferramenta free. [http://man7.org/linux/man-pages/man1/free.1.html]

O comando específico é free -m -s 1, onde a flag -m converte os valores para MBytes e a flag -s 1 leva a que as medições sejam feitas com intervalos de 1 segundo.

Também foram efetuadas medições com a ferramenta vmstat e desenvolvido o script correspondente. No entanto, demos maior preferência à ferramenta free, por acharmos as suas métricas mais úteis e simples. Um aspeto interessante da ferramenta vmstat é observar a utilização da swap. Porém, não foi necessário o uso desta em nenhum dos testes efetuados. Foi efetuada uma tentativa de usar a *swap* num nodo com com apenas 8GB mas sem sucesso.

No caso da memória, consideramos mais importante medir o valor máximo do que a média, uma vez que este valor nos vai permitir determinar se a máquina onde esta a ser testado consegue correr o programa sem problemas. Por exemplo, um *benchmark* que tenha utilização média de 5gb de memória facilmente corre numa máquina com 10gb de memória, no entanto, se ao longo da execução por algum motivo existir algum momento em que o programa necessite de 15gb a máquina não tem capacidade para executar o mesmo levando a que o processo seja cancelado ou à utilização de swap.

As medições para os diferentes tipos de *benchmark*, SP-MZ e BT-MZ, apresentam resultados muito próximos pelo que apenas são apresentados gráficos referentes apenas a um dos *benchmarks*, neste caso o BT-MZ.

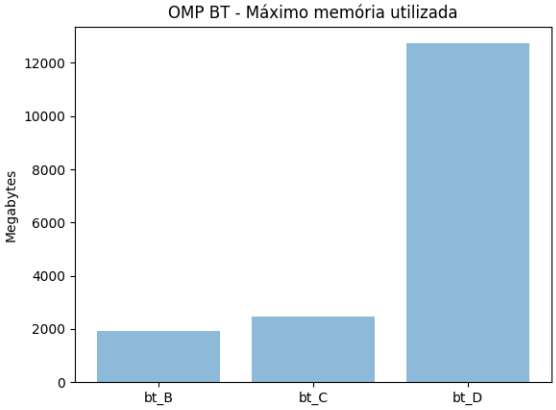

Fig. 15. Memória utilizada pela versão OMP do benchmark BT-MZ para as diferentes classes

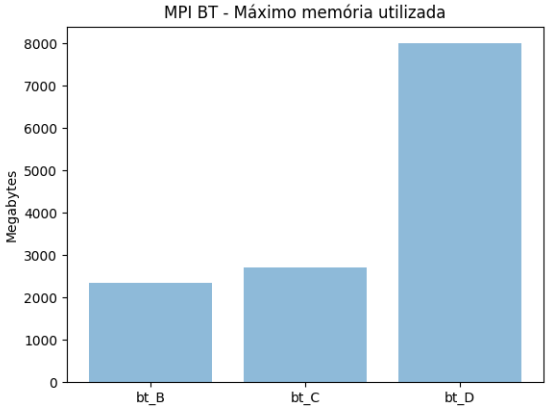

Fig. 16. Memória utilizada pela versão MPI do benchmark BT-MZ para as diferentes classes

Os valores máximos de utilização da memória são próximos aos mencionados na documentação.

Problem Class	Memory Requirement (approx.)
Class S	1 MB
Class W	6 MB
Class A	50 MB
Class B	200 MB
Class C	0.8 GB
Class D	12.8 GB
Class E	250 GB
Class F	5.0 TB

Fig. 17. Valores aproximados indicados pela documentação oficial

Os valores obtidos para as versões SER e OMP são aproximadamente iguais entre si, e a versão MPI é cerca de metade das duas, o que faz sentido tendo em conta que está a ser executado em duas máquinas.

C. Disco

Os resultados do mpstat indicam que o tempo que o cpu espera pelo disco não é significativo. No entanto, é sempre útil desenvolver métodos de analisar o desempenho do disco, pois este pode ser um fator limitante noutras aplicações.

Para medir a utilização do disco foi utilizado o comando iostat -mdx 1. De notar o uso da flag -x que permite o acesso a estatísticas mais detalhadas. [https://linux.die.net/man/1/iostat]

Achamos o impacto no desempenho de muitos dos valores absolutos fornecidos pelo iostat difícil de avaliar, principalmente devido a uma falta de referência de valores normais para o *hardware* utilizado. Como tal focámo-nos na métrica %util que, segundo a documentação, permite ter uma da saturação do disco.

Os gráficos seguintes representam a variação desta métrica nos primeiros 50 segundos de execução.

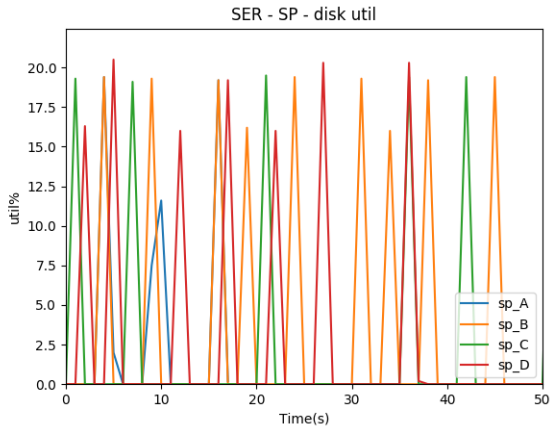

Fig. 18. Utilização do disco pelo benchmark SP-MZ para as diferentes classes da versão SER

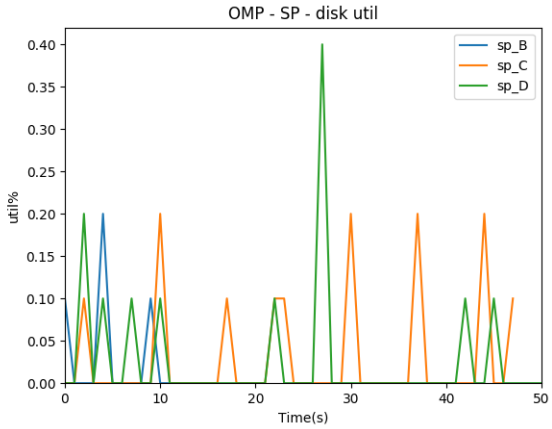

Fig. 19. Utilização do disco pelo benchmark SP-MZ para as diferentes classes da versão OMP

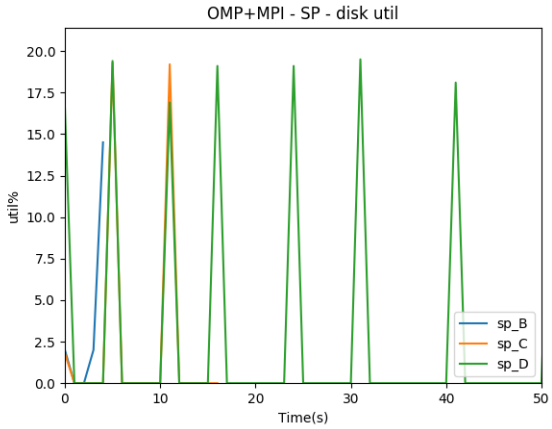

Fig. 20. Utilização do disco pelo benchmark SP-MZ para as diferentes classes da versão MPI+OMP

Os resultados da versão OMP apresentam picos de utilização significativamente menores que as restantes versões.

Como %util indica apenas a percentagem de tempo em que existe algum pedido I/O a ser servido e não a quantidade destes, é possível que várias *threads* executem pedidos paralelamente num curto espaço de tempo. [https://brooker.co.za/blog/2014/07/04/iostat-pct.html]

De qualquer forma, os resultados indicam que o disco não é um gargalo de desempenho dos *benchmarks* testados.

D. Rede

As medições sobre a rede foram feitas através do comando sar -n TCP,ETCP,DEV 1. As *flags* utilizadas disponibilizam estatísticas sobre a rede. [https://linux.die.net/man/1/sar]

O seguinte gráfico representa a quantidade de informação em kilobytes enviada através da interface *eth0* para a segunda máquina.

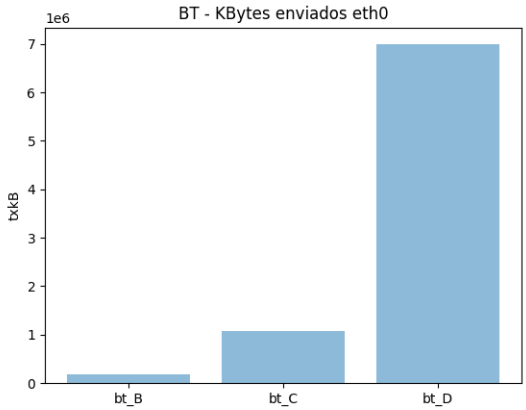

Fig. 21. Quantidade de dados enviados para a segunda máquina

O total de dados transmitidos corresponde a aproximadamente 120% da metade dos dados da classe. O excedente pode corresponder a custos de comunicação, tais como cabeçalhos, e a transmissões de informação não relacionadas com o *benchmark*.

VII. Trabalho futuro

Na maioria dos casos, o compilador da Intel apresentou melhores resultados. Seria interessante correr os *benchmarks* num processador não Intel e verificar se as diferenças de desempenho dos compiladores sofriam alterações.

Também seria interessante testar *benchmarks* que fossem *memory-bound* ou *I/O bound*, assim como *benchmarks* em C e verificar se a utilização dos diferentes compiladores e *flags* têm resultados comparáveis.

ESC-TP2

José Pinto A81317 - Pedro Barbosa A82068

I. Introdução

O Dtrace é uma ferramenta que permite a monitorização e a análise de performance não só de programas em específico mas também do sistema operativo em si.

Como com qualquer ferramenta, de forma a utilizar o Dtrace para analisar um programa, é necessário primeiro adquirir conhecimentos sobre o seu funcionamento e as suas capacidades. A melhor forma de adquirir esses conhecimentos é através de experiência prática.

Os exercícios abordam a invocação de chamadas de sistema, especialmente a abertura de ficheiros.

A resolução das perguntas é orientada ao Solaris 11. A utilização noutros sistemas requer algumas modificações.

II. Pergunta 1

Para obter informação sobre a abertura de ficheiros, são utilizadas as *probes* relativas à *system call openat*.

O nome do executável, o PID do processo, o UID do utilizador e o GID do grupo podem ser obtidos através das respetivas variáveis *built-in*.

```
syscall::openat:entry
{
    self->execname = execname;
    self->pid = pid;
    self->uid = uid;
    self->gid = gid;
    self->path = copyinstr(arg1);
}
```

Segundo a documentação da chamada *openat*, o caminho do ficheiro a abrir corresponde ao segundo argumento, que na sintaxe do dtrace corresponde a *arg1*. Como o *arg1* corresponde a um apontador da memória, é necessário invocar *copyinstr* para obter o conteúdo da string.

O inteiro *arg2* corresponde às *flags* com que o ficheiro é aberto. É possível determinar se uma *flag* foi utilizada observando os bits correspondentes. Para isso é utilizado o operador "&" que permite fazer *match* entre os bits da *flag* utilizada e os bits de uma das possíveis *flags*.

O argumento tem que incluir obrigatoriamente umas das *flags* do modo de abertura, "O_RDONLY", "O_WRONLY" ou "O_RDWR"

```
syscall::openat:entry
/(arg2 & 1) == 0 && (arg2 & 2) == 0/
{
    self->flags = "O_RDONLY";
}

syscall::openat:entry
/(arg2 & 1) == 1 && (arg2 & 2) == 0/
{
    self->flags = "O_WRONLY";
}

syscall::openat:entry
/(arg2 & 2) == 2/
{
    self->flags = "O_RDWR";
}
```

Existem diversas *flags* opcionais. Dentro destas, o *script* deteta a utilização de "O_APPEND" e de " O_CREAT".

```
syscall::openat:entry
/(arg2 & 8) == 8/
{
    self->flags = strjoin(self->flags,
       ", O_APPEND");
}

syscall::openat:entry
/(arg2 & 256) == 256/
{
    self->flags = strjoin(self->flags,
       ", O_CREAT");
}
```

Caso uma destas *flags* opcionais seja utilizada é necessário fazer *strjoin* para adicionar a *flag* à *flag* obrigatória.

