Exercícios sobre a Ferramenta Dtrace

Sérgio Caldas
Universidade do Minho
Escola de Engenharia
Departamento de Informática
Email: a57779@alunos.uminho.pt

Conteúdo

***Resumo*—Este relatório, exprime os resultados obtidos na resolução de exercícios sobre *Dtrace*, no âmbito da disciplina de Engenharia de Sistemas de Comuptação (ESC), inserida no perfil de Computação Paralela e Distribuída (CPD) do curso de Engenharia Informática. O objetivo deste trabalho é praticar o uso do *Dtrace*, numa máquina *Soláris 11*, para isso foram propostos dois exercícios, sendo que os seus resultados são apresentados ao longo deste relatório.**

1. Introdução

Como foi dito anteriormente, foram propostos dois exercícios para a praticar o uso do *Dtrace*. O primeiro exercício consiste em desenvolver uma script em *D* que faça um traçado das chamadas ao sistema *open()* (no caso de de uma máquina *Soláris 11* openat()), imprimindo por linha o nome do ficheiro executável, PID do processo, UID do utilizador e GID do grupo, o caminho absoluto para o ficheiro que for aberto, a cadeia de caracteres com as *flags* da chamada ao sistema openat() (O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT) e por fim o valor de retorno de chamada ao sistema. Este exercício contem um parte opcional, que consiste em modificar a script para sejam apenas detetados os ficheiros com *etc/* no caminho.

O segundo exercício proposto consiste, para todos os processos que estão no sistema, em contar o número de tentativas de abrir ficheiros existentes, o número de tentativas para criar ficheiros e contar o número de tentativas bem-sucedidas. Posteriormente a script deve imprimir, com um período (especificado em segundos) passado como argumento na linha de comandos, a hora e dia atual em formato legível e as estatísticas recolhidas por PID e respetivo nome.

Para além dos exercícios em cima referidos, no final do semestre ainda foi sugerido um exercício extra. Para a execução deste exercício, foi-nos fornecido um código paralelo, que gera um números aleatórios com gama num intervalo dado. Para além desse código foi-nos fornecido também uma *script DTrace* que faz um traçado dinâmico do código referido anteriormente. O objetivo deste trabalho extra passa por analiármos o comportamento das *threads* e da aplicação para os diferentes tipos de escalonamento.

2. Dtrace

O *Dtrace* é uma *Framework* para fazer traçados dinâmicos, esta *Framework* é usada para solucionar problemas no *Kernel* e aplicações em produção, em tempo real.

O *Dtrace* pode ser utilizado para se obter uma visão geral da execução do sistema, como a quantidade de memória, o tempo de CPU, os recursos usados por os processos activos. Esta *Framework* permite fazer traçados muito mais rebuscados e detalhados, tais como, por exemplo a lista de processos que tenta aceder a um ficheiro.

Os administradores de sistemas escrevem programas em *D*, ajustando esses programas à informação que querem obter. A linguagem *D*, em termos de estrutura é muito semelhante ao *Awk*. Estes programas em *D*, consistem numa lista de uma ou mais provas e a cada prova está associada uma acção.

3. Exercicio 1

Para este exercício, desenvolvi uma script em *D*, script essa que faz um traçado das chamadas ao sistema openat()

e imprime a seguinte informação por linha:

- Nome do ficheiro executável;
- PID do Processo;
- UID do Utilizador;
- GID do Grupo;
- Caminho absoluto para o ficheiro que for aberto;
- A cadeia de caracteres com as *flags* da chamada ao sistema openat() (O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT);
- O valor de retorno da chamada ao sistema.

Para alem de cada um destes tópicos, foi-nos proposto como exercício opcional, modificar a script de forma a só ser detetados os ficheiros com */etc* no caminho.

