University of Minho
School of Engineering

Dtrace

Utilização de *Dtrace* para a análise de *radix sort*

Hugo Afonso Da Gião
PG41073

22 de Junho de 2020

Resumo

Este trabalho consiste no teste e análise de diferentes implementações da variante *MSD* do algoritmo *radix sort* utilizando a ferramenta textitDtrace.

Palavras-chave:*Dtrace*,*Radix Sort*,*dynamic tracing*,*MPI*,*OpenMP*,*Pthreads*,*C++11*

Conteúdo

Capítulo 1

Introdução

Este documento relata o trabalho realizado para a análise de implementações sequenciais, paralelas de memória distribuída e partilhada do algoritmo *radix sort MSD* utilizando a ferramenta *Dtrace*.

O *Dtrace* é uma ferramenta de traçado dinâmico desenvolvida pela *Sun Microsystems*,esta ferramenta foi criada com intuito de ser utilizada para diagnosticar problemas relacionados com as aplicações e *kernel* em tempo real. Relativamente a outras ferramentas de traçado como por exemplo o *strace* esta tem a vantagem de poder ser utilizada em produção.

Este documento está organizado em vários capítulos o primeiro capítulo descreve os algoritmos e algumas opções tomadas nas várias implementações no capítulo subsequente são apresentadas as *probes* criadas e utilizadas na análise dos mesmos, posteriormente está o capítulo contendo os resultados e alguma discussão e análise dos mesmos, posteriormente são apresentadas algumas conclusões relativas ao trabalho realizado.

Capítulo 2

Algoritmo utilizado

2.1 Versão sequencial

O algoritmo utilizado neste caso de estudo consiste no *radix sort MSD*.A implementação utilizada ordena um *array* dividindo-o em *buckets* de acordo com os seus n *bits* mais significativos, nesta implementação $n = 128$.

Após a sua ordenação inicial em *buckets* esta função é chamada recursivamente para cada um dos *buckets* com os n *bits* mais significativos seguintes, até ao caso de paragem em que o *array* a ser ordenado tem tamanho inferior ou igual a 1 ou não existirem mais *bits* para ordenar.

Algorithm 1 Radix Sort sequencial with fixed number of buckets

Input:array of integers array,integer size,integer digit.
Output:none

1: **for** $i \in [0, size[$ **do**
2: $digit \leftarrow get\_digit(array[i], 0)$
3: $count[digit] + +$
4: **end for**
5: **for** $i \in [0, size[$ **do**
6: $digit \leftarrow get\_digit(array[i], 0)$
7: $temp[count[digit] + inserted[digit] + +] \leftarrow array[i]$
8: **end for**
9: **for** $i \in [0, size[$ **do**
10: $array[i] \leftarrow temp[i]$
11: **end for**
12: **for** $i \in [1, NR\_BUCKETS[$ **do**
13: $start[i] \leftarrow start[i-1] + count[i]$
14: **end for**
15: **for** $i \in [0, size[$ **do**
16: $radix\_sort(array[start[i] :], count[i], digit + 1)$
17: **end for**

2.2 Versões paralelas

2.2.1 Versão em memória partilhada *Omp*

A estratégia utilizada nesta implementação consistiu em após a primeira iteração ordenar cada um dos *buckets* em paralelo. Esta implementação também realiza outras secções do algoritmo como o cálculo do tamanho de cada *bucket*, a ordenação em *buckets* e a copia do *array* temporário para o

array principal em paralelo.

2.2.2 Versão em memória distribuída *MPI*

O algoritmo de base para a implementação em memória distribuída consiste em inicialmente realizar um primeira ordenação em *buckets* de acordo com os primeiros *n bits* mais significativos pelo processo *master*.Após ter realizado esta ordenação este processo divide a carga pelos processos *slave* e por si mesmo de modo a esta ser distribuída o mais equilibradamente possível, para tal este algoritmo utiliza um algoritmo de distribuição de carga que ordena os **buckets** da forma "$highest\ size\ bucket, lowest\ size\ bucket, second\ highest\ size\ bucket, second\ lowest\ size\ bucket, ...$",o que permite mitigar problemas de distribuição de carga ocorrentes no caso de um processo possuir um número significativo de *buckets* maiores.Esta implementação utiliza as primitivas *scatterv* e *gatterv* para distribuir os *buckets* pelos restantes processos e a primitiva *broadcast* para partilhar os tamanhos e os *buckets* a serem ordenados por cada um dos processos.

2.2.3 Versão em memória partilhada *Pthreads*

Esta implementação é semelhante a implementação *Omp*,a principal diferença relativamente a esta consiste na distribuição dos *buckets* ordenados em paralelo ser feito utilizando o algoritmo de divisão de carga utilizado para a versão *MPI*.As restantes diferenças comparativamente com a versão *Omp* consistem no uso de primitivas da biblioteca *Pthreads* em vez das de *Omp*.O número de *threads* é também passado como argumento ao programa em vez de ser controlado por uma variável de ambiente.

2.2.4 Versão em memória partilhada *C++11*

Para utilizar mais funcionalidades do *C++* em vez de utilizar um *array* auxiliar optei por utilizar a primitiva *vector* de *c++*, a utilização desta primitiva permite utilizar menos recursos relativamente a computação dos *digitos* sendo que apenas é necessário percorrer o *array* uma vez por cada chamada a função em vez de duas uma para contar e outra para inserções e esta primitiva preserva alguma eficiência nos acessos a memória sendo que as suas inserções tem um peso similar a inserções num *array*.As restantes diferenças comparativamente a implementação *Pthreads* consistem em adaptações para o uso da primitiva *Thread* do *C++* em vez das primitivas *Pthreads*.

Capítulo 3

Probes utilizadas

3.1 Custom probes

3.1.1 Sequencial

Probe	**argumentos**	**descrição**
start_sorting_into_buckets *finish_sorting_into_buckets*	(int, int)	Indicam o inicio e fim da ordenação dos elementos de um *array* em *buckets* de acordo com os seus $n * d - n * (d + 1)$ *bits*. O argumento arg_0 corresponde ao tamanho do *array* e arg_1 a d .
start_seq_radix *finish_seq_radix*	(int, int)	Indicam o inicio e fim de uma execução da função de *radix sort* sequencial. O argumento arg_0 indica o tamanho do *array* e arg_1 os *bits* pelos quais o *array* esta a ser ordenado.
start_count_digits *finish_count_digits*	(int, int)	Indicam o inicio e fim das contagens de cada ocorrência dos bits pelos quais o *array* esta a ser ordenado. Os argumento arg_0 representa o tamanho do *array* e arg_1 esses *bits*.
start_insert_into_buckets *finish_insert_into_buckets*	(int, int)	Indica o fim e o inicio da inserção nos *buckets*. O argumento arg_0 indica o tamanho do *array* e arg_1 os *bits* pelos quais o *array* esta a ser ordenado.
start_copy_to_main_array *finish_copy_to_main_array*	(int, int)	Indicam o fim e o inicio da copia do *array* temporário para o *main array*. O argumento arg_0 indica o tamanho do *array* e arg_1 os *bits* pelos quais o *array* esta a ser ordenado.
start_allocate_temp_array *finish_allocate_temp_array*	(int, int)	Indicam o inicio e fim da alocação do array temporário. O argumento arg_0 indica o tamanho do *array* e arg_1 os *bits* pelos quais o *array* esta a ser ordenado.

Tabela 3.1: *Custom probes* definidas para a versão sequencial

```
time spent sorting into buckets

                3                3
                1                3
                2                3
array sizes per digit
```

```
                1
      value  ------------- Distribution ------------- count
    4194304 |                                         0
    8388608 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 5
   16777216 |                                         0

                2
      value  ------------- Distribution ------------- count
    2097152 |                                         0
    4194304 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
    8388608 |                                         0

                3
      value  ------------- Distribution ------------- count
      16384 |                                         0
      32768 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1280
      65536 |                                         0

time spent per digit

                4                0
                3                4
                2                8
                1               12
calls by digit

                1                5
                2              640
                3             1280
                4           163840
time spent in different sections of the program

  allocate temp array                                    0
  copy to main array                                     0
  sorting into buckets                                  10
  total                                                 38
```

Listing 3.1: Output das *probes* sequenciais para a implementação sequencial utilizando tamanho 10000000

O uso destas *probes* consiste maioritariamente em examinar o tempo gasto em várias secções do programa para tal foram criadas várias sondas que permitem obter quando o programa entra e sai de determinadas secções. Tal como informação acerca dos tamanhos dos *arrays* a serem ordenados e o "digito"a ser ordenado.

O resultado destas *probes* consiste no tempo gasto na secção de ordenação dos elementos em *buckets* em segundos, a distribuição dos tamanhos dos *arrays* pelos dígitos a serem ordenados, o tempo gasto na ordenação de cada dígito incluindo as chamadas recursivas e as chamadas para a ordenação de cada "digito".

3.1.2 OpenMp

Para além das *probes* definidas para a versão sequencial que foram adaptadas para medir não apenas as chamadas recursivas mas também as ocorrências em todas as secções do algoritmo e devolver informação relativamente ao *bucket* paralelo ou secção sequencial a ser ordenado ,a implementação *OpenMp* possui também as seguintes *probes*.

Probe	**argumentos**	**descrição**
start_par_radix *finish_par_radix*	(int, int, int)	Indicam o inicio e fim da execução do função paralela. O argumento arg_0 indica o tamanho do *array* ,o arg_1 os *bits* pelos quais o *array* esta a ser ordenado e arg_2 devolve o *bucket*.