O valor de retorno da chamada é dado pelo *arg0* da *probe syscall::openat:return*. Também é utilizado o disparo dessa *probe* para imprimir a informação reunida.

```
syscall::openat:return
{
printf("Nome do executavel: %s \nPID
   do processo: %d \nUID do grupo: %d
   \nGID do grupo: %d \nPath:
   %s\nFlags:%s \nValor de retorno: %d
   \n",
    self->execname,
    self->pid,
    self->uid,
    self->gid,
    self->path,
    self->flags,
    arg0
    );
}
```

Para apenas detetar ficheiros com *etc/* no caminho basta adicionar a seguinte condição na última *probe*, sendo *strstr* uma função que retorna a localização da segunda string na primeira.

```
strstr(self->path, "etc/") != NULL
```

De seguida apresenta-se o output relevante da execução do comando "cat /etc/inittab >/tmp/test", sendo possível observar uma versão detalhada nos anexos:

```
Nome do executavel: bash
PID do processo: 22829
UID do grupo: 1015
GID do grupo: 5000
Path: /tmp/teste
Flags:O_WRONLY, O_CREAT
Valor de retorno: 4
--------------------------------------
Nome do executavel: cat
PID do processo: 22829
UID do grupo: 1015
GID do grupo: 5000
Path: /etc/inittab
Flags:O_RDONLY
Valor de retorno: 3
```

Output relevante da execução do comando "cat /etc/inittab ≫ /tmp/test":

```
Nome do executavel: bash
PID do processo: 22973
UID do grupo: 1015
GID do grupo: 5000
Path: /tmp/teste
Flags:O_WRONLY, O_APPEND, O_CREAT
Valor de retorno: 4
--------------------------------------
Nome do executavel: cat
PID do processo: 22973
UID do grupo: 1015
GID do grupo: 5000
Path: /etc/inittab
Flags:O_RDONLY
Valor de retorno: 3
```

Output relevante da execução do comando "cat /etc/inittab | tee /tmp/test":

```
Nome do executavel: cat
PID do processo: 23027
UID do grupo: 1015
GID do grupo: 5000
Path: /etc/inittab
Flags:O_RDONLY
Valor de retorno: 3
--------------------------------------
Nome do executavel: tee
PID do processo: 23028
UID do grupo: 1015
GID do grupo: 5000
Path: /tmp/teste
Flags:O_WRONLY, O_CREAT
Valor de retorno: 3
```

Output relevante da execução do comando "cat /etc/inittab | tee -a /tmp/test":

```
Nome do executavel: cat
PID do processo: 23066
UID do grupo: 1015
GID do grupo: 5000
Path: /etc/inittab
Flags:O_RDONLY
Valor de retorno: 3
--------------------------------------
Nome do executavel: tee
PID do processo: 23067
UID do grupo: 1015
GID do grupo: 5000
Path: /tmp/teste
Flags:O_WRONLY, O_APPEND, O_CREAT
Valor de retorno: 3
```

III. Pergunta 2

A. Alínea A

```
BEGIN{
    APPEND = 8;
    CREAT = 256;
}

syscall::openat:entry
{
    @attempts[execname, pid, "opens"]
       = count();


    creation_flags = "";

    ((arg2 & CREAT) != CREAT) ?
        creation_flags = "open
           existing file" : 0;

    ((arg2 & CREAT) == CREAT) ?
        creation_flags = "create new
           file" : 0;

    @attempts[execname, pid,
       creation_flags] = count();
}

syscall::openat:return
/arg0 >= 0/
{
    @attempts[execname, pid,
       "successful"] = count();
}
```

À semelhança da primeira pergunta, o segundo argumento (*arg2*) representa as *flags* com que o ficheiro é aberto, podendo ser utilizado o mesmo método para determinar qual das *flags* é utilizada.

A agregação *attempts* é utilizada para contabilizar a informação sobre as aberturas dos ficheiros. A chave "opens" corresponde às tentativas de abrir um ficheiro.

Se a flags O_CREAT for usada, é registada uma tentativa de criar um ficheiro novo. Caso contrário, é registada uma tentativa de abrir um ficheiro existente.

Se a *probe return* disparar e o valor de retorno seja não negativo então a tentativa é considerada bem sucedida.

B. Alínea B

```
tick-$1s
{
    printf("\%Y\n", walltimestamp);
    printa(@attempts);
}
```

A proposta da alínea B era imprimir os resultados obtidos anteriormente de X em X segundos, período de tempo indicado pelo utilizador, acompanhado da data e hora atual. Para isso, foi adicionada uma nova *probe tick* que contém dois *prints*, um para a data e hora atual (foi utilizado o walltimestamp para ir buscar esta informação) e outro para imprimir a agregação. O $1 corresponde ao intervalo de tempo introduzido pelo utilizador.

Resultado de uma iteração:

```
2020 May 21 22:36:39
```

execname	pid	action	amount
vmtoolsd	1212	create new file	1
dtrace	14118	open existing file	2
dtrace	14118	opens	2
dtrace	14118	successful	2
sshd	708	open existing file	3
sshd	708	opens	3
sshd	708	successful	3
sshd	14121	open existing file	3
sshd	14121	opens	3
sshd	14121	successful	3
sstored	905	open existing file	3
sstored	905	opens	3
sstored	905	successful	3
sshd	14119	successful	50
sshd	14119	open existing file	56
sshd	14119	opens	56
vmtoolsd	1212	successful	328
vmtoolsd	1212	open existing file	337
vmtoolsd	1212	opens	338

IV. Pergunta 3

```
syscall:::entry
/pid == $target/
{
    self->ts = timestamp;
    @totalCount[probefunc] = count();
}

syscall:::return
/pid == $target/
{
    @totalTime[probefunc] =
       sum(timestamp - self->ts);
}

END
{
    printa(@totalTime, @totalCount);
}
```

Inicialmente, quando é invocada uma *system call*, é registada na variável *self->ts* o valor do tempo atual. Para além disso também é atualizada a agregação *totalCount* onde vão ser guardadas o número de execuções de uma *system call*. Quando a *system call* termina, calcula-se o tempo de execução e soma-se esse valor ao correspondente contido na agregação *totalTime*.

No final do programa, imprimem-se ambas as agregações.

Execname	Total time	Total count
rexit	0	1
sysconfig	5870	1
getpid	7241	1
lwp_private	7580	1
sigpending	9380	1
getrlimit	14797	1
resolvepath	16856	1
setcontext	22250	2
close	26242	3
memcntl	28438	2
write	31943	1
getdents	33072	2
ioctl	43309	3
mmapobj	43517	1
brk	47218	5
fstatat	53353	5
openat	143704	6
mmap	317398	2

V. Conclusão

Como a programação do Dtrace não corresponde na totalidade aos paradigmas que estamos mais habituados, a resolução dos exercícios foi essencial para obter uma maior familiaridade com o uso da ferramenta.

VI. Tabela da pergunta 1

Execname	PID	UID	GID	Path	Flags	Return value
bash	22829	1015	5000	/tmp/teste	O_WRONLY, O_CREAT	4
cat	22829	1015	5000	/var/ld/64/ld.config	O_RDONLY	-1
cat	22829	1015	5000	/lib/64/libc.so.1	O_RDONLY	3
cat	22829	1015	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8	O_RDONLY	4
cat	22829	1015	5000	/usr/lib/locale/common/amd64/methods_unicode.so.3	O_RDONLY	-1
cat	22829	1015	5000	/usr/lib/locale/pt_PT.UTF-8/pt_PT.UTF-8	O_RDONLY	-1
cat	22829	1015	5000	/usr/lib/locale/en_US.UTF-8/LC_MESSAGES/solaris_linkers.mo	O_RDONLY	-1
cat	22829	1015	5000	/usr/lib/locale/en_US.UTF-8/LC_MESSAGES/solaris_lib_libc.mo	O_RDONLY	3
cat	22829	1015	5000	/etc/inittab	O_RDONLY	3

TABLE I

Output para o comando cat /etc/inittab >/tmp/test

ESC-TP3

José Pinto A81317 - Pedro Barbosa A82068

I. Introdução

A optimização de programas não requer apenas minimizar o número de operações que um programa efetua. É necessário perceber como essas operações afetam o sistema operativo e os recursos de *hardware*.

O Dtrace é uma ferramenta poderosa neste aspeto pois permite monitorizar uma vasta gama de eventos com um *overhead* mínimo. Para além disso também oferece formas de processar os resultados obtidos. Para além de permitir efetuar todos os testes realizados na cadeira de Paradigmas de Computação Paralela (PCP), o Dtrace permite realizar novas medições mais detalhadas que não foram possíveis no semestre anterior.

O Dtrace é aplicado a diferentes implementações do algoritmo de equalização do histograma. Uma implementação sequencial, três em memória partilhada (OpenMP, Pthreads e C++ 11) e uma em memória distribuída (MPI). Das versões paralelas, são estudadas em mais detalhe as implementações em memória partilhada. A versão MPI é demasiado limitada pelos custos de comunicação entre processos, tornando a análise de outros aspetos menos relevante e interessante.

II. Caracterização do ambiente de testes

Para obter informação sobre a máquina Solaris foi utilizado o seguinte comando:

```
psrinfo -pv
```

```
The physical processor has 4 virtual
   processors (0-3)
  x86 (AuthenticAMD 600F12 family 21
     model 1 step 2 clock 2600 MHz)
        AMD Opteron(tm) Processor 6282
           SE
The physical processor has 4 virtual
   processors (4-7)
  x86 (AuthenticAMD 600F12 family 21
     model 1 step 2 clock 2600 MHz)
        AMD Opteron(tm) Processor 6282
           SE
```

O resultado indica a existência de 8 cores virtuais. Por esta razão, os testes multi-threaded foram efetuados com um máximo de 8 threads, e os testes MPI com um máximo de 8 processos

III. Descrição do algoritmo

O algoritmo desenvolvido na cadeira de Paradigmas de Computação Paralela consistia na equalização do histograma de uma imagem. A imagem está no formato PGM, que consiste num array de cores que variam entre 0 e *maxGrey* (número definido no inicio do ficheiro). O foco nessa cadeira foi desenvolver e paralelizar a etapa da equalização do histograma em si, sendo a leitura e a escrita da imagem realizadas com o uso de uma biblioteca. Como tal, a análise com o Dtrace também irá ter como foco a etapa da equalização.

O primeiro passo da equalização é construir o histograma da imagem. Isto consiste em contar a frequência absoluta de cada cor. De seguida calculam-se as frequências acumuladas. Com as frequências acumuladas normaliza-se o histograma. Cada cor da imagem é então substituída pelo valor respetivo no histograma normalizado.

IV. Algoritmo sequencial

A implementação sequencial é relativamente simples e consiste em 3 ciclos. No primeiro ciclo é construído o histograma da imagem. Em cada iteração do segundo ciclo é calculada a frequência acumulada de uma cor, é normalizado o valor e este é colocado num novo histograma. No último ciclo são mapeadas as novas cores da imagem.

O Dtrace permite a definição e inserção de *probes* personalizadas. Foram definidas várias *probes* para delimitar cada secção do programa.

Estas *probes* podem ser utlizadas para várias razões, como por exemplo, medir o tempo que cada fase demora ou para limitar o disparo de outras *probes* para apenas uma região específica do código.

```
provider histogram_equalization {

   probe equalization__start(int size);
   probe computing__done(int size);
   probe normalizing__done(int size);
   probe mapping__done(int size);

   probe after__read(int size);
   probe before__write(int size);
};
```

É necessário definir no código fonte o local onde cada *probe* é disparada.

```
if(HISTOGRAM_EQUALIZATION_EQUALIZATION_START_ENABLED()){
    HISTOGRAM_EQUALIZATION_EQUALIZATION_START(size);
}

for(int i=0; i < size; i++) {
        histogram[data[i]]++;
```

```
}

if(HISTOGRAM_EQUALIZATION_COMPUTING_DONE_ENABLED()){
   HISTOGRAM_EQUALIZATION_COMPUTING_DONE(size);
}

int cumulativeDistribution = 0;
double constant = (double)maxgray / size;

for(int i=0; i <= maxgray; i++) {
        cumulativeDistribution += histogram[i];
        normalized[i] = round(cumulativeDistribution * constant);
}

if(HISTOGRAM_EQUALIZATION_NORMALIZING_DONE_ENABLED()){
    HISTOGRAM_EQUALIZATION_NORMALIZING_DONE(size);
}

for (int i = 0; i < size; i++) {
        data[i] = normalized[data[i]];
}

if(HISTOGRAM_EQUALIZATION_MAPPING_DONE_ENABLED()){
    HISTOGRAM_EQUALIZATION_MAPPING_DONE(size);
}
```

Ao usar as *probes* para medir o tempo de cada fase (phases.d) é possível determinar que zonas têm um maior impacto na *performance*.

Tempo(ms)	Computing	Normalizing	Mapping	Total
10 MB	12,31(52%)	0,06(∼0%)	11,37(48%)	23,74
50 MB	60,47(50%)	0,39(∼0%)	59,06(49%)	119,91
100 MB	122,49(50%)	0,34(∼0%)	121,79(50%)	244,63

TABLE I: Tempos (em milissegundos) e percentagens das respetivas fases de execução

V. Algoritmo em memória partilhada

A. Descrição do algoritmo paralelo

Assim como na versão sequencial, o algoritmo paralelo vai executar as mesmas operações que estão presentes nos 3 ciclos. No entanto, de forma a permitir o funcionamento de forma paralela foram feitas algumas alterações.

- 1ª fase - Os dados da imagem são divididos de forma igual para cada *thread* e cada uma calcula o histograma correspondente aos seus dados. Depois de calcular o histograma, cada *thread* adiciona o resultado a um histograma partilhado. Esta última etapa deve ser efetuada de forma síncrona de forma a evitar *data races*. Apesar de ter um ciclo extra, este não será muito significante quando comparado ao ganhos de paralelizar o ciclo que percorre a informação da imagem.
- 2ª fase - Na versão sequencial cada iteração dependia da anterior, devido ao cálculo da frequência acumulada. Em vez de ser usada uma variável escalar, as várias frequências acumuladas podem ser guardadas num pequeno *array*. Apesar do cálculo das frequências permanecer sequencial, a normalização dos valores, que é computacionalmente mais intensiva, é paralelizada.
  Em termos de desempenho o impacto desta fase é insignificante, independentemente do tamanho da imagem, o que não a torna um alvo apelativo para analisar em mais detalhe.
- 3ª fase - Nesta parte, como não há dependência de dados, não é preciso realizar nenhuma alteração significativa para ser possível paralelizar o ciclo.

