Lucas Saavedra Vaz - 120503, Maíra Baptista - 111816

Relatório - Programação *Multithreaded* para o Jogo da Vida

São José dos Campos - Brasil

Dezembro de 2020

Lucas Saavedra Vaz - 120503, Maíra Baptista - 111816

Relatório - Programação *Multithreaded* para o Jogo da Vida

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Programação Concorrente e Distribuída.

Docente: Prof. Dr. Álvaro Luiz Fazenda

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Dezembro de 2020

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Cálculos computacionais são chamados de concorrentes quando podem ser executados de forma arbitrária obtendo o mesmo resultado. O contrário acontece em execuções sequenciais, onde cada passo é feito em um ordem pré definida (1).

O termo programação concorrente denota a confecção de um programa tendo em mente unidades independentes, ou unidades de concorrência. A grande vantagem da programação concorrente é a modelagem mais próxima de problemas reais, além de obter-se um melhor aproveitamento das capacidades de hardware (1).

Neste relatório será realizado um estudo sobre as diferenças de performance entre programação concorrente e sequencial, utilizando como objeto de comparação, uma implementação do Jogo da Vida, de John H. Conway (2).

O Jogo foi implementado utilizando as ferramentas de *multi-threading* disponibilizados pelas bibliotecas *OpenMP* e *Pthreads* (em C++) e da classe *Threads* (em Java). Nas Seções a seguir serão discutidos pontos de interesse para cada uma das implementações (Seção 2) e as medidas de desempenho obtidas para cada um dos casos (Seção 3). Além disso, também trás as especificações de hardware da máquina utilizada para executar os experimentos.

No fim do relatório comparamos os diferentes métodos estudados para concluir qual o mais satisfatório para ser utilizado em nosso caso (Seção 4).

2 Implementação

Esta Seção aborda os pontos mais relevantes para para cada implementação do Jogo da Vida (*Pthreads* em C++, *OpenMP* em C++ e *Threads* em Java). Para este relatório, foi escolhido não focar em pontos que não são atrelados ao paralelismo, ou seja, pontos que são particulares a uma implementação qualquer do Jogo da Vida. Todos os testes foram executados em um tabuleiro de 2048 por 2048 por 2000 gerações.

2.1 *OpenMP*

Uma característica importante da biblioteca *OpenMP* é sua facilidade de uso. Ela proporciona ao programador um jeito mais simples de execução de tarefas em paralelo. Desta forma, a implementação utilizando *OpenMP* é muito similar à uma implementação puramente sequencial, e poucos trechos precisam de alteração.

Nesta implementação, foi escolhido executar a paralelização no momento em que cada célula é atualizada para uma nova rodada. Desta forma, é possível que múltiplos trabalhadores trabalhem em diferentes partes do tabuleiro ao mesmo tempo, sem alterar o resultado final.

A Listagem 2.1 mostra o método `Grid_Update`. Este método abre uma seção de execução paralela através de `#pragma omp parallel private(X, Y)shared(Grid, New_Grid)`. Aqui, as variáveis `X` e `Y` são privadas, ou seja, cada uma das *threads* invocadas pela biblioteca terão suas próprias variáveis `X` e `Y`. As variáveis `Grid` e `New_Grid` são compartilhadas por todas as *threads*, e representam o tabuleiro atual e o próximo, respectivamente.

```
1  void Grid_Update(vector<vector<bool>> &Grid, vector<vector<bool>> &New_Grid)
2  {
3      int X, Y;
4
5      #pragma omp parallel private(X, Y) shared(Grid, New_Grid)
6      {
7          #pragma omp for collapse(2)
8              for (X = 0; X < (int)Grid.size(); X++)
9                  for (Y = 0; Y < (int)Grid.size(); Y++)
10                     New_Grid[X][Y] = Cell_Update(Grid, X, Y);
11     }
12 }
```

Listing 2.1 – Trecho de código do método que inicializa a região paralela

Ainda dentro da região paralela, uma nova diretiva é chamada: `#pragma omp for collapse(2)`. Ela permite que dois laços sejam executados e seus resultados sejam unidos no momento em que sua execução termina.

Os laços percorrem todas as linhas e colunas do tabuleiro, atualizando os valores para cada célula através do `Cell_Update`.