Tabela 3.2: Custom probes definidas para a versão OpenMp

```
time spent sorting into buckets

                1                1
                3                1
                2                1
array sizes per digit

               1
           value  ------------- Distribution ------------- count
         2097152 |                                         0
         4194304 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
         8388608 |                                         0

               2
           value  ------------- Distribution ------------- count
         1048576 |                                         0
         2097152 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
         4194304 |                                         0

               3
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1280
           32768 |                                         0

time spent per digit

calls by digit

                2              640
                3             1280
                4           163840
time spent in different sections of the program

  allocate temp array                                                0
  copy to main array                                                 0
  total                                                              3
  sorting into buckets                                               4
```

Listing 3.2: Output das probes Omp para a implementação sequencial utilizando tamanho 5000000

Tal como para as *probes USDT* criadas para a versão sequencial a função principal destas *probes* consiste em medir o tempo gasto em diferentes secções do programa sejam estas do componentes do algoritmo ou chamadas recursivas para a ordenação de um digito.

O resultado do *script* criado utilizando estas *probes* é o mesmo que na versão anterior.

3.1.3 Mpi

Para a versão *Mpi* para além das *probes* definidas para as versões *Omp* para as quais o terceiro argumento passou a ser o processo em vez do *bucket* ,foram definidas as seguintes *probes*.

Probe	**argumentos**	**descrição**
start_ workload_ distribution *finish_ workload_ distribution*	(int, int, int)	Indicam o inicio e fim do processo de distribuição do trabalho pelos vários processos pelo processo *Master*.O argumento arg_0 indica o tamanho do *array*,o arg_1 os *bits* pelos quais o *array* esta a ser ordenado e o arg_2 corresponde ao *rank* do processo.
start_ scatter_ workload *finish_ scatter_ workload*	(int, int, int)	Indicam o inicio e fim do envio das porções do *array* a serem ordenados pelos *Slaves* por parte do processo *Master*.O argumento arg_0 indica o tamanho do *array* ,o arg_1 os *bits* pelos quais o *array* esta a ser ordenado e o arg_2 indica o *rank* do processo.
start_ gather_ workload *finish_ gather_ workload*	(int, int, int)	Indicam o inicio e fim da receção das porções do *array* a serem *ordenadas* pelos processos *Slave* por par parte do *Master*.O argumento arg_0 indica o tamanho do *array* ,o arg_1 os *bits* pelos quais o *array* esta a ser ordenado e o arg_2 o *rank* do processo.
start_ receive_ workload *finish_ receive_ workload*	*int*	Indicam o inicio e fim das receções das porções do *array* a serem ordenadas pelos processos *Slave* do processo *Master*.O argumento arg_0 refere-se ao *rank* do processo.
start_ send_ workload *finish_ send_ workload*	*int*	Indicam o inicio e fim do envio das porções do *array* ordenadas pelos processos *Slave* ao processo *Master*.O argumento arg_0 representa o *rank* do processo.

Tabela 3.3: Custom probes definidas para a versão Mpi

```
time spent sorting into buckets

                3                1
                1                1
                2                1
array sizes per digit

                1
           value  ------------- Distribution ------------- count
         2097152 |                                         0
         4194304 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 5
         8388608 |                                         0

                2
           value  ------------- Distribution ------------- count
         1048576 |                                         0
         2097152 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
         4194304 |                                         0

                3
           value  ------------- Distribution ------------- count
```

```
             8192 |                                                    0
            16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1280
            32768 |                                                    0

time spent per digit

                2                3
calls by digit

                2              640
                3             1280
                4           163840
time spent in different sections of the program

  workload distribution                                                  0
  allocate temp array                                                    0
  scatter workload                                                       0
  gather workload                                                        0
  copy to main array                                                     0
  recursive calls                                                        3
  sorting into buckets                                                   4
  total                                                                  6
```

Listing 3.3: Output das *probes MPI* para a implementação *MPI* utilizando tamanho 5000000 e 1 *thread*

Tal como para as *probes USDT* criadas para as versões anteriores a função principal destas *probes* consiste em medir o tempo gasto em diferentes secções partes do programa sejam estas secções do algoritmo ou chamadas para a ordenação de um digito.

O resultado do *script* criado utilizando estas *probes* é o mesmo que nas versões anteriores.

3.1.4 Pthreads

As *probes USDT* definidas para esta implementação consitem em todas as *probes* utilizadas para a versão *Omp* mais as *probes start_ workload_ distribution* e *finish_ workload_ distribution* sendo que neste caso o seu terceiro argumento consiste no *bucket* da primeira chamada recursiva a ser ordenado e não o processo correspondente.

```
time spent sorting into buckets

                3                0
                2                0
                1                3
array sizes per digit

                1
           value  ------------- Distribution ------------- count
          262144 |                                         0
          524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
         1048576 |                                         0

                2
           value  ------------- Distribution ------------- count
          131072 |                                         0
```

```
    262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
    524288 |                                         0

         3
     value  ------------- Distribution ------------- count
      1024 |                                         0
      2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1278
      4096 |                                         2
      8192 |                                         0

time spent per digit

         3                 0
         2                 1
         1                 4
calls by digit

         1                 4
         2               640
         3              1280
         4            163840
time spent in different sections of the program

  workload distribution                                    0
  allocate temp array                                      0
  copy to main array                                       0
  sorting into buckets                                     4
  total                                                    5
  recursive calls                                          6
```

Listing 3.4: Output das *probes Pthreads* para a implementação *Pthreads* utilizando tamanho 1000000 e 8 *threads*

3.1.5 C++11

Para esta implementação foram definidas estas *probes* presentes para a versão *Pthreads start_sorting_into_buckets,finish_sorting_into_buckets,start_seq_radix,finish_seq_radix,start_par_radix,finish_par_radix,start_copy_to_main_array,finish_copy_to_main_array,start_workload_distribution* e *finish_workload_distribution*.

```
time spent sorting into buckets

         2                 0
         3                 1
         1                 4
array sizes per digit

         1
     value  ------------- Distribution ------------- count
    262144 |                                         0
    524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
   1048576 |                                         0

         2
     value  ------------- Distribution ------------- count
```

```
 131072 |                                         0
 262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
 524288 |                                         0

      3
  value  ------------- Distribution ------------- count
   1024 |                                         0
   2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1278
   4096 |                                         2
   8192 |                                         0

time spent per digit

                3                 1
                2                 2
                1                 6
calls by digit

                1                 4
                2               640
                3              1280
                4            163840
time spent in different sections of the program

  workload distribution                                           0
  copy to main array                                              0
  sorting into buckets                                            6
  total                                                           8
  recursive calls                                                10
```

Listing 3.5: Output das *probes C++11* para a implementação *C++11* utilizando tamanho 1000000 e 8 *threads*

3.2 CPC probes

Probe	**Uso**
PAPI_l1_dcm-all-5000	Obter de uma forma aproximada as *l1 data cache misses.*
PAPI_l2_dcm-all-5000	Obter de uma forma aproximada as *l2 data cache misses.*
PAPI_l1_icm-all-5000	Obter de uma forma aproximada as *l1 instruction cache misses.*
PAPI_l2_icm-all-5000	Obter de uma forma aproximada as *l2 instruction cache misses.*

Tabela 3.4: *Custom probes* definidas para a versão *Mpi*

```
data cache misses l1

  l1 cache data misses                                      5510000

data cache misses l2

  l2 cache data misses                                       695000
```

```
instruction cache misses l1

  l1 cache instruction misses                                    1600000

instruction cache misses l2

  l2 cache instruction misses                                     100000
```

Listing 3.6: Output das *probes cpc* para a implementação *Omp* utilizando tamanho 50000000 e 1 *thread*

O objetivo por detrás do uso destas *probes* consiste em medir vários tipos de *cache misses* ocorrentes durante a execução do programa.

O resultado destas *probes* consiste nas *cache misses l1* e *l2* de dados e instruções.

3.3 SYSINFO probes

Probe	**Uso**
nthreads	Obter o número de *threads* criado.
pswitch	Obter o número de *cpu switches* das *threads*.
procovf	Obter falhas na criação de processos.
bwrite	Obter número e tamanho das escritas.
bread	Obter número e tamanho das escritas.
inv_swtch	Obter número de *switches* involuntarios.
sysfork	Obter número de chamadas a *system call fork* .
sysexec	Obter número de chamadas a *system call exec* .

Tabela 3.5: *Custom probes* definidas para a versão *Mpi*

```
info collected from the system

  number of threads created                                          1
  number of times threads where forced to give up cpu               50
  number of cpu switches between threads                            87
```

Listing 3.7: Output das *probes sysinfo* para a implementação *Omp* utilizando tamanho 50000000 e 1 *thread*

O objetivo destas *probes* consiste em medir informação relativa a processos do sistema.

O resultado destas *probes* consiste em contagem e quantificações destes eventos.

3.4 PLOCKSTAT probes

Probe	**Uso**
mutex-block *mutex-spin* *mutex-release* *mutex-error*	Obter os tempos de espera pelos e retenção dos *mutexes*.

Tabela 3.6: Custom probes definidas para a versão Mpi

```
info collected from the system

  wait for mutex
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@@@@                                     2
            4096 |@@                                       1
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 12
           16384 |@@@@@@@@                                 4
           32768 |                                         0

  time holding mutex
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@                                       1
            8192 |@@@@                                     2
           16384 |                                         0
           32768 |                                         0
           65536 |                                         0
          131072 |                                         0
          262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 16
          524288 |                                         0
```

Listing 3.8: Output das probes plockstat para a implementação *Omp* utilizando tamanho 10000000 8 *threads*

O objetivo destas *probes* consiste em criar quantificações relativas aos tempos de espera e de retenção de *mutexes*.