B. Implementação OpenMP

A primeira fase em OpenMP é relativamente simples. É utilizada a diretiva *for* e um *array* local a cada *thread*. É utilizado escalonamento estático para distribuir o trabalho de forma uniforme pelas diferentes *threads*.

```
#pragma omp for schedule(static,chunck)
for(int i=0; i < size; i++) {
  threadHistogram[ data[i] ]++;
}
```

De seguida é utilizada a diretiva *critical* para garantir que a junção dos histogramas é feita sem a ocorrência de *data races*.

```
#pragma omp critical
for(int i=0; i <= maxgray; i++) {
  histogram[i] += threadHistogram[i];
}
```

Tal como mencionado, na segunda fase é necessário calcular sequencialmente o *array* das frequências acumuladas. Para isso basta colocar o código fora da zona paralela.

```
for(int i=0; i <= maxgray; i++) {
  cumulativeDist += histogram[i];
  cumulativeDistArray[i] =
     cumulativeDist;
}
```

Uma vez calculado o *array* das frequências acumuladas, a normalização do histograma pode ser distribuída pelas *threads*.

```
#pragma omp for schedule(static,chunck)
for(int i=0; i <= maxgray; i++) {
  normalized[i] =
     round(cumulativeDistArray[i] *
     constant);
}
```

Para paralelizar a terceira fase é somente necessário adicionar a diretiva *for*.

```
#pragma omp for schedule(static,chunck)
for(int i = 0; i < size; i++) {
  data[i] = normalized[data[i]];
}
```

C. Pthreads

Para utilizar *threads* através da biblioteca Pthreads é necessário definir a função que as *threads* irão executar.

```
void* thread_func(void* thread_id){
    calculate_histogram(thread_id);
    pthread_barrier_wait(&barrier);

    cumulative_distribution(thread_id);
    pthread_barrier_wait(&barrier);

    map_new_values(thread_id);
    pthread_barrier_wait(&barrier);
}
```

Esta função invoca uma função para cada fase. Entre estas funções é necessária a utilização de barreiras para sincronizar as *threads*, algo que era feito implicitamente na versão OpenMP.

Na primeira fase a diretiva *for* do OpenMP divide automaticamente o ciclo pelas *threads*. Na implementação Pthreads (função *calculate_histogram*) isto é efectuado manualmente.

```
int thread_chunck_size = size /
   thread_count;
int start = chunck_size*thread_id;
int end = start + chunck_size;

if(thread_id == thread_count-1){
  end = size;
}

for(int i = start; i < end; i++) {
  threadHistogram[ data[i] ]++;
}
```

A diretiva *critical* necessita de ser substituída pelo uso explícito de um *mutex*.

```
pthread_mutex_lock(&mutex);
for(int i=0; i <= maxgray; i++) {
  histogram[i] += threadHistogram[i];
}
pthread_mutex_unlock(&mutex);
```

Na função *cumulative_distribution*, é necessário garantir que o *array* é calculado sequencialmente por uma única *thread*. Neste exemplo a secção sequencial é atribuída à *thread* com o ID zero. Também é necessário colocar uma barreira no final da secção sequencial para garantir que nenhuma *thread* se adianta na execução do programa.

```
if(thread_id == 0){
  for(int i=0; i <= maxgray; i++) {
    cumulativeDist += histogram[i];
    cumulativeDistArray[i] =
       cumulativeDist;
  }
}

pthread_barrier_wait(&barrier);
```

Na função *map_new_values* a divisão do ciclo é feita da mesma forma que na função *calculate_histogram*, calculando as variáveis *start* e *end*.

D. C++ 11

A implementação em C++ 11 é bastante semelhante à em Pthreads em termos de estrutura, sendo também necessário definir uma função para ser executada pelas *threads*.

A sintaxe da versão C++ 11 é menos restrita sendo possível indicar um número variado de argumentos à função a ser executada, não sendo necessário o uso de variáveis globais ou a definição de estruturas de dados.

```
void* thread_func(long thread_id, int
   size, ..., Barrier& b){

  calculate_histogram(thread_id, size,
     ..., data);
  b.wait();

  cumulative_distribution(thread_id,
     size, ..., std::ref(b));
  b.wait();

  map_new_values(thread_id, size,
     t..., data);
  b.wait();
}
```

Ao contrário do Pthreads, não é uma fornecida uma implementação de barreiras. Foi então desenvolvida uma implementação recorrendo às variáveis condicionais. Cada *thread* que chegue à barreira incrementa um contador. Caso este seja inferior ao número total de *threads* a *thread* adormece. Quando o contador atingir o total de *threads*, as adormecidas são acordadas.

```
class Barrier {

public:
    Barrier(int thread_count);
```

```
    void wait(void);

private:
    std::mutex counterMutex;
    std::condition_variable cond;

    int thread_count;
    int current;
};

void Barrier::wait(void){
    std::unique_lock<std::mutex>
       lock(counterMutex);
    current++;

    if(current < thread_count){
            cond.wait(lock);
    }
    else{
            cond.notify_all();
            current = 0;
    }
}
```

Na função *calculate_histogram*, em vez de os *mutexes* serem utilizados diretamente, é utilizado um *lock_guard* para o adquirir. Desta forma quando a função termina o *mutex* é libertado implicitamente.

VI. Medições em memória partilhada

Para contextualizar a exploração do programa com o Dtrace, é necessário medir o desempenho das diferentes implementações.

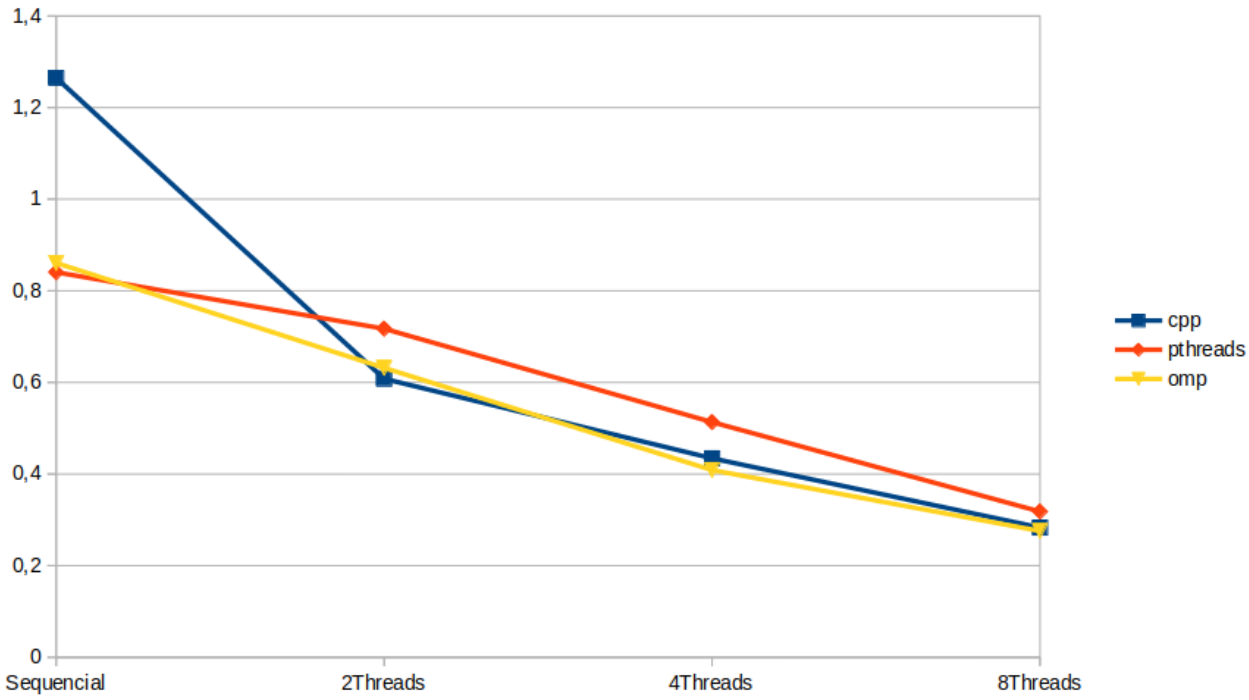

Fig. 1: Tempos de execução para as diferentes implementações

Como é possível observar pelo gráfico, independentemente da implementação, a escalabilidade do algoritmo fica um pouco aquém do ideal.

Diferenças entre as execuções sequenciais do OMP/Pthreads e do C++ podem-se dever ao facto de serem usados compiladores diferentes, uma vez que o código é o mesmo.

Foi determinado durante a cadeira de PCP do semestre anterior que o algoritmo é *memory bound*, o que limita a sua escalabilidade.

Mesmo sabendo previamente qual o gargalo de desempenho do programa, o Dtrace é na mesma utilizado para analisar diferentes aspectos que tendem a prejudicar o desempenho de aplicações paralelas, como o a sincronização e o mau balanceamento da carga.

A. *Performance Counters*

Nos computadores modernos uma das principais limitações é a velocidade de acesso à memória. Para minimizar este problema é importante obter a melhor *performance* possível da cache. Isto implica reduzir o máximo possível o número de *misses* na cache.

O *provider* cpc disponibiliza acesso a vários contadores. O comando ”cpustat -h” imprime os vários eventos suportados pela máquina. Entre estes existem alguns que são equivalentes aos eventos disponibilizados pelo PAPI. A listagem das *probes*

do cpc com o comando "dtrace -l -P cpc" indica a existência *probes* que suportam os eventos PAPI.

Destes eventos foram escolhidos os que contabilizam o número de misses nas *data caches* L1 e L2 e o número de instruções executadas.

```
cpc:::PAPI_l1_dcm-all-5000
/pid == $target && equalizing/
{
    @[ufunc(arg1)] = count();
}
```

A variável *equalizing* é atualizada por *probes* pid para que as medições sejam relativas à secção da equalização e não à leitura e escrita da imagem.

A tabela seguinte representa as medições dos eventos obtidas. Cada unidade da tabela corresponde a 5000 ocorrências do evento. Os resultados apresentados correspondem a uma execução representativa (tempo de execução próximo da média).

		Instructions	Misses L1	Misses L2
OMP	**Sequential**	214468	45	14
	2 Threads	216122	50	20
	4 Threads	216276	86	59
	8 Threads	216519	160	92
Pthreads	**Sequential**	214504	49	15
	2 Threads	274829	95	18
	4 Threads	274892	131	71
	8 Threads	275091	180	86
C++	**Sequential**	216221	65	15
	2 Threads	215759	71	23
	4 Threads	215711	119	75
	8 Threads	216159	136	87

TABLE II: Número de instruções e *misses* na *cache* L1/L2 para as diferentes implementações

Como seria de esperar, o número de *misses* aumenta consoante o aumento do número de *threads*. Um número elevado de *threads* a competir pelos mesmos recursos da *cache* leva a uma utilização da *cache* menos eficiente.

A implementação em Pthreads origina um número mais elevado de instruções e *misses* o que corresponde ao seu pior desempenho.

B. Sincronização

No paradigma de memória partilhada, várias *threads* concorrentes podem partilhar os mesmos dados. Para garantir o funcionamento correto do programa é necessário evitar *data races* através de mecanismos de sincronização, como os *mutexes*. No entanto, é necessário ter em consideração que os mecanismos de sincronização resultam num custo adicional. Por essa razão é sempre importante verificar o seu impacto no desempenho e se é possível mitigá-lo.

Nas diferentes implementações existe alguma sincronização entre *threads*. Na implementação OpenMP existe sincronização na região *critical* e no final de cada região *parallel*. Nas outras duas implementações a sincronização é mais explícita, correspondendo à utilização de *mutexes* e barreiras.

Na cadeira de PCP, tentativas de medir o impacto da sincronização consistiram apenas em medir o tempo imediatamente antes e imediatamente depois de entrar na região *critical*, o que não era muito prático nem muito preciso.

Uma forma melhor de medir os custos da sincronização é utilizar o *provider* plockstat do Dtrace. O plockstat disponibiliza vários eventos sobre os *mutexes*, como o *mutex-spin*, o *mutex-block* e o *mutex-acquire*.

Para ser possível efetuar medições correctamente, é importante compreender os detalhes do funcionamento dos *mutexes*, principalmente na implementação OpenMP, onde estes são utilizados de forma algo transparente.