3.1. Script

A *Script* por mim criada contêm 2 provas, a primeira deteta à entrada a chamada ao sistema e guarda o *arg1* (que contem o caminho) na variável *self->path* bem como o *arg2* (que contem a *flag*) na variável *self->flags*. Na segunda prova no inicio, contem o predicado */strstr(self->path, "/etc") ! = NULL/* que permite apenas detectar os ficheiros com */etc* no caminho. Posteriormente nesta prova faço *strjoin* à variável *this->flags_string* da *string O_WRONLY* caso a condição *self->flags & O_WRONLY* se verifique, caso contrário verifica se a condição *self->flags & O_RDWR* é verdadeira e faz *strjoin* da *string O_RDWR*, caso a condição seja falsa faz *strjoin* da *string O_RDONLY*. Ainda nesta acção faço *strjoin* da *string |O_APPEND* caso a condição *self->flags & O_APPEND* se verifique, caso contrário faço *strjoin* da *string* , o mesmo se acontece para a *flag O_CREAT*.

No fim é imprimido no ecrã o nome do executável o PID do processo, o UID do utilizador, o GID do grupo, o caminho absoluto para o ficheiro que for aberto, a string com o conteúdo da variável *this->flag_string* e por fim o valor de retorno da chamada ao sistema.

```
/*
inline int O_RDONLY = 0;
inline int O_WRONLY = 1;
inline int O_RDWR = 2;
inline int O_APPEND = 8;
inline int O_CREAT = 256;
*/

this string flag_string;

syscall::openat:entry
{
        self->path = copyinstr(arg1);
        self->flags = arg2;
}

syscall::openat:return
/strstr(self->path, "/etc") != NULL/
{
    this->flags_string = strjoin(
        self->flags & O_WRONLY
        ? "O_WRONLY"
        : self->flags & O_RDWR
            ? "O_RDWR"
            : "O_RDONLY",
        strjoin(
            self->flags & O_APPEND  ? "|O_APPEND" : "",
            self->flags & O_CREAT   ? "|O_CREAT" : ""));

    printf("Executável: %s,%d,%d,%d, \"%s\",%s,%d\n", execname, pid, uid, gid↩
        ,self->path, this->flags_string, arg1);
}
```

3.2. Resultados Obtidos

Para testar a script executei alguns comandos de teste, propostos no enunciado, sendo que o resultado de cada comando pode ser consultado nas subsecções em baixo.

- Comando *cat /etc/inittab > /tmp/test* - Como podemos verificar pela tabela da figura 1, ao executarmos o comando *cat /etc/inittab > /tmp/test* na consola, este é logo detetado pelo *Dtrace* (1ª linha da tabela). Como ao executarmos o comando reencaminhados o *Output* para um novo ficheiro, na coluna das *flags* aparece a *flag* O_WRONLY que corresponde a leitura do ficheiro *inittab* e a *flag O_CREAT* que corresponde a criação do ficheiro *test_sergio*.

Execname	PID	UID	GID	Path	Flags	Valor de Retorno
bash	22177	29231	5000	/tmp/test_sergio	O_WRONLY|O_CREAT	4
cat	22177	29231	5000	/var/ld/ld.config	O_RDONLY	-1
cat	22177	29231	5000	/lib/libc.so.1	O_RDONLY	3
cat	22177	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	22177	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/etc/resolv.conf	O_RDONLY	10

Figura 1. Resultado Comando *cat /etc/inittab > /tmp/test*

- Comando cat /etc/inittab ≫ /tmp/test - Ao executarmos este comando estamos a fazer uma leitura do ficheiro *inittab*. Além disso este comando acrescenta o *Output* ao ficheiro *test_sergio* já existente, como tal é de esperar que na coluna referente às *flags* apareçam as *flags O_WRONLY, O_APPEND* e a *flag O_CREAT*, o que se verifica pela análise da 1ª linha da tabela da figura 2.

