Engenharia de Sistemas de Computação

TP3: DTrace - Análise e Avaliação de uma Aplicação C/C++

André Ramalho
a76387

Vítor Gomes
a75362

4 de Julho de 2020

1 Introdução

Neste relatório é apresentada uma análise a uma aplicação desenvolvida no âmbito da UC PCP. Esta análise é realizada sobre as versões sequencial, OpenMP e OpenMPI do algoritmo e utilizando a ferramenta DTrace.

Adicionalmente, apresentam-se outras soluções para uma implementação paralela desta aplicação sobre memória partilhada, utilizando a biblioteca PThreads e a norma C++11.

2 Descrição da Aplicação

A aplicação a estudar calcula a transformada de distância de uma dada imagem binária. Esta aplicação é constituída por 4 fases. Na 1ª fase é efetuado o carregamento da imagem fornecida. Na 2ª fase é executado o algoritmo que trata de calcular a transformada de distância da imagem. Na 3ª fase é efetuada a conversão dos valores resultantes da fase anterior para valores entre o intervalo $[0, 255]$ para obter uma matriz que será guardada como uma imagem em tons de cinzento com 8 bits na 4ª fase. A análise da aplicação será focada na 2ª e na 3ª fase. Estas são fases sobre as quais foi explorada paralelização em memória partilhada e distribuída.

O algoritmo executado na 2ª fase é composto por 4 passos para determinar a transformada de distância. No 1º passo, a matriz é percorrida por colunas, da linha menor para a maior, enquanto que, no 2º passo, a matriz é percorrida pela ordem inversa. No 3º passo, a matriz é iterada por linhas, da coluna menor para a maior e no 4º passo, a matriz é percorrida no sentido inverso.

3 Análise do Algoritmo Sequencial

Numa primeira analise à aplicação, criou-se um script em D que imprime informação que, previamente, era impressa pela aplicação, isto é, a duração de

diferentes etapas do algoritmo, assim como o maior valor da matriz da transformada de distância. Este script utiliza sondas USDT para localizar as diferentes etapas do algoritmo da 2ª fase e para obter o valor máximo da transformada de distância. Ainda são utilizadas sondas do provider `pid` para descobrir os momentos de entrada e saída de funções. Apesar de o procedimento de cálculo do valor máximo de uma matriz se encontrar dentro de uma função `encontraMax`, não foi possível utilizar sondas `pid` para obter o valor de retorno porque esta função desaparece com a flag de compilação `-O3`, criando a necessidade de criação de sondas USDT para envolver esta função.

Na figura 1 apresenta-se o output resultante deste script, chamado `prints.d`. Os valores de `Ti` referem-se ao passo `i` da 2ª fase da aplicação, onde se encontra o algoritmo principal, enquanto `Tdt` é o tempo total da 2ª fase. `TtransformaGS` é o tempo que demora a completar a 3ª fase do algoritmo.

```
a75362@solaris:~/TP3/TP1$ dtrace -s prints.d -c "./dt_sequencial Imagens/3920x2080.png "
Loaded image with a width of 3920px, a height of 2080px and 1 channels
T1: 0.111054
T2: 0.009651
T3+T4: 0.027936
0.148906
52
|---------- DTrace Output ----------
| T1 = 0.111124
| T2 = 0.009741
| T3+T4 = 0.027914
| Tdt = 0.148822
| Max=52
| TtransformaGS = 0.017816

a75362@solaris:~/TP3/TP1$ dtrace -s prints.d -c "./dt_sequencial Imagens/10000x6892.png "
Loaded image with a width of 10000px, a height of 6892px and 1 channels
T1: 0.825856
T2: 0.089695
T3+T4: 0.295118
1.211037
2495
|---------- DTrace Output ----------
| T1 = 0.825936
| T2 = 0.089828
| T3+T4 = 0.295071
| Tdt = 1.210895
| Max=2495
| TtransformaGS = 0.195647

a75362@solaris:~/TP3/TP1$ dtrace -s prints.d -c "./dt_sequencial Imagens/20000x13784.png "
Loaded image with a width of 20000px, a height of 13784px and 1 channels
T1: 34.978753
T2: 0.400344
T3+T4: 1.202860
36.582280
4990
|---------- DTrace Output ----------
| T1 = 34.978855
| T2 = 0.400428
| T3+T4 = 1.202823
| Tdt = 36.582162
| Max=4990
| TtransformaGS = 0.659207
```

Figura 1: Impressão de tempos de cada etapa

Analisando a figura 1, podemos ver que na segunda fase do algoritmo, o 1º passo possui a maior parte do tempo de execução.

Para fazer uma análise mais profunda, recorreu-se a sondas do provider `cpc` para criar o script `perf_counters.d`. Na figura 2 encontra-se o resultado da utilização deste provider para contar os milhões de instruções executadas em cada passo do algoritmo para uma imagem de tamanho 10000x6892. Nesta figura pode-se observar que a diferença no número de instruções executadas, por si só, pode justificar os diferentes tempos obtidos, exceto o tempo T1.

T2	296
transformaGS	313
T1	350
T3+T4	979

Figura 2: Milhões de instruções executadas por etapa na versão sequencial do algoritmo para uma imagem de tamanho 10000x6892

Utilizando ainda o provider `cpc`, verificamos a quantidade de acessos e misses à cache efetuada por cada etapa. Nas figuras 3, 4 e 5 apresentam-se os resultados obtidos, em dezenas de milhares, para uma imagem de tamanho 10000x6892.

transformaGS	5903
T2	10593
T1	16151
T3+T4	30074

Figura 3: Dezenas de milhares de acessos à cache L1 por etapa na versão sequencial do algoritmo para uma imagem de tamanho 10000x6892

T2	9
T3+T4	15
T1	147
transformaGS	239

Figura 4: Dezenas de milhares de misses à cache L1 por etapa na versão sequencial do algoritmo para uma imagem de tamanho 10000x6892

T2	9
T3+T4	13
T1	35
transformaGS	306

Figura 5: Dezenas de milhares de misses à cache L2 por etapa na versão sequencial do algoritmo para uma imagem de tamanho 10000x6892

Nestes resultados é possível observar que para `T1` ocorrem mais misses à cache L1 e L2, podendo ser a razão pela qual este passo demora mais tempo. Esta diferença pode ser devido ao facto de neste passo ser realizada uma iteração por colunas e serem acedidas 2 matrizes diferentes, uma com valores de 8 bits, a imagem original, e com valores de 4 bytes, para guardar as distancias. O passo `transformaGS` também utiliza estas 2 matrizes, enquanto os restantes passos necessitam apenas da matriz de 4 bytes.

