

Portfólio de Trabalhos

- Engenharia dos sistemas de Computação -

autor: Daniel Malhadas

I. ÍNDICE

Introdução.....	1
TPC1: Bubble Sort.....	2
TPC2: Análise de Desempenho em Supercomputadores Utilizando NAS Parallel Benchmark (NPB).....	8
TPC3: Desenvolvimento de Programas em DTrace.....	21
TPC4: Análise de Desempenho com Benchmark Ativo e Passivo.....	29
TPC5: Profiling com PERF e DTrace.....	55
Conclusão.....	70

II. INTRODUÇÃO

Neste documento encontra-se um portefólio com 5 trabalhos realizados, ao longo de um semestre, relacionados com Computação Paralela e Distribuída e com análise de performance.

Todos os projetos aqui documentados têm como objetivo desenvolver várias competências nesta áreas, sendo elas:

- **TPC1** - Programação concorrente e optimizações com POSIX threads e OpenMP para o caso concreto de Bubble Sort;
- **TPC2** - Análise de desempenho entre várias máquinas distintas, utiliza-se aqui a NAS Parallel Benchmark de forma a realizar esta análise;
- **TPC3** - Realização de vários scripts com DTrace de forma a desenvolver capacidades no seu uso;
- **TPC4** - Estudo aprofundado de benchmark ativo VS benchmark passivo. É usado IOzone para benchmark passivo e DTrace para benchmark ativo;
- **TPC5** - Realização de um tutorial de PERF que resulta num estudo aprofundado sobre profiling de diferentes aplicações. Usa-se ainda DTrace com o intuito de repetir o profilling realizado por PERF.

Note-se ainda que cada um destes projetos foi inicialmente um documento por si só, logo quando se refere "este documento" ou "este projeto" ao longo de qualquer um destes trabalhos pretende-se referir apenas o trabalho em questão e não o portefólio num todo.

TPC1: Bubble Sort

- Engenharia dos Sistemas de Computação -

autor: Daniel Malhadas

Resumo—O presente documento apresenta um estudo aprofundado sobre três diferentes implementações do algoritmo: Bubble Sort. A primeira é a implementação sequencial clássica, a segunda uma implementação paralela que recorre às diretivas OpenMP para obter o referido paralelismo, a terceira é também uma implementação paralela, mas agora esse paralelismo é obtido com recurso a POSIX threads (Pthreads). Estas três implementações são então comparadas entre si em relação ao tempo de execução para diferentes conjuntos de dados com o intuito de entender quais as que permitem a obtenção de um menor tempo de execução. Os resultados são então organizados em gráficos com intuito também de entender a escalabilidade (relativamente aos conjuntos de dados) para cada implementação.

Index Terms—Bubble Sort, OpenMP, POSIX Threads, Computação Paralela e Distribuída.

I. CONSIDERAÇÕES INICIAIS

A. O que é o algoritmo Bubble Sort

O algoritmo bubble sort é um algoritmo de ordenação simples que baseia o seu funcionamento e o seu nome no fenómeno físico observável ao deixar cair em água parada vários objetos de diferentes densidades. Cada um desses objetos irá comportar-se de forma diferente conforme a sua densidade. Aqueles que tiverem uma densidade superior à da água irão deslocar-se até ao fundo enquanto que os que tiverem uma densidade menor irão subir à superfície, de forma análoga a uma bolha de água que, pela sua baixa densidade sobe ao longo da água. O algoritmo que se pretende estudar emula este processo sendo a estrutura de dados que se pretende ordenar uma metáfora para a estrutura que é a água e, os elementos da estrutura de dados, são os objetos que se fazem cair na água.

B. Caracterização das Máquinas Utilizadas

Durante este projeto usaram-se duas máquinas principais sendo que a validação da correção dos algoritmos implementados terá sido realizada nas duas máquinas, para assim garantir que a correção se mantinha entre arquiteturas distintas. As máquinas usadas foram: um laptop pessoal e o nodo 652 do Cluster SeARCH, no entanto, os resultados apresentados ao longo deste documento serão todos relativos apenas ao nodo 652 do Cluster, pois o laptop pessoal é bastante mais limitado, tornando assim uma comparação direta pouco interessante. As máquinas encontram-

se completamente caracterizadas na seguinte tabela:

Características		Nodo 652
Manufacturer	Intel	
CPU Designation	Dual CPU E5-2670v2	
CPU Microarchitecture	Ivy Bridge	
Clock Frequency	1.2, 2.50GHz	
RAM	64GB	
Cores	20 with 40 threads (2 per core)	
Peak FP Performance	400GFlops/s	
Cache	L1d: 32kB, L1i: 32kB L2: 256kB, L3: 25600kB	
Memory Bandwidth	59.7GB/s	
Características		Laptop Pessoal
Manufacturer	Intel	
CPU Designation	Dual CPU i5 4200U	
CPU Microarchitecture	Haswell	
Clock Frequency	1.6, 2.6GHz (with turbo clock speed)	
RAM	4GB	
Cores	2, 4 threads	
Peak FP Performance	41.6GFlops/s	
Cache	L1i: 32KB, L1d: 32KB, L2: 0.5MB, L3: 3MB	
Memory Bandwidth	2 channels, max 25.6GB/s	

Esta informação foi obtida apartir de várias fontes distintas. Sendo elas: - o comando Unix **lscpu**; - o site **cpu-world.com**; - o site **ark.intel.com**.

C. Conjuntos de Dados

O algoritmo bubble sort aplica-se a uma lista de valores que se entende estarem de alguma forma desordenados pretendendo-se então ordená-los. Para a obtenção das análises de performance apresentadas no decorrer deste documento teve então de se pensar em conjuntos de dados para teste que fossem relevantes. Considerou-se que os valores em si a ordenar não são de todo o fator relevante, por esse motivo, todos os testes foram realizados com valores aleatórios. No entanto o espaço que estas listas ocupam em memória é já relevante e, por esse motivo, decidiu-se que a diferença entre cada conjunto de dados deveria ser o seu tamanho em memória. Foram então escolhidos os seguintes quatro conjuntos de dados:

1 - Um Conjunto de dados que encha por completo a Cache L1

O nodo 652 do Cluster descrito na sub-secção anterior tem L1d: 32kB e L1i: 32kB, apenas nos interessa dados, por isso temos um total de 32kB na Cache L1 para os nossos dados. Cada valor na lista será representado por um inteiro de 32 bits (4Bytes), logo este nível da cache poderá conter $32000B/4B=8000$ valores. Mas não terminamos por aqui, podemos ainda ter em atenção o tamanho de uma linha da cache. Neste caso cada linha terá 64B, por isso cabem 16 valores por linha, logo o número total de valores na nossa lista a ordenar deve ser um múltiplo de 16 de forma a maximizar o proveito tirado da localidade espacial. $8000/16 = 500$, logo

a dimensão ideal para a lista a ordenar é 8000 valores.

Os restantes conjuntos de dados foram calculados de maneira análoga.

2 - Um Conjunto de dados que encha por completo a Cache L2

a dimensão ideal para a lista a ordenar é 64000 valores.

3 - Um Conjunto de dados que encha por completo a cache L3

a dimensão ideal para a lista a ordenar é 6400000 valores.

4 - Um Conjunto de dados que não caiba em nenhum dos níveis de Cache anteriores e force o CPU a carregá-lo da DRAM

Para este conjunto de dados precisamos somente de uma lista que seja significantemente maior que 6400000 elementos. Encontrou-se então um valor relevante que é também múltiplo do número de valores que cabem numa linha da cache. Uma boa dimensão é 7200000.

II. DIFERENTES IMPLEMENTAÇÕES

A. Implementação Sequencial

```
void Bubble_sort(int a[], int n) {
    int list_length, i, temp;
    for (list_length = n; list_length >= 2;
        list_length--) —2
        for (i = 0; i < list_length-1; i++) {
            if (a[i] > a[i+1]) {
                temp = a[i]; —1
                a[i] = a[i+1];
                a[i+1] = temp;
            }
        } /* Bubble_sort */
}
```

O algoritmo bubble sort sequencial clássico consiste em comparar cada par de elementos adjacentes na lista a ordenar e trocá-los se estiverem na ordem errada, esta troca está visível no ponto 1 circundado no código. Esta passagem pela lista é repetida até que mais nenhuma troca seja necessária. Uma possível otimização, presente no ponto 2, é diminuir o número de elementos a ordenar por iteração do ciclo exterior. Isto porque no fim da primeira iteração é garantido que o maior elemento fique na posição do fim da lista, na segunda iteração é garantido que o segundo maior elemento fique na posição antes da última, e por aí adiante. Logo entende-se que os elementos do fim já estão ordenados a cada iteração e podem portanto ser ignorados nas iterações posteriores.

Este algoritmo tem complexidade média e de pior caso (n^2), sendo n o número de elementos a ordenar. Quando a lista já está ordenada (melhor caso), a complexidade é somente $O(n)$. Note-se também que este algoritmo é particularmente ineficiente, comparado com outros, quando n é demasiado grande ou quando a lista está completamente desordenada, sendo vantajoso mais quando o número de elementos fora de ordem é bastante reduzido.

B. Implementação Paralela - OpenMP

```
void Bubble_sort_OMP(int a[], int n) {
    int list_length, block_count, i, temp;
    int block_size, lower, upper, block;
    block_count = num_threads;
    omp_lock_t m[block_count];
    block_size = n/block_count;

    for(i=0;i<block_count;i++)
        omp_init_lock(&m[i]);

    omp_set_num_threads(num_threads);
    #pragma omp parallel private(list_length,
                                block, lower, upper, i, temp)
    {
        for(list_length=block_size;
            list_length>=1;list_length--)
            for(block=0;block<block_count;block++) {
                lower=block_size*block;
                if(block==block_count-1) upper = n;
                else upper = block_size*(block+1);
                if(block==0) omp_set_lock(&m[0]);
                for(i=lower;i<upper-1;i++) {
                    if(a[i]>a[i+1]) swap(a+i,a+i+1);
                    if(block==block_count-1) {
                        omp_unset_lock(&m[block]);
                        break;
                    }
                    omp_set_lock(&m[block+1]);
                    if(a[i]>a[i+1]) swap(a+i,a+i+1);
                    omp_unset_lock(&m[block]);
                }
            }
    } /* Bubble_sort_OMP */
}
```

Na explicação do subcapítulo anterior referimos que o algoritmo bubble sort consiste em dois ciclos, sendo um interior ao outro. Foi então necessário decidir qual dos ciclos paralelizar. Dois critérios foram tidos em conta: já que estamos a usar paradigmas fork&join que funcionam com base em "pools" de threads, o ciclo a paralelizar, idealmente, deve ser aquele que cause o mínimo de overhead relativo ao fork&join das threads; E o ciclo a paralelizar deve, idealmente, ser aquele cujas iterações dependem menos umas das outras. Quanto ao primeiro critério é fácil entender que o ciclo a paralelizar deveria ser o exterior visto que as threads serão lançadas somente uma vez, antes do ciclo, e juntas de novo somente no fim do ciclo ao invés de serem lançadas e juntas a cada iteração. Desta forma o overhead que o fork&join poderia causar torna-se praticamente nulo e deixa de ser um problema. Quanto ao segundo critério, observamos que as iterações de ambos os ciclos são dependentes das iterações anteriores. O ciclo exterior depende da iteração anterior para, por exemplo, saber que o último elemento da iteração anterior já se encontra ordenado (otimização referida na subsecção anterior). No entanto a dependência no ciclo interior é ainda mais forte, porque numa dada iteração tem-se a intenção de pegar no elemento que trocou de posição na iteração anterior (se tiver trocado) e verificar se deve ser trocado novamente para a posição seguinte. Se este seguimento se quebrar, o efeito da bolha que sobe pela água mencionado no primeiro capítulo deixa de existir e já não se pode garantir também que o elemento do fim

fique ordenado a cada iteração do ciclo exterior. Ambos os ciclos são problemáticos, mas já vimos em cima vantagens de parallelizar o exterior.

De forma a reduzir a dependência entre iterações no ciclo exterior, dividiu-se a lista em T partes, sendo este T igual ao número de threads a utilizar. Desta forma cada thread poderá trabalhar em cada uma destas partes/blocos de forma independente. Estes blocos devem ser operados por somente uma thread de cada vez de forma a eliminar a possibilidade de "data races". Para evitar que outra thread entre no mesmo bloco enquanto este está a ser computado por outra, a thread que entrou primeiro deve recorrer ao uso de um mutex para bloquear acesso ao bloco atual, vemos isto no ponto 1 circundado no código. Reparamos então que há a necessidade de um diferente mutex para cada diferente bloco. Note-se ainda que, para manter a correção do algoritmo, cada thread deve avançar de bloco para bloco por ordem, percorrendo assim ordeiramente o array, indo libertando o lock que faz para cada bloco no momento em que acaba de o percorrer e adquire o lock para o próximo bloco. O facto de apenas libertar um bloco quando adquire o próximo permite que, caso seja necessário, realize um swap entre os elementos nas fronteiras dos blocos. Desta forma os maiores elementos são realmente levados para o fim a cada iteração, sendo que enquanto uma thread percorre um bloco mais avançado, as outras estão ocupadas nos anteriores. Temos então uma estratégia que funciona como um pipeline onde, por cada bloco computado, uma iteração é terminada. O fator limitante desta paralelização torna-se então o tamanho de cada bloco, tendo em conta que cada bloco diminui com o aumento de threads estima-se que este algoritmo seja bastante escalável ao nível das threads. Segue-se uma imagem ilustrativa do algoritmo:



Como se vê na imagem, quando uma thread termina de percorrer o último bloco deve adquirir o primeiro e voltar lá. Note-se que agora é importante que a thread liberte de imediato o lock mal acabe o último bloco de forma a evitar um deadlock, caso contrário a thread no primeiro bloco nunca seria capaz de avançar e por isso nunca o libertaria. Desta vez não tem problema libertar o lock de imediato, pois como é o bloco não precisamos de nos preocupar com o elemento na fronteira. Isto está visível no ponto 3 circundado no código. É ideal que cada bloco tenha o mesmo número de elementos,

para assim haver um bom balanceamento de computação entre threads (embora por vezes alguns blocos possam ser menos custosos, como em situações onde há muito poucos swaps), mas por vezes o número de elementos no array não é divisível pelo número de threads disponível. Uma possível solução para este problema encontra-se no ponto 2 circundado no código. Nesse ponto vemos que, nessa situação, os valores "extra"são computados no último bloco. Não é expectável que este pormenor seja muito negativo na performance pois o número de elementos "extra"será sempre menor que o número de threads e por isso demasiado pequeno para realmente se considerar que quebra o balanceamento da computação entre threads.

Esta estratégia permite que o ciclo interior tenha uma complexidade complexidade de $O(1)$ no caso em que temos n ou mais threads. Como o ciclo exterior continua com uma de $O(n)$ vemos que este algoritmo tem uma complexidade média de $O(n)$.

C. Implementação Paralela - POSIX Threads

```

void* sort_thread(void *param) {
    int list_length, block_count, i, temp;
    int block_size, lower, upper, block;
    block_count = num_threads;
    block_size = num/block_count;
    pthread_mutex_t *m = thread_mutexes;

    for(list_length = block_size; list_length>= 1; list_length--)
        for(block=0;block<block_count;block++) {
            lower = block_size *block;
            if(block==block_count-1) upper = num;
            else upper = block_size*(block+1);
            if(block==0)
                pthread_mutex_lock(&m[0]);
            for(i=lower;i<upper-1;i++)
                if(arr[i]>arr[i+1])
                    swap(arr+i,arr+i+1);
            if(block==block_count-1)
                pthread_mutex_unlock(&m[block]);
                break;
        }
        pthread_mutex_lock(&m[block+1]);
        if(arr[i]>arr[i+1])
            swap(arr+i,arr+i+1);
        pthread_mutex_unlock(&m[block]);
    }
    return NULL;
} /*sort_thread*/

void Bubble_sort_pthreads() {
    int ts[num_threads];
    for(thread=0;thread<num_threads;thread++) {
        ts[thread]=thread;
        ① pthread_create(&thread_handles[thread],NULL,
                        sort_thread,NULL);
    }
    for(thread=0; thread<num_threads;thread++)
        ② pthread_join(thread_handles[thread],NULL);
    for(thread=0; thread<num_threads; thread++)
        pthread_mutex_destroy(&thread_mutexes[thread]);
} /*Bubble_sort_pthreads*/

```

a implementação usando POSIX Threads com recurso à biblioteca pthreads.h mantém exatamente a mesma estratégia de parallelizar o ciclo exterior e de dividir a lista em blocos. Ao observar o código note-se que todas as variáveis não definidas no excerto apresentado são globais, podendo assim

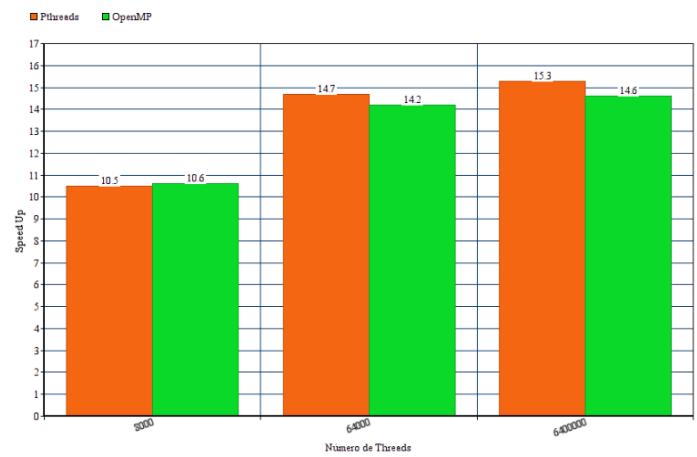
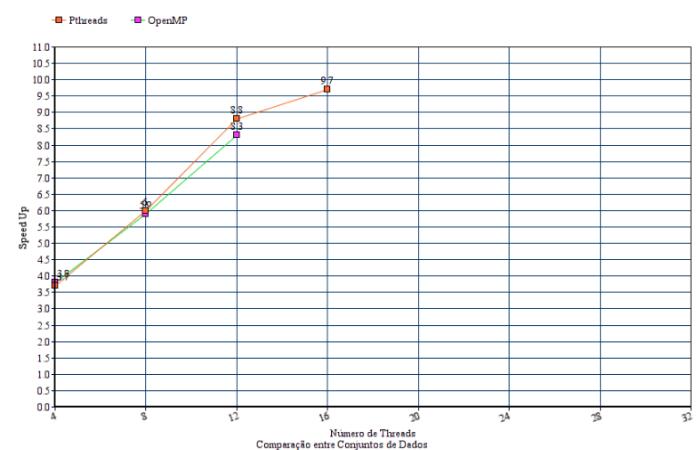
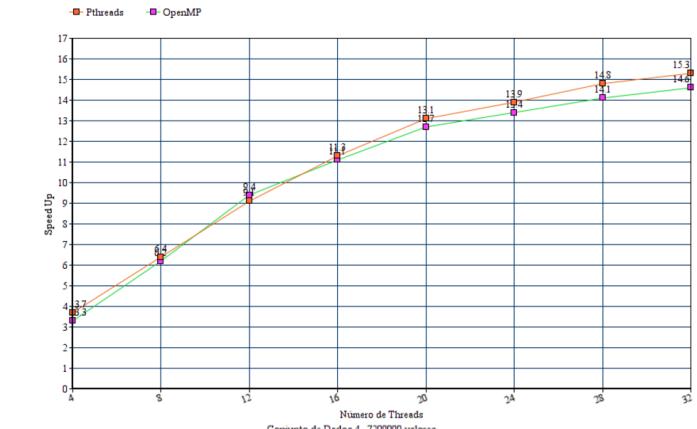
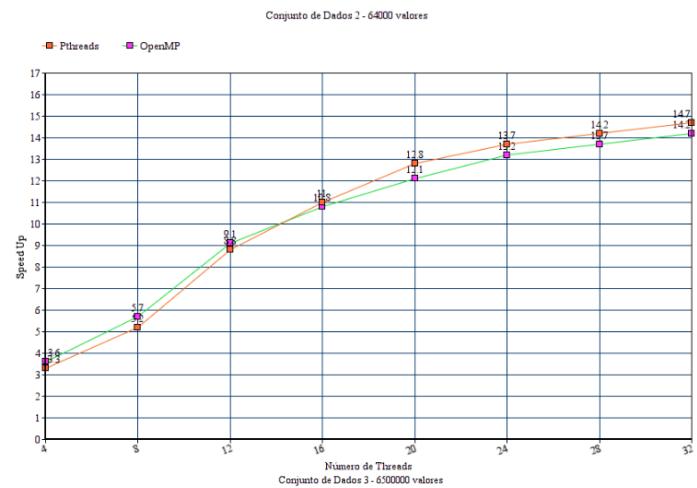
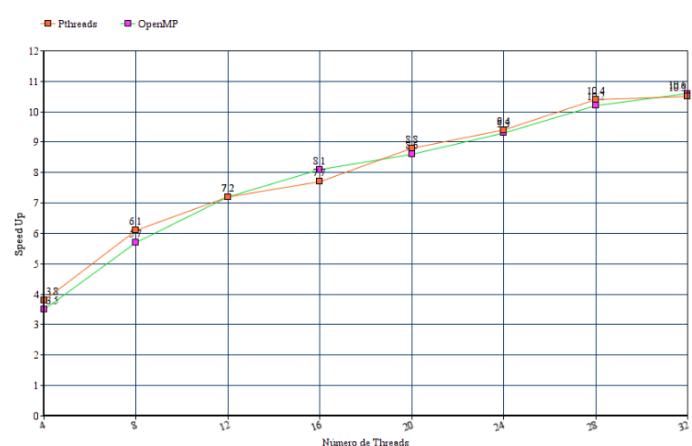
ser partilhadas entre diferentes threads. Basta olhar para o tamanho desta implementação para se notar que a natureza de baixo nível da biblioteca `pthreads.h` que implementa as POSIX threads nos permite um controlo muito maior sobre o que de facto está a acontecer, mas ao mesmo tempo aumenta bastante a complexidade do código em si. Nesta implementação temos de criar explícitamente uma "pool" de threads e temos também de arranjar um mecanismo para controlo de acesso à memória partilhada que, assim como na versão OMP, foi um conjunto de mutex. No código são visíveis duas funções. A função **Bubble_sort_pthreads** é responsável pela criação e posterior junção das threads que irão executar paralelamente a função **sort_thread** sendo que esta implementa os ciclos do nosso algoritmo em si. O ponto circundado 1 mostra esta criação e o 2 a junção. Note-se ainda que, como se vê no ponto circundado 3, optou-se por não passar nenhum argumento à função que as threads irão executar. Poder-se-ia ter optado por passar um apontador para uma estrutura com todos os dados necessários ao invés de ter que manter várias variáveis globais e essa opção seria bastante mais "limpa" e agradável visualmente, no entanto entendeu-se que o impacto na performance inerente da necessidade de alocar memória para essa referida estrutura e de a operar em cada thread não se justificava.

Como o algoritmo é exatamente igual ao anterior com OMP é possível observar no código as linhas que foram marcadas com pontos circundados no subcapítulo anterior e notar a forma como se comportam exatamente da mesma forma com o mesmo raciocínio implícito. Esta semelhança mantém-se também na complexidade que, pelas mesmas razões que na versão anterior, se mantém com complexidade média de $O(n)$.

III. MEDIÇÕES DE PERFORMANCE

De forma a obter os resultados apresentados ao longo deste capítulo o código foi compilado com recurso a `gcc` com a flag de optimização `-O3` e a flag relativa ao OMP `-fopenmp`. Com o intuito de medir tempos de execução utilizou-se a biblioteca `time.h` e usou-se a expressão `(stop-start)/(CLOCKS_PER_SEC / 1000)` permitindo visualizar o tempo em milisegundos, sendo esse valor guardado sobre a forma de um double, permitindo uma precisão de 64 bits.

Apresentam-se agora gráficos que nos permitem observar os speed ups obtidos:



Cada um destes gráficos representa os speed ups obtidos com as implementações paralelas (em relação à implementação sequencial). Os primeiros 4 gráficos são relativos a um só conjunto de dados e podem ser usados para comparar as implementações usando esse específico conjuntos de dados. O último gráfico é um gráfico de barras que apresenta os melhores speed ups para cada conjunto de dados com cada implementação, permitindo assim uma comparação direta entre os três.

Os valores apresentados foram obtidos da seguinte forma: primeiro mediu-se o tempo da versão sequencial, chamar-lhe-emos **Tseq**, de seguida, para o número de threads pretendido (imaginemos que seja **NumT**), mediu-se o tempo da versão OpenMP um total de 10 vezes. Desses 10 tempos fez-se a média aritmética de todos os valores que não se desviassem mais do que 10% do melhor tempo medido, chamaremos a esta média **Tomp**. O mesmo se repetiu para a versão Pthreads e ao tempo resultante desta versão chamaremos **Tpth**. Temos então que o speed up para **NumT** threads da versão OpenMP = **Tomp/Tseq** e para a versão Pthreads = **Tpth/Tseq**. Este processo repetiu-se para cada número de threads pretendido e para cada conjunto de dados (definidos no primeiro capítulo). Nota-se olhando para os gráficos que apenas se mede os speed ups até um máximo de 32 threads. Tal deve-se ao facto de a máquina usada ter apenas 20 cores físicos e, não haver grande relevância em ver os resultados obtidos com recurso a hyperthreading, pois é expectável que estes sejam piores. Apenas se mediu alguns valores depois das 20 threads com o intuito de verificar se realmente se notava alguma descida de speed up. Nota-se ainda que nos gráficos relativos ao último conjunto de dados nem todos os speed ups foram calculados. Isto acontece porque o tempo de execução com essas características se torna tão elevado que não foi possível obter resultados em tempo útil para assim fazer os cálculos necessários.

Observando agora os resultados obtidos notamos que o conjunto de dados sobre o qual se obteve um menor speed up foi o conjunto de dados 1 que contém 8000 valores. Por ser um conjunto de dados demasiado pequeno estima-se que o overhead causado pela espera das threads por um bloco que ainda não tenha sido libertado por outra foi demasiado elevado, isto é, os cálculos efetuados por thread não compensaram o tempo que teve que esperar antes de poder avançar para um novo bloco. Com 8000 valores, para 4 threads teremos 4 blocos com 2000 valores cada. Sabendo que ao iterar por estes blocos em muitas iterações não irão ocorrem swaps (ou seja, são praticamente instantâneas) podemos concluir que cada thread faz um trabalho mínimo, mas que mesmo assim ainda compensa um pouco já que se obtém algum speed up comparativamente com a versão sequencial. Há medida que o número de threads aumenta vemos que o speed up é cada vez menos significativo. Isto é normal, já que, por exemplo, com 20 threads cada thread fará à volta de $8000/20 = 400$ iterações por bloco, onde muitas serão sem swap. É expectável que se se continuasse a aumentar significantemente o número de threads continuariam a observar perdas graduais de speed up podendo até chegar a um ponto onde as threads causam um overhead tão escusado que teríamos uma perda de speed up

em relação à versão sequencial.

Nota-se depois que o conjunto de dados com o qual se obteve o maior speed up foi o terceiro, com 6400000 valores. É expectável que, com este algoritmo se obtenham melhores speed ups para maiores conjuntos de dados (e este conjunto de dados é o maior sobre o qual se conseguiu fazer todas as medições) já que são os que compensam mais o overhead da gestão da memória partilhada. Observamos então que estas implementações paralelas são **escaláveis a nível dos dados** sendo pior para menores conjuntos de dados e, o facto de que realmente se obteve melhores speed ups com maiores conjuntos de dados prova que houve um bom balanceamento da computação entre threads, sendo que um mau balanceamento iria puxar os speed ups para mais perto do valor sequencial. Como se disse no capítulo anterior, o fator limitante deste algoritmo é o tamanho dos blocos, quanto menores mais rápido será o algoritmo e maior o speed up, ou seja, o algoritmo também é **escalável a nível das threads**, no entanto há que entender que deve haver também um limite para quanto pequenos estes blocos devem ser, visto que, como vemos por observação dos speed ups obtidos com o conjunto de dados mais pequeno, blocos demasiado pequenos irão causar overhead muito mais difícil de compensar. Portanto aumentar o número de threads será vantajoso na maior parte dos casos, mas apenas uma análise com mais conjuntos de dados e um maior número de threads poderia realmente identificar um ratio interessante entre valores a ordenar e threads que permitissem alcançar o pico do speed up. De qualquer forma, qualquer ratio encontrado seria sempre uma estimativa crassa pois nunca temos forma de dividir de maneira justa a computação entre threads, já que não temos maneira de à priori saber que iterações são custosas (fazem swap) e quais não o são. Por isso é perfeitamente possível que haja uma thread que não faça nenhum swap e por isso termine de imediato enquanto outra encontre swaps em todas as posições por onde passa e por isso demore vários milisegundos com computação que deveria ser dividida pelas threads igualmente. Esta é uma segunda limitação deste algoritmo e uma provável razão para que haja speed ups mais acentuados com menor número de threads já que, quanto maior o número de threads maior é a probabilidade de que alguma delas não faça nada que seja realmente relevante num bloco que percorre. Inputs grandes são os únicos capazes de disfarçar esta falha já que mesmo com muitas threads os blocos serão grandes. O facto de com um elevado número de threads se estar a usar hyperthreading (a partir das 20 threads) também poderá influenciar e ser uma razão pela qual a curva de speed up deixa de ser tão acentuada por volta das 20 threads.

Por último vemos que das duas implementações paralelas, aquela que forneceu os maiores speed ups foi a implementação que usa Pthreads. Isto era o que se esperava visto que o uso da biblioteca pthreads.h permite a criação de um código mais "low-level" do que o uso de diretivas OpenMP. Isto permite-nos fazer escalonamento das threads diretamente sendo que o programa não terá que incorrer num overhead de ter de determinar o escalonamento das mesmas em tempo real.

IV. CONCLUSÕES

Ao longo deste documento explicaram-se três possíveis implementações para o algoritmo bubble sort e estudaram-se as limitações do mesmo comparando-se as diferentes implementações. Afirma-se que os speed ups obtidos foram bons e aquilo que se esperava, mas pecando apenas talvez por um pequeno número de conjuntos de dados para teste. Com mais tempo talvez fosse relevante estudar mais inputs intermédios (nem grandes nem pequenos) que permitissem melhor concluir algo sobre as implementações nessa fase até porque afirmar sem experimentar é em si só uma grande falácia e como tal mais experimentação permitir-nos-ia fazer afirmações mais certas e fundamentadas. poder-se-ia fazer uma maior quantidade de testes até ao ponto em que se pudesse prever uma ligação entre o número de valores a ordenar e o número de threads disponível de forma a poder obter o pico de speed up. Para já é expectável que essa relação seja possível de se encontrar e, não fosse este um algoritmo pior que outros com o mesmo propósito (como o quicksort, mergesort, etc...), poderia até ser algo de interessante a fazer.

Por fim, entende-se que as estimativas dos resultados foram bem sucedidas e confirmadas na sua maioria com os testes realizados. Pensa-se também que, embora poucos, os conjuntos de dados cubriram um terreno grande o suficiente para se ter no mínimo um panorama geral do comportamento do algoritmo implementado. Por estas razões considera-se que este estudo foi completado com sucesso.

TPC2: Análise de Desempenho em Supercomputadores Utilizando NAS Parallel Benchmark (NPB)

- Engenharia dos sistemas de Computação -

autor: Daniel Malhadas

Resumo—O presente documento apresenta um estudo aprofundado sobre análise de desempenho em supercomputadores, em particular, alguns nodos específicos do Cluster SeARCH. Para esta análise de desempenho foi utilizado o pacote NAS Parallel Benchmark (NPB) de forma a corretamente poder chegar a conclusões relevantes relativas ao desempenho de cada nodo do cluster mencionado e poder, de forma realista, comparar também os nodos entre si.

Index Terms—NAS Parallel Benchmark, Análise de Desempenho, Computação Paralela e Distribuída.

I. INTRODUÇÃO

A. Contextualização e Motivação

Com este projeto pretende-se recorrer ao uso de NAS Parallel Benchmarks (NPB) de forma a analisar o desempenho de nodos do cluster SeARCH. Estas análises consistem em executar repetidamente os mesmos benchmarks fazendo pequenas variações (por exemplo a ferramente de compilação, a rede para comunicação entre processos, o tamanho dos dados, etc) em diferentes máquinas de forma a poder-se entender os pontos fortes e fracos de cada máquina e também poder comparar esses mesmos pontos entre diferentes máquinas podendo assim com relativa certeza poder determinar que máquinas seriam mais indicadas para determinados tipos específicos de problemas e porquê.

Tendo em conta que o número de possibilidades de testes possíveis de realizar é praticamente infinita e sendo todos esses testes relativamente relevantes houve a necessidade de limitar um pouco o domínio dos testes a realizar. Esse domínio será explicitado ao longo deste capítulo nos próximos subcapítulos. No entanto, mesmo limitando o domínio, temos ainda uma enorme quantidade de dados a organizar, assim sendo o uso de comandos auxiliares como iostat, mpstat, top e dstat e de ferramentas como GNU plot permite-nos um tratamento mais rápido e mais eficiente dos dados obtidos com os benchmarks mencionados e tornam-se então uma necessidade para uma análise de desempenho aprofundada.

B. O que são as NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB) é um pacote com um pequeno conjunto de programas idealizados para avaliar o desempenho de supercomputadores paralelos. Estas benchmarks derivam de aplicações na área *Computational Fluid Dynamics*

(CFMD) e consistem em 5 kernels e 3 pseudo-aplicações na especificação original do NPB 1. Os tamanhos dos problemas NPB são pré-definidos e organizados por diferentes classes consoante o tamanho.[1]

II. CONSIDERAÇÕES INICIAIS

A. Caracterização das Máquinas Utilizadas

Com o intuito de realizar análises de desempenho em nós específicos do cluster SeARCH, teve-se o cuidado de observar calmamente a grande quantidade de nós disponíveis e assim escolher nós relevantes para serem analisados em conjunto. Isto porque certas combinações de nodos não seriam boas para este trabalho, pois não há grande interesse em escolher, por exemplo, nós com microarquiteturas semelhantes, já que estaremos a comparar análises de desempenho obtidas em duas máquinas que operam de forma semelhante sendo que a sua comparação não nos iria trazer nada de novo.

Houve ainda um cuidado especial de, embora se terem escolhido nodos com microarquiteturas distintas, de escolher nodos com capacidades semelhantes essencialmente a nível da frequência do processador e número de cores físicos. Isto porque assim a análise será mais "justa", visto que, por exemplo, o desempenho de uma aplicação com um número de threads que excede bastante o número de cores físicos numa máquina será em princípio menor do que numa outra máquina onde não excede o número de cores físicos, fazendo com que a comparação não seja "justa" já que a máquina com menos cores físicos pode realmente ser melhor que a outra para um menor número de threads, mas isto poderá estar demasiado ofuscado ao observar o panorama geral.

Pelas razões referidas optaram-se então pelas máquinas do cluster SeARCH caracterizadas nas tabelas seguintes:

Designação	Nodo 652
Fabricante	Intel
Modelo do CPU	Dual CPU E5-2670v2
Microarquitetura do CPU	Ivy Bridge
Frequência do Clock	2.50 GHz
#Cores	20, com 40 threads (2 por core)
Cache	L1d: 32kB, L1i: 32kB, L2: 256kB, L3: 25600kB
Largura de Banda de acesso à memória	59.7GB/s
Memória RAM	64GB
Rede para Comunicação	Gigabit Ethernet e Myrinet 10Gbps

Designação	Nodo 431
Fabricante	Intel
Modelo do CPU	Dual CPU X5650
Microarquitetura do CPU	Nehalem
Frequência do Clock	2.67 GHz
#Cores	12, com 24 threads (2 por core)
Cache	L1d: 32kB, L1i: 32kB, L2: 256kB, L3: 12288kB
Largura de Banda de acesso à memória	32GB/s
Memória RAM	48GB
Rede para Comunicação	Gigabit Ethernet e Myrinet 10Gbps

Esta informação foi obtida apartir de várias fontes distintas. Sendo elas: - o comando Unix **lscpu**; - o site **cpu-world.com**; - o site **ark.intel.com**; - o site http://search6.di.uminho.pt/wordpress/?page_id=55;

Note-se que, embora apenas se tenham escolhido dois nodos do cluster, estes dois nodos permitem ter em conta o panorama geral da maioria dos nodos presentes no cluster, já que, depois de consultar o site http://search6.di.uminho.pt/wordpress/?page_id=55 se pode observar que a microarquitetura mais comum nos nodos é Ivy Bridge seguida da Nehalem, por isso os nodos escolhidos são representativos de grande parte do domínio de nodos presentes no cluster.

Teve-se também o cuidado que, os nodos utilizados, permitissem ambos o uso de Gigabit Ethernet e de Myrinet para assim poderem ser comparadas as duas máquinas quanto à comunicação entre processos com cada um desses tipos de rede.

B. Caracterização dos Kernels utilizados

As 8 benchmarks especificadas no NPB 1 consistem em 5 kernels e 3 pseudo-aplicações. Visto que o objetivo deste estudo é o de comparar o desempenho em arquiteturas distintas com diferentes paradigmas de memória, diferentes compiladores, diferentes formas de comunicação entre máquinas, etc, temos maior interesse nos kernels do que nas pseudo-

aplicações. Por esse motivo apenas analisamos os kernels. Apresenta-se de seguida cada um dos 5 kernels especificados no NPB 1 junto com uma breve descrição de cada:[1]

IS - Integer Sort

Kernel de ordenação de interior inteiros usando o algoritmo bucket sort[2]. Como se trata de uma ordenação sabemos implicitamente que consiste em acessos aleatórios à memória sendo assim possível testar comunicação irregular ora via memória partilhada ora via memória distribuída. Permite ainda perceber o desempenho do tratamento de número inteiros.

EP - Embarassingly Parallel

Kernel embarrasosamente paralelo que gera variáveis Gaussianas aleatórias usando o *Marsaglia Polar method*[2]. esta geração de variáveis Gaussianas traduz-se em instruções muito paralelizáveis sendo então de esperar que estas escalem bastante com o número de threads. Podemos ainda perceber o desempenho de números de virgula flutuante já que as variáveis Gaussianas referidas são isso mesmo.

CG - Conjugate Gradient

Kernel que estima o menor autovalor de uma grande matriz simétrica positiva usando a iteração inversa com o método *Conjugate Gradient* como uma sub-rotina para resolver sistemas de equações lineares[2]. Aqui iremos ter acesso irregular à memória e comunicação também irregular o que poderá levar a uma má escalabilidade a nível de threads/processos e memória. É relevante ver como cada nodo se irá comportar para um kernel deste tipo.

MG - Multi-Grid on a sequence of meshes

Kernel que aproxima a solução de uma equação de Poisson discreta tridimensional usando o método *V-cycle multigrid*[2]. Baseia-se em comunicações de curto e longo alcance numa grelha sendo também bastante intensivo a nível de memória. Espera-se então que a comunicação intensiva necessária para este kernel seja um fator limitante para a escalabilidade a nível de processos e memória em diferentes nodos, podendo assim talvez o paradigma de memória partilhada ter vantagens sobre o paradigma de memória distribuída.

FT - discrete 3D fast Fourier Transform

Kernel que resolve uma equação diferencial tridimensional parcial usando a transformada rápida de Fourier[2]. O funcionamento deste kernel tem por base "comunicação de todos-para-todos" sendo então bom para estudar o desempenho quando o fator limitante é a comunicação. Embora haja outros kernels que também tenham a comunicação entre processos como um fator limitante, este é o único onde a comunicação é o único fator limitante. No entanto, como se sente que a comunicação já é suficientemente estudada nos restantes kernels, optou-se por deixar este de parte, focando-se mais o estudo nos 4 anteriores.

Na descrição de cada kernel indica-se as razões pelas quais é relevante estudá-lo. Dos 5 apresentados apenas se abdicou do estudo do FT, pelas razões indicadas na sua descrição.

C. Caracterização dos Conjuntos de Dados Utilizados

Como referido anteriormente, os conjuntos de dados utilizados para os NPB consistem em tamanhos pré-definidos que se dividem em várias classes identificadas por letras distintas. As classes referidas encontram-se caracterizadas para cada kernel na seguinte tabela:[3]

Benchmark	Parameter	Class S	Class W	Class A	Class B	Class C	Class D	Class E
CG	no. of rows	1400	7000	140000	75000	150000	1500000	9000000
	no. of nonzeros	7	8	11	13	15	21	26
	no. of iterations	15	15	15	75	75	100	100
	eigenvalue shift	10	12	20	60	110	500	1500
EP	no. of random-number pairs	2 ¹⁴	2 ¹⁵	2 ¹⁸	2 ¹⁰	2 ¹²	2 ¹⁶	2 ¹⁰
	grid size	64 × 64 × 64	128 × 128 × 128	256 × 256 × 256	512 × 512 × 512	2048 × 1024 × 1024	4096 × 2048 × 2048	
FT	no. of iterations	6	6	6	20	20	25	25
	no. of keys	2 ¹⁴	2 ¹⁰	2 ¹⁸	2 ¹⁸	2 ²⁷	2 ²¹	
	key max. value	2 ¹¹	2 ¹⁴	2 ¹⁹	2 ²¹	2 ²⁴	2 ²⁷	
IS	grid size	32 × 32 × 32	128 × 128 × 128	256 × 256 × 256	512 × 512 × 512	1024 × 1024 × 1024	2048 × 2048 × 2048	
	no. of iterations	4	4	4	20	20	50	50
	no. of keys	2 ¹⁴	2 ¹⁰	2 ¹⁸	2 ¹⁸	2 ²⁷	2 ²¹	
MG	grid size	32 × 32 × 32	128 × 128 × 128	256 × 256 × 256	512 × 512 × 512	1024 × 1024 × 1024	2048 × 2048 × 2048	
	no. of iterations	4	4	4	20	20	50	50

Considera-se que o espaço ocupado pelos conjuntos de dados em memória são o fator mais relevante e não os valores dos dados em si, por isso de forma a escolher tamanhos que sejam realmente relevantes teve-se em conta os seguintes critérios relacionados com o espaço em memória:

1 - Conjuntos de dados que caibam na Cache L1 das máquinas utilizadas

As máquinas usadas e descritas anteriormente têm L1d: 32kB e L1i: 32kB, apenas nos interessa dados, por isso temos um total de 32kB na Cache L1 para os nossos dados, logo este nível da cache poderá conter $32000B/4B=8000$ valores inteiros ou de vírgula flutuante de 4B cada. Mas não terminamos por aqui, podemos ainda ter em atenção o tamanho de uma linha da cache. Neste caso cada linha terá 64B, por isso cabem 16 valores por linha, logo o número ideal valores no conjunto de dados deve ser um múltiplo de 16 de forma a maximizar o proveito tirado da localidade espacial. $8000/16 = 500$, logo a dimensão ideal seria de 8000 valores de 4B ou 4000 valores de 8B (double). No entanto, note-se que não se encontrou uma classe cujos conjuntos de dados para os diferentes kernels caibam todos dentro da cache L1, não foi possível então escolher nenhum conjunto de dados com estas características.

