



Universidade Federal São João del Rei
Curso de Ciência da Computação

Trabalho Prático II:
Relatório
Computação Paralela

Discente: Julio Cesar da Silva Rodrigues

Docente: Rafael Sachetto Oliveira

Novembro
2023

1 Introdução

Neste trabalho, é apresentada uma solução de paralelização em um problema de se simular um ecossistema simples, habitado por duas espécies de animais, raposas e coelhos, cuja existência ao longo de gerações é mutuamente dependente. Pode também ser interpretado como o modelo predador-presa, ou modelo de Lotka-Volterra.

O objetivo principal é identificar porções da implementação que por talvez apresentarem maior uso computacional, possam ser paralelizadas utilizando a biblioteca **OpenMP**¹ para ambientes de memória compartilhada para diminuir significativamente o tempo de execução. Além disso, é realizada uma análise detalhada quanto a como cada tipo de entrada afeta no tempo de execução do programa, uso de recursos computacionais (CPU, memória principal, ...) e o potencial *speed up* que o aumento da paralelização (i.e., número de processos) proporciona.

2 Desenvolvimento

Toda a concepção da dinâmica de movimento de coelhos e raposas, assim como as funções responsáveis pela paralelização e manipulação de E/S foram implementadas na linguagem C (excetuando a análise gráfica de resultados e geração dos instâncias de teste, estes que foram gerados utilizando a linguagem Python² com auxílio das bibliotecas PyCryptodome³, Matplotlib⁴ e seaborn⁵). A implementação foi desenvolvida com o auxílio da ferramenta de controle de versionamento GitHub⁶, e encontra-se disponível publicamente no repositório https://github.com/juliorodrigues07/ecosystem_simulation.

A organização do código se apresenta de forma bastante simples. Existem um par de arquivos de código fonte (juntamente com seus *headers* *.h* associados) que são responsáveis por realizar as manipulações de entrada e saída (*data_management*) e movimentar os animais pela matriz do ecossistema (*simulation*). No código fonte presente no arquivo *simulation.c* encontra-se a utilização da biblioteca OpenMP para introdução do paralelismo ao programa.

¹Mais informações disponíveis em: <https://www.openmp.org/>

²Mais informações disponíveis em: <https://www.python.org/>

³Mais informações disponíveis em: <https://pycryptodome.readthedocs.io/en/latest/>

⁴Mais informações disponíveis em: <https://matplotlib.org/>

⁵Mais informações disponíveis em: <https://seaborn.pydata.org/>

⁶Mais informações disponíveis em: <https://github.com>

Além disso, existe um arquivo *header* (.h) base solitário que define as estruturas e constantes utilizadas na formulação da solução. A primeira estrutura possui o nome de *cell* e é responsável por armazenar as informações individuais de cada objeto do ecossistema. O único atributo comum a todos os possíveis objetos é *type*. Atributos correspondentes à procriação de coelhos e raposas ou fome de raposas só são utilizados quando um dos mesmos está presente naquela posição (por isso foram definidos dentro de *unions*). Além disso, alguns são mutuamente excludentes como *age_rabbit* e *age_fox*.

- *type*: Inteiro responsável pelo armazenamento da identificação do objeto presente naquela posição ($\{-1, 0, 1, 2\} \rightarrow \{\text{ROCHA, VAZIA, RAPOSA, COELHO}\}$);
- *age_rabbit*: Tempo desde a última procriação de um determinado coelho na posição;
- *age_fox*: Tempo desde a última procriação de uma determinada raposa;
- *hunger_fox*: Tempo desde que uma determinada raposa na posição comeu um coelho;

A segunda estrutura é responsável pelo armazenamento do ecossistema como um todo, e por isso, existe apenas uma desta alocada a qualquer momento durante a execução do programa. Os atributos que compõem esta estrutura são detalhados a seguir:

- *rabbit_gen*: Inteiro correspondente ao intervalo de gerações entre procriação dos coelhos;
- *fox_gen*: Inteiro correspondente ao intervalo de gerações entre procriação dos raposas;
- *fox_food*: Inteiro correspondente ao intervalo máximo que as raposas podem ficar sem comer um coelho;
- *n_gen*: Número de gerações para simulação;
- *r*: Número de linhas da matriz do ecossistema;
- *c*: Número de colunas da matriz do ecossistema;
- *n*: Número de objetos presentes na matriz do ecossistema;
- *cell*: Matriz^{*rc*} do ecossistema em que cada elemento é do tipo *cell* definido anteriormente;