4 Análise do Algoritmo Paralelo OpenMP

O programa implementado com OpenMP possui grandes falhas na paralelização dos passos 1 e 2 da 2ª fase da aplicação. Executando o algoritmo na máquina Solaris disponibilizada, o speedup atingido nestes passos é sempre menor que 1 para qualquer número de threads. Nesta secção, explora-se maioritariamente estes 2 passos problemáticos com a ferramenta DTrace. Nos resultados dos scripts, as zonas correspondentes a estes 2 passos identificam-se por `T1` e `T2`, enquanto `T3` identifica o passo 3 e 4. A execução das restantes fases do algoritmo fora destes 4 passos da 2ª fase será ignorada nesta secção porque estas não foram paralelizadas ou a sua paralelização é simples e curta.

Para observar o tempo de execução de cada passo, podemos voltar a utilzar o script `prints.d`. Nas seguintes figuras observam-se os tempos de execução de cada passo com 2, 4 e 8 threads para uma imagem de tamanho 10000x6892 quando executado na máquina Solaris. É possível observar o péssimo desempenho nos passos 1 e 2 à medida que o número de threads aumenta.

```
|---------- DTrace Output ----------
| T1 = 0.580646
| T2 = 0.172835
| T3+T4 = 0.144583
| Tdt = 0.898176
| Max=2495
| TtransformaGS = 0.075459
```

Figura 6: Tempos de execução com 2 fios de execução para uma imagem de tamanho 10000x6892

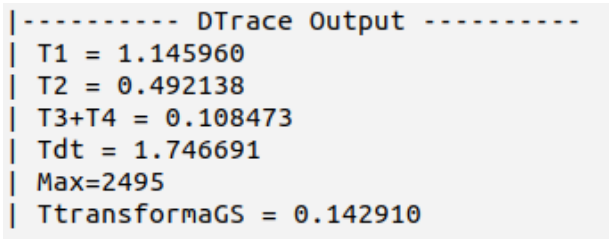

Figura 7: Tempos de execução com 4 fios de execução para uma imagem de tamanho 10000x6892

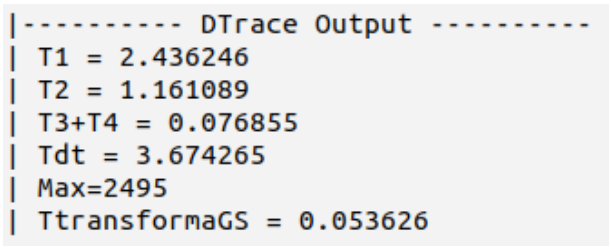

Figura 8: Tempos de execução com 8 fios de execução para uma imagem de tamanho 10000x6892

Para analisar a aplicação desenvolvida com OpenMP e a sua utilização de primitivas de sincronização foi explorado o provider `plockstat` para investigar a aquisição de mutexes duranto o processo. As seguintes figuras demonstram os resultados da execução de um script `plockstat.d` que imprime, utilizando agregações para cada secção do código, o número de vezes que alguma thread adquire um mutex, o número de vezes que terá de esperar por um mutex e o tipo de espera, e o número de iterações à espera de um mutex para o caso de uma espera ativa. Os resultados obtidos correspondem à utilização de 2, 4 e 8 fios de execução e uma imagem de tamanho 10000x6892. Nestas figuras é possível observar que existe um maior problema de sincronização nos passos 1 e 2 do algoritmo.

```
acquire
  T3                                                    3
  transformaGS                                          9
  Zona nao paralelizada                               322
  T2                                                 6898
  T1                                                 6937

block
  T2                                                    2
  T1                                                    4

spin
  T2                                                   62
  T1                                                   92

iterations
  T3                                                    0
  Zona nao paralelizada                                 0
  transformaGS                                          0
  T2                                                 1472
  T1                                                 2082
```

Figura 9: Informação sobre os mutexes adquiridos na execução com 2 fios de execução para uma imagem de tamanho 10000x6892

```
acquire
  T3                                                    5
  transformaGS                                         17
  Zona nao paralelizada                               324
  T2                                                13794
  T1                                                13860

block
  T1                                                    9
  T2                                                    9

spin
  T2                                                   96
  T1                                                  587

iterations
  T3                                                    0
  Zona nao paralelizada                                 0
  transformaGS                                          0
  T2                                                 6198
  T1                                                15333
```

Figura 10: Informação sobre os mutexes adquiridos na execução com 4 fios de execução para uma imagem de tamanho 10000x6892

```
acquire
  T3                                                    8
  transformaGS                                         34
  Zona nao paralelizada                               328
  T2                                                27590
  T1                                                27698

block
  T1                                                   14
  T2                                                   15

spin
  T3                                                    1
  transformaGS                                          4
  T2                                                   46
  T1                                                 1971

iterations
  Zona nao paralelizada                                 0
  T3                                                    1
  transformaGS                                        263
  T2                                                 1992
  T1                                                64516
```

Figura 11: Informação sobre os mutexes adquiridos na execução com 8 fios de execução para uma imagem de tamanho 10000x6892

Fazendo uso do provider `sched` e da sua sonda `sleep`, criou-se um script `sched.d` que regista a quantidade de vezes que cada fio de execução adormece para sincronização em cada etapa do código da seguinte forma:

```
sched:::sleep
/pid == $target/
{
    @sleep[curlwpsinfo->pr_lwpid,collect] = count();
}
```

onde `collect` identifica a fase em que o programa se encontra.

