Exemplo de programação com threads

Cálculo do valor de PI pelo método Montecarlo

Existem dois cenários típicos onde o uso de threads é vantajoso: i) a necessidade de um programa efetuar várias tarefas em simultâneo, e ii) a execução de um algoritmo que é paralelizável e se pretende obter mais desempenho aproveitando a existência de diversos núcleos/processadores. Este exemplo encaixa na segunda situação.

Este exemplo calcula o valor aproximado de PI a algumas casas decimais usando o método de Montecarlo. Este obtém o valor de PI simulando o lançamento de dardos num quadrado que contém um círculo. Os dados ficam em coordenadas aleatórias dentro do quadrado: alguns dados ficam no quadrado e dentro do círculo, outro dentro do quadrado, mas fora do círculo. O valor de PI é dado em função da razão entre a área do círculo e a área do quadrado; neste caso, assumindo uma distribuição uniforme das coordenadas dos dardos simulados, o valor de PI é dado em função da proporção de dardos que ficam dentro do círculo e do número total de dardos lançados.

O valor obtido é tão mais exato quando maior for o número de dardos. Interessa simular o maior número de dardos por unidade de tempo. Desta forma empregam-se threads para trabalhar em paralelo e simular mais dardos na mesma quantidade de tempo.

O programa permite escolher, por argumentos de linha de comandos, o tempo desejado e o número de threads. O seu uso permite observar, na prática, os assuntos teóricos debatidos nas aulas quanto à circunstâncias em que as threads permitem obter mais desempenho, e as circunstâncias em que o uso de threads adicionais deixa de trazer ganhos de performance. Sugere-se que a execução deste programa seja acompanhada da execução de um utilitário que permita ver a ocupação do processador (sugestão: htop).

Algumas observações quanto à implementação:

- Este programa usa tipos de dados nativos da linguagem C (double). Estes tipos de dados estão, naturalmente, limitados quando ao número de casas decimais que conseguem descrever. Independentemente de maios ou menor número de dardos simulados, o valor de PI nuca passará de uma determinada exatidão máxima. Interessa também saber que o valor obtido é condicionado por flutuações estatísticas dado que o algoritmo se baseia em valores aleatórios.
- Os computadores modernos são tão rápidos que rapidamente esgotam a capacidade de precisão dos tipos de dados nativos das linguagens de programação. Usar mais ou menos threads, num computados vulgar moderno não tem efeito visível, dado que uma única thread é suficiente para atingir a capacidade máxima de precisão de uma variável do tipo double. Assim, este programa introduz um compasso de espera artificial, que é configurável por linha de comandos (função simulaMaisLento). Desta forma consegue-se observar experimentalmente a influência de mais ou menos threads.
- São usados contadores de dardos do tipo long long int. Tendo em atenção o recurso ao compasso de espera artificialmente introduzido, um int seria suficiente. No entanto, optou-se por deixar permanecer o tipo lon long int para o caso de se remover o compasso de espera artificial.
- Este exemplo não abarca questões de sincronização, sendo o único aspeto desse tema a espera pela terminação de threads com pthread_join.

O código tem um nível de comentários moderado dado que foi explicado nas aulas.

Código a partir da página seguinte.

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


void * calculaPI(void *);

/* estrutura de dados para controlar as threads */

typedef struct {
   pthread_t tid;
   int continua;
   unsigned long long int total, dentro;
   int slowdown;
   int seed;
} TDADOS;

#define MAXIMUMT 20

/*
Função main
1.    Obtém a configuração de tempo e número de threads e fator slowdown
2.    Prepara e 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, SLOWDOWN;
              // as variáveis acima: apesar das maiusculas, não são const simbólicas
   TDADOS workers[MAXIMUMT];
   int i;
```

```
   unsigned long long int allDentro = 0, allTotal = 0;
   long double allPI = 0;
   long double localPI;

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

   if (argc<4) {
       printf("\n%s segundos threads slowdown (0=nenhum)\n", argv[0]);
      return 1;
   }