Execname	PID	UID	GID	Path	Flags	Valor de Retorno
bash	22182	29231	5000	/tmp/test_sergio	O_WRONLY|O_APPEND|O_CREAT	
cat	22182	29231	5000	/var/ld/ld.config	O_RDONLY	-1
cat	22182	29231	5000	/lib/libc.so.1	O_RDONLY	3
cat	22182	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	22182	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1

Figura 2. Resultado Comando cat /etc/inittab ≫ /tmp/test

- Comando cat /etc/inittab | tee /tmp/test - Ao executarmos este comando a saída do comando *cat /etc/inittab* vai ser gravada no ficheiro *test* ao mesmo tempo em que é exibida no ecrá. O resultado obtido pela execução deste comando pode ser consultado na tabela da figura 3.
- Comando cat /etc/inittab | tee -a /tmp/test - Esfe comando faz exactamente o que faz o comando do tópico anterior, com a diferença que com a *flag -a* este acrescenta a saída do comando *cat /etc/inittab* ao ficheiro *test*. O resultado obtido pela execução

Execname	PID	UID	GID	Path	Flags	Valor de Retorno
cat	22187	29231	5000	/var/ld/ld.config	O_RDONLY	-1
cat	22187	29231	5000	/lib/libc.so.1	O_RDONLY	3
cat	22187	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	22187	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
tee	22188	29231	5000	/var/ld/ld.config	O_RDONLY	-1
tee	22188	29231	5000	/lib/libc.so.1	O_RDONLY	3
tee	22188	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
tee	22188	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
tee	22188	29231	5000	/usr/lib/locale/en_US.UTF-8/LC_MESSAGES/SUNW_OST_OSLIB.mo	O_RDONLY	-1
nfsmapid	1351	1	12	/etc/resolv.conf	O_RDONLY	10

Figura 3. Resultado Comando cat /etc/inittab | tee /tmp/test

deste comando pode ser consultado na tabela da figura 4.

Execname	PID	UID	GID	Path	Flags	Valo de Retorno
tee	22197	29231	5000	/var/ld/ld.config	O_RDONLY	-1
tee	22197	29231	5000	/lib/libc.so.1	O_RDONLY	3
tee	22197	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
tee	22197	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
tee	22197	29231	5000	/usr/lib/locale/en_US.UTF-8/LC_MESSAGES/SUNW_OST_OSLIB.mo	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
cat	22196	29231	5000	/var/ld/ld.config	O_RDONLY	-1
cat	22196	29231	5000	/lib/libc.so.1	O_RDONLY	3
cat	22196	29231	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	22196	29231	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3

Figura 4. Resultado Comando cat /etc/inittab | tee -a /tmp/test

4. Exercicio 2

Neste exercício, foi-nos proposto criar uma script que para cada processo em execução no sistema mostra-se as seguintes estatísticas:

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

Estas estatísticas devem ser imprimidas no ecrã, juntamente com a hora e dia, repetidamente, com um período (em segundos) passado como argumento na linha de comandos.

4.1. Script

Esta *Script* é constituida por 3 provas, primeira vai contar o número de tentativas para criar ficheiros, para isso esta prova contém um predicado que testa se a *flag* da chamada ao sistema *openat* é a *flag O_CREAT*, caso se verifique este predicado é incrementado um contador através de uma agregação com um *array* associativo (*@create[execname,pid]*). As outras duas provas, uma conta o numero de tentativas para abrir um ficheiro e a outra o numero de tentativas bem sucedidas, as ações associadas a estas provas processam-se da mesma forma que foi explicado atrás. No final é imprimido no ecrã o dia e a hora, o nome do executável, o PID, o contador do *array* *@create*, o contador do *array* *@open* e o contador do *array* *@success*, com um período de 2 segundos

```
/*
inline int O_RDONLY = 0;
inline int O_WRONLY = 1;
inline int O_RDWR = 2;
inline int O_APPEND = 8;
inline int O_CREAT = 256;
*/

syscall::openat:entry
/(arg2 & O_CREAT)==O_CREAT/
{
        @create[execname,pid] = count();
}

syscall::openat*:entry
/(arg2 & O_CREAT) == 0/ {
         @open[execname, pid] = count();
}

syscall::openat*:return
/ arg1 > 0 / {
        @success[execname, pid] = count();
}

tick-$1s
{

         printf ("%Y\n",walltimestamp);
         printf("%12s %6s %6s %6s %s\n","EXECNAME", "PID", "CREATE", "OPEN", "↩
               SUCCESS");
         printa ("%12s %6d %@6d %@6d %@d\n", @create, @open, @success);
         trunc(@create);
         trunc(@open);
         trunc(@success);
}
```

