ESC-TP3

José Pinto A81317 - Pedro Barbosa A82068

I. Introdução

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

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

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

II. Caracterização do ambiente de testes

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

```
psrinfo -pv
```

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

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

III. Descrição do algoritmo

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

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

IV. Algoritmo sequencial

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

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

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

```
provider histogram_equalization {

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

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

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

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

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

```
}

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

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

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

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

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

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

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

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

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

V. Algoritmo em memória partilhada

A. Descrição do algoritmo paralelo

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

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

B. Implementação OpenMP

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

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

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

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

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

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

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

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

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

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

C. Pthreads

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

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

    cumulative_distribution(thread_id);
    pthread_barrier_wait(&barrier);

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

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

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

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

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

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

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

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

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

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

pthread_barrier_wait(&barrier);
```

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

D. C++ 11

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

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

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

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

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

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

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

```
class Barrier {

public:
    Barrier(int thread_count);
```

```
    void wait(void);

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

    int thread_count;
    int current;
};

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

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

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

VI. Medições em memória partilhada

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

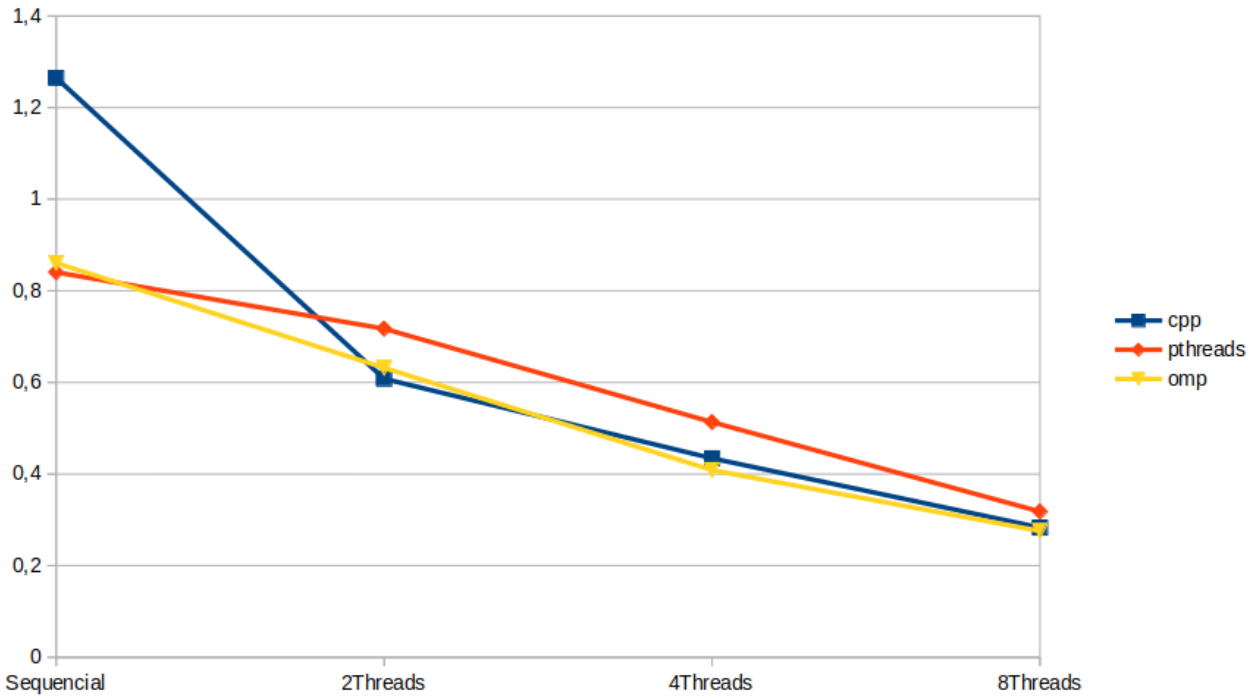

Fig. 1: Tempos de execução para as diferentes implementações

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

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

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

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

A. *Performance Counters*

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

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

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

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

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

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

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

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

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

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

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

B. Sincronização

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

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

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

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

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

Para tal, observamos as ocorrências dos seguintes eventos:

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

Também foram medidos os tempos despendidos nas seguintes fases

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

(Ficheiro plockstat_all_stats.d:)

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


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

```
    self->spinStart = 0;
}

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

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

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

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

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

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

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

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

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

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

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

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

(Ficheiro plockstat_thread_wait.d)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

C. Mapeamento das threads

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

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

(Ficheiro sched_detailed.d:)

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

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

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

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

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

    self->ts = 0;
}
```

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

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

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

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

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

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

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

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

(Ficheiro sched_per_cpu.d:)

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

}

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

    self->ts = 0;
}
```

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

```
Por cpu (unidades em milissegundos)

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

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

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

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

VII. Algoritmo MPI

A. Descrição do algoritmo MPI

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

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

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

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

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

B. Medições

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

(Ficheiro mpiProbes.d:)

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

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

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

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

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

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

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

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

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

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

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

```
}

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

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

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

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

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

TABLE VI: Tempos de execução para as diferentes primitivas

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

TABLE VII: Número de invocações das diferentes primitivas

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

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

VIII. Conclusão

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

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

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

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

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