3.5 SCHED probes

Probe	**Uso**
on-cpu *off-cpu*	Obter os tempo total gasto em cada *CPU* e os tempos por cada *strech* por cada em cada *CPU*.

Tabela 3.7: *Custom probes* definidas para a versão *Mpi*

```
time spent per strech per  cpu

        6
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@                                 3
           32768 |@@@@@@@@                                 3
           65536 |                                         0
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |                                         0
         2097152 |                                         0
         4194304 |                                         0
```

```
 8388608 |@@@                                      1
16777216 |                                        0
33554432 |                                        0
67108864 |@@@@@@@@@@@@@@@@                        6
134217728 |@@@@@                                   2
268435456 |                                        0

                              ...


5
    value  ------------- Distribution ------------- count
     8192 |                                        0
    16384 |@@                                      1
    32768 |@@                                      1
    65536 |                                        0
   131072 |                                        0
   262144 |                                        0
   524288 |                                        0
  1048576 |                                        0
  2097152 |                                        0
  4194304 |                                        0
  8388608 |                                        0
 16777216 |@@@@@                                   2
 33554432 |@@@@@                                   2
 67108864 |@@@@@                                   2
134217728 |@@@@@@@@@@@@@@@@@@@@@@@@@               9
268435456 |                                        0

time per cpu

        6                 0
        4                 1
        2                 1
        7                 1
        1                 1
        0                 1
        3                 1
        5                 2
```

Listing 3.9: Output das probes sched para a implementação *Omp* utilizando tamanho 100000000 e 8 *threads*

O objetivo destas *probes* consiste em quantificar o tempo em cada *core* por *strech* e somar o tempo total do programa em cada *core*.

3.6 VMINFO probes

Probe	**Uso**
todas	Obter as ocorrências dos diferentes eventos na execução do programa.

Tabela 3.8: Probes VMINFO

```
info collected from the system

  number of threads created                                              7
  number of times threads where forced to give up cpu                  131
  number of cpu switches between threads                               263
```

Listing 3.10: Output das *probes vminfo* para a implementação *Omp* utilizando tamanho 100000000 e 8 *threads*

Quantifica varias informações dessas *probes*.

3.7 libmpi probes

Probe	**Uso**
MPI_Bcast *MPI_Scatterv* *MPI_Gathrev* *MPI_Send* *MPI_Recv*	Obter o tempo gasto em envios de mensagens utilizando esta primitiva e tamanhos das mensagens enviadas com a mesma.

Tabela 3.9: Probes libmpi

```
                       Communication times using different primitives

  gatherv                                                                0
  bcast                                                                  0
Integers sent or received for each function call

  bcast
           value  ------------- Distribution ------------- count
              64 |                                         0
             128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
             256 |                                         0

  gatherv
           value  ------------- Distribution ------------- count
         2097152 |                                         0
         4194304 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 80
         8388608 |                                         0


Communication times using different primitives

  gatherv                                                                0
  bcast                                                                  0
Integers sent or received for each function call

  bcast
           value  ------------- Distribution ------------- count
              64 |                                         0
             128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
```

```
           256 |                                                 0

gatherv
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@                                      5
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |                                         0
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |                                         0
            1024 |                                         0
            2048 |                                         0
            4096 |                                         0
            8192 |                                         0
           16384 |                                         0
           32768 |                                         0
           65536 |                                         0
          131072 |                                         0
          262144 |                                         0
          524288 |                                         0
         1048576 |                                         0
         2097152 |                                         0
         4194304 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 75
         8388608 |                                         0
```

Listing 3.11: Output das probes libmpi para a implementação mpi utilizando tamanho 100000000 com 8 processos

Quantifica várias informações relativas as *probes*.

Estas *probes* foram utilizadas para quantificar o tamanho das mensagens enviadas e medir o tempo de comunicação total utilizado por cada uma das primitivas utilizadas na implementação em memória distribuída.

Capítulo 4

Teste e Resultados

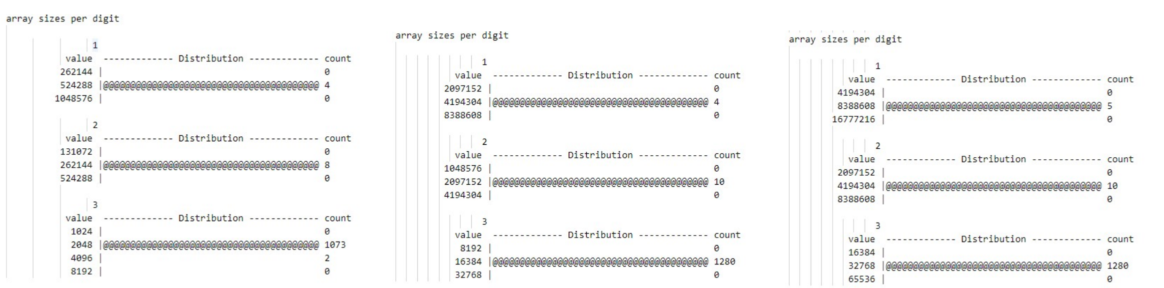
4.1 Metodologia e ambiente de teste

As varias implementações foram corridas utilizando números variáveis de processos e *threads* para as implementações paralelas numa maquina virtual *Solaris*,com um *CPU AMD*,os vários *scripts* criados foram utilizados para obter estatísticas acerca da execução dos programas.

Para o efeito de testes foram utilizados *arrays* de tamanhos 1000000, 5000000 e 10000000 e cada medição efetuada mede a ordenação de um *array* de um determinado tamanho 5 vezes. Os testes foram automatizados utilizando *bash scripts* e foram também criados gráficos que permitam melhor visualizar alguns dos resultados obtidos.

4.2 Resultados

4.2.1 Versão Sequencial



(a) 1000000 (b) 5000000 (c) 10000000

Figura 4.1: Distribuição do tamanho dos *arrays* a serem ordenados pelas chamadas as funções para diferentes *bits* para diferentes tamanhos do *array* original.

Os testes realizados permitem verificar que existe uma distribuição uniforme do *array* utilizado para teste, isto acontece porque este *array* é gerado aleatoriamente com uma distribuição uniforme. Visto que os *arrays* são gerados sequencialmente para todas as versões do programa é esperado que os valores do mesmo sejam uniformes para todas as versões, isto foi verificado com os testes realizados nas outras versões.

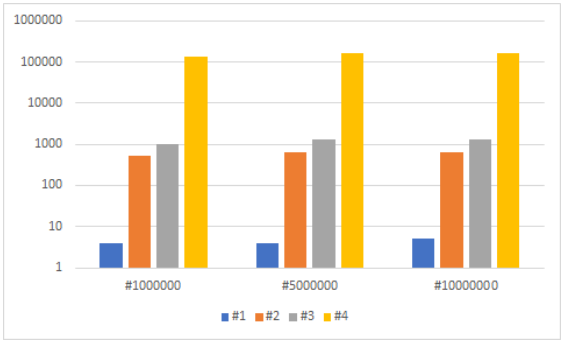


Figura 4.2: número de chamadas as funções *radix sort* por digito

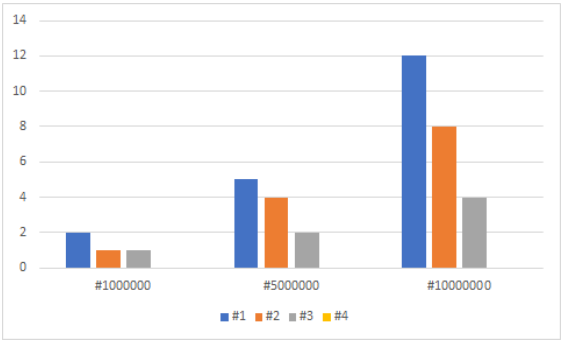


Figura 4.3: Tempo gasto por digito em segundos em escala logarítmica com base 10

A partir dos testes pude verificar um aumento de tempo considerável gasto no cálculo do primeiro conjunto de *bits* comparativamente com os restantes a medida que o tamanho de input aumenta. Como seria esperado pelo facto dos *arrays* serem distribuídos uniformemente e por serem de tamanho substancial o número de chamadas as funções por cada digito tem pouca variação com o aumento do tamanho de dados.

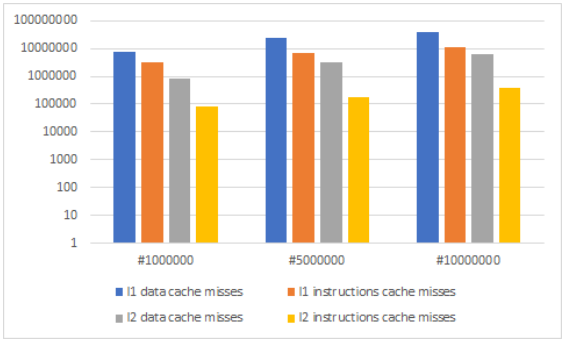


Figura 4.4: número de cache misses para l1 e l2 valor absoluto escala logarítmica base 10

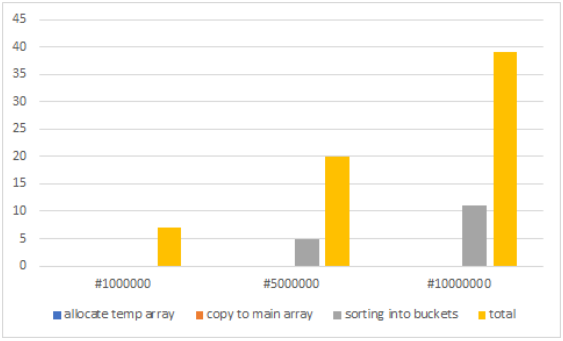


Figura 4.5: tempo gasto por secção em segundos

Para os vários tamanhos pode-se verificar que o número de *cache misses* relacionadas com os dados é bastante superior as de instruções, neste programa isto era esperado pelo tamanho dos dados e por este possuir varias operações que envolvem copiar valores entre posições de memória. Pude também notar um aumento dos vários tipos de *cache misses* com o aumento do tamanho do *array* original. Pude notar também a secção *sorting into buckets* representa uma maior secção de execução do programa relativamente as restantes.