Para simular o ecossistema, são utilizadas duas matrizes do tipo *cell*. Com isto, é possível movimentar os objetos de forma que cada movimento em determinada geração possa influenciar na movimentação de objetos vizinhos, o que por exemplo, eliminaria uma das questões da especificação do trabalho para resolução de conflitos quando objetos tentam se mover para uma mesma posição. A análise de posições adjacentes e atributos de cada objeto são sempre realizadas na matriz fonte (**A**), enquanto a movimentação em si é realizada apenas na cópia desta (**A'**) enquanto a matriz está sendo percorrida, atualizando a base no final de cada geração simulada.

Para testar a implementação de forma minimamente satisfatória, foram geradas quatro instâncias contendo estados iniciais de ecossistemas de dimensões bastante distintas. Além dos parâmetros já definidos na especificação do trabalho como *GEN_PROC_COELHOS*, *GEN_PROC_RAPOSAS*, *GEN_COMIDA_RAPOSAS*, *N_GEN*, *L*, *C*, é definida também a porcentagem de posições que serão destinadas para cada objeto ocupar. No entanto, a posição inicial (*X*, *Y*) de cada objeto é gerada aleatoriamente.

2.1 Movimentação de Coelhos

A assinatura da função responsável pelo movimento de cada coelho é definida como *move_rabbits*. No interior desta, a matriz **A** é percorrida ($r \times c$) analisando as posições em que existem coelhos. Dado que um coelho existe em determinada posição **X** e **Y** (que correspondem aos índices das matrizes), as células adjacentes vazias (horizontal e vertical) são numeradas de 0 à 3 em um *array* auxiliar *adj* ($\{0, 1, 2, 3\} \rightarrow \{\text{NORTE, LESTE, SUL, OESTE}\}$) respeitando os limites do espaço da matriz e a variável acumuladora do número de posições disponíveis também é incrementado.

A enumeração das posições possíveis é realizada na função *empty_cells*, enquanto o índice que corresponde à posição no vetor *adj* que contém a posição vazia adjacente selecionada é dado pela fórmula especificada no trabalho $((\mathbf{G} + \mathbf{X} + \mathbf{Y}) \pmod{\mathbf{P}})$ na função *calculate_next_position*. Por exemplo, para um coelho na geração 2 e na posição (1, 0) em que as entre as posições adjacentes, (1, 1) está ocupada e (0, 0) e (2, 0) estão vazias, o *array adj* (este é inicializado com valores -1) será definido da seguinte forma:

$$\begin{aligned} \{0, 2, -1, -1\} &\rightarrow \{\text{NORTE, SUL}\} \\ i &= 2 + 1 + 0 \pmod{2} = 1 \\ adj[i] &= adj[1] = 2 = \text{SUL} \end{aligned}$$

Calculada a próxima posição do coelho, se existir uma posição vazia adjacente, este é movido na matriz **A'** carregando seus atributos consigo na

função *move_animal_to_adj*. Caso este coelho ainda não possa procriar, a posição que este deixou é definida como vazia e os atributos pertencentes aos objetos do tipo animal são degenerados para evitar possíveis erros em movimentações futuras. Caso contrário, o coelho ainda é movimentado na matriz \mathbf{A}' , deixando um filho em sua posição anterior e definindo o tempo sem procriar de ambos como zero. Além disso, o possível conflito de dois ou mais coelhos tentarem se mover para uma mesma posição é resolvido nesta porção, persistindo o coelho que está a menos tempo sem procriar.

2.2 Movimentação de Raposas

A assinatura da função responsável pelo movimento de cada raposa é definida como *move_foxes*. De forma similar à movimentação de coelhos, a matriz \mathbf{A} é percorrida desta vez analisando as posições em que existem coelhos ou vazias.

Dado que uma raposa existe em determinada posição \mathbf{X} e \mathbf{Y} , as células adjacentes que contém coelho (horizontal e vertical) são numeradas de 0 à 3 em um *array* auxiliar *eat* ($\{0, 1, 2, 3\} \rightarrow \{\text{NORTE, LESTE, SUL, OESTE}\}$) respeitando os limites do espaço da matriz e a variável acumuladora (*available_to_eat*) do número de posições disponíveis também é incrementada. Caso não haja nenhum coelho nas células adjacentes, o mesmo processo de enumeração é realizado, desta vez com o *array* auxiliar sendo *adj* e a variável acumuladora *empty*.