Nas seguintes figuras observam-se os resultados da execução do script. É possível observar que para as zonas T1 e T2, cada fio de execução adormece um número de vezes maior que metade dos pixeis de altura da imagem de entrada. Isto acontece devido ao facto de, nestes passos, se paralelizar um ciclo `for` interior, criando assim várias barreiras implícitas no fim de cada ciclo `for` paralelo:

```
for(x = bs; x <= height - bs; x += bs) {
    #pragma omp parallel for
    for (int y = 0; y < width; y++) {
        ...
    }
}
```

Esta paralelização poderia ter sido melhorada ao criar uma região paralela exterior a ambos os ciclos `for` e inserindo a cláusula `nowait` ao ciclo `for` a distribuir pelas threads.

```
Thread  1, Zona T3                              : 1
Thread  2, Zona T3                              : 1
Thread  1, Zona Zona nao paralelizada           : 2
Thread  2, Zona Zona nao paralelizada           : 2
Thread  2, Zona transformaGS                    : 3
Thread  1, Zona transformaGS                    : 4
Thread  2, Zona T2                              : 3447
Thread  1, Zona T2                              : 3451
Thread  1, Zona T1                              : 3454
Thread  2, Zona T1                              : 3455
```

Figura 12: Informação sobre a quantidade de vezes e o local em que as threads entram no estado sleep para sincronização com 2 fios de execução para uma imagem de tamanho 10000x6892

```
Thread  1, Zona T3                              : 1
Thread  2, Zona T3                              : 1
Thread  3, Zona T3                              : 1
Thread  3, Zona Zona nao paralelizada           : 1
Thread  1, Zona Zona nao paralelizada           : 2
Thread  2, Zona Zona nao paralelizada           : 2
Thread  4, Zona T3                              : 2
Thread  2, Zona transformaGS                    : 3
Thread  1, Zona transformaGS                    : 4
Thread  3, Zona transformaGS                    : 4
Thread  4, Zona transformaGS                    : 4
Thread  2, Zona T2                              : 3448
Thread  4, Zona T2                              : 3448
Thread  1, Zona T2                              : 3449
Thread  3, Zona T2                              : 3450
Thread  3, Zona T1                              : 3510
Thread  2, Zona T1                              : 3518
Thread  4, Zona T1                              : 3549
Thread  1, Zona T1                              : 3670
```

Figura 13: Informação sobre a quantidade de vezes e o local em que as threads entram no estado sleep para sincronização com 4 fios de execução para uma imagem de tamanho 10000x6892

Utilizando ainda o provider `sched`, elaborou-se o script `downtime.d` para tentar descobrir o tempo total que as threads passam fora de um CPU. Para o efeito foram definidas as 3 ações seguintes:

```
proc:::lwp-create
/pid==$target/
{
    tempo[curlwpsinfo->pr_addr]=timestamp;
    passo[curlwpsinfo->pr_addr]="";
}

sched:::on-cpu
/tempo[curlwpsinfo->pr_addr] && passo[curlwpsinfo->pr_addr]==collect/
{
    @downtime[collect] = llquantize((timestamp - tempo[curlwpsinfo->pr_addr])
                                    /1000,10,1,3,10);
}

sched:::off-cpu
/tempo[curlwpsinfo->pr_addr]/
{
    passo[curlwpsinfo->pr_addr] = collect;
    tempo[curlwpsinfo->pr_addr] = timestamp;
}
```

Na sonda `lwp-create` registam-se as threads que vão executar o algoritmo. Na sonda `off-cpu` registam-se o tempo e o passo em que as threads se encontravam antes de saírem do CPU. Na sonda `on-cpu` regista-se, num histograma, o tempo que a thread esteve fora do CPU para cada passo, caso esta thread ainda esteja a executar o mesmo passo.

Nas seguintes figuras observa-se que, para a zona T2, à medida que o número de threads é aumentado, maior é o tempo que as threads passam fora do CPU. Este aumento leva à diminuição do speedup obtido neste passo. O mesmo se observa para o 1^{o} passo.

```
T2
         value  ------------- Distribution ------------- count
          < 10 |                                         10
            10 |@@@@@@@                                  586
            20 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            2486
            30 |@                                        112
            40 |@                                        64
            50 |@                                        124
            60 |@                                        53
            70 |                                         6
            80 |                                         1
            90 |                                         0
           100 |                                         6
           200 |                                         0
           300 |                                         0
           400 |                                         0
           500 |                                         0
           600 |                                         0
           700 |                                         0
           800 |                                         0
           900 |                                         0
          1000 |                                         0
          2000 |                                         0
          3000 |                                         1
          4000 |                                         0
```

Figura 14: Tempo em microssegundos que as threads passam fora de um CPU para o passo 2 do algoritmo com 2 fios de execução para uma imagem de tamanho 10000x6892