2.2 *Java Threads*

A implementação em Java é muito semelhante à implementação em C++ utilizando-se de *OpenMP*. A diferença ocorre nos passos extras que são necessários para garantir a paralelização, já que a Classe *Threads* em Java proporciona ao programador as mesmas facilidades que a *OpenMP*.

Na Listagem 2.2, é mostrado a criação de um vetor de *threads* (`thread_array`) que serão os trabalhadores do programa. Cada objeto dentro de `support_array` será capaz de fazer os cálculos necessários para atualização do tabuleiro.

```
1 Thread[] thread_array = new Thread[MAX_THREADS];
2 GridUpdate[] support_array = new GridUpdate[MAX_THREADS];
```

Listing 2.2 – Trecho de código do método mostra a crição do vetor the *threads*.

Na Listagem 2.3, é mostrado o laço que itera sobre todas as gerações do programa. Dentre deste laço é feita a divisão de trabalho para todas as *threads*.

```
1  for (i = 1; i <= N_GENERATIONS; i++){ //iterate trhough all generations
2
3          for(int idx = 0; idx < MAX_THREADS; idx++)
4          {
5              support_array[idx] = new GridUpdate(Grid, newGrid, MAX_THREADS, N, idx);
6              thread_array[idx] = new Thread(support_array[idx]);
7              thread_array[idx].start();
8          }
9
10         try {
11             for(aux_i = 0; aux_i < MAX_THREADS; aux_i++ ) {
12                 thread_array[aux_i].join();
13             }
14         } catch (InterruptedException e) {
15             System.out.println("Error ocurred");
16         }
17
18         Grid = newGrid;
19         newGrid = null;
20         newGrid = new boolean[N][N];
21
22         System.out.println("Generation " + (i) + ": " + CellsTotal(Grid));
23         System.out.println("End of a generation");
24
25 }
```

Listing 2.3 – Trecho de código do método mostra o laço que calcula todas as gerações.

O objeto `GridUpdate` possui em seu construtor (mostrado na Listagem 2.4) divisão das linhas do tabuleiro para cada uma das *threads* que é inicializada com ele. Os blocos de

linhas são divididos igualmente entre as *threads*, de modo à evitar overwork em uma delas e também para evitar que um *thread* atue na região de memória de outra.

```
1  public GridUpdate(boolean[][] _old, boolean[][] _new, int num_threads, int
       _grid_dimension, int _thread_id) {
2          Grid = _old;
3          newGrid = _new;
4          grid_dimension = _grid_dimension;
5          thread_id = _thread_id;
6
7          num_lines = grid_dimension/num_threads;
8          if(thread_id == 0) {
9              start_idx = 0;
10             end_idx = num_lines - 1;
11         } else if(thread_id == num_threads - 1) {
12             start_idx = grid_dimension - num_lines;
13             end_idx = grid_dimension - 1;
14         } else {
15             start_idx = num_lines * thread_id;
16             end_idx = start_idx + num_lines - 1;
17         }
18 }
```

Listing 2.4 – Trecho de código mostra o construtor da Classe a ser paralelizada

Feita as divisões de trabalho, o programa segue de forma essencialmente igual, a partir do laço apresentado na Listagem 2.5, onde o método `CellUpdate` é chamado para cada célula.

```
1  public void run() {
2      int idx;
3      for(idx = start_idx; idx <= end_idx; idx++) {
4          for(int inner_idx = 0; inner_idx < grid_dimension; inner_idx++) {
5              newGrid[idx][inner_idx] =  CellUpdate(Grid, idx, inner_idx);
6          }
7      }
8  }
```