2 - Conjuntos de dados que caibam na Cache L2 das máquinas utilizadas

Calculando da mesma forma vemos que, para a cache L2, o número ideal de valores seria 64000 inteiros ou 32000 doubles. Na classe S vemos que há alguns conjuntos de dados que cabem nesta cache L2.

3 - Conjuntos de dados que caibam na cache L3 das máquinas utilizadas

Calculando da mesma forma, para o nodo com menor cache L3 (o nodo 431 com cache L3 de 12288kB), vemos que o número ideal de valores seria 3072000 inteiros ou 1536000 doubles. Na classe S, W, A, B e C vemos que há alguns conjuntos de dados que cabem nesta cache L3.

4 - Conjuntos de dados que não caibam em nenhum dos níveis de Cache anteriores e que forcem o CPU a carregá-lo da DRAM

Para este conjunto de dados precisamos somente de conjuntos que sejam significantemente maiores que a cache L3. A, B, C, D e E têm vários conjuntos de dados que verificam esta

condição.

Concluímos agora que todas as classes têm conjuntos de dados que são de alguma forma relevantes, no entanto, utilizá-las todas não seria de todo relevante. Por exemplo, as classes S e W enquadraram-se as duas num escopo com conjuntos de dados pequenos em comparação com as outras. São demasiado parecidas e como tal, o uso das duas ao mesmo tempo seria redundante. Depois de alguns testes para se determinar qual seria mais relevante a usar concluiu-se que ambas são demasiado pequenas em comparação com as restantes e, por essa razão, tornam-se obsoletas para os nodos utilizados que as correm demasiado rápido, não obtendo dados particularmente interessantes. O mesmo se conclui para a classe D e E, mas pela razão oposta. Ambas se enquadraram num escopo demasiado grande em comparação com as restantes. São tão grandes até que, ao fim de alguns testes se previu que não se obteriam bons resultados em tempo útil se estas fossem utilizadas. Por estes motivos optou-se por utilizar as classes **A, B e C**, sendo que estas classes respeitam, dos critérios mencionados a negrito anteriormente, o critério 2, 3 e 4. Ao "desistir" das outras classes não perdemos nada no panorama geral, visto que nenhuma classe respeitava o critério 1, ou seja, continuamos a ter conjuntos de dados relevantes para a 2, 3 e 4 apesar de apenas utilizarmos as classes A, B e C.

De forma a usar as classes selecionadas com os kernels selecionados observou-se a seguinte tabela com as distribuições dos NPB disponíveis[1]:

Version	Benchmarks Included	Problem Classes	Programming Models Used	Major Changes
NPB 3.3.1	IS, EP, CG, MG, FT, BT, BT-IO, SP, LU, UA, DC, DT	S,W,A,B,C,D,E	MPI, OpenMP, serial	added Class E
NPB 3.3.1-MZ	BT-MZ, SP-MZ, LU-MZ	S,W,A,B,C,D,E,F	MPI+OpenMP, OpenMP, serial	nested OpenMP version
GridNPB 3.1	ED, HC, VP, MB	S,W,A,B	Globus, Java, serial	added Globus version
NPB 3.0	IS, EP, CG, MG, FT, BT, SP, LU	S,W,A,B,C	OpenMP, HPC, Java	new programming paradigms
NPB 2.4.1	IS, EP, CG, MG, FT, BT, BT-IO, SP, LU	S,W,A,B,C,D	MPI	added BT-IO, Class D
NPB 2.3	IS, EP, CG, MG, FT, BT, SP, LU	S,W,A,B,C	MPI, serial	added CC, serial version

Observou-se que se teria de usar a distribuição **NPB3.3.1**, pois esta é a única que, para os kernels pretendidos e as classes pretendidas, permite avaliar tanto a versão sequencial, como uma versão em memória partilhada (OpenMP) e uma versão em memória distribuída (MPI).

D. Caraterização do Procedimento de Medição Utilizado

Entende-se que seja relevante realizar medidas de desempenho relativas à execução de 3 principais tipos de paradigmas: **Paradigma Sequencial; Paradigma Paralelo - Memória Partilhada; Paradigma Paralelo - Memória Distribuída**.

Para cada um destes paradigmas iremos testar com as versões adequadas de cada kernel, sendo elas respetivamente, **as versões sequenciais; as versões com OpenMP; as versões com MPI**.

De forma a realizar bons testes que adjetivem as capacidades máximas das máquinas utilizadas entende-se que a máquina deve estar a ser usada na sua totalidade. Isto é, no mínimo, todos os cores físicos da máquina devem ser utilizados em todos os testes a realizar. O uso de um maior número de threads/processos do que cores físicos nos testes relacionados com os paradigmas paralelos é também bastante interessante

para observar de forma direta o impacto de HyperThreading na execução, nas medições dos kernels em paradigma paralelo isto é tido em conta e faz-se também um pequeno estudo relacionado com o assunto.

Embora se reserve toda a máquina em cada teste sabe-se também que haverá sempre algum "ruído" implícito, por exemplo, por causa de processos externos ao controlo do utilizador da máquina que correm por omissão no background, ou por mudanças súbitas da frequência de clock talvez por um aquecimento excessivo da máquina em questão. Por estas razões indicadas nem sempre é fácil garantir que os testes realizados utilizam a máquina a cem por cento, no entanto, por falta de uma maneira de o fazer iremos assumir que o pedido de um job com o comando qsub com argumentos **-lnodes=1:r652:ppn=40** ou **-lnodes=1:r431:ppn=24** serão suficientes para garantir a exclusividade de cada nodo.

De forma a preparar cada execução, olhando para as ferramentas disponíveis nos dois nodos a utilizar, decidiu-se fazer todos os testes compilando com recurso ao compilador gcc e, de seguida, novamente todos os testes com o compilador icc. Para o icc tencionava-se usar a única versão disponível nas duas máquinas icc 2013.1.117. No entanto ao tentar usar esta ferramente uma falha na licença impediu o seu uso em ambos os nodos, por essa razão optou-se antes pelo gcc versão 4.7.2 já que o gcc versão 4.7.0 (não presente nos nodos usados) é a versão diretamente compatível com o icc referido. Para a outra versão do gcc escolheu-se a mais recente presente nos dois nodos (4.9.3), já que assim temos comparação direta entre ferramentas mais distintas o que se torna mais relevante. Para a compilação de kernels com MPI utiliza-se a ferramenta mpicc, sendo a versão do MPI usada a 1.6.2. Quanto às flags de optimização, usa-se apenas O2 e O3 comparando-se de seguida as execuções com estas flags com a execução do código compilado sem nenhuma flag de optimização.

Olhamos agora para um exemplo de um excerto de um script utilizado para executar os kernels para os testes:

```
for ((threads = 1; threads <= max_threads;
      threads++)) {
    do
        for ((seq=1; seq <= \$sample_size; seq++))
        do
            for exec in *.x
            do
                1. /home/a72293/resultados_gcc/dstat
                -tvfnml --output "\$exec_\$seq_"
                \${node_info}.csv >> /dev/null &
                2. ./\$exec > DSTAT_\$exec_\$seq_
                \${node_info}.txt
                kill \$!
                sleep 4
            done
        done
    done
}
```

Note-se no ponto circundado 1 como a ferramenta dstat é utilizada com o propósito de reunir dados relativos a ocupação em memória, ocupação do CPU, a transferência de dados em

rede, etc, causada pela execução de cada kernel, que ocorre no segundo ponto circundado. Caso estivessemos a querer testar o paradigma de memória partilhada com OpenMP, bastaria adicionar uma linha no ponto circundado 3 que atualizasse a variável de ambiente OMP_NUM_THREADS para o valor da variável threads. Note-se que, para os testes sequenciais, max_threads será igual a 1. Para o caso da situação de memória distribuída com MPI, podemos modificar a linha do ponto circundado 2, com o intuito de usar a ferramenta mpirun que permite executar programas compilados com mpicc e dar o valor da variável threads como argumento a mpirun.

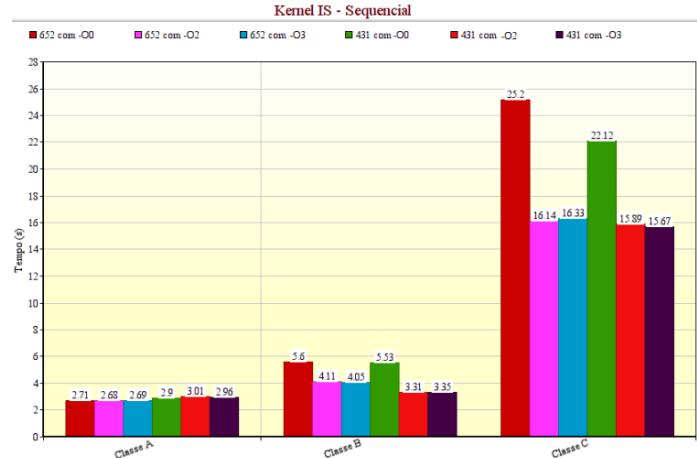
Por último, repare-se no ponto circundado 4, onde se tem o cuidado de usar o comando kill para de imediato impedir a ferramenta dstat de continuar a executar sem que haja necessidade. Depois é usado o comando sleep para que assim haja um compasso de espera entre cada medição de forma a tentar manter alguma imparcialidade na mesma.

Cada teste é repetido 10 vezes fazendo-se a média dos 5 melhores. No caso dos 5 melhores terem variações demasiado grandes entre si então repete-se os testes de forma a que os 5 melhores referidos não tenham uma divergência entre si de mais do que 10%.

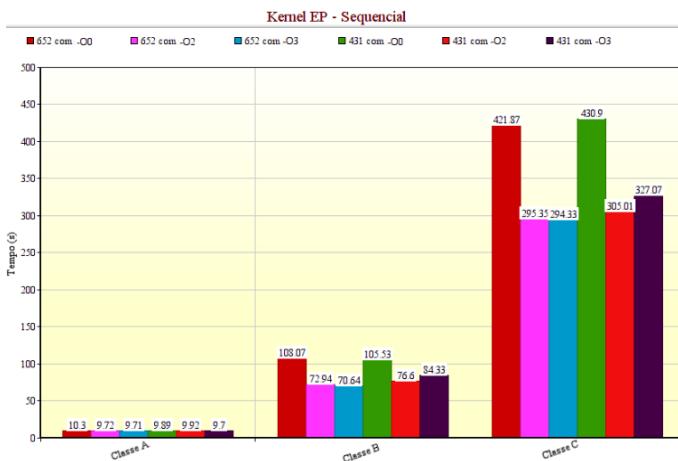
III. MEDIÇÕES DE DESEMPENHO - PARADIGMA SEQUENCIAL

Para cada kernel mediu-se o tempo em segundos que demorou em cada máquina com cada classe para cada compilador com diferentes flags de otimização. Apresentam-se agora gráficos demonstrativos dos resultados obtidos sendo cada gráfico relativo a um kernel distinto:

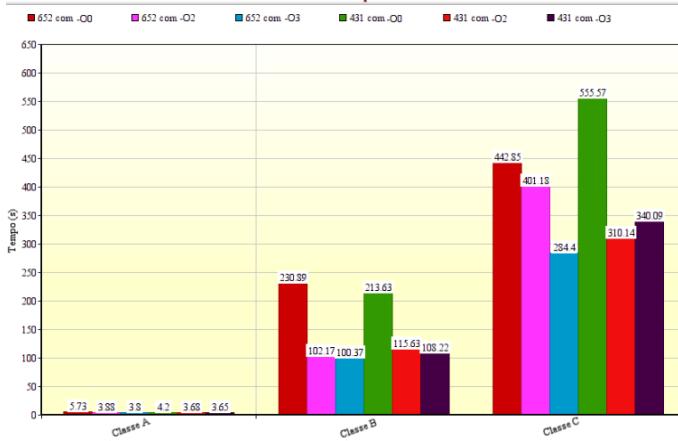
Kernel IS com compilador gcc 4.9.3:



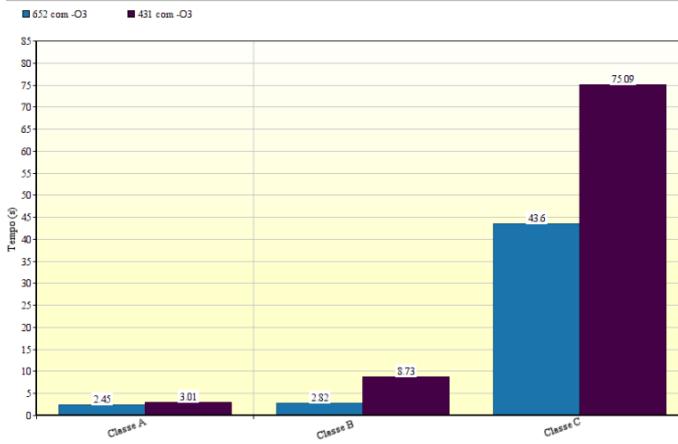
Kernel EP com compilador gcc 4.9.3:



Kernel CG com compilador gcc 4.9.3:
Kernel CG - Sequencial



Kernel MG com compilador gcc 4.9.3:
Kernel MG - Sequencial



Olhando para os gráficos podemos observar facilmente o impacto do uso de diferentes classes em cada kernel, em cada nodo e o impacto de variar as flags de optimização para o primeiro compilador.

Note-se que houve problemas com os resultados obtidos com o kernel MG já que estes foram valores sem sentido que decerto estavam errados por alguma situação exterior que não foi possível identificar (eram sempre obtidos valores perto do zero), por essa razão, não foi possível obter resultados sem flags de optimização nem com flag -O2. Os resultados obtidos com -O3 pensa-se serem confiáveis, no entanto, tendo em

conta a situação e o desconhecimento da sua causa, não se pode ter cem por cento de certeza.

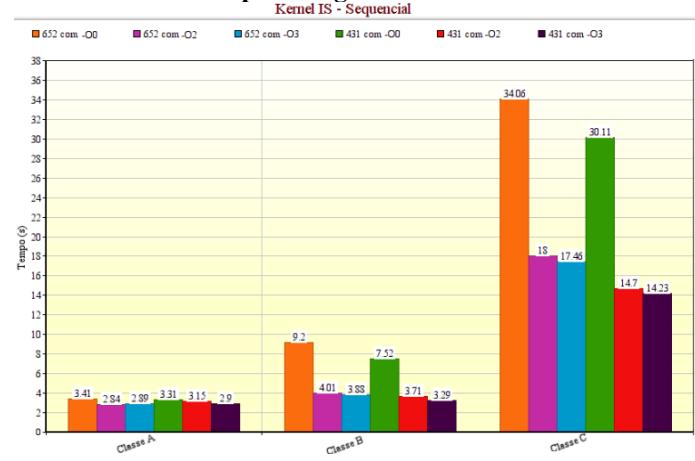
Olhando para os gráficos no geral vemos que com todos os kernels a classe A tem valores demasiado semelhantes nas diferentes situações o que leva a que não possamos concluir nada de especial olhando para os resultados da mesma. O pequeno tamanho da classe A em comparação com as restantes é a causa mais provável para este comportamento. Por esta razão, em testes posteriores não iremos avaliar a classe A.

Vemos também um aumento no tempo de execução à medida que se usam classes maiores, como seria expectável. No geral usar flag de otimização -O2 neste compilador é sempre melhor em termos de tempo de execução em comparação com o uso de nenhuma flag, no entanto com a flag -O3 os ganhos, quando existem, são mínimos. Uma possível razão para isto é a flag O2 fazer já a maioria das optimizações possíveis aos kernels utilizados, sendo que a flag O3 pouco mais pode fazer já que, dependências de dados, situações que possam dar azo a pointer aliasing, etc podem estar a impedir uma maior performance com o uso da flag -O3.

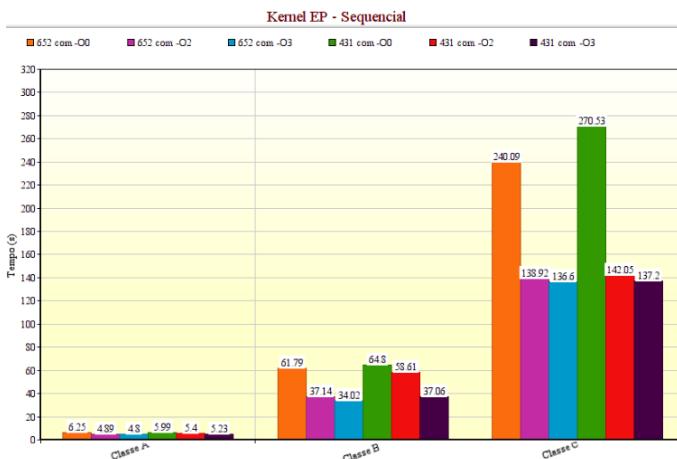
Notamos ainda que, na maioria dos casos, o nodo 652 tem uma melhor performance tanto sem flags de otimização como com flags de otimização. O kernel IS é talvez o que mais diverge desta afirmação o que pode indicar uma maior capacidade do nodo 431 para o tratamento de inteiros e/ou acessos aleatórios à memória (como é característico de algoritmos de ordenação).

Vemos agora os mesmos gráficos mas para o segundo compilador:

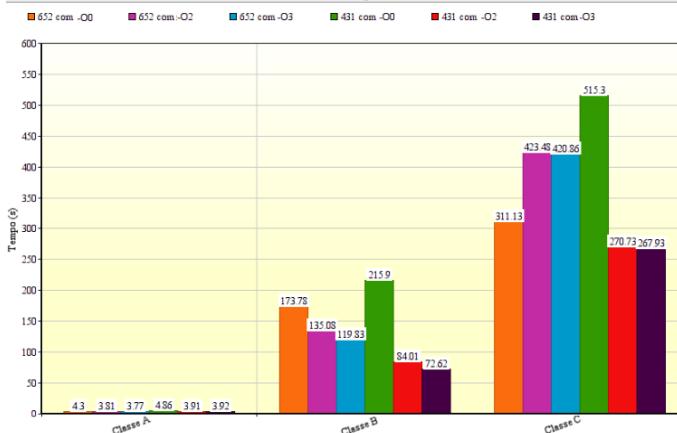
Kernel IS com compilador gcc 4.7.2:



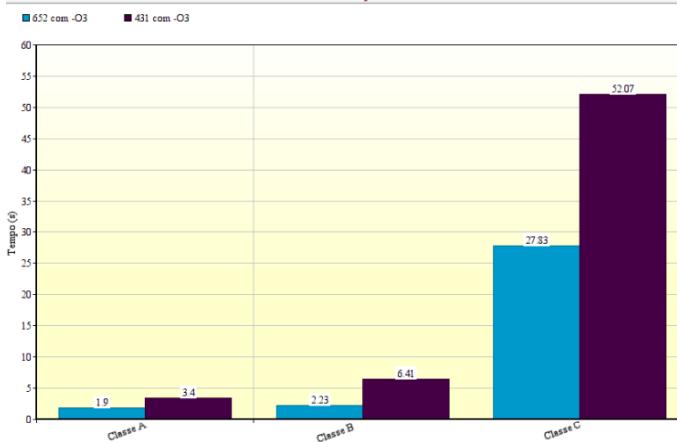
Kernel EP com compilador gcc 4.7.2:



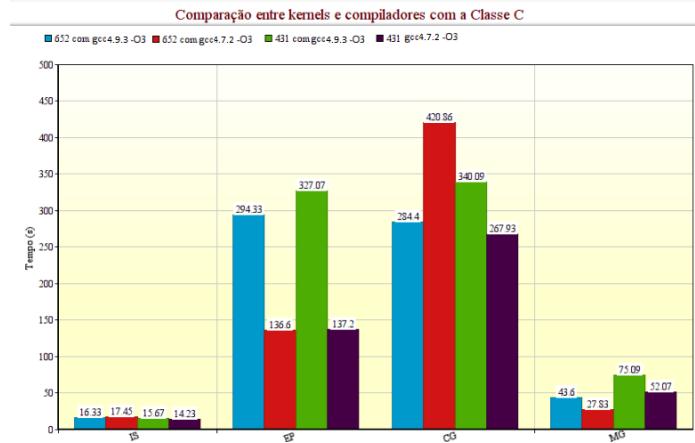
Kernel CG com compilador gcc 4.7.2:
Kernel CG - Sequencial



Kernel MG com compilador gcc 4.7.2:
Kernel MG - Sequencial



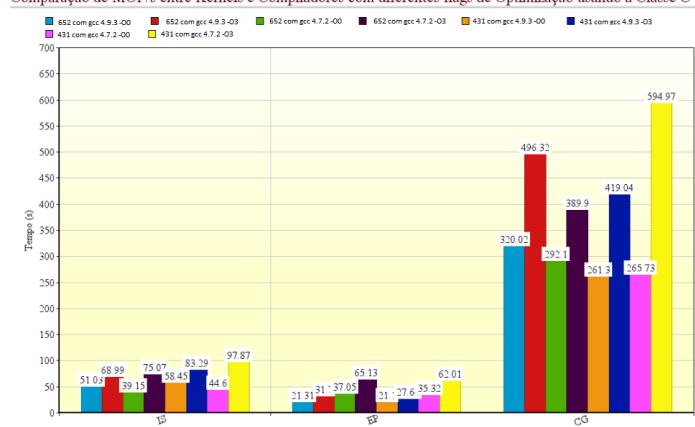
Relativamente aos nodos as mesmas conclusões que foram obtidas para o compilador anterior mantêm-se. De forma a poder comparar diretamente os resultados obtidos com os dois compiladores distintos apresenta-se um gráfico que mostra para cada nodo e cada kernel os resultados dos dois compiladores com a flag -O3:



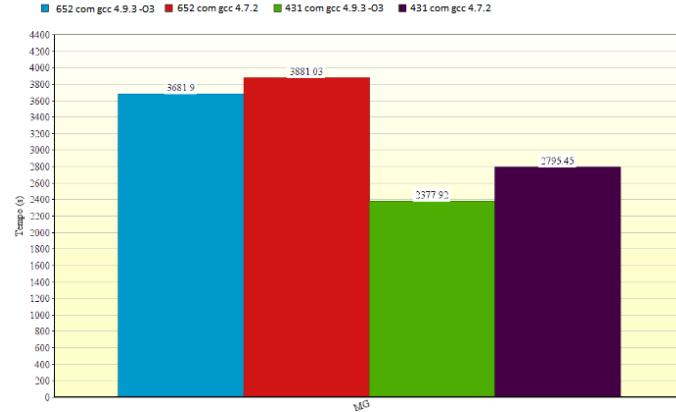
Com este gráfico Observamos que o segundo compilador obtém melhores resultados com a flag -O3 do que o compilador 1, no entanto observando os gráficos anteriores vemos que para as restantes flags o compilador 1 tem melhores resultados. Isto leva a crer que o segundo compilador consegue ultrapassar mais facilmente as barreiras dos kernels no que diz respeito a otimizações.

Observamos agora, também para o paradigma sequencial, MOP/s (Milhões de Operações por segundo) para cada kernel de forma a entendermos melhor a diferença entre cada compilador sem flags de otimização e com a flag -O3 e entender também melhor as diferenças de performance obtidas num kernel em relação a outro já que seremos capazes de entender em que kernels foi possível obter mais optimizações com as flags de optimização. Note-se que agora apenas usamos a classe C, pois como observamos nos gráficos anteriores é a que causa maiores tempos de execução, logo é aquela que se torna mais relevante sendo a mais fácil de observar variações, mesmo que pequenas.

Comparação de MOP/s entre Kernels e Compiladores com diferentes flags de Optimização usando a Classe C



Comparação de MOP/s entre Kernels e Compiladores com diferentes flags de Optimização usando a Classe C



Aqui podemos concluir que o uso de flags de otimização superiores possibilitam um maior número de operações por segundo em ambos os compiladores, o que era expectável, mostrando assim realmente que níveis de otimização superiores podem alcançar melhores tempos de execução como vimos anteriormente já que no mesmo período de tempo é realizado um maior número de instruções.

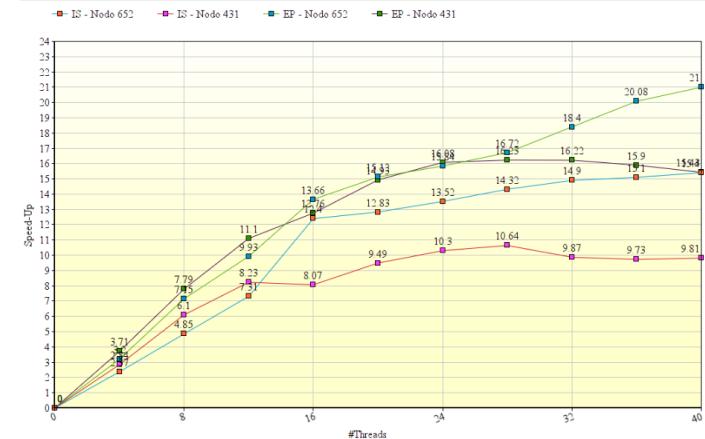
Vemos também que em várias situações o nodo 431 se revela melhor a nível de MOPS/s com flag de otimização O3, no entanto para os kernels mais paralelizáveis como o EP esta diferença já não é tão acentuada. Conclui-se então que código pouco paralelizável é mais facilmente otimizado no nodo 431, enquanto que código muito paralelizável como o EP obtém melhores otimizações no nodo 652 levando a crer que este último terá resultados melhores que o 431 para as versões paralelas quando otimizadas com a flag -O3.

Em suma, Se usada a flag de compilação -O3 então é vantajoso o uso do segundo compilador, no caso contrário o uso do primeiro é preferível. Se o código sequencial a executar tiver demasiadas dependências sendo então código muito pouco paralelizável então há vantagens em o usar na microarquitetura Nehalem como a que temos no nodo 431. Caso o código a executar seja muito paralelizável, até se for embarrasosamente paralelo como o kernel EP então será preferido o uso da microarquitetura Ivy Bridge como temos no nodo 652. Podemos ainda ter em conta que código que envolva operações de ordenação de inteiros tem melhores resultados no nodo 431 sendo então justificável o uso do mesmo para executar programas que recorram a ordenação.

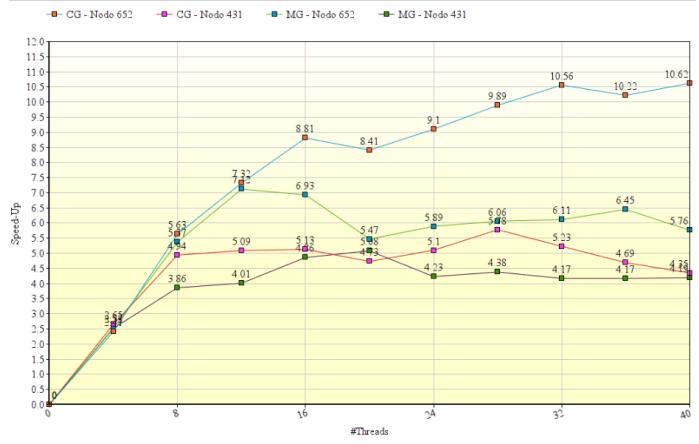
IV. MEDIÇÕES DE DESEMPENHO - PARADIGMA PARALELO: MEMÓRIA PARTILHADA

Apresentam-se agora gráficos dos ganhos obtidos na versão paralela em memória distribuída com OpenMP em relação à versão sequencial. Note-se que, como referimos anteriormente, a classe C é a que se torna mais relevante e, por esse motivo, é a classe que usamos em todos os testes daqui para a frente. O compilador usado é o compilador 1 apenas com a flag -O3 já que esta foi a flag que proporcionou os melhores resultados sequenciais.

Speed-Up da versão OpenMP com a classe C



Speed-Up da versão OpenMP com a classe C



Infelizmente houve a necessidade de separar os kernels em 2 gráficos distintos para evitar que fosse confuso demais. Desta forma torna-se mais complicado entender para cada número de threads utilizado as diferenças de desempenho dos kernels em comparação uns com os outros. No entanto podemos facilmente ver que o kernel EP é o que obtém os melhores ganhos relativamente à sua versão sequencial, isto já era de esperar já que tínhamos visto a sua natureza paralela explícita no facto de o número de MOP/s da versão sequencial ser o menor de entre todos os kernels. Como os MOP/s são tão reduzidos a divisão da computação entre threads torna-se fácil e sendo ele embarrasosamente paralelo, tendo poucas dependências, a divisão dos dados também se torna simples. Contrastando com o kernel EP temos o kernel MG que é o que obtém menos ganhos em ambos os nodos. O facto de ser o kernel que obteve o maior número de MOP/s na versão sequencial pode ser sinal de que a divisão da computação pelas threads é um processo complicado. Em norma, no nodo 652, os ganhos são superiores aos obtidos no nodo 431, mas apenas apartir de cerca das 12 threads. Como tínhamos visto no capítulo anterior era expectável que o nodo 652 paralelizasse mais facilmente. Era também já esperado que o nodo 431 após as 12 threads comece a ter uma curva de speed up menos acentuada já que apenas tem 12 cores físicos e, apartir das 12 threads estaremos a recorrer a hyperthreading o que terá algum impacto no desempenho. Entendemos de imediato que 12 cores físicos nunca poderão competir com os 20 cores físicos presentes no outro nodo onde só se começa

a ver uma quebra na curva por volta das 20 threads. Para além disso, o facto de o nodo 652 ter uma cache de nível 3 maior que o nodo 431 também é um aspeto relevante que leva a um melhor desempenho por parte do 652. O facto de, em algumas situações, antes das 12 threads o nodo 431 obter melhores resultados pode ter sido influenciado pela sua maior frequência de clock que, embora seja uma diferença não muito significativa, poderá ser mais explícita em código em execução paralela já que temos vários cores a operar com uma frequência superior aos cores do outro nodo. Além disso, o facto de o nodo 431 ter tido resultados muito mais baixos na versão sequencial poderá fazer com que o aumento do desempenho com a versão paralela seja mais significativo.

No kernel EP usando o nodo 652 notamos também que a quebra na curva dos ganhos mencionada anteriormente é praticamente inexistente. Uma possível razão poderá ser a natureza embaraçosamente paralela do kernel em questão que apenas tem a beneficiar com um maior número de threads e a razão pela qual isto não acontece tanto para o nodo 431 poderá ser precisamente por o nodo 652 ser capaz de melhores otimizações a código paralelizável, como vimos no capítulo anterior.

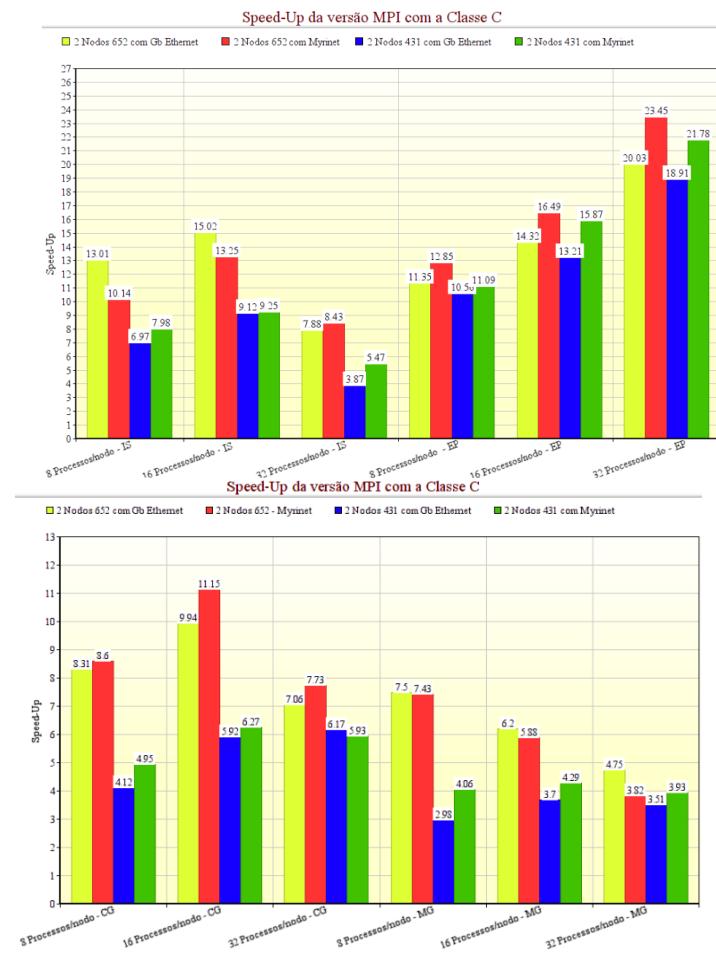
Notemos ainda que, para um número de threads superior a 24, o nodo 431 passa a não ter threads suficientes para que todas possam agir concurrentemente, nem mesmo com recurso a hyperthreading. Deste modo incorre em vários overheads que poderão ser a razão pela qual para um grande número de threads os resultados do nodo 431 são tão inferiores aos resultados do nodo 652. Já que no nodo 431 um certo número de threads terá que estar em espera sendo que, quando finalmente forem escalonadas terão que copiar tudo o que foi feito pelas outras para a sua cache de forma a manter a coerência de cache já que os dados que tinham poderão agora estar obsoletos ao terem sido modificados pelas restantes threads.

Em suma, para grandes números de threads o nodo 652 mostra vantagens significativas, como já seria de esperar, no entanto, se o objetivo for usar um algoritmo que apenas escala com um pequeno número de threads (12 ou menos) o nodo 431 já será uma melhor opção mesmo tendo uma menor cache de nível 3.

V. MEDIÇÕES DE DESEMPENHO - PARADIGMA PARALELO: MEMÓRIA DISTRIBUÍDA

Com o intuito de fazer testes em ambiente paralelo com memória distribuída surgiu uma questão a solucionar. Não tem propósito o teste em ambiente de memória distribuída se for feito num só nó como foi feito nos capítulos anteriores, pois assim os processos irão comunicar com recurso a memória partilhada e não através de rede Ethernet ou Myrinet impedindo-nos de avaliar as diferenças de desempenho com estas diferentes maneiras de comunicação, portanto há que decidir quantos nodos usar. Bem sabemos que quantos mais nodos usarmos mais relevante será o nosso estudo já que a memória e a computação estarão muito mais distribuídas entre diferentes nodos sendo possível avaliar melhor o impacto no desempenho. No entanto, embora tenhamos 10 nodos 431,

apenas temos 2 do tipo 652[4]. Por este motivo apenas foram usados 2 nodos iguais em cada teste já que se entende que deve ser usado o mesmo número de nós para os testes em cada tipo de nó distinto de forma a manter o máximo de parcialidade e 'justiça' nos resultados obtidos. Foram então realizados testes para 2 nós 652 comunicando entre si usando Gigabit Ethernet, 2 nós 652 comunicando entre si usando Myrinet com 10Gbps e o mesmo para o tipo de nodo 431. Seguem-se gráficos de barras demonstrativos dos ganhos obtidos nesta versão, usando openmpi 1.6.2, em relação à versão sequencial.



Mais uma vez foi necessário dividir o gráfico em dois para facilitar a visualização da informação e, já que agora temos ainda mais informação a mostrar (testes com Ethernet e testes com Myrinet), optou-se pelo uso de gráficos de barras aos invés de gráficos de linhas.

Olhando ainda num panorama muito geral notamos logo que o uso de Myrinet para comunicação entre processos de diferentes máquinas permite a obtenção de maiores ganhos do que o uso de Ethernet. No entanto seria de esperar que o kernel EP obtivesse resultados muito parecidos com Ethernet e Myrinet, pois sendo um kernel embaraçosamente paralelo cada processo irá ser praticamente independente, logo a comunicação será extremamente reduzida. No entanto nota-se nos gráficos uma variação significativa entre EP com Ethernet e com Myrinet, a razão para isto é desconhecida, talvez o facto de terem sido feitas medições em momentos diferentes pode ter tido influência,

pois embora se tente ao máximo que todos os nodos estejam reservados somente para estes testes há sempre processos que correm em 'background' que em certos momentos poderão ser mais pesados que outros adulterando um pouco os resultados finais. O próprio hardware pode ter estado numa situação de maior calor aquando de algum teste o que pode ter impedido a frequência do clock de algum nodo de subir tão alto como poderia ter subido. As possibilidades são muitas, no entanto não é possível ter a certeza da razão pela qual o resultado para o kernel EP não foi o expectável.

Note-se agora que embora realmente fosse esperado que comunicação com recurso a Myrinet com 10Gbps obtesse melhor desempenho do que o uso de Gigabit Ethernet, este desempenho ficou aquém das expetativas. Isto porque a latência de comunicação por Myrinet será à volta de 10 microsegundos enquanto que por Ethernet será à volta de 100 microsegundos o que significa que, teóricamente deveríamos ter um ganho de 10 vezes com Myrinet em relação a Ethernet[5]. No entanto uma possível explicação para isto não acontecer é que estes valores de latência encontrados em várias fontes distintas normalmente apenas têm em consideração a própria comunicação em si e não têm em consideração a latência que a própria aplicação em execução pode causar. Isto porque, mesmo que tenhamos um método de comunicação praticamente instantâneo a comunicação terá sempre que obedecer aos 'caprichos' da aplicação, se é necessário fazer algum cálculo importante este pode causar que a 'bandwidth' da rede não esteja a ser ocupada sempre a cem por cento já que, por vezes, a comunicação acontece somente em certos pontos do programa e não constantemente. Por isso mesmo que tenhamos uma rede com muito mais latência o impacto no desempenho causado por essa latência diminui se o programa for mais intensivo a nível de computação independente e tiver menos dependências de dados. Podendo até certos cálculos serem realizados ao mesmo tempo que é efetuada a comunicação mascarando, em certas situações, a latência praticamente por completo. Esta afirmação comprova-se quando se repara que nos gráficos os ganhos obtidos com Myrinet nos kernels com pouca dependência de dados são mais próximos dos ganhos obtidos com Ethernet, embora que mesmo assim sejam ganhos maiores.

Por fim notamos mais uma vez que os ganhos obtidos com nodos 652 são significantemente maiores do que os ganhos obtidos com nodos 431. Apenas no kernel EP vemos o desempenho nos dois tipos de nodos ser mais semelhante do que com os outros kernels.

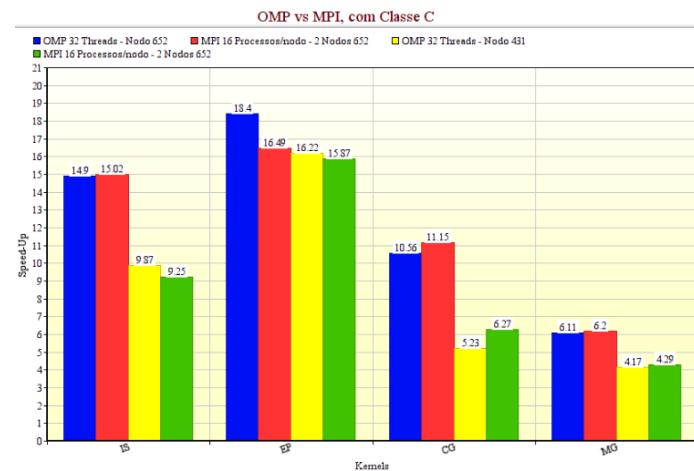
Quanto à escalabilidade a nível de processos nota-se mais uma vez que após a utilização de mais processos do que o número de cores físicos temos uma quebra no desempenho, o que seria de esperar pelas razões já mencionadas no capítulo anterior. Esta quebra mais uma vez surge mais cedo com nodos 431 já que, usando dois nodos de uma vez temos somente 24 cores físicos por isso, a não ser no kernel EP, temos uma quebra quando passamos de 16 processos por nodo para 32 por nodo. E até mesmo com 16 por nodo, já que cada nodo apenas tem 12 cores físicas estamos também já a recorrer a hyperthreading. Para os nodos 652 o impacto é menor pois passa a ter 40 cores físicas, mas mesmo assim sente-se o impacto ao usar 32

processos por nodo já que cada nodo tem somente 20 cores físicos. Notamos que tanto os nodos 652 como os nodos 431 obtêm resultados piores com 32 processos por nodo do que com 8 por nodo (excepto no kernel EP). Isto significa que é alcançado o pico da escalabilidade entre 16 processos por nodo e 32 por nodo. Por razões óbvias o ponto de escalabilidade dos nodos 652 é alcançado mais tarde do que os 431 já que alcançam os 32 processos por nodo com melhores ganhos do que os nodos 431. O que significa que os nodos 652 escalam melhor a nível de threads do que os nodos 431, sendo esta a mesma conclusão a que se chegou ao olhar os gráficos do capítulo anterior.

Em suma, mais uma vez os nodos 652 mostram vantagem em serem usados se for necessário um número elevado de processos. No entanto a não ser que o algoritmo seja embarrascosamente paralelo não se deve usar mais do que cerca de 20 processos por nodo que é o ponto de pico de speed up estimado. Ao contrário da versão anterior, não há vantagens no uso de nodos 431 para um pequeno número de processos. Em todos os casos o uso de Myrinet mostra-se vantajoso.

VI. CONCLUSÕES

Em jeito de conclusão olhamos agora para um gráfico que nos permite relacionar diretamente os melhores resultados obtidos com OpenMP e MPI:



Com este gráfico vemos que para os kernels MG e IS a Versão MPI obtém praticamente os mesmos resultados do que a versão OMP. No entanto note-se que a versão OMP tem menos recursos à sua disposição. Isto porque estamos a comparar OMP com 32 threads OMP com MPI com 16 processos por nodo (e dois nodos). Desta forma percebemos que na versão MPI temos mais cores físicas do que na versão OMP, no entanto para estes dois kernels mencionados o resultado foi mesmo assim praticamente igual. Com isto conclui-se que os kernels MG e IS têm vantagens em ser aplicados com num só nodo com OMP já que assim pouparam-se recursos. No entanto estima-se que se os recursos não forem um problema então qualquer uma das opções é viável.

Para o kernel CG notamos que a versão MPI obtém melhores resultados do que a versão OMP. Isto é um indicador que o tipo de algoritmos no qual o kernel CG se insere beneficia do

paradigma de memória distribuída mais do que o de memória partilhada. No entanto estes melhores resultados também podem ser fruto do maior número de cores físicos presentes ao usar MPI como referido anteriormente. Mesmo assim, o uso de MPI é vantajoso, pois com OMP não é possível obter mais ganhos significativos, pois podemos ver no gráfico do capítulo da versão OMP que com 32 threads o kernel CG encontra-se já a converger sendo que se adicionassemos mais threads veríamos os ganhos a alcançarem um pico máximo rapidamente. Como este pico máximo seria alcançado através de HyperThreading seria, em princípio, mais baixo do que o pico máximo do que com MPI, contando que o overhead da comunicação por Myrinet não mascare os ganhos do maior número de cores físicos, já que na versão com OMP a comunicação entre threads será bastante rápida pois será feita com recurso a memória partilhada.

