ID: Abner Canellas; Jonatas Oliveira; Raphael Autuori

Atividade de Programação 1 Jogo da Vida - PThreads/OpenMP/JavaThreads

São José dos Campos - Brasil

Dezembro de 2020

ID: Abner Canellas; Jonatas Oliveira; Raphael Autuori

Atividade de Programação 1 Jogo da Vida - PThreads/OpenMP/JavaThreads

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

Docente: Prof. Dr. Alvaro Luiz Fazenda e Prof. Dra. Denise Stringhini

Discentes: Abner Y. D. C. Canellas, RA: 150738 ; Jonatas da Silva Oliveira, RA: 99640; Raphael Gomes Autuori, RA:120569.

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Dezembro de 2020

Resumo

Este relatório tem por finalidade apresentar uma aplicação da teoria de Programação Concorrente Distribuída, implementando o Jogo da Vida, criado por John H. Conway, em versão serial e três (3) versões concorrentes deste código, em linguagem C/C++ utilizando *PThreads*,em linguagem C/C++ utilizando *OpenMP*, e outra em *Java* com *JavaThreads*.

Palavras-chaves: Programação Concorrente Distribuída, Jogo da Vida, PThreads, OpenMP, JavaThreads.

Lista de ilustrações

Sumário

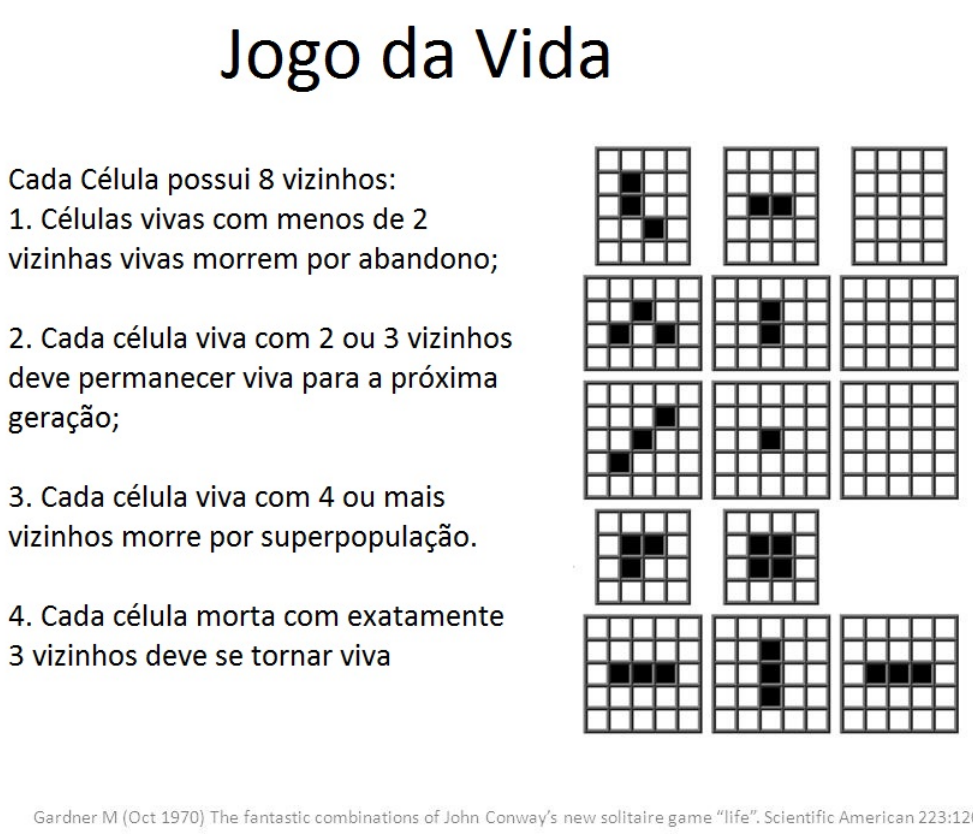
1 Introdução

O jogo da vida, proposto pelo matemático John Conway, é um jogo jogado de forma bidimensional num tabuleiro de matriz NxN, com suas células quadradas idênticas nas respectivas linhas e colunas. Tal jogo não possui jogadores e seu objetivo é simular gerações sucessivas de uma determinada sociedade de organismos vivos. Cada célula do tabuleiro, possui 8 células vizinhas, e tem somente dois estados booleanos possíveis durante o jogo: 1, representando que a célula está viva, ou 0, representando que ela está morta. Uma célula viva necessita de outras células vivas para sobreviver e procriar, mas caso haja um excesso de células vivas ao seu redor ocorre a morte das mesmas devido à escassez de alimento. As regras pré-definidas, mostradas na Figura 1, são aplicadas a cada nova geração e são as seguintes:

- Células vivas com menos de 2 (dois) vizinhas vivas morrem por abandono;
- Cada célula viva com 2 (dois) ou 3 (três) vizinhos deve permanecer viva para a próxima geração;
- Cada célula viva com 4 (quatro) ou mais vizinhos morre por superpopulação;
- Cada célula morta com exatamente 3 (três) vizinhos deve se tornar viva.