Calculada a próxima posição da raposa, se existir uma posição adjacente que contém um coelho, esta é movida na matriz \mathbf{A}' carregando seus atributos consigo na função, consequentemente eliminando o coelho e redefinindo seu atributo de fome como zero. Caso não exista uma posição adjacente que contém um coelho, mas sim uma posição adjacente vazia, esta também é movida na matriz \mathbf{A}' carregando seus atributos.

Caso esta raposa ainda não possa procriar, a posição que este deixou é definida como vazia e os atributos pertencentes aos objetos do tipo animal são degenerados para evitar possíveis erros em movimentações futuras. Caso contrário, a raposa é movimentada na matriz \mathbf{A}' , deixando um filho em sua posição anterior e definindo o tempo sem procriar de ambas como zero.

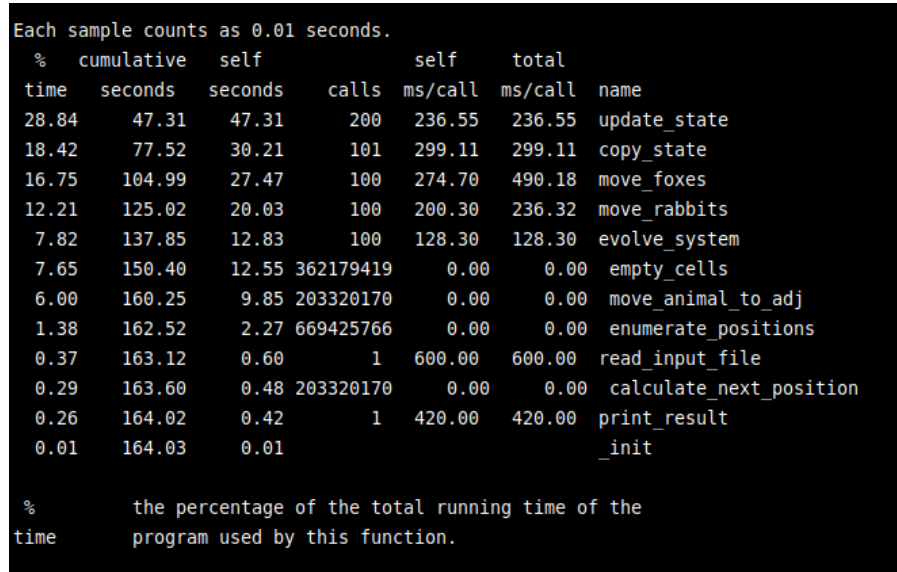
Novamente, o possível conflito de duas ou mais raposas tentarem se mover para uma mesma posição é resolvido nesta porção, persistindo o raposas que está a menos tempo sem procriar. Por fim, quando uma raposa procria, o tempo sem comer do filhote é definido como zero, enquanto o tempo sem comer da raposa genitora é inalterado.

2.3 Perfil de Desempenho Sequencial

Analisando a execução da solução para o problema sem quaisquer tipos de paralelismo introduzidos, é nítida a ociosidade dos recursos computacionais, principalmente por parte da CPU, utilizando apenas uma *thread* para simular o ecossistema de acordo com o estado inicial do mesmo dado pelo arquivo de entrada.

Com auxílio da ferramenta de perfilamento **gprof**⁷, é observado que em média 50% do tempo de processamento é oriundo apenas da execução das funções *copy_state* e *update_state* (Figura 1). Tais funções são responsáveis por realizar a cópia da matriz **A** e por atualizá-la, respectivamente.