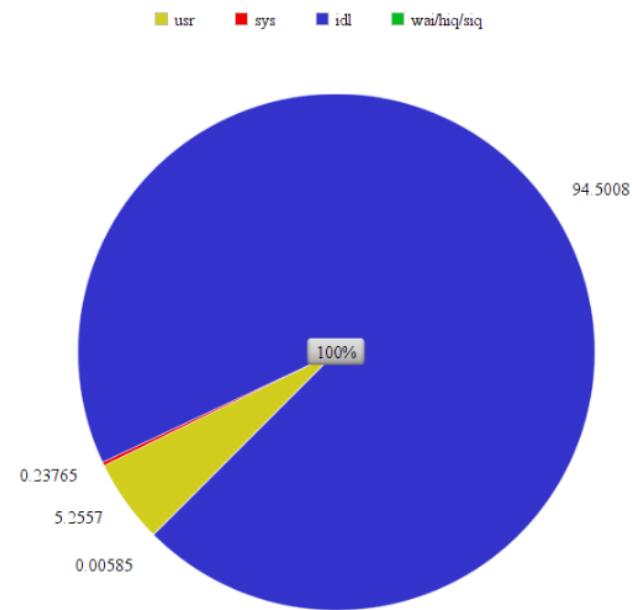
Para o kernel EP a versão OMP é superior e tal já seria de esperar pelas razões já referidas nos capítulos anteriores. Seria então lógico escolher uma implementação em OMP para executar algoritmos embarracosamente paralelos já que estes são os que mais beneficiam do paradigma de memória partilhada.

Por último voltamos a verificar que o nodo 652 é o que permite obter melhores resultados para todos os kernels quando são usados números elevados de threads. Como ambos os nodos 652 e 431 têm microarquiteturas desenvolvidas pela Intel e como 652 é uma microarquitetura mais recente concluímos também que a Intel está a seguir a filosofia de que ter um maior número de cores físicos um pouco mais fracos é melhor do que ter menos cores com uma maior frequência. Estes dois nodos descrevem esta situação na perfeição já que o nodo 431 é mais antigo, tem menor número de cores, mas uma frequência mais elevada, no entanto o nodo 652 tem maior número de cores e uma menor frequência. E realmente verificamos que esta filosofia adoptada no nodo 652 traz vantagens quando o nosso interesse é a execução de programas paralelizando com um grande número de threads.

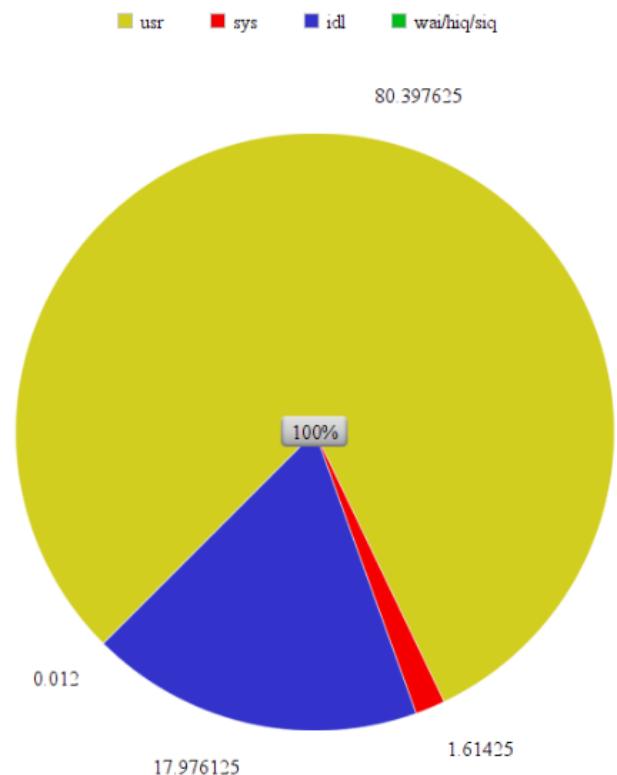
Tendo considerado o nodo 652 como o melhor na generalidade dos casos, já que mais vezes é a escolha mais acertada do que o outro nodo, achou-se relevante observar alguns testes que indiquem como se comportou ao serem efetuados os testes para a melhor medição obtida do kernel EP com o intuito de entender como os algoritmos a testar nesta máquina podem ser modificados de forma a tirar melhor partido da mesma. Os graficos que iremos agora observar foram obtidos com recurso à ferramenta **Dstat** como referido no capítulo 1.D:

Utilização do CPU:

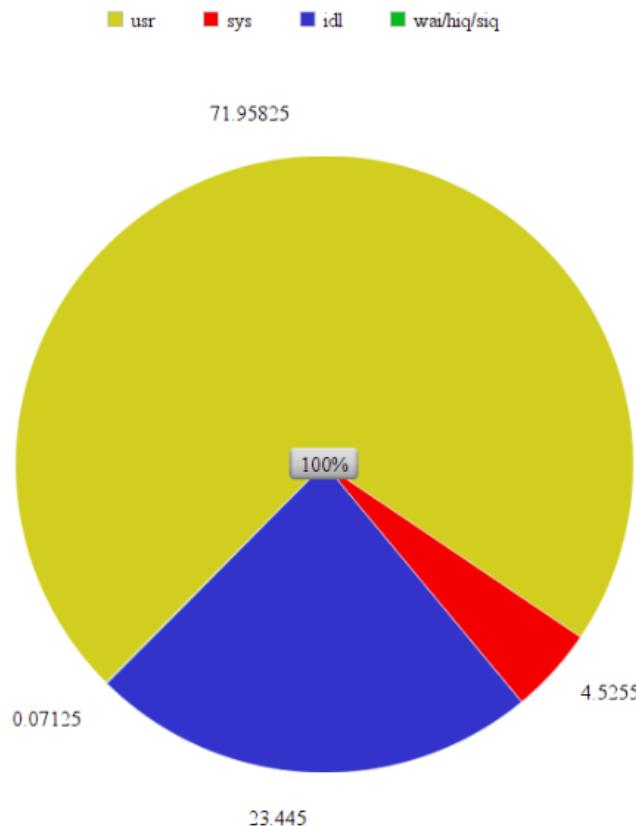
Kernel EP versão Sequencial - Utilização do CPU



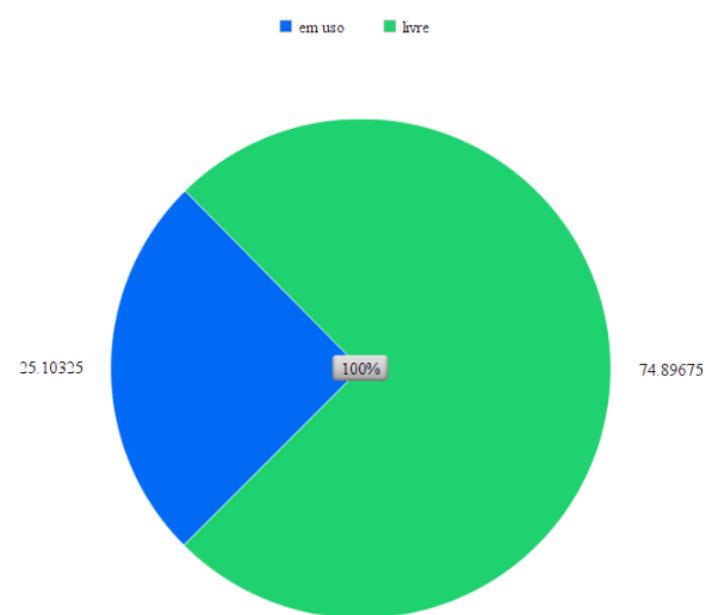
Kernel EP versão OpenMP - Utilização do CPU



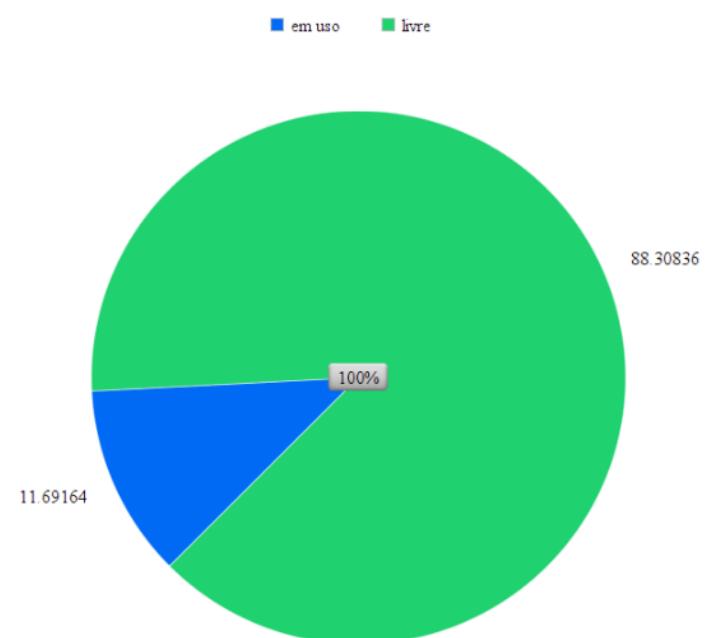
Kernel EP versão MPI - Utilização do CPU



Kernel EP versão Sequencial - Utilização da Memória



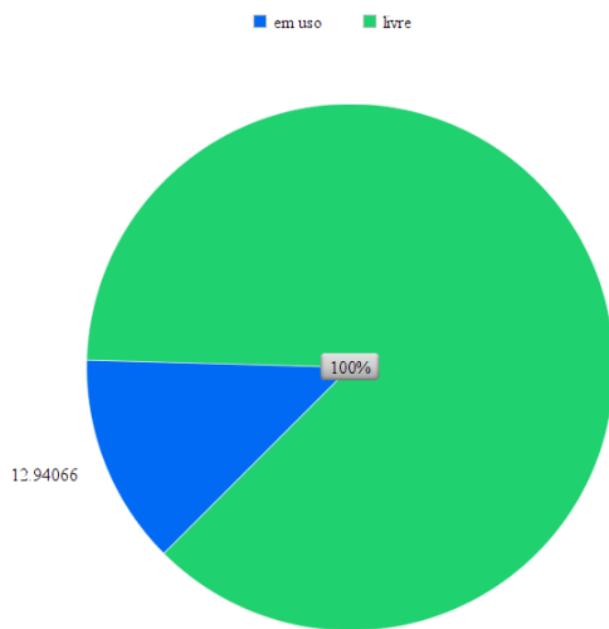
Kernel EP versão OpenMP - Utilização da Memória



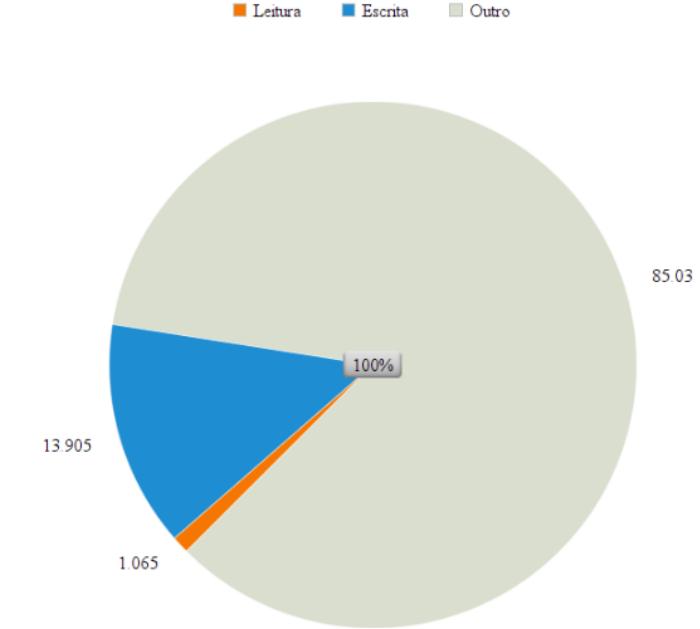
Podemos observar que na versão sequencial o CPU encontra-se na maior parte do tempo parado (percentagem representada pela sigla **idl**). Isto deve-se à natureza embarcada do kernel EP e prova-se isto ao se notar nos outros dois gráficos das versões paralelas como agora o CPU é muito mais utilizado, principalmente a executar em user-mode (representado pela sigla **usr**). No entanto o gráfico relativo à versão MPI mostra uma menor ocupação do CPU, provavelmente permanece idl uma maior percentagem de tempo devido à latência de comunicação entre processos de máquinas distintas que têm que esperar pela comunicação uns dos outros. Daqui concluímos algo que já era expectável, mas que fica agora provado. Nesta máquina com este kernel o fator limitante principal surge na versão MPI com a comunicação entre processos de máquinas distintas. De forma a melhorar as versões MPI de forma a igualar o OMP ou até superar, devemos focar-nos nos custos de comunicação de forma a reduzir a percentagem de tempo **idl** do CPU.

Utilização da Memória:

Kernel EP versão MPI - Utilização da Memória



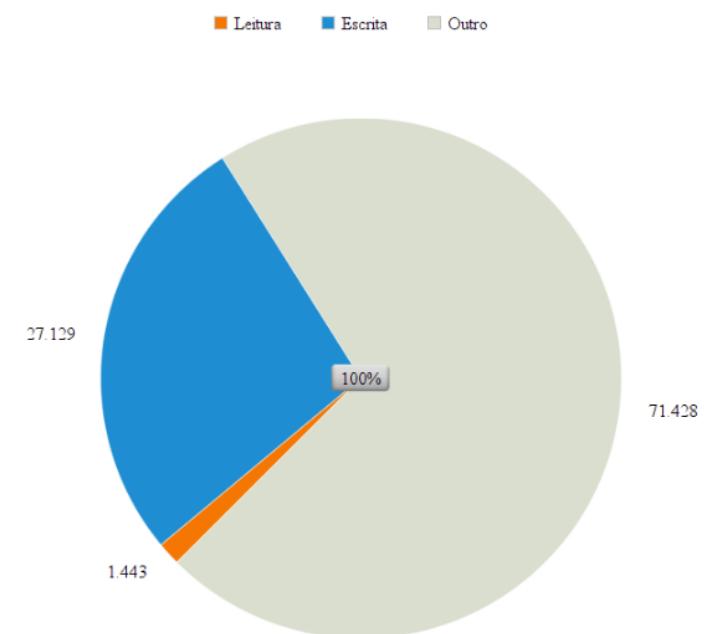
Kernel EP versão Sequencial - Escritas/Leituras



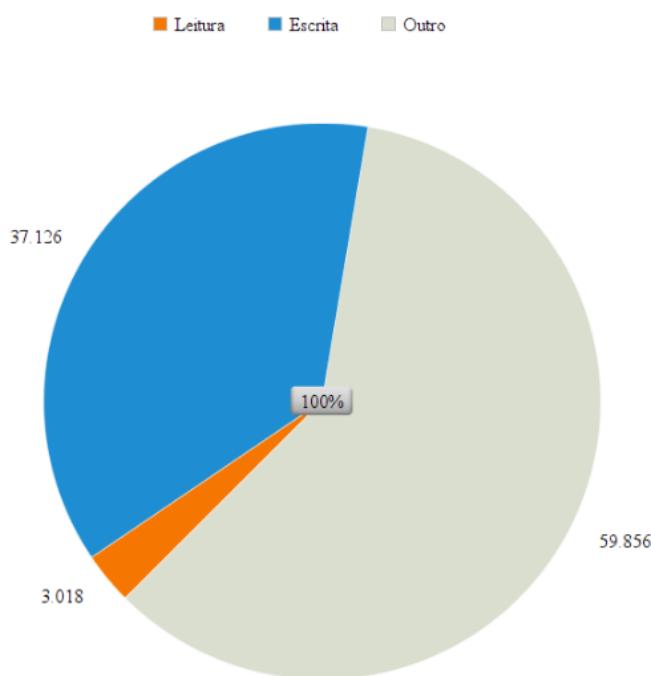
Estes gráficos indicam-nos a percentagem de memória em uso e livre ao longo do teste efectuado. Os resultados são o esperado já que os valores de memória livre da versão sequencial são um pouco maiores e os valores da versão OMP e MPI são bastante parecidos, pois ambos têm necessidades extra ligadas à memória, como por exemplo o acesso de parte de uma thread para atualizar valores escritos por outra na cache do seu próprio core de forma a manter a coerência de cache. Isto é algo que não acontece na versão sequencial e poderá ser uma limitação para as versões paralelas em casos extremos. Por esse motivo poderá ser um ponto a focar de forma a melhorar o desempenho das versões paralelas.

Balanceamento Escritas/Leituras:

Kernel EP versão OpenMP - Escritas/Leituras



Kernel EP versão MPI - Escritas/Leituras



Aqui vemos para este tipo de algoritmos (embaraçosamente paralelos) as leituras são mínimas comparadas com as escritas, sendo que as escritas têm muito maior impacto na versão sequencial. As versões paralelas resolvem um pouco a questão, no entanto a versão MPI tem ainda assim muito mais escritas do que a versão OMP. Por esta razão as escritas são um factor mais limitante do que as leituras e, se isso for uma preocupação, deve ser usada a versão OMP já que é a que sofre menos com as escritas. Se se tiver o objetivo de melhorar a versão MPI de forma a igualar ou a superar a versão OMP então as escritas poderá ser um ponto interessante a tentar optimizar.

VII. TRABALHO FUTURO

Tendo estudado pormenoradamente nodos representativos da maior parte das máquinas pertencentes ao cluster utilizado, chegando à conclusão dos pontos fortes e fracos de cada uma e avaliando fatores limitantes e pontos a focar de forma a otimizar código para o nodo que se considerou o melhor na maioria dos casos, entende-se que o projeto terá sido concluído com sucesso. No entanto muita coisa ainda poderia ser feita, por exemplo, foi avaliado o impacto das diferentes classes para a versão sequencial, no entanto nas versões paralelas apenas se usou a maior classe. Se se testasse os ganhos para as restantes classes seria possível também compreender a escalabilidade ao nível dos dados, um assunto que foi pouco abordado neste relatório mas que é de todo relevante. O uso de mais do que um compilador para teste foi interessante, no entanto os dois compiladores usados são ambos do mesmo tipo (gcc) apenas variando na versão. Seria mais interessante talvez a comparação entre compiladores bastante distintos com o gcc e o icc. No entanto não foi possível usar o icc pelas razões já indicadas no capítulo 2.

Estes extras revelam-se como algo de interessante a ponderar

no futuro e como possíveis adições ao relatório final aquando da entrega do portefólio.

VIII. BIBLIOGRAFIA

- [1] <https://www.nas.nasa.gov/publications/npb.html>
- [2] https://en.wikipedia.org/wiki/NAS_Parallel_Benchmarks
- [3] https://www.nas.nasa.gov/publications/npb_problem_sizes.html
- [4] http://search6.di.uminho.pt/wordpress/?page_id=55
- [5] <https://pdfs.semanticscholar.org/3994/7b2fbcc5d25d5a827edf1c234099>

TPC3: Desenvolvimento de Programas em DTrace

- Engenharia dos sistemas de Computação -

autor: Daniel Malhadas

Resumo—O presente documento apresenta um estudo aprofundado sobre a ferramenta DTrace. Para isso foram realizados vários programas onde se tem o intuito de explorar várias das funcionalidades desta ferramenta. Estes programas permitem fazer traçados de chamadas de sistema e assim organizar informação relativa à sua instrumentação, que foi depois cuidadosamente estudada de forma a comprovar a correção dos programas desenvolvidos e assim verificar uma boa aprendizagem da ferramenta em questão. Note-se ainda que todo este estudo foi realizado em ambiente Solaris 11 e, por essa razão, os dados instrumentados serão relativos somente a esse ambiente.

Index Terms—DTrace, Solaris 11, Computação Paralela e Distribuída.

I. INTRODUÇÃO

A. Contextualização e Motivação

Com este projeto pretendeu-se escrever programas em DTrace que permitissem fazer o traçado de chamadas ao sistema. De seguida vários testes são realizados com o intuito de comprovar a correção dos programas desenvolvidos. Os resultados destes testes foram depois organizados e apresentados ao longo deste documento. A razão do desenvolvimento deste projeto é então o de melhor entender o uso e a utilidade da ferramenta DTrace e, já que os testes são realizados em ambiente Solaris 11, entender também melhor como funcionam as chamadas ao sistema e a própria ferramenta DTrace neste ambiente específico.

B. O que é a ferramenta DTrace

A Oracle Solaris DTrace é uma ferramenta de rastreio/traçado avançada usada para testar troços problemáticos de um programa em tempo real. Para isto, esta ferramenta permite observar questões de performance tanto em pequenas aplicações como do próprio sistema operativo em si de forma dinâmica e segura permitindo a quem a utilize a identificação de problemas que seriam difíceis de detetar com outras ferramentas similares por estarem demasiado disfarçados sobre várias camadas de software.

A ferramenta permite também a instrumentação de várias estatísticas em tempo real relativas ao programa a testar. Estatísticas como: consumo de memória, tempo de CPU despendido, que chamadas de função foram realizadas, etc.[1] De forma a poder traçar o que está a acontecer, a ferramenta DTrace recorre à monitorização de diversos "sinais"/"pontos de interesse" marcados no sistema operativo a que se dá o nome de **probes**. Estes probes são lançados em momentos específicos e, cabe ao utilizador da ferramenta DTrace, saber quais os probes que deve intercetar e o que fazer quando os intercetar de forma a alcançar os resultados que pretende.

Segue-se uma tabela com os probes disponíveis em ambiente Solaris 11 e uma breve descrição: [2]

Common DTrace Providers	Description
dtrace	Start, end and error probes
syscall	Entry and return probes for all system calls
fbt	Entry and return probes for all kernel calls
profile	Timer driven probes
proc	Process creation and lifecycle probes
pid	Entry and return probes for all user-level processes
io	Probes for all I/O related events
sdt/usdt	Developer defined probes at arbitrary locations/names within source code for kernel and user-level processes
sched	Probes for scheduling related events
lockstat	Probes for locking behavior within the operating system

II. PROGRAMAS DESENVOLVIDOS

A. Desenvolvimento do Programa 1

De início quis-se desenvolver um programa que permitisse fazer o traçado das chamadas ao sistema open() (como em Solaris 11 não existem probes para fazer o traçado de open fazemos antes o traçado foi de openat() que é uma chamada de sistema mais geral que engloba também chamadas de sistema open())[2]). entendeu-se que este programa deveria ser capaz de imprimir a seguinte informação por linha:

- 1 - Nome do ficheiro executável e respetivos: PID do processo, UID do utilizador e GID do grupo;
- 2 - Caminho absoluto para o ficheiro que for aberto;
- 3 - A cadeia de caracteres com as "flags" da chamada ao sistema openat(), O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT;
- 4 - O valor de retorno da chamada de sistema.

O programa desenvolvido encontra-se apresentado de seguida:

```
1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3
4 /**
5  * Tracer de chamadas ao sistema openat().
6  * int openat(int fildes, const char *path, int oflag, mode_t mode);
7 */
8
9 dtrace:::BEGIN { 1
10    printf("Tracer de chamadas ao sistema openat().\n");
11    printf("_____  
");
12    printf("|| %s, %s, %s, %s, %s, %s\n",
13          "Executable", "Path", "Flags", "PID", "UID", "GID", "Return");
14    printf("_____  
");
15    printf("_____  
");
16    printf("_____  
");
17 }
18
19 /*Catch openat system call entry*/
20 syscall::openat*:entry { 2
21     /*Print executable name and absolute path*/
22     printf("|| %s, %s", execname, copyinstr(arg1));
23
24     printf("%s", arg2 & O_RDONLY ? ", O_RDONLY" :
25             (arg2 & O_WRONLY ? ", O_WRONLY" : ", O_RDWR" ) );
26     printf("%s", arg2 & O_APPEND ? "|O_APPEND" : "",
27             arg2 & O_CREAT ? "|O_CREAT" : "" );
28 }
29
30 /*Catch openat system call return*/
31 syscall::openat*:return { 3
32     /*Print pid, uid, gid and return value*/
33     printf(", %d, %d, %d, %d\n", pid, uid, gid, arg1);
34     printf("_____  
");
35     printf("_____  
");
36 }
```

Note-se como o programa desenvolvido se encontra dividido em três blocos principais. No ponto circundado 1 podemos observar o primeiro bloco que indica o que fazer quando o DTrace se encontra perante o probe **dtrace:::BEGIN**. Este probe é um probe especial lançado aquando do início da execução do DTrace e pode portanto ser usado para inicializar variáveis ou imprimir cabeçalhos. Neste caso este probe está a ser usado para imprimir um pequeno cabeçalho.

No segundo ponto circundado estamos já a fazer o traçado de aberturas de ficheiro em tempo real e vemos já a utilização de outro probe distinto chamado **syscall:::entry** que é lançado no início de uma chamada de sistema. Como damos a informação **openat*** então as únicas chamadas de sistema que o nosso programa irá notar são as da chamada de sistema **openat()**.

Vemos ainda o uso da variável "builtin" `execname` que contém o nome do processo a ser executado. Temos também acesso aos argumentos fornecidos ao probe que, para o probe entry, representam os argumentos da chamada de sistema realizada. Neste bloco vemos `arg1` que é um exemplo de um desses argumentos e representa o caminho absoluto para o ficheiro em questão. No entanto, os argumentos do probe são sempre todos inteiros sem sinal (`uint64_t`) e, de forma a obter-se a string dada em "*user-level*" como argumento para a chamada de sistema num local legível pelo DTrace (kernel), usa-se `copyinstr()`. Desta forma indicamos explicitamente que o número `arg1` deve ser interpretado como uma string. `arg2` é também um argumento do probe que simboliza as "*flags*" dadas como argumento à chamada de sistema. Esta variável permite saber as "*flags*" de permissão que estão envolvidas na abertura do ficheiro em questão.

Por fim vemos no terceiro ponto circundado a utilização do probe **syscall:::return**. Como se fornece a indicação **openat*** então este probe é lançado aquando do fim de chamadas de sistema openat(). Novamente temos acesso aos argumento do probe, mas desta vez **arg1** irá simbolizar o valor de retorno da chamada de sistema realizada. Fazemos ainda a impressão

do **PID** do processo, **UID** do utilizador e o **GID** do grupo que são obtidos com variáveis "*builtin*" de nome homónimo. Antes de avançarmos é importante ainda referir que, como o ambiente utilizado é Solaris 11, estes probes lançados para as chamadas de sistema `openat()` englobam também chamadas de sistema `open()` e as suas versões de 64-bit `openat64()` e `open64()`.[2]

B. Testes para Programa 1

Depois de desenvolvido o programa descrito anteriormente este foi testado minuciosamente. Inicialmente observou-se o output depois de correr o seguinte comando: **dtrace -qs programa1.d > output.txt**, com este comando foi possível observar o comportamento do programa com o funcionamento normal do sistema operativo usado. Uma imagem de um excerto do output fornecido pode ser agora observada:

```
| tracer de chamadas ao sistema openat().  
| Executable, Path, Flags, PID, UID, GID, Return  
| nfsmapid, /etc/resolv.conf, O_RDWR, 10327, 1, 12, 8  
| nscd, /system/volatile/repository_door, O_RDWR, 488, 0, 0, 12  
| nscd, /system/volatile/repository_door, O_RDWR, 488, 0, 0, 12  
| nscd, /system/volatile/repository_door, O_RDWR, 488, 0, 0, 12  
| gnome-settings-d, /etc/vfstab, O_RDWR, 20218, 101, 10, 22  
| gnome-settings-d, /etc/vfstab, O_RDWR, 7232, 502, 10, 22  
| gnome-settings-d, /etc/vfstab, O_RDWR, 6983, 0, 0, 23  
| gnome-settings-d, /etc/vfstab, O_RDWR, 6091, 0, 0, 24  
| nfsmapid, /etc/resolv.conf, O_RDWR, 10327, 1, 12, 8  
| utmpd, /system/volatile/utmpx, O_RDWR|O_CREAT, 217, 1, 12, 5  
| utmpd, /system/volatile/utmpx, O_RDWR, 217, 1, 12, 6  
| utmpd, /proc/28596/psinfo, O_RDWR, 217, 1, 12, 7  
| utmpd, /proc/6343/psinfo, O_RDWR, 217, 1, 12, 7  
| utmpd, /proc/28660/psinfo, O_RDWR, 217, 1, 12, 7
```

Ao observarmos este output o programa parece funcionar corretamente, no entanto testou-se ainda com o comando cat de 4 formas distintas. Os comandos utilizados e o output fornecido podem ser agora consultados de forma a provar assim a correção do programa descrito:

```
cat /etc/inittab > /tmp/test72293
```

```
| tracer de chamadas ao sistema operat.

| Executable, Path, Flags, PID, UID, GID, Return

| bash, /tmp/test72293, O_WRONLY|O_CREAT, 28795, 40196, 5018, 4

| cat, /var/lib/ld.config, O_RDWR, 28795, 40196, 5018, -1

| cat, /lib/libc.so.1, O_RDWR, 28795, 40196, 5018, 3

| cat, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28795, 40196, 5018, 3

| cat, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28795, 40196, 5018, 3

| cat, /etc/inittab, O_RDWR, 28795, 40196, 5018, 3
```

```
cat /etc/inittab » /tmp/test72293
```

```
fazer de chamadas ao sistema openat().
```

```
Executable, Path, Flags, PID, UID, GID, Return
```

```
bash, /tmp/test72293, O_WRONLY|O_APPEND|O_CREAT, 28821, 40196, 5018, 4
```

```
cat, /var/lib/ld.so.config, O_RDWR, 28821, 40196, 5018, -1
```

```
cat, /lib/libc.so.1, O_RDWR, 28821, 40196, 5018, 3
```

```
cat, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28821, 40196, 5018, 3
```

```
cat, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28821, 40196, 5018, 3
```

```
cat, /etc/inittab, O_RDWR, 28821, 40196, 5018, 3
```

```
cat /etc/inittab | tee /tmp/test72293
```

```
tracer de chamadas ao sistema openat().  
  
Executable, Path, Flags, PID, UID, GID, Return  
  
tee, /var/lib/ld/ld.config, O_RDWR, 28826, 40196, 5018, -1  
  
tee, /lib/libc.so.1, O_RDWR, 28826, 40196, 5018, 3  
  
tee, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28826, 40196, 5018, 3  
  
tee, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28826, 40196, 5018, 3  
  
tee, /tmp/test72293, O_WRONLY|O_CREAT, 28826, 40196, 5018, 3  
  
cat, /var/lib/ld/ld.config, O_RDWR, 28825, 40196, 5018, -1  
  
cat, /lib/libc.so.1, O_RDWR, 28825, 40196, 5018, 3  
  
cat, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28825, 40196, 5018, 3  
  
cat, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28825, 40196, 5018, 3  
  
cat, /etc/inittab, O_RDWR, 28825, 40196, 5018, 3
```

```
cat /etc/inittab | tee -a /tmp/test72293
```

```
rocer de chamadas do sistema openat().
```

Executable	Path	Flags	PID	UID	GID	Return
cat	/var/lib/ld.so.config	O_RDONLY	28849	40196	5018	-1
cat	/lib/libc.so.1	O_RDONLY	28849	40196	5018	3
cat	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	28849	40196	5018	3
cat	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	28849	40196	5018	3
cat	/etc/inittab	O_RDONLY	28849	40196	5018	3
tee	/var/lib/ld.so.config	O_RDONLY	28850	40196	5018	-1
tee	/lib/libc.so.1	O_RDONLY	28850	40196	5018	3
tee	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	28850	40196	5018	3
tee	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	28850	40196	5018	3
tee	/tmp/test72293	O_WRONLY O_APPEND O_CREAT	28850	40196	5018	3
tee	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	28850	40196	5018	3
tee	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	28850	40196	5018	3
tee	/tmp/test72293	O_WRONLY O_APPEND O_CREAT	28850	40196	5018	3

Depois de observar estes 4 exemplos podemos perceber que realmente o programa se encontra correto. Nota-se, por exemplo, um tratamento correto da impressão das flags usadas ao notar a diferença entre as flags do primeiro e do segundo exemplo para o executável de nome bash, já que no segundo o uso de » leva a que se adicione a flag **O_APPEND**.

```
1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3
4 /**
5 * Tracer de chamadas ao sistema openat().
6 * int openat(int fildes, const char *path, int oflag, mode_t mode);
7 */
8
9 dtrace:::BEGIN {
10     printf("Tracer de chamadas ao sistema openat().\n");
11     printf("_____");
12     printf("_____ \n");
13     printf("| %s, %s, %s, %s, %s, %s\n",
14         "Executable", "Path", "Flags", "PID", "UID", "GID", "Return");
15     printf("_____");
16     printf("_____ \n");
17 }
18
19 /*Catch openat system call entry*/
20 syscall::openat*:entry {
21
22     self->print_ret = (strstr(copyinstr(arg1), "/etc") != NULL) ? 1 : 0;
23     self->path = copyinstr(arg1);
24     self->flag = arg2;
25 }
26
27 /*Catch openat system call return*/
28 syscall::openat*:return
29 self->print_ret == 1/ { 
30
31     /*Print executable name and absolute path*/
32     printf("| %s, %s", execname, self->path);
33
34     printf("%s", self->flag & O_RDONLY ? ", O_RDONLY" :
35             (self->flag & O_WRONLY ? ", O_WRONLY" : ", O_RDWR"));
36     printf("%s", self->flag & O_APPEND ? ", O_APPEND" : "",
37             self->flag & O_CREAT ? ", O_CREAT" : "");
38
39     /*Print pid, uid, gid and return value*/
40     printf(", %d, %d, %d, %d\n", pid, uid, gid, arg1);
41     printf("_____");
42     printf("_____ \n");
43 }
```

Podemos observar no ponto circundado 1 como se usa uma estrutura condicional de forma a registar numa variável o valor 1 ou o valor 0. Caso seja 1 então, aquando do return da chamada de sistema em questão, isso é detetado pelo predicado no ponto circundado 2, sabemos assim que o caminho absoluto contém "/etc" e devemos, portanto, imprimir a sua informação.

D. Testes à extensão do Programa 1

Depois de extendido o programa 1 observou-se o output depois de correr o seguinte comando: **dtrace -qs programa1extendido.d » output.txt**, com este comando foi possível observar o comportamento do programa com o funcionamento normal do sistema operativo usado. Uma imagem de um excerto do output fornecido pode ser agora observada:

```
| /usr/bin/directo -o output > /var/log/fornevedo.log & log4j:WARN|
```

```
| Executable, Path, Flags, PID, UID, GID, Return
```

```
| nfsmapid, /etc/resolv.conf, O_RDWR, 10327, 1, 12, 8
```

```
| gnome-settings-d, /etc/vfstab, O_RDWR, 20218, 101, 10, 22
```

```
| gnome-settings-d, /etc/vfstab, O_RDWR, 7232, 502, 10, 22
```

```
| gnome-settings-d, /etc/vfstab, O_RDWR, 6091, 0, 0, 24
```

```
| gnome-settings-d, /etc/vfstab, O_RDWR, 6983, 0, 0, 23
```

```
| nfsmapid, /etc/resolv.conf, O_RDWR, 10327, 1, 12, 8
```

```
| sshd, /etc/system.d/crypto:fips-140, O_RDWR, 29180, 0, 0, -1
```

```
| sshd, /etc/crypto/pkcs11.conf, O_RDWR, 29180, 0, 0, 4
```

```
| sshd, /etc/system.d/crypto:fips-140, O_RDWR, 29180, 0, 0, -1
```

```
| sshd, /etc/gss/mech, O_RDWR, 29180, 0, 0, 4
```

```
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
```

```
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
```

```
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
```

```
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
```

```
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
```

```
| sshd, /etc/ssh/moduli, O_RDWR, 29180, 0, 0, 4
```

C. Extensão ao Programa 1

Seguidamente teve-se a intenção de modificar o programa criado para que apenas se faça o traceamento de chamadas do sistema openat() relativas a ficheiros com "/etc" no caminho sejam detetados. O programa 1 final ficou então como se segue em baixo:

Vemos realmente que só aparecem entradas em que o caminho absoluto contenha "/etc" no entanto isto ainda não comprova a validade do programa pois, por uma enorme coincidência, pode não ter havido nenhum caso com um caminho absoluto sem "/etc". De forma a provar a correção do programa fez-se correr novamente o comando **cat /etc/inittab | tee -a /tmp/test72293** já que podemos comparar com o output fornecido pela versão anterior do programa. Se esta extensão estiver correta então, olhando para o output anterior, vemos que deve ser imprimida apenas uma entrada. Olhamos agora o output obtido:

```
Tracer de chamadas ao sistema openat().
Executable, Path, Flags, PID, UID, GID, Return
| cat, /etc/inittab, O_RDWR, 29187, 40196, 5018, 3
```

Vemos que realmente apenas imprimiu a entrada esperada, fica então assim provada a correção do programa.

```
1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3
4 /**
5 * Tracer de chamadas ao sistema openat() que conta o numero.
6 * de tentativas e de sucessos.
7 * int openat(int fildes, const char *path, int oflag, mode_t mode);
8 */
9
10 dtrace:::BEGIN { ━━━━①
11     printf("#Tentativas e #sucessos.\n");
12     printf("-----\n");
13     printf("-----\n");
14 }
15
16 /*Catch openat system call entry*/
17 syscall::openat*:entry ━━━━②
18 /($arg2 & O_CREAT) == 0/ {
19     /*Entry that tries to open an existing file*/
20     @try_open[pid] = count();
21     printf("%s on PID: %d, open try.\n", execname, pid);
22 }
23
24 /*Catch openat system call entry*/
25 syscall::openat*:entry ━━━━③
26 /($arg2 & O_CREAT) == O_CREAT/ {
27     /*Entry that tries to create a file*/
28     @try_create[pid] = count();
29     printf("%s on PID: %d, create try.\n", execname, pid);
30 }
31
32 /*Catch openat system call return*/
33 syscall::openat*:return ━━━━④
34 /$arg1 >= 0/ {
35     /*Count successes*/
36     @success[pid] = count();
37     printf("%s on PID: %d, success.\n", execname, pid);
38 }
39
40 /*print results*/
41 dtrace:::END { ━━━━⑤
42     /*Print*/
43     printf("-----\n");
44     printf("-----\n");
45     printf(" | %8s | %5s | %5s | %10s | \n",
46             "PID", "Open", "Create", "Successes");
47     printf(" | %8d | %05d | %05d | %010d | \n",
48             @try_open, @try_create, @success);
49     printf("-----\n");
50     printf("-----\n");
51     printf("-----\n");
52     printf("-----\n");
53
54     /*Clear Counters*/
55     clear(@try_open); ━━━━⑥
56     clear(@try_create); ━━━━⑦
}
```

E. Desenvolvimento do Programa 2

Para um segundo estudo quis-se desenvolver um programa que, para os processos que estão a correr no sistema, apresenta os valores obtidos em cada iteração das seguintes estatísticas:

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

Apresenta-se agora o programa desenvolvido:

Note-se como no ponto circundado 1 temos novamente um bloco que indica o que fazer ao encontrar um probe **dtrace:::BEGIN**. Novamente este bloco é usado apenas para imprimir um cabeçalho.

De seguida observamos que temos dois blocos para o probe **syscall:::entry** relativo à chamada de sistema **openat()**. No bloco do ponto circundado 2 é usado um predicado que verifica que a chamada de sistema é relativa à abertura de um ficheiro já existente, como se pode ver pela condição (**arg2 & O_CREAT**) == 0. No ponto circundado 3 vemos outro bloco que verifica se a chamada de sistema refere a criação de um ficheiro, como vemos pelo predicado com a condição (**arg2 & O_CREAT**) == O_CREAT. Depois de identificar que tipo de acesso ao ficheiro se está a tentar fazer, é necessário atualizar essa informação nos respetivos contadores que irão acumular o valor de tentativas de cada uma destas duas possibilidades. Estas tentativas de abertura e criação são registadas em duas variáveis distintas: **try_create** para a tentativa de criação e **try_open** para a tentativa de abertura. Vemos esta atualização destes acumuladores no ponto circundado 7. Observa-se que **count()** é usado com o intuito de contar o número de tentativas respetivas a cada variável e que, cada variável, tem atribuído a cada contador o **PID** do processo responsável pela chamada de sistema. Esta agregação de cada PID ao seu contador observa-

se aquando da atualização dos contadores como se evidencia, por exemplo, pelo seguinte troço de código: `@try_create[pid] = count();`

No ponto circundado 4 temos que, ao encontrar um probe **syscall:::return** de uma chamada de sistema openat(), se verifica com recurso a um predicado com a condição **arg1 >= 0** se a chamada de sistema foi realizada com sucesso ou não (caso tenha devolvido um valor superior a -1 sabemos foi bem sucedida, sendo **arg1** o valor de retorno) e, caso seja um sucesso, usamos novamente **count()** para atualizar um outro acumulador, a variável **success** que se encontra, assim como os outros acumuladores mencionados anteriormente, relacionado com o PID do processo que realizou a chamada de sistema. Note-se agora que sempre que há a atualização de um contador é feito um print que a descreve indicando também o **PID** e o **nome do executável** correspondentes. Desta forma é possível no fim validar os resultados acumulados obtidos como iremos ver na próxima secção deste documento.

Por fim, no ponto circundado 5, observamos o que fazer ao encontrar um probe do tipo **dtrace:::END**. Aqui será feita a impressão dos contadores que foram sendo atualizados ao longo da execução do programa sendo que depois da impressão, no ponto circundado 6, estes contadores são eliminados com o uso de **clear**.

```
#Tentativas e #sucessos.

gnome-settings-d on PID: 20218, open try.
gnome-settings-d on PID: 20218, success.
gnome-settings-d on PID: 7232, open try.
gnome-settings-d on PID: 7232, success.
gnome-settings-d on PID: 6091, open try.
gnome-settings-d on PID: 6091, success.
gnome-settings-d on PID: 6983, open try.
gnome-settings-d on PID: 6983, success.
nfsmapid on PID: 10327, open try.
nfsmapid on PID: 10327, success.
nscd on PID: 488, open try.
nscd on PID: 488, success.
init on PID: 1, open try.
init on PID: 1, success.
init on PID: 1, create try.
init on PID: 1, success.
init on PID: 1, create try.
init on PID: 1, success.
init on PID: 1, open try.
init on PID: 1, success.
nfsmapid on PID: 10327, open try.
nfsmapid on PID: 10327, success.
nscd on PID: 488, open try.
nscd on PID: 488, success.
^C


```

	PID	Open	Create	Successes
	6091	1	0	1
	6983	1	0	1
	7232	1	0	1
	20218	1	0	1
	488	2	0	2
	10327	2	0	2
	1	2	2	4

É possível ver na imagem impressões intermédias que vão indicando as tentativas e os sucessos que se vão encontrando em tempo real ao longo da execução do programa assim como o seu PID e nome de executável respetivos. Depois de o programa terminar vemos uma tabela com os valores agregados que podem ser confirmados com as impressões anteriores. Se o leitor prestar atenção pode verificar que de facto as impressões estão coerentes, provando assim a correção do programa.

F. Testes para Programa 2

Depois de desenvolvido o programa descrito este foi testado com o seguinte comando **dtrace -qs programa2a.d**. Segue-se uma imagem do output obtido que comprova a sua correção:

G. Extensão ao Programa 2

Seguidamente teve-se a intenção de modificar o programa 2 para que repetidamente, com um período especificado em segundos passado como argumento da linha de comandos, imprimisse a seguinte informação:

- 1 - Hora e dia atual em formato legível.
- 2 - As estatísticas recolhidas por PID e respetivo nome.

