UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI
CIÊNCIA DA COMPUTAÇÃO

[Computação Paralela] Trabalho Prático II

Lucas Rômulo de Souza Resende

Trabalho Prático II de Computação Paralela do curso de Ciência da Computação da Universidade Federal de São João del-Rei.

Prof. Dr. Rafael Sachetto Oliveira

São João del-Rei
24 de novembro de 2022

Sumário

1 Introdução **1**

2 Descrição do programa **1**
2.1 OpenMP 2

3 Testes **3**
3.1 gprof 3
3.2 clock 4

4 Conclusão **4**

1 Introdução

O objetivo desse trabalho é implementar um modelo predador-presa de forma sequencial e paralelizá-lo utilizando a API OpenMP.

A modelagem de sistemas predador-presa, também é conhecidos por Lokta-Volterra, consiste na criação de um ambiente onde convivem duas espécies diferentes, e uma delas tenta predar a outra.

No modelo gerado temos uma simulação de um ecossistema populado por raposas e coelhos, e esse ecossistema possúi as seguintes regras:

- Raposas predam coelhos
- Coelhos reproduzem
- Raposas reproduzem
- Raposas morrem de fome

Os resultados do modelo implementado são condizentes com os da especificação, porém, a avaliação da eficácia da paralelização foi inconclusiva.

2 Descrição do programa

As informações dos animais são armazenadas em TADs.

```
typedef struct predator_t { // raposas
  int gen_proc;      // geracoes ate procriacao
  int gen_food;      // geracoes ate morrer de fome
} predator_t;

typedef struct prey_t {     // coelhos
  int gen_proc;      // geracoes ate prociacao
} prey_t;
```

Existe também um outro TAD que representa as informações completas de cada indivíduo em relação com o ambiente, e o ambiente é representado por uma lista dos indivíduos presentes.

```
typedef struct subject_t {  // inividuo
  char type;     // tipo: raposa, coelho, rocha, nulo
  int x, y;      // posicao no ambiente
  union {
    predator_t predator;
    prey_t prey;
  };
} subject_t;

// lista de individuos
subject_t subjects[L*C];
```

A lógica das gerações segue o seguinte pseudocódigo:

```
para cada geracao{

    lista_proximos = lista_atual

    movimenta_coelhos(lista_proximos, const lista_atual)
```

```
6
7       lista_atual = lista_proximos
8
9       movimenta_raposas(lista_proximos, const lista_atual)
10
11      resolve_conflitos(lista_proximos)
12
13      lista_atual = lista_proximos
14  }
```

Nas funções movimenta_coelhos e movimenta_raposas os indivíduos são modificados na lista_proximos enquanto a lista_atual é mantida constante, no intuito de simular o movimento de todos os animais da mesma espécie mutualmente.

A função resolve_conflitos percorre a lista de forma semelhante ao algoritmo *selection sort*: cada indivíduo na lista interage com todos os seus próximos e, caso estejam na mesma posição, apenas um deles é mantido, baseado nas regras do ecossistema.

2.1 OpenMP

A API foi utilizada na paralelização das funções movimenta_coelhos, movimenta_--raposas e resolve_conflitos.

As funções de movimentação foram totalmente paralelizadas devido ao fato de que as modificações ocorrem em uma lista e as verificações em outra, e duas ou mais threads nunca trabalham em mais de um indivíduo.

```
movimenta_coelhos(lista_proximos, const lista_atual){

    #pragma omp parallel for schedule(dynamic) shared(N_tmp)
    for (int id = 0; id < R*C; id++){
        if (lista_atual[id] == COELHO){
            /* code */
        }
        /* code */
    }
}

movimenta_raposas(lista_proximos, const lista_atual){

    #pragma omp parallel for schedule(dynamic) shared(N_tmp)
    for (int id = 0; id < R*C; id++){
        if (lista_atual[id] == RAPOSA){
            /* code */
        }
        /* code */
    }
}
```

