Grupo 4 - Tiago Koji Castro Shibata (8988730), Adilson Torres Gregório de Souza (8988254)

Sistemas de Computação de Alto Desempenho

Aula 3

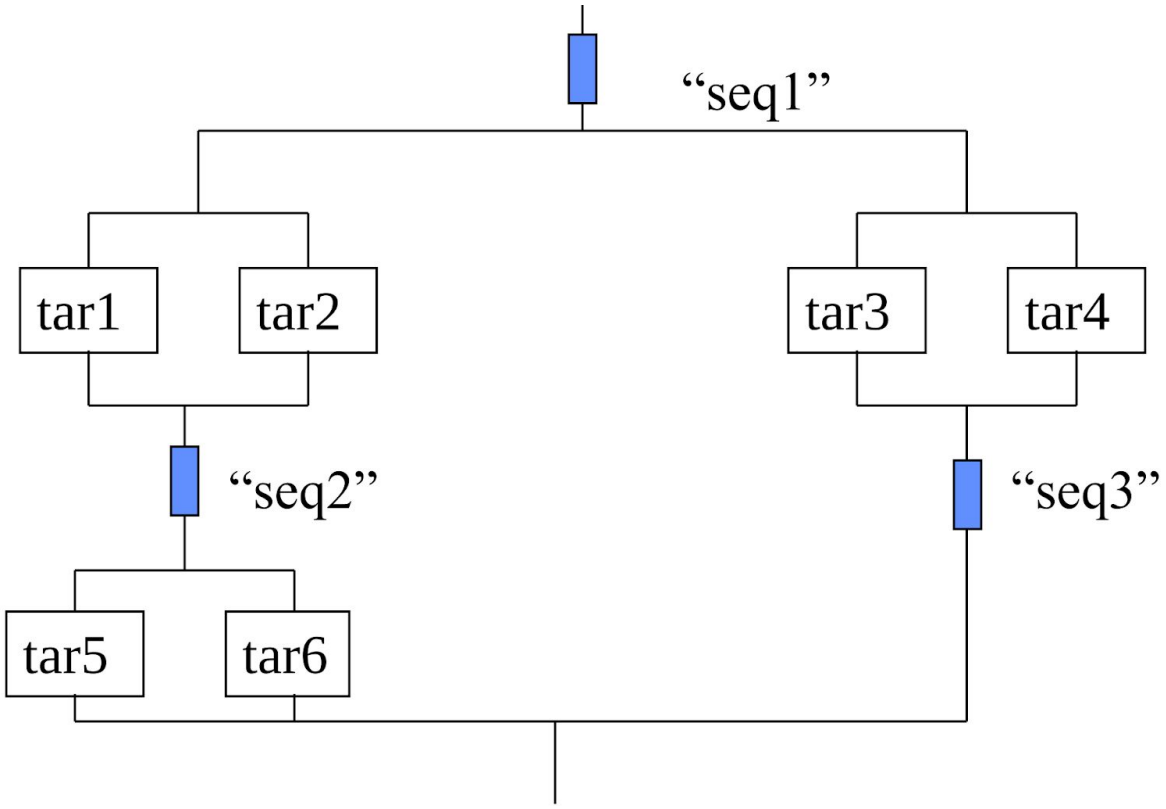
Relatório: Paralelização com CPAR

- **Introdução**

Em aula, estudamos a linguagem CPAR, uma extensão de C com suporte a estruturas paralelas. No laboratório, implementamos alguns programas em CPAR e comparamos o desempenho.

- **Programas**

No **exercício 1**, implementamos as seguintes dependências entre tasks :



Em cada tarefa, chamamos *printf* com o nome da tarefa:

```
#include <stdio.h>

task spec task1();
task spec task2();
task spec task3();
task spec task4();
task spec task5();
task spec task6();

task body task1() {
```

```
    printf("tar1\n");
}

task body task2() {
    printf("tar2\n");
}

task body task3() {
    printf("tar3\n");
}

task body task4() {
    printf("tar4\n");
}

task body task5() {
    printf("tar5\n");
}

task body task6() {
    printf("tar6\n");
}

int main() {
    printf("seq1\n");
    cobegin
        create 1, task1();
        create 1, task2();
        wait_proc(task1);
        wait_proc(task2);
        printf("seq2\n");
        create 1, task5();
        create 1, task6();
        wait_proc(task5);
        wait_proc(task6);
    also
        create 1, task3();
        create 1, task4();
        wait_proc(task3);
        wait_proc(task4);
        printf("seq3\n");
    coend
}
```

Executamos o programa algumas vezes. Algumas saídas obtidas foram:

```
$ ./ex1
seq1
tar1
tar2
seq2
tar3
tar5
tar4
seq3
tar6
$ ./ex1
seq1
tar1
tar2
tar3
seq2
tar4
tar5
seq3
tar6
```

Observamos que a ordem das tarefas muda, mas as dependências sempre são respeitadas.