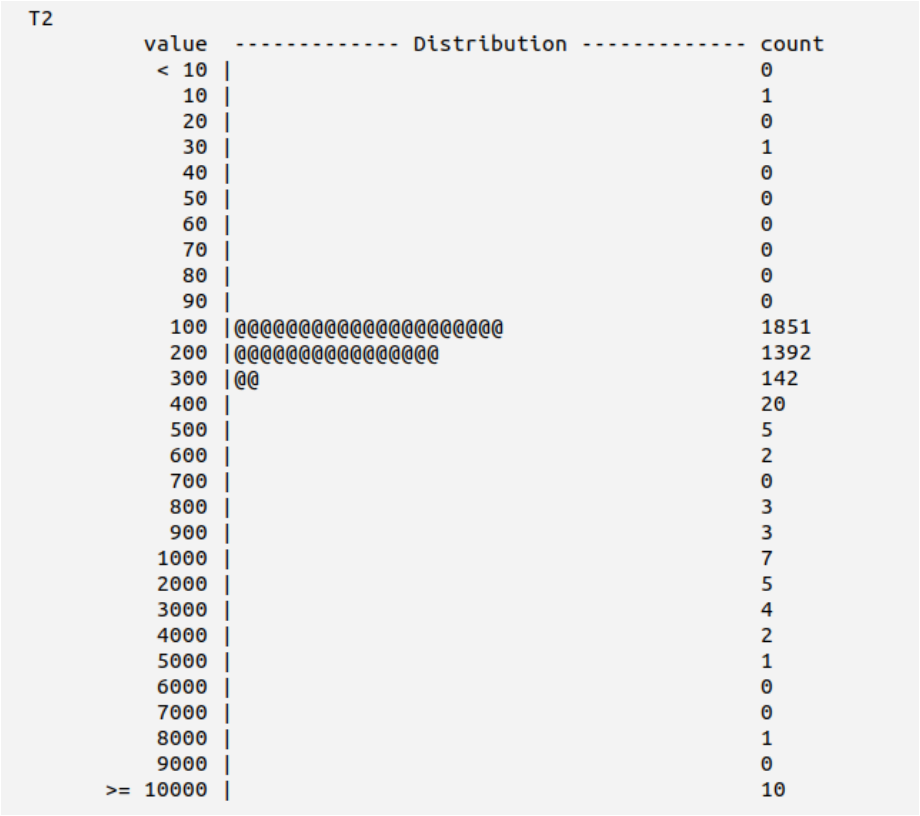

Figura 15: Tempo em microssegundos que as threads passam fora de um CPU para o passo 2 do algoritmo com 8 fios de execução para uma imagem de tamanho 10000x6892

Para observar o aproveitamento do cpu nas diferentes etapas do programa, criou-se um script, `usoCPU.d`, fazendo uso do provider `profile`. Utilizando a sonda `profile` com uma frequência de 997 Hertz, verificou-se que thread cada CPU estava a correr e em que estado se encontrava a thread caso esta pertencesse ao processo do nosso programa. A ação definida no script para armazenar esta informação num histograma é a seguinte:

```
profile-997
{
    estado = (pid==$target) ? curlwpsinfo->pr_state : 0;
    @name[curlwpsinfo->pr_sname] = count();
    @proc[collect] = lquantize(estado, 0, 10, 1);
}
```

Nas figuras seguintes observam-se os resultados da execução do script para 2 e 8 threads. Nos resultados, o valor 0 significa que a thread presente no CPU

não pertence ao processo que queremos observar, o valor de 1 significa que a thread está a dormir, o valor 2 significa que a thread está no estado runnable e o valor 6 significa que está a ser executada. Nas figuras observa-se que, para uma execução com 2 threads, os passos 1 e 2, já têm um mau aproveitamento do CPU quando comparados com os passos 3 e 4. Para uma execução com 8 threads, observa-se o grande aproveitamento que os passos 3 e 4 conseguem tirar do CPU, fazendo uso de todos os seus processadores, enquanto que os passos 1 e 2, não conseguem tirar nenhum aproveitamento adicional do CPU com o aumento do número de threads.

```
T2
           value  ------------- Distribution ------------- count
             < 0 |                                        0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   1219
               1 |                                        7
               2 |                                        6
               3 |                                        0
               4 |                                        0
               5 |                                        0
               6 |@@@@@@                                  224
               7 |                                        0

T3+T4
           value  ------------- Distribution ------------- count
             < 0 |                                        0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   878
               1 |                                        0
               2 |                                        0
               3 |                                        0
               4 |                                        0
               5 |                                        0
               6 |@@@@@@@@@@@@                            314
               7 |                                        0

T1
           value  ------------- Distribution ------------- count
             < 0 |                                        0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  31338
               1 |                                        23
               2 |                                        3
               3 |                                        0
               4 |                                        0
               5 |                                        0
               6 |@@@@@@                                  5708
               7 |                                        0
```

Figura 16: Estado em que se encontram as threads no CPU com 2 fios de execução para uma imagem de tamanho 10000x6892

```
T3+T4
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@                                      34
               1 |                                         0
               2 |                                         0
               3 |                                         0
               4 |                                         0
               5 |                                         0
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  374
               7 |                                         0

T2
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  8327
               1 |                                         42
               2 |                                         0
               3 |                                         0
               4 |                                         0
               5 |                                         0
               6 |@@@                                      688
               7 |                                         0

T1
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  32232
               1 |                                         70
               2 |                                         19
               3 |                                         0
               4 |                                         0
               5 |                                         0
               6 |@@@@@@                                   5910
               7 |                                         0
```

Figura 17: Estado em que se encontram as threads no CPU com 8 fios de execução para uma imagem de tamanho 10000x6892

5 Análise do algoritmo paralelo MPI

Para implementar esta aplicação em MPI, adotaram-se 2 estratégias de paralelização diferentes. Nesta secção faz-se uma análise à estratégia que produziu melhores resultados. Esta estratégia divide a matriz a processar na fase 2 da aplicação pelos processos existentes, de forma a obter uma matriz particionada em $P \times P$ blocos, onde P representa o número de processos utilizados. A matriz deve ser dividida da forma que se apresenta na figura 18, no caso de uma divisão por 4 processos.