4.2. Resultados Obtidos

Os resultados obtidos na execução da *script*, referente ao exercício 2, podem ser consultados na figura 5. Estes resultados foram obtidos através da execução do seguinte comando:

```
dtrace -qs exercicio2.d 2
```

Sendo que 2, é o período (segundos) pelo qual os resultados são imprimidos no ecrã. De referir que nem sempre as provas detectam alguma coisa, pode haver períodos em que não há nenhuma atividade que satisfaça as provas definidas.

```
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:20
        sshd  23318      0      5 6
        bash  23320      0      6 3
        nscd   1447      0     12 12
        sshd  23319      0     20 19
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:22
      dtrace  23375      0      2 2
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:24
       utmpd    259      0      1 2
          ls  23376      0      5 4
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:26
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:28
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:30
        bash  23334      0      1 1
    nfsmapid   1351      0      2 0
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:32
  automountd   1443      0      1 1
        bash  23334      0      1 1
    nfsmapid   1351      0      3 1
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:34
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:36
        bash  23334      0      1 1
    nfsmapid   1351      0      4 0
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:38
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:40
        bash  23334      0      1 1
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:42
        bash  23334      0      1 1
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:44
          ls  23377      0      5 4
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:46
        bash  23334      0      1 1
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:48
        bash  23334      0      2 2
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:50
          rm  23378      0      4 3
    EXECNAME    PID CREATE   OPEN SUCCESS
2016 Apr 10 17:45:52
```

5. Análise de Aplicação Paralela (OpenMP) com Ferramenta Dtrace

Nesta secção irei analisar, a execução de uma aplicação paralela, fornecida pelo professor, com uma *script* em *Dtrace*. A aplicação em causa é um gerador de números aleatórios, numa gama de um intervalo dado. A aplicação está paralelizada seguindo um paradigma de memória partilhada, mais propriamente com a utilização de pragmas *OpenMP*. O código referente à aplicação fornecida pelo professor pode ser consultado em baixo:

```
/*
     APina - 2016
   */
/*
   compilar com:  g++-mp-5 --std=c++11 -Wall -O2 -fopenmp -o ex2_v2 ex2_v2.cxx
   usar modos de escalonamento : export OMP_SCHEDULE="guided", OMP_SCHEDULE="dynamic", OMP_SCHEDULE="static"
 */
/*
   Execuo:
   ./ex2_v2 <n.threads><opcional- intervalo>
 */
#include <cstdlib>
#include <iostream>
#include <random>
using namespace std;
#include <omp.h>

#define S 1024*1024
//#define S 100

int main(int argc, char *argv[])
{
 int i, r, a[S], np, nr;
 double T1,T2;

 np = atoi(argv[1]);
 if (argc == 2)  nr= 99; else nr= atoi(argv[2]);

 std::random_device d;
 std::default_random_engine e1(d());
 // a distribution that takes randomness and produces values in specified range
 std::uniform_int_distribution<> dist(1,nr);

 omp_set_num_threads(np);
 T1 = omp_get_wtime();
#pragma omp parallel for private (r) schedule (runtime)
 for (i=0 ; i < S ; i++) {
  a[i] = 0.;
  for (r = dist(e1) ; r > 0 ; r -= 20) {
   a[i] += r;
  }
 }
 T2 = omp_get_wtime();
 cout << "fiosExecucao =" << np << " Intervalo=" << nr <<  ": tempo -> " << (T2-T1)*1e6 << " usecs\n";
}
```

A *script DTrace*, também fornecida pelo professor, permite-nos obter vários tipos de informação relativamente à aplicação em causa. A *script* permite-nos saber quando a *Thread* foi criada e finalizada, em que CPU está a correr (se trocar de CPU também é possível observar essa troca), quando é interrompida e como é interrompida. A *script* em *DTrace* pode ser consultada em baixo:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