Listing 2.5 – Trecho de código que mostra o funcionamento do método *run()*

2.3 *Pthreads*

A abordagem utilizando *PThreads* em C++ é uma expansão da versão sequencial, adicionando uma etapa de preparação onde as tarefas de cada *thread* são divididas previamente e então são alocados os processos para a execução (Listagem 2.6).

```
1  void Grid_Update(vector<vector<bool>> &Grid, vector<vector<bool>> &New_Grid)
2  {
3      pthread_t Threads[N_THREADS];
4      int X, Y;
5      int PT_Count = 0;
6      struct PT_Args Args[N_THREADS];
7      stack <pair <int, int>> Coord_Stack[N_THREADS];
8
9      for (X = 0; X < (int)Grid.size(); X++)
```

```
10      {
11          for (Y = 0; Y < (int)Grid.size(); Y++)
12          {
13              R_DEBUG(cout << "Coord: " << X << " " << Y << endl);
14              Coord_Stack[PT_Count % N_THREADS].push(make_pair(X, Y));
15              PT_Count++;
16          }
17      }
18
19      for (int i = 0; i < N_THREADS; i++)
20      {
21          Args[i].Grid = &Grid;
22          Args[i].New_Grid = &New_Grid;
23          Args[i].Coord_Stack = &(Coord_Stack[i]);
24          pthread_create(&Threads[i], NULL, Cell_Update, (void*) &(Args[i]));
25      }
26
27      for (int i = 0; i < N_THREADS; i++)
28          pthread_join(Threads[i], NULL);
29  }
```

Listing 2.6 – Trecho de código que divide as tarefas entre os *threads* e depois os aloca

Essa divisão prévia foi realizada com o proposito de diminuir a quantidade de sincronizações entre os *threads*, assim evitando tempo ocioso dentro dos processos.

Logo em seguida cada *thread* executa a função `Cell_Update` (Listagem 2.7) até sua lista de coordenadas a serem processadas estiver vazia. Após essa etapa, os processos são sincronizados e o cálculo de células vivas ocorre da mesma maneira que a versão sequencial do código.

```
1   void *Cell_Update(void *Args)
2   {
3       vector <vector<bool>> *Grid, *New_Grid;
4       stack <pair<int, int>> *Coord_Stack;
5       int X, Y;
6
7       Grid = ((struct PT_Args*) Args)->Grid;
8       New_Grid = ((struct PT_Args*) Args)->New_Grid;                    //Extracts data
            from Args Struct
9       Coord_Stack = ((struct PT_Args*) Args)->Coord_Stack;
10
11      while(!(*Coord_Stack).empty())
12      {
13          X = (*Coord_Stack).top().first;
14          Y = (*Coord_Stack).top().second;
15          (*Coord_Stack).pop();
16
17          int Nb_Count = Neighbours_Count(Grid, X, Y);
18
19              if ((*Grid)[X][Y] == 0)
20              {
21                  if (Nb_Count == 3)
22                  {
23                      (*New_Grid)[X][Y] = 1;
24                      continue;
25                  }
26              }
```

```
27              else
28              {
29                  if (Nb_Count < 2 || Nb_Count >= 4)
30                  {
31                      (*New_Grid)[X][Y] = 0;
32                      continue;
33                  }
34              }
35
36          (*New_Grid)[X][Y] = (*Grid)[X][Y];
37      }
38
39      pthread_exit(NULL);
40  }
```

Listing 2.7 – Trecho de código paralelizado que cada *thread* executa

3 Medidas de Desempenho

Nesta Seção, serão apresentados as medidas de desempenho para cada uma das implementações.

A máquina que executou os experimentos possui um processador Ryzen 5 2600 com 6 cores e 12 *threads*, e 16Gb de memória RAM em uma frequência de 3200MHz. As versões dos compiladores utilizados são:

- g++ Ubuntu 9.3.0-17ubuntu1 20.04;
- OpenJDK Ubuntu 14.0.2+12-1 20.04.

As medidas de desempenho escolhidas para este relatório são o tempo *speedup* e Eficiência.

Speedup é a métrica que relaciona o tempo gasto para realizar uma atividade em um único processador e em um número N de processadores. Possui a seguinte fórmula:

Já a Eficiência é a medida de desempenho que relaciona o *speedup* com o número de processadores em si.

A seguir, são mostrados os resultados para cada implementação feita. Os experimentos foram feitos variando-se a o número de processadores envolvidos na computação, com 1, 2, 4, 8 e 12 *threads*. Para todos os experimento em todas as implementações, o tempo de execução foi medido para o laço que computa as gerações sucessivas.

3.1 *OpenMP*

A Tabela 1 mostra os resultados para a implementação com *OpenMP*. Foram feitos três experimentos e feita uma média entre os tempos de execução de cada um. As colunas *speedup* e Eficiência mostram os valores de *speedup* e eficiência para cada contagem de *threads*.