Na cópia, todos os elementos do tipo *cell* (consequentemente seus atributos) são copiados para a matriz auxiliar de movimentação **A'**. No entanto, na atualização o processo inverso é realizado, copiando todos os elementos de **A'** para **A**. A cópia é realizada sempre antes da simulação de cada geração, enquanto a atualização é executada sempre duas vezes durante a simulação de cada geração, após mover os coelhos e após mover as raposas.



```
Each sample counts as 0.01 seconds.
%   cumulative   self           self       total
time  seconds    seconds   calls  ms/call  ms/call  name
28.84    47.31    47.31      200    236.55   236.55  update_state
18.42    77.52    30.21      101    299.11   299.11  copy_state
16.75   104.99    27.47      100    274.70   490.18  move_foxes
12.21   125.02    20.03      100    200.30   236.32  move_rabbits
 7.82   137.85    12.83      100    128.30   128.30  evolve_system
 7.65   150.40    12.55 362179419     0.00     0.00  empty_cells
 6.00   160.25     9.85 203320170     0.00     0.00  move_animal_to_adj
 1.38   162.52     2.27 669425766     0.00     0.00  enumerate_positions
 0.37   163.12     0.60         1    600.00   600.00  read_input_file
 0.29   163.60     0.48 203320170     0.00     0.00  calculate_next_position
 0.26   164.02     0.42         1    420.00   420.00  print_result
 0.01   164.03     0.01                      _init

%
time    the percentage of the total running time of the
        program used by this function.
```

Figura 1: Relatório gerado com a instância *instance_4.txt*

O restante do tempo de processamento é dividido entre as funções de movimento de coelhos e raposas, enumeração das posições adjacentes disponíveis

⁷Mais informações disponíveis em: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html

e evolução do sistema (20%, 10%, 10% e 10% aproximadamente, respectivamente). A função responsável pela evolução do sistema apenas percorre a matriz **A** incrementando o tempo sem procriar dos animais e eliminando raposas que atingiram o limiar de fome.

2.4 Identificação das Oportunidades de Paralelização

Como as funções de cópia e atualização das matrizes e de evolução do sistema juntas correspondem em média a aproximadamente 60% do tempo total de processamento com as instâncias geradas, estas foram as primeiras analisadas como oportunidades para paralelização. Um aspecto em comum entre todas estas funções, é que estas executam apenas operações pontuais em cada célula das matrizes sem que exista quaisquer dependências de dados em relação a quaisquer outras células da matriz (problema embaraçosamente paralelo).

Neste caso, podemos realizar uma decomposição simples de domínio por linhas, colunas, submatrizes ou até mesmo o escalonamento de célula por célula para cada *thread* na tentativa de atender requisitos como balanceamento de carga. No caso da função de cópia por exemplo, dado que existe um processador com $r \times c$ núcleos, seria possível realizar a cópia de todos os elementos ao mesmo tempo, com cada $r \times c$ *threads* realizando a atribuição de um único elemento da matriz.

Por fim, foram analisadas as funções de movimentação dos animais que, embora possuam a resolução de conflitos no caso de dois ou mais tentarem mover-se para a mesma posição, a operação de movimento como um todo possui dependência de dados na análise e atribuições (movimentar um animal corresponde a um cadeia de atribuições condicionadas, e por isso não é uma instrução atômica) entre as células adjacentes de toda posição (X, Y) analisada. Estas dependências levam a criação de seções críticas em certas porções dos códigos responsáveis pela movimentação dos animais para serem executadas de forma sequencial entre as *threads*.

2.5 Paralelização

Em todas as porções de código selecionadas para paralelização estamos lidando com operações em uma matriz. Diante disso, o conjunto de diretivas $D = \{\text{\texttt{\#pragma omp parallel}}, \text{\texttt{\#pragma omp for schedule(type, c)}}, \text{\texttt{\#pragma omp for}}, \text{\texttt{\#pragma omp critical}}\}$ foram utilizadas para introduzir o paralelismo ao programa.

As estratégias utilizadas na paralelização são bastante simples. Nas cinco funções paralelizadas, foi realizada uma decomposição de domínio por linhas

da matriz de forma não cíclica. Em outras palavras, para uma matriz de ecossistema com R linhas e N threads, a matriz seria dividida em blocos, com cada bloco contendo possivelmente N linhas da matriz, ou seja, a cada bloco, cada *thread* executaria operações em uma linha completa da matriz.

Para as funções *copy_state*, *update_state* e *evolve_system*, utilizou-se apenas a diretiva `#pragma omp for` de forma que cada *thread* realiza operações em uma linha completa da matriz como mencionado anteriormente. Já para as funções *move_rabbits* e *move_foxes*, utilizou-se a diretiva `#pragma omp for schedule`, dividindo a matriz horizontalmente em partes $(R/N \times C)$ para cada *thread*. No entanto, nas operações de movimentação dos animais na matriz A' , foi introduzida a diretiva `#pragma omp critical` devido à existência de dependências de dados entre posições adjacentes na mesma.

