Atividade 1 - Programação Paralela e Distribuída - Prof. Hélio Crestana Guardia

Maria Luiza Edwards Cordeiro, 802645
Matheus Goulart Ranzani, 800278

Nesta atividade avaliamos diferentes formas de paralelizar o seguinte trecho de código que realiza a multiplicação entre matrizes:

```
for(i=0; i < lin_c; i++)

      for(j=0; j < col_c; j++) {

        // pode ser útil usar uma variável auxiliar para os cálculos
        C[i*col_c+j]=0;

        for(k=0; k < col_a; k++)
          C[i*col_c+j] = C[i*col_c+j] + A[i*col_a+k] * B[k*col_b+j];

        // ser usou variável auxiliar, atribui-se seu valor à C[i][j]
      }
```

Para realizar a tarefa de paralelização utilizamos a API OpenMP da linguagem C. Como a multiplicação de matrizes envolve três loops: o externo (de índice i), o intermediário (de índice j) e o interno (de índice k) pensamos em paralelizar cada um deles separadamente para ver como o programa desempenha.

Além disso, utilizamos diferentes números de threads (1, 2, 4, 8, 16, 32, 64, 128) para fazer o processamento paralelo.

Como opção para avaliar o tempo de processamento dos *fors* utilizamos a função *omp_get_wtime()* da biblioteca <*omp.h*> que calcula o tempo de execução de um trecho de código, dado seu tempo inicial e seu tempo final. Esta função calcula o tempo em segundos.
Antes de começar a paralelizar avaliamos o desempenho da multiplicação de matrizes de forma sequencial. O código base é o seguinte:

```
double tempo_inicio, tempo_fim;

tempo_inicio = omp_get_wtime();

for (i = 0; i < lin_c; i++) {
  for (j = 0; j < col_c; j++) {
    C[i * col_c + j] = 0;
    for (k = 0; k < col_a; k++) {
      C[i * col_c + j] += A[i * col_a + k] * B[k * col_b + j];
    }
```

```
  }
}

tempo_fim = omp_get_wtime();

printf("Threads\tTempo\n");
printf("%d\t%f\n", 1, tempo_fim - tempo_inicio);
```

Compilando o programa e executando-o inserindo os valores de linhas e colunas das matrizes A e B como 2048 obtivemos o seguinte resultado:

```
ranzani in PPD/Atividades on ᚠ main [!?] took 3.0s
→ gcc -Wall -Wno-unused-result mm.c -o mm -O3 -fopenmp && ./mm
Linhas A: 2048
Colunas A / Linhas B: 2048
Colunas B: 2048

Threads Tempo
1       55.748324
```

Agora, vamos partir para a paralelização do for externo em busca de melhorar o desempenho do programa. Para isso, fizemos um outro *for* que engloba a multiplicação das matrizes para obtermos os resultados utilizando diferentes números de threads. O código ficou assim:

```
double tempo_inicio, tempo_fim;

printf("Threads\tTempo\n");

#pragma omp parallel for private(i, j, k) shared(A, B, C)
for (num_threads = 1; num_threads <= MAX_THREADS; num_threads *= 2) {
  omp_set_num_threads(num_threads);

  tempo_inicio = omp_get_wtime();

  for (i = 0; i < lin_c; i++) {
    for (j = 0; j < col_c; j++) {
      C[i * col_c + j] = 0;
      for (k = 0; k < col_a; k++) {
        C[i * col_c + j] += A[i * col_a + k] * B[k * col_b + j];
      }
    }
  }

  tempo_fim = omp_get_wtime();
```

```
  printf("%d\t%f\n", num_threads, tempo_fim - tempo_inicio);
}
```

Compilando e executando da maneira já mostrada anteriormente obtivemos o resultado:

```
ranzani in PPD/Atividades on  main [!?] took 6.2s
→ gcc -Wall -Wno-unused-result mm.c -o mm -O3 -fopenmp && ./mm
Linhas A: 2048
Colunas A / Linhas B: 2048
Colunas B: 2048

Threads Tempo
1       54.375862
2       27.988749
4       13.824070
8       15.548805
16      14.063165
32      15.068771
64      14.007220
128     13.696105
```

Como pode-se observar o uso da diretiva *#pragma omp parallel for* da OpenMP proporcionou uma melhora significativa no desempenho do programa. Especialmente, quando o número de threads aumenta.