Assim, a partir de uma geração da sociedade, representada pelas células vivas, é possível gerar e visualizar uma imagem no tabuleiro. E com a evolução dessa geração, (através da aplicação das regras, e recontagem de células vivas), percebem-se mudanças muitas vezes inesperadas a cada nova geração, variando a imagem de padrões fixos a caóticos. O objetivo desse trabalho foi implementar três (3) versões concorrentes do jogo da vida, em linguagem C/C++ utilizando utilizando *PThreads*,em linguagem C/C++ utilizando *OpenMP*, e outra em *Java* com *JavaThreads*, em um tabuleiro finito, NxN com bordas infinitas, (a fronteira esquerda liga-se com a fronteira direita e a fronteira superior liga-se com a fronteira inferior) a fim de colocar em prática os conceitos de Programação concorrente em Sistemas de Memória Compartilhada.

Figura 1 – Regras do Jogo da Vida



2 Objetivos

2.1 Geral

O seguinte relatório tem por finalidade verificar o desempenho (tempo de processamento) para a versão serial e versões concorrentes em C/C++ (*PThreads* e *OpenMP*) e *Java*, variando a quantidade de threads em 1, 2, 4 e 8 (mesmo que a máquina testada não tenha essa quantidade de núcleos). Considerou-se ainda um tabuleiro quadrado de dimensões 2048*2048 e um total de 2000 gerações sucessivas desse tabuleiro.

Tal atividade proporciona uma introdução aos estudos de Programação Multithread e sedimenta a prática dos conceitos ministrados na disciplina Programção Concorrente Distribuída.

2.2 Específico

Para alcançar o objetivo geral apresentado, foi estabelecido as seguintes etapas a serem realizadas:

- Implementação do código serial em linguagem C.
- Implementação do código utilizando *PThreads*, em linguagem C.
- Implementação do código utilizando *OpenMP* em linguagem C.
- Implementação do código utilizando *Java*.
- Realizar a verificação do desempenho dos códigos.
- Realizar simulações de teste.
- Analisar o funcionamento de cada código.

3 Fundamentação Teórica

Para possibilitar o desenvolvimento do projeto foi necessário entender os conceitos teóricos da disciplina de Programção Concorrente Distribuída.

3.1 Programação Concorrente e Distribuída

A programação concorrente trata-se do estudo, desenvolvimento e aplicação de um conjunto de programas sequenciais (ou de processos) que podem ser executados em paralelo (paralelismo em potencial), compartilhando dos mesmos recursos da máquina.

3.2 Threads

Thread é uma tarefa que um determinado programa realiza. Trata-se de um processo "leve", que possui um fluxo de execução interno a um dado processo, apontadores de programa e de pilha próprios. A Thread é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrencialmente.

3.3 Programação Multithread -*PThreads*

POSIX threads é um padrão POSIX para threads, o qual define uma API padrão para criar e manipular threads. As bibliotecas que implementam a POSIX threads são chamadas Pthreads.

3.4 Programação Multithread - *OpenMP*

O OpenMP (Open Multi-Processing, ou Multi-processamento aberto) é uma interface de programação de aplicativo (API) para a programação multi-processo de memória compartilhada em múltiplas plataformas. Permite acrescentar simultaneidade aos programas escritos em C, C++ e Fortran sobre a base do modelo de execução fork-join.

3.5 Programação Multithread -*Java Threads*

Possui o funcionamento similar a *PThreads* mas com orientação a objetos.

3.6 Implementação Serial

Para implementar a versão serial do código, escrito em linguagem C e que foi base para os demais códigos, primeiramente foi definida algumas constantes e feita a alocacao de memória necessária no Main para construir a matriz, porém em um vetor, a fim de melhorar a performance de acesso em memória, conforme a figura abaixo.