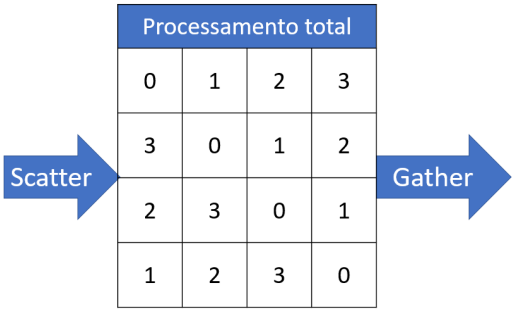

Figura 18: Estratégia de paralelização da 2ª fase da aplicação para 4 threads

A 2ª fase do algoritmo pode agora ser divida pelas fases de *scatter*, *dt* e *gather*. Na fase *scatter*, a matriz é dividida e enviada para os diferentes processos. Na fase *dt*, é efetuado o algoritmo principal do programa, envolvendo comunicação entre os processos utilizando as rotinas de MPI para receção de mensagens (`MPI_Recv()`) e envio de mensagens não bloqueantes (`MPI_Isend()`). Na fase *gather*, os fragmentos da matriz espalhados pelos processos são reunidos num só processo.

Para iniciar a análise deste aplicação em MPI, criou-se um script `tempos.d` para registar o tempo decorrido de cada processo nos passos *scatter*, *dt* e *gather*. Este script corre para cada processo, identificando as zonas em que cada processo se encontra pelas funções `MPI_Wtime()` existentes no código. Para que este script corra para cada processo do programa fazendo uso do provider `pid`, criou-se outro script, `trackProc.d`, que, através da sonda `create` do provider `proc`, executa um script Dtrace ligado a cada processo criado da seguinte forma:

```
proc:::create
/pid==$target/
{
    n_procs++;
    system( "dtrace -s %s -p %d %d &", script, args[0]->pr_pid, pid );
}
```

Assim, o script passado como argumento a `trackProc.d` será executado com o PID do processo pai no 1º parâmetro para cada processo filho. Esta solução falha quando o tamanho do problema é pequeno porque o processo pode executar as funções em que temos interesse, antes de o dtrace se ligar ao processo, resultando em sondas que ficam por ativar. Por exemplo, ao correr um problema com o tamanho 3920x2000, os processos atingem as funções `MPI_Wtime()` antes do DTrace iniciar o rastreio ao processo, disto resulta que as zonas em que se encontram os processos deixam de poder ser corretamente identificadas pois esta identificação depende da contagem de vezes que a sonda `entry` desta função foi ativada.

Nas seguintes figuras, exibe-se o resultado do script `trackProc.d` com o script `prints.d` como argumento. Nestas figuras observa-se que, no geral, não se

observam melhorias significativas no speedup com o aumento do nº de processos e com 8 processos o desempenho das 3 fases piora bastante.

```
PID=11380: Scatter=234ms; DT=469ms; Gather=87ms
PID=11381: Scatter=126ms; DT=580ms; Gather=27ms
```

Figura 19: Impressão de tempos de cada etapa na execução com 2 processos e um tamanho 10000x6892

```
PID=11397: Scatter=209ms; DT=566ms; Gather=73ms
PID=11400: Scatter=192ms; DT=535ms; Gather=58ms
PID=11398: Scatter=156ms; DT=569ms; Gather=53ms
PID=11399: Scatter=177ms; DT=546ms; Gather=56ms
```

Figura 20: Impressão de tempos de cada etapa na execução com 4 processos e um tamanho 10000x6892

```
PID=28037: Scatter=862ms; DT=1669ms; Gather=204ms
PID=28040: Scatter=969ms; DT=1562ms; Gather=203ms
PID=28038: Scatter=865ms; DT=1666ms; Gather=201ms
PID=28042: Scatter=973ms; DT=1558ms; Gather=207ms
PID=28046: Scatter=977ms; DT=1554ms; Gather=176ms
PID=28061: Scatter=998ms; DT=1507ms; Gather=208ms
PID=28036: Scatter=1005ms; DT=1502ms; Gather=219ms
PID=28039: Scatter=874ms; DT=1656ms; Gather=209ms
```

Figura 21: Impressão de tempos de cada etapa na execução com 8 processos e um tamanho 10000x6892

```
PID=12196: Scatter=1542ms; DT=2939ms; Gather=362ms
PID=12197: Scatter=997ms; DT=3494ms; Gather=118ms
```

Figura 22: Impressão de tempos de cada etapa na execução com 2 processos e um tamanho 20000x13784

```
PID=12297: Scatter=832ms; DT=2374ms; Gather=150ms
PID=12294: Scatter=900ms; DT=2127ms; Gather=243ms
PID=12296: Scatter=746ms; DT=2309ms; Gather=152ms
PID=12295: Scatter=680ms; DT=2369ms; Gather=150ms
```

Figura 23: Impressão de tempos de cada etapa na execução com 4 processos e um tamanho 20000x13784

```
PID=12237: Scatter=3267ms; DT=4060ms; Gather=1265ms
PID=12232: Scatter=3370ms; DT=3937ms; Gather=1473ms
PID=12235: Scatter=3184ms; DT=3951ms; Gather=1219ms
PID=12233: Scatter=3042ms; DT=4390ms; Gather=1194ms
PID=12234: Scatter=3020ms; DT=4047ms; Gather=1246ms
PID=12239: Scatter=3347ms; DT=4048ms; Gather=1231ms
PID=12238: Scatter=3295ms; DT=4029ms; Gather=1123ms
PID=12236: Scatter=3231ms; DT=3903ms; Gather=1179ms
```

Figura 24: Impressão de tempos de cada etapa na execução com 8 processos e um tamanho 20000x13784

Com o aumento do nº de processos, o número de comunicações também aumenta na fase `dt`. O número de comunicações nesta fase é igual a $4 \times (P^2 - P)$. Este aumento do número de comunicações é uma das causas para a falha na escalabilidade da aplicação. Para analisar o tempo total que cada processo fica à espera de dados de outro processo, criou-se o script `waitingRecv.d` que regista a soma das diferenças entre as entradas e saídas da função `MPI_Recv()`. Nas figuras seguintes observam-se resultados da execução deste script para diferentes números de processos. Para 2 processos, o tempo de comunicação é bastante pequeno. Para 4 processos, o tempo de comunicação aumenta, mas o tempo de processamento restante do passo `dt` diminui, o que mitiga os efeitos do aumento de comunicação. Para 8 processos, o tempo de comunicação aumenta bastante e representa a maior parte do tempo do passo `dt`.

```
PID=13459: Scatter=145ms; DT=628ms; Gather=28ms
Tempo bloqueado em MPI_Recv(): 1ms

PID=13458: Scatter=258ms; DT=513ms; Gather=88ms
Tempo bloqueado em MPI_Recv(): 58ms
```