Para tal, observamos as ocorrências dos seguintes eventos:

- Entrar em *spinning*
- Transitar de *spinning* para *acquired*
- Transitar de *spinning* para *blocked*
- Transitar de *blocked* para *acquired*
- Adquirir imediatamente o *lock*

Também foram medidos os tempos despendidos nas seguintes fases

- Entre *spinning* e *acquired*
- Entre *spinning* e *blocked*
- Entre *blocked* e *acquired*

(Ficheiro plockstat_all_stats.d:)

```
plockstat$target:::mutex-spin
/pid == $target && equalizing/
{
    @occurrences["spin"] = count();
    self->spinStart = timestamp;
}


plockstat$target:::mutex-acquire
/pid == $target && self->spinStart && equalizing/
{
    @occurrences["spin->acquire"] = count();
    @average["average time(ns): spin->acquire"] = avg(timestamp -
       self->spinStart);
    @hist["time(ns): spin->acquire"] = quantize(timestamp - self->spinStart);
```

```
    self->spinStart = 0;
}

plockstat$target:::mutex-block
/pid == $target && self->spinStart && equalizing/
{
    self->blockStart = timestamp;
    @occurrences["spin->block"] = count();
    @average["average time(ns): spin->block"] = avg(timestamp - self->spinStart);
    @hist["time(ns): spin->block"] = quantize(timestamp - self->spinStart);
    self->spinStart = 0;
}

plockstat$target:::mutex-acquire
/pid == $target && self->blockStart && equalizing/
{
    @occurrences["block->acquire"] = count();
    @average["average time(ns): block->acquire"] = avg(timestamp -
       self->blockStart);
    @hist["time(ns): block->acquire"] = quantize(timestamp - self->blockStart);
    self->blockStart = 0;
}

plockstat$target:::mutex-acquire
/pid == $target && self->blockStart == 0 && self->spinStart == 0  && equalizing/
{
    @occurrences["immediate acquire"] = count();
}
```

O seguinte resultado corresponde à utilização de 8 *threads* na implementação OpenMP.

```
spin->acquire                          6
block->acquire                        12
spin->block                           12
spin                                  18
immediate acquire                    271
average time(ns): spin->acquire    23697
average time(ns): spin->block      25835
average time(ns): block->acquire  422489
  time(ns): spin->acquire
       value  ------------- Distribution ------------- count
        4096 |                                         0
        8192 |@@@@@@@@@@@@@@@@@@@@                     3
       16384 |@@@@@@@@@@@@@                            2
       32768 |@@@@@@@                                  1
       65536 |                                         0

time(ns): spin->block
       value  ------------- Distribution ------------- count
        1024 |                                         0
        2048 |@@@@@@@                                  2
        4096 |@@@@@@@@@@                               3
        8192 |@@@                                      1
       16384 |@@@@@@@                                  2
       32768 |@@@@@@@@@@@@@                            4
       65536 |                                         0

time(ns): block->acquire
```

```
       value  ------------- Distribution ------------- count
       32768 |                                         0
       65536 |@@@@@@@                                  2
      131072 |@@@                                      1
      262144 |@@@@@@@@@@@@@                            4
      524288 |@@@@@@@@@@@@@@@@@@                       5
     1048576 |                                         0
```

Quando uma *thread* tenta adquirir um *lock*, ou este é adquirido imediatamente ou então a *thread* entra em *spinning*. Após um curto período tempo em *spinning*, ou o *lock* é adquirido ou a *thread* bloqueia. O tempo passado no estado *blocked* é significativamente superior ao tempo passado no estado *spinning*.

Existe um também número elevado de *imediate acquires*. A maioria destes devem estar relacionado com outras secções do código que não a região *critical*.

Após percebido o comportamento dos *mutexes*, é possível medir o tempo total cada *thread* gasta à espera nos mesmos.

Como nos resultados da secção anterior não houve nenhum bloqueio sem primeiro ocorrer *spinning* então o tempo de espera foi medido como sendo o tempo entre o evento *spin* e o evento *acquire*.

(Ficheiro plockstat_thread_wait.d)

```
proc:::lwp-create
/pid == $target/
{
    printf("%s, %d\n", stringof(args[1]->pr_fname), args[0]->pr_lwpid);
}

proc:::lwp-start
/pid == $target/
{
    self->threadStart = timestamp;
}

plockstat$target:::mutex-spin
/pid == $target && equalizing/
{
        self->spinStart = timestamp;
}

plockstat$target:::mutex-acquire
/pid == $target && self->spinStar && equalizingt/
{
        @time["total time(ns) waiting on mutexes:", tid] = sum(timestamp -
           self->spinStart);
        self->spinStart = 0;
}

proc:::lwp-exit
/pid == $target && self->threadStart && equalizing/
{
    @time["total time(ns)", tid] = sum(timestamp - self->threadStart);
    self->threadStart = 0;
}
```

Em teoria, quanto maior o número de *threads*, maior será a competição pela aquisição do *lock*, resultando em tempos médios de espera maiores

Time(ms)	ID	Waiting time	Total Time
2 Threads	**1**	0,01	-
	2	0,04	108,04
4 Threads	**1**	0,22	-
	2	0,10	81,81
	3	0,04	81,56
	4	0,46	81,65
8 Threads	**1**	0,47	-
	2	0,28	50,80
	3	0,81	50,80
	4	1,75	50,57
	5	0,96	50,67
	6	1,44	50,54
	7	1,20	50,33
	8	0,78	50,38

TABLE III: Tempos de espera nos mutexes (VERSÃO OMP)

Time(ms)	ID	Waiting Time	Total Time
2 Threads	2	0,02	156,96
	3	0,33	157,07
4 Threads	2	0,83	84,97
	3	0,83	84,88
	4	0,53	84,65
	5	0,12	83,84
8 Threads	2	0,30	55,22
	3	1,03	55,13
	4	0,52	54,94
	5	0,01	55,10
	6	0,47	55,07
	7	0,02	54,67
	8	0,54	54,62
	9	0,39	54,74

TABLE IV: Tempos de espera nos mutexes (VERSÃO C++)

Time(ms)	ID	Waiting Time	Total Time
2 Threads	2	0,19	148,22
	3	0,01	147,74
4 Threads	2	0,06	99,07
	3	0,30	98,84
	4	0,70	99,21
	5	0,40	98,69
8 Threads	2	1,20	72,07
	3	1,43	71,98
	4	0,78	71,72
	5	0,86	71,66
	6	0,19	71,33
	7	0,40	71,19
	8	0,73	71,21
	9	0,43	70,80

TABLE V: Tempos de espera nos mutexes (VERSÃO PTHREADS)

Os resultados confirmam a teoria, permitindo também aproximar a ordem pelo qual o *lock* foi adquirido.

Quando comparado com o *total time* de cada *thread*, que corresponde ao tempo decorrido entre esta começar a ser executada e ser destruída, o tempo despendido em *mutexes* é pouco significativo. Isto elimina a sincronização como um dos potenciais *bottlenecks* do programa.

A *thread* 1 não apresenta *total time* na implementação OpenMP pois não ativou a *probe* lwp-start quando o programa se encontrava na etapa de equalização. Nas outras versões a *thread* 1 não é contabilizada por não participar diretamente na equalização do histograma, ficando somente à espera que as restantes *threads* terminem.

Ao imprimir informação sobre a criação de *threads* foi possível verificar que embora fossem usadas várias zonas paralelas, as *threads* apenas foram criadas na primeira, o que indica que a implementação do OMP usada utiliza *thread pools*.

C. Mapeamento das threads

Em programas paralelizados em memória partilhada, o mapeamento das *threads* pode ser um factor importante no desempenho de um programa. Por exemplo, duas *threads* em *sockets* diferentes a utilizarem os mesmos dados é menos eficiente do que duas *threads* na mesma *socket*, pois resulta em custos adicionais de transferência de dados e de garantia de coerência.

Uma das tarefas em PCP consistia em optimizar o mapeamento das *threads*. O maior desafio dessa tarefa consistiu em conseguir analisar e monitorizar o mapeamento que era efetuado. O Dtrace, mais especificamente o *provider* sched, disponibiliza uma forma acessível de a efetuar, que é utilizada no *script* seguinte.

(Ficheiro sched_detailed.d:)

```
sched:::on-cpu
/pid == $target && equalizing/
{
    self->ts = timestamp;

    printf("Thread %d started running on CPU %d\n", tid, cpu);
}

sched:::off-cpu
/self->ts && equalizing/
{
```

```
    printf("Thread %d stopped running on CPU %d\n", tid, cpu);

    @sum[tid,cpu] = sum(timestamp - self->ts);
    @hist[tid] = quantize(timestamp - self->ts);

    self->ts = 0;
}
```

Ao imprimir o identificador do CPU quando uma *thread* começa e pára de correr, obtém-se um simples histórico do percurso de execução da *thread*.

A primeira agregação indica o tempo que cada *thread* executa em cada CPU.

O resultado seguinte corresponde execução com 2 *threads* em OpenMP, sendo a primeira coluna o *id* da *thread* e a segunda o *id* do processador.

```
    1         0          45488884
    1         3          102753751
    2         4          144937254
```

A segunda agregação é útil para observar a duração dos intervalos de execução. Intervalos muito curtos com interrupções constantes não são muito desejáveis.

O resultado seguinte corresponde aos intervalos de execução contínuos da *thread* 2, numa execução com 2 *threads* em OpenMP.

```
        2
           value  ------------- Distribution ------------- count
            4096 |                                         0
            8192 |@@@@                                     1
           16384 |@@@@@@@@@@@@@@@@@@@@@@                   5
           32768 |@@@@                                     1
           65536 |                                         0
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |                                         0
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         0
        16777216 |                                         0
        33554432 |                                         0
        67108864 |@@@@@@@@@@                               2
       134217728 |                                         0
```

Para utilizar eficientemente os recursos da máquina, é importante que haja uma boa distribuição de carga pelos vários CPUs. Verificar o total do tempo de execução em cada CPU é uma forma rudimentar de verificar o balanceamento da carga.

(Ficheiro sched_per_cpu.d:)

```
sched:::on-cpu
/pid == $target && equalizing/
{
    self->ts = timestamp;

}

sched:::off-cpu
/self->ts && equalizing/
{
    @sum[cpu] = sum(timestamp - self->ts);
    @hist[cpu] = quantize(timestamp - self->ts);

    self->ts = 0;
}
```

O *tracing* de uma execução com 8 *threads* resultou nos seguintes valores.

```
Por cpu (unidades em milissegundos)

        VERSAO OMP
            CPU         TEMPO(ms)
            6           44,06
            2           47,77
            3           47,77
            1           48,51
            7           52,46
            5           53,77
            0           54,41
            4           54,78

        VERSAO PTHREADS
            CPU         TEMPO(ms)
            6           49,34
            4           58,04
            2           58,27
            5           59,10
            0           59,22
            1           62,49
            3           63,08
            7           65,39

        VERSAO C++
            CPU         TEMPO(ms)
            3           41,67
            7           47,70
            1           50,88
            0           50,95
            6           51,04
            4           51,62
            5           51,69
            2           54,16
```

De acordo com este resultado, o tempo de execução é distribuído de forma algo equilibrada, o que era de esperar, dado que cada secção do histograma resulta na mesma carga.

VII. Algoritmo MPI

A. Descrição do algoritmo MPI

Esta versão do algoritmo baseia-se nos padrões de execução de *master-slave* e *heartbeat*.

Tal como o *master-slave*, um dos processos particiona os dados e recebe os resultados, e assim como o *heartbeat*, existe comunicação intermédia entre os processos.

O processo *master* é o responsável pela leitura e escrita da imagem, sendo o processo de equalização descrito da seguinte forma:

- ***Broadcast* dos metadados (MPI_Bcast)** - Tamanho de cada segmento de dados e valor máximo da escala de cizentos;
- **Envio das secções da imagem (MPI_Scatter)**;
- **Cálculo dos histogramas locais**;
- **Envio/receção e junção dos histogramas locais (MPI_Send/Receive)** - Cada *slave* envia o seu histograma ao *master* e este soma os valores de cada;
- **Normalização do histograma** - efetuada apenas pelo *master*;
- ***Broadcast* do histograma normalizado (MPI_Bcast)**;
- **Mapeamento das novas cores**;
- **Envio/receção das secções da imagem final(MPI_Gather)**.

Foi determinado durante a cadeira de PCP do semestre anterior, que o algoritmo não escala bem para a implementação MPI. Como não é um algoritmo *cpu-bound* o aumento das capacidades computacionais obtido com um maior número de processos não se traduz em melhorias suficientes para compensar os custos de comunicação entre processos.

B. Medições

Um dos fatores mais importantes na execução do algoritmo é a comunicação entre processos. De maneira a verificar o tempo gasto em cada primitiva MPI foram utilizadas as seguintes *probes*:

(Ficheiro mpiProbes.d:)

```
pid$target:libmpi:MPI_Send:entry
/uid==1015/
{
        self->sendStart = timestamp;
}

pid$target:libmpi:MPI_Send:return
/uid==1015/
{
        @count["sends", pid] = count();
        @ft["total time(ns) sending", pid] = sum(timestamp - self->sendStart);
        self->sendStart = 0;
}

pid$target:libmpi:MPI_Recv:entry
/uid==1015/
```

```
{
        self->receiveStart = timestamp;
}

pid$target:libmpi:MPI_Recv:return
/uid==1015/
{
        @count["receives", pid] = count();
        @ft["total time(ns) receiving", pid] = sum(timestamp -
           self->receiveStart);
        self->receiveStart = 0;
}

pid$target:libmpi:MPI_Scatter:entry
/uid==1015/
{
        self->scatterStart = timestamp;
}

pid$target:libmpi:MPI_Scatter:return
/uid==1015/
{
        @count["scaters", pid] = count();
        @ft["total time(ns) scattering", pid] = sum(timestamp -
           self->scatterStart);
        self->scatterStart = 0;
}

pid$target:libmpi:MPI_Gather:entry
/uid==1015/
{
        self->gatherStart = timestamp;
}

pid$target:libmpi:MPI_Gather:return
/uid==1015/
{
        @count["gathers", pid] = count();
        @ft["total time(ns) gathering", pid] = sum(timestamp -
           self->gatherStart);
        self->gatherStart = 0;
}

pid$target:libmpi:MPI_Bcast:entry
/uid==1015/
{
        self->bcastStart = timestamp;
}

pid$target:libmpi:MPI_Bcast:return
/uid==1015/
{
        @count["broadcasts", pid] = count();
        @ft["total time(ns) broadcasting", pid] = sum(timestamp -
           self->bcastStart);
        self->bcastStart = 0;
```

```
}

histogram_equalization$target:::after-read
{
        self->masterStart = timestamp;
}

histogram_equalization$target:::before-write
{
        printf("total equalization time(ns):%d\n", (timestamp -
           self->masterStart));
}
```

Foram realizados testes para diferentes tamanhos de imagem, 10, 50 e 100 megabytes, e para diferentes números de processos, 2, 4 e 8.