Para isto extendeu-se o programa 2 da seguinte forma:

```

1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3
4 /**
5  * Tracer de chamadas ao sistema openat() que conta o numero.
6  * de tentativas e de sucessos.
7  * int openat(int fildes, const char *path, int oflag, mode_t mode),
8  */
9
10 dtrace:::BEGIN {
11     printf("#Tentativas e #sucessos.\n");
12     printf("_____\n");
13     printf("_____ \n");
14 }
15
16 /*Catch openat system call entry*/
17 syscall::openat:entry
18 /($arg2 & O_CREAT) == 0/ {
19     /*Entry that tries to open an existing file*/
20     @try_open[execname, pid] = count();
21     @try_open_global[execname, pid] = count();
22 }
23
24 /*Catch openat system call entry*/
25 syscall::openat:entry
26 /($arg2 & O_CREAT) == O_CREAT/ {
27     /*Entry that tries to create a file*/
28     @try_create[execname, pid] = count();
29     @try_create_global[execname, pid] = count();
30 }
31
32 /*Catch openat system call return*/
33 syscall::openat:return
34 /$arg1 >= 0/ {
35     /*Count successes*/
36     @success[execname, pid] = count();
37     @success_global[execname, pid] = count();
38 }
39
40 tick-S1s { ①
41     /*Print*/
42     printf("_____\n");
43     printf("_____ \n");
44     printf("$%20s | %8s | %5s | %7s | %10s |\n",
45           "Executable", "PID", "Open", "Create", "Successes");
46     printf("_____ \n");
47     printf("_____\n");
48     printf("$%20s | %8d | %8d | %8d | %8d |\n",
49           @try_open, @try_create, @success);
50     printf("TIME: %Y\n", walltimestamp);
51     printf("_____\n");
52     printf("_____ \n");
53
54     /*Discard last measurements*/
55     trunc(@try_create);
56     trunc(@try_open); ②
57     trunc(@success);
58 }
59
60 /*print results*/
61 dtrace:::END {
62     /*Print*/
63     printf("_____\n");
64     printf("_____ \n");
65     printf("$%20s | %8s | %5s | %7s | %10s |\n",
66           "Executable", "PID", "Open", "Create", "Successes");
67     printf("_____ \n");
68     printf("_____\n");
69     printf("$%20s | %8d | %8d | %8d | %8d |\n",
70           @try_open_global, @try_create_global, @success_global);
71     printf("_____\n");
72     printf("_____ \n");
73
74     /*Clear Counters*/
75     clear(@try_open);
76     clear(@try_create);
77     clear(@success); ③
78     clear(@try_open_global);
79     clear(@try_create_global);
80     clear(@success_global);
81 }

```

Note-se a adição do bloco no ponto circundado 1. Este bloco indica o que fazer de **\$1** em **\$1** segundos, sendo **\$1** o valor em segundos entregue pela linha de comandos, como era pretendido. No interior deste bloco vemos que são imprimidos os valores encontrados desde a última medição à **\$1** segundos atrás. É também apresentada a data permitindo assim comprovar a correção desta funcionalidade. Esta data é obtida através de **walltimestamp**. De forma a reiniciar os contadores note-se como, no ponto circundado 2, se recorre

ao uso de **trunc()**. Estas variáveis apresentadas reiniciadas de **\$1** em **\$1** segundos são relativas somente ao tempo desde a última medição, por isso, é necessário ter ainda outras variáveis que guardem os valores globais (como era feito na versão anterior). Ao olhar o código é possível notar que sempre que um acumulador é atualizado para a apresentação de **\$1** em **\$1** segundos, um acumulador global também o é para assim, quando se encontrar o probe **dtrace:::END** estes acumuladores globais serem apresentados corretamente. Note-se também que, enquanto na versão anterior cada acumulador estava associado ao **PID**, agora encontra-se associado não só ao **PID**, mas também ao **nome do executável**, como podemos ver pelo seguinte troço de código: **@success[execname, pid] = count();**. Desta forma é possível no fim mostar os valores agregados por estas duas variáveis, como era pretendido nos requisitos apresentados anteriormente.

Por fim, no ponto circundado 3, todas as variáveis agregadas a serem eliminadas com o uso de **clear()**.

H. Testes à extensão do Programa 2

Vemos agora o output do programa de forma a comprovar a sua correção, desta vez é usado o comando **dtrace -qs programa2.b.d 5**:

#Tentativas e #sucessos.				
Executable	PID	Open	Create	Successes
TIME: 2017 Apr 3 23:46:25				
Executable	PID	Open	Create	Successes
gnome-settings-d	7232	1	0	1
gnome-settings-d	20218	1	0	1
dtrace	29344	2	0	2
nsqd	488	2	0	2
TIME: 2017 Apr 3 23:46:30				
Executable	PID	Open	Create	Successes
gnome-settings-d	6091	1	0	1
gnome-settings-d	6983	1	0	1
TIME: 2017 Apr 3 23:46:35				
Executable	PID	Open	Create	Successes
nfsmapid	10327	1	0	1
TIME: 2017 Apr 3 23:46:40				
^C				
Executable	PID	Open	Create	Successes
gnome-settings-d	6091	1	0	1
gnome-settings-d	6983	1	0	1
gnome-settings-d	7232	1	0	1
gnome-settings-d	20218	1	0	1
nfsmapid	10327	1	0	1
dtrace	29344	2	0	2
nsqd	488	2	0	2

Como o argumento dado ao nosso programa foi o número 5 vemos que a cada 5 segundos são imprimidos os valores obtidos desde a última medição (há 5 segundos atrás). Prova-se a correção destas medições intermédias ao olharmos para o tempo imprimido que realmente mostra cada medição separada por um intervalo de 5 segundos. Vemos ainda o caso de nos últimos 5 segundos não ter ocorrido nenhuma chamada de sistema, que é o caso logo da primeira medição intermédia já que apenas se imprimiu o tempo. Notamos depois que o programa imprime no fim os valores agregados, como fazia na versão anterior. Podemos comprovar a correção destes valores ao olhar para as medições intermédias já que vemos que há

processos que aparecem mais do que uma vez e os seus valores aparecem somados no final como era suposto.

I. Desenvolvimento do Programa 3

Por último houve a intenção de desenvolver ainda um outro programa que permitisse replicar o comportamento do seguinte comando: **strace -c <programa>**. Onde a opção **-c** permite contabilizar o número de ocorrências de cada chamada ao sistema e o tempo despendido. Segue-se o código desenvolvido:

```
1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3
4
5 dtrace:::BEGIN {
6     printf("#Chamadas de sistema.\n");
7     printf("_____\n");
8     printf("_____ \n");
9     self->started = 0;
10 }
11
12 /*Catch system call entry*/
13 syscall:::entry
14 /execname == $$1/ { _____ ①
15     @times_called[probefunc] = count();
16     self->started = timestamp;
17 }
18
19 /*Catch system call return*/
20 syscall:::return
21 /*Ignore also syscalls where entry was not traced*/
22 /execname == $$1 && self->started!=0/ { _____ ②
23     @syscall_time_elapsed[probefunc] = sum((timestamp - self->started));
24     self->started = 0;
25 }
26
27 /*print results*/
28 dtrace:::END {
29     /*Print*/
30     printf("Program: %s\n", $$1); _____ ③
31     printf("_____\n");
32     printf("_____\n");
33     printf("| %15s | %15s | %s\n",
34         "Sys Call", "times Called", "Time spent");
35     printf("_____\n");
36     printf("_____\n");
37     printf("| %15s | %15d | %d ns\n",
38         @times_called, @syscall_time_elapsed);
39     printf("_____\n");
40     printf("_____\n");
41
42     /*Clear Counters*/
43     clear(@times_called); _____ ④
44     clear(@syscall_time_elapsed);
45 }
```

No ponto circundado 1 vemos um cabeçalho a ser imprimido assim como a atribuição da variável **started** da estrutura **self** com o valor 0. Esta variável servirá para, quando uma chamada ao sistema começar, se poder registar a sua data de início. Torna-se importante defini-la a zero de imediato, pois, se este nosso programa começar depois de o programa argumento ter começado pode acontecer ser detetado o seu fim sem nunca ter sido detetado o seu início. Isto traria problemas pois o tempo que ficaria registado que a chamada ao sistema demorou seria erróneo, assim como o seu número de ocorrências já que este apenas é atualizado quando a chamada ao sistema é iniciada. Ao inicializar a variável **self->started** a 0 podemos verificar o seu valor com um predicado, como vemos no ponto circundado 3, com a condição **self->started!=0** e assim, não registar dados o tempo de fim da chamada ao sistema se um tempo inicial não tiver sido registado. No ponto circundado 3 vemos ainda como o tempo é agregado por chamada ao sistema (@**syscall_time_elapsed[probefunc]**), sendo o nome da chamada identificado pela variável "builtin" **probefunc**. Este tempo é atualizado com o uso de **sum()** que recebe como argumento o tempo que

deve somar. No final deste bloco a variável **self->started** é novamente inicializada a 0 para assim o processo descrito se poder repetir.

No ponto circundado 2 vemos a variável agregada que irá acumular o número de vezes que cada chamada ao sistema foi invocada com o uso de **count()** e vemos o tempo inicial da chamada de sistema a ser registado em **self->started**.

No ponto circundado 4 observamos os valores agregados a serem imprimidos aquando da terminação do programa. No ponto circundado 5 vemos essas variáveis agregadas a serem por fim eliminadas com o uso de **clear()**.

Note-se ainda que houve a necessidade de, sempre que se refere a variável recebida como argumento na linha de comandos (o nome do programa a observar), este é referenciado da seguinte forma: **\$\$1**. Ao usarmos dois \$ estamos a indicar explicitamente que o argumento recebido deve ser interpretado como uma string. Temos interesse que isto aconteça para assim esta string argumento poder ser comparada com a string na variável **execname** nos pontos circundados 2 e 3.

J. Testes para Programa 3

Por fim testa-se o terceiro e último programa com o seguinte comando: **dtrace -qs programa3.d ls**. Note-se que ao fornecer **ls** como argumento, apenas serão imprimidas informações relativas a chamadas de sistema invocadas por um programa de nome **ls**. De forma a testar fez-se correr concorrentemente com o comando anterior o seguinte comando: **ls -l**. O output resultante pode agora ser consultado:

Sys Call	times Called	Time spent
rexit	1	0 ns
getpid	1	1735 ns
getrlimit	1	2773 ns
systeminfo	1	5900 ns
read	1	20068 ns
munmap	1	25508 ns
fcntl	2	4276 ns
setcontext	2	6650 ns
ioctl	2	26120 ns
getdents	2	41593 ns
doorfs	4	103220 ns
mmapobj	4	124016 ns
getuid	5	9511 ns
memcntl	6	50632 ns
resolvepath	6	72009 ns
close	7	25334 ns
mmap	7	47817 ns
brk	11	44880 ns
openat	11	157458 ns
pathconf	22	62058 ns
write	22	162218 ns
acl	44	217631 ns
fstatat	53	236580 ns

Ao observar o output temos a prova da correção do programa sendo que apenas as chamadas de sistema invocadas no programa em questão se encontram detalhadas com o número de vezes que foram invocadas e o tempo total despendido para cada.

III. CONCLUSÃO E TRABALHO FUTURO

Dado por concluído o projeto pensa-se ter sido mostrado um bom conhecimento da ferramenta DTrace e do seu uso. Ao longo do desenvolvimento do projeto foi-se tornando cada vez mais fácil a programação dos programas, permitindo assim talvez melhores práticas de programação nos programas finais do que o que se observa nos programas iniciais. Agora, depois de concluídos todos os programas propostos, talvez fosse interessante para trabalho futuro, voltar a implementar cada um sem olhar para esta implementação aqui apresentada. Certamente que, agora com mais conhecimento no assunto, eles ficariam mais sintéticos e mais fáceis de entender por olhos estrangeiros ao seu processo de desenvolvimento. Em síntese, conclui-se que o projeto foi um sucesso, visto que, como indicado na introdução, o seu objetivo era o de desenvolver competências com o uso da ferramenta DTrace e, como mencionado no parágrafo anterior, foi exatamente isso que aconteceu e encontra-se aqui provado com os programas desenvolvidos e explicados.

IV. BIBLIOGRAFIA

REFERÊNCIAS

- [1] <http://www.oracle.com/technetwork/server-storage/solaris11/documentation/dtrace-cheatsheet-1930738.pdf>
- [2] <http://dtrace.org/blogs/brendan/2011/11/09/solaris-11-dtrace-syscall-provider-changes/>

TPC4: Análise de Desempenho com Benchmark Ativo e Passivo

- Engenharia dos sistemas de Computação -

autor: Daniel Malhadas

Resumo—O presente documento apresenta um estudo aprofundado sobre duas técnicas distintas de análise de desempenho, sendo elas benchmark ativo e benchmark passivo. De forma a entender as diferenças entre os resultados obtidos com cada uma destas técnicas ambas são usadas para analisar o desempenho de aplicações nas mesmas condições (como aplicação de referência será usado IOzone), sendo depois os resultados comparados e avaliados tornando-se possível observar as diferenças e vantagens do método ativo para o método passivo. Note-se ainda que todo este estudo foi realizado em ambiente Solaris 11 e, por essa razão, os dados instrumentados serão relativos somente a esse ambiente.

Index Terms—Análise de Desempenho, Benchmark Ativo, Benchmark Passivo, IOzone, DTrace, Solaris11, Computação Paralela e Distribuída.

I. INTRODUÇÃO

A. Contextualização e Motivação

Com este projeto pretende-se recorrer ao uso de IOzone enquanto aplicação de referência para fazer a avaliação de desempenho para uma multiplicidade de operações, incluindo entre outras: Ler/escrever, reler/reescrever, ler/escrever para trás, etc., em diferentes sistemas de ficheiros (ufs, zfs, nfs). Esta ferramenta irá permitir-nos obter resultados relativos a um benchmark passivo. Será também realizado um benchmark ativo da aplicação de referência IOzone com recurso a DTrace e outras ferramentas auxiliares cujos resultados serão comparados com os do benchmark passivo tornando-se possível observar as diferenças e as vantagens do método ativo em relação ao método passivo. Por consequência acaba também por se fazer uma análise das desvantagens do uso de IOzone em comparação com DTrace para benchmarking.

Estudar estes métodos de benchmark e compreender as suas diferenças é essencial no desenvolvimento de aplicações onde a performance é importante já que muitos sistemas operativos são afinados para terem um bom desempenho de entradas e saídas de dados em algumas das aplicações mais utilizadas, mas tal afinação pode não ser adequada para outras aplicações[1]. Outras vezes, os padrões de acesso aos dados podem mudar, por exemplo passando de leituras e escritas sequenciais para leituras e escritas com padrões aleatórios, o que se pode traduzir em aumentos significativos dos tempos esperados para as operações nos sistemas de entrada e saída de dados[1]. Logo o uso destas técnicas de benchmarking torna-se relevante de forma a detetar estas situações para, com isso em mente, construir melhores aplicações que tirem proveito de características da máquina em questão. Estas técnicas de benchmarking permitem também que um possível comprador

da máquina entenda se esta se adapta ou não para os propósitos que pretende, sendo assim possível uma decisão ponderada sobre se deve ou não comprar essa máquina[1].

B. Em que Consiste Benchmark Passivo?

Com o benchmark passivo, um determinado teste é deixado livremente a correr e a análise dos resultados executada por uma dada ferramenta é feita apenas no final sem que seja realizada uma contra-análise que permita atestar a veracidade dos resultados obtidos.

Neste projeto usaremos os resultados da própria ferramenta usada como teste de referência, IOzone, de forma a realizar o benchmark passivo.

C. Em que Consiste Benchmark Ativo?

A ideia do benchmark ativo é analisar a aplicação enquanto o teste de referência está a correr (e não apenas depois de concluído), usando outras ferramentas. Desta forma é possível não apenas confirmar a correção dos testes realizados, mas, ao mesmo tempo, obter informações específicas e relevantes sobre o sistema ou aplicação em análise que permitam, por exemplo, identificar os reais limitadores do sistema em teste, ou do próprio benchmark.

Neste projeto usaremos a ferramenta DTrace e outras auxiliares de forma a realizar este benchmark ativo.

II. CONSIDERAÇÕES INICIAIS

A. Caracterização do Ambiente de Testes

De forma a realizar os testes propostos no capítulo anterior foi escolhido como ambiente de testes o sistema operativo Solaris11 e, por essa razão, os dados instrumentados serão relativos somente a esse ambiente. Mais concretamente, a máquina utilizada encontra-se caracterizada pormenorizadamente na tabela que se segue:

Sistema Operativo	Solaris11
Fabricante	Intel
Modelo do CPU	Dual Intel(R) Xeon(R) X5650
Microarquitetura do CPU	Westmere
Frequência do Clock	2.67 GHz
#Cores	8, com 16 threads (2 por core)
Cache	L1: 32kB por core, L2: 256kB por core, L3: 20480kB partilhada
Largura de Banda de acesso à memória	59.7GB/s
Memória RAM	64GB
FileSystems	zfs, ufs, nfs, smb, autofs, smbfss

read, write	Estes testes medem a performance de escrita num novo ficheiro (write) e de leitura de um ficheiro já existente (read). Quando se escreve num novo ficheiro temos um overhead chamado 'metadata' que representa o impacto na performance de ter de guardar os dados escritos no ficheiro e de registar onde os dados se localizam na mídia de armazenamento.
re-read, re-write	Estes testes medem a performance de leitura de um ficheiro que foi lido recentemente (re-read) ou de escrita num ficheiro que já existe (re-write). É normal que estes testes resultem numa melhor performance que um simples read ou write já que, para o re-read, vamos ter os dados necessários em cache e, para o re-write, já vamos ter o overhead 'metadata' previamente realizado.
read backwards	Este teste mede a performance de ler um ficheiro para trás. Embora muitos sistemas operativos estejam otimizados para leituras para a frente, mais convencionais, poucos estão para uma leitura mais fora do normal como esta para trás.
read striped	Este teste mede o desempenho de ler um ficheiro com um acesso espaçado. Um exemplo seria: Ler no offset zero para um comprimento de 4 Kbytes, em seguida, procurar 200 Kbytes e ler para um comprimento de 4 Kbytes, depois, procurar 200 Kbytes e assim por diante. Aqui o padrão é ler 4 Kbytes e depois procurar 200 Kbytes e repetir o padrão. Este é um comportamento típico para aplicações que têm estruturas de dados contidas dentro de um ficheiro e estão a tentar aceder a uma região específica da estrutura de dados. Este comportamento de acesso também pode, por vezes, produzir anomalias de desempenho interessantes.
fread, fwrite, freread, frewrite	Estes testes fazem o mesmo que os testes de read, write, reread, rewrite, no entanto utilizam a função de biblioteca fwrite(). Desta forma usa-se um buffer ao nível do utilizador e reduz-se as chamadas ao sistema, o que pode melhorar bastante a performance.
random read, random write, random mix	Estes testes medem a performance de ler um ficheiro com vários acessos aleatórios (random read), escrever um ficheiro com vários acessos aleatórios (random write) e de escrever e ler um ficheiro com vários acessos aleatórios (random mix). O impacto deste tipo de atividades
record rewrite	Este teste mede o desempenho de escrever e reescrever num determinado local dentro de um ficheiro. Se o tamanho do local é pequeno o suficiente para caber na cache de dados do CPU, o desempenho é muito alto. No entanto níveis diferentes de desempenho serão obtidos noutras situações.

B. Em que consiste a ferramenta IOzone?

IOzone é uma ferramenta para benchmark de sistemas de ficheiros, sendo que para efetuar esse benchmarking executa e avalia de forma passiva diversas operações com ficheiros[1]. A ferramenta encontra-se em várias máquinas e sistemas operativos distintos não sendo portanto específica a nenhum tipo de ambiente.

IOzone é útil para analisar a performance de uma dada máquina relativamente a operações com ficheiros, nomeadamente:[1]

async I/O	Teste especializado que mede a performance do mecanismo POSIX async I/O utilizando coisas como aio_write(), aio_read() e aio_error().
mmap	Teste especializado que mede a performance do uso do mecanismo mmap() para operações de I/O.

Como mencionado no capítulo anterior, por vezes uma dada máquina e um dado sistema operativo podem estar otimizados para determinados tipos de aplicações específicos sendo que o uso de ferramentas como IOzone permite detetar as situações para as quais a máquina se encontra otimizada permitindo, com isso em mente, construir melhores aplicações que tirem proveito dessas características da máquina em questão ou até de um possível comprador da máquina entender se esta se adapta ou não para os propósitos que pretende, sendo assim possível uma decisão ponderada sobre se deve ou não comprar essa máquina[1].

Note-se ainda que esta ferramenta permite o uso de diversas flags diferentes. No Anexo A pode encontrar-se então uma tabela que menciona cada flag existente e uma breve descrição sendo que as flags utilizadas serão futuramente referenciadas sem uma explicação adicional, fica portanto aqui uma tabela que permite ao leitor entender o uso de uma determinada flag.

C. Em que consiste a ferramenta DTrace?

A Oracle Solaris DTrace é uma ferramenta de rastreio/traçado avançada usada para testar troços problemáticos de um programa em tempo real. Para isto, esta ferramenta permite observar questões de performance tanto em pequenas aplicações como do próprio sistema operativo em si de forma dinâmica e segura permitindo a quem a utilize a identificação de problemas que seriam difíceis de detetar com outras ferramentas similares por estarem demasiado disfarçados sobre várias camadas de software.

A ferramenta permite também a instrumentação de várias estatísticas em tempo real relativas ao programa a testar. Estatísticas como: consumo de memória, tempo de CPU despendido, que chamadas de função foram realizadas, etc.[2] De forma a poder traçar o que está a acontecer, a ferramenta DTrace recorre à monitorização de diversos "sinais"/"pontos de interesse" marcados no sistema operativo a que se dá o nome de **probes**. Estes probes são lançados em momentos específicos e, cabe ao utilizador da ferramenta DTrace, saber quais os probes que deve intercetar e o que fazer quando os intercetar de forma a alcançar os resultados que pretende. Segue-se uma tabela com os probes disponíveis em ambiente Solaris 11 e uma breve descrição: [3]

Common DTrace Providers	Description
dtrace	Start, end and error probes
syscall	Entry and return probes for all system calls
fbt	Entry and return probes for all kernel calls
profile	Timer driven probes
proc	Process creation and lifecycle probes
pid	Entry and return probes for all user-level processes
io	Probes for all I/O related events
sdt/usdt	Developer defined probes at arbitrary locations/names within source code for kernel and user-level processes
sched	Probes for scheduling related events
lockstat	Probes for locking behavior within the operating system

III. MEDIÇÕES DE DESEMPENHO - BENCHMARK PASSIVO

A. Estudo do Ambiente Disponível

De forma a fazer um benchmark passivo da máquina descrita anteriormente utiliza-se, como mencionado anteriormente, a ferramenta IOzone. Inicialmente determina-se a versão instalada com o comando **iozone -v** o que nos permite saber que a versão presente na máquina em questão é a versão 3.434, como é possível comprovar na seguinte imagem que demonstra o output do comando indicado:

```
a72293@solarisEdu:~/a72293$ /opt/csw/bin/iozone -v
'Iozone' Filesystem Benchmark Program

Version $Revision: 3.434 $
Compiled for 64 bit mode.

Original Author: William Norcott (wnorcott@us.oracle.com)
        4 Dunlap Drive
        Nashua, NH 03060

Enhancements: Don Capps (capps@iozone.org)
        7417 Crenshaw
        Plano, TX 75025

Copyright 1991, 1992, 1994, 1998, 1999, 2002 William D. Norcott

License to freely use and distribute this software is hereby granted
by the author, subject to the condition that this copyright notice
remains intact. The author retains the exclusive right to publish
derivative works based on this work, including, but not limited to,
revised versions of this work

Other contributors:
Don Capps      (Network Appliance)  capps@iozone.org

a72293@solarisEdu:~/a72293$
```

De seguida, com o comando **df -h** foi possível determinar que sistemas de ficheiros estão montados no sistema, podendo assim observar o sistemas sobre os quais os testes serão realizados. Segue-se uma imagem demonstrativa do resultado obtido para este comando:

```
a72293@solarisEdu:~/a72293$ df -h
Filesystem      Size  Used Available Capacity  Mounted on
rpool/ROOT/solaris   109G  21G    60G  26%   /
/devices          0K   0K    0K  0%   /devices
/dev              0K   0K    0K  0%   /dev
ctfs              0K   0K    0K  0%   /system/contract
proc              0K   0K    0K  0%   /proc
mntonab           0K   0K    0K  0%   /etc/mntonab
swap              8.6G  1.6M   8.6G  1%   /system/volatile
objfs             0K   0K    0K  0%   /system/object
sharefs            0K   0K    0K  0%   /etc/dfs/sharetab
/usr/lib/libc/libc_hwcap1.so.1
                 81G  21G    60G  26%   /lib/libc.so.1
fd                0K   0K    0K  0%   /dev/fd
rpool/ROOT/solaris/var
                 109G 363M   60G  1%   /var
swap              9.1G  465M   8.6G  6%   /tmp
rpool/VARSHARE     109G  2.1M   60G  1%   /var/share
rpool/export       109G  32K   60G  1%   /export
rpool/export/home  109G  54K   60G  1%   /export/home
rpool/export/home/a57812
                 109G  35K   60G  1%   /export/home/a57812
rpool/export/home/a64365
                 109G  35K   60G  1%   /export/home/a64365
rpool/export/home/a70377
                 109G  35K   60G  1%   /export/home/a70377
rpool/export/home/a70719
                 109G  20M   60G  1%   /export/home/a70719
rpool/export/home/a70880
                 109G  2.1M   60G  1%   /export/home/a70880
rpool/export/home/a71184
                 109G  35K   60G  1%   /export/home/a71184
rpool/export/home/a71240
                 109G  35K   60G  1%   /export/home/a71240
rpool/export/home/a71492
                 109G  3.1M   60G  1%   /export/home/a71492
rpool/export/home/a71874
                 109G  62K   60G  1%   /export/home/a71874
rpool/export/home/a72227
                 109G  53K   60G  1%   /export/home/a72227
rpool/export/home/a72293
                 109G 105K   60G  1%   /export/home/a72293
rpool/export/home/a72424
                 109G  4.7M   60G  1%   /export/home/a72424
rpool/export/home/a72502
                 109G  35K   60G  1%   /export/home/a72502
rpool/export/home/a73269
                 109G 110K   60G  1%   /export/home/a73269
rpool/export/home/admin
                 109G  975K   60G  1%   /export/home/admin
rpool/export/home/amp
                 109G  11G   60G  16%  /export/home/amp
rpool/export/home/pg31384
                 109G  15M   60G  1%   /export/home/pg31384
rpool/export/home/pg31544
                 109G  35K   60G  1%   /export/home/pg31544
rpool/export/home/pg32995
                 109G  35K   60G  1%   /export/home/pg32995
rpool/export/home/vo
                 109G  35K   60G  1%   /export/home/vo
rpool
                 109G  4.9M   60G  1%   /rpool
rpool/VARSHARE/zones
                 109G  31K   60G  1%   /system/zones
rpool/VARSHARE/pkg
                 109G  32K   60G  1%   /var/share/pkg
rpool/VARSHARE/pkg/repositories
                 109G  31K   60G  1%   /var/share/pkg/repositories
172.27.6.237:/storage/san/v2p1
                 1.8T 2016G  1.5T 12%  /share/jade
/dev/dsk/c3d0s0
                 147G  2.3G  143G  2%   /discoHitachi
a72293@solarisEdu:~/a72293$
```

Dentro dos sistemas montados destacam-se os sistemas montados em **/discoHitachi** e **/share/jade** que estão formatados, respetivamente, com sistema de ficheiros **UFS** e **NFS**, permitindo assim testar resultados para estes dois tipos de sistemas de ficheiros. Podemos ainda testar o sistema de ficheiros **ZFS** montado noutras partições presentes na máquina como, por exemplo, em **home/a72293**. Ficam assim estabelecidos os três tipos de sistemas de ficheiros que serão testados ao longo deste documento.

Torna-se agora importante referir que, para cada um destes três tipos de sistemas de ficheiro disponíveis teve-se a intenção de realizar três testes principais com IOzone:

• 1 -

Primeiro Será testado em ambiente single-threaded/single-process os seguintes testes: write e rewrite, read e reread, random read e random write, backwards read e stride. Para isto as flags **-i0 -i1 -i2 -i5** serão utilizadas. Com este teste poder-se-á observar em pormenor a performance da máquina na escrita de um ficheiro não existente e na escrita num ficheiro já existente, a leitura de um ficheiro não lido anteriormente e da leitura de um ficheiro lido recentemente, a leitura e escrita de um ficheiro em pontos aleatórios e a leitura de um ficheiro com um certo espaçamento. Será apresentado

um gráfico individual para cada um destes testes.

• 2 -

De seguida utilizar-se-á as flags **-i0 -i1 -i2 -i5 -l# -u#** de maneira a realizar os testes em ambiente multi-threaded/multi-process. Desta forma podemos ver como a máquina se comporta em paralelo. Tendo em conta que máquina tem oito cores físicos então torna-se relevante testar desde um a oito threads/processos de forma a ver quão bem escala com o número de cores físicos.

• 3 -

Por último, depois de se ter um melhor conhecimento da performance da máquina, utilizar-se-á as flags **-i0 -i1 -i2 -i3 -i5 -I** de maneira a realizar os testes utilizando DIRECT I/O em todas as operações de ficheiros. Desta forma podemos ver como a máquina se comporta se tiver que ir sempre ao disco em cada operação de ficheiros.

Infelizmente o não foi possível realizar o terceiro teste para o sistema de ficheiros ZFS, podemos ver no seguinte output que **directio** não se encontrou disponível:

```
a72293@solarisEdu:~/a72293$ /opt/csw/bin/iozone -a -+u -i0 -i1 -i2 -i5 -R z2 -s524288 -I
IOzone: Performance Test of File I/O
Version ERevision: 3.434 E
Compiled For 64 bit mode.
Build: Solaris10

Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
               Al Slater, Scott Rhine, Mike Wisner, Ken Goss
               Steve Landheer, Brad Smith, Mark Kelly, Dr. Alain CYR,
               Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
               Jean-Marc Zuconi, Jeff Blomberg, Benny Halevy, Dave Boone,
               Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
               Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
               Vangel Bojaxhi, Ben England, Vikentsi Lapa,
               Alexey Skidanov.

Run began: Wed May  3 18:44:02 2017

Auto Mode
CPU utilization Resolution = 0.000 seconds.
CPU utilization Excel chart enabled
Excel chart generation enabled
File size set to 524288 kB
DIRECT feature enabled
Command line used: /opt/csw/bin/iozone -a -+u -i0 -i1 -i2 -i5 -R z2 -s524288 -I
Output is in kBytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 kBytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.

random      random      bkwd      r
KB reclen  fread     write    rewrite   read   reread   read   write   read   re
fread      524288      4
directio not available.
```

Entenda-se que nenhuma informação sobre a cache da máquina em questão será fornecida à ferramenta IOzone pois esta serve para testar o desempenho da máquina para aplicações potencialmente genéricas que não irão, por razões óbvias, conhecer esses valores. Ao não fornecer valores da cache (como o tamanho de uma linha e o tamanho de um set) simulamos melhor o funcionamento com aplicações genéricas e os resultados são assim mais relevantes. Quando nenhum valor é fornecido, para esta máquina, o IOzone assume um tamanho de linha de cache de 32 bytes e um tamanho de set de 1024 KBytes, como podemos ver na seguinte imagem que demonstra um exemplo do output inicial da ferramenta IOzone:

```
Auto Mode
Command line used: /opt/csw/bin/iozone -a -i0 -i2 -b ver2.csv
Output is in kBytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 kBytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.
```

Por fim pode-se ler nas regras de utilização da ferramenta IOzone [4] que, para o modo automático, o tamanho do buffer da cache deve ser menor ou igual a 128 Megabytes de forma a garantir que são realizados testes dentro e fora do buffer

da cache. Utilizando o comando **sysdef** obtemos o seguinte output:

```
* Tunable Parameters
*
513724416    maximum memory allowed in buffer cache (bufhwm)
 30000    maximum number of processes (v.v_proc)
 99    maximum global priority in sys class (MAXCLSPRI)
 29995    maximum processes per user id (v.v_maxup)
 30    auto update time limit in seconds (NAUTOP)
 25    page stealing low water mark (GPGSLO)
 1    fsflush run rate (FSFLUSHR)
 25    minimum resident memory for avoiding deadlock (MINARMEM)
 25    minimum swapable memory for avoiding deadlock (MINASMEM)
*
* Utsname Tunables
*
```

Por aqui vemos que o tamanho do buffer disponível é 513724416 KB = 501684 MB. No entanto o tamanho de buffer a usar neste cálculo não será diretamente o valor apresentado na imagem anterior, pois sabemos que o tamanho do buffer é truncado se ultrapassar o menor dos seguintes valores: [5] 20% da memória física, 2 TB, ou 1/4 do tamanho máximo da kernel heap. Por esta lógica temos que o tamanho resultante do buffer em questão é 13107,3 MB. Sendo este valor bastante superior a 128 MB então percebemos que os testes realizados com o modo automático não iriam mostrar resultados que caiam fora da cache, por essa razão o método automático não foi utilizado e teve que se escolher o tamanho para os ficheiros de teste. Voltando a olhar para as regras do uso de IOzone [4] vemos que, para comparação de performance entre vários discos se aconselha um tamanho de ficheiro de pelo menos 3*tamanho do buffer da cache disponível. Sendo 3*tamanho do buffer = 39321,6 MB, escolheu-se então um tamanho de ficheiro para teste que fosse maior do que 39321,6 MB. Um tamanho relevante encontrado foi de **40GB**, decidiu-se ainda utilizar um tamanho de **20 GB** de forma a notar a possível aceleração ao testar com um ficheiro muito menor do que os 3*buffer cache aconselhados. Quanto ao record (tamanho de cada leitura/escrita num teste) optou-se pelas opções fornecidas com a flag -a que irá testar para os seguintes tamanhos: **4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 KB**. Não se achou relevante testar com maiores records, pois poucas aplicações escrevem ou leem mais do que 0.5 GB de uma só vez, para além disso, estes testes cobrem uma vasta gama de records permitindo ver o comportamento da máquina com records muito pequenos, médios e grandes. É ainda utilizada a opção **-u** de forma a obter informação relativa à utilização do CPU.

B. Análise dos Resultados com Benchmark Passivo - Sistema ZFS

Mostram-se e analisam-se agora os resultados para os testes descritos no subcapítulo anterior.

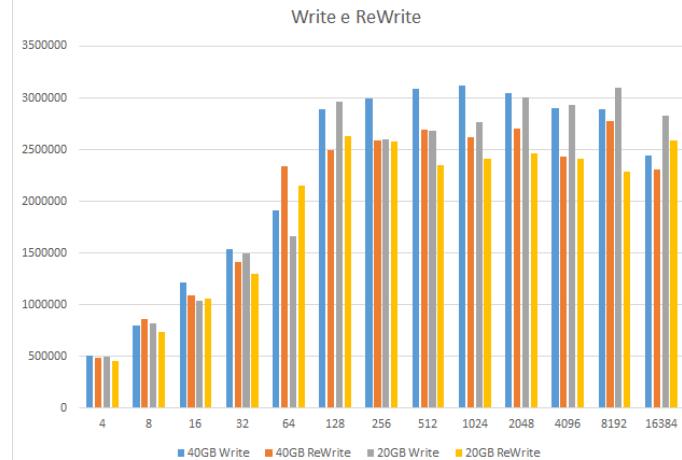
TESTE 1:

Usando os comandos:

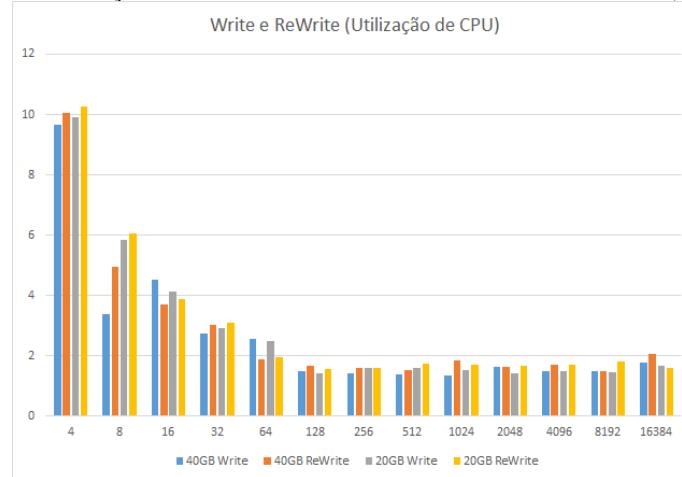
```
/opt/csw/bin/iozone -+u -a -i0 -R -b teste1ZFS40.xls -s 40g
/opt/csw/bin/iozone -+u -a -i0 -R -b teste1ZFS20.xls -s 20g
```

obtiveram-se os seguintes gráficos:

(note-se que neste primeiro gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste segundo gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



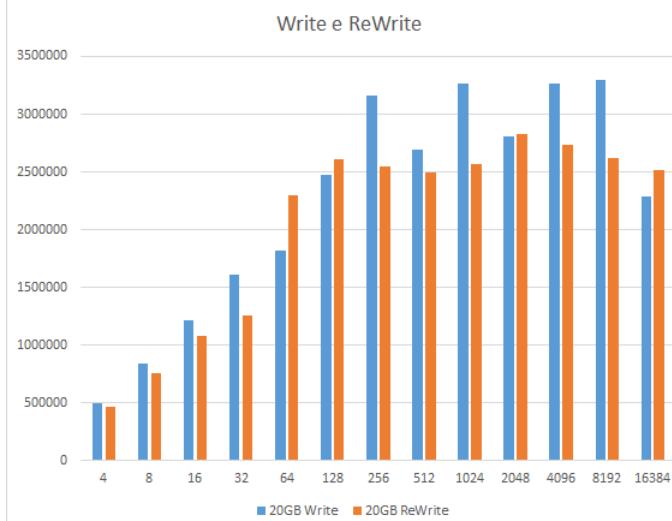
Olhando para estes gráficos que relatam os resultados para os testes de write e rewrite obtidos com -i0, observa-se que, de forma geral, para ambos os tamanhos de ficheiro (40 GB e 20 GB), há um menor número de KB por segundo para testes de rewrite e uma maior percentagem de utilização do CPU, no entanto estes resultados são muito contra intutivos, visto que, aquando do primeiro write temos um overhead chamado 'metadata' que representa o impacto na performance de ter de guardar os dados escritos no ficheiro e de registar onde os dados se localizam no disco. Quando fazemos um rewrite este overhead não ocorre e temos ainda o ficheiro em cache, por esta razão rewrite devia ter um maior número de KB por segundo e menor percentagem de utilização do CPU do que o write. De forma a tentar compreender melhor o que ocorreu correu-se o seguinte programa:

```
/opt/csw/bin/iozone -a -i0 -R -b teste1ZFS.xls -s 40g -N
```

Note-se o uso da flag **-N** que nos permitirá obter os resultados em microsegundos por operação. O seguinte gráfico foi obtido:

(note-se que neste gráfico o eixo y representa

microsegundos/operação e o eixo x o tamanho do record testado)



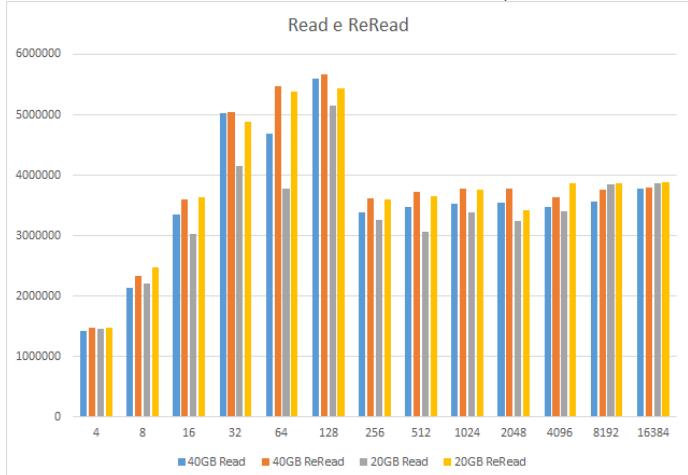
Neste gráfico vemos que, de forma geral, os testes para uma primeira escrita (write) demoram mais tempo por operação, isto realmente seria de esperar. Talvez os resultados contra intuitivos obtidos nos gráficos anteriores possam ser resultado de uma situação esporádica fora do comum, como por exemplo o disco estar mais quente aquando dos testes de rewrite do que nos testes de write o que poderia levar a uma pior performance.

Correndo agora os comandos:

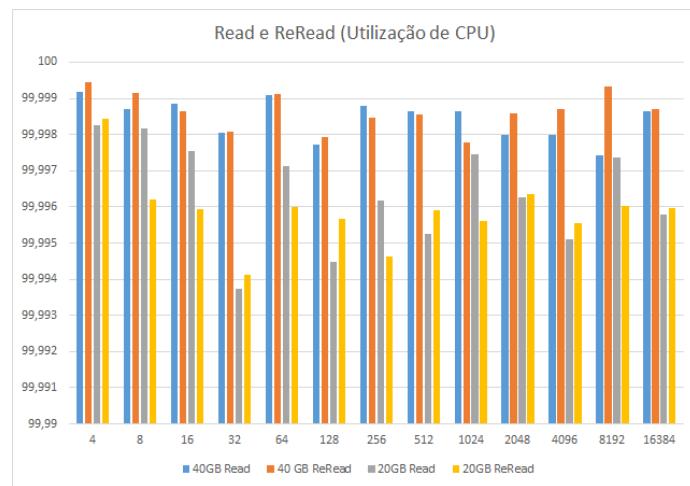
```
/opt/csw/bin/iozone -+u -a -i0 -i1 -R -b teste12ZFS40.xls
-s 40g
/opt/csw/bin/iozone -+u -a -i0 -i1 -R -b teste12ZFS20.xls
-s 20g
```

Foram obtidos os seguintes gráficos:

(note-se que neste primeiro gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste segundo gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



Pelo primeiro gráfico facilmente se observa que a operação de read resulta numa pior performance do que reread já que é realizada com um número inferior de KB/s. Isto seria de esperar pois várias leituras consecutivas (reread) irão tirar um melhor proveito da cache do que uma primeira leitura isolada. Pelo segundo gráfico pouco se conclui pois embora graficamente os valores pareçam oscilar bastante, ao olhar para os valores no eixo y verifica-se que na verdade estão todos extremamente próximos uns dos outros, já que variam só na ordem das milésimas, o que leva a crer que, a nível de utilização do CPU não há muita diferença entre read e reread. Podemos ainda notar que estas operações de read e reread alcançaram um uso de CPU muito superior às operações de write e rewrite, mas tendo em conta que com read e reread também se obteve um maior número de KB/s em praticamente todas as situações então não é de estranhar que tenha havido um maior uso do CPU.