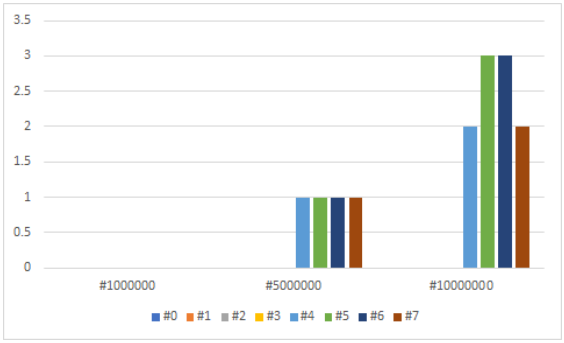


Figura 4.6: Tempo total por *thread* em segundos

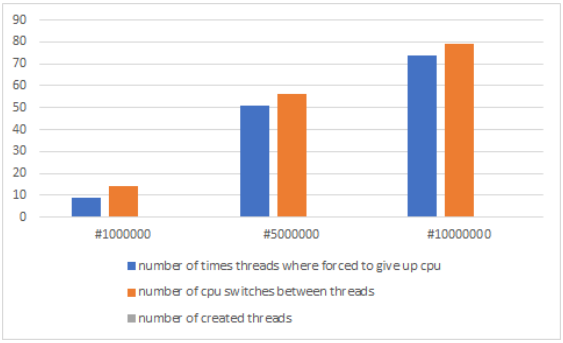


Figura 4.7: *threads* criadas pelo programa, *CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

A partir dos testes pude também notar que a versão sequencial do programa executa em diferentes *threads* do processador, este salto entre *threads* poderia explicar alguns problemas de performance devido ao uso menos eficiente de cache sendo que não permite que o programa utilize os valores já incluídas numa cache. Este problema subsede parcialmente por o ambiente de teste ser partilhado, o mesmo poderia não acontecer no caso de este ser o único programa em execução. Com o aumento do tamanho verifiquei também um aumento considerável das comutações de contexto.

4.2.2 Versão OpenMp

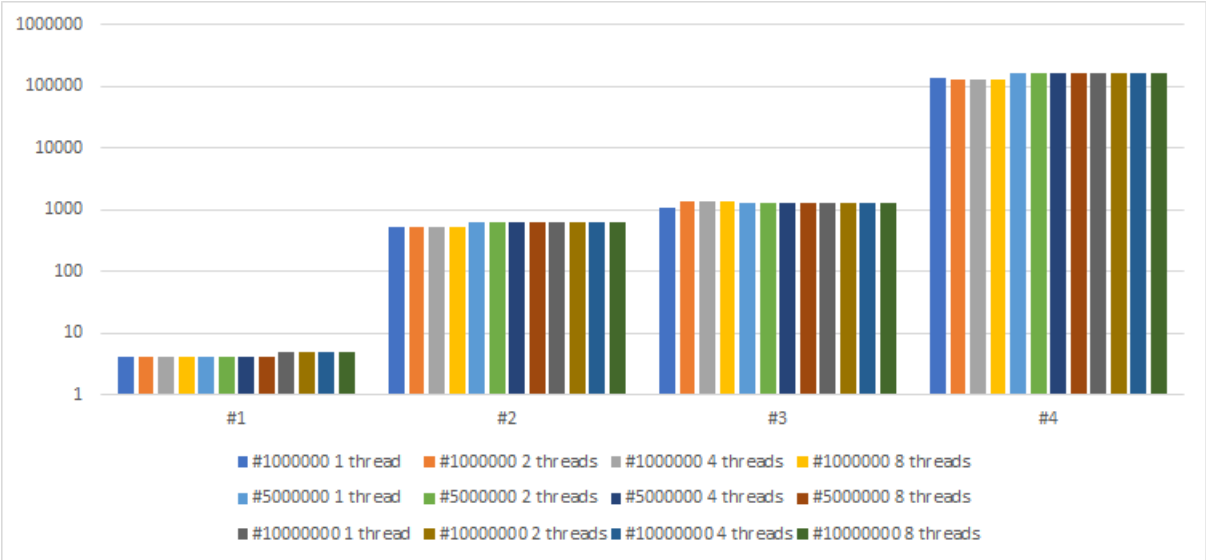


Figura 4.8: número de chamadas as funções *radix sort* por digito

Como seria esperado pude verificar que o número de chamadas não varia consoante o tamanho ou o número de *threads*.

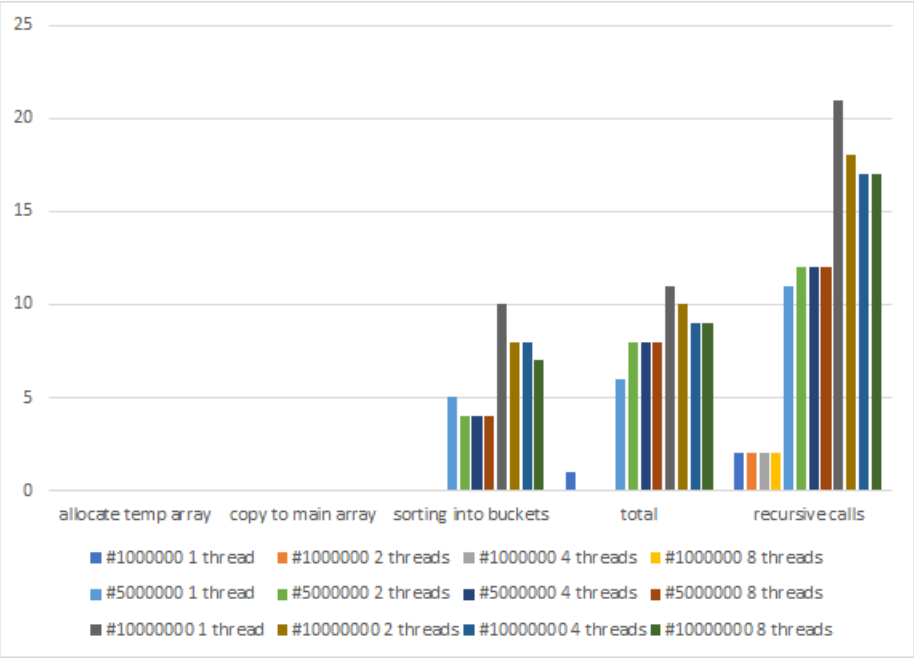


Figura 4.9: tempo gasto por secção em segundos

Em termos de tempo gasto por secção do programa verifiquei que o tempo gasto na ordenação em *buckets* continua a ser superior ao tempo gasto na copia para o *array* original e o tempo de alocação do *array* temporário. Algo verificado que seria inesperado é o facto de o tempo cumulado das chamadas recursivas ser consideravelmente superior para o maior tamanho, isto poderá acontecer devido a existir possivelmente um maior número de saltos entre diferentes localizações das *threads*.

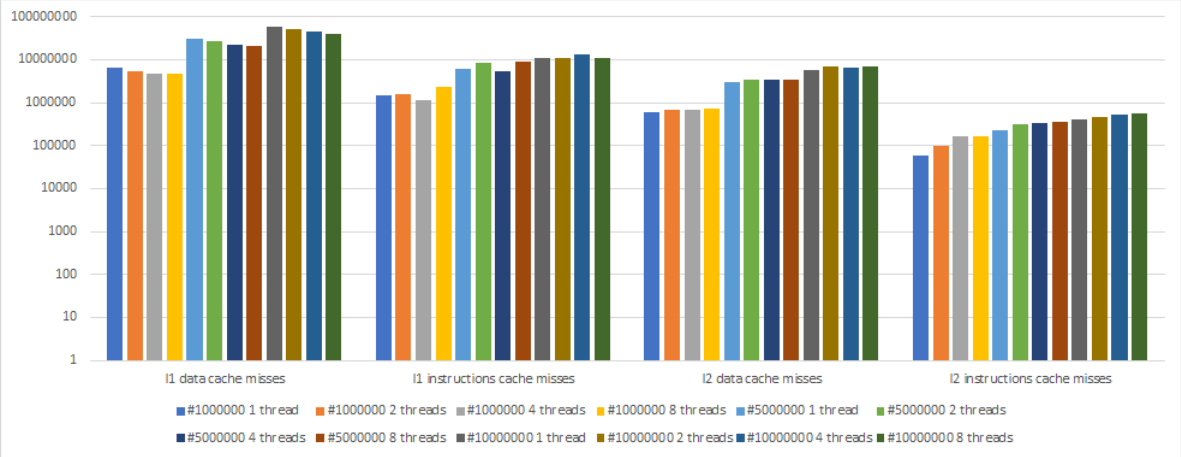


Figura 4.10: número de cache misses para *l1* e *l2* valor absoluto escala logarítmica base 10

Em termos de *cache misses* como seria esperado estas são maiores para dados e para *l1* e tendem a aumentar com o tamanho dos dados. Relativamente ao número de *threads* verifica-se que que estas diminuem com o número de *threads* para dados em *l1* mas possuem uma tendência a aumentar com o número de *threads* para os restantes tipos de *misses* e níveis.