Enquanto a função resolve_conflitos possui seções críticas por utilizar apenas uma lista, e as threads alterarem indivíduos que outras podem estar utilizando.

```
resolve_conflitos(lista_proximos){

  #pragma omp parallel for schedule(dynamic) shared(N_tmp)
  for (int i = 0; i < size; i++){
    for (int j = i+1; j < size; j++){

            s1 = get_subject1;
```

```
8               s2 = get_subject2;
9
10              if (s1->posicao == s2->posicao)
11
12                  #pragma omp critical
13                  {
14                      if (s1->posicao == s2->posicao)
15                          /* modifica_indiciduos */
16                  }
17          }
18      }
19  }
```

3 Testes

Para os testes do modelo foram utilizados os dados apresentados na própria especificação.

O resultado obtido foi o esperado e, quando observado as atualizações de cada geração pode-se perceber que o modelo funciona de acordo com as especificações.

Para a avaliação da paralelização foram utilizados o profiler gprof e a função de contagem de clocks, em execuções com 1, 4 e 8 threads.

3.1 gprof

```
lucasromulo@DESKTOP-Kubuntu:~/Projects/parallel/TP2$ make gprof NUM_THREADS=1; cat profile
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        6     0.00     0.00  move_predators
  0.00      0.00     0.00        6     0.00     0.00  move_preys
  0.00      0.00     0.00        6     0.00     0.00  solve_conflicts
lucasromulo@DESKTOP-Kubuntu:~/Projects/parallel/TP2$
```

Figura 1: gprof utilizando 1 thread

```
lucasromulo@DESKTOP-Kubuntu:~/Projects/parallel/TP2$ make gprof NUM_THREADS=4; cat profile
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        6     0.00     0.00  move_predators
  0.00      0.00     0.00        6     0.00     0.00  move_preys
  0.00      0.00     0.00        6     0.00     0.00  solve_conflicts
lucasromulo@DESKTOP-Kubuntu:~/Projects/parallel/TP2$
```

Figura 2: gprof utilizando 4 threads

```
lucasromulo@DESKTOP-Kubuntu:~/Projects/parallel/TP2$ make gprof NUM_THREADS=8; cat profile
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        6     0.00     0.00  move_predators
  0.00      0.00     0.00        6     0.00     0.00  move_preys
  0.00      0.00     0.00        6     0.00     0.00  solve_conflicts
lucasromulo@DESKTOP-Kubuntu:~/Projects/parallel/TP2$
```

Figura 3: gprof utilizando 8 threads

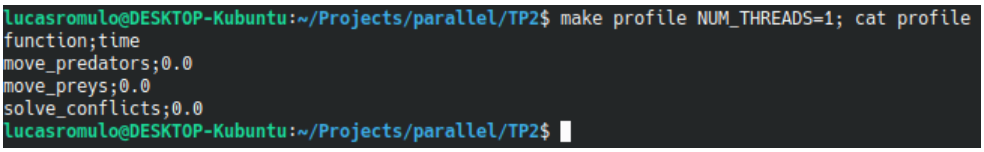

Figura 4: clock utilizando 1 thread

Figura 5: clock utilizando 4 threads

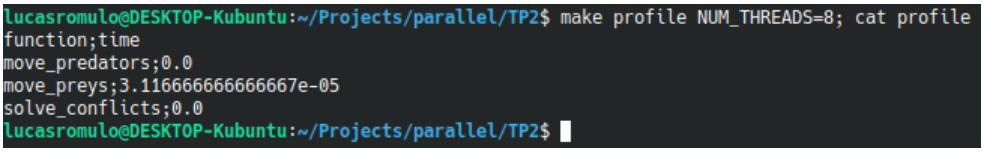

Figura 6: clock utilizando 8 threads

3.2 clock

4 Conclusão

Pode-se observar que o tempo de execução medido pelo gprof não trás nenhum insight para a análise de comparação entre a execução sequencial e paralela, enquanto nos tempos medidos com a utilização da função de clock mostra que a paralelização levou mais tempo utilizando 4 threads ao invés de uma, e esse tempo quase dobra quando comparadas as execuções com 4 e 8 threads.