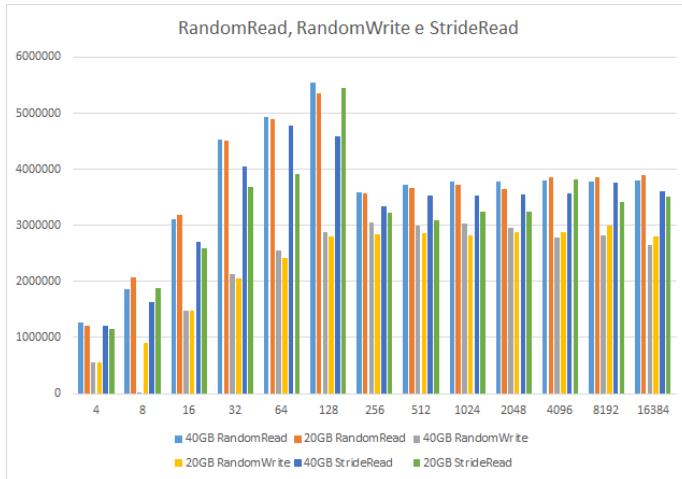
Correndo agora os seguintes comandos (note-se o uso de vários comandos em vez de um só com várias flags para evitar que testes para as flags finais beneficiem demasiado do uso da cache o que poderia retornar resultados um pouco irrealistas não replicáveis numa aplicação genérica):

```
/opt/csw/bin/iozone --+u -a -i0 -i2 -R -b teste131ZFS40.xls
-s 40g
/opt/csw/bin/iozone --+u -a -i0 -i5 -R -b teste133ZFS40.xls
-s 40g
```

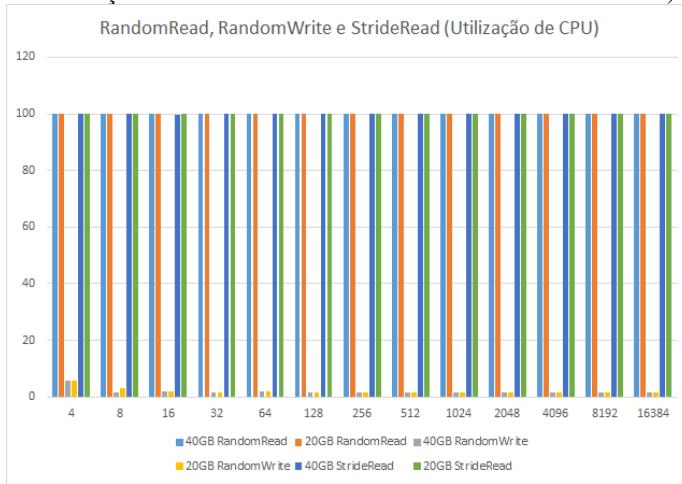
```
/opt/csw/bin/iozone --+u -a -i0 -i2 -R -b teste131ZFS20.xls
-s 20g
/opt/csw/bin/iozone --+u -a -i0 -i5 -R -b teste133ZFS20.xls
-s 20g
```

Foram obtidos os seguintes gráficos:

(note-se que neste primeiro gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste segundo gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



Nota-se que há um maior número de KB por segundo para as operações de RandomRead e de StrideRead do que para a operação de RandomWrite, isto mantém-se coerente com os resultados obtidos nos gráficos anteriores onde a mesma relação se encontrava entre read, reread e write, rewrite. De facto é intuitivo que uma leitura obtenha uma melhor performance que uma escrita. Mais uma vez se nota para as operações de escrita uma muito menor utilização do CPU do que para as operações de escrita mantendo-se coerente com os resultados anteriores.

Num panorama geral, olhando agora para todos os gráficos até agora, Podemos derivar algumas conclusões. Para tamanhos de record muito pequenos nota-se uma melhor performance com o ficheiro de 40GB, mas para tamanhos grandes de record nota-se que o ficheiro de 20GB obtém melhores resultados, provavelmente o facto de ser mais pequeno proporciona um ganho por parte do uso da cache mais acentuado permitindo que este ficheiro escale melhor para maiores records. Nota-se ainda que, no geral, a partir de records de 256KB se nota uma descida na performance para os dois ficheiros.

TESTE 2:

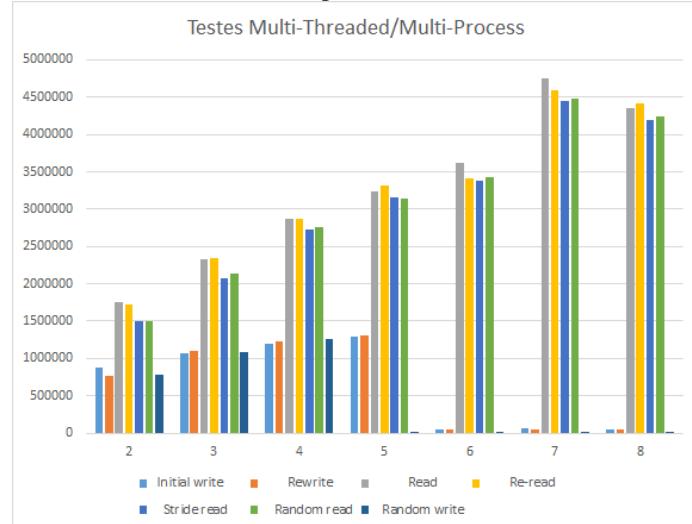
Testa-se agora o comportamento multi-threaded/multi-processo, mas tendo em conta o grande número de testes a realizar para este comportamento apenas se usa um tamanho de record (4KB) e de ficheiro (20GB) para teste para cada thread/processo.

Correndo agora os comandos:

```
/opt/csw/bin/iozone -i0 -R -b teste21ZFS20.xls -s 20g -l2 -u8
/opt/csw/bin/iozone -i0 -i1 -R -b teste22ZFS20.xls -s 20g -l2 -u8
/opt/csw/bin/iozone -i0 -i2 -R -b teste231ZFS20.xls -s 20g -l2 -u8
/opt/csw/bin/iozone -i0 -i5 -R -b teste233ZFS20.xls -s 20g -l2 -u8
```

foi possível obter o seguinte gráfico:

(note-se que neste gráfico o eixo y representa KB/s e o eixo x o número de threads/processos)



Observa-se que, com a exceção de com 2 threads/processos, se obtém uma maior performance ao fazer um rewrite do que um initial write, o que, pelas razões já indicadas anteriormente seria de esperar. No panorama geral vê-se também de novo que se obtém maior performance com reread do que com read.

Nota-se também que com qualquer um dos números de threads/processos testado se obtém uma performance bastante superior ao obtido com os testes single-threaded/single-process. Isto seria de esperar já que se está a usar um número de threads \leq ao número de cores físicos na máquina. Isto apenas não acontece para os testes de Initial Write, ReWrite e Random Write já que vemos uma quebra bastante acentuada na performance destes testes a partir das 5 threads/processos. Vemos que escalam muito menos nesta máquina do que operações de leitura já que estas têm um speed-up bastante acentuado até às 8 threads/processos não inclusive. Há uma descida na performance para as operações

de leitura ao usar 8 threads/processos o que indica que, nesta máquina, o pico de escalabilidade para estas operações de leitura são as 7 threads/processos.

TESTE 3:

Como mencionado no subcapítulo anterior não foi possível realizar este teste nesta partição.

C. Análise dos Resultados com Benchmark Passivo - Sistema UFS

TESTE 1:

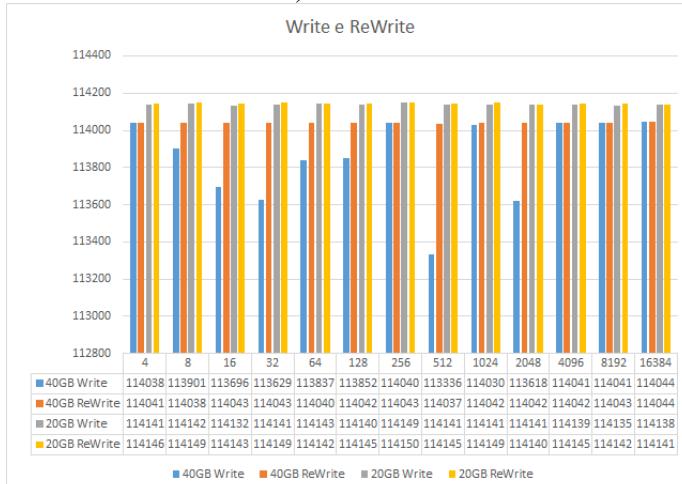
Usando os comandos:

```
/opt/csw/bin/iozone -f /share/jade/iozone.tmp --u -a -i0 -R -b teste1UFS40.xls -s 40g
/opt/csw/bin/iozone -f /share/jade/iozone.tmp --u -a -i0 -i1 -R -b teste12UFS40.xls -s 40g
/opt/csw/bin/iozone -f /share/jade/iozone.tmp --u -a -i0 -i2 -R -b teste131UFS40.xls -s 40g
/opt/csw/bin/iozone -f /share/jade/iozone.tmp --u -a -i0 -i5 -R -b teste133UFS40.xls -s 40g
```

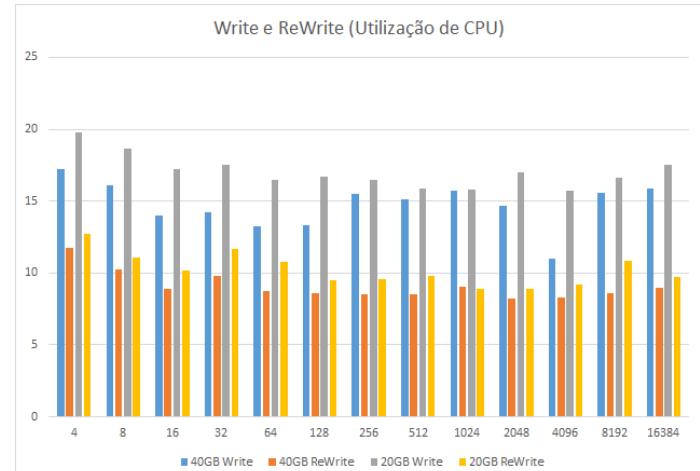
```
/opt/csw/bin/iozone -f /share/jade/iozone.tmp --u -a -i0 -R -b teste1UFS20.xls -s 20g
/opt/csw/bin/iozone -f /share/jade/iozone.tmp --u -a -i0 -i1 -R -b teste12UFS20.xls -s 20g
/opt/csw/bin/iozone -f /share/jade/iozone.tmp --u -a -i0 -i2 -R -b teste131UFS20.xls -s 20g
/opt/csw/bin/iozone -f /share/jade/iozone.tmp --u -a -i0 -i5 -R -b teste133UFS20.xls -s 20g
```

obtiveram-se os seguintes gráficos:

(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



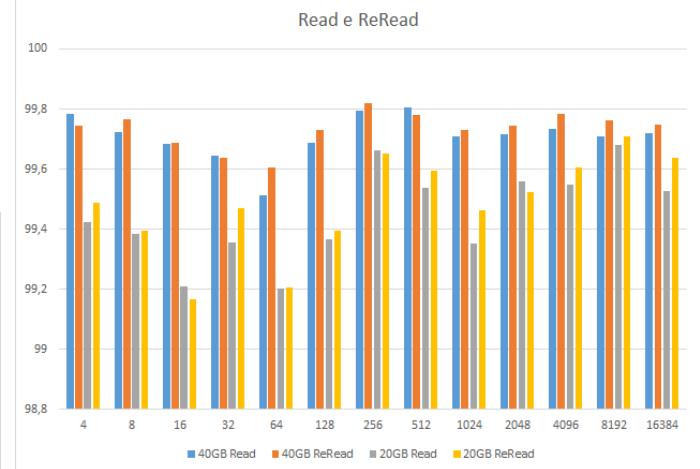
(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



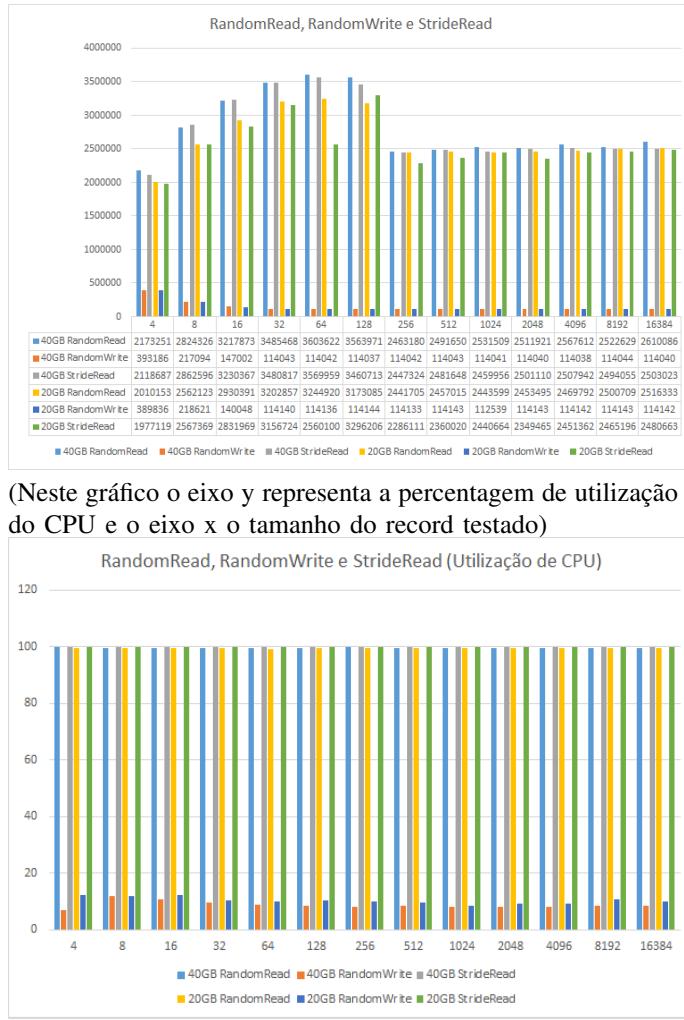
(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)

RandomRead, RandomWrite e StrideRead (Utilização de CPU)

realizar para este comportamento apenas se usa um tamanho de record (4KB) e de ficheiro (20GB) para teste para cada thread/processo.

Correndo agora os comandos:

```
/opt/csw/bin/iozone -i0 -R -b teste21UFS20.xls -s 20g
-l2 -u8 -F /share/jade/iozone1.tmp /share/jade/iozone2.tmp
/share/jade/iozone3.tmp /share/jade/iozone4.tmp
/share/jade/iozone5.tmp /share/jade/iozone6.tmp
/share/jade/iozone7.tmp /share/jade/iozone8.tmp
```

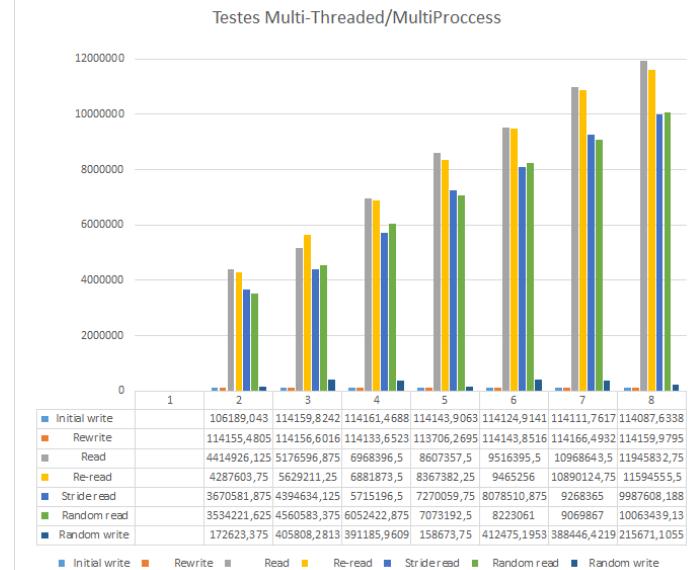
```
/opt/csw/bin/iozone -i0 -i1 -R -b teste22UFS20.xls -s 20g
-l2 -u8 -F /share/jade/iozone1.tmp /share/jade/iozone2.tmp
/share/jade/iozone3.tmp /share/jade/iozone4.tmp
/share/jade/iozone5.tmp /share/jade/iozone6.tmp
/share/jade/iozone7.tmp /share/jade/iozone8.tmp
```

```
/opt/csw/bin/iozone -i0 -i2 -R -b teste231UFS20.xls -s 20g
-l2 -u8 -F /share/jade/iozone1.tmp /share/jade/iozone2.tmp
/share/jade/iozone3.tmp /share/jade/iozone4.tmp
/share/jade/iozone5.tmp /share/jade/iozone6.tmp
/share/jade/iozone7.tmp /share/jade/iozone8.tmp
```

```
/opt/csw/bin/iozone -i0 -i5 -R -b teste233UFS20.xls -s 20g
-l2 -u8 -F /share/jade/iozone1.tmp /share/jade/iozone2.tmp
/share/jade/iozone3.tmp /share/jade/iozone4.tmp
/share/jade/iozone5.tmp /share/jade/iozone6.tmp
/share/jade/iozone7.tmp /share/jade/iozone8.tmp
```

foi possível obter o seguinte gráfico:

(note-se que neste gráfico o eixo y representa KB/s e o eixo x o número de threads/processos)



Neste teste facilmente notamos que este sistema de ficheiros consegue alcançar uma performance muito superior ao sistema ZFS em ambiente multi-threaded/multi-process. Notamos que, embora as operações de escrita não sofram muita variação de speed-up para um diferente número de threads/processos,

Olhando para os gráficos de um panorama geral vemos que as conclusões alcançadas para os gráficos do sistema de ficheiros testado anteriormente (ZFS) se mantêm. Notamos apenas que, no primeiro gráfico (Write e ReWrite), se obtém uma maior performance com ReWrite do que Write que era o esperado, mas que não estava a acontecer nos primeiros testes ao sistema de ficheiros ZFS.

Comparando agora os resultados com os obtidos no sistema de ficheiros anterior nota-se que o sistema anterior permitiu consistentemente a obtenção de uma maior performance já que os valores de KB por segundo eram bastante superiores em todos os gráficos. Da mesma maneira que aconteceu no sistema de ficheiros anterior, a partir de records de tamanho 256KB nota-se uma queda abrupta na performance, sendo que até esse tamanho se vai notando subidas na performance na maior parte dos casos. Isto indica novamente que este tamanho de record poderá ser o limite da escalabilidade para este tipo de testes nesta máquina para ambos estes dois discos e tipos de sistemas de ficheiros. Novamente se nota, como esperado, uma melhor performance por parte das leituras do que por parte das escritas.

TESTE 2:

Testa-se agora o comportamento multi-threaded/multi-process, mas tendo em conta o grande número de testes a

as operações de leitura sofrem um speed-up brusco com o aumento de threads/processos não dando possibilidade de detetar o limite da escalabilidade (Ao contrário do sistema ZFS onde a escalabilidade tinha o seu pico nas 7 threads/processos). Por aqui podemos concluir então que, se o objetivo for realizar operações em ficheiros em ambiente multi-threaded/multi-processos então é vantajoso o sistema de ficheiros UFS, caso contrário, em ambiente single-threaded/single-process entao ZFS será uma escolha mais acertada.

TESTE 3:

Testa-se agora o terceiro teste que não foi possível realizar para o sistema de ficheiros ZFS, como mencionado anteriormente.

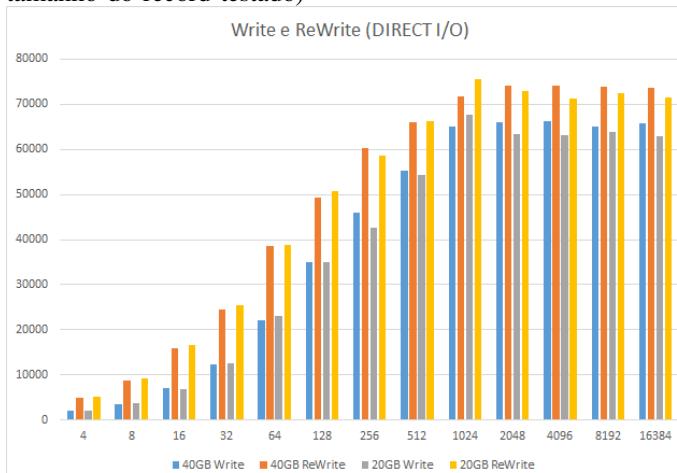
Usando os comandos:

```
/opt/csw/bin/iozone -f /share/jade/iozone.tmp -+u -a -i0 -R -b teste3UFS40.xls -s 40g -I
/opt/csw/bin/iozone -f /share/jade/iozone.tmp -+u -a -i0 -i1 -R -b teste32UFS40.xls -s 40g -I
/opt/csw/bin/iozone -f /share/jade/iozone.tmp -+u -a -i0 -i2 -R -b teste331UFS40.xls -s 40g -I
/opt/csw/bin/iozone -f /share/jade/iozone.tmp -+u -a -i0 -i5 -R -b teste333UFS40.xls -s 40g -I

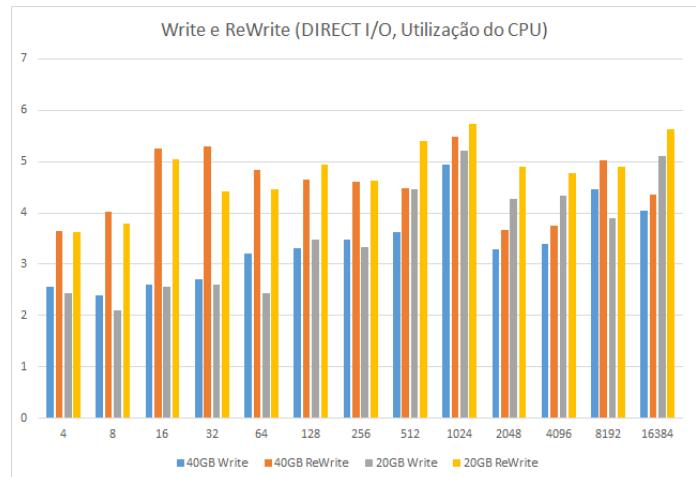
/opt/csw/bin/iozone -f /share/jade/iozone.tmp -+u -a -i0 -R -b teste3UFS20.xls -s 20g -I
/opt/csw/bin/iozone -f /share/jade/iozone.tmp -+u -a -i0 -i1 -R -b teste32UFS20.xls -s 20g -I
/opt/csw/bin/iozone -f /share/jade/iozone.tmp -+u -a -i0 -i2 -R -b teste331UFS20.xls -s 20g -I
/opt/csw/bin/iozone -f /share/jade/iozone.tmp -+u -a -i0 -i5 -R -b teste333UFS20.xls -s 20g -I
```

obtiveram-se os seguintes gráficos:

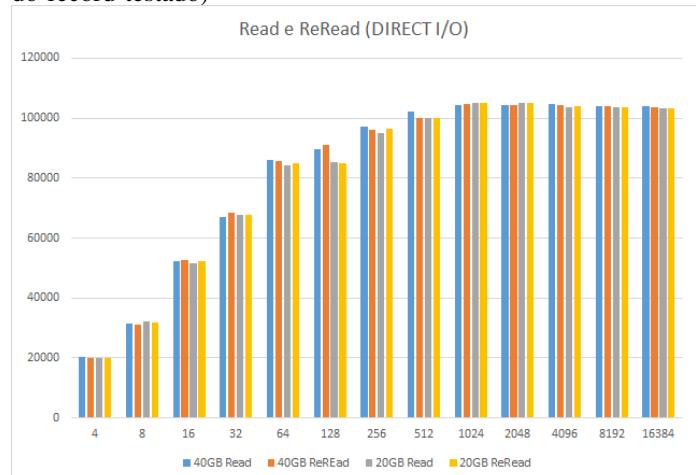
(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



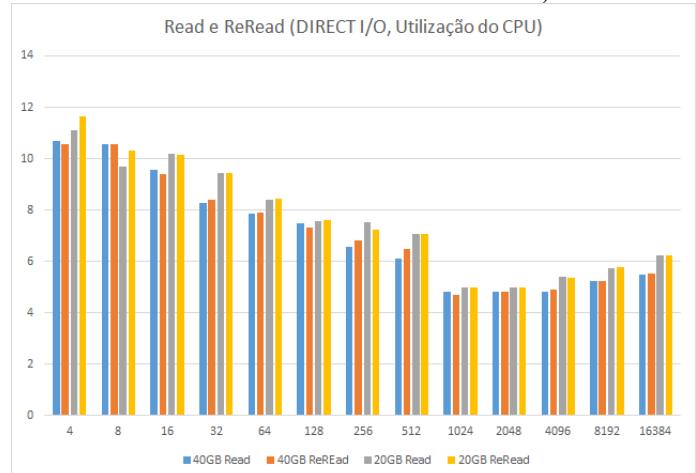
(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



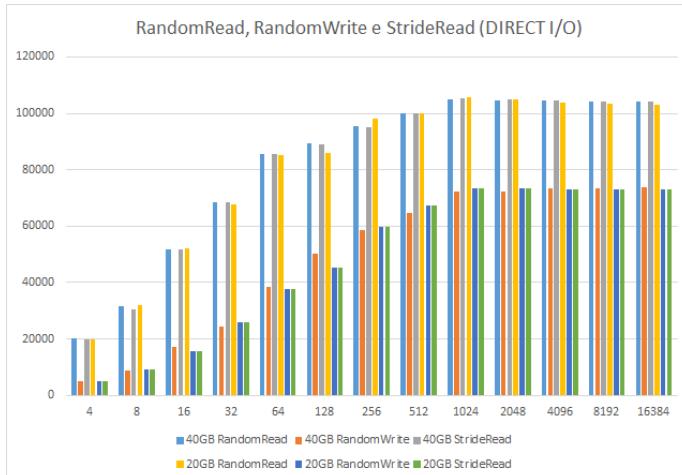
(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



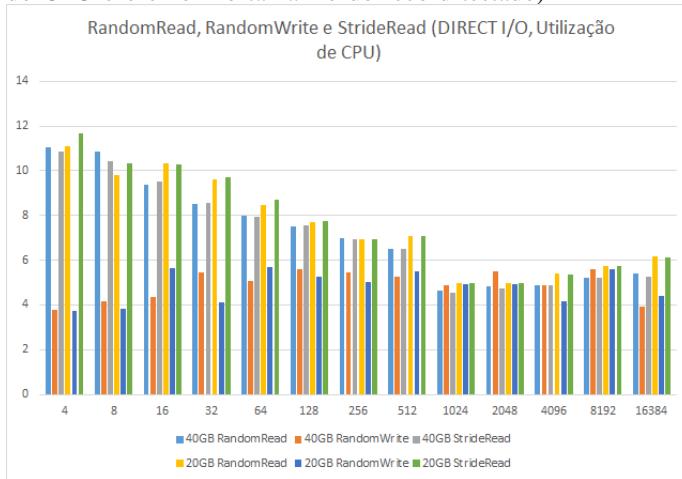
(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



Neste teste nota-se que é obtida uma performance muito inferior à performance obtida neste sistema de ficheiros sem o uso de DIRECT I/O. Isto é um comportamento que já se esperava, pois acessos diretos ao disco em cada operação com ficheiros têm um impacto na performance muito superior a simplesmente aceder, por exemplo, a uma cache externa ao disco. No entanto note-se ainda que, ao contrário de quando não temos acessos diretos ao disco, não se nota o limite de escalabilidade para o tamanho de record. Isto leva a querer que esse limite de escalabilidade estava a acontecer por se estarem a usar tamanhos de record demasiado grandes para se conseguir tirar completo partido dos benefícios da cache da mesma forma que se obtinha para records menores. Como estes testes não utilizam uma cache e vão diretos ao disco então nunca há benefícios desses e por isso estima-se que a performance não sofra um impacto tão acentuado assim de repente como sofria sem acessos diretos ao disco.

D. Análise dos Resultados com Benchmark Passivo - Sistema NFS

TESTE 1:

Usando os comandos:

```
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-R -b teste1NFS40.xls -s 40g
```

```
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i1 -R -b teste12NFS40.xls -s 40g  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i2 -R -b teste131NFS40.xls -s 40g  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i5 -R -b teste133NFS40.xls -s 40g
```

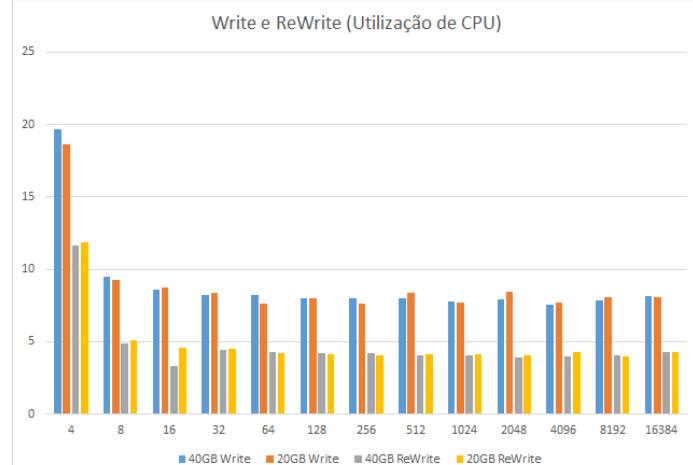
```
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a  
-i0 -R -b teste1NFS20.xls -s 20g  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i1 -R -b teste12NFS20.xls -s 20g  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i2 -R -b teste131NFS20.xls -s 20g  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i5 -R -b teste133NFS20.xls -s 20g
```

obtiveram-se os seguintes gráficos:

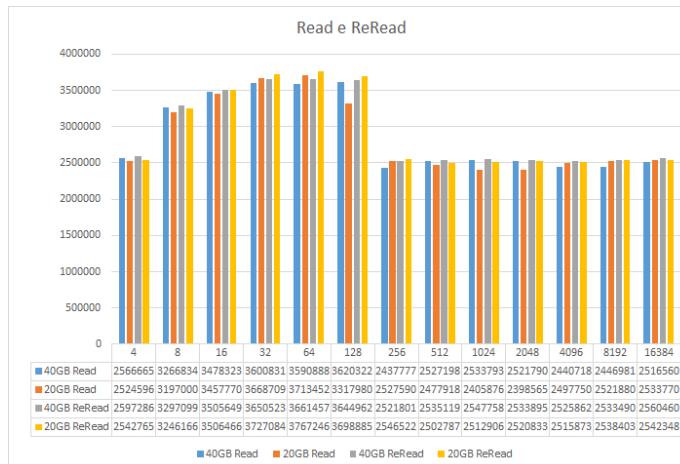
(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



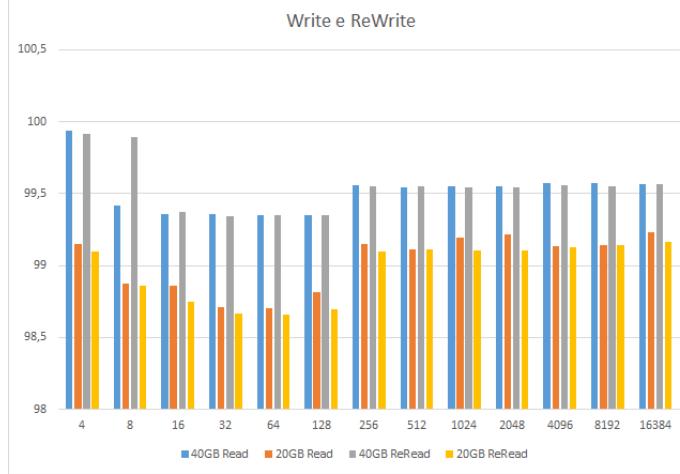
(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)

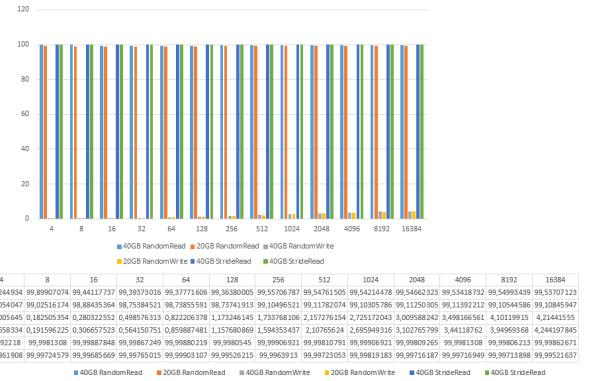


(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)

RandomRead, RandomWrite e StrideRead (Utilização de CPU)



Dos três sistemas de ficheiros testados este parece ser o que obtém a pior performance, em especial para as escritas que é onde a discrepância de resultados entre este e os anteriores se nota mais acentuada. Notam-se que, de um modo geral, os KB/s obtidos são menores do que os obtidos nos testes anteriores. Da mesma forma que com os outros sistemas de ficheiros, nota-se um impacto na performance com o uso de records de tamanho 256KB ou mais, entende-se que isto ocorre pelas razões já mencionadas anteriormente e, o facto de isto acontecer nos três sistemas de ficheiros, não surpreende, pois embora os sistemas de ficheiros sejam diferentes, a máquina é a mesma e, por consequência, a cache que têm ao seu dispor é também a mesma e terá as mesmas limitações.

TESTE 2:

Testa-se agora o comportamento multi-threaded/multi-process, mas tendo em conta o grande número de testes a realizar para este comportamento apenas se usa um tamanho de record (4KB) e de ficheiro (20GB) para teste para cada thread/processo.

Correndo agora os comandos:

```
/opt/csw/bin/iozone -i0 -R -b teste21NFS20.xls -s 20g -l2 -u8 -F /discoHitachi/iozone1.tmp /discoHitachi/iozone2.tmp /discoHitachi/iozone3.tmp /discoHitachi/iozone4.tmp /discoHitachi/iozone5.tmp /discoHitachi/iozone6.tmp /discoHitachi/iozone7.tmp /discoHitachi/iozone8.tmp
```

```
/opt/csw/bin/iozone -i0 -i1 -R -b teste22NFS20.xls -s 20g -l2 -u8 -F /discoHitachi/iozone1.tmp /discoHitachi/iozone2.tmp /discoHitachi/iozone4.tmp /discoHitachi/iozone5.tmp /discoHitachi/iozone6.tmp /discoHitachi/iozone7.tmp /discoHitachi/iozone8.tmp
```

```
/opt/csw/bin/iozone -i0 -i2 -R -b teste231NFS20.xls -s 20g -l2 -u8 -F /discoHitachi/iozone1.tmp /discoHitachi/iozone2.tmp /discoHitachi/iozone4.tmp /discoHitachi/iozone5.tmp /discoHitachi/iozone6.tmp /discoHitachi/iozone7.tmp /discoHitachi/iozone8.tmp
```

```
/opt/csw/bin/iozone -i0 -i5 -R -b teste233NFS20.xls -s 20g -l2 -u8 -F /discoHitachi/iozone1.tmp
```

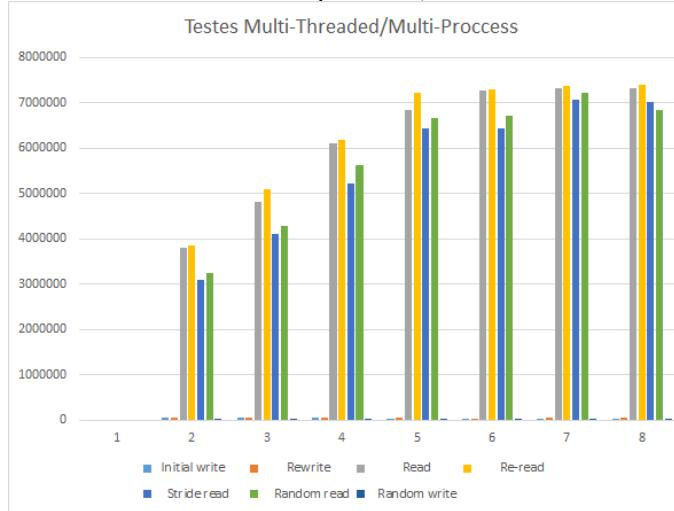
```
/discoHitachi/iozone2.tmp  
/discoHitachi/iozone4.tmp  
/discoHitachi/iozone6.tmp  
/discoHitachi/iozone8.tmp
```

```
/discoHitachi/iozone3.tmp  
/discoHitachi/iozone5.tmp  
/discoHitachi/iozone7.tmp
```

```
-i0 -R -b teste3NFS20.xls -s 20g -I  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i1 -R -b teste32NFS20.xls -s 20g -I  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i2 -R -b teste331NFS20.xls -s 20g -I  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i5 -R -b teste333NFS20.xls -s 20g -I
```

foi possível obter o seguinte gráfico:

(note-se que neste gráfico o eixo y representa KB/s e o eixo x o número de threads/processos)



Observamos agora que, embora os testes single-threaded/single-process se tenham revelado os testes com a pior performance entre os três sistemas de ficheiros, estes testes multi-threaded/multi-process revelam uma melhor performance que o sistema de ficheiros ZFS, mas pior que o sistema de ficheiros UFS. Novamente, assim como para o sistema de ficheiros UFS, esta melhoria na performance é só para as operações de leitura, pois as operações de escrita não demonstram uma performance particularmente significante, praticamente não escalando com o número de threads/processos. Para as operações de leitura, a nível da escalabilidade com o número de threads/processos, vemos que escala melhor que o sistema ZFS, pois não se nota uma descida na performance em nenhum momento, no entanto a escalabilidade parece inferior do que com o sistema de ficheiros UFS, pois vemos que a curva de speed-up neste gráfico parece estagnar por volta das 5 threads/processos levando a crer que o pico de escalabilidade se encontra bastante próximo.

TESTE 3:

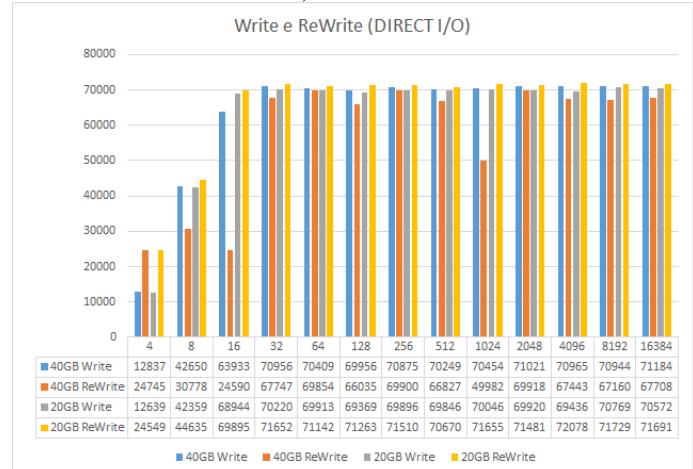
Usando os comandos:

```
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-R -b teste3NFS40.xls -s 40g -I  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i1 -R -b teste32NFS40.xls -s 40g -I  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i2 -R -b teste331NFS40.xls -s 40g -I  
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a -i0  
-i5 -R -b teste333NFS40.xls -s 40g -I
```

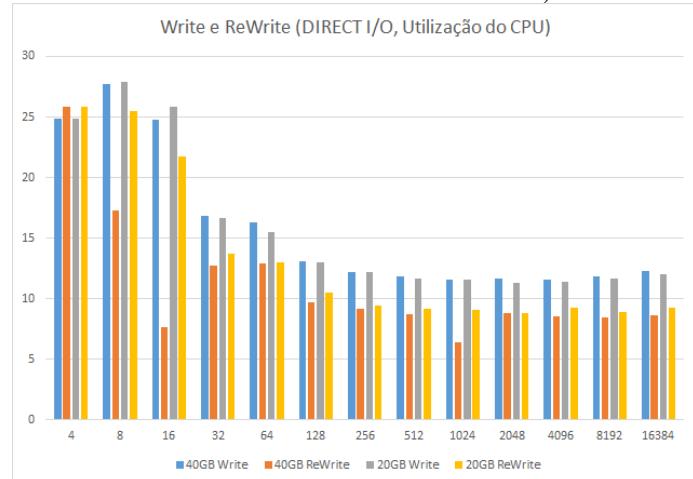
```
/opt/csw/bin/iozone -f /discoHitachi/iozone.tmp -+u -a
```

obtiveram-se os seguintes gráficos:

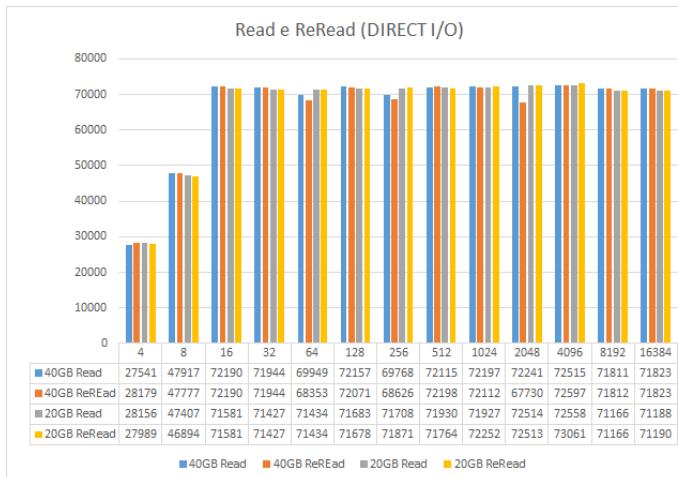
(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



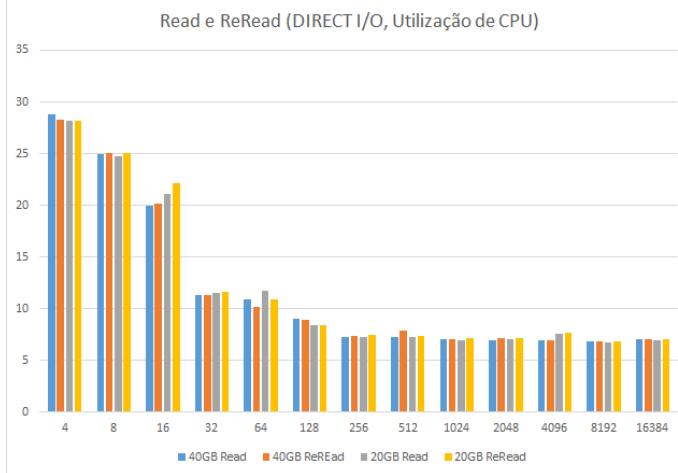
(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



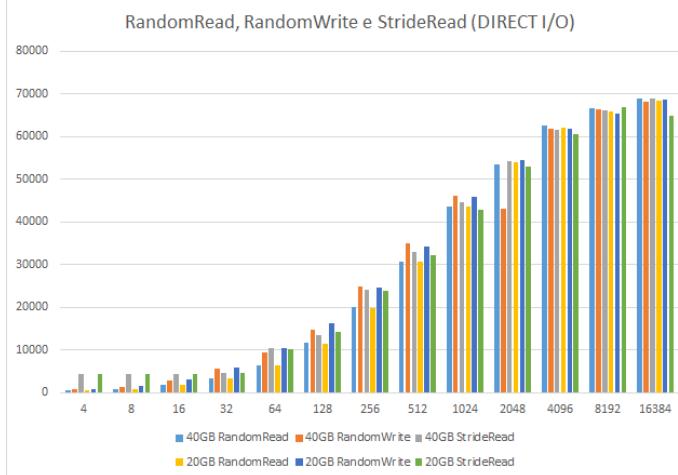
(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



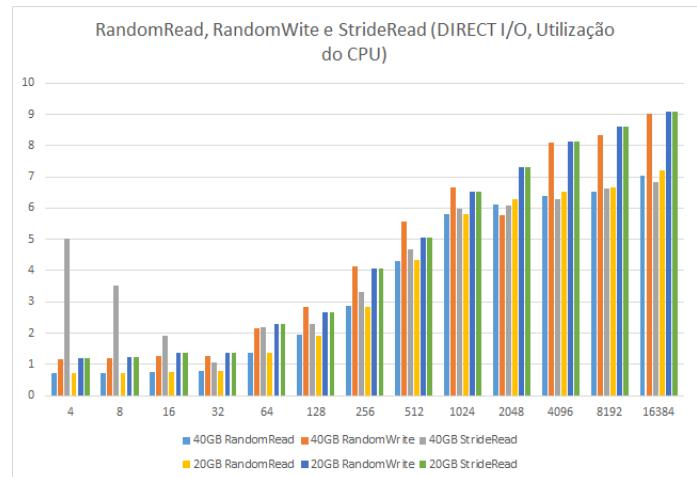
(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa a percentagem de utilização do CPU e o eixo x o tamanho do record testado)



Nestes gráficos vemos uma performance pior do que sem acessos ao disco, como já se esperava pelas razões mencionadas para os mesmos testes feitos para o sistema de ficheiros UFS. A performance das escritas é a que se encontra mais próxima com a performance obtida para estes testes no sistema UFS, mas a das leituras é bastante inferior. Vê-se também que a performance escala com o tamanho dos records, como era de esperar e já se tinha visto no sistema UFS.