As seguintes tabelas representam o tempo de execução de cada primitiva, e o número de invocações de cada uma.

Master	time(ms)	gathering	receiving	scattering	broadcasting	total
2 Processos	**10mb**	4,81	0,17	18,37	13,99	66,63
	50mb	26,09	0,19	100,14	17,50	282,70
	100mb	75,43	0,20	190,53	12,22	585,71
4 Processos	**10mb**	6,95	0,15	42,41	11,98	75,43
	50mb	27,05	0,38	178,09	6,04	283,98
	100mb	59,37	0,18	238,06	4,44	419,32
8 Processos	**10mb**	16,89	0,25	405,44	132,26	563,32
	50mb	48,99	0,36	1127,00	108,14	1323,21
	100mb	69,24	0,24	1519,89	20,08	1696,36

TABLE VI: Tempos de execução para as diferentes primitivas

Master	gather	receives	scatters	broadcasts
2 Processos	1	1	1	2
4 Processos	1	3	2	2
8 Processos	1	7	2	2

TABLE VII: Número de invocações das diferentes primitivas

Como é possível observar, à medida que o número de processos aumenta, a comunicação ocupa uma maior percentagem do tempo de execução. Como seria de esperar, o tempo de execução dos *scatters*, *gathers* e*broadcasts* aumenta significativamente com o aumento de processos uma vez que vão ser enviados dados para mais processos.

O número de invocações das primitivas *gather*, *scatter* e *broadcast* não dependem do número de processos uma vez que são funções do tipo *one-to-many*. No entanto, o número de *receives* vai aumentar uma vez que vão haver mais *slaves* a comunicar informação para o processo *master*.

VIII. Conclusão

Em cadeiras anteriores, os efeitos de conceitos como sincronização e mapeamento de *threads* eram apenas observados através do tempo de execução. Em várias situações, alterações que em teoria eram optimizações resultavam em piores tempos de execução.

O Dtrace providencia uma forma acessível de verificar o impacto das alterações, de detetar potenciais problemas e de perceber o que realmente se passa no sistema.

O facto do objeto de estudo ter sido código desenvolvido por nós significou que o comportamento do programa já nos era familiar, permitindo-nos focar com mais detalhe na utilização do Dtrace.

Devido à sua natureza *low-level*, Pthreads e as bibliotecas utilizadas em C++ 11 permitem um maior controlo sobre o comportamento das *threads*. No entanto este maior controlo corresponde a um maior grau de dificuldade de utilização se optimizações complexas forem desejadas. Por sua vez, o OpenMP oferece uma interface simples que permite facilmente paralelizar o código, fornecendo também acesso a optimizações como o escalonamento.

Em situações específicas as directivas do OpenMP podem não ser suficientes para extrair o desempenho máximo das threads, sendo necessário implementar uma solução mais detalhada em Pthreads/C++. No algoritmo estudado, o OpenMP é suficientemente adequado para obter um desempenho razoável. Uma das razões de a implementação OpenMP desenvolvida ter obtido o melhor desempenho é a relativa simplicidade das implementações Pthreads e C++. Optimizações utilizadas em OpenMP não foram replicadas devido à elevada complexidade que a implementação destas requer.

ESC-TP4

José Pinto A81317 - Pedro Barbosa A82068

I. Introdução

Existem várias ferramentas diferentes para fazer a análise de um algoritmo. Neste relatório vai ser abordada a ferramenta perf.

Foi decido que o tutorial iria ser realizado primeiro, apesar de ser apresentado em segundo lugar no enunciado do projeto, uma vez que este ajuda a entender melhor o que é o perf e a utilizar melhor as suas funcionalidades. Ao longo do tutorial foi usado a implementação *naive* da multiplicação de matrizes.

Em primeiro lugar é apresentado um resumo de conceitos abordados no tutorial, como a metodologia a utilizar, os modos de medição do perf, entre outros.

De seguida as medições efetuadas no tutorial foram repetidas e os resultados analisados.

A primeira fase repetida consiste em identificar o *hotspot* do algoritmo de multiplicação de matrizes. Na segunda fase é introduzida a versão *interchange* da multiplicação de matrizes, que altera a ordem de execução dos ciclos de forma a otimizar o funcionamento da *cache*. O modo *counting* é utilizado para medir os contadores de *hardware* e comparar as duas versões. Na terceira fase são efetuadas medições com os modos *counting* e *sampling* de forma a verificar a exatidão deste último. Por fim, foram realizados os *flamegraphs* para as quatro versões do algoritmo(naive, interchange, large_naive, large_interchange).

Uma vez concluído o tutorial passou-se à aplicação da ferramenta em diferentes versões do algoritmo de ordenação de *arrays*.

Inicialmente são medidos vários *performance counters* para as quatro versões do algoritmo, para poderem ser comparados e dessa forma justificar a diferença nos tempos de execução. O segundo passo consistia na análise do tempo despendido pelos algoritmos nos diferentes tipos de requisitos (*kernel*, chamadas de bibliotecas de C ou funções do utilizador). De seguida, foram analisados os *flamegraphs* e por fim foi feita uma análise aprofundada (analisado o código *assembly*) do algoritmo *sort2*, de forma a encontrar a secção do código que seria mais apropriada para ser otimizada.

II. Ambiente de testes

As medições foram efetuadas nos nodos 431 do SeARCH. Existem duas variações dos nodos 431, uma com o processador X5650 e outra com o processador E5649. Decidimos utilizar as máquinas com o processador X5650.

Processador	Xeon X5650
Microarchitecture	Nehalem
Processor's frequency	2.66 GHz
#Cores	6
#Threads	12
Cache L1	32KB
Cache L2	256KB
Cache L3	12288KB

III. Eventos disponíveis

O perf permite instrumentar contadores de hardware e *tracepoints*.

A. Tracepoints

Os *tracepoints* permitem analisar comportamentos como chamadas do sistema ou operações sobre ficheiros.

```
syscalls:sys_exit_read
   [Tracepoint]
syscalls:sys_enter_write
   [Tracepoint]
sched:sched_stat_wait
   [Tracepoint]
sched:sched_stat_sleep
   [Tracepoint]
```

B. Contadores de hardware

Os contadores de *hardware* são registos no CPU que permitem contabilizar estatísticas como *misses* na *cache* e instruções executadas, permitindo de uma forma básica analisar o desempenho e encontrar *hotspots* de uma aplicação.

Cada processador disponibiliza um conjunto específico de contadores de *hardware* com nomenclaturas distintas. Para facilitar o seu uso o perf disponibiliza uma interface de eventos comuns. Como os eventos comuns do perf correspondem na realidade a contadores de *hardware*, não há garantia que estes sejam todos suportados ou que a correspondência seja total. Por exemplo, o processador mencionado no tutorial não distingue entre *loads* e *stores* na *cache*. Então ambos os eventos mapeiam para o mesmo contador o que pode induzir o utilizador em erro. Por esta razão é muitas vezes útil consultar a documentação disponível sobre o processador para evitar que os valores medidos sejam interpretados incorrectamente. Também é possível especificar os contadores a medir através do seu *raw identifier*. É importante ter em conta que o número de registos disponíveis é limitado. No entanto, é possível medir mais eventos do que os contadores disponíveis em troca de alguma precisão, recorrendo ao *multiplexing*.

```
cpu-cycles OR cycles   [Hardware event]
instructions           [Hardware event]
cache-references       [Hardware event]
cache-misses           [Hardware event]
```

De forma semelhante aos eventos de hardware, existem os eventos de software. Estes existem ao nível do *kernel*.

```
cpu-clock                [Software event]
task-clock               [Software event]
page-faults OR faults    [Software event]
```

C. Métricas derivadas

Interpretar cada evento isoladamente não é muito útil. Por exemplo, saber o número absoluto dos *cache-misses* ou de instruções executadas não é suficiente para determinar se existe ou não um problema de desempenho. Por sua vez, o número de *cache-misses* por acesso à memória ou instruções por ciclo já permite tirar algumas conclusões. O próprio perf apresenta algumas destas métricas quando os eventos correspondentes são medidos.

Da mesma forma que as métricas derivadas ajudam a interpretar os valores absolutos, é importante não os ignorar. Por exemplo, uma optimização a um algoritmo pode resultar num *miss-rate* superior e melhorar o desempenho se, por exemplo, diminuir drasticamente o número total de acessos à memória.

IV. Modos de medição

O perf disponibiliza duas alternativas para efetuar medições, perf stat e perf record, que correspondem aos modos *counting* e *sampling*, respectivamente.

A. Counting

O modo *counting* é relativamente simples. Após configurados e inicializados, os contadores só são lidos no final da medição. Desta forma os valores medidos correspondem à aplicação inteira, não sendo úteis para avaliar o desempenho de *hotspots* específicos. Em contrapartida isto significa que o *overhead* é baixo.

B. Sampling

O modo *sampling* permite efetuar medições sobre partes específicas do código sem ser necessário alterá-lo. O modo *sampling* funciona com base em interrupções. De cada vez que um contador origina uma interrupção, o *core* executa o *handler* do perf. O contador é lido e reiniciado, a secção do código que estava a ser executada é registada e outras métricas (CPU *core number*, *program counter*, etc), também são guardadas. A informação obtida, designada de amostra (*sample*), é então escrita num *buffer* e eventualmente no ficheiro *perf.data*.

Quanto maior o número de amostras, melhor será a exatidão e a resolução dos valores obtidos. Uma forma de aumentar o número de amostras é simplesmente aumentar a carga de trabalho a efetuar. Outra forma é repetir o teste várias vezes e agregar os resultados. Também é possível controlar a taxa de amostragem, existindo duas formas diferentes de o fazer.

A primeira opção consiste em definir um período fixo de amostragem. O tutorial utiliza como exemplo 100 000 instruções. Ou seja, de 100 000 em 100 000 instruções é gerada uma interrupção.

Alternativamente pode ser definida uma frequência em amostragens por segundo. O valor definido corresponde a uma média e não ao valor fixo, sendo que o perf ajusta dinamicamente a frequência durante a execução.

O valor definido afeta não só a qualidade das medições como também a execução do programa em si. Como as interrupções são processadas usando os mesmos recursos do programa a ser medido (desde tempo de execução no CPU a espaço na *cache*), o desempenho do programa a ser medido é afetado.

Ao definir o período fixo de amostragem é necessário ter em conta a frequência do evento a contar. Medir eventos como instruções executadas ou ciclos com um período curto resulta em *overhead* significativo para o programa. Medir eventos como *mispredicted branches* com um período longo resulta numa precisão baixa. É necessário encontrar um período equilibrado, principalmente se eventos com frequências muito diferentes forem observados simultaneamente.

Naturalmente, a exatidão do *sampling* é bastante limitada no que toca a atribuir eventos a instruções individuais, dado o reduzido tempo de execução destas. Em norma, ocorre *skid* ou seja, o evento é atribuído a uma instrução na vizinhança e não à responsável pela interrupção. No intervalo entre uma interrupção ser gerada e ser processada podem ser executadas instruções. A instrução que é registada na amostra é a que vai ser executada após a interrupção ser processada. Tendo estes fatores em conta, apenas é possível atribuir eventos com alguma certeza a regiões de código e não a instruções individuais.

V. Metodologia

O autor do tutorial descreve uma metodologia para a análise de desempenho de uma aplicação.

Em primeiro lugar é necessário descobrir quais as regiões de código responsáveis pela maioria do tempo de execução, pois é nessas regiões que optimizações vão ter o maior impacto. O modo *sampling* do perf pode ser utilizado para medir o tempo de execução de cada região do código.

O segundo passo consiste em determinar se existe algum problema de desempenho nas secções mais responsáveis pelo tempo de execução. Novamente, isto pode ser efetuado com o perf através dos contadores de *hardware*. Se, por exemplo, a secção apresentar um *miss rate* muito elevado então é um sinal pode existir um problema de desempenho.

O terceiro passo consiste em tentar resolver o problema. A utilidade do perf nesta situação é em permitir verificar o impacto das alterações efetuadas.

VI. Baseline

O código corresponde a uma implementação simples da multiplicação de matrizes. O tamanho das matrizes foi definido como sendo 1000. Desta forma o tempo de execução é semelhante ao do tutorial, sendo suficientemente significativo para permitir medições com alguma exatidão e para observar o efeito de possíveis alterações no código, sendo na mesma rápido efetuar múltiplas medições.

A aplicação foi compilada da seguinte forma:

```
gcc -o naive -ggdb -O2
   -fno-inline-small-functions
   -fno-omit-frame-pointer naive.c
```

A opção ggdb é utilizada para gerar informação de *debug* de forma a que o perf possa anotar as medições com informação como o nome das funções.

O *inlinig* é desativado para facilitar a compreensão das medições do perf, caso contrário a função multiply_matrixes pode ser incluída no *main*.

A optimização *omit-frame-pointer* é desativada para permitir a obtenção de *stack traces*.

Para evitar que o tempo medido corresponda a uma execução anómala foram efetuadas várias medições com perf stat.

```
1629.769994 cpu-clock (msec)
1.636000337 seconds time elapsed
981         faults

1632.187755 cpu-clock (msec)
1.637484172 seconds time elapsed
980         faults

1628.872191 cpu-clock (msec)
1.634181234 seconds time elapsed
980         faults

1628.047518 cpu-clock (msec)
1.633233210 seconds time elapsed
980         faults

1624.097122 cpu-clock (msec)
1.629273683 seconds time elapsed
980         faults
```

VII. Encontrar hotspots

A aplicação fornecida no tutorial é relativamente pequena. No entanto, numa situação real pode ser necessário encontrar os *hotspots* de aplicações com números significativos de ficheiros e bibliotecas. Desta forma deve-se começar por uma análise mais alto nível e ir progressivamente aprofundando.