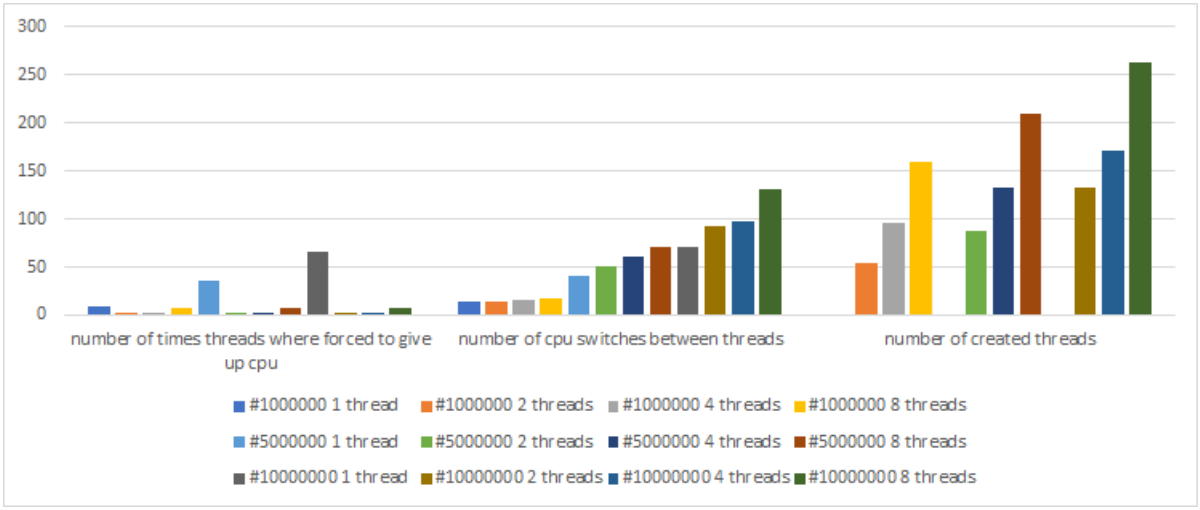


Figura 4.11: *threads* criadas pelo programa,*CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

Verifiquei também um aumento das *threads* criadas com o aumento do número de *threads* máximo, verifiquei também que ocorre um maior número de comutações de contexto com um aumento do número de *threads* para o tamanho maior o que explica os resultados vistos anteriormente.

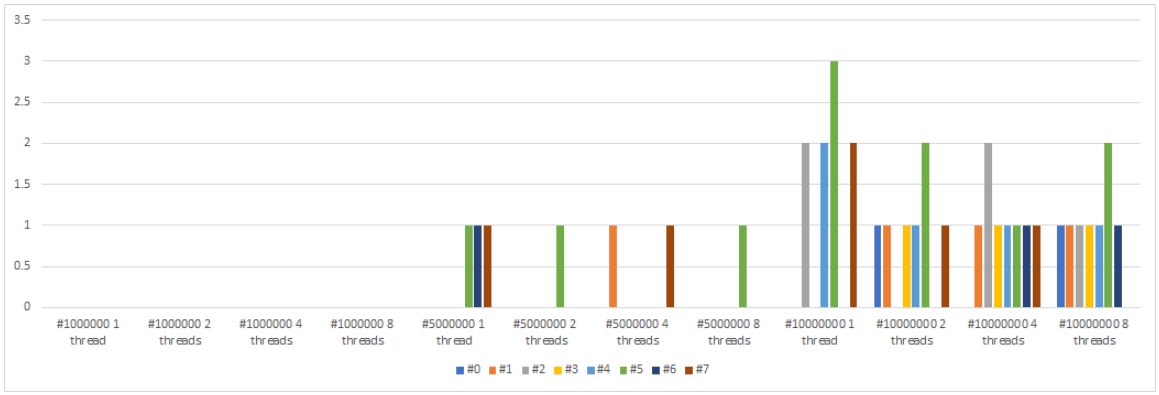


Figura 4.12: Tempo total por *thread* em segundos

É possível concluir que a carga foi realizada por varias *threads* do *CPU*.Verifiquei também que a carga não está bem distribuída pela varias *threads* do *CPU* mesmo para os maiores tamanhos isto pode ser dado por existir uma secção do programa realizada sequencialmente.

4.2.3 Versão Mpi

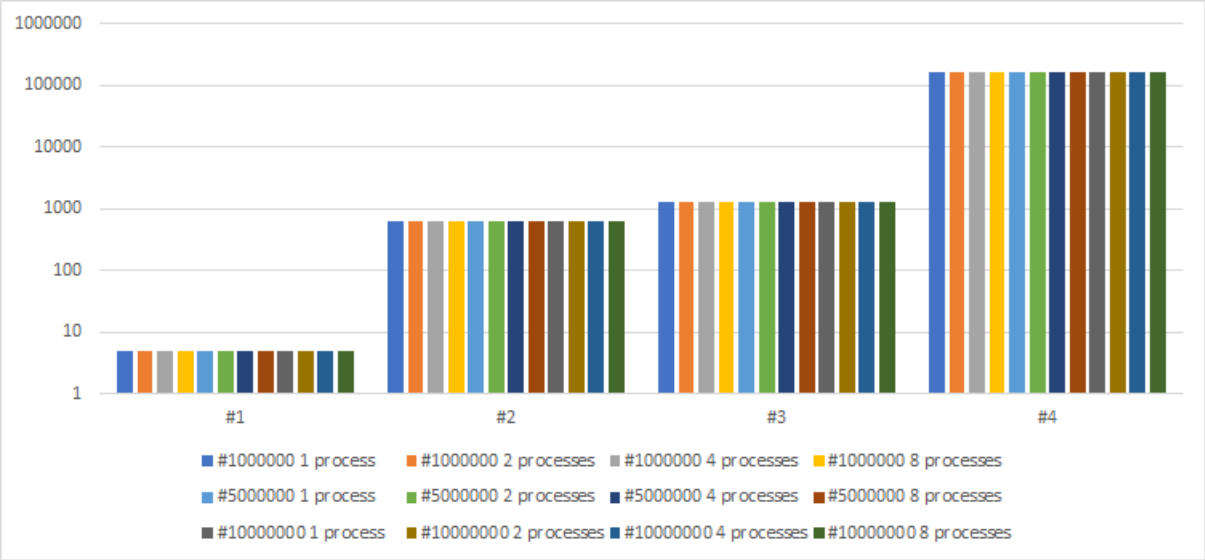


Figura 4.13: número de chamadas as funções *radix sort* por digito

Mais uma vez o número de chamadas por cada digito apresenta pouca variação com o número de *threads* e tamanho do problema.

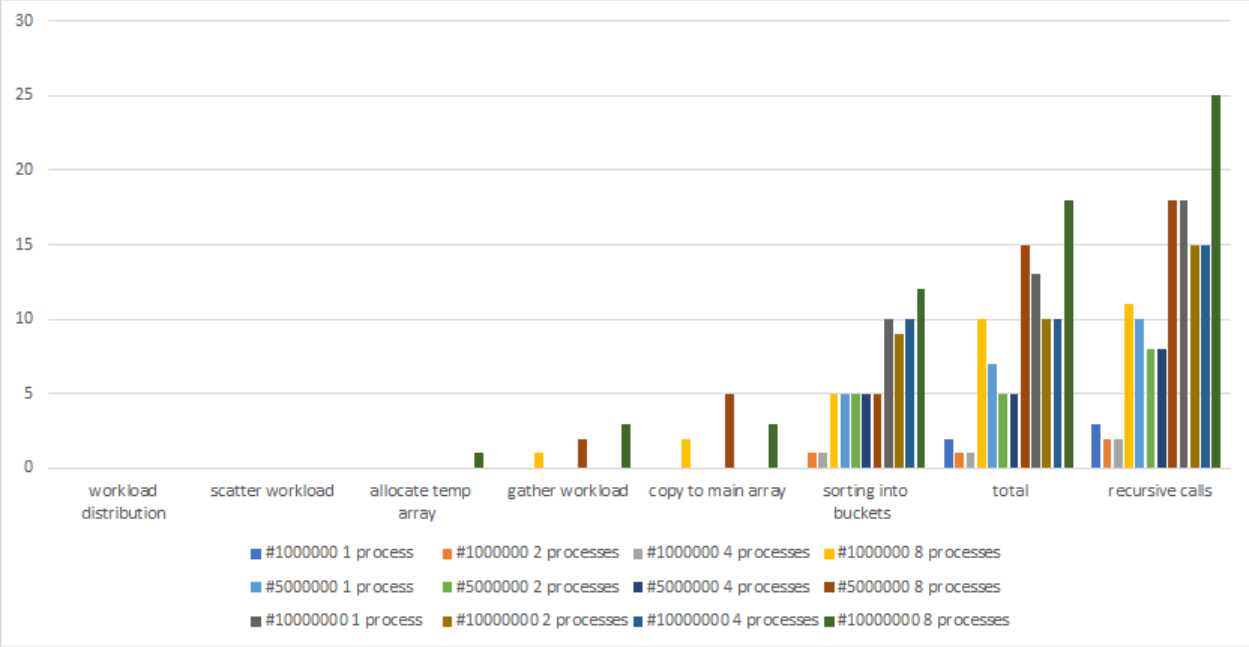


Figura 4.14: tempo gasto por secção em segundos

Comparativamente a versão em memória partilhada a versão em memória distribuída apresenta tempos mais elevados de copia para o *array* principal, isto poderá acontecer devido ao maior uso de memória devido a existência de comunicação entre os processos. verifiquei também piores tempos passando de 2 e 4 *threads* para 8 *threads* essas são dadas maioritariamente pelos acréscimos no tempo de comunicação.