E. Análise dos Resultados com Benchmark Passivo - Conclusão

Por fim, depois de um minucioso estudo da ferramenta IOzone nos três sistemas de ficheiros indicados podemos tirar algumas conclusões. Vemos que, em ambiente single-threaded/multi-threaded o sistema ZFS é sem dúvida o que obtém a melhor performance e portanto deve ser preferido quando usado nestas condições na presente máquina. Em ambiente multi-threaded/multi-processos a análise torna-se mais complexa. Para operações de leitura vemos que o sistema UFS é o que obtém a melhor performance e escalabilidade a nível de threads/processos de longe, no entanto, para operações de escrita nota-se que o sistema ZFS é o que apresenta a melhor escalabilidade (mesmo que seja apenas até às 5 threads/processos) e a melhor performance. Nestas condições, caso se queira fazer operações de leitura, deve-se preferir o sistema UFS, mas para operações de escrita é preferível o sistema ZFS.

IV. MEDIÇÕES DE DESEMPENHO - BENCHMARK ATIVO

A. Estudo do Ambiente com Truss

De forma a usar DTrace para replicar as medições realizadas por IOzone utilizou-se a ferramenta **truss**. Esta ferramenta permite saber em tempo real as chamadas de sistema que uma determinada aplicação invoca. Ao executarmos o seguinte comando: **truss -o outputTruss.txt -df /opt/csw/bin/iozone -+u -i0 -i1 -i2 -i5 -s200**, é-nos possível interpretar o funcionamento da ferramenta IOzone de forma a replicá-la em benchmark ativo.

Segue-se um excerto do output de truss com o comando anterior que nos demonstra os testes realizados para -i0:

Por observação vemos que, no ponto circundado 1, para os testes -i0 (write e rewrite) é aberto um ficheiro de nome **iozone.tmp** com permissões para leitura e escrita. De seguida, no ponto circundado 2, vemos já vários rewrites consecutivos a serem feitos no mesmo ficheiro. Observamos que se trata do mesmo ficheiro pois é usado o descriptor devolvido pela chamada de sistema open no ponto circundado 1 (valor 3), vemos que é escrito um record de 4KB em cada write. No ponto circundado 3 vemos o ficheiro a ser fechado depois de realizados todos os testes para i0. Por fim, no ponto circundado 4, vê-se ainda que a chamada de sistema `textbf{getrusage()}` é utilizada para medir os valores de utilização do CPU.

Veja-se agora um excerto do output relativo à leitura do ficheiro para os testes em -i1 (read e reread):

```
40455: 1.2179 getrusage(0xFFFF80FBFFFF800) = 0
40455: 1.2179 open("iozone.tmp", _O_RDONLY) = 3
40455: 1.2180 fdfsync(3, FSYNC) = 0
40455: 1.2181 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2182 lseek(3, 0, SEEK_SET) = 0
40455: 1.2182 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2183 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2184 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2185 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2186 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2187 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2188 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2189 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2190 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2191 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2192 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2193 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2194 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2195 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2196 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2197 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2198 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2199 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2200 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2201 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2202 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2203 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2203 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2204 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2205 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2206 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2207 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2208 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2208 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2209 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2210 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2211 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2212 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2212 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2213 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2214 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2214 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2215 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2216 read(3, "y u y y u y y \0\0\0\0...", 4096) = 4096
40455: 1.2217 fdfsync(3, FSYNC) = 0
40455: 1.2217 close(3) = 0
40455: 1.2218 getrusage(0xFFFF80FBFFFF800) = 0
40455: 1.2218 getrusage(0xFFFF80FBFFFF800) = 0
```

Vemos no ponto circundado 1 o mesmo ficheiro a ser aberto e, no ponto circundado 2, consecutivas leituras. No ponto 3 o ficheiro é novamente fechado e no 4 vemos a chamada de sistema `textbf{getrusage()}` que é utilizada para medir os valores de utilização do CPU.

Observamos agora uma terceira e ultima imagem:

Por aqui percebemos que é usada a chamada ao sistema **lseek()** com o argumento **SEEK_SET** para serem realizados os testes de random write/read. Mais em baixo vemos o uso de **lseek()** com o argumento **SEEK_CUR** para se realizar os testes de stride read. Pode-se provar esta interpretação do objetivo do uso de **lseek** ao olhar para esta tabela tirada da documentação da chamada ao sistema [6]:

Value	Meaning
SEEK_SET	Offset is to be measured in absolute terms.
SEEK_CUR	Offset is to be measured relative to the current location of the pointer.
SEEK_END	Offset is to be measured relative to the end of the file.

Aqui vemos que devemos usar **SEEK_SET** quando a próxima posição a procurar no ficheiro não depende da posição atual já que é relativo a todo o ficheiro, para além disso vemos na imagem anterior que quando **lseek** usa **SEEK_SET** como argumento tem um offset sempre diferente (aleatório) como segundo argumento. Nota-se também que devemos usar **SEEK_CUR** quando a próxima posição depende da posição atual já que é relativo ao local a que se aponta agora, para além disso na imagem anterior, quando **SEEK_CUR** é usado, vemos um offset sempre igual (stride) a ser fornecido como segundo argumento.

B. Desenvolvimento do script DTrace

Podemos agora já ter uma ideia sobre como instrumentar corretamente a ferramenta IOzone em tempo real com DTrace de forma a validar, com benchmark ativo, os valores obtidos com IOzone. O nosso objetivo deverá ser o de capturar, com um script DTrace, os probes relativos à chamada de sistema `open()` para o nome de ficheiro que vemos no output das imagens anteriores e depois as chamadas `close()` correspondentes de forma a imprimir informação sobre um determinado teste que foi encontrada entretanto. Serão também capturados os probes lançados por `write()` e `read()` correspondentes de forma a calcular o total de bytes escritos/lidos e o tempo despendido nestas operações. No final poderemos usar estes valores de bytes e o valor do tempo de forma a saber os bytes escritos/lidos por segundo tanto para first write, para rewrites, para reads, rereads, random read, random write e stride read. Mostra-se agora o script DTrace resultante deste raciocínio:

```

1#!/usr/bin/dtrace -s
2#pragma D option quiet
3
4dtrace:::BEGIN {
5    printf("\n");
6    printf("\tInstrumentador para IOzone\n");
7    printf("\t-----");
8
9    self->ok      = 0;
10   self->teste   = 1;
11   self->kb_read = 0;
12   self->kb_write = 0;
13   self->time_read = 0;
14   self->time_write = 0;
15   self->time_close = 0;
16
17   self->read = 0;
18   self->write = 0;
19
20   total_kb = 0;
21   total_time = 0;
22   self->time_init = 0;
23 }
24
25 syscall::openat*:entry {
26     self->fd = arg1;
27 }
28
29 syscall::openat*:return
30 /self->fd/ {
31     self->ok = (strchr(copyinstr(self->fd, $$1) != NULL) ? 1 : 0;
32 }
33
34 syscall::read*:entry
35 /self->ok/ {
36     self->kb_read = self->kb_read + (arg2/1024);
37
38     self->time_init = timestamp;
39 }
40
41 syscall::read*:return
42 /self->ok && arg1/ {
43     self->time_read = self->time_read + ((timestamp - self->time_init));
44 }
45
46 syscall::write*:entry
47 /self->ok/ {
48     self->kb_write = self->kb_write + (arg2/1024);
49
50     self->time_init = timestamp;
51 }
52
53 syscall::write*:return
54 /self->ok && arg1/ {
55     self->time_write = self->time_write + ((timestamp - self->time_init));
56 }
57
58 syscall::close*:return
59 /!(self->kb_read>0 || self->kb_write>0) && self->ok/ {
60
61     kb      = self->kb_write+self->kb_read;
62     total_kb = total_kb + kb;
63     time   = self->time_read + self->time_write;
64     total_time = total_time + time;
65     kbr    = self->kb_read;
66     kbw   = self->kb_write;
67
68     printf("\n");
69     printf("-----\n");
70     printf(" / Test File Closed \n");
71     printf("-----\n");
72     printf("| $s\n", $$2);
73     printf("%s| %5s| %5s| %15s| %16s| %17s| %20s|\n", "Test", "KB", "KB",
74     "Read", "KB Write", "Time Spent", "Time Read", "Time Write");
75     printf("%5d| %5d| %15d| %15d| %16d| %17d| %20d|\n", self->teste, kb,
76     kbr, kbw, time, self->time_read, self->time_write);
77     printf("\n");
78     printf("-----\n");
79     self->ok      = 0;
80     self->teste   = self->teste +1;
81     self->time_init = 0;
82     self->kb_read = 0;
83     self->kb_write = 0;
84     self->fd      = 0;
85     self->time_read = 0;
86     self->time_write = 0;
87 }
88
89 dtrace:::END {
90
91     printf(" ----- \n");
92     printf(" | DTrace stoped. \n");
93     printf(" | ----- \n");
94     printf(" | Total Time spent: %20d|\n", total_time);
95     printf(" | Total KB: %20d|\n", total_kb);
96     printf(" | ----- \n");
97 }

```

Note-se como, no ponto circundado 1, se usa o ficheiro com

o nome argumento (\$\$1) como o ficheiro a observar (onde se espera que os testes do iozone sejam realizados). O facto de fornecermos este argumento permite-nos usar este script também para instrumentar os testes multi-threaded/multi-process e não só os single-threaded/single-process. O segundo argumento (\$\$2) será uma indicação por parte do utilizador sobre o tipo de teste que está a realizar, por exemplo: "multi-threaded/process Thread/processo: 1". E aparecerá apresentado junto com os valores imprimidos.

Nos pontos circundados 2 e 3 podemos já notar como quando é capturado um probe entry de uma chamada ao sistema read ou write se regista os KB envolvidos na chamada e se começa a contar o tempo que irá demorar. Nos pontos circundados 4 e 5 vemos a serem capturados os probes dos respetivos return das chamadas anteriores e a terminar a contagem do tempo demorado (em nanosegundos). Desta forma conseguimos obter para qualquer tipo de teste os KB envolvidos no teste e o tempo que passou.

No ponto circundado 6 vemos que os valores que foram encontrados são imprimidos no ecrã quando é capturado um probe da chamada ao sistema close. O facto de ser encontrada esta chamada implica que se acabou um teste (um initial write ou um rewrite, etc.), os valores não globais são então reinicializados a zero para agora guardarem informação relativa ao teste seguinte.

Por fim o ponto circundado 7 demonstra a impressão dos resultados globais que foram sendo guardados ao longo de todos os testes. Esta situação vai acontecer quando o script DTrace for parado, por exemplo com o uso de `é`. O script deve ser parado quando a ferramenta IOZone terminar.

```
a72293@solarisEdu:~/a72293$ dtrace -qs d1 a72293iozone.tmp single-threaded
Instrumentador para IOZone
-----
/ Test File Closed
-----| single-threaded
Test| 0| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
-----| 200| 0| 200| 1373166| 0| 1373166|
\

/ Test File Closed
-----| single-threaded
Test| 1| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
-----| 200| 0| 200| 733610| 0| 733610|
\

/ Test File Closed
-----| single-threaded
Test| 2| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
-----| 204| 204| 0| 332241| 332241| 0|
\

/ Test File Closed
-----| single-threaded
Test| 3| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
-----| 204| 204| 0| 313269| 313269| 0|
\

/ Test File Closed
-----| single-threaded
Test| 4| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
-----| 200| 200| 0| 325743| 325743| 0|
\

/ Test File Closed
-----| single-threaded
Test| 5| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
-----| 200| 0| 200| 737491| 0| 737491|
\

/ Test File Closed
-----| single-threaded
Test| 6| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
-----| 200| 200| 0| 322978| 322978| 0|
\

DTrace stoped.

Total Time spent: 4138498
Total KB: 1408
-----|
```

```
e o output da ferramenta IOZone:
a72293@solarisEdu:~/a72293$ /opt/csw/bin/iozone -fa72293iozone.tmp -i0 -i1 -i2 -i5 -s200 -r4
IOzone: Performance Test of File I/O
Version $Revision: 3.434 $
Compiled for 64 bit mode.
Build: Solaris10

Contributors: William Norcott, Don Capps, Isom Crawford, Kirby Collins
Al Slater, Scott Rhine, Mike Wisner, Ken Goss
Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
Vangel Bojaxhi, Ben England, Vikentsi Lapa,
Alexey Skidanov.

Run began: Mon May 8 04:32:59 2017

File size set to 200 kB
Record Size 4 kB
Command line used: /opt/csw/bin/iozone -fa72293iozone.tmp -i0 -i1 -i2 -i5 -s200 -r4
Output is in kBytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 kB.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.

random random
bkwd record stride
          kB reculen write rewrite read reread read write
read   rewrite      read fwrite frewrite fread freread
      200       4 130201 230722 450563 454468 408044 221463
            399304

iozone test complete.
a72293@solarisEdu:~/a72293$
```

C. Testes ao script DTrace

Antes de usar o script desenvolvido e organizar os seus resultados em gráficos, realizaram-se dois testes para garantir que se encontra a funcionar corretamente.

Primeiro quis-se testar em ambiente single-threaded/single-process, para isso correu-se o script com o seguinte comando: **dtrace -qs instrumentador.d a72293iozone.tmp single-threaded**

noutro terminal, correu-se o seguinte comando em paralelo: **/opt/csw/bin/iozone -fa72293iozone.tmp -i0 -i1 -i2 -i5 -s200 -r4**

Observe-se agora o output do script desenvolvido:

Notam-se aqui algumas discrepâncias nos resultados. Nota-se, por exemplo, que o script DTrace obteve resultados maiores

de KB/s. Se olharmos para o output podemos, por exemplo, ver que o script obtém para o teste de Initial write (que é o teste 0 apresentado na imagem do output) os seguintes valores: 1373166 nanosegundos = 0.001373166 segundos, logo KB/s = $200/0.001373166 = 145648.81449$ KB/s. Para o teste de ReWrite (que é o teste 1 apresentado na imagem do output) podemos ver os seguintes valores: 733610 nanosegundos = 0.00073361 segundos, logo KB/s = $200/0.00073361 = 272624.21897$ KB/s. No entanto para a ferramenta IOzone foram obtidos os valores 130201 KB/s e 230722 KB/s para Initial Write e ReWrite respetivamente. O facto de haver esta diferença revela já diferenças entre o cálculo dos valores com benchmark passivo e com benchmark ativo. Uma possível razão poderá ser simplesmente a certeza dos cálculos dos tempos já que o valor de KB envolvidos nos testes são os mesmos com o script e com a ferramenta IOzone, logo a única variável que pode variar significantemente é o tempo. Vemos no início do output de IOzone que este tem uma resolução de 6 casas decimais no cálculo dos segundos, no entanto o nosso script calcula o tempo em nanosegundos por isso tem uma resolução de 9 casas decimais logo para valores temporais muito grandes estima-se que os resultados sejam ainda mais distintos, já que o erro inserido no cálculo dos tempos por parte da ferramenta IOzone se notará ainda mais.

De seguida quis-se testar em ambiente multi-threaded/multi-process, para isso correu-se o script com os seguintes comandos (cada um em paralelo):

(este comando vai observar a thread/processo 1)
dtrace -qs instrumentador.d a72293iozone1.tmp thread/process_1

(este comando vai observar a thread/processo 2)
dtrace -qs instrumentador.d a72293iozone2.tmp thread/process_2

(este comando vai observar a thread/processo 3)
dtrace -qs instrumentador.d a72293iozone3.tmp thread/process_3

noutro terminal, correu-se o seguinte comando em paralelo:
/opt/csw/bin/iozone -i0 -s200 -r4 -l2 -u3 -F a72293iozone1.tmp a72293iozone2.tmp a72293iozone3.tmp

Observe-se agora o output do script desenvolvido:

(output do script relativo à thread/processo 1)

```
a72293@solarisEdu:~/a72293$ dtrace -qs d1 a72293iozone1.tmp thread_1
Instrumentador para IOzone
-----
/ Test File Closed
|-----|
| thread_1
| Test| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
| 0| 200| 0| 200| 85589643| 0| 85589643|
| \-----|
| Test File Closed
|-----|
| thread_1
| Test| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
| 0| 200| 0| 200| 823376| 0| 823376|
| \-----|
| Test File Closed
|-----|
| thread_1
| Test| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
| 0| 200| 0| 200| 1370326| 0| 1370326|
| \-----|
| Test File Closed
|-----|
| thread_1
| Test| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
| 0| 200| 0| 200| 874520| 0| 874520|
| \-----|
^C
| DTrace stopped.
| Total Time spent: 88657865
| Total KB: 800
a72293@solarisEdu:~/a72293$
```

(output do script relativo à thread/processo 2)

```
a72293@solarisEdu:~/a72293$ dtrace -qs d1 a72293iozone2.tmp thread_2
Instrumentador para IOzone
-----
/ Test File Closed
|-----|
| thread_2
| Test| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
| 0| 200| 0| 200| 57197860| 0| 57197860|
| \-----|
| Test File Closed
|-----|
| thread_2
| Test| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
| 0| 52| 0| 52| 257477| 0| 257477|
| \-----|
| Test File Closed
|-----|
| thread_2
| Test| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
| 0| 200| 0| 200| 1465568| 0| 1465568|
| \-----|
| Test File Closed
|-----|
| thread_2
| Test| KB| KB Read| KB Write| Time Spent| Time Read| Time Write|
| 0| 180| 0| 180| 575925| 0| 575925|
| \-----|
^C
| DTrace stopped.
| Total Time spent: 59496830
| Total KB: 632
a72293@solarisEdu:~/a72293$
```

(output do script relativo à thread/processo 3)

```
*Ca72293@solarisEdu:/a72293$ dtrace -qs d1 a72293iozone3.tmp thread_3
Instrumentador para IOzone
-----
/ Test File Closed
-----
| thread_3
| Test KB KB Read KB Write Time Spent Time Read Time Write
| 0 200 0 200 73429381 0 73429381
|
\

/ Test File Closed
-----
| thread_3
| Test KB KB Read KB Write Time Spent Time Read Time Write
| 0 148 0 148 654701 0 654701
|
\

C
| DTrace stopped.
| Total Time spent: 74084082
| Total KB: 348

72293@solarisEdu:/a72293$
```

e um excerto do output da ferramenta IOzone:

```
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 kBytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.
Min process = 2
Max process = 3
Throughput test with 2 processes
Each process writes a 200 kBByte file in 4 kBByte records

Children see throughput for 2 initial writers = 3488.03 kB/sec
Parent sees throughput for 2 initial writers = 1173.14 kB/sec
Min throughput per process = 3488.03 kB/sec
Max throughput per process = 3488.03 kB/sec
Avg throughput per process = 1744.02 kB/sec
Min xfer = 200.00 kB

Children see throughput for 2 rewriters = 333403.55 kB/sec
Parent sees throughput for 2 rewriters = 8159.50 kB/sec
Min throughput per process = 137536.42 kB/sec
Max throughput per process = 195867.12 kB/sec
Avg throughput per process = 166701.77 kB/sec
Min xfer = 48.00 kB

Each process writes a 200 kBByte file in 4 kBByte records

Children see throughput for 3 initial writers = 228248.72 kB/sec
Parent sees throughput for 3 initial writers = 2090.13 kB/sec
Min throughput per process = 0.00 kB/sec
Max throughput per process = 128781.98 kB/sec
Avg throughput per process = 76082.91 kB/sec
Min xfer = 0.00 kB

Children see throughput for 3 rewriters = 617299.25 kB/sec
Parent sees throughput for 3 rewriters = 11372.62 kB/sec
Min throughput per process = 178649.66 kB/sec
Max throughput per process = 251018.80 kB/sec
Avg throughput per process = 205766.42 kB/sec
Min xfer = 144.00 kB
```

Observando o output da ferramenta IOzone vemos que, para 2 processos, a média, por thread/processo, do throughput para o teste de Initial Write foi de 1744.02 KB/s. Observamos agora o output do script de forma a ver qual a média obtida. Como estamos a ver a situação com dois processos apenas olhamos para o output relativo ao processo 1 e ao processo 2. Vemos que o throughput do teste initial write para o processo 1 foi de 2336.73132 KB/s e, para o segundo processo, foi de 3496.63431 KB/s. Assim, temos uma média de 2916.68282 KB/s. Mais uma vez, assim como no teste anterior, pode notar-se que o script obteve resultados maiores. No entanto note-se que o facto de obter resultados maiores acontece simplesmente por estarmos a testar com ficheiros muito pequenos porque, como iremos ver no próximo subcapítulo, para grandes ficheiros, a média foi o script devolver resultados menores.

Ao fim destes dois testes comprava-se que o script retorna

resultados com sentido e apresenta output para todos os testes exatamente como pretendido, assim como output final com a acumulação dos valores intermédios. Na falta de alguma maneira de saber exatamente os valores mais corretos iremos confiar que o script nos providencia os melhores valores, visto que estes foram obtidos com uma instrumentação constante em tempo real o que, por si só, seria capaz em teoria de retornar valores mais fidedignos do que a instrumentação obtida pela ferramenta passiva IOzone. Para além disto temos ainda a questão mencionada anteriormente da precisão no cálculo dos tempos onde, como antes explicado, o script tem mais precisão. Isto pode levar a valores cada vez mais distintos entre o script e a ferramenta IOzone à medida que se aumenta o tamanho do ficheiro e, por conseguinte, se aumenta o tempo demorado nos testes.

Depois de testado o script desenvolvido e de detetadas as principais diferenças entre o benchmark com IOzone e com este script, apresentam-se os resultados para os testes realizados anteriormente para cada sistema de ficheiros, mas agora usando o script para obter resultados com benchmark ativo. Ao realizar estes testes podemos validar se as conclusões alcançadas com benchmark passivo estavam corretas. Dos testes realizados anteriormente apenas não se realizaram os testes em que cada operação acedia diretamente ao disco, pois os outros testes serão suficientes para provar certas ou provar erradas as conclusões alcançadas anteriormente relativamente à eficiência de cada sistema de ficheiros em relação a cada teste específico.

D. Análise dos Resultados com Benchmark Ativo - Sistema ZFS

TESTE 1:

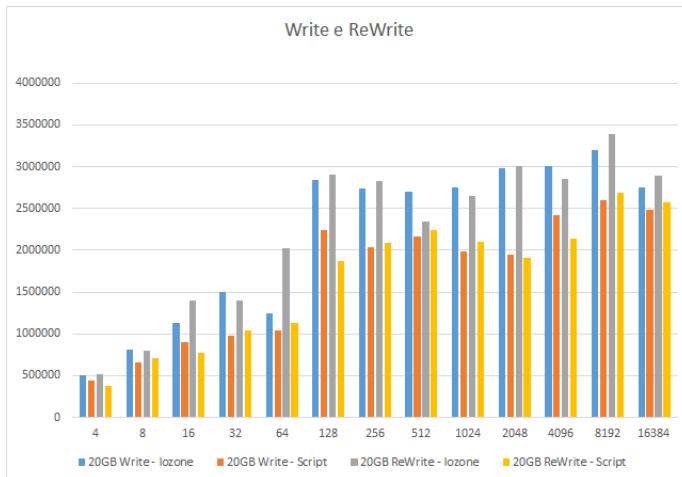
Usando os comandos:

```
/opt/csw/bin/iozone -f a72293iozone.tmp -a -i0 -s 20g
/opt/csw/bin/iozone -f a72293iozone.tmp -a -i0 -i1 -s 20g
/opt/csw/bin/iozone -f a72293iozone.tmp -a -i0 -i2 -s 20g
/opt/csw/bin/iozone -f a72293iozone.tmp -a -i0 -i5 -s 20g
```

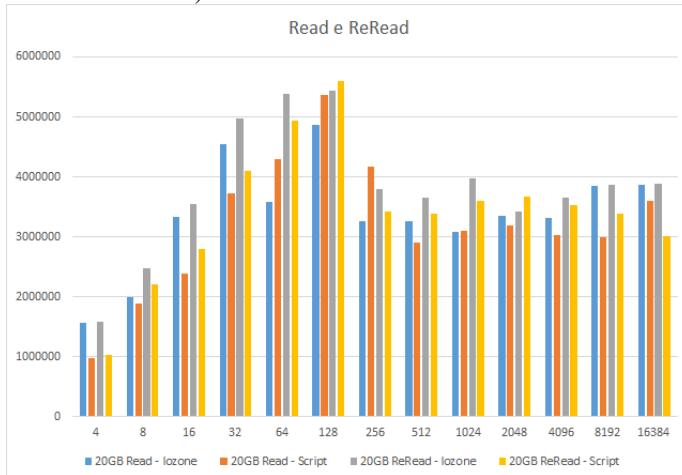
dtrace -qs instrumentador.d a72293iozone.tmp single-threaded

obtiveram-se os seguintes gráficos:

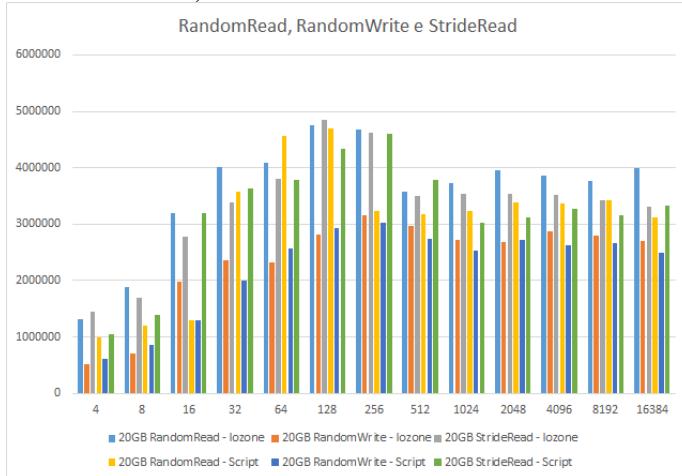
(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



Com estes gráficos vemos que o panorama geral se manteve ao estudar as operações de I/O com o script, a diferença principal acabou por ser o facto de o script demonstrar valores muito menores de KB/s na grande maioria dos casos. Nota-se também que alguns resultados obtidos com o script revelam uma escalabilidade quanto ao tamanho do record usado menor. É normal que o script detecte nuances deste género de forma muito mais realista do que a ferramenta IOZone, pois o script

faz uma instrumentação constante sempre que há um read ou um write enquanto que a ferramenta IOZone apenas reúne no final a quantidade de KB final envolvidos e o tempo total final e deriva os seus resultados a partir daí. Com a instrumentação em tempo real por parte do script então é possível (e realmente foi possível) determinar pequenas perturbações que possam ter acontecido. Por exemplo, como se vê no segundo gráfico, o teste de read tem uma subida brusca para o último record calculado pelo script. Esta subida não foi detetada pela ferramenta IOZone. Desta forma notamos que o script nos permite uma observação mais pormenorizada e realista, enquanto que os resultados da ferramenta IOZone são mais uma média do que se espera ter acontecido. Claro que, o facto do script instrumentar em tempo real poderá ter algum impacto na performance das operações I/O no ficheiro em si, por esta razão é sempre melhor, se queremos comparar máquinas ou sistemas de ficheiros distintos, compara-los com dados instrumentados obtidos da mesma forma, pois como se vê, se comparássemos os resultados obtidos com o script com os obtidos noutro lado com a ferramenta IOZone de forma a ver a melhor máquina seria injusto quanto à máquina onde se usou o script.

TESTE 2:

Testa-se agora o comportamento multi-threaded/multi-processo, mas tendo em conta o grande número de testes a realizar para este comportamento apenas se usa um tamanho de record (4KB).

Correndo agora os comandos:

```
/opt/csw/bin/iozone -i0 -s 20g -l2 -u8 -F
a72293iozone1.tmp a72293iozone2.tmp a72293iozone3.tmp
a72293iozone4.tmp a72293iozone5.tmp a72293iozone6.tmp
a72293iozone7.tmp a72293iozone8.tmp
```

```
/opt/csw/bin/iozone -i0 -i1 -s 20g -l2 -u8 -F
a72293iozone1.tmp a72293iozone2.tmp a72293iozone3.tmp
a72293iozone4.tmp a72293iozone5.tmp a72293iozone6.tmp
a72293iozone7.tmp a72293iozone8.tmp
```

```
/opt/csw/bin/iozone -i0 -i2 -s 20g -l2 -u8 -F
a72293iozone1.tmp a72293iozone2.tmp a72293iozone3.tmp
a72293iozone4.tmp a72293iozone5.tmp a72293iozone6.tmp
a72293iozone7.tmp a72293iozone8.tmp
```

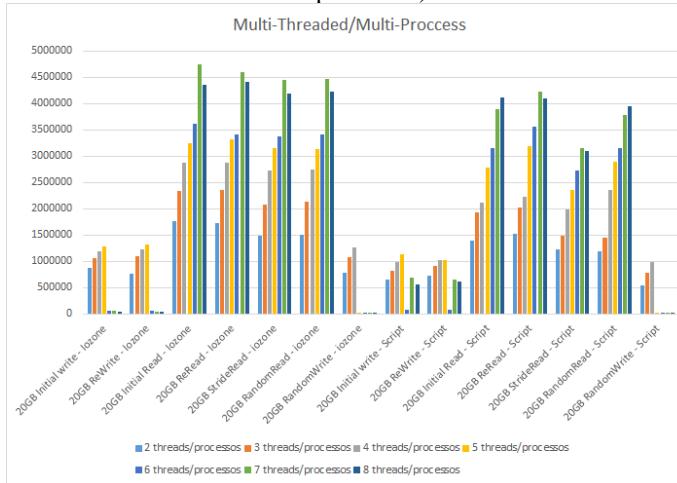
```
/opt/csw/bin/iozone -i0 -i5 -s 20g -l2 -u8 -F
a72293iozone1.tmp a72293iozone2.tmp a72293iozone3.tmp
a72293iozone4.tmp a72293iozone5.tmp a72293iozone6.tmp
a72293iozone7.tmp a72293iozone8.tmp
```

```
dtrace -qs instrumentador.d a72293iozone1.tmp thread_1
dtrace -qs instrumentador.d a72293iozone2.tmp thread_2
dtrace -qs instrumentador.d a72293iozone3.tmp thread_3
dtrace -qs instrumentador.d a72293iozone4.tmp thread_4
dtrace -qs instrumentador.d a72293iozone5.tmp thread_5
dtrace -qs instrumentador.d a72293iozone6.tmp thread_6
```

dtrace -qs instrumentador.d a72293iozone7.tmp thread_7
dtrace -qs instrumentador.d a72293iozone8.tmp thread_8

foi possível obter o seguinte gráfico:

(note-se que neste gráfico o eixo y representa KB/s e o eixo x o número de threads/processos)



Embora os resultados do script sejam muito parecidos com os obtidos pela ferramenta pode de facto notar-se diferenças, principalmente no teste de StrideRead onde os resultados são muito piores para qualquer número de threads do que os obtidos com IOzone. Nota-se também uma escalabilidade bastante diferente para as operações de Initial write e ReWrite. Vemos que com 7 e 8 threads/processos obtém-se resultados ainda acima dos resultados sequenciais, enquanto que a ferramenta IOzone mostra resultados muito piores. Vemos com os resultados do script que com 6 threads/processos temos um grande impacto na performance por alguma razão que não se pode determinar, mas vemos que logo a seguir com 7 e 8 threads/processos esse impacto parece ultrapassado. O facto de a ferramenta IOzone demonstrar este enorme impacto para 6, 7 e 8 threads/processos revela que não foi capaz de detetar o problema como estando somente na situação de 6 threads/processos e induzindo-nos assim em erro levando-nos a crer que os testes em questão não escalavam minimamente acima do valor sequencial para 7 e 8 threads/processos, quando de facto vemos pelos resultados do script que escalam.

Vemos de seguida os testes em abiente single-threaded/single-process para os restantes sistemas de ficheiros. Entendeu-se que não seria muito relevante testar extensivamente os restantes sistemas de ficheiros quanto aos testes em ambiente multi-threaded/multi-process, pois com alguns pequenos testes realizados com tamanhos de ficheiros mais pequenos notou-se que se verificava o mesmo que se verificou neste sistema de ficheiros: resultados menores em média e algumas diferenças pequenas, mas relevantes o suficiente para se notarem minimamente, na escalabilidade ao nível das threads, em especial dos testes que envolvem operações de escrita. Por esta razão não foram realizados gráficos para este segundo teste nos restantes sistemas de ficheiros.

E. Análise dos Resultados com Benchmark Ativo - Sistema UFS

TESTE 1:

Usando os comandos:

```
/opt/csw/bin/iozone -f /share/jade/a72293iozone.tmp -a -i0 -s 20g  

/opt/csw/bin/iozone -f /share/jade/a72293iozone.tmp -a -i0 -i1 -s 20g  

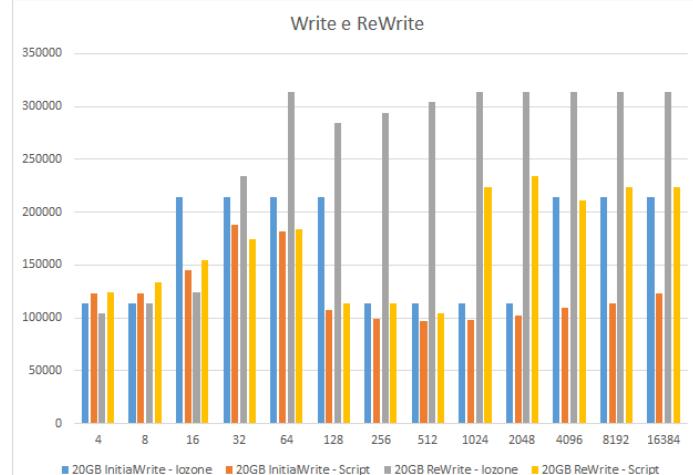
/opt/csw/bin/iozone -f /share/jade/a72293iozone.tmp -a -i0 -i2 -s 20g  

/opt/csw/bin/iozone -f /share/jade/a72293iozone.tmp -a -i0 -i5 -s 20g
```

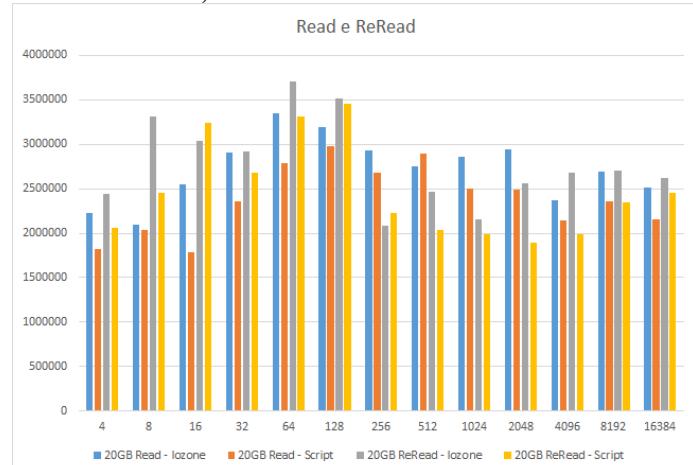
dtrace -qs instrumentador.d a72293iozone.tmp single-threaded

obtiveram-se os seguintes gráficos:

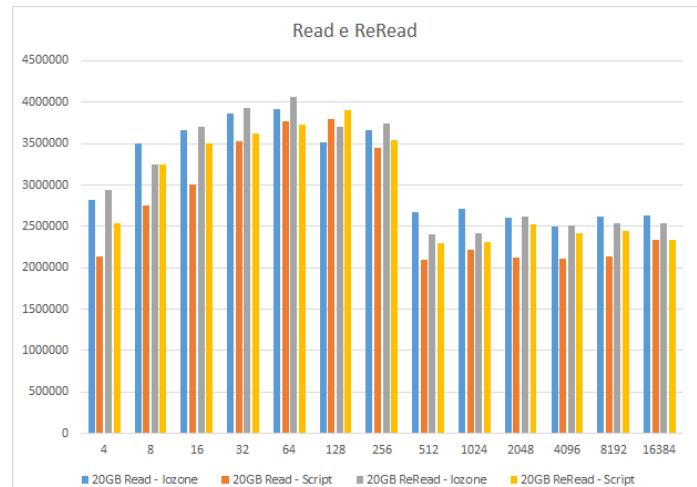
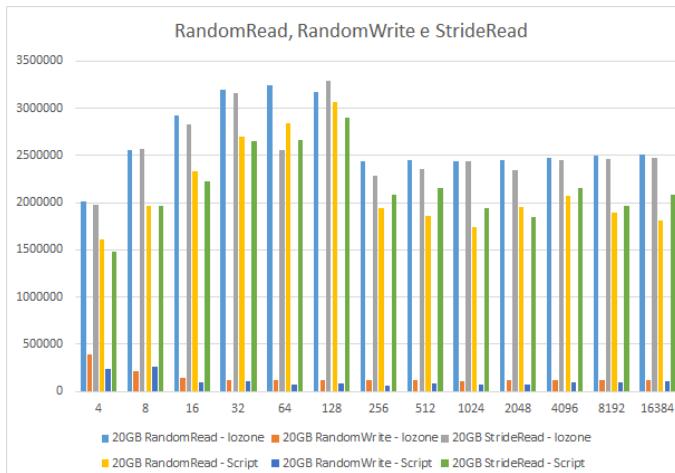
(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



F. Análise dos Resultados com Benchmark Ativo - Sistema NFS

TESTE 1:

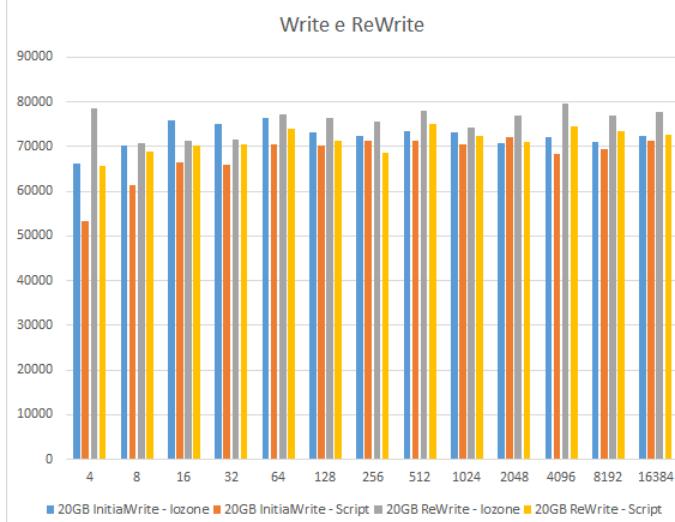
Usando os comandos:

```
/opt/csw/bin/iozone -f /discoHitachi/a72293iozone.tmp -a
-i0 -s 20g
/opt/csw/bin/iozone -f /discoHitachi/a72293iozone.tmp -a
-i0 -i1 -s 20g
/opt/csw/bin/iozone -f /discoHitachi/a72293iozone.tmp -a
-i0 -i2 -s 20g
/opt/csw/bin/iozone -f /discoHitachi/a72293iozone.tmp -a
-i0 -i5 -s 20g
```

dtrace -qs instrumentador.d a72293iozone.tmp single-threaded

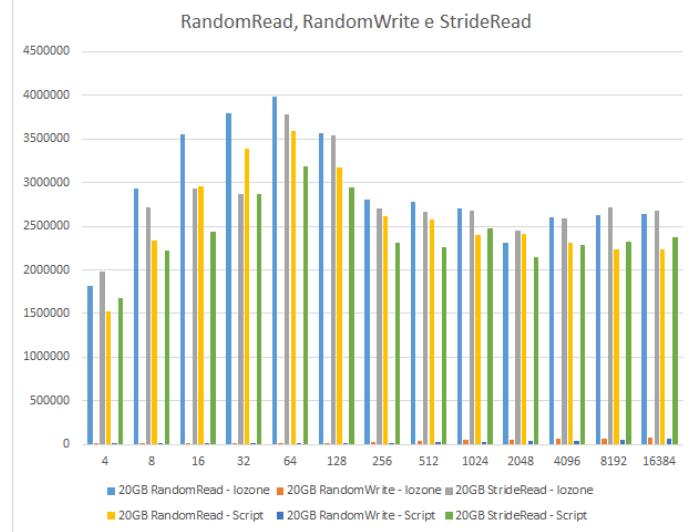
obtiveram-se os seguintes gráficos:

(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)

(Neste gráfico o eixo y representa KB/s e o eixo x o tamanho do record testado)



G. Análise dos Resultados com Benchmark Ativo - Conclusão

Por fim, depois de um minucioso estudo do script e comparando com a ferramenta IOzone nos três sistemas de ficheiros indicados podemos tirar algumas conclusões. Vemos que em média se mantêm as conclusões obtidas anteriormente sendo que se nota a diferença entre a performance de escritas e de leituras de uma forma um pouco mais acentuada, embora tal só tenha sido notado ao calcular o rácio dos resultados o que leva a crer que esta diferença poderá ser mera coincidência, pois é normal que dois testes exatamente iguais efetuados em alturas temporais diferentes retornem resultados que possam ser um pouco diferentes já que as condições nunca são as mesmas, poderá sempre haver mais ou menos processos externos ao nosso controlo a correr em background ou a máquina pode estar mais quente que o normal em determinado momento não permitindo a obtenção de uma melhor performance em determinados casos. em suma, novamente se afirma que o sistema ZFS é sem dúvida o que obtém a melhor performance em ambiente single-threaded/single-process e portanto deve ser preferido quando usado nestas condições na presente máquina. Em ambiente multi-threaded/multi-process, pelos pequenos testes realizados nos restantes sistemas de ficheiros,

vemos que se manteve a tendência do sistema UFS ser o que obtém a melhor performance e escalabilidade a nível de threads/processos para operações de escrita e leitura. Para operações de escrita continuou o sistema ZFS a apresentar a melhor escalabilidade e a melhor performance.