Agora vamos paralelizar apenas o loop intermediário do programa para analisar se há diferença comparado com quando paralelizamos apenas o loop externo. Segue o código:

```
double tempo_inicio, tempo_fim;

printf("Threads\tTempo\n");

for (num_threads = 1; num_threads <= MAX_THREADS; num_threads *= 2) {
  omp_set_num_threads(num_threads);

  tempo_inicio = omp_get_wtime();

  for (i = 0; i < lin_c; i++) {
    for (j = 0; j < col_c; j++) {
      #pragma omp parallel for private(j, k) shared(A, B, C)
      C[i * col_c + j] = 0;
      for (k = 0; k < col_a; k++) {
        C[i * col_c + j] += A[i * col_a + k] * B[k * col_b + j];
      }
    }
  }
```

```
  tempo_fim = omp_get_wtime();

  printf("%d\t%f\n", num_threads, tempo_fim - tempo_inicio);
}
```

Compilando e executando novamente, obtivemos:

```
ranzani in PPD/Atividades on  main [!?] took 2m 52.5s
→ gcc -Wall -Wno-unused-result mm.c -o mm -O3 -fopenmp && ./mm
Linhas A: 2048
Colunas A / Linhas B: 2048
Colunas B: 2048

Threads Tempo
1       55.928611
2       31.998416
4       20.589457
8       25.585488
16      19.754170
32      20.094526
64      22.146651
128     23.502201
```

Dessa forma percebe-se que ainda há um ganho de desempenho ao fazer uma paralelização no programa, entretanto quando paralelizamos o loop externo essa melhoria no desempenho foi significativamente melhor.

Por fim, paralelizamos apenas o loop interno. É importante ressaltar que essa paralelização não é muito adequada no caso deste programa, pois o loop interno realiza cálculos dependentes dos outros dois loops, portanto paralelizá-lo pode não proporcionar ganhos significativos na questão de desempenho e pode, inclusive, gerar problemas de concorrência. O código do loop interno paralelizado é o seguinte:

```
double tempo_inicio, tempo_fim;

printf("Threads\tTempo\n");

for (num_threads = 1; num_threads <= MAX_THREADS; num_threads *= 2) {
  omp_set_num_threads(num_threads);

  tempo_inicio = omp_get_wtime();

  for (i = 0; i < lin_c; i++) {
    for (j = 0; j < col_c; j++) {
      C[i * col_c + j] = 0;
```

```
      #pragma omp parallel for private(k) shared(A, B, C)
      for (k = 0; k < col_a; k++) {
        C[i * col_c + j] += A[i * col_a + k] * B[k * col_b + j];
      }
    }
  }

  tempo_fim = omp_get_wtime();

  printf("%d\t%f\n", num_threads, tempo_fim - tempo_inicio);
}
```

Finalmente, compilando e executando o programa obtivemos um resultado de certa forma inesperado:

```
ranzani in PPD/Atividades on  main [!?] took 3m 44.4s
→ gcc -Wall -Wno-unused-result mm.c -o mm -O3 -fopenmp && ./mm
Linhas A: 2048
Colunas A / Linhas B: 2048
Colunas B: 2048

Threads Tempo
1       61.333598
2       28.038421
4       20.118429
8       894.204761
^C
```

Desta vez o programa nem terminou de executar, ele demorou certa de 15 minutos ao usar 8 threads e após isso achamos melhor forçar a parada da execução. A grande demora ao utilizar 8 threads pode ter ocorrido devido a uma condição de espera ou a um impasse (deadlock) que ocorreu no código. Além disso, outro problema pode estar relacionado à criação de um grande número de threads, causando sobrecarga e possível falta de recursos para a execução, assim gerando um bloqueio.

Como pudemos observar, paralelizar o loop interno não é vantajoso em níveis de desempenho e devemos evitar a sua paralelização para mantermos o bom funcionamento do algoritmo.

Dessa forma, concluímos que a paralelização contribui bastante para melhorar a performance da multiplicação de matrizes. Entretanto é importante analisar e definir bem qual loop deve ser paralelizado para assim, obter o melhor desempenho possível e ainda manter a corretude do programa.