BEGIN
{
  baseline = walltimestamp;
  scale = 1000000;
}
sched:::on-cpu
/pid == $target && !self->stamp /
{
  self->stamp = walltimestamp;
  self->lastcpu = curcpu->cpu_id;
  self->lastlgrp = curcpu->cpu_lgrp;
  self->stamp = (walltimestamp - baseline) / scale;
  printf("%9d:%-9d TID %3d CPU %3d(%d) created\n",
      self->stamp, 0, tid, curcpu->cpu_id, curcpu->cpu_lgrp);
```

```
  /*ustack(); */
}
sched:::on-cpu
/pid == $target && self->stamp && self->lastcpu\
      != curcpu->cpu_id/
{
  self->delta = (walltimestamp - self->stamp) / scale;
  self->stamp = walltimestamp;
  self->stamp = (walltimestamp - baseline) / scale;
  printf("%9d:%-9d TID %3d  from-CPU %d(%d) ",self->stamp,
      self->delta, tid, self->lastcpu, self->lastlgrp);
  printf("to-cpu %d(%d) CPU migration\n",
      curcpu->cpu_id, curcpu->cpu_lgrp);
  self->lastcpu =curcpu->cpu_id;
  self->latgrp = curcpu->cpu_lgrp;
}
sched:::on-cpu
/pid == $target && self->stamp && self->lastcpu\
      == curcpu->cpu_id/
{
  self->delta = (walltimestamp - self->stamp) / scale;
  self->stamp = walltimestamp;
  self->stamp = (walltimestamp - baseline) / scale;
  printf("%9d:%-9d TID %3d CPU %3d(%d) ",self->stamp,
      self->delta, tid, curcpu->cpu_id, curcpu->cpu_lgrp);
  printf("restarted on the same CPU\n");
}
sched:::off-cpu
/pid == $target && self->stamp /
{
  self->delta = (walltimestamp - self->stamp) / scale;
  self->stamp = walltimestamp;
  self->stamp = (walltimestamp - baseline) / scale;
  printf("%9d:%-9d TID %3d CPU %3d(%d) preempted\n",
      self->stamp, self->delta, tid, curcpu->cpu_id,
      curcpu->cpu_lgrp);
}
sched:::sleep
/pid == $target /
{
  self->sobj = (curlwpsinfo->pr_stype == SOBJ_MUTEX ?
      "kernel mutex" : curlwpsinfo->pr_stype == SOBJ_RWLOCK ?
      "kernel RW lock" : curlwpsinfo->pr_stype == SOBJ_CV ?
      "cond var" : curlwpsinfo->pr_stype == SOBJ_SEMA ?
      "kernel semaphore" : curlwpsinfo->pr_stype == SOBJ_USER ?
      "user-level lock" : curlwpsinfo->pr_stype == SOBJ_USER_PI ?
      "user-level PI lock" : curlwpsinfo->pr_stype == SOBJ_SHUTTLE ?
      "shuttle" : "unknown");
  self->delta = (walltimestamp - self->stamp) /scale;
  self->stamp = walltimestamp;
  self->stamp = (walltimestamp - baseline) / scale;
  printf("%9d:%-9d TID %3d sleeping on  %s \n",
      self->stamp, self->delta, tid, self->sobj);
  /* @sleep[curlwpsinfo->pr_stype, curlwpsinfo->pr_state, ustack()]=count(); ↩
       */
}
sched:::sleep
/ pid == $target && ( curlwpsinfo->pr_stype == SOBJ_CV ||
    curlwpsinfo->pr_stype == SOBJ_USER ||
    curlwpsinfo->pr_stype == SOBJ_USER_PI) /
{
  /*ustack()*/
}
sched:::sleep
/pid!=$pid && 0/
{
  @sleep[execname,curlwpsinfo->pr_stype, curlwpsinfo->pr_state, ustack()]=↩
      count();
}
```

O objetivo desta análise passa por analisar o comportamento da aplicação para os diferentes tipos de escalonamento:

- ***static*** - Este tipo de escalonamento divide o ciclo para *chunks* iguais ou o mais igual possível, caso o numero de iterações do ciclo não seja divisível por o número de *Threads* multiplicado pelo tamanho do *chunk*.