3 Compilação e Execução

Para compilar e executar o programa sequencialmente sem ou com perfilamento, basta executar os respectivos comandos via terminal:

```
make sequential
```

```
make with_prof
```

Para compilar a versão paralelizada do programa, basta executar o seguinte comando via terminal:

```
make
```

Para executar a versão paralelizada do programa, basta executar o seguinte comando via terminal:

```
make parallel
```

Os parâmetros de execução podem e devem ser alterados no arquivo *Makefile*, alterando o arquivo de teste por exemplo. Também é possível gerar suas próprias instâncias de teste utilizando o *script Python* presente no diretório *instances*. Basta selecionar os parâmetros do estado inicial do ecossistema e executar o seguinte comando como exemplo:

```
python3 instance_generator.py -l 5 -c 5 -gr 2 -gf 4 -hf 3 -n 6 -r 0.1 -f 0.1 -s 0.2
```


4 Análise de Resultados

Os resultados obtidos foram, no geral, bastante condizentes com as expectativas, apresentando reduções nos tempos de execução, conforme o aumento do número de *threads*. Todos os testes foram realizados em um computador com processador Intel® Core™ i5-8265U CPU que possui 4 núcleos, 8 *threads* (Hyper-threading) e frequência base 1,60 GHz.

As informações sobre cada instância gerada para teste são apresentadas na Tabela 1. Os parâmetros *rabbits*, *foxes* e *stones* correspondem à porcentagem do total de posições na matriz ocupadas pelo objeto.

Tabela 1: Parâmetros das Instâncias de Teste

Instância	r	c	rabbit_gen	fox_gen	fox_food	n_gen	rabbits	foxes	stones
instance_1.txt	7	10	4	6	3	100	20%	20%	10%
instance_2.txt	100	120	8	10	7	100	20%	20%	20%
instance_3.txt	500	400	17	20	15	100	30%	20%	20%
instance_4.txt	5000	5000	5	10	15	100	12%	16%	8%

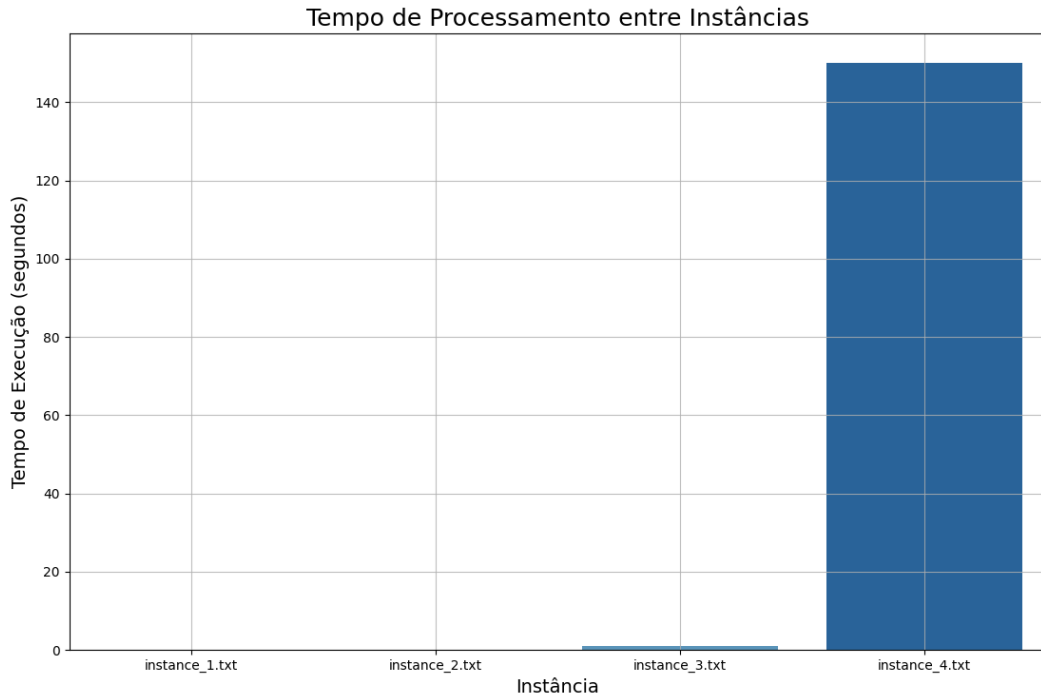


Figura 2: Tempo de execução sequencial

Na Figura 2, podemos observar que o tempo de execução cresce rapidamente conforme os tamanhos do ecossistema crescem (dimensões da matriz), um comportamento dentro do esperado. Já para os resultados obtidos com

execuções em paralelo, foram realizados vários testes com três valores distintos para o número de *threads* com todas as instâncias ($\{2, 3, 4\}$).