N Threads	Execução 1	Execução 2	Execução 3	Média	Speedup	Eficiência
1	956.08	953.63	955.84	955.18	1	100.00%
2	502.86	504.52	503.50	503.62	1.896616656	94.83%
4	275.92	271.62	272.94	273.49	3.492543953	87.31%
8	229.74	230.80	228.25	229.60	4.160277415	52.00%
12	165.72	163.35	167.91	165.66	5.765881328	48.05%

Tabela 1 – Medidas de desempenho entre quantidades variadas de *threads*, usando *OpenMP*.

A Figura 1 resume os dados da Tabela 1.

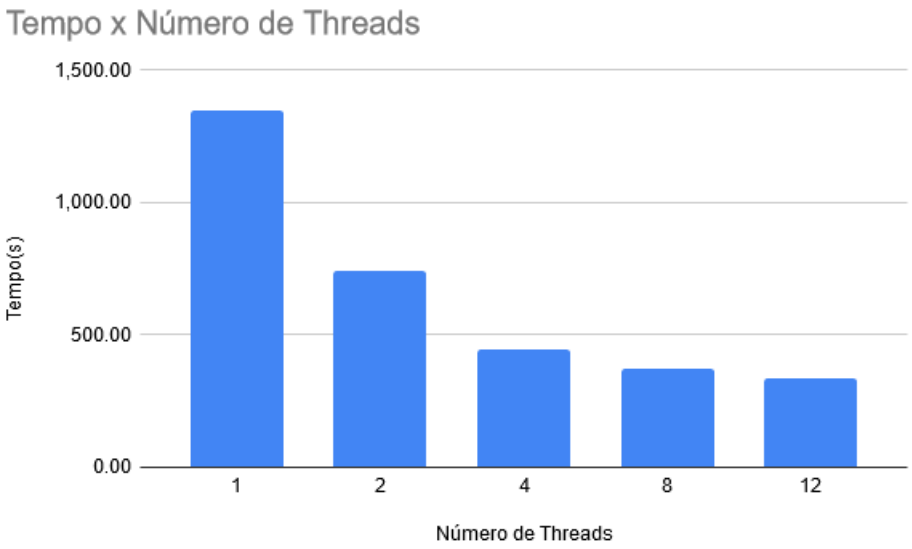

Figura 1 – Medidas de tempo usando *JavaThreads*

O Figura 2 mostra uma comparação entre o *speedup* real entre o número de *threads* e o *speedup* perfeito.

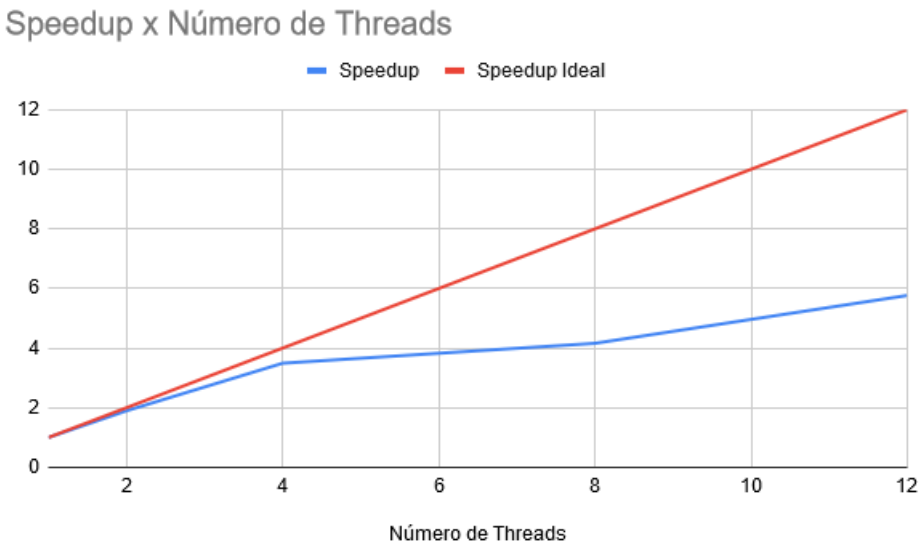

Figura 2 – Comparação entre *speedup* ideal e real para a implementação usando *OpenMP*

3.2 *JavaThreads*

Por sua vez, a Tabela 2 mostra os resultados para a implementação do Jogo da Vida usando *JavaThreads*. Novamente, o Jogo foi executado três vezes e uma média dos

tempos de execução foi tirada.

