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

Sistemas de Computação de Alto Desempenho

Aula 7

Relatório:
OpenMP - Tasks

- **Introdução**

Em aula, estudamos o uso de OpenMP para paralelismo, com foco no uso de tasks.

- **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 uma versão paralela para o programa *exemplo-lista-liga.c*, onde para uma lista de 10 elementos realizada um processamento (sleep) para cada elemento da lista ligada.

Versão serial, com medição de tempo de execução:

```c
#include <stdio.h>
#include <omp.h>
#include <time.h>
#define n 10
typedef struct {int a; int *proximo;} tipo_lista;
tipo_lista *lista;
tipo_lista *lista_first;
void inicia(int size)
{
  int i;
  tipo_lista *lista_anterior;
  lista=(tipo_lista *) malloc(sizeof(tipo_lista));
  lista->a=0;
  lista_first=lista;
  for (i=1;i<size;i++)
   {
     lista_anterior=lista;
     lista=(tipo_lista *) malloc(sizeof(tipo_lista));
     lista->a=i%5;
     lista_anterior->proximo=lista;
   }
   lista->proximo=(int *) 0;
}


void main()
```

```
{
struct timespec start_time, end_time;
    tipo_lista *prox;
     inicia(n);
     prox=lista_first;
     clock_gettime(CLOCK_MONOTONIC, &start_time);
     while (prox) {
        lista=prox;
        printf("%d \n",lista->a);
        sleep(lista->a);
        prox=lista->proximo;
     }
     clock_gettime(CLOCK_MONOTONIC, &end_time);
     printf("FIM MAIN\nTime: %.5fms\n",
    (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec * 1000 +
1e-6 * start_time.tv_nsec));
}
```

Versão paralela, com medição de tempo de execução:

```
#include <stdio.h>
#include <omp.h>
#include <time.h>
#define n 10
typedef struct {int a; int *proximo;} tipo_lista;
tipo_lista *lista;
tipo_lista *lista_first;
void inicia(int size)
{
  int i;
  tipo_lista *lista_anterior;
  lista=(tipo_lista *) malloc(sizeof(tipo_lista));
  lista->a=0;
  lista_first=lista;
  for (i=1;i<size;i++)
   {
     lista_anterior=lista;
     lista=(tipo_lista *) malloc(sizeof(tipo_lista));
     lista->a=i%5;
     lista_anterior->proximo=lista;
   }
   lista->proximo=(int *) 0;
}


void main()
{
    struct timespec start_time, end_time;
    tipo_lista *prox;
    inicia(n);
    prox=lista_first;
```

```
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    #pragma omp parallel num_threads(4)
    {
        #pragma omp single
        while (prox) {
            lista=prox;
            #pragma omp task firstprivate(lista)
            {
                printf("%d  thread=%d\n",lista->a, omp_get_thread_num());
                fflush(stdout);
                sleep(lista->a);
            }
            prox=lista->proximo;
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("FIM MAIN\nTime: %.5fms\n",
     (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec * 1000 +
1e-6 * start_time.tv_nsec));
}
```

Comparação de desempenho (desconsiderando a inicialização da lista):

Serial	Paralela
20002.19ms	7004.58ms

No **exercício 2**, paralelizamos uma implementação de merge sort. A versão serial, com chamadas para medida de tempo, é:

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

int i;
int n;
int v[1000];

void inicia_vetor(int n) {
    int i;
    for (i = 0; i < n; i++) {
        v[i] = random() % 1000;
    }
}

void merge(int p,int q,int r, int v[]) {
    int i, j, k;
    int v_aux[1000];
```

```
    i = p;
    j = q;
    k = 0;
    while (i < q && j < r) {
        if (v[i] <= v[j])
            v_aux[k++] = v[i++];
        else
            v_aux[k++] = v[j++];
    }
    while (i < q)
        v_aux[k++] = v[i++];
    while (j < r)
        v_aux[k++] = v[j++];
    for (i = p; i < r; i++)
        v[i] = v_aux[i - p];
}

void merge_sort(int p,int r,int v[]) {
    int q;
    if (p < r - 1){
        q = (p + r) / 2;
        merge_sort(p, q, v);
        merge_sort(q, r, v);
        merge(p, q, r, v);
    }
}

int main() {
    int i;
    printf("n: ");
    scanf("%d", &n);
    inicia_vetor(n);
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    merge_sort(0, n, v);
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    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));
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
    return 0;
}
```

Executando esse programa 20 vezes com 1000 elementos, adquirimos como melhor tempo de execução 0.24ms. Em seguida, paralelizamos o programa com *tasks*:

- Na *main*, na chamada a *merge_sort*, criamos um região com 4 threads, e chamamos *merge_sort* em um única thread. Escolhemos criar as 4 threads na *main* para evitar a criação em cada chamada de *merge_sort*:

```
#pragma omp parallel num_threads(4)
{
    #pragma omp single
    merge_sort(0, n, v);
}
```

- Em *merge_sort*, separamos cada chamada recursiva de *merge_sort* em uma *task* diferente:

```
#pragma omp task
merge_sort(p, q, v);
#pragma omp task
merge_sort(q, r, v);
#pragma omp taskwait
merge(p, q, r, v);
```

Executando esse programa 20 vezes com 1000 elementos, adquirimos como melhor tempo de execução 0.96ms. Atribuímos o aumento de tempo a *overhead* de criação de *tasks* e alocação em *threads*. O código está criando tasks até para arrays pequenos, o que é bastante custoso.

Como otimização,decidimos não criar tasks e executar diretamente na thread atual quando o tamanho do array for pequeno. Em *merge_sort*, mudamos a chamada para:

```
#pragma omp task if(r - p >= 32)
merge_sort(p, q, v);
#pragma omp task if(r - p >= 32)
merge_sort(q, r, v);
#pragma omp taskwait
merge(p, q, r, v);
```

Com essa pequena otimização, o tempo de execução caiu para 0.72ms.

Serial	Paralelo	Paralelo (otimizado)
0.24ms	0.96ms	0.72ms