Grupo 4 - Tiago Koji Castro Shibata (8988730)

Sistemas de Computação de Alto Desempenho
Aula 2

Relatório: Paralelização com processos e threads de soma de matrizes

- **Introdução**

Em aula, estudamos estratégias de paralelização com processos e threads e comparamos vantagens e desvantagens. No laboratório prático, parelelizamos um programa de soma de matrizes e comparamos a performance em diferentes cenários paralelos.

- **Programas**

Iniciei com um programa puramente sequencial de soma de matrizes. O programa salva o tempo atual baseado em um *clock* monotônico, realiza a soma de matrizes com laços *for*, lê o tempo atual e mostra alguns resultados da soma e a diferença de tempo.

O programa simplificado (com inclusão de *headers* omitida) é:

```
#define SIZE 10000

double a[SIZE][SIZE];
double b[SIZE][SIZE];
double c[SIZE][SIZE];

void initialize() {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++){
            a[i][j] = 1.5;
            b[i][j] = 2.6;
        }
    }
}

int main() {
    initialize();

    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    for (int row = 0; row < SIZE; row++) {
        for (int column = 0; column < SIZE; column++) {
            c[row][column] = a[row][column] + b[row][column];
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);

     printf("c[0][0] = %f, c[SIZE - 1][SIZE - 1] = %f, c[SIZE / 2][SIZE / 2] =
%f\n",
```

```
        c[0][0],
        c[SIZE - 1][SIZE - 1],
        c[SIZE / 2][SIZE / 2]);
          printf("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));
}
```

Com o tamanho da matriz em 10000x10000, obti os seguintes resultados:

```
./matrix_add
c[0][0] = 4.100000, c[SIZE - 1][SIZE - 1] = 4.100000, c[SIZE /
2][SIZE / 2] = 4.100000
Time: 339.82562ms
```

Realizei múltiplas execuções e os resultados observados não oscliaram muito, ficando entre 337ms e 352ms.

Implementei uma versão usando processos. O número de processos usado foi o número de processadores lógicos da máquina, detectado com *get_nprocs*. Usei uma página de memória compartilhada e anônima usando *mmap*. Cada processo recebeu um conjunto contíguo de linhas da matrix:

```
#define SIZE 10000

typedef struct {
    double a[SIZE][SIZE];
    double b[SIZE][SIZE];
    double c[SIZE][SIZE];
} matrices_t;

void initialize(matrices_t *m) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++){
            m->a[i][j] = 1.5;
            m->b[i][j] = 2.6;
        }
    }
}

void child_process(matrices_t *m, int start, int end) {
    for (int row = start; row < end; row++) {
        for (int column = 0; column < SIZE; column++) {
            m->c[row][column] = m->a[row][column] + m->b[row][column];
        }
```

```
    }
    exit(0);
}

int main() {
    int nprocs = get_nprocs();
    printf("nprocs = %d\n", nprocs);

    matrices_t *matrices = mmap(NULL, sizeof(matrices_t), PROT_READ |
PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (matrices == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
    initialize(matrices);

    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    pid_t pids[nprocs];
    for (int p = 0; p < nprocs; p++) {
        pids[p] = fork();
        if (!pids[p]) {
            int start = (p * SIZE) / nprocs;
            int end = ((p + 1) * SIZE) / nprocs;
            child_process(matrices, start, end);
        }
    }

    for (int p = 0; p < nprocs; p++) {
        if (waitpid(pids[p], NULL, 0) == -1) {
            perror("waitpid");
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    printf("c[0][0] = %f, c[SIZE - 1][SIZE - 1] = %f, c[SIZE / 2][SIZE / 2] =
%f\n",
        matrices->c[0][0],
        matrices->c[SIZE - 1][SIZE - 1],
        matrices->c[SIZE / 2][SIZE / 2]);
    munmap(matrices, sizeof(matrices_t));
    printf("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));
}
```