Figura 25: Impressão de tempos bloqueados à espera de dados na execução com 2 processos e um tamanho 10000x6892

```
PID=13229: Scatter=202ms; DT=482ms; Gather=32ms
Tempo bloqueado em MPI_Recv(): 120ms

PID=13228: Scatter=249ms; DT=483ms; Gather=54ms
Tempo bloqueado em MPI_Recv(): 290ms

PID=13231: Scatter=231ms; DT=461ms; Gather=36ms
Tempo bloqueado em MPI_Recv(): 100ms

PID=13230: Scatter=217ms; DT=477ms; Gather=34ms
Tempo bloqueado em MPI_Recv(): 122ms
```

Figura 26: Impressão de tempos bloqueados à espera de dados na execução com 4 processos e um tamanho 10000x6892

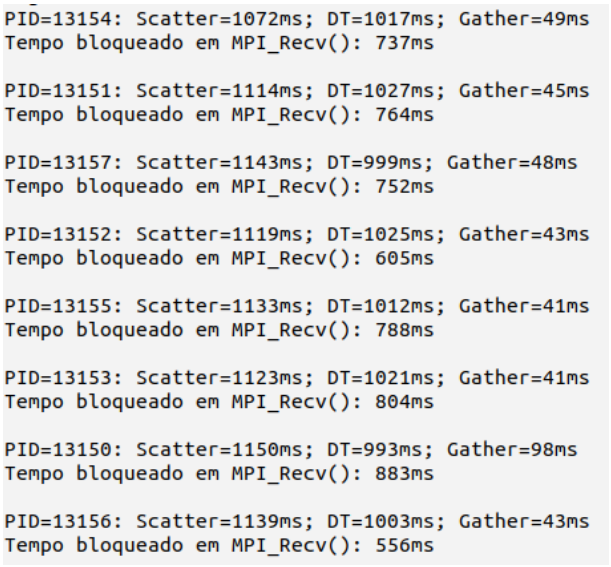

Figura 27: Impressão de tempos bloqueados à espera de dados na execução com 8 processos e um tamanho 10000x6892

Para observar a utilização do CPU pela aplicação, criou-se o script `usoCPU.d` que verifica, para cada CPU, se o processo a correr pertence à aplicação. Esta identificação é feita observando se o pid do pai do processo que ativa a sonda `profile-997` é igual ao pid passado como argumento no script `trackProc.d`. Neste caso, o processo pertence à nossa aplicação.

```
profile-997
{
    estado = ($1 == ppid) ? curlwpsinfo->pr_state : 0;
    @proc[collect] = lquantize(estado,0,10,1);
}
```

Nas figuras seguintes demonstra-se o aproveitamento do CPU para diferentes números de processos.

```
Zona dt                                :
             value  ------------- Distribution ------------- count
               < 0 |                                         0
                 0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@           4300
                 1 |                                         0
                 2 |                                         1
                 3 |                                         0
                 4 |                                         0
                 5 |                                         0
                 6 |@@@@@@@@@@                               1431
                 7 |                                         0

Zona Zona sequencial                   :
             value  ------------- Distribution ------------- count
               < 0 |                                         0
                 0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    39166
                 1 |                                         8
                 2 |                                         0
                 3 |                                         0
                 4 |                                         0
                 5 |                                         0
                 6 |@@@@                                     4146
                 7 |                                         0
```

Figura 28: Distribuição do uso do CPU na zona dt e zona sequencial na execução com 2 processos e um tamanho 10000x6892

```
Zona dt                                :
             value  ------------- Distribution ------------- count
               < 0 |                                         0
                 0 |@@@@@@@@@@@@@@@@@@@@@                    2926
                 1 |                                         0
                 2 |                                         1
                 3 |                                         0
                 4 |                                         0
                 5 |                                         0
                 6 |@@@@@@@@@@@@@@@@@@@                      2745
                 7 |                                         0

Zona Zona sequencial                   :
             value  ------------- Distribution ------------- count
               < 0 |                                         0
                 0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    64077
                 1 |                                         24
                 2 |                                         0
                 3 |                                         0
                 4 |                                         7
                 5 |                                         0
                 6 |@@@@                                     6665
                 7 |                                         0
```

Figura 29: Distribuição do uso do CPU na zona dt e zona sequencial na execução com 4 processos e um tamanho 10000x6892

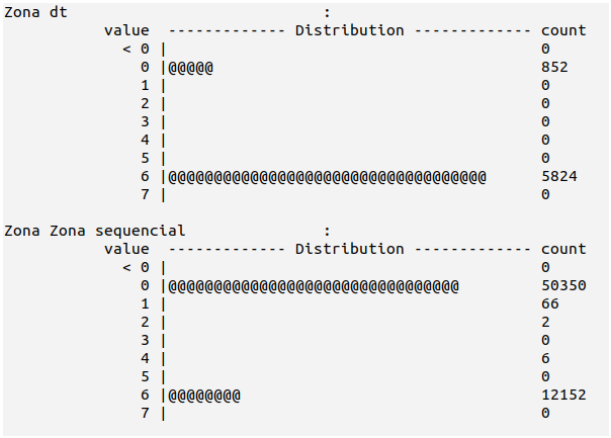

Figura 30: Distribuição do uso do CPU na zona dt e zona sequencial na execução com 8 processos e um tamanho 10000x6892

6 Paralelização e análise em PThreads e C++11

Nesta secção são apresentadas e analisadas implementações paralelas em memória partilhada alternativas, utilizando a biblioteca PThreads e a norma C++11.