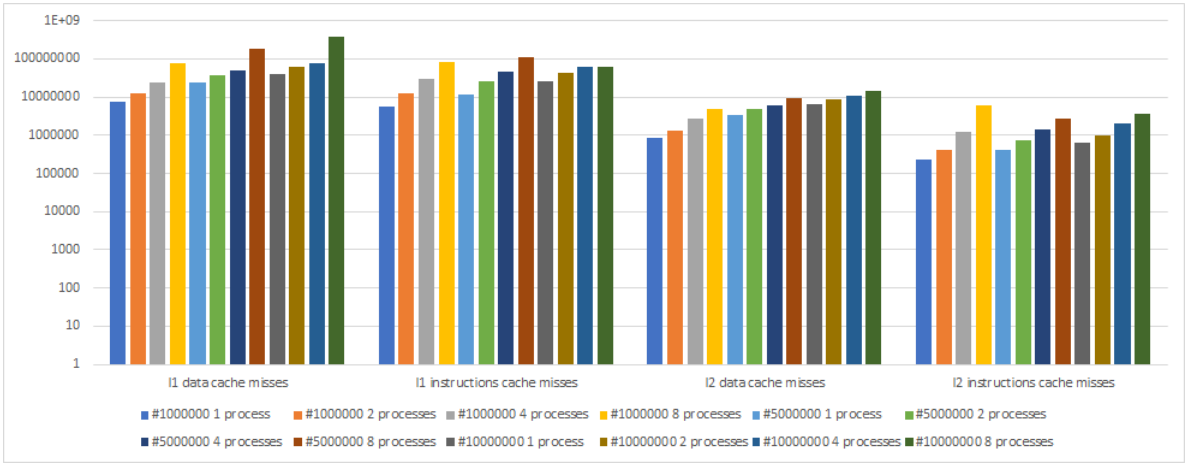


Figura 4.15: número de cache misses para *l1* e *l2* valor absoluto escala logarítmica base 10

Em termos de cache misses os resultados são semelhantes a versão em memória partilhada *OMP*.

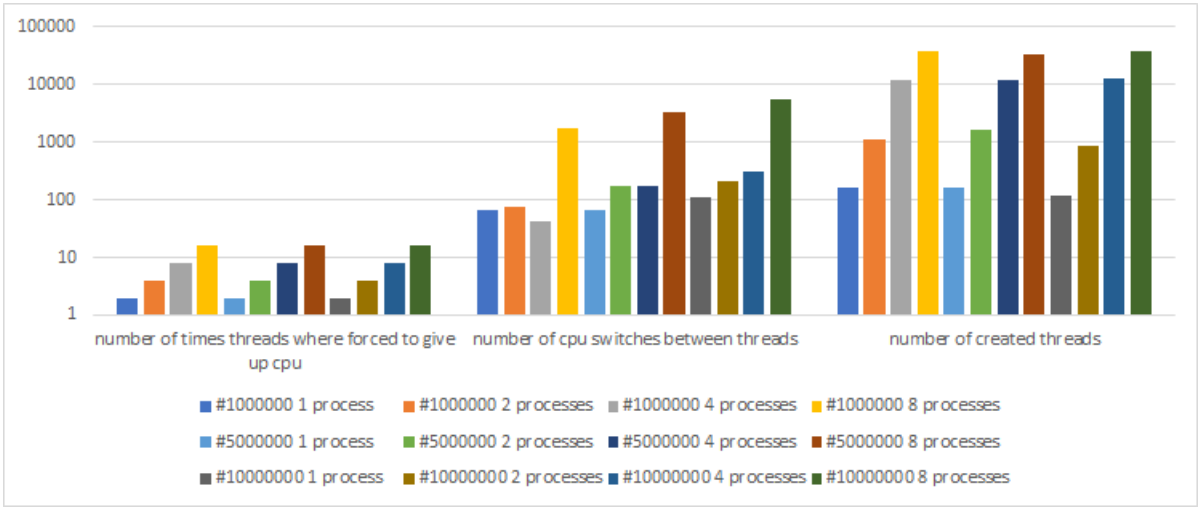


Figura 4.16: *threads* criadas pelo programa,*CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

Em termos de comutações de contexto estes aumentam com o número de cores, mas não com o tamanho do problema, estes valores explicam alguns dos problemas de performance encontrados.

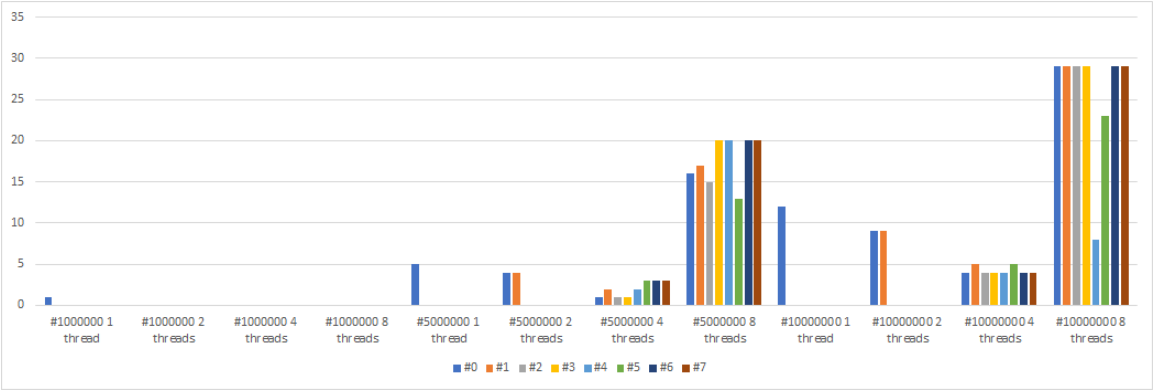


Figura 4.17: Tempo total por *thread* em segundos

Para 8 *threads* e os dois maiores tamanhos é possível verificar uma distribuição não uniforme da carga.

```
bcast
     value  ------------- Distribution ------------- count
        64 |                                         0
       128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
       256 |                                         0

gatherv
     value  ------------- Distribution ------------- count
        -1 |                                         0
         0 |@@@                                      5
         1 |                                         0
         2 |                                         0
         4 |                                         0
         8 |                                         0
        16 |                                         0
        32 |                                         0
        64 |                                         0
       128 |                                         0
       256 |                                         0
       512 |                                         0
      1024 |                                         0
      2048 |                                         0
      4096 |                                         0
      8192 |                                         0
     16384 |                                         0
     32768 |                                         0
     65536 |                                         0
    131072 |                                         0
    262144 |                                         0
    524288 |                                         0
   1048576 |                                         0
   2097152 |                                         0
   4194304 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 75
   8388608 |                                         0
```

Figura 4.18: Distribuição das mensagens pelo seu tamanho

Relativamente a comunicação realizada pude verificar que esta é realizada maioritariamente uti-

lizando as primitivas *gathrev* e *scatterv* e as restantes utilizando *broadcast* em termos de tempo gasto nas comunicações medir o tempo passado entre as entradas e retornos das funções permitiu visualizar corretamente os tempos de comunicação totais sendo que estas não chegam a um segundo o que não indica eventuais tempos de espera para o acesso a estes dados visto que o programa poderá estar a utilizar *buffer* para realizar as escritas nas novas posições de memória. Em termos de distribuição do tamanho de dados enviado por mensagens este é distribuído uniformemente em cada uma das primitivas utilizadas.

4.2.4 Versão Pthreads

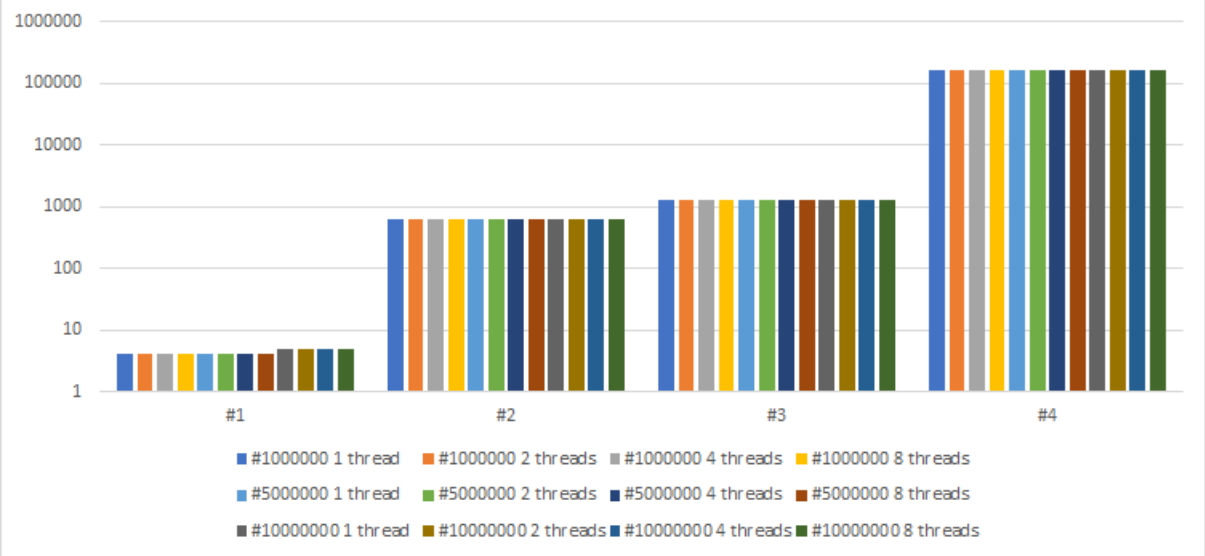


Figura 4.19: número de chamadas as funções *radix sort* por digito

Como seria esperado pude verificar que o número de chamadas não varia consoante o tamanho ou o número de *threads*.