Resultados:

```
nprocs = 4
c[0][0] = 4.100000, c[SIZE - 1][SIZE - 1] = 4.100000, c[SIZE /
2][SIZE / 2] = 4.100000
Time: 202.90468ms
```

Por fim, realizei uma implementação com threads:

```c
#define SIZE 10000

typedef struct {
    double a[SIZE][SIZE];
    double b[SIZE][SIZE];
    double c[SIZE][SIZE];
} matrices_t;

typedef struct {
    int start;
    int end;
} thread_args_t;

matrices_t matrices;

void initialize(matrices_t *m) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++){
            m->a[i][j] = 1.5;
            m->b[i][j] = 2.6;
        }
    }
}

void *child_thread(void *void_args) {
    thread_args_t *args = (thread_args_t *)void_args;
    for (int row = args->start; row < args->end; row++) {
        for (int column = 0; column < SIZE; column++) {
                        matrices.c[row][column]  =  matrices.a[row][column]  +
matrices.b[row][column];
        }
    }
}

int main() {
    int nprocs = get_nprocs();
    printf("nprocs = %d\n", nprocs);
    initialize(&matrices);
```

```
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    pthread_t tid[nprocs];
    thread_args_t thread_args[nprocs];
    for (int t = 0; t < nprocs; t++) {
        thread_args[t].start = (t * SIZE) / nprocs;
        thread_args[t].end = ((t + 1) * SIZE) / nprocs;
        if (pthread_create(&tid[t], NULL, child_thread, &thread_args[t])) {
            perror("pthread_create");
            exit(1);
        }
    }

    for (int t = 0; t < nprocs; t++) {
        pthread_join(tid[t], NULL);
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);

     printf("c[0][0] = %f, c[SIZE - 1][SIZE - 1] = %f, c[SIZE / 2][SIZE / 2] =
%f\n",
        matrices.c[0][0],
        matrices.c[SIZE - 1][SIZE - 1],
        matrices.c[SIZE / 2][SIZE / 2]);
          printf("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));
}
```

Resultado:

```
nprocs = 4
c[0][0] = 4.100000, c[SIZE - 1][SIZE - 1] = 4.100000, c[SIZE /
2][SIZE / 2] = 4.100000
Time: 147.12047ms
```

- **Análise**

Os resultados iniciais foram:

Sequencial (SIZE = 10000)	340ms
Processos (SIZE = 10000)	203ms
Threads (SIZE = 10000)	147ms

Como esperado, o código com threads é mais rápido, já que a criação de threads é leve e todo o mapa de memória é compartilhado. O código com processos é rápido também, mas perde para threads. A criação de um novo processo no *kernel* é mais pesada, já que envolve a criação de um novo PID com máscara de sinais e mapa de memória próprios, entre outros.

Variando SIZE, percebemos que em certo ponto o *overhead* de criação e sincronização de threads e processos se torna um gargalo. Além disso, percebemos que o aumento de tempo não é muito linear, provavelmente por intervenção do tamanho do cache da CPU:

SIZE	Sequencial	Processos	Threads
16	0.00106ms	0.25241ms	0.14359ms
32	0.00236ms	0.33021ms	0.15875ms
128	0.06518ms	0.56423ms	0.28011ms
512	1.15339ms	0.98583ms	0.73158ms
2048	13.65825ms	9.28496ms	6.52055ms

Por fim, realizei um pequeno teste do efeito do cache e ordem dos loops na performance do programa. Desativei otimizações de compilação (*-O0*) para que o compilador não reordenasse os loops e inverti a ordem de iteração (iterando coluna a coluna). O acesso menos sequencial deve reduzir a performance do programa.

Resultados (SIZE = 2048 e *-O0*):

SIZE	Sequencial	Processos	Threads
Por linha	24.16977ms	13.65163ms	11.00280ms
Por coluna	210.87445ms	104.16595ms	95.99537ms