Para a primeira instância, como a dimensão do ecossistema é relativamente pequena, não ocorreram diferenças significativas nos tempos de execução mensurados. O gráfico que apresentam as curvas de tempo de processamento para tal instância é apresentado na Figura 3.

Para o restante das instâncias, é possível observar um padrão de descréscimo do tempo de execução em razão do aumento do número de *threads*. No entanto, podemos observar nitidamente um *speed up* sublinear, este que poderia ser explicado pela presença de três regiões críticas presentes na movimentação de coelhos e raposas no total, o que torna a movimentação em si uma operação sequencial, colocando *threads* em ociosidade a maior parte do tempo em que o programa está em execução.

Considerando somente os testes com a instância de maior relevância para os resultados (*instance_4.txt*), o *speed up* obtido com 4 *threads* foi de aproximadamente $1,5\times$ em relação ao tempo de execução sequencial. Os gráficos que apresentam as curvas de tempo de processamento para tais instâncias são apresentados na Figura 4, 5 e 6. Por fim, podemos observar na Figura 7 o quadro geral das execuções sequenciais e em paralelo, com diferentes números de *threads* e instâncias distintas.

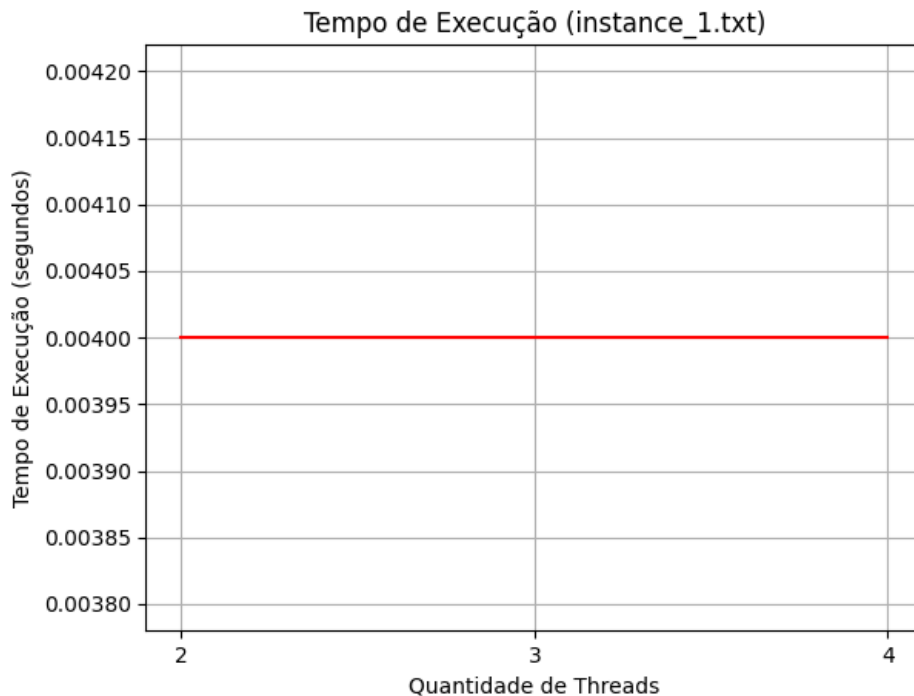


Figura 3: Tempos de execução com a instância *instance_1.txt*



Figura 4: Tempos de execução com a instância *instance_2.txt*

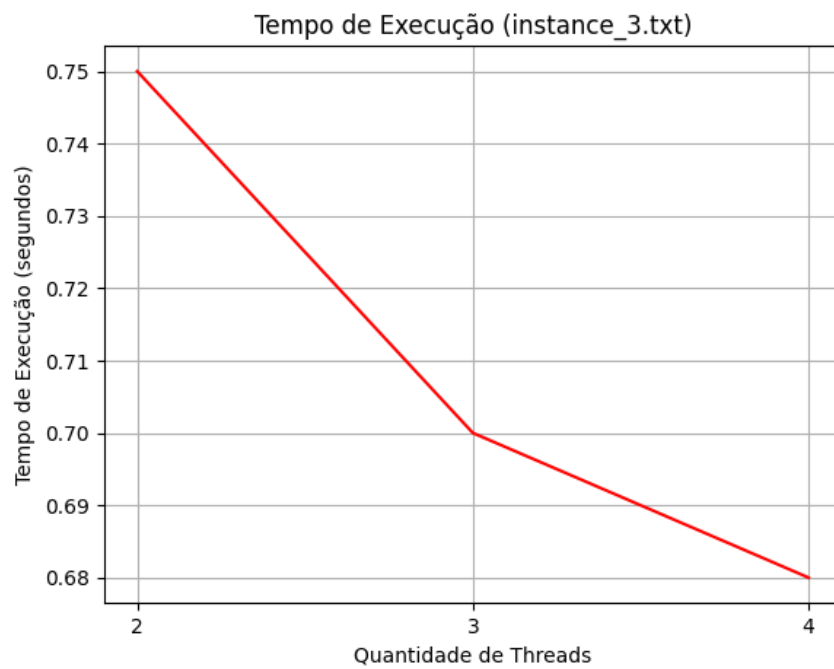