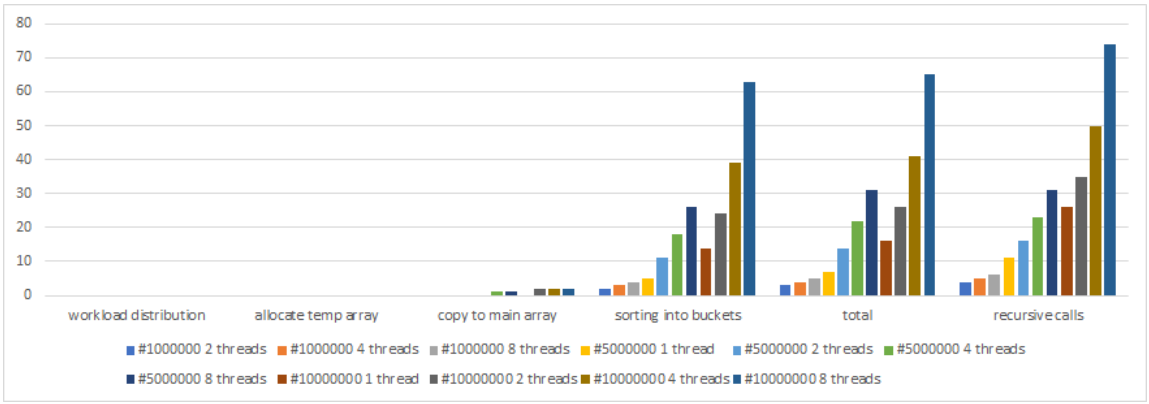


Figura 4.20: tempo gasto por secção em segundos

Algo que foi possível verificar na execução desta aplicação foi que para os vários tamanhos e nas varias secções os tempos de execução aumentam com o número de *threads* utilizado.

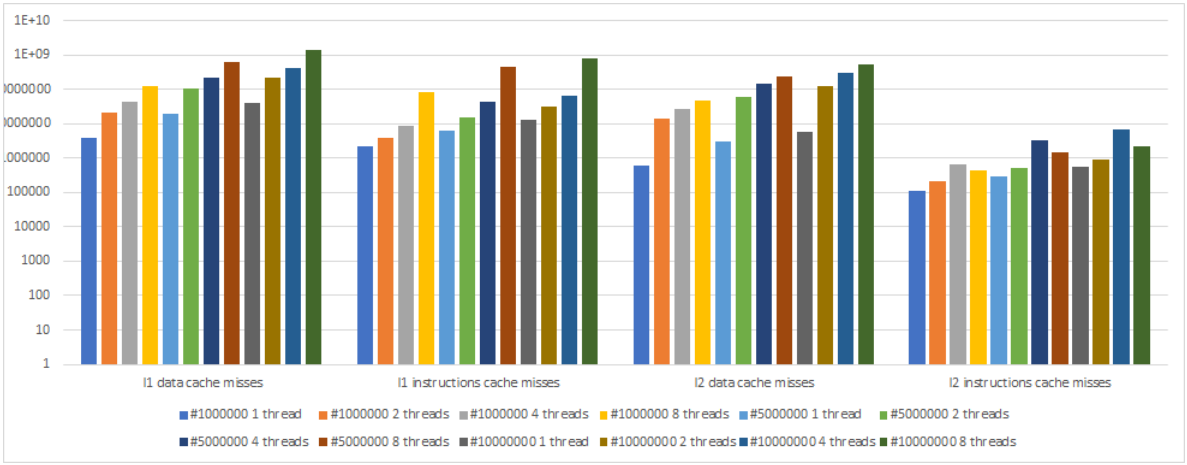


Figura 4.21: número de cache misses para l1 e l2 valor absoluto escala logarítmica base 10

Mais uma vez as *cache misses* para *l1* são maiores do que as de *l2* e as de dados do que as de instruções. Verifica-se também um aumento das *misses* com o número de *threads*, isto acontece devido a menor eficiência nos acessos a memória devido a vários fatores como a partilha de recursos pelas *threads*,*false sharing* e as *threads* serem movidas entre os *CPUs* da maquina.

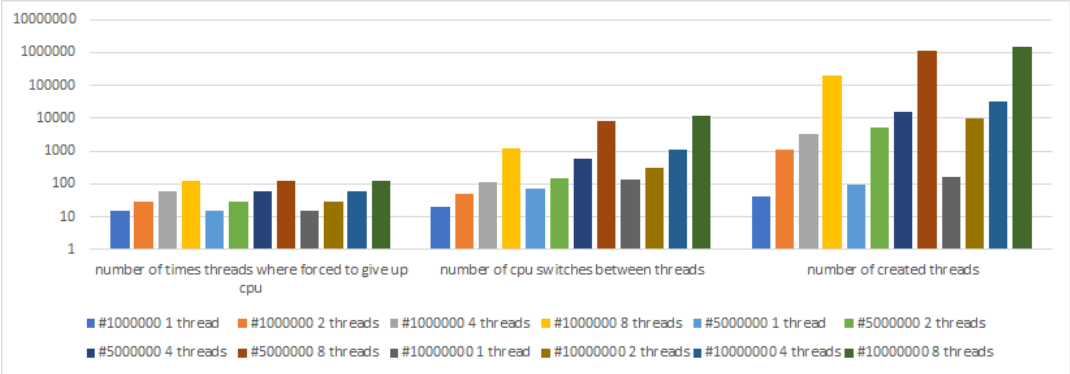


Figura 4.22: *threads* criadas pelo programa,*CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

Mais uma vez os números de comutações de contexto e criações de *threads* aumentam com o número de *threads* criadas.

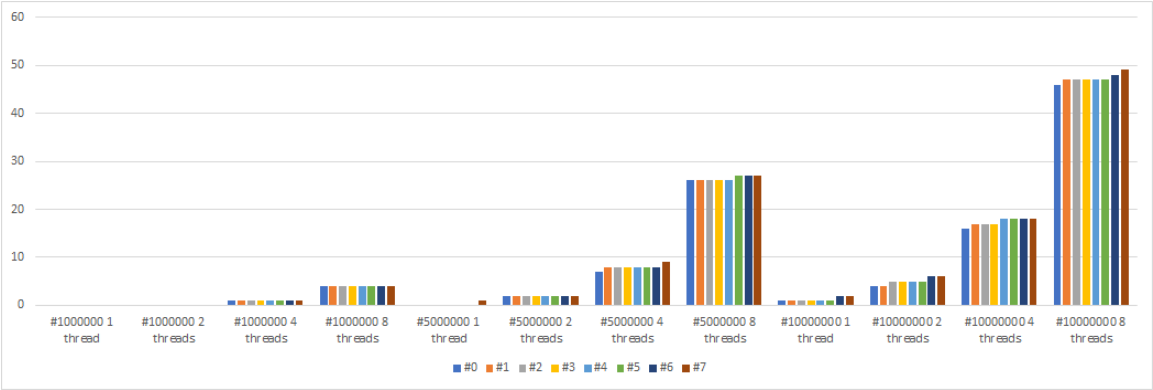


Figura 4.23: Tempo total por *thread* em segundos

A partir deste gráfico pude verificar que a repartição do trabalho é feita de maneira equilibrada entre os *cores* lógicos do *CPU*,mesmo em casos em que o programa utiliza um número máximo de *threads* inferior aos *cores* lógicos do *CPU*.Estes resultados explicam também alguns dos problemas relacionados com o mau uso da memória quando são utilizadas mais do que 1 *thread*.

4.2.5 Versão C++11

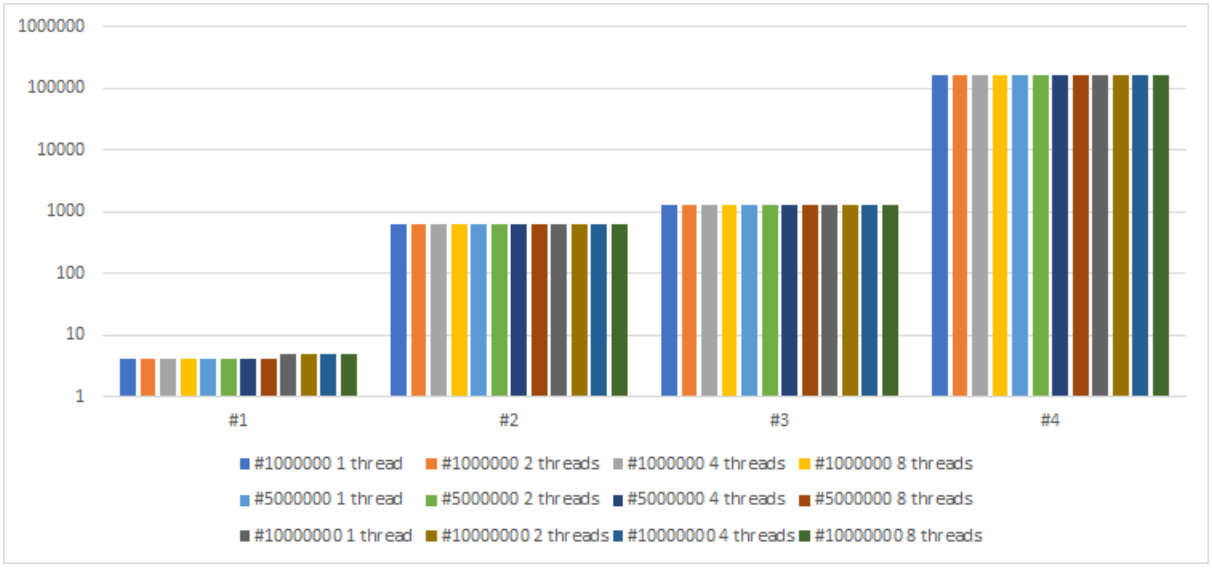


Figura 4.24: número de chamadas as funções *radix sort* por digito

Como seria esperado pude verificar que o número de chamadas não varia consoante o tamanho ou o número de *threads*.