No **exercício 2**, implementamos uma versão sequencial e paralela de soma e multiplicação de matrizes. A operação implementada com matrizes 100x100 foi A * B + C * D. Inserimos chamadas a *clock_gettime* para analizar o desempenho:

```c
#include <stdio.h>
#include <time.h>

double a[100][100], b[100][100], c[100][100], d[100][100];
double a_b[100][100], c_d[100][100], result[100][100];

double time_difference(struct timespec *start, struct timespec *end) {
    return (end->tv_sec * 1000 + 1e-6 * end->tv_nsec) -
        (start->tv_sec * 1000 + 1e-6 * start->tv_nsec);
}

int main() {
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            a[i][j] = i + j;
        }
    }
    for (int i = 0; i < 100; i++) {
```

```
        for (int j = 0; j < 100; j++) {
            b[i][j] = i + 2 * j;
        }
    }
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            c[i][j] = 2 * i + 3 * j;
        }
    }
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            d[i][j] = 2 * i + j;
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("Initialization time: %.5fms\n", time_difference(&start_time, &end_time));

    clock_gettime(CLOCK_MONOTONIC, &start_time);
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            a_b[i][j] = 0;
            for (int k = 0; k < 100; k++) {
                a_b[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            c_d[i][j] = 0;
            for (int k = 0; k < 100; k++) {
                c_d[i][j] += c[i][k] * d[k][j];
            }
        }
    }
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            result[i][j] = a_b[i][j] + c_d[i][j];
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("Processing time: %.5fms\n", time_difference(&start_time, &end_time));
    printf("[0][0] = %f, [50][50] = %f, [99][99] = %f\n",
        result[0][0], result[50][50], result[99][99]);
}
```

Obtivemos os seguintes resultados:

```
$ ./ex2_serial
Initialization time: 0.17251ms
```

```
Processing time: 2.49590ms
[0][0] = 2298450.000000, [50][50] = 5773450.000000, [99][99] = 11119350.000000
```

Implementamos uma versão paralela do programa:

```
#include <stdio.h>
#include "mede_time.h"

shared double a[100][100], b[100][100], c[100][100], d[100][100];
shared double a_b[100][100], c_d[100][100], result[100][100];

task spec initialize_a();
task spec initialize_b();
task spec initialize_c();
task spec initialize_d();
task spec mult_a_b();
task spec mult_c_d();
task spec add();

task body initialize_a() {
    int i;
    forall i = 0 to 99 {
        for (int j = 0; j < 100; j++) {
            a[i][j] = i + j;
        }
    }
}

task body initialize_b() {
    int i;
    forall i = 0 to 99 {
        for (int j = 0; j < 100; j++) {
            b[i][j] = i + 2 * j;
        }
    }
}

task body initialize_c() {
    int i;
    forall i = 0 to 99 {
        for (int j = 0; j < 100; j++) {
            c[i][j] = 2 * i + 3 * j;
        }
    }
}

task body initialize_d() {
    int i;
    forall i = 0 to 99 {
```

```
        for (int j = 0; j < 100; j++) {
            d[i][j] = 2 * i + j;
        }
    }
}

task body mult_a_b() {
    int i;
    forall i = 0 to 99 {
        for (int j = 0; j < 100; j++) {
            a_b[i][j] = 0;
            for (int k = 0; k < 100; k++) {
                a_b[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

task body mult_c_d() {
    int i;
    forall i = 0 to 99 {
        for (int j = 0; j < 100; j++) {
            c_d[i][j] = 0;
            for (int k = 0; k < 100; k++) {
                c_d[i][j] += c[i][k] * d[k][j];
            }
        }
    }
}

task body add() {
    int i;
    forall i = 0 to 99 {
        for (int j = 0; j < 100; j++) {
            result[i][j] = a_b[i][j] + c_d[i][j];
        }
    }
}

int main() {
    TIMER_START;
    create 4, initialize_a();
    create 4, initialize_b();
    create 4, initialize_c();
    create 4, initialize_d();
    wait_all();
    TIMER_STOP;
    printf("Initialization time: %.5fms\n", 1000 * TIMER_ELAPSED);
    TIMER_START;
    create 4, mult_a_b();
```

```
    create 4, mult_c_d();
    wait_all();
    create 4, add();
    wait_all();
    TIMER_STOP;
    printf("Processing time: %.5fms\n", 1000 * TIMER_ELAPSED);
         printf("[0][0] = %f, [50][50] = %f, [99][99] = %f\n", result[0][0],
result[50][50], result[99][99]);
 }
```