Figura 5: Tempos de execução com a instância *instance_3.txt*

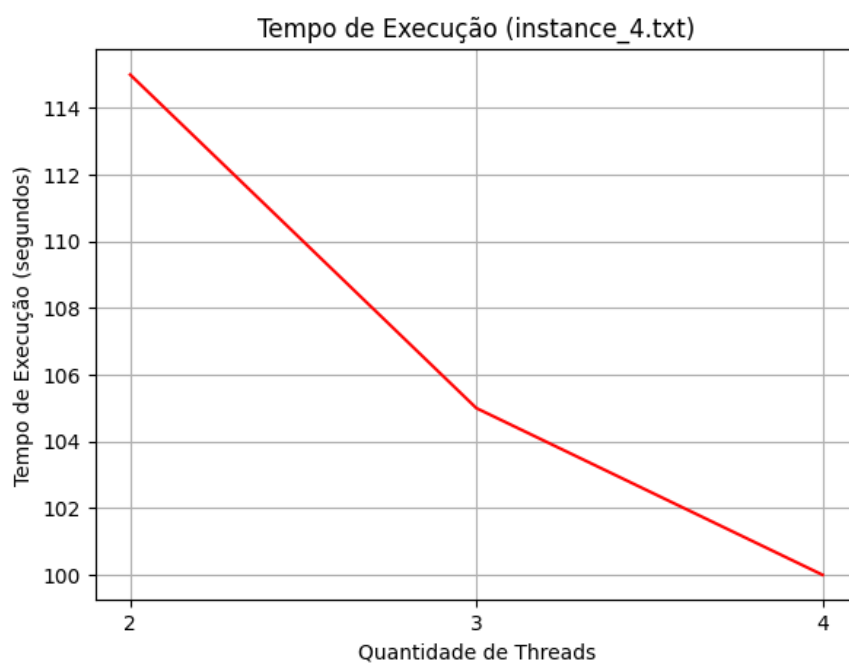


Figura 6: Tempos de execução com a instância *instance_4.txt*

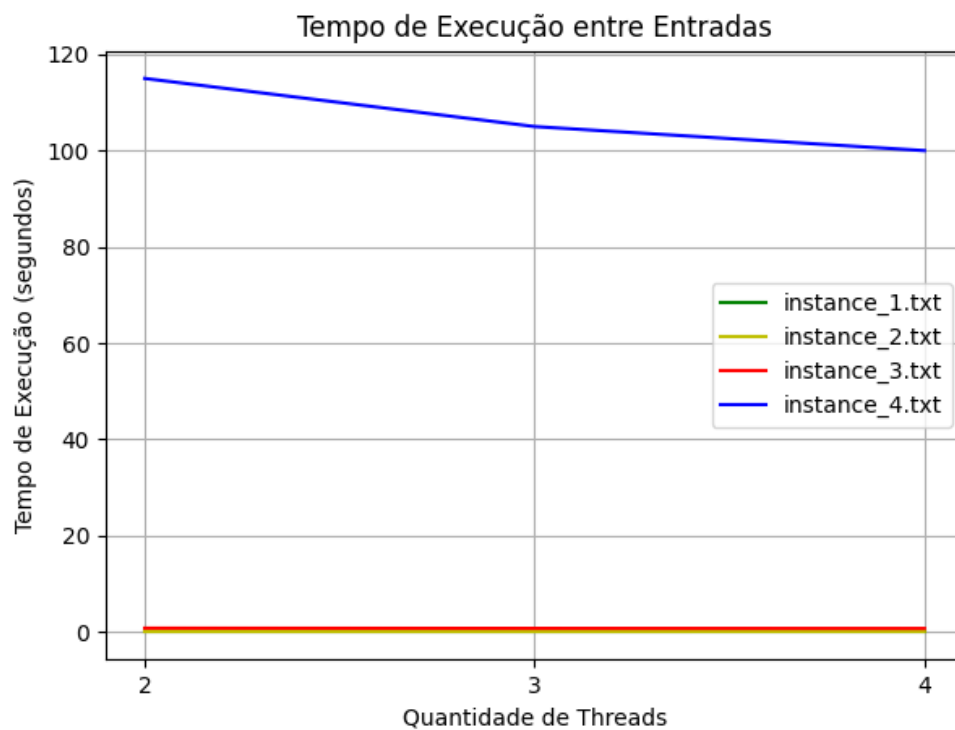


Figura 7: Tempos de execução variando número de *threads* e entradas

5 Limitações

Como principais limitações, é nítido que o *speed up* sublinear pode ser analisado mais a fundo, na tentativa de eliminar regiões críticas e otimizar certas porções do código. Seria necessário uma análise mais detalhada de como as diretivas OpenMP estão sendo utilizadas, quais otimizações podem ser feitas para melhorar os resultados ou até se existem erros de implementação que estão provocando uma melhora apenas discreta no tempo de execução.

Além disso, também pode ser de grande relevância analisar como os dados sobre o ecossistema são armazenados. Nesta implementação, por questões de simplificação são utilizadas duas matrizes para manipulá-lo e representá-lo. No entanto, idealmente deveria ser utilizada somente uma matriz para realizar todas as operações, ou até mesmo uma lista, que neste caso, armazenaria somente as posições do ecossistema em que existe um objeto associado (posições vazias não seriam armazenadas).

Refatorações deste tipo possivelmente proveriam uma redução significativa na memória utilizada, principalmente para ecossistemas esparsos (grande quantidade de posições vazias). É importante citar que tais mudanças poderiam também elevar o número de operações necessárias para simulação do ecossistema, possivelmente aumentando o custo computacional em detrimento da menor utilização de armazenamento.