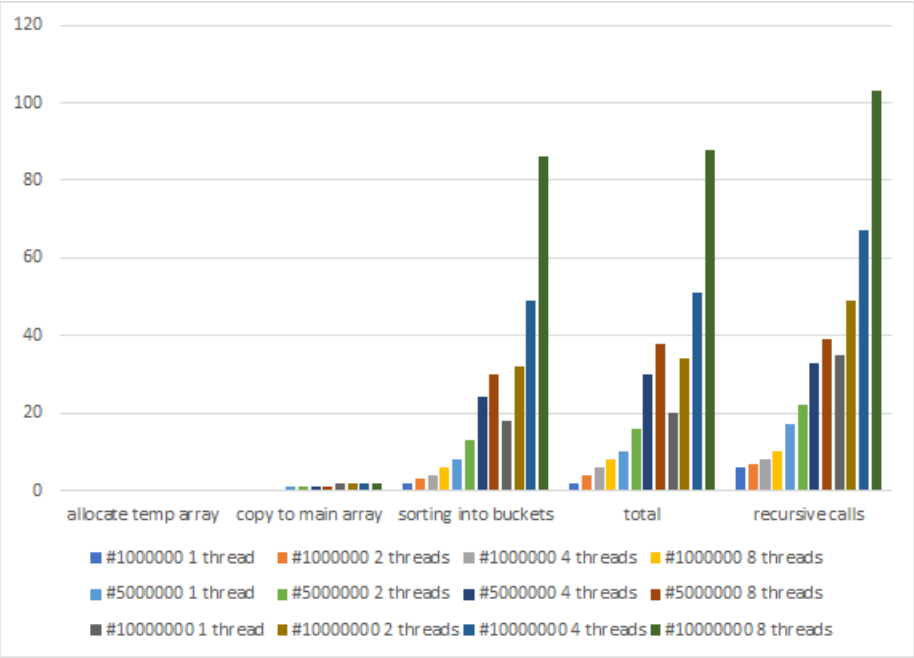


Figura 4.25: tempo gasto por secção em segundos

Tal como na implementação *Pthreads* foi possível verificar na execução desta aplicação que para os vários tamanhos e nas varias secções os tempos de execução aumentam com o número de *threads* utilizado.

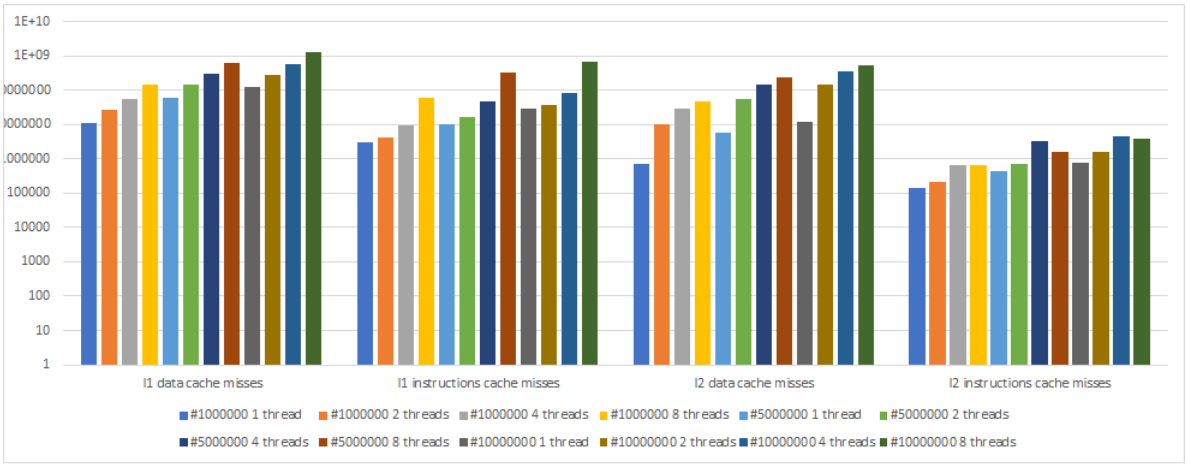


Figura 4.26: número de cache misses para *l1* e *l2* valor absoluto escala logarítmica base 10

Os resultados foram similares aos das versões anteriores, maior número de *misses* para *l1* do que *l2* e para dados relativamente a instruções.

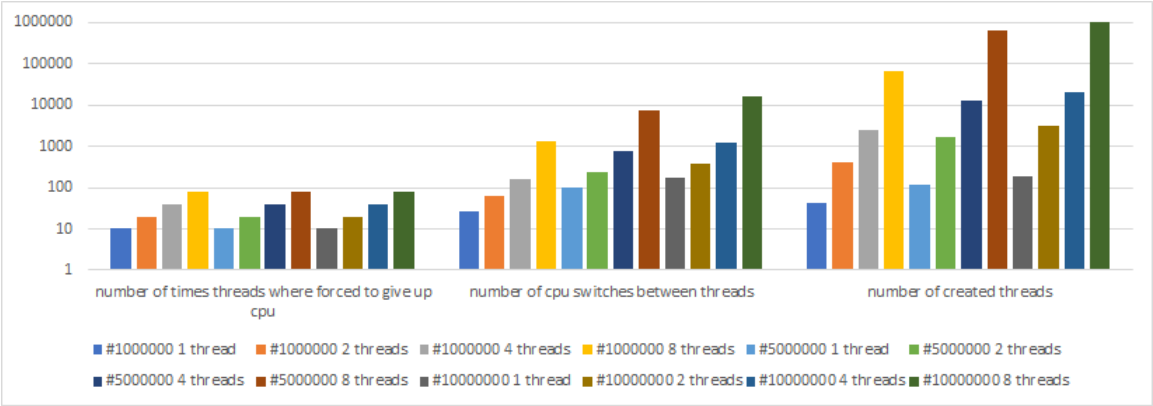


Figura 4.27: threads criadas pelo programa, *CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

Como em versões anteriores as comutações de contexto e número de *threads* criadas aumentam com o número de *threads* utilizado pelo programa.

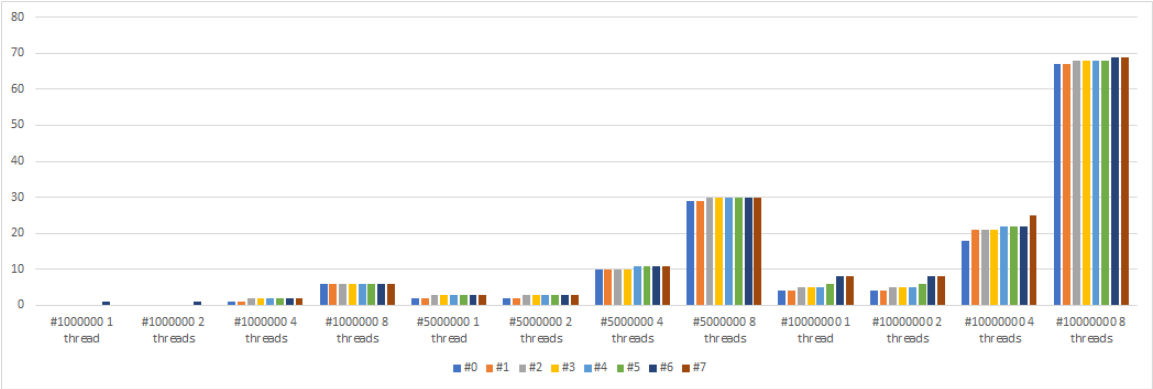


Figura 4.28: Tempo total por *thread* em segundos

Os resultados foram similares aos da versão *Pthreads* em que o trabalho esta repartido de uma forma equilibrada pelos *cores* lógicos do *CPU*.

Capítulo 5

Conclusões e trabalho futuro

O trabalho realizado para a concretização das tarefas descritas neste relatório permitiu obter algum conhecimento sobre o funcionamento e uso da ferramenta *Dtrace* bem como outros conhecimentos relacionados com sistemas operativos, *hardware* e na sua influência na execução de programas.

Em termos de trabalho futuro este poderá consistir numa exploração mais profunda das funcionalidades da ferramenta *Dtrace* e o uso desta e outras ferramentas de traçado e monitorização para a monitorização e análise de outros programas e sistemas.

Bibliografia

[1] *dtrace.org*. 2020. URL: http://dtrace.org/blogs/about/.

[2] *DTracing Hardware Cache Counters*. 2013. URL: https://www.joyent.com/blog/dtracing-hardware-cache-counters.

[3] *Oracle Linux DTrace Tutorial*. 2019.

[4] *Oracle Solaris 11.4 DTrace (Dynamic Tracing) Guide*. 2019.

[5] *Sun HPC ClusterTools 8 Software User's Guide*. 2008. URL: https://docs.oracle.com/cd/E19356-01/820-3176-10/Dtrace-mpiperuse.html.