- ***dynamic*** - Com este escalonamento é distribuído *chunck's*, com tamanho fixo, pelo numero de *threads* e estes segmentos são organizados numa *queue*. Quando um segmento termina é atribuido outro segmento com o mesmo tamanho. Este tipo de escalonamento tem um *overhead* associado.
- ***guided*** - Este tipo de escalonamento é semelhante ao *dynamic*, contudo o tamanho do *chunk* começa grande e diminuindo de forma a melhorar o balanceamento das cargas entre as iterações. Por defeito o tamanho do *chunk* é aproximadamente $loop\_count/number\_of\_threads$.

Para a execução e teste da aplicação criei uma *shell script* em que testei todos os tipos de escalonamento (*static, guided, dynamic*) para um numero variável de *Threads* (1, 2, 4, 8, 16, 32 e 64), recolhendo para cada número de *threads* um total de 10 amostras, das quais escolhi a amostra com melhor tempo.

5.1. Análise dos Tempos para os Diferentes Tipos de Escalonamento

No gráfico da figura 6, podemos consultar os diferentes tempos para os diferentes tipos de escalonamento para diferentes números de *threads*.

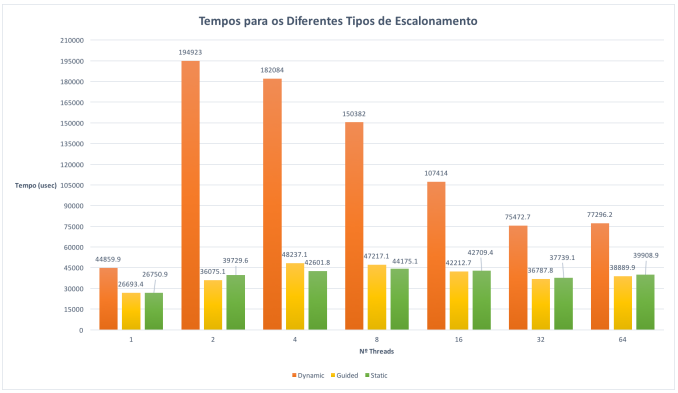

Figura 6. Tempos para os Diferentes Tipos de Escalonamento para Diferentes Nº de Threads

Ao analisarmos o gráfico, verificamos que o escalonamento *dynamic* é sempre o que apresenta piores tempos, isto pode ser justificado, pelo facto deste escalonamento ter um *overhead* associado. Este *overhead* está associado ao tempo que é necessário para se fazer as divisões das iterações em blocos e posteriormente organiza-las na *queue*.

Quanto aos outros dois tipos de escalonamento (*guided, static*), estes apresentam tempos relativamente iguais à medida que se aumenta o número de *threads*. Como tal não me é possível concluir qual a melhor política de escalonamento em termos de tempo, uma vez que estes tempos são semelhantes para o *static* e para o *guided*.

Relativamente aos resultados obtidos para 64 *threads*, tenho de admitir que os resultados, são surpreendentes, uma vez que quando a aplicação é executada, estamos a usar o dobro das *threads* disponíveis na máquina. Como tal, seria de esperar que os tempos fossem um pouco superiores, uma vez que a máquina está a usar *threads* partilhadas, e como podemos verificar pela a análise do gráfico da figura 6, vemos que os tempos continuam mais ao menos semelhantes de quando a aplicação é executada com 32 *threads*.

Fazendo ainda uma análise ao paralelismo da aplicação, podemos verificar que é com a versão sequencial (1 *thread*)desta que se obtém o melhor tempo, relativamente

ás versões com um maior número de *threads*. O que permite concluir que este código não é o melhor código para ser paralelizado uma vez que o melhor tempo é obtido sequêncialmente.

5.2. Análise do Comportamento das Threads para os Diferentes Tipos de Escalonamento

Para a análise dos resultados, escolhi analisar um excerto do output da *Script Dtrace* da melhor execução, para 32 *threads* para o escalonamento *static* com um intervalo de 100. Escolhi 32 *threads* porque é o número máximo de *threads* suportado pela máquina.