Realizamos algumas execuções, e obtivemos resultados bastante próximos:

```
$ ./ex2
Initialization time: 1.23600ms
Processing time: 1.08200ms
[0][0] = 2298450.000000, [50][50] = 5773450.000000, [99][99] = 11119350.000000
```

Observamos que a inicialização, por ser mais simples, é mais rápida no programa serial, que não possui overhead de criação de *threads*. O processamento é bastante mais rápido no programa paralelo:

Etapa	Sequencial (ms)	Paralelo (ms)
Inicialização	0.173	1.236
Processamento	2.496	1.082

No **exercício 3**, realizamos a inicialização de uma matriz 1000x1000, com o objetivo de realizar a busca de um dado elemento (de entrada) e mostrar suas posições no vetor, utilizando 4 processadores:

```
#include <stdio.h>
#include <time.h>
#include "mede_time.h"

shared int a[1000][1000];
shared int valor_busca;

task spec init_vetor();
task body init_vetor()
{
  int i, j;
  forall i=0 to 999
  {
    for (j=0; j<1000; j++)
```

```
    {
      a[i][j] = i + j;
    }
  }
}

task spec busca_vetor();
task body busca_vetor()
{
  int i, j;
  forall i=0 to 999
  {
    for (j=0; j<1000; j++)
    {
      if (a[i][j] == valor_busca)
      {
        printf("econtrado posicao %d %d\n", i, j);
        fflush(stdout);
      }
    }
  }
}

main()
{
  int elemento;

  printf("Busca um elemento no vetor.\n");
  TIMER_CLEAR;
  TIMER_START;
  create 4, init_vetor();
  wait_proc(init_vetor);
  TIMER_STOP;
  printf("Vetor inicializado, Time: %.5fms\n", 1000 * TIMER_ELAPSED);
  printf("Informe um elemento: ");
  scanf("%d", &valor_busca);
  TIMER_CLEAR;
  TIMER_START;
  create 4, busca_vetor();
  wait_proc(busca_vetor);
  TIMER_STOP;
  printf("Busca finalizada, Time: %.5fms\n", 1000 * TIMER_ELAPSED);
}
```

Também foi implementado uma versão do exercicio 3, sequencial:

```
#include <stdio.h>
#include <time.h>
```

```c
int a[1000][1000];
int valor_busca;

void init_vetor()
{
  int i,j;
  for(i=0; i<1000; i++)
  {
    for(j=0; j<1000; j++)
    {
      a[i][j] = i + j;
    }
  }
}

void busca_vetor()
{
 int i,j;
  for(i=0; i<1000; i++)
  {
    for(j=0; j<1000; j++)
    {
      if(a[i][j] == valor_busca)
      {
        printf("encontrado posicao %d %d\n", i, j);
        fflush(stdout);
      }
    }
  }
}

int main(){
  struct timespec start_time, end_time;

  printf("Busca um elemento no vetor.\n");
  clock_gettime(CLOCK_MONOTONIC, &start_time);
  init_vetor();
  clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("Vetor inicializado, Time: %.5fms\n", (end_time.tv_sec * 1000 + 1e-6 *
end_time.tv_nsec) - (start_time.tv_sec * 1000 + 1e-6 * start_time.tv_nsec));
  printf("Informe um elemento: ");
  scanf("%d", &valor_busca);
  clock_gettime(CLOCK_MONOTONIC, &start_time);
  busca_vetor();
  clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("Busca finalizada, Time: %.5fms\n", (end_time.tv_sec * 1000 + 1e-6 *
end_time.tv_nsec) - (start_time.tv_sec * 1000 + 1e-6 * start_time.tv_nsec));
}
```

Após a execução de cada programa (paralelo e sequencial), obtivemos os seguintes resultados

Etapa	Sequencial (ms)	Paralelo (ms)
Inicialização	2.632	1.219
Busca	1.719	1.123

Onde observamos que o programa paralelo obteve melhor desempenho tanto na inicialização quanto na busca.