6.1 PThreads

Na implementação desta aplicação utilizando a biblioteca PThreads é definido explicitamente um domínio da matriz sobre o qual cada thread deve operar. Esta definição faz com que um dos maiores problemas que impedia a escalabilidade do algoritmo em OpenMP desapareça pois deixarão de existir barreiras dentro dos passos 1 e 2 da 2ª fase da aplicação. Esta 2ª fase passa agora a possuir apenas uma barreira, que se encontra entre o 2º e 3º passo.

Nas figuras seguintes podem-se observar os resultados do script `prints.d` para execuções com 2, 4 e 8 threads e uma matriz de tamanho 10000x6892. Nestes resultados observa-se que foi possível obter um bom speedup para os passo 1 e 2, o que não se verificava implementação anterior.

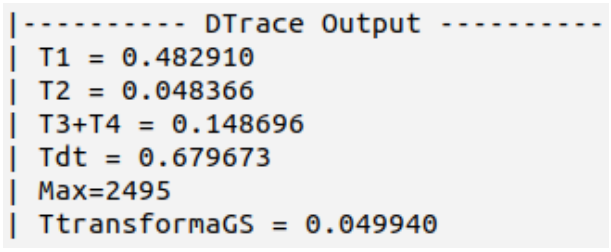

Figura 31: Impressão dos tempos obtidos em cada passo para 2 threads e um tamanho 10000x6892

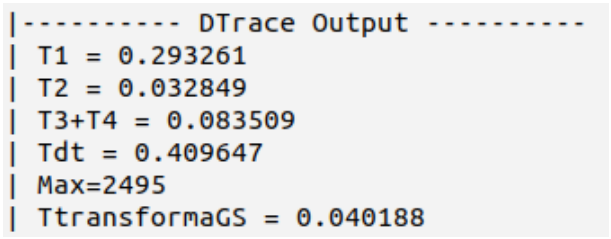

Figura 32: Impressão dos tempos obtidos em cada passo para 4 threads e um tamanho 10000x6892

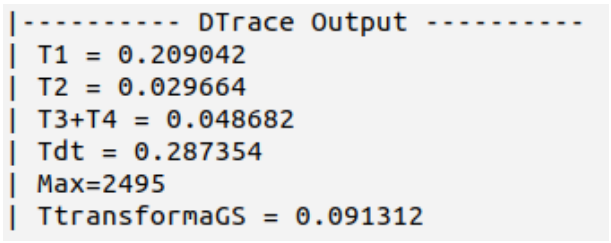

Figura 33: Impressão dos tempos obtidos em cada passo para 8 threads e um tamanho 10000x6892

Utilizando o script `usoCPU.d`, podemos observar bem o uso completo do CPU nestes 2 passos que anteriormente viam pouco uso do CPU. Na figura seguinte demonstra-se o uso do CPU com 2 e 8 threads para o passo 1, onde 0 corresponde a um processador utilizado para outras aplicações e 6 corresponde a um processador a executar uma das threads do nosso processo.

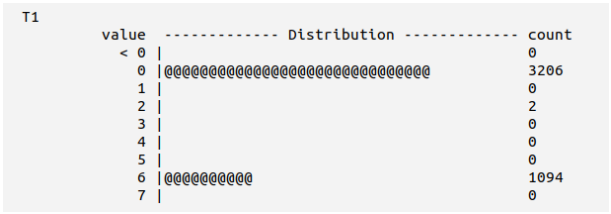

Figura 34: Uso do CPU no passo 1 para 2 threads e um tamanho 10000x6892

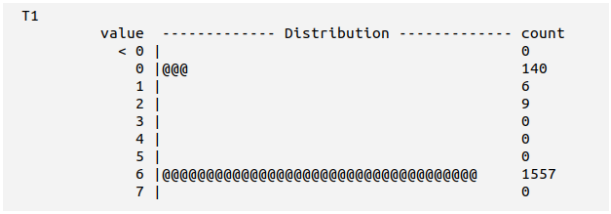

Figura 35: Uso do CPU no passo 1 para 8 threads e um tamanho 10000x6892

Utilizando o script `downtime.d`, podemos ver que o número de vezes que uma thread sai do CPU e por quanto tempo. Na figura 36 observa-se que para a imagem de tamanho 10000x6892 e uma execução com 8 threads, o processo raramente sai do CPU quando este se encontra no passo 1.

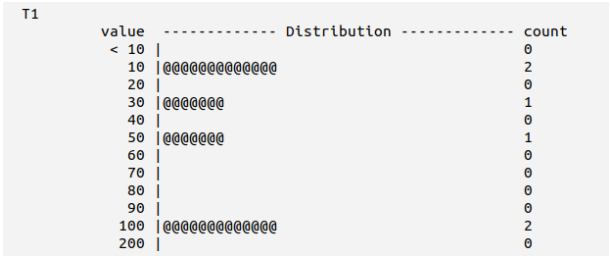

Figura 36: Tempo em microssegundos que as threads passam fora de um CPU no passo 1 para 8 threads e um tamanho 10000x6892

Correndo o script `plockstat.d`, observa-se que praticamente deixam de existir problemas de sincronização quando comparado com a versão OpenMP.

```
acquire
  T3+T4                                                    11
  T1                                                       18
  T2                                                       24
  transformaGS                                             47
  Zona nao paralelizada                                   657

block
  T1                                                        2
  T2                                                        2
  Zona nao paralelizada                                     5

spin
  T1                                                        2
  T3+T4                                                     3
  T2                                                        6
  Zona nao paralelizada                                     8

iterations
  T1                                                        0
  transformaGS                                              0
  Zona nao paralelizada                                     3
  T2                                                      198
  T3+T4                                                  2001
```

Figura 37: Informação sobre mutexes adquiridos na execução com 8 threads e um tamanho 10000x6892