V. CONCLUSÕES E TRABALHO FUTURO

Terminado o estudo proposto verificaram-se diferenças bastante significativas entre benchmark passivo e ativo. Notou-se que, com benchmark passivo não podemos verdadeiramente afirmar sobre o que estamos a medir realmente. Neste estudo em específico realmente as conclusões alcançadas com ambos os tipos de benchmark foram semelhantes, mas noutro tipo de situação isto podia já não acontecer. Isto porque em nenhum lado a ferramenta IOzone nos dá certezas da forma como os seus resultados são calculados, pois não temos como saber se os tempos calculados incluem o tempo de sincronização dos dados no disco com o uso da chamada fsync ou não. No script esse tempo não é contabilizado, mas como os resultados de tempo são em média maiores podemos assumir em princípio que a ferramenta IOzone também não os contabiliza, no entanto, sem ter feito este script e estes testes não poderíamos saber isso. Aqui vemos já um grande problema de fazer benchmark passivo: Achamos que medimos uma coisa, mas na realidade estamos a medir uma completamente distinta. Isto porque, nada impediria alguém de usar a ferramenta IOzone e assumir erradamente que esta contabilizava os tempos com fsync chegando assim a conclusões que poderiam não estar realmente corretas. Outro problema de benchmark passivo já mencionado anteriormente consiste no facto de pequenas perturbações no meio de um teste afetarem o resultado de forma ambígua. Com isto quer-se dizer: Se a meio de um teste que consiste em 100 leituras, uma das leituras, por alguma situação fora do normal, demora muito mais tempo que as restantes então este tempo a mais vai afetar o resultado do teste todo (100 leituras) impossibilitando que se comprehenda se foi uma só leitura que demorou mais do que o normal (por isso devia até ser ignorada) ou então se foram todas as leituras que demoraram muito (o que implica uma baixa performance de parte da máquina). O benchmark ativo permitiria imprimir o resultado de cada leitura individualmente e assim o utilizador seria capaz de distinguir estas duas situações descritas.

Olhando agora para as conclusões alcançadas relativaente á performance entre os 3 tipos de sistemas de ficheiro testados pretende-se apenas acrescentar que os resultados não surpreenderam já que, o sistema que obteve a melhor performance em ambiente single-threaded/single-process (ZFS) encontrava-se monta em princípio acessos à memória mais rápidos do que um disco SAN ligado por rede, por exemplo.

Por fim deixa-se algum trabalho futuro que se acredita que poderia ser relevante. Olhando para o script desenvolvido nota-se que talvez não revele suficientemente benchmark ativo. Seria interessante imprimir os resultados de cada leitura/escrita

individual em tempo real e assim ver se determinadas descidas bruscas na performance se devem a uma só leitura/escrita que demorou demasiado ou se realmente foi uma culpa do todo.

VI. BIBLIOGRAFIA

REFERÊNCIAS

- [1] http://www.iozone.org/docs/IOzone_msword_98.pdf
- [2] <http://www.oracle.com/technetwork/server-storage/solaris11/documentation/dtrace-cheatsheet-1930738.pdf>
- [3] <http://dtrace.org/blogs/brendan/2011/11/09/solaris-11-dtrace-syscall-provider-changes/>
- [4] http://gec.di.uminho.pt/minf/cpd1314/PAC/material1314/IOZONE-Run_rules.pdf
- [5] http://docs.oracle.com/cd/E23823_01/html/817-0404/chapter2-37.html
- [6] <http://codewiki.wikidot.com/c:system-calls:lseek>

VII. ANEXOS

A. Anexo A

Flag	Descrição
-a	modo automático completo. Produz output que cobre todos os testes disponíveis para tamanhos de registo de 4k até 16M e tamanhos de ficheiro de 64k até 512M.
-A	igual a -a, mas permite uma análise mais densa dos testes realizados ao fazer testes até mesmo para situações pouco relevantes, em troca é muito mais demoroso do que -a.
-b filename	É criado um ficheiro binário compatível com Excel com o output dos resultados
-B	Faz com que todos os testes que acedem a ficheiros temporários que os criam e aceadam com recurso à interface mmap(). É útil para saber como a máquina se comportaria com aplicações que tratam um ficheiro como um arrays de memória.
-c	Inclui close() no cálculo dos tempos. Permite observar o comportamento do sistema operativo em relação a close().
-C	Mostrar os bytes transferidos em testes de throughput. Permite observar problemas de starvation no I/O de ficheiros ou no gerenciamento de processos.
-d #	Permite atrasar um dado tempo em milisegundos entre o qual se liberta cada processo/thread durante testes de throughput.
-D	Provoca que todos os dados em espaço mmap seja escrito para o disco de forma assíncrona.
-e	Inclui flush (fsync,fflush) no cálculo dos tempos.
-f filename	Usado para especificar o nome do ficheiro temporário a ser usado durante os testes.
-F filename ...	Usado para especificar o nome dos ficheiros temporários a usar durante os testes de throughput. O número de ficheiros deve ser igual ao número de processos/threads especificado.
-g #	Definir um tamanho máximo em Kbytes para o modo automático.
-G	Provoca que todos os dados em espaço mmap seja escrito para o disco de forma síncrona.
-h	Apresenta um ecrã de ajuda.
-H #	Permite o uso de POSIX async I/O com # operações asíncronas.
-i #	Usado para especificar o tipo de testes a executar (0=write/rewrite, 1=read/re-read, 2=random-read/write, 3=Read-backwards, 4=Re-write-record, 5=stride-read, 6=fwrite/re-fwrite, 7=fread/Re-fread, 8=random mix, 9=pwrite/Re-pwrite, 10=pread/Re-pread, 11=pwritev/Re-pwritev, 12=preadv/Repreadv). A opção 0 terá de ser sempre escolhida e várias podem ser escolhidas em simultâneo para permitir vários testes diferentes de uma só vez.

Flag	Descrição
-I	Usa DIRECT I/O para todas as operações de ficheiros. Diz ao sistema de ficheiros para ignorar o buffer da cache e ir direto ao disco. em Solaris esta opção também irá utilizar directio().
-j #	Especifica o tamanho do espaçamento de leitura (que será # * tamanho do record) para testes de stride read.
-J #	Atrasa # milisegundos antes de cada operação de I/O.
-k #	Usar POSIX async I/O (sem bcopy) com # operações asíncronas.
-K	Gerar alguns acessos aleatórios durante os testes.
-l #	Especificar o número mínimo de processos/threads a usar quando se correrem testes de throughput.
-L #	Especifica o tamanho da linha da cache em bytes. Usado internamente para acelerar os testes.
-m	Provoca o usos de multiplos buffers internamente em vez de um só. Assim simula-se aplicações que leiam para um mesmo buffer continuamente e aplicações que leiam para um array de buffers.
-M	Coloca a string resultante da chamada de uname() no ficheiro de output.
-n #	Define o tamanho de ficheiro mínimo para o modo automático.
-N	Reporta os resultados em microsegundos por operação.
-o	Executa cada write de forma síncrona no disco abrindo os ficheiros sempre com a flag O_SYNC.
-O	Retorna os resultados em operações por segundo.
-p	Limpa a cache do processador entre cada operação de ficheiro permite observar o comportamento dos testes sem a aceleração obtida com o uso da cache em questão.
-q #	Define o tamanho máximo para o tamanho do record em Kbytes para o modo automático.
-Q	Gera ficheiros com dados sobre offset/latência de forma a permitir observar o impacto na latência que determinados offsets provocam.
-r #	Usado para especificar o tamanho do record em Kbytes a usar.
-R	Gera um ficheiro com o output obtido que pode ser lido com Excel e escreve-o no stdout.
-s #	Usado para especificar o tamanho em Kbytes do ficheiro a testar.
-S #	Indica o tamanho da cache do processador em Kbytes. É usado internamente, por exemplo, para fazer alinhamento dos buffers.

Flag	Descrição
-t #	Correr os testes em modo throughput. Permite especificar o número de processos/threads a usar.
-T	Usa POSIX pthreads para os testes de throughput.
-u #	Define o limite superior para o número de processos/threads a utilizar em testes de throughput.
-U mountpoint	Para fazer mount e ummount entre cada teste, desta forma garante-se que a cache não tem nenhuma parte do ficheiro usado nos testes.
-v	Apresenta a versão da ferramenta IOzone.
-V #	Especifica um padrão a ser escrito no ficheiro temporário e validado em cada um dos testes de leitura.
-w	Deixar os ficheiros temporários presentes no sistema de ficheiros depois da sua utilização.
-W	Reservar o acesso aos ficheiros aquando de uma leitura ou de uma escrita.
-x	Não usar stone-walling (técnica usada nos testes de throughput).
-X filename	Usar o ficheiro especificado para escrever triplos com a seguinte informação: Byte offset, size of transfer, compute delay in milliseconds.
-y #	Especificar o tamanho mínimo para o tamanho de record em Kbytes para o modo automático.
-Y filename	Usar o ficheiro especificado para ler triplos com a seguinte informação: Byte offset, size of transfer, compute delay in milliseconds.
-z	Usado em conjunção com -a para testar todos os tamanhos de record possíveis, já que -a ignora os tamanhos demasiado pequenos quando os tamanhos de ficheiro são demasiado grandes.
-Z	Permite o uso de mixing mmap I/O e file I/O.
-+m filename	Usa o ficheiro especificado para obter informação sobre diversos clients para teste em cluster. O ficheiro deve conter uma linha por client e cada linha deve ter três campos separados por espaço. O primeiro campo é o nome do client, o segundo o caminho no cliente para a directória onde IOzone deve trabalhar e o terceiro é o caminho no cliente para o executável do IOzone.
-+n	Usar esta flag para impedir retests.
-+N	Impedir a truncagem ou eliminação sw ficheiros de teste anteriores antes do teste de escrita sequencial.
-+u	Permitir modo de utilização do CPU.
-+d	Permitir modo de diagnóstico.

Flag	Descrição
+p percentage	Definir a percentagem de threads/processos que vão realizar testes de random read. Apenas válido em modo de throughput com mais de 1 processo/thread.
+r	Permitir O_RSYNC e O_SYNC para todos os testes de I/O.
+t	Permitir testes de performance de rede.
+A	Permitir madvise. 0 = normal, 1=random, 2=sequential, 3=dontneed, 4=willneed. Para uso com opções que ativem mmap().

TPC5: Profiling com PERF e DTrace

- Engenharia dos sistemas de Computação -

autor: Daniel Malhadas

Resumo—O presente documento apresenta um estudo aprofundado de um tutorial de PERF [1] desenvolvido por P.J. Drongowski. Este tutorial divide-se em três partes: A primeira foca-se no uso de PERF de forma a identificar e analisar as partes mais intensivas da computação de um determinado programa; A segunda parte introduz o conceito de eventos de performance de hardware e demonstra como os medir ao longo de uma aplicação; A terceira parte usa os eventos estudados na parte 2 para identificar e analisar as partes mais intensivas da computação de um determinado programa, assim como estudado na parte 1. Depois deste estudo, voltam-se a estudar as 3 partes, no entanto usa-se DTrace em vez de PERF. Para este estudo foram usados sempre os mesmos dois programas exemplo: Uma multiplicação de matrizes com um algoritmo ‘naive’; Uma multiplicação de matrizes mais otimizada.

Index Terms—Análise de Desempenho, PERF profiling, DTrace profiling, IOzone, DTrace, Computação Paralela e Distribuída.

I. INTRODUÇÃO

A. Contextualização e Motivação

Com o intuito de solidificar conhecimentos estruturantes de profiling recorre-se a um tutorial idealizado para a melhor compreensão da ferramenta PERF [1]. Este tutorial consiste em três partes:

- **Parte 1** - Foca-se no uso de PERF de forma a identificar e analisar as partes mais intensivas da computação de um determinado programa;
- **Parte 2** - Introduz o conceito de eventos de performance de hardware e demonstra como os medir ao longo de uma aplicação;
- **Parte 3** - Usa os eventos estudados na parte 2 para identificar e analisar as partes mais intensivas da computação de um determinado programa, assim como estudado na parte 1.

Todos os testes realizados ao longo deste documento são relativos a duas aplicações específicas, sendo elas as usadas para o tutorial em questão e por isso sendo as suas implementações obtidas também em [1]. Segue-se uma breve descrição de cada aplicação:

- **Aplicação 1 ‘naive’** - Multiplicação de matrizes ‘naive’. A forma como acede à memória é bastante ineficiente tendo isso um impacto acentuado no desempenho da aplicação, daí a designação de ‘naive’;
- **Aplicação 2 ‘interchanged’** - Introduz uma multiplicação de matrizes mais otimizada que a versão ‘naive’ com diferentes padrões de acesso à memória tornando esta versão mais eficiente.

No final da realização deste tutorial ele é repetido em ambiente Solaris 11 utilizando a ferramenta DTrace aprofundando assim ainda mais o conhecimento da mesma.

B. Em que consiste a ferramenta **PERF**?

PERF ou ‘Performance Events for Linux’ é uma ferramenta de profiling standard para ambiente Linux. Permite analisar a performance de um programa contendo dois subsistemas principais:

- **Subsistema 1** - Um kernel SYSCALL que providencia acesso a dados de desempenho, como eventos do sistema de software e de desempenho de hardware;
- **Subsistema 2** - Uma coleção de ferramentas em ‘user-space’ para coletar, visualizar e analisar dados de desempenho .

Em suma, PERF oferece suporte a eventos de hardware e eventos de software. Os eventos de hardware são medidos usando os contadores de desempenho de hardware. Estes eventos disponíveis são específicos da implementação do processador.

De seguida segue-se o output de **perf –help** que demonstra a sua utilização:

```
usage: perf [-v --version] [--help] COMMAND [ARGS]
The most commonly used perf commands are:
annotate      Read perf.data (created by perf record) and display annotated code
archive       Create archive with object files with build-ids found in perf.data file
bench         General framework for benchmark suites
buildid-cache Manage build-id cache.
buildid-list  List the buildids in a perf.data file
diff          Read perf.data files and display the differential profile
evlist        List the event names in a perf.data file
inject        Filter to augment the events stream with additional information
kmem          Tool to trace/measure kernel memory(slab) properties
kvm           Tool to trace/measure kvm guest os
list          List all symbolic event types
lock          Analyze lock events
mem           Profile memory accesses
record        Run a command and record its profile into perf.data
report        Read perf.data (created by perf record) and display the profile
sched         Tool to trace/measure scheduler properties (latencies)
script        Read perf.data (created by perf record) and display trace output
stat          Run a command and gather performance counter statistics
test          Runs sanity tests
timechart    Tool to visualize total system behavior during a workload
top           System profiling tool.
trace         strace inspired tool
probe         Define new dynamic tracepoints

See 'perf help COMMAND' for more information on a specific command.
```

C. Em que consiste a ferramenta **DTrace**?

A Oracle Solaris DTrace é uma ferramenta de rastreio/traçado avançada usada para testar troços problemáticos de um programa em tempo real. Para isto, esta ferramenta permite observar questões de performance tanto em pequenas aplicações como do próprio sistema operativo em si de forma dinâmica e segura permitindo a quem a utilize a identificação de problemas que seriam difíceis de detetar com outras ferramentas similares por estarem demasiado disfarçados sobre várias camadas de software.

A ferramenta permite também a instrumentação de várias estatísticas em tempo real relativas ao programa a testar. Estatísticas como: consumo de memória, tempo de CPU despendido, que chamadas de função foram realizadas, etc.[2] De forma a poder traçar o que está a acontecer, a ferramenta

DTrace recorre à monitorização de diversos "sinais"/"pontos de interesse" marcados no sistema operativo a que se dá o nome de **probes**. Estes probes são lançados em momentos específicos e, cabe ao utilizador da ferramenta DTrace, saber quais os probes que deve intercetar e o que fazer quando os intercetar de forma a alcançar os resultados que pretende. Segue-se uma tabela com os probes disponíveis em ambiente Solaris 11 e uma breve descrição:[3]

Common DTrace Providers	Description
dtrace	Start, end and error probes
syscall	Entry and return probes for all system calls
fbt	Entry and return probes for all kernel calls
profile	Timer driven probes
proc	Process creation and lifecycle probes
pid	Entry and return probes for all user-level processes
io	Probes for all I/O related events
sdt/usdt	Developer defined probes at arbitrary locations/names within source code for kernel and user-level processes
sched	Probes for scheduling related events
lockstat	Probes for locking behavior within the operating system

```
list of pre-defined events (to be used in -e):
cpu-cycles OR cycles [Hardware event]
instructions [Hardware event]
cache-references [Hardware event]
cache-misses [Hardware event]
branch-instructions OR branches [Hardware event]
branch-misses [Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend [Hardware event]
stalled-cycles-backend OR idle-cycles-backend [Hardware event]

cpu-clock [Software event]
task-clock [Software event]
page-faults OR faults [Software event]
context-switches OR cs [Software event]
cpu-migrations OR migrations [Software event]
minor-faults [Software event]
major-faults [Software event]
alignment-faults [Software event]
emulation-faults [Software event]

L1-dcache-loads [Hardware cache event]
L1-dcache-load-misses [Hardware cache event]
L1-dcache-stores [Hardware cache event]
L1-dcache-store-misses [Hardware cache event]
L1-dcache-prefetches [Hardware cache event]
L1-dcache-prefetch-misses [Hardware cache event]
L1-icache-loads [Hardware cache event]
L1-icache-load-misses [Hardware cache event]
LLC-loads [Hardware cache event]
LLC-load-misses [Hardware cache event]
LLC-stores [Hardware cache event]
LLC-store-misses [Hardware cache event]
LLC-prefetches [Hardware cache event]
LLC-prefetch-misses [Hardware cache event]
dTLB-loads [Hardware cache event]
dTLB-load-misses [Hardware cache event]
dTLB-stores [Hardware cache event]
dTLB-store-misses [Hardware cache event]
...
```

II. CONSIDERAÇÕES INICIAIS

A. Caracterização das Máquinas Utilizadas

De forma a realizar o tutorial de perf escolheu-se a seguinte máquina do cluster **Search6**, tendo esta a ferramenta disponível:

Designação	Nodo 431
Fabricante	Intel
Modelo do CPU	Dual CPU X5650
Microarquitetura do CPU	Nehalem
Frequência do Clock	2.67 GHz
#Cores	12, com 24 threads (2 por core)
Cache	L1d: 32kB, L1i: 32kB, L2: 256kB por core, L3: 20480kB partilhada
Largura de Banda de acesso à memória	32GB/s
Memória RAM	48GB
Rede para Comunicação	Gigabit Ethernet e Myri- net 10Gbps

Esta informação foi obtida a partir de várias fontes distintas. Sendo elas: - o comando Unix **lscpu** e os sites [4] [5] [6].

Segue-se um excerto do output do comando **perf list** que permite ver a lista de eventos disponíveis na máquina e demonstram assim a disponibilidade da ferramenta na máquina em questão:

Nota-se que há tanto eventos de hardware como de software, assim como dito anteriormente e que estes se encontram disponíveis nesta máquina, como demonstrado na imagem.

Para os testes com DTrace utiliza-se uma diferente máquina em ambiente Solaris 11 descrita na seguinte tabela:

Sistema Operativo	Solaris11
Fabricante	Intel
Modelo do CPU	Dual Intel(R) Xeon(R) X5650
Microarquitetura do CPU	Westmere
Frequência do Clock	2.67 GHz
#Cores	8, com 16 threads (2 por core)
Cache	L1: 32kB por core, L2: 256kB por core, L3: 20480kB partilhada
Largura de Banda de acesso à memória	59.7GB/s
Memória RAM	64GB
FileSystems	zfs, ufs, nfs, smb, autofs, smbfss

B. Caracterização dos Conjuntos de Dados

Antes de começar o tutorial é importante decidir os tamanhos dos dados a utilizar nos testes a realizar. Já que o tutorial não foca neste assunto temos liberdade para tomar as nossas próprias decisões. Observamos então o caso de estudo, de forma a compreendermos a natureza dos conjuntos de dados a usar.

Visto que as duas aplicações a usar consistem numa multiplicação de matrizes então sabemos que 3 diferentes matrizes

irão ser criadas: Duas para serem multiplicadas A e B, e uma outra para guardar o resultado C. Serão usadas matrizes quadradas sendo que todas irão ter exatamente o mesmo tamanho. Iremos agora escolher quatro conjuntos de dados relevantes para este estudo, garantindo assim que os testes têm real importância e não são só casos aleatórios e/ou demasiado específicos.

- Conjunto de Dados 1 - Um conjunto de dados que encha por completo a cache de nível 1** O nodo 431 do cluster Search descrito na subsecção anterior tem L1d:32kB e L1i:32kB, apenas nos interessa os dados e por isso temos um total de 32kB na cache de nível 1 para as nossas matrizes. Os valores das matrizes são declarados como números de vírgula flutuante *single-precision* (*float*), logo são necessários 4 bytes para cada um. desta forma, neste nível da cache, podemos guardar $32000B/4B = 8000$ floats. Se queremos que as 3 matrizes caibam neste nível da cache então temos de ter um máximo de $8000/3=2666.(6)$ elementos por matriz, logo $\text{sqrt}(2666.(6))=51.64$ é o máximo número de elementos que podemos ter por dimensão da matriz. Como não podemos ter 0.64 float, temos que arredondar para baixo para 51, no entanto ainda não terminamos por aqui pois ainda podemos ter em conta o tamanho de uma linha da cache. Neste caso é de 64B, logo podemos ter 16 floats numa só linha de cache, por esta razão seria indicado que o total de floats por matriz fosse um múltiplo deste número 16 de forma a tomar total partido da espacialidade local da cache. $(51*51)/16=162.56$, logo não é múltiplo, devemos então encontrar o número mais próximo de 51 (e menor que o mesmo) que cumpra essa condição, esse número é 48 já que $(48*48)/16=144$. Em suma: As máximas dimensões por matriz são 51*51, no entanto as dimensões ideais são 48*48.
- Conjunto de Dados 2 - Um conjunto de dados que encha por completo a cache de nível 2** este conjunto de dados é calculado de forma análoga ao primeiro. Sendo assim obtidos os resultados: As máximas dimensões por matriz são 146*146, no entanto as dimensões ideais são 144*144.
- Conjunto de Dados 3 - Um conjunto de dados que encha por completo a cache de nível 3** este conjunto de dados é calculado de forma análoga aos anteriores. Sendo assim obtidos os resultados: As máximas dimensões por matriz são 1011*1011, no entanto as dimensões ideais são 1008*1008. O tamanho máximo aconselhado no programa *naive.c* é de 1000, mas o programa realiza a seguinte expressão para contar o número de instruções ARM no interior do ciclo mais interior da multiplicação: $9*N*N*N$, sendo N o tamanho do lado de cada matriz e, tendo em conta que o resultado desta expressão deve ser guardado numa variável inteira, então, o maior valor possível para N que não cause um overflow é o número 620. Como 620*620 é múltiplo de 16 620*620 foi usado para este conjunto de dados.

III. TUTORIAL - PERF

A. Antes de Começar

Antes de começar importa referir que o foco deste tutorial será essencialmente no evento *cpu-clock* que permite medir a passagem de tempo de CPU. Este evento torna-se assim relevante por o tempo de execução ser um bom ponto de início para uma análise de desempenho, para além disso é um evento fácil de usar e perceber, tornando-se assim mais apropriado para o uso num tutorial inicial em vez de eventos de hardware mais complicados.

O primeiro passo que devemos tomar numa análise de desempenho de um programa é estabelecer a linha de base do desempenho para cada uma das duas aplicações a usar neste tutorial. Esta linha de base permite-nos avaliar a eficiência e comparar com qualquer alteração que se faça ao código fonte ou ao processo de compilação.

De forma a estabelecer esta linha de base usa-se então o seguinte comando:

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

Sendo este programa *naive* a primeira aplicação, de implementação mais simples, descrita na Introdução. Segue-se o output do comando referido para cada conjunto de dados:

Para Conjunto de dados 1 (48*48):

```
[a72293@compute-431-10 perf]$ perf stat -e cpu-clock,faults ./naive
Performance counter stats for './naive':
          0.764263      cpu-clock (msec)
                         118      faults
  0.002033361 seconds time elapsed
```

Para Conjunto de dados 2 (144*144):

```
[a72293@compute-431-10 perf]$ perf stat -e cpu-clock,faults ./naive
Performance counter stats for './naive':
          6.179444      cpu-clock (msec)
                         172      faults
  0.007414057 seconds time elapsed
```

Para Conjunto de dados 3 (620*620):

```
[a72293@compute-431-10 perf]$ perf stat -e cpu-clock,faults ./naive
Performance counter stats for './naive':
        426.076220      cpu-clock (msec)
                         726      faults
  0.427738443 seconds time elapsed
```

No exemplo destes outputs vemos que é possível o uso de mais do que um evento em simultâneo, neste caso vemos os eventos de software: *cpu-clock* e *page-faults*.

Tendo agora estabelecido a linha de base do desempenho o tutorial pretende responder às seguintes questões:

- Questão 1** - Que partes do programa demoram mais tempo de execução?
- Questão 2** - Será que o grande número de eventos de software ou hardware indicam um impacto significativo no desempenho que deva ser corrigido?

- **Questão 3** - Como podemos corrigir este impacto no desempenho?

B. Parte 1 - Que partes do programa demoram mais tempo de execução?

De forma a responder a esta pergunta estabelecemos primeiro uma pequena base teórica. Estas regiões de código num programa que demoram mais tempo de execução são chamadas *hotspots*. Estes locais são então os mais indicados para otimizar, pois identificam-se como os *bottlenecks* do tempo de execução. Já que estamos a lidar com o tempo de execução então o evento de software *cpu – clock* torna-se relevante para este estudo. De forma a encontrar estes *hotspots* vamos realizar dois passos:

Passo 1 - Identificação de 'hot spots' de execução

Aqui vamos utilizar o comando **perf record -e cpu-clock,faults ./naive**

Segue-se o output deste comando para cada conjunto de dados:

Para Conjunto de dados 1 (48*48):

```
[a72293@compute-431-10 perf]$ perf record -e cpu-clock,faults ./naive
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.010 MB perf.data (~417 samples) ]
[a72293@compute-431-10 perf]$
```

Para Conjunto de dados 2 (144*144):

```
[a72293@compute-431-10 perf]$ perf record -e cpu-clock,faults ./naive
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.010 MB perf.data (~449 samples) ]
[a72293@compute-431-10 perf]$
```

Para Conjunto de dados 3 (620*620):

```
[a72293@compute-431-10 perf]$ perf record -e cpu-clock,faults ./naive
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.090 MB perf.data (~3945 samples) ]
[a72293@compute-431-10 perf]$
```

Passo 2 - Apresentar dados de profiling

Agora iremos usar o comando **perf report** de modo a apresentar no ecrã a leitura dos resultados obtidos com o comando do passo anterior. Como foram usados dois eventos no comando anterior (*cpu – clock* e *page – faults*) será possível observar os resultados para estes dois eventos.

O tutorial apresenta várias possibilidades para apresentar esta informação tais como abrir um '**Terminal User Interface**' **TUI** ou imprimir a informação através da '**standard I/O interface**' **stdio**. Optou-se pela a opção de **stdio**, escreve-se então o comando:

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

A flag **-- header** imprime um parágrafo inicial com informação relevante como a versão do kernel, a arquitetura da máquina, etc. A flag **-- stdio** indica a intenção de obter o output no **stdio** e a flag **-- sortcomm, dso** indica que queremos agrregar os resultados por objeto partilhado **dso**, que neste caso será o ficheiro executável.

Vemos agora o **header** imprimido para o conjunto de dados 1:

```
# captured on: Mon Jun 5 20:41:47 2017
# hostname : compute-431-10.local
# release : 2.6.32-279.141.el6.x86_64
# kernel version: 3.16.0-1.el6.elrepo.x86_64
# arch : x86_64
# ncpus online : 24
# ncpus total : 24
# processor : Intel(R) Xeon(R) CPU E5649 @ 2.53GHz
# cpuid : GenuineIntel 6,44,2
# total memory : 45414804 KB
# cmdline : perf record -e cpu-clock,faults ./naive
# event : name = cpu-clock, type = 1, config = 0x0, config1 = 0x0, excl_usr = 0, excl_kern = 0, excl_host = 0
# event : name = faults, type = 1, config = 0x2, config1 = 0x0, config2 = 0x0, excl_usr = 0, excl_kern = 0, excl_host = 0, ex
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# CPU mappings: cpu = 4, tracepoint = 2, software = 1
```

Observa-se informação pertinente como:

- **versão do kernel** - 2.6.32-279.141.el6.x86_64
- **versão do perf** - 3.16.3-1.el6.elrepo.x86_64
- **arquitetura da máquina** - x86_64
- **#cores disponíveis** - 24

Vemos agora o excerto do output relativo ao evento *cpu – clock* para cada conjunto de dados:

Para Conjunto de dados 1 (48*48):

```
# Samples: 2 of event 'cpu-clock'
# Event count (approx.): 2
#
# Overhead Command Shared Object
# ..... .....
#
50.00%    naive ld-2.12.so
50.00%    naive naive
```

Para Conjunto de dados 2 (144*144):

```
# Samples: 2 of event 'cpu-clock'
# Event count (approx.): 2
#
# Overhead Command Shared Object
# ..... .....
#
50.00%    naive ld-2.12.so
50.00%    naive naive
```

Para Conjunto de dados 3 (620*620):

```
# Samples: 1K of event 'cpu-clock'
# Event count (approx.): 1786
#
# Overhead Command Shared Object
# ..... .....
#
97.87%    naive naive
1.57%    naive libc-2.12.so
0.56%    naive [kernel.kallsyms]
```

Para o maior conjunto de dados (o terceiro) nota-se que 97.87% do tempo de execução é atribuído ao programa *naive* a que fizemos profiling. Este programa invoca *libc – 2.12.so* que, para o terceiro conjunto de dados se nota ser responsável por 1.57% do tempo de execução total.

Para os restantes conjuntos de dados nota-se o padrão de o programa *naive* e de chamadas a *ld – 2.12.so* serem os pontos mais custosos a nível de tempo de execução (*hotspots*) do sistema, no entanto, como podemos ver nos outputs anteriores, para o conjunto 1 e 2 o maior *hotspot* do sistema são as chamadas a *ld – 2.12.so* em vez do programa *naive* em si. Isto acontece por o conjunto de dados ser demasiado pequeno o que evita que a multiplicação de matrizes seja o maior *bottleneck* do tempo tempo de execução. No conjunto 3 vimos já que tal não acontece e que a multiplicação de matrizes aí é realmente o maior *bottleneck* do tempo de execução.

Tendo identificado o programa *naive* e as chamadas a *libc – 2.12.so* ou *ld – 2.12.so* como os maiores *bottlenecks* do tempo de execução, podemos agora identificar quais as funções da aplicação *naive* que causam o maior *bottleneck*. Para isto temos o comando:

perf report --stdio --dso=naive

Este comando irá restringir o output a dados relativos ao programa principal *naive* ordenando as funções utilizadas pelo

programa desde a mais custosa para o tempo de execução até a menos custosa. Segue-se o output do comando relativo ao evento *cpu-clock* para cada conjunto de dados:

Para Conjunto de dados 1 (48*48):

```
# Samples: 2 of event 'cpu-clock'
# Event count (approx.): 2
#
# Overhead Command Shared Object Symbol
# ..... .....
# 50.00%  naive  naive  [...] run_no_events
```

Para Conjunto de dados 2 (144*144):

```
# Samples: 2 of event 'cpu-clock'
# Event count (approx.): 2
#
# Overhead Command Shared Object Symbol
# ..... .....
# 50.00%  naive  naive  [...] multiply_matrices
```

Para Conjunto de dados 3 (620*620):

```
# Samples: 1K of event 'cpu-clock'
# Event count (approx.): 1786
#
# Overhead Command Shared Object Symbol
# ..... .....
# 97.20%  naive  [...] multiply_matrices
# 1.05%   naive  [...] initialize_matrices
# 0.06%   naive  [...] rand@plt
```

Para o menor conjunto de dados vemos que a função mais custosa para o tempo de execução é *run_no_events* da aplicação *naive*. Para os outros conjuntos vemos ser *multiply_matrices*. *multiply_matrices* não é uma surpresa já que é expectável que a multiplicação de matrizes seja o passo mais custoso do programa *naive*, no entanto *run_no_events* merece um estudo mais aprofundado, olhamos então para o código dessa função:

```
void run_no_events()
{
    initialize_matrices();
    multiply_matrices();
}
```

Aqui vemos que esta função consiste na inicialização e na multiplicação das matrizes. Como a multiplicação de matrizes em si está inserida nesta função e como o conjunto de dados mais pequeno é provavelmente pequeno o suficiente para que a inicialização seja um *bottleneck* semelhante ao da multiplicação em si então o facto de esta função ser o maior *bottleneck* não surpreende. Isto leva a concluir que a razão pela qual chamadas a *libc-2.12.so* e *ld-2.12.so* têm uma influência tão grande no tempo de execução acaba por ser pela utilização de funções como *rand* ao inicializar as matrizes, daí o facto de serem um maior *bottleneck* para os conjuntos de dados menores já que estes irão ter maior tempo de execução despendido na geração aleatória de pequenas matrizes do que na multiplicação dessas mesmas matrizes. Isto demonstra um caso interessante que é: por vezes algumas aplicações podem perder tempo significativo no uso de bibliotecas externas e não no código da aplicação em si.

Podemos agora observar mais profundamente o código de *multiply_matrices()* ao correr o seguinte comando para o terceiro conjunto de dados:

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

Isto irá apresentar o código da função *multiply_matrices()* *disassembled* e anotado indicando quais as linhas de código mais custosas. Como é normal em código *disassembled* certos troços do código poderão estar repetidos ou re-escritos de maneiras menos intuitivas, no entanto deverá ser possível mesmo assim identificar os troços de código mais relevantes. Segue-se um excerto do output obtido, mas ao invés do uso direto do comando apresentado anteriormente usa-se a interface TUI de forma a observar a informação mais claramente:

<pre>20.46 0.80 37.47 13.68 0.06 27.30 I 0.17</pre>	<pre>float sum = 0.0; for (k = 0 ; k < MSIZE ; k++) { sum = sum + (matrix_a[i][k] * matrix_b[k][j]); } 20.46: movss (%rcx),%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++) { add \$0x1,%eax sum = sum + (matrix_a[i][k] * matrix_b[k][j]); 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++) { add \$0x4,%rcx add \$0x9b0,%rdx cmp \$0x26c,%eax sum = sum + (matrix_a[i][k] * matrix_b[k][j]); 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++) { jne 50 void multiply_matrices() { int i, j, k; for (i = 0 ; i < MSIZE ; i++) { for (j = 0 ; j < MSIZE ; j++) { 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; movss %xmm1,(%rdi) void multiply_matrices() { int i, j, k; for (i = 0 ; i < MSIZE ; i++) { for (j = 0 ; j < MSIZE ; j++) { add \$0x4,%rdi cmp \$0x26c,%esi jne 30 void multiply_matrices() { int i, j, k; for (i = 0 ; i < MSIZE ; i++) { add \$0x1,%r10d cmp \$0x26c,%r10d jne 11 void run_no_events() { initialize_matrices(); multiply_matrices(); } add \$0x18,%rsp retq</pre>
---	--

Aqui vemos que as linhas com mais impacto estão todas relacionadas com *sum = sum + (matrix_a[i][k] * matrix_b[k][j])*; que é o corpo do ciclo mais interno e também relacionadas com a própria condição do ciclo mais interno.

Não é surpresa que o troço de código *sum = sum + (matrix_a[i][k] * matrix_b[k][j])*; seja custoso já que por

cada iteração do ciclo mais interno esta linha de código terá que aceder a valores das matrizes que estão em memória e sabe-se que instruções com acessos à memória são mais custosas. Para além disto é feita uma multiplicação nesta linha, o que é uma operação muito mais custosa que uma simples soma, logo é normal que tenha mais impacto que outras linhas já que é a única que realiza uma multiplicação.

Quanto à condição do ciclo interior ser assim tão custosa surpreende um pouco já que consiste numa operação bastante simples comparada com outras mais custosas como por exemplo: **matrix_r[i][j] = sum ;**. Quanto a isto o tutorial diz que não se devem levar demasiado a sério estes valores para linhas individuais já que os seus resultados são resultantes de uma aproximação instantânea.

O tutorial indica que a ferramenta PERF, de forma a reunir os dados necessários para apresentar os resultados de profiling, usa o clock do sistema Linux para medir a passagem do tempo. Sempre que um dado intervalo de tempo passa o programa é interrompido e a ferramenta PERF determina o que o CPU estava a fazer aquando da interrupção guardando várias informações relevantes num buffer a que chamamos de uma amostra que é depois escrita num ficheiro. No fim do profiling este ficheiro irá ter várias amostras que serão então posteriormente observadas por *perf report* e por *perf annotate* agregando estas amostras de forma a mostrar os dados de profiling finais. Como PERF opera com este modo estatística de organização de amostras, entende-se que é necessário que o número de amostras seja realmente grande o suficiente para se obter uma boa estatística como resultado reduzindo assim o erro das medições. Vemos então que o programa deve correr tempo suficiente para que as estatísticas sejam mais fiáveis, por essa razão não mostramos o código ‘disassembled’ dos conjuntos de dados mais pequenos já que o processo de profiling terminou demasiado rápido e por isso os valores não são tão relevantes como os obtidos para o maior conjunto de dados que demorou mais tempo.

Surge então a questão: Quanto tempo é suficiente para os dados estatísticos serem realmente fiáveis? A resposta depende das características do programa a ser testado. Vemos nos outputs anteriores que o nosso programa *naive* correu, para o maior conjunto de dados, durante 426.076220 msec e que, durante esse tempo, foram feitas 1786 amostras. Estes valores são bastante superiores aos obtidos para os conjuntos de dados menores onde só houve tempo para realizar 2 amostras. Aplicações como esta multiplicação de matrizes normalmente têm ciclos internos com atividade bastante intensa, logo podemos concluir que a grande maioria destas amostras foram realizadas quando o programa estava no ciclo mais interno e entende-se que seja esta a razão pela qual a condição do ciclo mais interno e o corpo do mesmo sejam as linhas indicadas como as mais custosas pelo código ‘disassembled’ anotado.

Agora que entendemos o processo como os *hotspots* de um programa são encontrados, entendemos que aplicações mais complexas necessitariam de mais amostras e, por consequência mais tempo, enquanto que outras dariam na mesma resultados fiáveis com menos amostras e menos tempo. O tutorial aconselha que se tente obter pelo menos entre 100 a 500 amostras nas zonas que a ferramenta considera serem *hotspots*.

Uma outra forma de aumentar o número de amostras (para além de fazer o programa correr mais tempo) poderá ser reduzir o intervalo entre interrupções e amostras do programas, assim aumenta-se a resolução e diminui-se o erro ao se aumentar a frequência de amostragem. Para isto utiliza-se o seguinte comando:

```
perf record -e cpu-clock --freq=8000 ./naive
```

obtendo o seguinte output para o maior conjunto de dados:

Para Conjunto de dados 3 (620*620):

```
[a72293@compute-431-10 perf]$ perf record -e cpu-clock --freq=8000 ./naive
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.142 MB perf.data (~6183 samples) ]
[a72293@compute-431-10 perf]$
```

Seguidamente usa-se o comando:

```
perf report -stdio -show-nr-samples -dsos=naive
```

obtendo o seguinte output para o maior conjunto de dados:

Para Conjunto de dados 3 (620*620):

```
[a72293@compute-431-10 perf]$ perf report --stdio --show-nr-samples --dsos=naive
# To display the perf.data header info, please use --header/--header-only options.
#
# dso: naive
# Samples: 3K of event 'cpu-clock'
# Event count (approx.): 3492
#
# Overhead      Samples   Command           Symbol
# .....          .....
#
# 96.48%        3369    naive  [.] multiply_matrices
  0.66%         23     naive  [.] initialize_matrices
  0.06%         2     naive  [.] rand@plt
[a72293@compute-431-10 perf]$
```

Aqui vemos que foram feitas perto do dobro das amostras que anteriormente. Em concreto fizeram-se 3492 amostras. Não surpreende o facto de se fazer o dobro das amostras já que, ao usar uma resolução de freqüência de 8000 estamos a usar o dobro da resolução usada anteriormente (que por defeito é 4000), logo seria de esperar que se realizassem o dobro das amostras.

Podemos confirmar que por defeito a frequência de amostragem usada é 4000 fazendo de novo a amostragem sem especificar nenhuma freqüência e de seguida usar o comando:

```
perf evlist -F
```

que nos mostra os eventos medidos e a freqüência de amostragem usada para cada um. segue-se o respetivo output:

```
[a72293@compute-431-10 perf]$ perf evlist -F
cpu-clock: sample_freq=4000
faults: sample_freq=4000
[a72293@compute-431-10 perf]$
```

Embora mais amostras reduzam o erro é preciso ter cuidado com a freqüência a usar. Uma maior freqüência de amostragem significa que vamos ter muito mais amostras, logo temos um muito maior *overhead* de coletar e armazenar toda essa informação. Se tivermos a correr em *single-core* então tanto o PERF como a aplicação têm de competir pelos mesmos recursos o que pode aumentar o tempo de execução drasticamente. Desta forma, com demasiadas amostras, o resultado final do profiling não será realista e perde-se a relevância que este profiling poderia vir a ter tornando-se então muito relevante ter cuidado na escolha da freqüência de amostragem.

segue-se um estudo dos comandos iniciais para o evento *page-faults*. Para o comando:

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

obteu-se o seguinte output:

Para Conjunto de dados 1 (48*48):

```
# Samples: 11 of event 'faults'
# Event count (approx.): 403
#
# Overhead Command Shared Object
# ..... .....
#
 96.77%  naive ld-2.12.so
  1.99%  naive naive
  0.74%  naive [kernel.kallsyms]
  0.50%  naive libc-2.12.so

[a72293@compute-431-10 perf]$
```

Para Conjunto de dados 2 (144*144):

```
# Samples: 12 of event 'faults'
# Event count (approx.): 505
#
# Overhead Command Shared Object
# ..... .....
#
 70.50%  naive ld-2.12.so
 27.52%  naive libc-2.12.so
  1.39%  naive naive
  0.59%  naive [kernel.kallsyms]
```

Para Conjunto de dados 3 (620*620):

```
# Samples: 35 of event 'faults'
# Event count (approx.): 794
#
# Overhead Command Shared Object
# ..... .....
#
 79.09%  naive naive
 20.28%  naive ld-2.12.so
  0.38%  naive [kernel.kallsyms]
  0.25%  naive libc-2.12.so
```

e para o seguinte comando:

perf report -stdio -dsos=naive

Obteve-se o seguinte output:

Para Conjunto de dados 1 (48*48):

```
# Samples: 12 of event 'faults'
# Event count (approx.): 505
#
# Overhead Command Shared Object           Symbol
# ..... .....
#
 27.13%  naive libc-2.12.so  [.]
 1.39%  naive naive      [.]
  0.40%  naive libc-2.12.so  [.]
                                __exit
                                initialize_matrices
                                strstr_sse2
```

Para Conjunto de dados 2 (144*144):

```
# Samples: 12 of event 'faults'
# Event count (approx.): 505
#
# Overhead Command Shared Object           Symbol
# ..... .....
#
 27.13%  naive libc-2.12.so  [.]
 1.39%  naive naive      [.]
  0.40%  naive libc-2.12.so  [.]
                                __exit
                                initialize_matrices
                                strstr_sse2
```

Para Conjunto de dados 3 (620*620):

```
# Samples: 35 of event 'faults'
# Event count (approx.): 794
#
# Overhead Command           Symbol
# ..... .....
#
 66.74%  naive  [.]
                                initialize_matrices
```

Observamos agora o output obtido de forma a confirmar se os resultados obtidos para o evento *page-faults* fazem

sentido. Para os menores conjuntos de dados nota-se que não é o programa 'naive' que causa a maioria das 'page faults', nota-se até que não chega a contribuir para as 'page faults' mais do que 2%. No entanto para o maior conjunto de dados notamos já que o programa 'naive' é o que mais contribui. Mesmo assim vemos ainda que, dentro do programa para todos os conjuntos de dados, a função que mais contribui é **initialize_matrices**. Isto leva-nos a crer que, no programa 'naive' testado, a grande maioria das 'page faults' se deve à inicialização natural das matrizes e, por essa razão, são valores normais onde não há muito que possamos fazer para melhorar.