6 Conclusão

Neste trabalho, foi explorada uma abordagem de computação paralela para ambiente de memória compartilhada utilizando a biblioteca OpenMP para um problema bastante simples, mas no qual foi possível explorar conceitos como decomposição de domínio e paradigmas SMPD (Single Program Multiple Data).

Com este exemplo didático, podemos observar de forma nítida, a complexidade por trás de construir um programa que será executado em todas as *threads* de forma indiscriminada. Por isto, sempre é necessário ter extrema cautela no controle de dependências de dados e regiões de risco (críticas) entre as *threads*, assim como a alocação e o acesso à memória realizado por cada um.

Referências

- Parallel Programming with OpenMP: https://web.njit.edu/~shahriar/class_home/HPC/omp3.pdf;
- OpenMP Programming: https://www.cs.cmu.edu/afs/cs/academic/class/15418-s21/www/lectures/rec_04.pdf;
- Profiling Programs With prof, gprof, and tcov: <https://docs.oracle.com/cd/E19059-01/stud.10/819-0493/OtherTools.html>;
- Unions: <https://www.learn-c.org/en/Unions>;
- Working with a union of structs in C: <https://stackoverflow.com/questions/20752551/working-with-a-union-of-structs-in-c>;
- How to use gettimeofday function in C language?: https://linuxhint.com/gettimeofday_c_language/;
- Material disponível no SIGAA na disciplina de Computação Paralela.