O tutorial segue esta lógica, apresentando primeiro a opção *–sort comm,dso* que agrega os resultados por comando e por objeto partilhado.

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

# Samples: 6K of event 'cpu-clock'
# Event count (approx.): 6594
#
# Overhead  Command      Shared Object
# ........  .......  .................
#
    98.57%    naive  naive
     1.23%    naive  libc-2.12.so
     0.18%    naive  [kernel.kallsyms]
     0.02%    naive  ld-2.12.so
```

```
# Samples: 29  of event 'faults'
# Event count (approx.): 1003
#
# Overhead  Command      Shared Object
# ........  .......  .................
#
    82.45%    naive  naive
    14.66%    naive  ld-2.12.so
     2.29%    naive  libc-2.12.so
     0.60%    naive  [kernel.kallsyms]
```

Em termos de execução, a maioria do tempo corresponde ao executável *naive*, sendo o tempo gasto em bibliotecas externas ou chamadas de sistemas insignificante.

Em termos de *page faults*, a maioria ocorreu na biblioteca ld-2.12.so.

Após determinado qual o *shared object* responsável pela maioria do tempo de execução pode-se aprofundar mais a análise. Para isso é removida a *flag sort* e é adicionada a *flag dsos* para filtrar apenas as funções dos objetos a analisar.

```
    perf report --stdio
       --dsos=naive,libc-2.12.so

# Samples: 6K of event 'cpu-clock'
# Event count (approx.): 6594
#
# Overhead  Command  Shared Object
                     Symbol
# ........  .......  .............
   ......................
#
    98.20%    naive  naive
       [.] multiply_matrices
     0.55%    naive  libc-2.12.so
        [.] __random
     0.50%    naive  libc-2.12.so
        [.] __random_r
     0.35%    naive  naive
        [.] initialize_matrices
     0.18%    naive  libc-2.12.so
        [.] rand
     0.03%    naive  naive
        [.] rand@plt
```

```
# Samples: 29  of event 'faults'
# Event count (approx.): 1003
#
# Overhead  Command  Shared Object
                     Symbol
# ........  .......  .............
   ......................
#
    82.45%    naive  naive
       [.] initialize_matrices
     2.29%    naive  libc-2.12.so
```

```
        [.] _exit
```

Em termos de execução, a maioria do tempo corresponde à função multiply_matrices. A maioria das *page-faults* ocorre no curto intervalo de tempo em que a função initialize_matrices é executada. Em vez de partir diretamente para a conclusão que isto representa um problema de desempenho é necessário contextualizar os valores obtidos nas medições com o algoritmo em si. A concentração elevada de *page-faults* na função initialize_matrices é normal pois é nesta que os *arrays* das matrizes são acedidos pela primeira vez.

Para analisar em mais detalhe o tempo despendido numa função é necessário analisar em termos de linhas e instruções. O *perf annotate* é útil nestas situações pois apresenta os valores das amostras em relação às instruções *assembly* executadas. Se o programa tiver sido compilado com informação de *debugging* então o código fonte é apresentado junto das instruções *assembly*.

Novamente são usadas *flags* para filtrar o resultado.

```
perf annotate --stdio --dsos=naive --symbol=multiply_matrices
 Percent |      Source code & Disassembly of naive for cpu-clock
------------------------------------------------------------------
         :
         :
         :
         :      Disassembly of section .text:
         :
         :      00000000004004d0 <multiply_matrices>:
         :          }
         :        }
         :      }
         :
         :      void multiply_matrices()
         :      {
    0.00 :        4004d0:       xor    %r10d,%r10d
    0.00 :        4004d3:       xor    %r8d,%r8d
         :        int i, j, k ;
         :
         :        for (i = 0 ; i < MSIZE ; i++) {
    0.00 :        4004d6:       movslq %r10d,%r9
    0.00 :        4004d9:       xor    %esi,%esi
    0.00 :        4004db:       imul   $0xfa0,%r9,%r9
    0.00 :        4004e2:       lea    0x601220(%r9),%rdi
         :          for (j = 0 ; j < MSIZE ; j++) {
    0.00 :        4004e9:       add    $0xda2420,%r9
    0.06 :        4004f0:       movslq %esi,%rax
    0.00 :        4004f3:       mov    %r8d,-0x4(%rsp)
    0.00 :        4004f8:       mov    %r9,%rcx
    0.00 :        4004fb:       lea    0x9d1b20(,%rax,4),%rdx
    0.03 :        400503:       movss  -0x4(%rsp),%xmm1
    0.00 :        400509:       xor    %eax,%eax
    0.00 :        40050b:       nopl   0x0(%rax,%rax,1)
         :            float sum = 0.0 ;
         :            for (k = 0 ; k < MSIZE ; k++) {
         :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
   20.27 :        400510:       movss  (%rcx),%xmm0
         :        int i, j, k ;
         :       [a82068@compute-431-6 tutorial]$ time perf record -e cpu-clock
            --freq=100000 ./naive
[ perf record: Woken up 28 times to write data ]
[ perf record: Captured and wrote 6.993 MB perf.data (~305515 samples) ]

real    0m2.026s
user    0m1.873s
sys     0m0.079s
```

```
       :        for (i = 0 ; i < MSIZE ; i++) {
       :          for (j = 0 ; j < MSIZE ; j++) {
       :            float sum = 0.0 ;
       :            for (k = 0 ; k < MSIZE ; k++) {
  0.64 :        400514:       add    $0x1,%eax
       :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
  0.00 :        400517:       mulss  (%rdx),%xmm0
       :        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++) {
 33.03 :        40051b:       add    $0x4,%rcx
 18.02 :        40051f:       add    $0xfa0,%rdx
  0.00 :        400526:       cmp    $0x3e8,%eax
       :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
  0.00 :        40052b:       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++) {
 27.91 :        40052f:       jne    400510 <multiply_matrices+0x40>
       :      void multiply_matrices()
       :      {
       :        int i, j, k ;
       :
       :        for (i = 0 ; i < MSIZE ; i++) {
       :          for (j = 0 ; j < MSIZE ; j++) {
  0.00 :        400531:       add    $0x1,%esi
       :            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 ;
  0.00 :        400534:       movss  %xmm1,(%rdi)
       :      void multiply_matrices()
       :      {
       :        int i, j, k ;
       :
       :        for (i = 0 ; i < MSIZE ; i++) {
       :          for (j = 0 ; j < MSIZE ; j++) {
  0.05 :        400538:       add    $0x4,%rdi
  0.00 :        40053c:       cmp    $0x3e8,%esi
  0.00 :        400542:       jne    4004f0 <multiply_matrices+0x20>
       :
       :      void multiply_matrices()
       :      {
       :        int i, j, k ;
       :
       :        for (i = 0 ; i < MSIZE ; i++) {
  0.00 :        400544:       add    $0x1,%r10d
  0.00 :        400548:       cmp    $0x3e8,%r10d
```

```
    0.00 :         40054f:       jne    4004d6 <multiply_matrices+0x6>
    0.00 :         400551:       retq
    }}}}}}}}}}}}}}}}}}}}}}
```

A *flag –no-source* faz com que apenas o *assembly* seja impresso, melhorando a legibilidade

```
 Percent |      Source code & Disassembly of naive for cpu-clock
------------------------------------------------------------------
         :
         :
         :
         :      Disassembly of section .text:
         :
         :      00000000004004d0 <multiply_matrices>:
    0.00 :        4004d0:       xor    %r10d,%r10d
    0.00 :        4004d3:       xor    %r8d,%r8d
    0.00 :        4004d6:       movslq %r10d,%r9
    0.00 :        4004d9:       xor    %esi,%esi
    0.00 :        4004db:       imul   $0xfa0,%r9,%r9
    0.00 :        4004e2:       lea    0x601220(%r9),%rdi
    0.00 :        4004e9:       add    $0xda2420,%r9
    0.00 :        4004f0:       movslq %esi,%rax
    0.00 :        4004f3:       mov    %r8d,-0x4(%rsp)
    0.00 :        4004f8:       mov    %r9,%rcx
    0.00 :        4004fb:       lea    0x9d1b20(,%rax,4),%rdx
    0.02 :        400503:       movss  -0x4(%rsp),%xmm1
    0.00 :        400509:       xor    %eax,%eax
    0.00 :        40050b:       nopl   0x0(%rax,%rax,1)
   19.94 :        400510:       movss  (%rcx),%xmm0
    0.90 :        400514:       add    $0x1,%eax
    0.00 :        400517:       mulss  (%rdx),%xmm0
   33.79 :        40051b:       add    $0x4,%rcx
   16.03 :        40051f:       add    $0xfa0,%rdx
    0.00 :        400526:       cmp    $0x3e8,%eax
    0.00 :        40052b:       addss  %xmm0,%xmm1
   29.33 :        40052f:       jne    400510 <multiply_matrices+0x40>
    0.00 :        400531:       add    $0x1,%esi
    0.00 :        400534:       movss  %xmm1,(%rdi)
    0.00 :        400538:       add    $0x4,%rdi
    0.00 :        40053c:       cmp    $0x3e8,%esi
    0.00 :        400542:       jne    4004f0 <multiply_matrices+0x20>
    0.00 :        400544:       add    $0x1,%r10d
    0.00 :        400548:       cmp    $0x3e8,%r10d
    0.00 :        40054f:       jne    4004d6 <multiply_matrices+0x6>
    0.00 :        400551:       retq
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

Tal como no tutorial o código apresentado tem partes repetidas e está ordenado diferentemente devido às optimizações efectuadas pelo compilador.

A maioria do tempo de execução é atribuído a quatro instruções. Devido ao *skid* não temos garantias que as percentagens atribuídas a cada instrução, o *hotspot* corresponde à linha:

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

VIII. Contagem de eventos de Hardware

A. Workload

Excluindo algumas métricas como os *miss rates*, observar isoladamente as medições não tem muito significado. As medições são úteis para analisar o efeito de optimizações e efetuar comparações entre várias versões. Por esta razão o tutorial apresenta outra versão da multiplicações das matrizes,

a versão *interchange*. Nesta versão a ordem dos ciclos é trocada de forma a melhorar o desempenho da *cache*.

B. Contadores

De seguida são medidos vários contadores.

	Naive	Interchange
cpu-cycle	4970127590	3135878608
instructions	7123378359	7118274722
cache-references	63421642	1943455
cache-misses	188753	255515
branch-instructions	1033432676	1033788517
branch-misses	1016621	1011773
bus-cycles	<not supported>	<not supported>
L1-dcache-loads	2037152091	2037250979
L1-dcache-load-misses	1076807616	63232368
L1-dcache-stores	22892989	1022366081
L1-dcache-store-misses	1328118	367035
L1-icache-loads	95694104	3058250948
L1-icache-load-misses	360426	238722
LLC-loads	62921700	1601439
LLC-load-misses	28221	22363
LLC-stores	812519	656571
LLC-store-misses	114185	153299
dTLB-load-misses	20994	28158
dTLB-store-misses	3839	3239
iTLB-load-misses	611	656
branch-loads	1033406797	1033097827
branch-load-misses	41241170	1001792821

Naturalmente os valores obtidos são diferentes do tutorial mas as conclusões são as mesmas.

A troca da ordem dos ciclos não afeta o número de instruções executadas nem os *branch-instructions* e os *brach-misses*.

O padrão de acessos à memória do *interchange* é substancialmente melhor, o que é confirmado pela redução no número de ciclos, *misses* na *cache L1*, *data TLB misses*, entre outros.

C. Métricas derivadas:

Tal como discutido anteriormente, é muitas vezes útil derivar métricas a partir dos valores obtidos que auxiliam na sua compreensão.

(PTI = per thousand instructions)

	Naive	Interchange
Elapsed Time (seconds)	1.72	1.08
Instructions per cycle	1,4332	2,2699
L1 cache miss ratio	0,5286	0,031
L1 cache miss rate PTI	151,1653	8,8831
Data TLB miss ratio	0,0003	0,0145
Data TLB miss rate PTI	0,0029	0,004
Branch mispredict ratio	0,001	0,001
Branch mispredict rate PTI	0,1427	0,1421

Apesar de o número de instruções executadas ser semelhante, o IPC (instruções por ciclo) é superior pois a quantidade de ciclos gastos à espera da memória é menor. Isto é reforçado pelas diferenças significativas nas métricas *L1 cache miss ratio* e *L1 cache miss rate PTI*.

IX. Perfis de Eventos de Hardware

A terceira parte do tutorial foca-se no modo *sampling*.

Tal como no tutorial, os tamanhos das matrizes foram aumentados. Para obter tempos de execução comparáveis aos do tutorial, o tamanho das matrizes passou de 1000 para 2500. Com este aumento o processo de *sampling* efectuará mais amostras e terá mais exatidão.

Para avaliar a precisão do *sampling*, foram também efetuadas medições com o modo *counting*.