C. Parte 2 - Contagem de eventos de desempenho de hardware

Tendo agora identificado a própria multiplicação de matrizes em si como a parte mais intensiva a nível de tempo de execução do programa 'naive' podemos comparar o desempenho entre esta implementação 'naive' e a implementação 'interchange' (explicada na introdução) com eventos de desempenho. Estes eventos são específicos da microarquitetura da máquina a usar e resultam diretamente de condições do hardware. Observamos agora brevemente como os medir, depois uma breve explicação teórica sobre o que esperar do desempenho das duas versões de multiplicação de matrizes a testar e depois ainda mostra-se uma tabela com os resultados obtidos para os eventos de desempenho para cada implementação e conjunto de dados.

Temos, por exemplo, o seguinte comando:

perf stat -e cpu-cycles,instructions ./naive

Que nos permite medir os seguintes eventos de desempenho:

- *cpu-cycles* - indica o número de ciclos do processador necessários para o programa terminar;
- *instructions* - indica o número de instruções executadas e bem sucedidas.

Segue-se o output do comando especificado:

Para Conjunto de dados 3 (620*620):

```
[a72293@compute-431-10 perf]$ perf stat -e cpu-cycles,instructions ./naive
Performance counter stats for './naive':
                                         cpu-cycles
                                         instructions          #    1.56  insns per cycle
                                         1252641108
                                         1957400008
                                         0.433907353 seconds time elapsed
[a72293@compute-431-10 perf]$
```

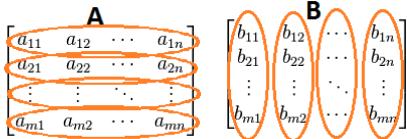
Observamos agora teoricamente as implementações da multiplicação de matrizes usadas neste documento. A primeira abordagem (implementação 'naive') consiste na implementação clássica da multiplicação de matrizes, ou seja três ciclos onde o índice de cada um será relativo a diferentes 'componentes' da multiplicação. Neste caso as diferentes 'componentes' são as linhas (i), colunas (j) e acumulação de multiplicações para cada valor da matriz resultante (k). As diferentes implementações referidas acima consistem na alteração da ordem desses índices. Nesta implementação 'naive' os índices dos ciclos estão ordenados da forma ijk sendo esta a implementação mais básica:

```
void multiply_matrices()
{
    int i, j, k ;
    for (i = 0 ; i < MSIZE ; i++ ) {
```

```

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

```



Neste algoritmo, temos dois acessos matriciais por iteração do ciclo interno e podemos notar que, enquanto os valores da matriz $matrix_a$ são acedidos por linhas, a $matrix_b$ é acedida por colunas. Isso significa que procurar valores da $matrix_b$ é mais custoso do que procurar na $matrix_a$, pois não são acessados em memória contígua o que significa que utiliza a cache de forma inefficiente. Esta ordem dos ijk usada para os índices dos ciclos irá então acentuar a latência do acesso à memória principal.

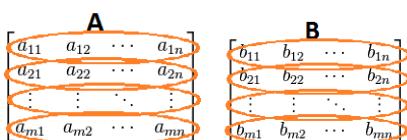
Olhamos agora para a outra implementação a que chamamos 'interchanged' e que segue a ordem de índices ijk para os ciclos:

```

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

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

```



Desta vez acedemos aos elementos das matrizes linha por linha em memória contígua, portanto, não temos o problema de desempenho com acessos de memória descontínua como na versão anterior. Vemos que esta versão nos permitiria até explorar o facto de que o segundo ciclo está a ser executado com o índice k para fazer alguns cálculos que de outra forma deveriam ser feitos para cada iteração do terceiro ciclo. Desta forma, minimizariamós a sobrecarga de ter que procurar valores de matrizes que seriam exatamente iguais aos da iteração anterior.

Comparamo agora os resultados obtidos para diferentes eventos de desempenho para os diferentes conjuntos de dados e a implementação 'interchanged':

e a implementação 'naive':

Para Conjunto de dados 1 (48*48):

Performance counter stats for './naive':

	cpu-cycles	instructions	#	1.16 insns per cycle
1482604	cache-references	#	10.443 % of all cache refs	
1727012	cache-misses	#	10.443 % of all cache refs	
17246	bus-cycles			
1801	L1-dcache-loads			
<not supported>	L1-dcache-load-misses	#	2.78% of all L1-dcache hits	
436291	L1-dcache-store-misses			
12122	LLC-loads			
<not counted>	LLC-load-misses			
<not counted>	LLC-stores			
<not counted>	LLC-store-misses			
<not counted>	dTLB-load-misses			
<not counted>	dTLB-store-misses			
<not counted>	iTLB-load-misses			
<not counted>	branch-loads			
<not counted>	branch-load-misses			

0.002356601 seconds time elapsed

Para Conjunto de dados 2 (144*144):

Performance counter stats for './naive':

	cpu-cycles	instructions	#	17.286 % of all cache refs
<not counted>	cache-references	#	17.286 % of all cache refs	
19235	cache-misses	#	17.286 % of all cache refs	
3325	bus-cycles			
<not supported>	L1-dcache-loads			
2613178	L1-dcache-load-misses	#	3.97% of all L1-dcache hits	
103630	L1-dcache-store-misses		[78.46%]	
2092	LLC-loads		[65.61%]	
893	LLC-load-misses	#	0.00% of all LLC-cache hits	
0	LLC-stores		[56.47%]	
6051	LLC-store-misses		[19.73%]	
92	dTLB-load-misses		[6.53%]	
15412	dTLB-store-misses		[6.53%]	
506	iTLB-load-misses		[6.53%]	
<not counted>	branch-loads			
<not counted>	branch-load-misses			

0.009170949 seconds time elapsed

Para Conjunto de dados 3 (620*620):

Performance counter stats for './naive':

	cpu-cycles	instructions	#	[31.28%]
1209331374	cache-references	#	1.39 insns per cycle	[37.56%]
1675531133	cache-misses	#	0.078 % of all cache refs	[37.86%]
17327953	bus-cycles			
13435	L1-dcache-loads			
<not supported>	L1-dcache-load-misses	#	52.67% of all L1-dcache hits	[38.16%]
481052106	L1-dcache-store-misses		[25.63%]	
253359215	LLC-loads		[25.57%]	
511386	LLC-load-misses	#	0.02% of all LLC-cache hits	[25.51%]
16838526	LLC-stores		[12.71%]	
3155	LLC-store-misses		[12.48%]	
341945	dTLB-load-misses		[18.61%]	
10443	dTLB-store-misses		[24.73%]	
34930	iTLB-load-misses		[24.56%]	
2936	branch-loads		[25.15%]	
118	branch-load-misses		[25.09%]	
238156090				
9188669				

0.425479069 seconds time elapsed

Comparamo agora os resultados obtidos para diferentes eventos de desempenho para os diferentes conjuntos de dados e a implementação 'interchanged':

Para Conjunto de dados 1 (48*48):

Performance counter stats for './interchange':

	cpu-cycles	instructions	#	1.23 insns per cycle
1502916	cache-references	#	13.030 % of all cache refs	
1854830	cache-misses	#	13.030 % of all cache refs	
17513	bus-cycles			
2282	L1-dcache-loads			
<not supported>	L1-dcache-load-misses	#	2.77% of all L1-dcache hits	
440766	L1-dcache-store-misses			
12223	LLC-loads			
<not counted>	LLC-load-misses			
<not counted>	LLC-stores			
<not counted>	LLC-store-misses			
<not counted>	dTLB-load-misses			
<not counted>	dTLB-store-misses			
<not counted>	iTLB-load-misses			
<not counted>	branch-loads			
<not counted>	branch-load-misses			

0.002185576 seconds time elapsed

Para Conjunto de dados 2 (144*144):

Performance counter stats for './interchange':				
<not counted>	cpu-cycles			
2420886	instructions			
20597	cache-references			
1957	cache-misses	# 9.501 % of all cache refs		
<not supported>	bus-cycles			
3695384	L1-dcache-loads			
135270	L1-dcache-load-misses	# 3.66% of all L1-dcache hits	[65.63%]	
73	L1-dcache-store-misses		[56.38%]	
729	LLC-loads			
257	LLC-load-misses	# 35.25% of all LLC-cache hits	[44.67%]	
20540	LLC-stores		[4.17%]	
<not counted>	LLC-store-misses			
<not counted>	dTLB-load-misses			
<not counted>	dTLB-store-misses			
<not counted>	iTLB-load-misses			
<not counted>	branch-loads			
<not counted>	branch-load-misses			
0.009005738 seconds time elapsed				
Para	Conjunto	de	dados	3 (620*620):
Performance counter stats for './interchange':				
745042943	cpu-cycles	[31.53%]		
2164856711	instructions	# 2.91 insns per cycle	[37.99%]	
644330	cache-references		[38.24%]	
96721	cache-misses	# 15.011 % of all cache refs	[38.48%]	
<not supported>	bus-cycles			
479742280	L1-dcache-loads		[38.71%]	
14860324	L1-dcache-load-misses	# 3.10% of all L1-dcache hits	[38.95%]	
112018	L1-dcache-store-misses		[25.86%]	
477482	LLC-loads		[25.52%]	
4319	LLC-load-misses	# 0.90% of all LLC-cache hits	[25.14%]	
294913	LLC-stores		[12.17%]	
28669	LLC-store-misses		[12.75%]	
12518	dTLB-load-misses		[19.05%]	
2033	dTLB-store-misses		[25.29%]	
12	iTLB-load-misses		[25.18%]	
241671100	branch-loads		[25.08%]	
8320927	branch-load-misses		[24.98%]	
0.264739807 seconds time elapsed				

Como vemos os menores conjuntos de dados são tão pequenos que alguns eventos não puderam ser calculados. Por esta razão iremos agora comparar as duas implementações mas iremos usar somente o valor dos eventos obtido para o maior conjunto de dados. Segue-se uma tabela a comparar os valores indicados:

Evento	'Naive'	'interchanged'
cpu-cycles	1209331374	745042943
instructions	1675531133	2164856711
cache-references	17327953	644330
cache-misses	13435	96721
L1-dcache-loads	481052106	479742280
L1-dcache-load-misses	253359215	14860324
L1-dcache-store-misses	511386	112018
LLC-loads	16838526	477482
LLC-load-misses	3155	4319
LLC-stores	341945	294913
LLC-store-misses	10443	28669
dTLB-load-misses	34930	12518
dTLB-store-misses	2936	2033
iTLB-load-misses	118	12
branch-loads	238156090	241671100
branch-load-misses	9188669	8320927

Observando estes valores notamos em vários pormenores de interesse:

- O resultado do evento *cpu-cycles* é menor para a versão 'interchanged'. Isto implica que a versão 'interchanged' necessita de menos ciclos de clock do CPU para completar a multiplicação, desta forma, é uma versão mais rápida que a 'naive' em termos de tempo de execução (como já se tinha previsto anteriormente).
- O resultado do evento *instructions* indica o número de instruções realizadas com sucesso e é semelhante para as duas versões, sendo que a versão 'interchanged' obteve um valor um pouco maior.

- Nota-se um resultado muito baixo com ambas as versões para o evento *iTLB-load-misses* o que nos leva a acreditar que 'instruction misses' não são um fator limitante do desempenho em nenhuma das implementações.
- No geral a implementação 'interchanged' tem muito menos *cache-references* já que o facto de aceder às matrizes linha por linha permite um melhor uso da localidade espacial da cache. Isto explica também o menor número de *LLC-loads* que a versão 'interchanged' tem, já que o programa 'naive' terá que carregar informação da memória principal com mais frequência.

Com estes pormenores observados conseguimos provar as vantagens da versão 'interchanged' para a 'naive' e prova-se também que estas acontecem pelas razões indicadas quando a teoria das duas implementações foi explicitada.

Agora, de forma a ainda melhor se conseguir interpretar os resultados na tabela anterior determinam-se rácios e percentagens que facilitam a interpretação (ao invés de ter que ler e compreender grandes números). Segue-se uma tabela com os rácios e percentagens a calcular e a sua formula, de seguida apresenta-se uma outra tabela com os valores calculados de acordo com essas formulas:

Rácio ou Percentagem	Formula
Instructions per cycle	instructions / cpu-cycles
L1 cache miss ratio	L1-dcache-loads / L1-dcache-load-misses
L1 cache miss rate PTI	L1-dcache-load-misses / (instructions/1000)
LLC miss ratio	LLC-loads / LLC-load-misses
LLC miss rate PTI	LLC-load-misses / (instructions/1000)
Data TLB miss ratio	dTLB-load-misses / cache-references
Data TLB miss rate PTI	dTLB-load-misses / (instructions / 1000)
Branch load mispredict ratio	branch-load-misses / branch-loads
Branch load mispredict rate PTI	branch-load-misses / (instructions / 1000)

Nome do Evento	'Naive'	'interchanged'
Tempo de execução(s)	0.425479069	0.264739807
Instructions per cycle	1.385502079	2.905680446
L1 cache miss ratio	1.898695913	32.28343339
L1 cache miss rate PTI	151.2112846	6.864345305
LLC miss ratio	5337.092235	110.5538319
LLC miss rate PTI	0.018829850	0.01995051
Data TLB miss ratio	0.002015818	0.01942793289
Data TLB miss rate PTI	0.02084712084	0.005782368
Branch load mispredict ratio	0.03858254895	0.03443079044
Branch load mispredict rate PTI	5.484033581	3.843638684

Com estes valores vemos confirmadas as conclusões alcançadas anteriormente. Vemos, por exemplo que, para a cache primeiro nível (L1) há um rácio de loads/misses maior para

a implementação 'naive'. Isto resultou de um maior número de misses por parte da implementação 'naive'. Isto reflete a forma inefficiente como os índices das matrizes são acessados. Outra coisa que reflete isto é o facto de para a cache de último nível (LLC) haver um rácio loads/misses muito maior para a implementação 'naive'. Como isto acontece por haver um muito maior número de loads nesta cache do que na outra implementação isto prova que as suas linhas da cache são invalidadas muito mais vezes e terá que ir novamente à memória principal carregar valores das matrizes, como o conjunto de dados usado é grande então é na cache de último nível que se verifica isto.

O tempo de execução da versão 'interchanged' é muito menor, cerca de metade do tempo de execução, e isto reflete-se nas Instructions per cycle já que se vê que por ciclo a verão 'interchanged' realiza perto do dobro das instruções do que a versão 'naive'.

D. Parte 3 - Observação de perfis de eventos de hardware

Nesta parte do tutorial iremos agora fazer profiling como fizemos na primeira parte, mas em vez de usarmos eventos de software vamos usar eventos de hardware que já introduzimos na parte 2.

Iremos agora usar, para os restantes testes, um tamanho para o conjunto de dados ainda maior que o usado até agora. O valor 2048×2048 é um bom valor pois 3 matrizes 2048×2048 irão encher por completo a cache de último nível e obrigar o programa a ir carregar dados à memória principal.

Faz-se agora o profiling das duas implementações com **perf record** como feito na parte 1. Desta vez usam-se eventos de desempenho de hardware e usamos a flag **-c** de modo a definir a frequência de amostragem a 100000 ciclos de CPU por segundo. Assim, com o seguinte comando:

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

obteve-se o seguinte output:

Para	conjunto	de	dados
2048*2048	e	implementação	'naive'

```
[ perf record: Woken up 11 times to write data ]
[ perf record: Captured and wrote 4.683 MB perf.data (~204620 samples) ]
```

depois o mesmo comando é feito para a implementação 'interchanged' e surge o seguinte output:

Para	conjunto	de	dados	2048*2048
e	implementação	'interchanged'		

```
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.509 MB perf.data (~22252 samples) ]
```

de forma a observar os dados do profiling realizado usa-se o seguinte comando:

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

Este comando utiliza a flag **--percent-limit** com o valor **0.1** de forma a suprimir do output eventos com percentagens inferiores ao valor 0.1. Percentagens tão pequenas normalmente não têm influência significativa na performance do programa e por isso podem ser ignoradas.

para o programa 'naive' obteve-se o seguinte output:

```

# Samples: 79K of event 'cpu-cycles'
# Event count (approx.): 7926800000
#
# Overhead  Command      Shared Object          Symbol
# .....      .....      .....      .....      .....
#
#      99.42%    naive  naive           [.] run_no_events

# Samples: 28K of event 'instructions'
# Event count (approx.): 2846500000
#
# Overhead  Command      Shared Object          Symbol
# .....      .....      .....      .....      .....
#
#      94.53%    naive  naive           [.] run_no_events
#      2.63%    naive  naive           [.] initialize_matrices
#      1.37%    naive  libc-2.12.so     [.] __random_
#      0.54%    naive  libc-2.12.so     [.] __random_r
#      0.34%    naive  libc-2.12.so     [.] rand
#      0.18%    naive  naive           [.] rand@plt

# Samples: 3K of event 'cache-references'
# Event count (approx.): 329500000
#
# Overhead  Command      Shared Object          Symbol
# .....      .....      .....      .....      .....
#
#      99.73%    naive  naive           [.] run_no_events
#      0.12%    naive  [kernel.kallsyms] [k] clear_page_c

# Samples: 2K of event 'cache-misses'
# Event count (approx.): 285500000
#
# Overhead  Command      Shared Object          Symbol
# .....      .....      .....      .....      .....
#
#      99.75%    naive  naive           [.] run_no_events
#      0.21%    naive  [kernel.kallsyms] [k] clear_page_c

# Samples: 3K of event 'LLC-loads'
# Event count (approx.): 326800000
#
# Overhead  Command      Shared Object          Symbol
# .....      .....      .....      .....      .....
#
#      99.94%    naive  naive           [.] run_no_events

# Samples: 2K of event 'LLC-load-misses'
# Event count (approx.): 284300000
#
# Overhead  Command      Shared Object          Symbol
# .....      .....      .....      .....      .....
#
#      100.00%   naive  naive           [.] run_no_events

# Samples: 0  of event 'dTLB-load-misses'
# Event count (approx.): 0
#
# Overhead  Command      Shared Object  Symbol
# .....      .....      .....      .....      .....
#
#      100.00%   naive  naive           [.] run_no_events

# Samples: 2K of event 'LLC-load-misses'
# Event count (approx.): 284300000
#
# Overhead  Command      Shared Object          Symbol
# .....      .....      .....      .....      .....
#
#      100.00%   naive  naive           [.] run_no_events

# Samples: 0  of event 'dTLB-load-misses'
# Event count (approx.): 0
#
# Overhead  Command      Shared Object  Symbol
# .....      .....      .....      .....      .....
#
#      100.00%   naive  naive           [.] run_no_events

# Samples: 2K of event 'branches'
# Event count (approx.): 254500000
#
# Overhead  Command      Shared Object          Symbol
# .....      .....      .....      .....      .....
#
#      91.32%    naive  naive           [.] run_no_events
#      3.81%    naive  naive           [.] initialize_matrices
#      2.32%    naive  libc-2.12.so     [.] __random_
#      0.90%    naive  libc-2.12.so     [.] __random
#      0.83%    naive  naive           [.] rand@plt
#      0.28%    naive  libc-2.12.so     [.] rand

# Samples: 0  of event 'branch-misses'
# Event count (approx.): 0

```

para o programa 'interchanged' obteve-se o seguinte output:

```
# Samples: 3K of event 'cpu-cycles'
# Event count (approx.): 313100000
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
  93.48% interchange  interchange  [.]
  1.82% interchange  interchange  [.] run_no_events
  1.37% interchange  libc-2.12.so  [.] initialize_matrices
  1.09% interchange  libc-2.12.so  [.] __random
  0.42% interchange  libc-2.12.so  [.] __random_r
  0.38% interchange  [kernel.kallsyms] [k] clear_page_c
  0.29% interchange  interchange  [.] rand@plt
  0.13% interchange  [kernel.kallsyms] [k] __mem_cgroup_commit_charge

# Samples: 9K of event 'instructions'
# Event count (approx.): 908900000
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
  85.79% interchange  interchange  [.]
  7.09% interchange  interchange  [.] run_no_events
  3.71% interchange  libc-2.12.so  [.] initialize_matrices
  1.49% interchange  libc-2.12.so  [.] __random
  1.12% interchange  libc-2.12.so  [.] __random_r
  0.59% interchange  interchange  [.] rand
  0.59% interchange  interchange  [.] rand@plt

# Samples: 7 of event 'cache-references'
# Event count (approx.): 700000
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
   71.43% interchange  [kernel.kallsyms] [k] clear_page_c
   28.57% interchange  interchange  [.] run_no_events

# Samples: 8 of event 'cache-misses'
# Event count (approx.): 800000
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
   75.00% interchange  [kernel.kallsyms] [k] clear_page_c
   25.00% interchange  interchange  [.] run_no_events

# Samples: 1 of event 'LLC-loads'
# Event count (approx.): 100000
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
  100.00% interchange  interchange  [.] run_no_events

# Samples: 1 of event 'LLC-load-misses'
# Event count (approx.): 100000
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
  100.00% interchange  interchange  [.] run_no_events

# Samples: 0 of event 'dTLB-load-misses'
# Event count (approx.): 0
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
# Samples: 886 of event 'branches'
# Event count (approx.): 88600000
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
   71.11% interchange  interchange  [.] run_no_events
   13.32% interchange  interchange  [.] initialize_matrices
   8.24% interchange  libc-2.12.so  [.] __random_r
   3.61% interchange  libc-2.12.so  [.] __random
   2.37% interchange  interchange  [.] rand@plt
   1.02% interchange  libc-2.12.so  [.] rand
   0.11% interchange  [kernel.kallsyms] [k] clear_page_c
   0.11% interchange  [kernel.kallsyms] [k] hrtimer_interrupt
   0.11% interchange  [kernel.kallsyms] [k] rcu_process_gp_end

# Samples: 0 of event 'branch-misses'
# Event count (approx.): 0
#
# Overhead    Command   Shared Object           Symbol
# .....      .....      .....      .....      ...
#
```

mais uma vez vemos uma maior percentagem para a função **run_no_events** que chama **multiply_matrices**, reforçando assim a ideia de que esta função é o maior 'hot spot' dos programas testados.

Por fim vemos agora somente qual o overhead de amostragem para cada implementação já que este overhead, se for superior numa das versões do que noutra introduz alguma 'injustiça' nos resultados obtidos. De forma a fazer este cálculo usa-se a fórmula:

$$\frac{\text{tempo_com_amostragem} - \text{tempo_sem_amostragem}}{\text{tempo_sem_amostragem}} \quad (1)$$

Com o tempo_sem_amostras obtido com recurso ao comando **time** da seguinte forma:

Para o terceiro conjunto de dados 620*620

```
[a72293@compute-431-10 perf]$ time ./naive
real    0m0.431s
user    0m0.428s
sys     0m0.002s
[a72293@compute-431-10 perf]$ time ./interchange
real    0m0.264s
user    0m0.262s
sys     0m0.000s
[a72293@compute-431-10 perf]$
```

Calculou-se para implementação 'naive', para o terceiro conjunto de dados:

$$\frac{0.425479069 - 0.431}{0.425479069} = 1.30\% \quad (2)$$

e depois para a implementação 'interchanged', para o terceiro conjunto de dados:

$$\frac{0.264739807 - 0.264}{0.264739807} = 2,79\% \quad (3)$$

Como vemos o overhead da amostragem é semelhante para ambas as implementações, logo as medições foram 'justas' e podemos confiar nos resultados obtidos e nas conclusões tiradas

E. FlameGraphs

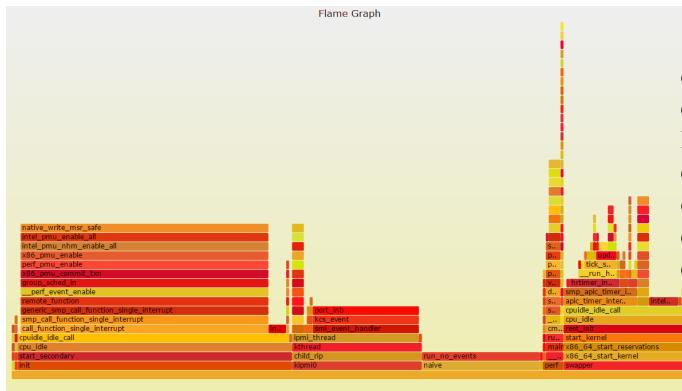
Terminado o tutorial reparou-se que o output dado pelo PERF por vezes era demasiado grande. Se estamos a observar um programa tão simples e "pequeno" quanto a multiplicação de matrizes (como se vê pelas imagens anteriores) então assume-se que para um programa mais complexo o output seria demasiado extenso para ser facilmente interpretado. Como solução a isto pretendeu-se estudar como fazer um **FlameGraph** que é um gráfico de barras que nos permite a apresentação do output de PERF de uma maneira mais concisa e fácil de entender.

Seguem-se os comandos usados para realizar um perfil de desempenho com PERF e posterior tratamento do output para geração dos FlameGraphs (1 para cada implementação com o terceiro conjunto de dados (620*620)):

Para 'naive' com conjunto de dados 620*620

```
perf record -ag -F 99 ./naive
perf script | /share/jade/SOFT/FlameGraph/stackcollapse-perf.pl > out.perf-folded
cat out.perf-folded | /share/jade/SOFT/FlameGraph/flamegraph.pl > perf_naive.svg
```

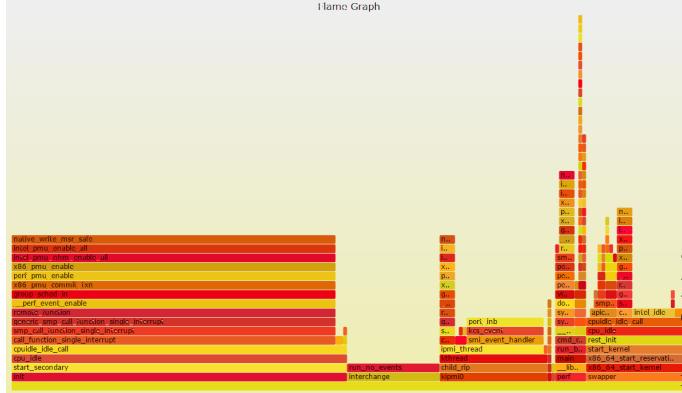
que gerou o seguinte FlameGraph:



Para 'interchanged' com conjunto de dados 620*620

```
perf record -ag -F 99 ./interchange
perf script | /share/jade/SOFT/FlameGraph/stackcollapse-perf.pl > out.perf-folded
cat out.perf-folded | /share/jade/SOFT/FlameGraph/flamegraph.pl > perf_interchange.svg
```

que gerou o seguinte FlameGraph:



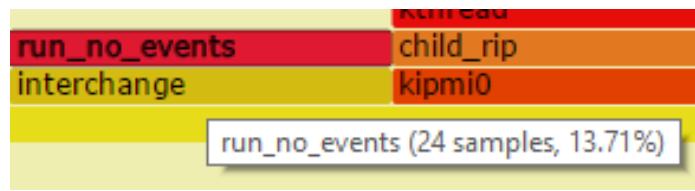
Por estas imagens é difícil ler o gráfico, no entanto estes podem ser consultados nos ficheiros enviados junto com este documento.

Ambos os FlameGraphs resultaram em resultados semelhantes e ambos, assim como durante o tutorial, identificam **run_no_events** como o **hot spot** da aplicação 'naive', como se mostra nas próximas imagens:

Para 'naive'



Para 'interchange'



IV. TUTORIAL - DTRACE

De forma a encontrar os 'hot spots' dos programas desenvolveu-se um script com DTrace que inicia um processo com o programa desejado e para todos os probes entry lançados para um certo pid e um certo executável argumento calcula o wall time e o CPU time total ao longo da execução do programa da cada função utilizada, assim como as vezes que a função foi chamada. Segue-se o código do script em questão:

```
1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3
4
5 dtrace:::BEGIN {
6     printf("Identificação de Hot Spots.\n");
7     printf
8     ("
9
10 pid$target:$::entry
11 /self->start[probefunc] == 0/ {
12     self->start[probefunc] = timestamp;
13     self->vstart[probefunc] = vtimestamp;
14
15     @function_count[ufunc(uregs[R_PC])] = count();
16 }
17
18 19 pid$target:$::return
20 /self->start[probefunc]/{
21
22     this->wall_elapsed = timestamp - self->start[probefunc];
23     self->start[probefunc] = 0;
24     this->cpu_elapsed = vtimestamp - self->vstart[probefunc];
25     self->vstart[probefunc] = 0;
26
27     @function_walltime[ufunc(uregs[R_PC])] = sum(this->wall_elapsed);
28     @function_cputime[ufunc(uregs[R_PC])] = sum(this->cpu_elapsed);
29 }
30
31 dtrace:::END{
32     normalize(@function_walltime, 1000000);
33     normalize(@function_cputime, 1000000);
34
35     printf("%30s | %10s | %10s | %s\n", "Function", "Wall Time", "CPU Time", "Times Called");
36     printf("%30s | %10d ms | %10d ms | %d\n", @function_walltime, @function_cputime,
37     @function_count);
37 }
```

No ponto circundado 1 vemos a maneira como se restringe a captura dos probes entry e return para o pid argumento com **pid\$target**. O ponto circundado 2 mostra o mesmo para o executável argumento que é restrinido com **\$1**.

No ponto circundado 3 vemos que as funções são caracterizadas pelo valor no array uregs na posição R_PC. Este array é uma variável built-in do DTrace e a posição R_PC indica-nos o valor do program counter. Desta forma podemos saber o módulo ao qual a função pertence assim como o seu próprio nome.

No ponto circundado 4 temos já a impressão dos resultados, sendo esta impressão realizada automaticamente mal o processo com pid argumento termine.

Mostra-se agora o output obtido com este script usando o seguinte comando:

```
dtrace -qs hotspots.d -c ./naive naive
a72293@solarisEdu:~/a72293$ dtrace -qs o.d -c ./naive naive
Identificação de Hot Spots.

Function | Wall Time | CPU Time | Times Called
naive'_start | 0 ms | 0 ms | 1
naive'_main | 0 ms | 0 ms | 1
naive'_fsr | 0 ms | 0 ms | 1
naive'_create_result_file | 0 ms | 0 ms | 1
naive'_initialize_matrices | 40 ms | 40 ms | 1
naive'_multiply_matrices | 8340 ms | 8340 ms | 1
```

Neste output vemos que a função do módulo 'naive' que mais tempo consome é a função **multiply_matrices**. Vemos também que a função **initialize_matrices** é a segunda que mais consome. Isto também aconteceu nos resultados obtidos com PERF, onde se pode observar também que a razão pela qual a função **initializa_matrices** é a segunda que mais tempo consome era a geração de números aleatórios, algo exterior ao nosso programa e portanto não era problema de desempenho do nosso programa. Podemos confirmar isto com este script ao utilizar o seguinte comando:

```
dtrace -qs hotspots.d -c ./naive libc.so
```

O output irá agora apresentar as funções utilizadas no decorrer do programa provenientes de libc.so:

a72293@solarisEdu:~/a72293\$ dtrace -qs o.d -c ./naive libc.so			
Identificacao de Hot Spots.			
Function	Wall Time	CPU Time	Times Called
libc.so.1`exit	0 ms	0 ms	1
libc.so.1`_getfp	0 ms	0 ms	1
libc.so.1`strchr	0 ms	0 ms	1
libc.so.1`libc_fini	0 ms	0 ms	1
libc.so.1`strlen	0 ms	0 ms	1
libc.so.1`_ti_critical	0 ms	0 ms	1
libc.so.1`mutex_setup	0 ms	0 ms	1
libc.so.1`_getpid	0 ms	0 ms	1
libc.so.1`set_curthread	0 ms	0 ms	1
libc.so.1`envsize	0 ms	0 ms	1
libc.so.1`syscall	0 ms	0 ms	1
libc.so.1`clean_env	0 ms	0 ms	1
libc.so.1`__lwp_private	0 ms	0 ms	1
libc.so.1`_getrthrt	0 ms	0 ms	1
libc.so.1`getrlimit	0 ms	0 ms	1
libc.so.1`fflush_u	0 ms	0 ms	2
libc.so.1`membar_producer	0 ms	0 ms	1
libc.so.1`get_zone_offset	0 ms	0 ms	1
libc.so.1`__getcontext	0 ms	0 ms	1
libc.so.1`sys186	0 ms	0 ms	1
libc.so.1`atomic_swap_32	0 ms	0 ms	2
libc.so.1`_memset	0 ms	0 ms	4
libc.so.1`setustack	0 ms	0 ms	1
libc.so.1`memcpy	0 ms	0 ms	3
libc.so.1`strncpy	0 ms	0 ms	1
libc.so.1`fflush_l_iops	0 ms	0 ms	1
libc.so.1`tls_modinfo_alloc	0 ms	0 ms	1
libc.so.1`_fpstart	0 ms	0 ms	1
libc.so.1`signal_init	0 ms	0 ms	1
libc.so.1`membar_consumer	0 ms	0 ms	4
libc.so.1`fflush	0 ms	0 ms	1
libc.so.1`findenv	0 ms	0 ms	4
libc.so.1`getiop	0 ms	0 ms	4
libc.so.1`cleanup	0 ms	0 ms	1
libc.so.1`strcmp	0 ms	0 ms	10
libc.so.1`_tls_setup	0 ms	0 ms	1
libc.so.1`getbucketnum	0 ms	0 ms	13
libc.so.1`lmutex_lock	0 ms	0 ms	13
libc.so.1`lmutex_unlock	0 ms	0 ms	13
libc.so.1`_lmutex_unlock	0 ms	0 ms	13
libc.so.1`pthread_setcancelstate	0 ms	0 ms	2
libc.so.1`initial_allocation	0 ms	0 ms	1
libc.so.1`_findiop	0 ms	0 ms	1
libc.so.1`mutex_lock_Impl	0 ms	0 ms	13
libc.so.1`__openat	0 ms	0 ms	1
libc.so.1`pthread_atfork	0 ms	0 ms	1
libc.so.1`_systemcall	0 ms	0 ms	2
libc.so.1`setprogname	0 ms	0 ms	1
libc.so.1`atfork_init	0 ms	0 ms	1
libc.so.1`_open	0 ms	0 ms	1
libc.so.1`thr_keycreate	0 ms	0 ms	1
libc.so.1`set_thread_vars	0 ms	0 ms	1
libc.so.1`lfree	0 ms	0 ms	3
libc.so.1`open	0 ms	0 ms	1
libc.so.1`pthread_key_create	0 ms	0 ms	1
libc.so.1`_endopen	0 ms	0 ms	1
libc.so.1`set_cancel_pending_flag	0 ms	0 ms	22
libc.so.1`init_sigev_thread	0 ms	0 ms	1
libc.so.1`initenv	0 ms	0 ms	4
libc.so.1`init_progname	0 ms	0 ms	1
libc.so.1`fopen	0 ms	0 ms	1
libc.so.1`atexit	0 ms	0 ms	3
libc.so.1`_lmutex_lock	0 ms	0 ms	13
libc.so.1`__tls_static_mods	0 ms	0 ms	1
libc.so.1`_getenv	0 ms	0 ms	4
libc.so.1`init_aio	0 ms	0 ms	1
libc.so.1`lmalloc	0 ms	0 ms	10
libc.so.1`_ti_bind_clear	0 ms	0 ms	20
libc.so.1`_exithandle	0 ms	0 ms	1
libc.so.1`_ti_bind_guard	0 ms	0 ms	10
libc.so.1`libc_init	1 ms	0 ms	1
libc.so.1`rand_r	962 ms	741946 ms	393218
libc.so.1`rand	3338 ms	766992 ms	393373

```
a72293@solarisEdu:~/a72293$
```

Aqui vemos que realmente, das funções do módulo libc.so, é a geração de números aleatórios que as mais influência tem no desempenho do programa. Obtemos assim novamente os mesmos resultados que com PERF.

Podemos agora confirmar os resultados obtidos ainda com um programa chamado **hotuser** desenvolvido por **Brendan Gregg** com o uso de DTrace e obtido em [7]. Este código, ao invés de esperar para capturar probes que surjam naturalmente ao longo do programa, interrompe o programa com uma frequência de 1000 e observa então a função em que o programa se encontra. Vemos assim que este programa funciona de maneira mais semelhante a PERF do que o script apresentado anteriormente. Vemos agora o output deste programa onde se confirmam os resultados anteriores:

```
a72293@solarisEdu:~/a72293$ ./nd.d -c ./naive
Sampling... Hit Ctrl-C to end.

FUNCTION          COUNT   PCNT
naive`0x8050ee0      1   0.0%
ld.so.1`enter        1   0.0%
libc.so.1`rand        5   0.1%
libc.so.1`rand_r       6   0.1%
naive`initialize_matrices 12   0.1%
naive`multiply_matrices 8442  99.7%
a72293@solarisEdu:~/a72293$
```

A. FlameGraphs

Como se fez com a ferramenta PERF, gera-se também aqui um FlameGraph para o output do script DTrace. Uma adaptação do script hotuser descrito anteriormente é usada, também ela encontrada em [7]. Segue-se o uso do script de forma a gerar um ficheiro com o output para 'naive' e depois para 'interchanged':

```
a72293@solarisEdu:~$ dtrace -x ustckframes=100 -n 'profile-99
'/pid == $target && arg1/ { @[ustck()] = count(); } tick-60s
exit(0); }' -c ./naive -o out.stacks
dtrace: description 'profile-99' matched 2 probes
dtrace: pid 5244 has exited
a72293@solarisEdu:~$ dtrace -x ustckframes=100 -n 'profile-99
'/pid == $target && arg1/ { @[ustck()] = count(); } tick-60s {
exit(0); }' -c ./interchange -o outi.stacks
dtrace: description 'profile-99' matched 2 probes
dtrace: pid 5246 has exited
a72293@solarisEdu:~$
```

de seguida foi-se buscar a [7] um script escrito em perl, **stackcollapse.pl** de forma a interpretar o output obtido nos comandos anteriores. Depois volta-se á máquina 431 do cluster search6 (para posteriormente usar o mesmo flamegraph.pl usado para os flamegraphs desenvolvidos com o output de PERF no último capítulo) e utiliza-se os seguintes comandos:

Comandos Para 'naive'

```
./stackcollapse.pl out.stacks > out.folded
/share/jade/SOFT/FlameGraph/flamegraph.pl out.folded
```

> **dtrace_naive.svg**

que resultou no seguinte FlameGraph:

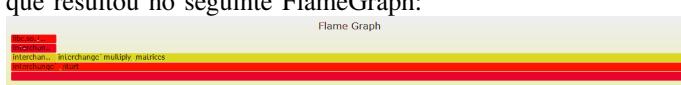


Comandos para 'interchange'

```
./stackcollapse.pl outi.stacks > outi.folded
/share/jade/SOFT/FlameGraph/flamegraph.pl outi.folded
```

> **dtrace_interchange.svg**

que resultou no seguinte FlameGraph:



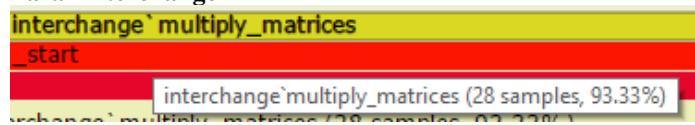
Mais uma vez estes gráficos podem ser observados no seu tamanho original ao consultar os ficheiros enviados com o documento.

Nestes gráficos nota-se **multiply_matrices** como sendo o **hot spot** do programa como se vê nas próximas imagens.

Para 'naive'



Para 'interchange'



Isto não contraria os flame graphs obtidos com PERF já que aí o FlameGraph relatava muito mais informação tendo assim feito uma análise menos específica o que levou a indicar como hot spot uma função onde a função que realmente é o hot spot foi chamada. S

V. CONCLUSÃO

Terminado o projeto pensa-se que o tutorial de PERF foi percebido com sucesso e que o foi possível acompanhar sem grande dificuldade. O maior problema que surgiu no uso de PERF foi por vezes alguns valores serem significativamente diferentes dos apresentados no tutorial tendo então sido necessário encontrar justificações para isso. Pensa-se que estas justificações resultaram de assunções corretas e que a sua explicação se encontra sucinta e clara ao longo do documento sendo assim possível afirmar que se desenvolveu de uma forma significativa o uso de PERF, podendo até agora ser capaz de o usar para fazer profiling de uma outra aplicação sem necessitar de re-ler o tutorial.

Uma outra dificuldade foi repetir o profiling com a ferramenta DTrace já que exigiu alguns conhecimentos a mais do que os scripts com DTrace que se foram fazendo nos trabalhos anteriores. No entanto pensa-se que se encontrou uma boa solução para o profiling com DTrace embora que funcione de forma bastante diferente de PERF e da opção desenvolvida por Brendan Gregg [7]

VI. BIBLIOGRAFIA

REFERÊNCIAS

- [1] P.J. Drongowski, *PERF tutorial* <http://sandsoftware.net/perf/perf-tutorial-hot-spots/>, Internet: accessed May 2017.
- [2] Internet: <http://www.oracle.com/technetwork/server-storage/solaris11/documentation/dtrace-cheatsheet-1930738.pdf>, accessed May 2017
- [3] Internet: <http://dtrace.org/blogs/brendan/2011/11/09/solaris-11-dtrace-syscall-provider-changes/>, accessed May 2017
- [4] Internet: cpu-world.com, accessed May 2017
- [5] Internet: ark.intel.com, accessed May 2017
- [6] Internet: http://search6.di.uminho.pt/wordpress/?page_id=55, accessed May 2017
- [7] Internet: <http://www.brendangregg.com/dtrace.html>, accessed Jun 2017

III. CONCLUSÃO

Terminado o portefólio pensa-se ser possível observar uma melhoria substancial ao longo dos projetos desenvolvidos mostrando assim que foi possível desenvolver capacidades de construção de documentos deste género que não tiveram oportunidade de se desenvolver no resto do percurso académico.

Os conceitos referidos ao longo dos projetos permitiram também alcançar um conhecimento muito mais profundo de computação paralela e distribuída e de análise de performance. Penso ser interessante a maneira como estes trabalhos levaram a ver de forma prática conceitos que anteriormente apenas conhecia na teoria e até mesmo aqueles que pensava conhecer bem revelaram-se mais complexos do que me havia apercebido anteriormente.

Em suma, sem dúvida que todos estes trabalhos e o estudo da bibliografia que os acompanhou permitiu que, de agora para a frente, seja capaz de fazer análise de desempenho de programas e máquinas de uma forma muito mais eficiente e instruída do que anteriormente.