Grupo 4 - Tiago Koji Castro Shibata (8988730)

Sistemas de Computação de Alto Desempenho
Aula 6

Relatório: OpenMP - Sections e locks

- **Introdução**

Em aula, estudamos o uso de OpenMP para paralelismo, com foco no uso de seções e locks. Em laboratório, implementamos dois programas.

- **Configuração do sistema**

Testes foram realizados em um processador Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz, com 4 núcleos. O sistema utilizado é Linux e os processos foram executados com *niceness* mínimo para que outros processos do sistema interferissem menos nos resultados.

Todos os programas foram compilados com *`-O3 -DNDEBUG -Wall -Wextra -pedantic`*.

- **Programas**

No **exercício 1**, implementamos a seguinte operação matricial: *`R = A*B + C*D`*. A, B, C, D e R são matrizes 1000x1000. Multiplicações foram realizadas em 2 seções. Em cada seção, a multiplicação foi feita com 2 threads, conforme o enunciado. A soma foi realizada e laço *for* (com paralelismo entre as linhas da matriz).

```c
#include <stdio.h>
#include <sys/sysinfo.h>
#include <time.h>

#include <omp.h>

#define SIZE 1000
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE], D[SIZE][SIZE];

void matrix_multiply(int result[SIZE][SIZE], int a[SIZE][SIZE], int b[SIZE][SIZE]) {
    #pragma omp parallel for num_threads(2)
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            result[i][j] = 0;
            for (int k = 0; k < SIZE; k++) {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

int main() {
    static int A_B[SIZE][SIZE], C_D[SIZE][SIZE], result[SIZE][SIZE];
    omp_set_nested(1);

    struct timespec start_time, end_time;
```

```
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    #pragma omp parallel sections
    {
        #pragma omp section
        matrix_multiply(A_B, A, B);
        #pragma omp section
        matrix_multiply(C_D, C, D);
    }

    #pragma omp parallel for
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            result[i][j] = A_B[i][j] + C_D[i][j];
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("[0][0] = %d, [SIZE / 2][SIZE / 2] = %d, [SIZE - 1][SIZE - 1] = %d\nTime:
%.5fms\n",
        result[0][0], result[SIZE / 2][SIZE / 2], result[SIZE - 1][SIZE - 1],
        (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec *
1000 + 1e-6 * start_time.tv_nsec));

    return 0;
}
```

Em OpenMP, por padrão, em regiões paralelas aninhadas apenas a região externa é paralelizada, e as internas não (essa função é desativada para evitar explosão de threads. Ver https://msdn.microsoft.com/en-us/library/sk3zt8e1.aspx: *because the number of threads increases exponentially when nesting parallel regions. For example a function that recurses 6 times with the number of OMP threads set to 4 requires 4,096 (4 to the power of 6) threads In general*). Por isso, chamei `omp_set_nested` no início do programa, que ativa regiões paralelas aninhadas.

Após 20 execuções, o menor tempo atingido pelo programa foi 2214.47ms. Aumentar o número de threads na operação de multiplicação piora o desempenho do programa (com 3, o tempo de execução subiu para 3220.67ms). Isso é experado pois o sistema possui 4 núcleos e, com 2 seções com 2 threads, há 4 threads em execução, e aumentar esse número irá saturar o sistema.

Supreendentemente, desativar totalmente o paralelismo interno às multiplicações (usando `num_threads(1)` na multiplicação ou removendo `omp_set_nested`) melhora o desempenho do programa para 761.25ms, mesmo que com apenas 2 threads. Isso provavelmente se deve ao comportamento mais previsível dos acessos seriais e melhor desempenho do cache de memória. Talvez em matrizes menores, que caibam inteiramente nos caches mais rápidos da CPU, o desempenho fosse diferente.