A. Counting

	Large Naive	Large Interchange
elapsed time	39.54s	18.81s
instructions	110,187,455,946	110,153,817,648
cycles	104,494,655,972	50,864,728,797
cache references	1,035,603,039	44,153,667
cache misses	951,246,798	37,964,157
LLC loads	1,026,833,714	39,682,429
LLC load misses	947,700,011	35,698,753
dTLB load misses	422,440	111,405
branches	15,846,029,420	15,835,536,333
branch misses	6,384,232	6,321,085

TABLE I: Performance events(Large Naive vs Large Interchange)

	Large Naive	Large Interchange
Instructions per cycle	1.05	2.17
Cache miss ratio	0.92	0.86
Cache miss ratio PTI	8.63	0.34
LLC load miss ratio	0.92	0.90
LLC load miss rate PTI	8.60	0.32
Data TLB miss ratio	0.0004	0.002
Branch mispredict ratio	0.0004	0.0003
Branch mispredict rate PTI	0.06	0.06

TABLE II: Ratios and rates

B. Sampling

	Large Naive	#S	Large Interchange	#S
elapsed time	41.78s	-	20.11s	-
instructions	110,681,000,000	1M	110,622,100,000	1M
cycles	105,061,800,000	1M	51,473,500,000	514K
cache references	1,038,700,000	10K	43,400,000	434
cache misses	954,300,000	9K	37,600,000	376
LLC loads	1,028,800,000	10K	38,700,000	387
LLC load misses	950,700,000	9K	35,300,000	353
dTLB load misses	400,000	4	100,000	1
branches	15,859,900,000	158K	15,850,600,000	158K
branch misses	6,300,000	63	6,300,000	63

TABLE III: Performance events(Large Naive vs Large Interchange)

	Large Naive	Large Interchange
Instructions per cycle	1.05	2.15
Cache miss ratio	0.91	0.87
Cache miss ratio PTI	8.62	0.34
LLC load miss ratio	0.92	0.91
LLC load miss rate PTI	8.59	0.32
Data TLB miss ratio	0.0004	0.002
Branch mispredict ratio	0.0004	0.0004
Branch mispredict rate PTI	0.06	0.06

TABLE IV: Ratios and rates

Para o período de 100000 selecionado, os valores obtidos foram bastante próximos dos do modo *counting*.

Das métricas derivadas, o *cache miss ratio* é bastante elevado, o que à primeira vista não parece correto. Um estudo mais detalhado da documentação indica que os eventos *cache references* e *cache misses* não correspondem à definição originalmente assumida. *Cache misses* não corresponde à soma de todos os *misses* em todos os níveis de cache mas sim ao número de vezes que a informação requerida não se encontrava em nenhum dos níveis.

X. Análise de uma aplicação C/C++

A. Pergunta 1

A primeira pergunta consistia na realização da medição de diferentes eventos de forma a conseguir explicar a diferença no desempenho das diferentes versões do algoritmo *sort*. Para isso foram medidos os seguintes eventos:

- cpu-cycles
- instructions
- L1-dcache-loads
- L1-dcache-load-misses
- L1-dcache-stores
- L1-dcache-store-misses

Exemplo do comando utilizado:

```
perf stat -e <eventos> -r 5 ./sort 1 1
   100000000
```

A *flag* "-e" permite especificar a lista de eventos a ser processada e a *flag* "-r" é para repetir o teste várias vezes e apresentar uma média das 5 execuções, de forma a evitar resultados anómalos.

De seguida apresenta-se o resultado da execução do *perf stat* para os diferentes algoritmos:

```
Sort 1(quick):
45741754299      cpu-cycles                    ( +-  0.08% ) [66.67%]
46216295260      instructions                  #    1.01  insns per cycle
   ( +-  0.02% ) [83.33%]
 5833143235      L1-dcache-loads
    ( +-  0.02% ) [83.33%]
  166557029      L1-dcache-load-misses         #    2.86% of all L1-dcache hits
     ( +-  0.08% ) [83.33%]
 2698165128      L1-dcache-stores
    ( +-  0.02% ) [83.34%]
   25350574      L1-dcache-store-misses
      ( +-  0.05% ) [83.33%]

17.290361691 seconds time elapsed                                ( +-  0.07% )
```

```
Sort 2(radix):
39902821244      cpu-cycles                ( +-  0.75% ) [66.66%]
44044848956      instructions              #    1.10  insns per cycle
   ( +-  0.01% ) [83.33%]
 5825446326      L1-dcache-loads
    ( +-  0.02% ) [83.33%]
  296577428      L1-dcache-load-misses     #    5.09% of all L1-dcache hits
     ( +-  0.05% ) [83.33%]
 4313715032      L1-dcache-stores
    ( +-  0.02% ) [83.34%]
  131362815      L1-dcache-store-misses
     ( +-  0.03% ) [83.33%]

15.085599820 seconds time elapsed                          ( +-  0.76% )



Sort 3(heap):
96011691876      cpu-cycles                ( +-  4.31% ) [66.67%]
56495993164      instructions              #    0.59  insns per cycle
   ( +-  0.01% ) [83.34%]
 5736117794      L1-dcache-loads
    ( +-  0.01% ) [83.33%]
 3200369848      L1-dcache-load-misses     #   55.79% of all L1-dcache hits
    ( +-  0.18% ) [83.34%]
 3847956384      L1-dcache-stores
    ( +-  0.01% ) [83.33%]
   26692099      L1-dcache-store-misses
      ( +-  0.32% ) [83.34%]

36.335892748 seconds time elapsed                          ( +-  4.33% )

3 200 369 848
444 504 933

Sort 4(merge):
64523497888      cpu-cycles                ( +-  0.19% ) [66.66%]
83975618026      instructions              #    1.30  insns per cycle
   ( +-  0.01% ) [83.33%]
15719519329      L1-dcache-loads
   ( +-  0.02% ) [83.33%]
  444504933      L1-dcache-load-misses     #    2.83% of all L1-dcache hits
     ( +-  0.05% ) [83.34%]
10297660059      L1-dcache-stores
   ( +-  0.01% ) [83.34%]
  238026325      L1-dcache-store-misses
     ( +-  0.07% ) [83.34%]

24.387129328 seconds time elapsed                                          ( +-
   0.20% )
```

Geralmente, o número de instruções está diretamente relacionado com o tempo de execução de um programa, ou seja, um menor número de instruções resulta num melhor tempo de execução (existindo sempre exceções pois o tempo de execução de cada instrução pode variar significativamente). Esta "regra" verifica-se para o *sort* 1 e 2, no entanto, o *sort4* apresenta um número de instruções bastante superior ao *sort3*, apresentando no entanto um tempo de execução bastante inferior. Esta discrepância deve-se à má utilização da *cache* por parte do *sort3*, que apresenta aproximadamente 7 vezes mais *misses* que o *sort4* (cerca de 55% dos acessos à *cache* L1 por parte do *sort3* resultam num *miss*).

	Tempo(s)	CPU-cycles	Instructions	L1-dcache-loads	L1-dcache-load-misses	L1-dcache-stores	L1-dcache-store-misses
Sort 1	17.29	45,741,754,299	46,216,295,260	5,833,143,235	166,557,029	2,698,165,128	25,350,574
Sort 2	15.09	39,902,821,244	44,044,848,956	5,825,446,326	296,577,428	4,313,715,032	131,362,815
Sort 3	36.34	96,011,691,876	56,495,993,164	5,736,117,794	3,200,369,848	3,847,956,384	26,692,099
Sort 4	24.39	64,523,497,888	83,975,618,026	15,719,519,329	444,504,933	10,297,660,059	238,026,325

TABLE V: Consolidação dos resultados obtidos

B. Pergunta 2

```
[a82068@compute-431-1 prog_sort]$ perf record -F 99 ./sort 1 1 100000000
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.072 MB perf.data (~3154 samples) ]

[a82068@compute-431-1 prog_sort]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only
   options.
#
# Samples: 1K of event 'cycles'
# Event count (approx.): 45775580946
#
# Overhead       Samples  Command      Shared Object                      Symbol
# ........  ............  .......  .................  ........................
#
    90.98%          1552     sort  sort               [.] sort1(int*, int, int)
     2.28%            39     sort  sort               [.] main
     1.88%            32     sort  libc-2.12.so       [.] __random_r
     1.70%            29     sort  libc-2.12.so       [.] __random
     0.85%             9     sort  libc-2.12.so       [.] rand
     0.82%            14     sort  [kernel.kallsyms]  [k] clear_page_c
     0.29%             5     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.26%             1     sort  [kernel.kallsyms]  [k] __list_add
     0.23%             4     sort  sort               [.] rand@plt
     0.18%             1     sort  [kernel.kallsyms]  [k] sha_transform
     0.12%             2     sort  [kernel.kallsyms]  [k] scheduler_tick
     0.12%             2     sort  [kernel.kallsyms]  [k] rcu_process_gp_end
     0.06%             1     sort  [kernel.kallsyms]  [k] rb_insert_color
     0.06%             1     sort  [kernel.kallsyms]  [k] list_del
     0.06%             1     sort  [kernel.kallsyms]  [k] x86_pmu_enable
     0.06%             1     sort  [kernel.kallsyms]  [k] task_tick_fair
     0.06%             1     sort  [kernel.kallsyms]  [k] ____pagevec_lru_add
     0.00%            12     sort  [kernel.kallsyms]  [k] native_write_msr_safe

     Bibliotecas:4.43%
     Kernel:2.09%
     Utilizador:93.49%


[a82068@compute-431-1 prog_sort]$ perf record -F 99 ./sort 2 1 100000000
[ perf record: Woken up 1 times to write data ]
```

```
[ perf record: Captured and wrote 0.063 MB perf.data (~2759 samples) ]

[a82068@compute-431-1 prog_sort]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only
   options.
#
# Samples: 1K of event 'cycles'
# Event count (approx.): 39278756729
#
# Overhead       Samples  Command      Shared Object                      Symbol
# ........  ............  .......  .................  ........................
#
    89.83%          1315     sort  sort               [.] sort2(int*, int)
     3.07%            45     sort  sort               [.] main
     2.05%            30     sort  libc-2.12.so       [.] __random
     1.84%            27     sort  libc-2.12.so       [.] __random_r
     1.50%            22     sort  [kernel.kallsyms]  [k] clear_page_c
     0.55%             8     sort  libc-2.12.so       [.] rand
     0.30%             1     sort  [kernel.kallsyms]  [k] acl_permission_check
     0.27%             4     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.27%             4     sort  sort               [.] rand@plt
     0.24%             1     sort  [kernel.kallsyms]  [k] strnlen_user
     0.07%             1     sort  [kernel.kallsyms]  [k] __free_pages_ok
     0.00%            12     sort  [kernel.kallsyms]  [k] native_write_msr_safe

     Bibliotecas:4.44%
     Kernel:2.38%
     Utilizador:94.17%


[a82068@compute-431-1 prog_sort]$ perf record -F 99 ./sort 3 1 100000000
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.147 MB perf.data (~6440 samples) ]

[a82068@compute-431-1 prog_sort]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only
   options.
#
# Samples: 3K of event 'cycles'
# Event count (approx.): 98561337878
#
# Overhead       Samples  Command      Shared Object                          Symbol
# ........  ............  .......  .................  ............................
#
    95.70%          3512     sort  sort               [.] sort3(int*, int)
     1.12%            41     sort  sort               [.] main
     1.01%            37     sort  libc-2.12.so       [.] __random_r
     0.73%            27     sort  libc-2.12.so       [.] __random
     0.35%            13     sort  [kernel.kallsyms]  [k] clear_page_c
     0.24%             9     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.22%             8     sort  libc-2.12.so       [.] rand
     0.12%             1     sort  [nfs]              [k] nfs_follow_link
     0.12%             1     sort  ld-2.12.so         [.] strlen
     0.11%             4     sort  sort               [.] rand@plt
     0.03%             1     sort  [kernel.kallsyms]  [k] _spin_unlock_irqrestore
     0.03%             1     sort  [kernel.kallsyms]  [k] idle_cpu
```

```
     0.03%             1     sort  [kernel.kallsyms]  [k] rebalance_domains
     0.03%             1     sort  [kernel.kallsyms]  [k] apic_timer_interrupt
     0.03%             1     sort  [kernel.kallsyms]  [k] rcu_bh_qs
     0.03%             1     sort  [kernel.kallsyms]  [k] irq_exit
     0.03%             1     sort  [kernel.kallsyms]  [k] __rcu_pending
     0.03%             1     sort  [kernel.kallsyms]  [k] update_cpu_load
     0.03%             1     sort  [kernel.kallsyms]  [k] perf_ctx_lock
     0.03%             1     sort  [kernel.kallsyms]  [k] tick_sched_timer
     0.00%            16     sort  [kernel.kallsyms]  [k] native_write_msr_safe

     Bibliotecas:2.08%
     Kernel:0.79%
     Utilizador:96.93%


[a82068@compute-431-1 prog_sort]$ perf record -F 99 ./sort 4 1 100000000
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.098 MB perf.data (~4275 samples) ]

[a82068@compute-431-1 prog_sort]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only
   options.
#
# Samples: 2K of event 'cycles'
# Event count (approx.): 63736629205
#
# Overhead       Samples  Command      Shared Object
                              Symbol
# ........  ............  .......  ..................
   ..................................
#
    75.74%          1799     sort  sort               [.] aux_sort4(int*, int,
       int, int)
     4.97%           118     sort  libc-2.12.so      [.] _int_malloc
     4.13%            98     sort  libc-2.12.so      [.] _int_free
     3.07%            73     sort  sort               [.] sort4(int*, int, int)
     2.78%            66     sort  libc-2.12.so      [.] malloc
     2.10%            50     sort  sort               [.] main
     1.51%            36     sort  [kernel.kallsyms]  [k] clear_page_c
     1.35%            32     sort  libc-2.12.so      [.] __random
     1.22%            29     sort  libc-2.12.so      [.] free
     1.14%            27     sort  libc-2.12.so      [.] __random_r
     0.67%            16     sort  libc-2.12.so      [.] malloc_consolidate
     0.34%             8     sort  libc-2.12.so      [.] rand
     0.20%             1     sort  [kernel.kallsyms]  [k] acl_permission_check
     0.18%             1     sort  [kernel.kallsyms]  [k] load_elf_binary
     0.17%             4     sort  sort               [.] malloc@plt
     0.17%             4     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.04%             1     sort  [kernel.kallsyms]  [k] _spin_lock
     0.04%             1     sort  [kernel.kallsyms]  [k] apic_timer_interrupt
     0.04%             1     sort  [kernel.kallsyms]  [k] page_fault
     0.04%             1     sort  sort               [.] rand@plt
     0.04%             1     sort  [kernel.kallsyms]  [k] raise_softirq
     0.04%             1     sort  [kernel.kallsyms]  [k] get_page_from_freelist
     0.00%            12     sort  [kernel.kallsyms]  [k] native_write_msr_safe
```

```
Bibliotecas:19.67%
Kernel:2.26%
Utilizador:81.12%
```

Como seria de esperar, em todas as variações do *sort* a maior percentagem do tempo é despendida na execução de funções do utilizador.