   NSECS = atoi(argv[1]);      // estas conversões
   NTHREADS = atoi(argv[2]);   // deviam ser todas verificadas
   SLOWDOWN = atoi(argv[3]);   // (fica para TPC)
   // Estes valores deviam ser sanitizados (-> TPC tb)

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

// 2 - Prepara e lança as threads

   for (i = 0; i< NTHREADS; i++) {
       workers[i].continua = 1;
       workers[i].seed = i;  // cada thread tem um i diferente (seed p/random)
       workers[i].slowdown = SLOWDOWN;
       // restantes dados são inicializados na thread
       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);
      localPI = (long double) 4.0 * workers[i].dentro / workers[i].total;
      printf("\n%d -> dentro = %llu, total = %llu, PI = %Lf",
            i,
            workers[i].dentro, workers[i].total,
            localPI);
      allDentro += workers[i].dentro;
      allTotal += workers[i].total;
   }

// 7 - Apresenta o valor de PI e termina

   allPI = (long double) 4.0 * allDentro / allTotal;
   printf("\n\n PI global = %Lf", allPI);
   setlocale(LC_NUMERIC, "");
   printf("\n\n Dardos total = %llu", 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 (reentrante).
// drand48 segue uma distribuição uniforme melhor que srand() / rand().
// a inicialização da seed com time pode não ser suficiente para distinguir
// as sequências pseudoaleatórias em cada uma das threads (umas das outras)
// porque todas irão passar pela chamada a time() em momentos
// muito próximos umas das outras) (eventualmente obtendo o mesmo valor)
// Por isso usa-se o inteiro i passado à thread como seed para as distinguir


void simulaMaisLento();

void * calculaPI(void * p) {
   TDADOS * mydados = (TDADOS *) p;
   double x,y;
   unsigned long long int dentro = 0, total = 0;
   // inicializa randomize (rand e rand_r estão deprecated)
   struct drand48_data randBuffer;
   srand48_r(time(NULL) + mydados->seed * 123000, &randBuffer);

   while (mydados->continua) {
      drand48_r(&randBuffer, &x);  // 0 <= x >= 1
      drand48_r(&randBuffer, &y);  // 0 <= y >= 1
      // if (sqrt(x*x + y*y) <= sqrt(1)) ... sqrt pode ser omitido
      if (x*x + y*y <= 1)
         ++dentro;
      ++total;
      // simula lento
      for (int i = 0; i< mydados->slowdown; ++i)
         simulaMaisLento();
   }
   mydados->total = total ;
   mydados->dentro = dentro ;
   return NULL;
}
```

```
// o PC é demasiado rápido para perceber diferencas de qualidade
// no valor de PI
// -> simnular um calculo mais lento perdendo tempo em cada iteracao
//    a funcoa simulaMaisLento() gasta tempo de processamento
//    . usleep também poderia ser usado mas sleep não ocupa processador
//      e pretende-se mostrar um ou mais nucleos da CPU ocupados
// valores dos ciclos ajustados manualmente por experimentação
// esta função é chamada 0 ou mais vezes consoante o valor passado
//   na linha de argumentos

void simulaMaisLento() {
    int a=0, b=0;
    for (int i=1; i<=100; ++i)
       for (int j=1; j<=1000; ++j)
          a += j*i % (j+i);  // contas sem significado para perder algum tempo
    b = a;
}


// Usar este programa para fazer as seguintes experiências
// - Ocupação de 1 núcleo por thread lançada
// - Ganho em exartidão do valor obtido sem custo adicional de tempo
//      enquanto não se esgotam os núcleos disponíveis
// - Estabilização da qualidade do valor (para o mesmo tempo) mesmo
//    que se aumentem as threads após se terem esgortado os
//    núcleos disponíveis
// Importa saber quantos núcleos existem e ver a ocupação destes
//   comando htop   (se não estiver instalado; sudo apt install htop)
```