```
 0:0          TID    1 CPU    1(1) created
        0:0          TID    1 CPU    1(1) created
fiosExecucao =32 Intervalo=99
 tempo -> 37739.1 usecs
        9:0          TID    3 CPU    0(1) created
        9:1465575350619 TID    3 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID    3 sleeping on  cond var
        9:1465575350619 TID    3 CPU    0(1) preempted
        9:0          TID    7 CPU    0(1) created
        9:1465575350619 TID    7 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID    7 sleeping on  cond var
        9:1465575350619 TID    7 CPU    0(1) preempted
        9:0          TID    9 CPU    0(1) created
        9:1465575350619 TID    9 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID    9 sleeping on  cond var
        9:1465575350619 TID    9 CPU    0(1) preempted
        9:0          TID   11 CPU    0(1) created
        9:1465575350619 TID   11 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   11 sleeping on  cond var
        9:1465575350619 TID   11 CPU    0(1) preempted
        9:0          TID   13 CPU    0(1) created
        9:1465575350619 TID   13 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   13 sleeping on  cond var
        9:1465575350619 TID   13 CPU    0(1) preempted
        9:0          TID   15 CPU    0(1) created
        9:1465575350619 TID   15 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   15 sleeping on  cond var
        9:1465575350619 TID   15 CPU    0(1) preempted
        9:0          TID   17 CPU    0(1) created
        9:1465575350619 TID   17 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   17 sleeping on  cond var
        9:1465575350619 TID   17 CPU    0(1) preempted
        9:0          TID   19 CPU    0(1) created
        9:1465575350619 TID   19 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   19 sleeping on  cond var
        9:1465575350619 TID   19 CPU    0(1) preempted
        9:0          TID   21 CPU    0(1) created
        9:1465575350619 TID   21 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   21 sleeping on  cond var
        9:1465575350619 TID   21 CPU    0(1) preempted
        9:0          TID   23 CPU    0(1) created
        9:1465575350619 TID   23 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   23 sleeping on  cond var
        9:1465575350619 TID   23 CPU    0(1) preempted
        9:0          TID   25 CPU    0(1) created
        9:1465575350619 TID   25 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   25 sleeping on  cond var
        9:1465575350619 TID   25 CPU    0(1) preempted
        9:0          TID   27 CPU    0(1) created
        9:1465575350619 TID   27 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   27 sleeping on  cond var
        9:1465575350619 TID   27 CPU    0(1) preempted
        9:0          TID   29 CPU    0(1) created
        9:1465575350619 TID   29 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   29 sleeping on  cond var
        9:1465575350619 TID   29 CPU    0(1) preempted
        9:0          TID   31 CPU    0(1) created
        9:1465575350619 TID   31 CPU    0(1) restarted on the same CPU
        9:1465575350619 TID   31 sleeping on  cond var
        9:1465575350619 TID   31 CPU    0(1) preempted
        9:1465575350619 TID    7 CPU    0(1) restarted on the same CPU
       39:1465575350649 TID    7 sleeping on  cond var
       39:1465575350649 TID    7 CPU    0(1) preempted
       49:1465575350659 TID    7 CPU    0(1) restarted on the same CPU
       49:1465575350659 TID    7 sleeping on  cond var
       49:1465575350659 TID    7 CPU    0(1) preempted
       49:1465575350659 TID    7 CPU    0(1) restarted on the same CPU
       49:1465575350659 TID    1  from-CPU 19(1) to-cpu 0(1) CPU migration
       49:1465575350659 TID    1 CPU    0(1) restarted on the same CPU
       49:1465575350659 TID    1 sleeping on  cond var