N Threads	Execução 1	Execução 2	Execução 3	Média	Speedup	Eficiência
1	1,354.55	1,347.50	1,343.70	1,348.58	1	100.00%
2	755.73	751.13	727.08	744.65	1.811037897	90.55%
4	442.11	441.93	450.74	444.93	3.031023839	75.78%
8	381.58	366.40	361.04	369.67	3.648040612	45.60%
12	339.10	339.30	334.70	337.70	3.993435989	33.28%

Tabela 2 – Medidas de desempenho entre quantidades variadas de *threads*, usando *JavaThreads*

A Figura 3 resume os dados apresentados na Tabela 2.

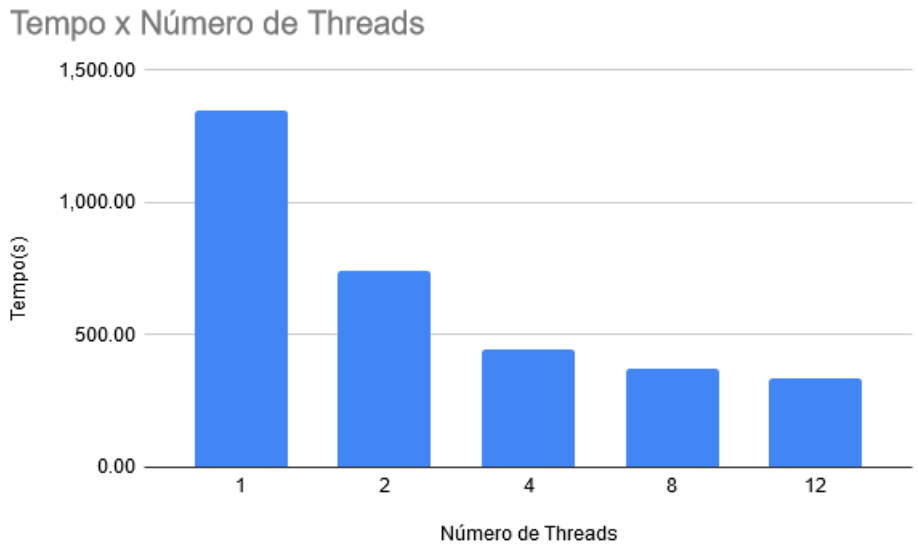

Figura 3 – Medidas de tempo usando *JavaThreads*

A Figura 4 mostra a comparação entre o *speedup* real entre o número de *threads* e o *speedup* perfeito.

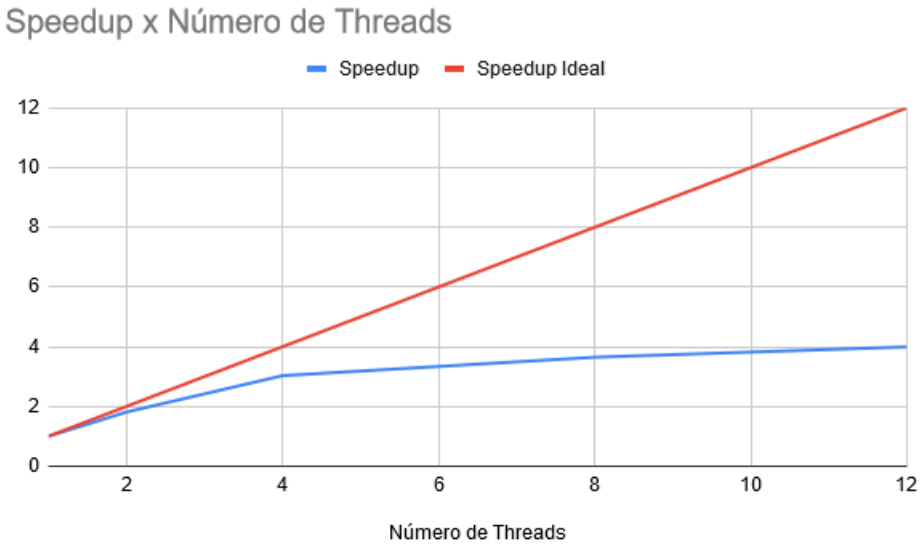

Figura 4 – Comparação entre *speedup* ideal e real para Implementação usando *JavaThreads*