Correndo o script `sched.d` que indica o número de vezes que o cada thread ativou a sonda `sleep` observa-se que ainda existem alguns problemas de desempenho no passo 1, como também se observa pelo seu tempo de execução.

```
Thread  1, Zona transformaGS                 : 1
Thread  2, Zona T2                           : 1
Thread  3, Zona T2                           : 1
Thread  4, Zona T2                           : 1
Thread  5, Zona T2                           : 1
Thread  6, Zona T2                           : 1
Thread  7, Zona T2                           : 1
Thread  8, Zona T2                           : 1
Thread  9, Zona T2                           : 1
Thread  1, Zona T1                           : 7
Thread  1, Zona Zona nao paralelizada        : 19
Thread  9, Zona T1                           : 375
Thread  5, Zona T1                           : 524
Thread  6, Zona T1                           : 551
Thread  7, Zona T1                           : 551
Thread  8, Zona T1                           : 597
Thread  4, Zona T1                           : 615
Thread  3, Zona T1                           : 624
Thread  2, Zona T1                           : 768
```

Figura 38: Informação sobre a quantidade de vezes e o local que as threads entram no estado sleep com 8 threads e um tamanho 10000x6892

6.2 C++11

A versão paralela do programa implementada em C++ seguiu a estratégia da versão em PThreads e conseguiu obter tempos de execução ainda melhores que esta na máquina Solaris, principalmente devido à melhoria de desempenho obtido no passo 1. Nas figuras seguintes observam-se os tempos obtidos para cada passo para a matriz 10000x6892 com 2, 4 e 8 threads.

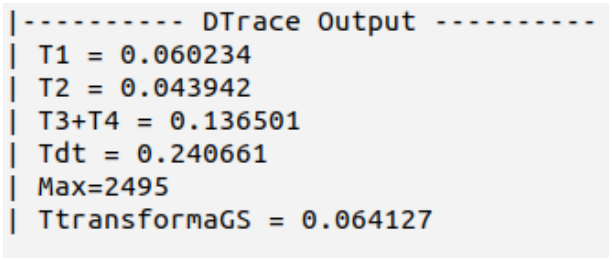

Figura 39: Impressão dos tempos obtidos em cada passo para 2 threads e um tamanho 10000x6892

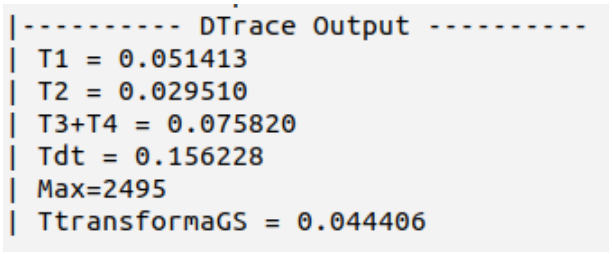

Figura 40: Impressão dos tempos obtidos em cada passo para 4 threads e um tamanho 10000x6892

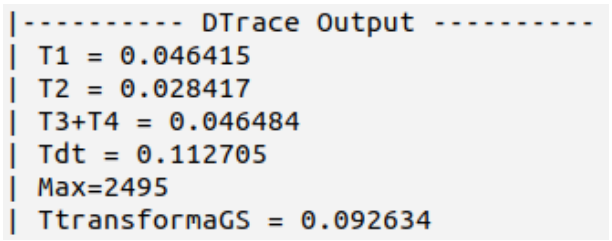

Figura 41: Impressão dos tempos obtidos em cada passo para 8 threads e um tamanho 10000x6892

Utilizando o script `usoCPU.d` observa-se também o aproveitamento total do CPU na figura seguinte, onde 0 indica que um CPU executou outro processo qualquer e 6 indica que um CPU executou o nosso processo.

```
T1
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@                                       12
               1 |                                         0
               2 |                                         0
               3 |                                         0
               4 |                                         0
               5 |                                         0
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   266
               7 |                                         0

T3+T4
           value  ------------- Distribution ------------- count
               5 |                                         0
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 271
               7 |                                         0

T2
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@                                      22
               1 |                                         0
               2 |                                         0
               3 |                                         0
               4 |                                         0
               5 |                                         0
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    274
               7 |                                         0
```

Figura 42: Utilização do CPU para 8 threads e um tamanho 10000x6892

Executando o script `sched.d` observa-se uma redução elevada do número de vezes que a sonda `sleep` do provider `sched` foi ativada no passo 1 quando comparada com a versão PThreads, levando ao aumento de desempenho deste passo para esta versão implementada em C++.

```
Thread  1, Zona T1                               : 1
Thread  1, Zona T3+T4                            : 1
Thread  2, Zona T2                               : 1
Thread  5, Zona T2                               : 1
Thread  9, Zona T2                               : 1
Thread 18, Zona transformaGS                     : 1
Thread 21, Zona transformaGS                     : 1
Thread  4, Zona T2                               : 2
Thread  1, Zona T2                               : 5
Thread  1, Zona Zona nao paralelizada            : 6
Thread  1, Zona transformaGS                     : 13
```

Figura 43: Informação sobre a quantidade de vezes e o local que as threads entram no estado sleep com 8 threads e um tamanho 10000x6892

7 Conclusão

Neste trabalho procurou-se analisar uma aplicação por nós desenvolvida com a intenção de identificar os problemas de desempenho que esta possa ter e como ultrapassar estes problemas. A aplicação analisada possuía uma versão sequencial, uma em OpenMP e outra em MPI. As três versões desta aplicação foram investigadas e para efetuar esta análise, fez-se uso da ferramenta DTrace e explorou-se diferentes providers existentes para analisar a aplicação de várias perspetivas.

Após a análise desta aplicação, foram desenvolvidas mais 2 versões paralelas em memória distribuída desta aplicação. Implementou-se uma versão em PThreads e outra em C++11. Com os resultados da análise das versões implementadas anteriormente, também se procurou que estas versões fossem melhor implementadas, superando as versões anteriores. Após a implementação destas versões, analisaram-se e compararam-se as diferentes versões recorrendo aos scripts de DTrace já desenvolvidos anteriormente.