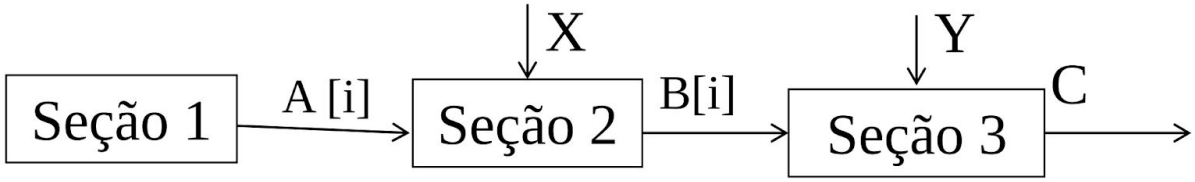
Uma versão serial do programa também foi feita:

```c
#include <stdio.h>
#include <sys/sysinfo.h>
#include <time.h>

#define SIZE 1000
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE], D[SIZE][SIZE];

void matrix_multiply(int result[SIZE][SIZE], int a[SIZE][SIZE], int b[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            result[i][j] = 0;
            for (int k = 0; k < SIZE; k++) {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

int main() {
    static int A_B[SIZE][SIZE], C_D[SIZE][SIZE], result[SIZE][SIZE];

    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    matrix_multiply(A_B, A, B);
    matrix_multiply(C_D, C, D);

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            result[i][j] = A_B[i][j] + C_D[i][j];
        }
    }

    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("[0][0] = %d, [SIZE / 2][SIZE / 2] = %d, [SIZE - 1][SIZE - 1] = %d\nTime:
%.5fms\n",
        result[0][0], result[SIZE / 2][SIZE / 2], result[SIZE - 1][SIZE - 1],
        (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec *
1000 + 1e-6 * start_time.tv_nsec));

    return 0;
}
```

Desempenho final:

Serial	2 regiões, 2 threads por multiplicação	2 regiões, 1 thread por multiplicação
1446.99ms	2214.47ms	761.25ms

No **exercício 2**, o seguinte pipeline de dados foi implementado:



Conforme cada seção gera dados, eles são passados para a próxima elemento a elemento. Foi usada uma estrutura produtor-consumidor entre as seções, com o uso de 4 locks:

```c
#include <stdio.h>
#include <sys/sysinfo.h>
#include <time.h>

#include <omp.h>

int A[1999], B[2000], C[2000], X[2000], Y[2000];
omp_lock_t a_empty, a_full, b_empty, b_full;

void a_worker() {
    for (int i = 0; i < 1999; i++) {
        omp_set_lock(&a_empty);
        A[i] = 3 * i + 15;
        omp_unset_lock(&a_full);
    }
}

void b_worker() {
    B[0] = 1;
    for (int i = 1; i < 2000; i++) {
        omp_set_lock(&a_full);
        omp_set_lock(&b_empty);
        B[i] = X[i] + A[i - 1];
        omp_unset_lock(&a_empty);
        omp_unset_lock(&b_full);
    }
}

void c_worker() {
    C[0] = 1;
    for (int i = 1; i < 2000; i++) {
        omp_set_lock(&b_full);
        C[i] = Y[i] + B[i - 1] * 2;
        omp_unset_lock(&b_empty);
    }
}

int main() {
    for (int i = 1; i < 2000; i++) {
        X[i] = i;
    }
    for (int i = 1; i < 2000; i++) {
        Y[i] = i + 1;
    }
    struct timespec start_time, end_time;
```

```
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    omp_init_lock(&a_empty);
    omp_init_lock(&b_empty);
    omp_init_lock(&a_full);
    omp_init_lock(&b_full);
    omp_set_lock(&a_full);
    omp_set_lock(&b_full);
    #pragma omp parallel sections
    {
        #pragma omp section
        a_worker();
        #pragma omp section
        b_worker();
        #pragma omp section
        c_worker();
    }
    omp_destroy_lock(&a_empty);
    omp_destroy_lock(&b_empty);
    omp_destroy_lock(&a_full);
    omp_destroy_lock(&b_full);

    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("C[0] = %d, C[1] = %d, C[1999] = %d\nTime: %.5fms\n",
        C[0], C[1], C[1999],
        (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec *
1000 + 1e-6 * start_time.tv_nsec));

    return 0;
}
```

O código executou em 3.14ms.