3.3 *PThreads*

Por fim, a Tabela 3, mostra os resultados para as execuções utilizando-se de *Pthreads*. A média dos tempos de execução foi feita com base em três execuções e a Figura 6 mostra um comparativo entre o *speedup* real e o *speedup* ideal para esta implementação.

N Threads	Execução 1	Execução 2	Execução 3	Média	Speedup	Eficiência
1	1,048.00	1,097.00	1,080.00	1,075.00	1	100.00%
2	651.00	667.00	644.00	654.00	1.643730887	82.19%
4	535.00	530.00	528.00	531.00	2.024482109	50.61%
8	379.00	388.00	388.00	385.00	2.792207792	34.90%
12	321.00	328.00	320.00	323.00	3.328173375	27.73%

Tabela 3 – Medidas de desempenho entre quantidades variadas de *threads*, usando *PThreads*

A Figura 5 resume os dados apresentados na Tabela 3.

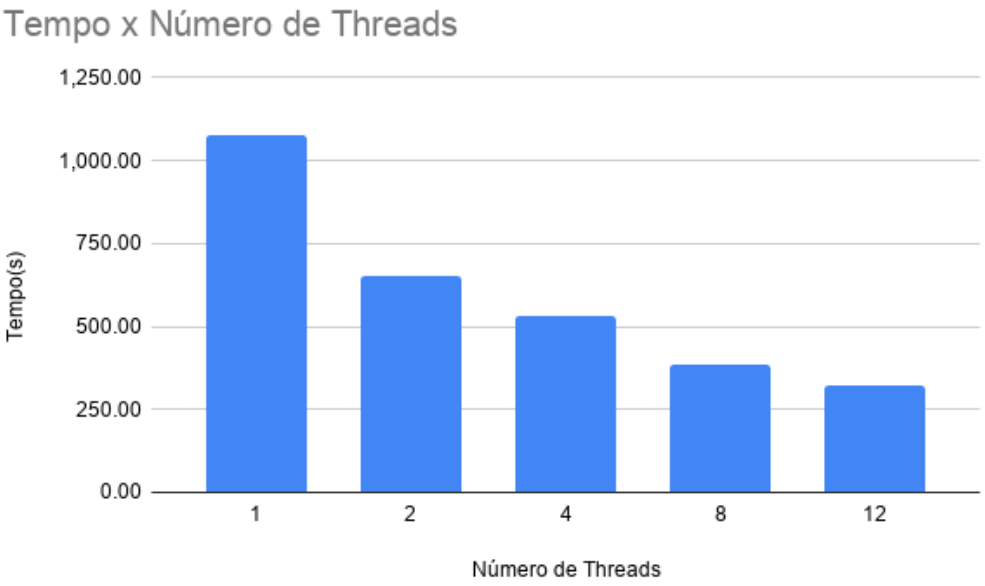

Figura 5 – Medidas de tempo usando *PThreads*

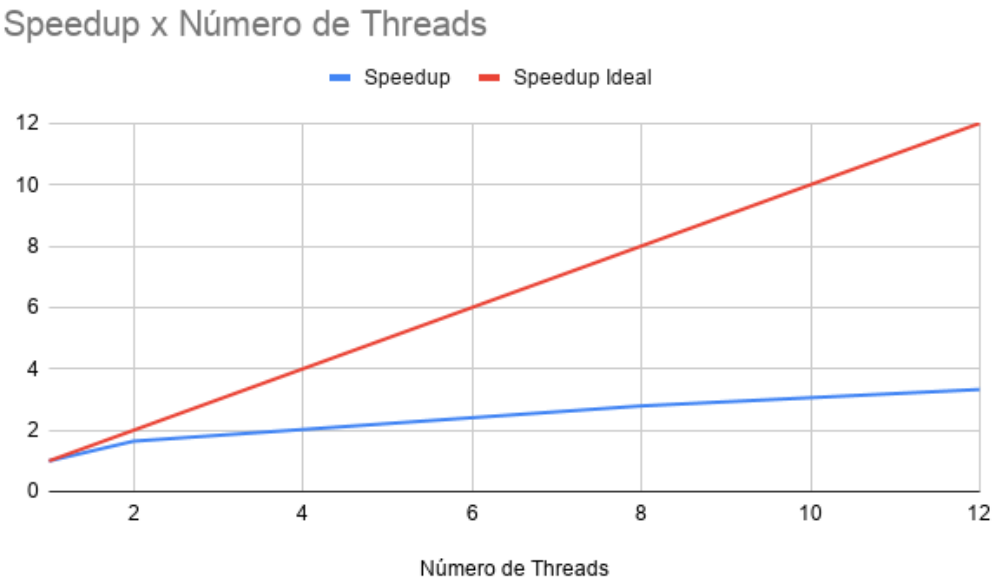

Figura 6 – Comparação entre *speedup* ideal e real para Implementação usando *PThreads*

4 Resultados