       49:1465575350659 TID    1 CPU    0(1) preempted
       49:1465575350659 TID    1 CPU    0(1) restarted on the same CPU
       49:1465575350659 TID    1 sleeping on  cond var
       49:1465575350659 TID    1 CPU    0(1) preempted
       49:1465575350659 TID    1 CPU    0(1) restarted on the same CPU
       49:1465575350659 TID    1 sleeping on  cond var
       49:1465575350659 TID    1 CPU    0(1) preempted
```

Escolhi apenas este excerto do output para o escalonamento *static* para 32 *threads*, uma vez, que depois da analise dos outros outputs não verifica grandes diferenças.

Como podemos verificar pelo output em cima, as *threads* são criadas, em diferentes CPU's e à medida que o programa é executado, estas por sua vez podem ser interrompidas (*prempted*), ou seja saem do CPU, depois de interrompidas, as *threads* normalmente recomeçam a sua execução no mesmo CPU (*restarted on the same CPU*).

As *threads* podem ficar adormecidas (*sleeping*), isto acontece porque são forçadas a parar pelas variáveis de condição (*sleeping on 'cond var'* no caso do output apresentado) ficando a espera que haja uma alteração na variável de condição. Estas também podem ser forçadas a parar por *mutex* e a semáforos.

No output apresentado, também podemos verificar que por várias vezes as *threads* podem migrar de um CPU para outro (*from-CPU 19(1) to-cpu 0(1) CPU migration*).

Como o tipo de escalonamento é *static*, sempre que termina um segmento associado a essa *thread* logo a seguir é atribuído um novo segmento para execução, segmentos esses que têm sempre o mesmo tamanho como já foi referido anteriormente.

Podemos analisar que o aumento do tempo total para a conclusão da aplicação não está directamente associado a tempo "perdido"em *sleep* por, p.e. variáveis de condição ou desafectação forçada. É a própria biblioteca OpenMP que implica uma adição em termos de *overhead* em tempo de computação – neste traçado incluído nas linhas *restarted on the same CPU*, ou seja, CPU-Time. Não nos é possível distinguir o CPU-TIME despendido em porções de código da aplicação ou na biblioteca OpenMP.

6. Conclusão

Como podemos verificar, ao longo deste trabalho, tivemos desenvolver duas scripts que nos permiti-se traçar/detectar o que nos foi pedido no enunciado. Ao longo do desenvolvimento apercebi-me cada vez mais das capacidades da *Framework Dtrace*, sendo que esta ferramenta nos permite, monitorizar ao pormenor tudo o que acontece num sistema. Para além desta grande vantagem, esta ferramenta também apresenta uma linguagem própria, a linguagem *D*, que como já foi referido anteriormente, é uma linguagem muito parecida (estruturalmente) com o *awk*.

Com esta linguagem, a *Framework* torna-se ainda muito mais "poderosa", uma vez que nos permite aproximar até ao máximo detalhe daquilo que pretendemos analisar no sistema.

Em relação ao trabalho extra, sugerido pelo professor no final do semestre, podemos verificar que com uma simples *script Dtrace* podemos fazer um traçado dinâmico do comportamento de todas as *threads* envolvidas na execução da aplicação paralela, para os diferentes tipos de escalonamento. Com esta *script* temos total controlo do percurso de cada *thread* ao longo da toda a execução. Uma vez mais, com este exemplo, podemos apercebermo-nos da potencia-

lidade e utilidade da ferramenta *DTrace*, quer para versões sequências quer para versões paralelas de código.

Como trabalho futuro, pretendo aperfeiçoar os meus conhecimentos em *Dtrace*, porque, como já disse, para além de ser uma ferramenta poderosa, no que toca a administração de sistemas, também é uma ferramenta muito útil e acessível uma vez que a sua sintaxe não é muito difícil nem a sua estrutura, o que se torna difícil no estudo desta ferramenta é a numerosa informação que se dispõem, bem como as numerosas provas que é permitido se criar com o *Dtrace*.