Seguem-se as funções das bibliotecas de C, com especial atenção para a função *random* que é utilizada para preencher duas das três matrizes do programa, demorando assim uma percentagem significativa(2-4%) da execução do programa em todas as versões do algoritmo.

É de notar que no *sort* 4 as funções *malloc* e *free* ocupam percentagem bastantes superiores às restantes. Após analisar o código foi possível verificar que o mesmo faz invocações recursivas da função *aux_sort4*, onde é feita a alocação de memória para um *array* por cada invocação da função, levando assim a uma elevada percentagem do tempo de execução.

O resto do tempo é despendido na execução de "funções" do *kernel* (*system calls*), que apesar de serem bastantes executam quase imediatamente ocupando uma baixa percentagem do tempo de execução.

C. Pergunta 3

Na terceira pergunta é pedido que sejam feitos os *FlameGraphs* das quatro versões do algoritmo, utilizando as *flags* -ag.

A *flag* "-a" é utilizada para recolher informação de todos os processadores disponíveis na máquina. Uma vez que todas as implementações são sequenciais apenas um processador vai ser utilizado. Os restantes processadores ficam a executar um processo *swapper*, que é um processo com prioridade mínima que só corre se não existirem mais processos para correr. Isto leva a que o perf recolha informação relativa ao processo *swapper*, que vai ocupar uma porção significativa do *FlameGraph* com informação irrelevante e dificultar a leitura do mesmo.

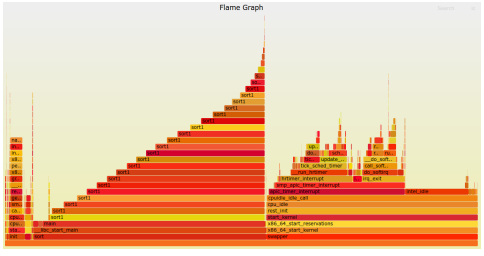

Fig. 1: Flamegraph do sort1 com a flag -a

Se retirar a *flag* "-a", ou seja se correr o seguinte comando,

```
perf record -F 99 -g ./sort 1 1
    100000000
```

o resultado obtido não inclui essa secção.

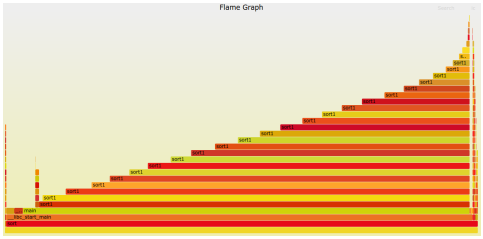

Fig. 2: Flamegraph do sort1 sem a flag -a

Para além disso foi ainda adicionada a *flag -fno-omit-frame-pointer* uma vez que sem ela não é possível obter *reliable stack traces*.

Como é possível observar pelas seguintes imagens, o *sort* 1 e 4 são invocados múltiplas vezes e cada vez demoram menos tempo a executar. No caso do *sort4* a execução alterna entre *sort4* e *aux_sort4*. Assim é possível identificar que ambos são algoritmos recursivos. As duas outras versões do *sort* apenas são invocadas uma vez (não recursivas).

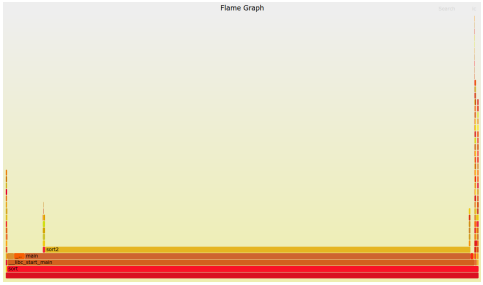

Fig. 3: FlameGraph do sort 2

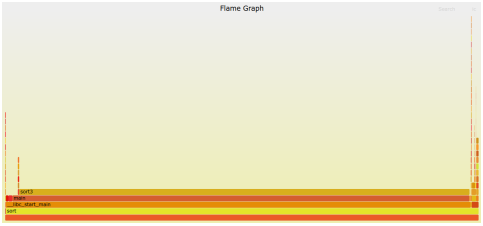

Fig. 4: FlameGraph do sort 3

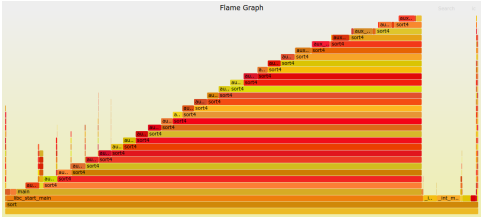

Fig. 5: FlameGraph do sort 4

D. Pergunta 4

Para imprimir o código *assembly* anotado com a distribuição do tempo de execução foi utilizado o seguinte comando:

```
perf annotate --stdio --dsos=sort
   --symbol="sort2(int*, int)"
```

Devido ao *skid* algumas das percentagens podem estar mal atribuídas sendo por isso complicado atribuir o valor medido a uma instrução específica sendo por isso mais apropriado atribui-lo à região do código.

Foram encontradas duas zonas de grande impacto na execução do algoritmo.

Zona 1:

```
         :           while (maior/exp > 0) {
         :               int bucket[10] = { 0 };
    0.00 :        400c7a:       movq   $0x0,-0x60(%rbp)
    0.00 :        400c82:       movq   $0x0,-0x58(%rbp)
    0.00 :        400c8a:       movq   $0x0,-0x50(%rbp)
    0.00 :        400c92:       movq   $0x0,-0x48(%rbp)
    0.00 :        400c9a:       movq   $0x0,-0x40(%rbp)
         :               for (i = 0; i < tamanho; i++)
    0.00 :        400ca2:       jle    400d95 <sort2(int*, int)+0x1b5>
    0.00 :        400ca8:       mov    %r12,%r8
    0.00 :        400cab:       nopl   0x0(%rax,%rax,1)
         :                   bucket[(vetor[i] / exp) % 10]++;
    2.36 :        400cb0:       mov    (%r8),%eax
    0.50 :        400cb3:       add    $0x4,%r8
    0.00 :        400cb7:       cltd
    0.00 :        400cb8:       idiv   %ecx
    6.13 :        400cba:       mov    %eax,%esi
    0.49 :        400cbc:       imul   %r10d
    3.88 :        400cbf:       mov    %esi,%eax
    0.05 :        400cc1:       sar    $0x1f,%eax
    1.84 :        400cc4:       sar    $0x2,%edx
    0.98 :        400cc7:       sub    %eax,%edx
    1.65 :        400cc9:       lea    (%rdx,%rdx,4),%eax
    1.74 :        400ccc:       add    %eax,%eax
    2.34 :        400cce:       sub    %eax,%esi
    2.06 :        400cd0:       movslq %esi,%rdx
    2.05 :        400cd3:       addl   $0x1,-0x60(%rbp,%rdx,4)
         :                   maior = vetor[i];
         :           }
         :
         :           while (maior/exp > 0) {
         :               int bucket[10] = { 0 };
         :               for (i = 0; i < tamanho; i++)
   15.55 :        400cd8:       cmp    %r14,%r8
```

Esta zona demora cerca de 41.62% do tempo total de execução do algoritmo.

Zona 2:

```
         :               for (i = tamanho - 1; i >= 0; i--)
    0.00 :        400d03:       test   %ebx,%ebx
    0.00 :        400d05:       mov    %r13,%rsi
    0.00 :        400d08:       jle    400d6f <sort2(int*, int)+0x18f>
    0.00 :        400d0a:       nopw   0x0(%rax,%rax,1)
         :                   b[--bucket[(vetor[i] / exp) % 10]] = vetor[i];
    2.47 :        400d10:       mov    (%rsi),%r8d
    0.58 :        400d13:       sub    $0x4,%rsi
    0.00 :        400d17:       mov    %r8d,%eax
    0.01 :        400d1a:       cltd
```

```
    2.28 :         400d1b:       idiv   %ecx
    6.60 :         400d1d:       mov    %eax,%r9d
    2.11 :         400d20:       imul   %r10d
    2.38 :         400d23:       mov    %r9d,%eax
    1.51 :         400d26:       sar    $0x1f,%eax
    0.72 :         400d29:       sar    $0x2,%edx
    0.17 :         400d2c:       sub    %eax,%edx
    0.96 :         400d2e:       lea    (%rdx,%rdx,4),%eax
    2.27 :         400d31:       mov    %r9d,%edx
    0.00 :         400d34:       add    %eax,%eax
    1.09 :         400d36:       sub    %eax,%edx
    1.13 :         400d38:       movslq %edx,%rdx
    2.33 :         400d3b:       mov    -0x60(%rbp,%rdx,4),%eax
    5.52 :         400d3f:       sub    $0x1,%eax
         :               for (i = tamanho - 1; i >= 0; i--)
   13.03 :         400d4f:       jne    400d10 <sort2(int*, int)+0x130>
   }
```

Esta zona demora cerca de 45.16% do tempo total de execução do algoritmo.

Uma vez encontradas estas duas zonas foi consultado o código fonte para verificar se a percentagem de tempo de execução era justificada. Após uma breve análise é possível verificar que ambos os ciclos, correspondentes às secções apresentadas, são de facto aqueles com maior intensidade computacional (ciclos com maior número de iterações, tamanho do vetor, e com várias instruções custosas, tais como acessos a *arrays*, divisões, restos de divisões...).

Código correspondente à zona 1:

```
for (i = 0; i < tamanho; i++)
    bucket[(vetor[i] / exp) % 10]++;
```

Código correspondente à zona 2:

```
for (i = tamanho - 1; i >= 0; i--)
    b[--bucket[(vetor[i] / exp) % 10]]
       = vetor[i];
```

Tendo em conta que estas duas zonas ”ocupam” cerca de 86.78%, ou seja, a maioria do tempo de execução do algoritmo, podemos concluir que são um alvo ótimo de otimização.

XI. Conclusão

Ao longo do curso, várias formas de analisar o desempenho de uma aplicação foram estudadas, como o PAPI ou o Dtrace. Nesse contexto, o Perf é uma ferramenta com um bom equilíbrio entre funcionalidades e facilidade de utilização. Em particular, é bastante robusto em termos de apresentação de resultados e é útil não ser necessário alterar o código fonte. No entanto é sempre necessário ter em conta alguns detalhes, como a implementação dos eventos variar conforme o processador e a frequência escolhida impactar a medição em termos de precisão e *overhead*.

Os *flame graphs* são uma forma de observar facilmente a distribuição de tempo por função e respetivo *stack*, tornando-os bastante úteis nas situações em que a função responsável pela maioria do tempo de execução é chamada múltiplas vezes em zonas diferentes.

Conclusão

Um dos focos principais de várias cadeiras do perfil de Computação Paralela foi a optimização de aplicações. Por sua vez, os trabalhos desenvolvidos na cadeira de Engenharia de Sistemas de Computação abordaram principalmente dois aspetos. Um foi a interação entre a aplicação e o seu ambiente de execução, nomeadamente o sistema operativo e os recursos de *hardware* disponíveis. O outro foco foi as diversas formas de medir e analisar o desempenho das aplicações.

O primeiro trabalho elucidou-nos que fatores externos à aplicação são também um alvo importante de configuração e optimização. Para além disso introduziu-nos a várias ferramentas úteis para avaliar a utilização dos diversos recursos de *hardware*. Para além de saber utilizar as ferramentas também é importante conseguir processar e sumarizar a informação que estas disponibilizam, o que nos levou a aplicar conhecimentos de outras áreas de informática para desenvolvermos ferramentas capazes de o fazer.

O segundo trabalho introduziu-nos não só ao DTrace mas ao conceito de *tracing* em si. *Tracing* provou ser uma forma bastante útil e poderosa de analisar em detalhe o comportamento de uma aplicação.

O terceiro trabalho permitiu-nos aprofundar os conhecimentos sobre o DTrace e explorar novas formas de programação em memória partilhada, quebrando assim a dependência no OpenMP. Estas formas podem ser úteis em situações em que o OpenMP não é a ferramenta mais adequada ou onde a sua interface é demasiado restritiva.

No último trabalho o Perf foi estudado em detalhe. Este permite analisar o desempenho de uma forma simples e rápida sem ser necessário alterar o código fonte.

Com a realização destes quatro trabalhos expandimos o nosso conhecimento sobre formas de analisar o desempenho de aplicações. Algumas destas são mais adequadas para uma abordagem adicional, outras para um estudo mais detalhado. Desta forma ficamos com um arsenal de ferramentas de análise de desempenho mais completo.