Comparando os resultados obtidos (que podem ser acessados neste link), podemos analisar qual dos métodos possuí o melhor desempenho para nosso caso de teste e sistema utilizado. Podemos ver na Figura 7 que o método no qual houve destaque foi o *OpenMP*, sendo o mais rápido dos candidatos. Em segundo lugar temos a implementação utilizando *PThreads* que, apesar de ao aumentar o número de *threads* sendo executadas se iguala ao método *JavaThreads*, com poucos processos paralelos fica entre as outras implementações.

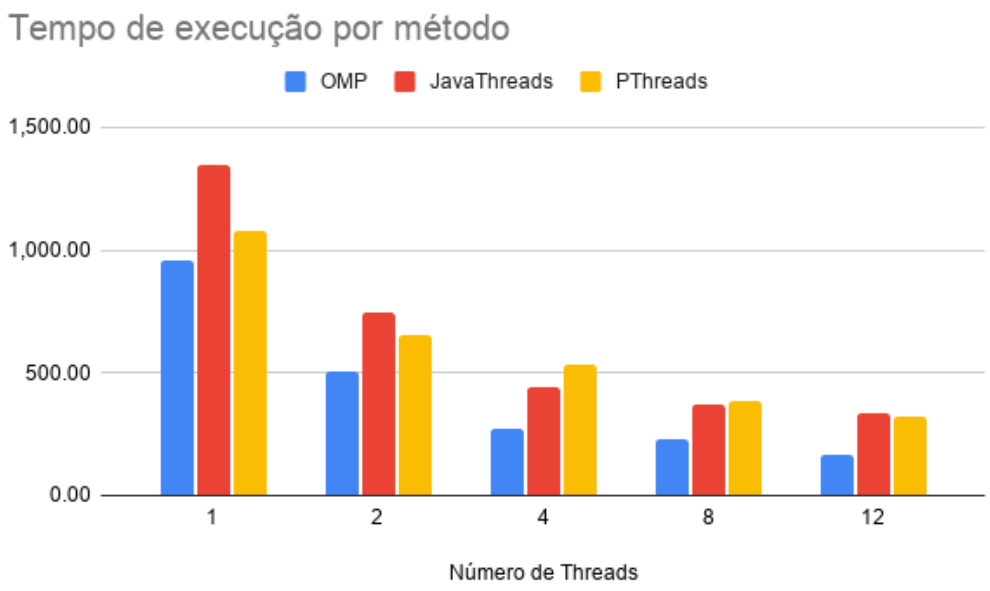

Figura 7 – Comparação de medidas de tempo obtidas

Analisando o *Speedup* (Figura 8) e a eficiência (Figura 9) entre os diferentes códigos, novamente o *OpenMP* fica na liderança, porém desta vez as métricas da implementação com *JavaThreads* são melhores que as de *PThreads*. Isso significa que há um *delta* maior ganho relativo de desempenho quando se paraleliza utilizando *JavaThreads* do que *PThreads*.

