Exemplo

Programação de algoritmos paralelizáveis com *threads*

Cálculo do valor de PI com o método Monte Carlo

(demonstração do impacto na performance)

Este exemplo:

- **Utiliza várias threads para calcular o valor de PI**
  - **O valor de PI, pelo método Monte Carlo, é obtido pela razão entre o número total de dardos que são atirados sobre um quadrado e aqueles que caem sobre um círculo inscrito nesse quadrado**
    - Os dados são simulados sorteando as suas coordenadas x e y
    - Métodos de monte Carlo: https://en.wikipedia.org/wiki/Monte_Carlo_method
  - **Quantos mais dardos, mais casas decimais estarão corretas**
    - Desde que o gerador de número aleatórios seja, de facto, uniforme
  - **É necessário um número muito grande de dardos**
  - **Para otimizar a performance, lançam-se várias threads (numero parametrizável)**
    - Cada thread lança ("simula") dardos
    - No final, usam-se todos os dardos lançados por todas as threads
  - **Enquanto o número de threads não ultrapassar o número de núcleos disponíveis ganha-se performance por cada nova thread utilizada**
    - **Isto é visível experimentalmente:**
      - a performance (número total de dardos lançados na mesma quantidade de tempo)
      - a utilização intensiva dos núcleos de processador, tantos quantas as threads em uso (programa htop)
      - Presume-se que o sistema não está a ocupar núcleos cm outras coisas alheias a este programa
    - **Se as threads ultrapassarem o número de núcleos disponíveis, então já não se ganha performance, eventualmente até se perde (maior competição entre as threads)**

Estratégia de implementação usada:

- Todas as threads são baseadas na mesma função (é o mesmo algoritmo, apenas os dados variam)
    - É passado um ponteiro para as threads de forma a que cada uma saiba o que deve fazer (estratégia habitual)

- A thread inicial, a correr a função main, lança as threads e depois aguarda um intervalo de tempo. Findo esse intervalo indica às threads que devem terminar
    - O número de threads é indicado em linha de comandos
    - O intervalo de tempo é indicado em linha de comandos
    - As threads são terminadas pela estratégia habitual "variável continua"

- A função main aguarda que as threads tenham garantidamente terminado, e só então é que vai verificar os totais de dardos que cada thread produziu
    - pthread_join

Ficheiros *header* envolvidos

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <unistd.h>
#include <locale.h>
```

Protótipo para a função da thread

```
void * calculaPI(void *);
```

Estrutura de dados para controlar as threads

```
typedef struct {
   pthread_t tid;
   int continua;
   int total, dentro;
} TDADOS;
```

Constantes simbólica: número máximo de threads

```
#define MAXIMUMT 20
```

Função main

1. **Obtém a configuração de tempo e número de threads**
2. **Lança as threads**
3. **Aguarda o tempo especificado**
4. **Indica às threads para terminarem findo o tempo especificado**
5. **Aguarda que as threads realmente tenham terminado**
6. **Recolhe a informação de dardos lançados e dardos dentro do círculo**
7. **Apresenta o valor de PI e termina**

```
int main(int argc, char * argv[]) {
   int NTHREADS, NSECS;
   TDADOS workers[MAXIMUMT];
   int i;
   int allDentro = 0, allTotal = 0;
   double allPI = 0;
```

1 - Obtém dados da linha de comandos: intervalo de tempo e número de threads a usar

```
   if (argc<3) {
       printf("\n%s segundos threads\n", argv[0]);
      return 1;
   }

   NSECS = atoi(argv[1]);
   NTHREADS = atoi(argv[2]);

   printf("\nNum segundos = %d", NSECS);
   printf("\nNum Threads = %d", NTHREADS);
   printf("\n");
```

2 - Lança as threads

```
   for (i = 0; i< NTHREADS; i++) {
       workers[i].continua = 1;
       pthread_create(& workers[i].tid, NULL, calculaPI, workers + i);
   }
```

3 - Aguarda o tempo especificado

```
   // aguarda N segundos
   printf("\nA aguardar %d segundos\n", NSECS);
   sleep(NSECS);
```

4 - Indica às threads para terminarem findo o tempo especificado

```
   // informa threads devem parar;
   for (i=0; i< NTHREADS; ++i)
      workers[i].continua = 0;
   printf("\nThreads informadas para terminar");
```

5 e 6 - Aguarda que as threads realmente tenham terminado, obtendo os valores produzidos

```
   // aguarda threads realmente terminarem
   for (i=0; i<NTHREADS; ++i) {
      pthread_join(workers[i].tid, NULL);
      printf("\n%d -> dentro = %d, total = %d, PI = %f",
            i,
            workers[i].dentro, workers[i].total,
            4.0 * workers[i].dentro / workers[i].total);
      allDentro += workers[i].dentro;
      allTotal += workers[i].total;
   }
```

7 - Apresenta o valor de PI e termina

```
   allPI = 4.0 * allDentro / allTotal;
   printf("\n\n PI global = %f", allPI);
   setlocale(LC_NUMERIC, "");
   printf("\n\n Dardos total = %'d", allTotal);
   printf("\n\n");
   return 0;
}
```

Função da thread

Encontra-se em ciclo (controlado pela estratégia "variável continua") sorteando dardos. No final passa os valores totais para variáveis que a função main poderá consultar

srand_r() e **drand_r** --> ***_r*** indica que é uma função thread safe. ***drand48*** segue uma distribuição uniforme melhor que srand() / *rand()*.

```
void * calculaPI(void * p) {
   TDADOS * mydados = (TDADOS *) p;
   double x,y;
   int dentro = 0, total = 0;
   // inicializa randomize (rand e rand_r estão deprecated)
   struct drand48_data randBuffer;
   srand48_r(time(NULL), &randBuffer);
```

```
   while (mydados->continua) {
      drand48_r(&randBuffer, &x);  // 0 <= x >= 1
      drand48_r(&randBuffer, &y);  // 0 <= y >= 1
      if (x*x + y*y <= 1)
         ++dentro;
      ++total;
   }
   mydados->total = total ;
   mydados->dentro = dentro ;
   return NULL;
}
```

Nota. Caso tenha um cenário semelhante no trabalho prático deverá procurar aplicar o conhecimento produzindo código seu. Não faça meros decalques código de exemplos dado que essa prática não é bem vista no momento da avaliação.

Experiêcia:

Ocupação intensiva de 4 núcleos do processador quando se especificam 4 threads (figura)