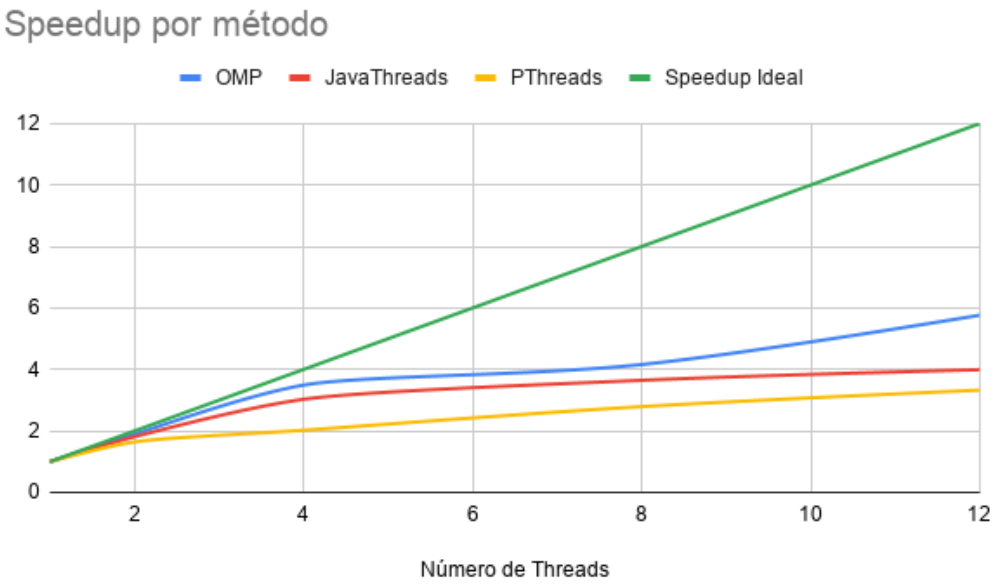

Figura 8 – Comparação de medidas de *Speedup* obtidas

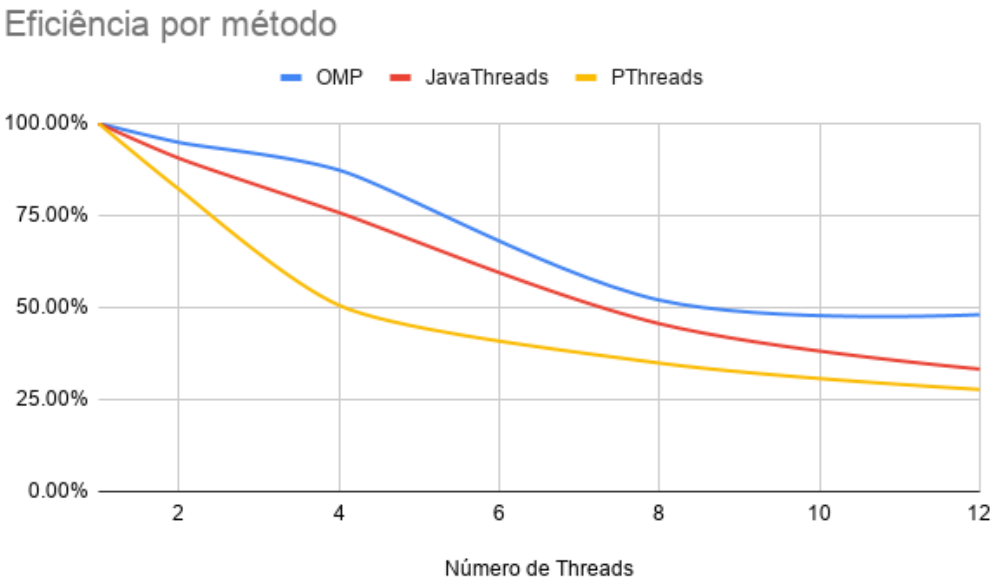

Figura 9 – Comparação de medidas de eficiência obtidas

Portanto, analisando os resultados, podemos concluir que o método mais satisfatório de implementação de paralelismo em nossos casos de testes e algoritmos desenvolvidos é utilizando a biblioteca *OpenMP*.

Referências

1 PADUA, D. *Encyclopedia of parallel computing*. [S.l.]: Springer Science & Business Media, 2011. Citado na página 5.

2 GARDNER, M. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, vol, v. 223, n. 4, p. 120–123, 1970. Citado na página 5.