Figura 2 – Constantes e Bibliotecas

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define SRAND_VALUE 1985
#define DIMENTION 2048 //2048*2048 e um total de 2000 geracoes
#define NGENERATIONS 2000
```

Figura 3 – Main

```c
int main(){

    //declara wallTime pra teste de performance
    struct timeval begin, end;
    //inicia a contagem de wallTime
    gettimeofday(&begin,0);

   int *grid = (int *)malloc(DIMENTION * DIMENTION *sizeof(int)), *auxGrid = (int *)malloc(DIMENTION * DIMENTION *sizeof(int));
```

Logo em seguida essa matriz foi aleatoriamente populada com a seguinte função:

Figura 4 – Função Populate

```c
void populate(int *grid){ //popula aleatoriamente a matriz
    srand(SRAND_VALUE);
    int i, j;
    for(i = 0; i<DIMENTION; i++) {
        for(j = 0; j<DIMENTION; j++) {
            grid[i * DIMENTION + j] = rand() % 2;
        }
    }
}
```

Depois de populada a matriz, foi chamada uma função para contar o total de células vivas:

Figura 5 – Função CountFinalCells

```c
void countFinalCells(int *grid, int generation){
    int n=0, i, j;

    for(i=0; i < DIMENTION; i++){
        for (j = 0; j < DIMENTION; j++)
            if(grid[i * DIMENTION + j]==1) n++ ;
    }

    printf("Geracao %d: %d\n", generation, n);
}
```

Após a primeira varredura de células vivas, foi usado um comando for para criar a alocação de uma matriz auxiliar e foram usadas as seguintes funções a fim de popular as 2000 gerações, seguindo as regras pré estabelecidas pelo jogo:

Figura 6 – For para as 2000 novas gerações

```
for(int i = 1; i<=NGENERATIONS; i++){
    newGen(grid, auxGrid);
    countFinalCells(grid, i);
}
```

Função usada para atualizar a geração:

Figura 7 – Função newGen

```
void newGen(int *gen, int* auxGen){

    int i,j;
    for(i=0; i < DIMENTION; i++){
        for (j = 0; j < DIMENTION; j++){
            auxGen[i * DIMENTION + j] =  newCellState(gen, i, j);
        }
    }
    for (i=0;i<(DIMENTION*DIMENTION);i++){
        gen[i]=auxGen[i];
    }
}
```

Na função newGen é chamada a função newCellState que define o futuro da celula de acordo com as regras:

Figura 8 – Função newCellState

```
int newCellState(int* gen, int i, int j){
    int n = getNeighbors(gen, i,j); //conta vizinhos

    if(gen[i * DIMENTION + j] == 1){ //se a celula esta viva
        if(n < 2){
            return 0;  //Morre por abandono                    ;
        }else if(n == 2 || n == 3){
            return 1;  //Permanece viva
        }else if(n >= 4){
            return 0;  //Morre por superpopulacao
        }
    }
    else if(n==3){ //se a celula esta morta
        return 1;  //Ganha vida
    }
}
```

Logo em seguida, dentro da função newCellState é chamada a função getNeighbors para contar o número de vizinhos atualmente vivos:

Figura 9 – Função newCellState

```
int getNeighbors(int *grid, int i, int j){ //conta numero de vizinhos atualmente vivos
    int n=0, ni = i-1, nj = j-1, pi = i+1, pj = j+1 ; // ni, nj, pi, pj representam i-1, j-1, i+1, j+1

    ni = ni%DIMENTION < 0 ? DIMENTION-1 : ni;
    nj = nj%DIMENTION < 0 ? DIMENTION-1 : nj;
    pi = pi%DIMENTION;
    pj = pj%DIMENTION;


    n = grid[ni*DIMENTION +nj] + grid[ni*DIMENTION +j] + grid[ni*DIMENTION +pj] +
        grid[i*DIMENTION +nj] + grid[i*DIMENTION +pj] +
        grid[pi*DIMENTION +nj] + grid[pi*DIMENTION +j] + grid[pi*DIMENTION +pj];
 return n;
}
```

Por fim foi usada a função gettimeofday da biblioteca <sys/time.h> no início e fim do código para realizar a contagem de wallTime a fim de printar na tela.

Figura 10 – Função gettimeofday

```
gettimeofday(&end,0);
long sec = end.tv_sec - begin.tv_sec;
long mic = end.tv_usec - begin.tv_usec;
double elap = sec - mic*1e-6;

printf("Tempo total: %.4f secs", elap);
```

3.7 Implementação Pthreads

O código serial foi usado na implementação da versão concorrente com Pthreads, pertencente a biblioteca <pthread.h>. A princípio foram declaradas as bibliotecas, constantes e seguintes estruturas de dados para lidar com as Threads:

Figura 11 – Constantes e Bibliotecas Pthreads

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>

#define SRAND_VALUE 1985
#define DIMENTION 2048 //2048*2048 e um total de 2000 geracoes
#define NGENERATIONS 2000
#define THREADS 2
```

Figura 12 – Struct das Threads

```
struct argsCF_t{
    int id, n;
    int* gen;
};
typedef struct argsCF_t argsCF_t;

struct argsNCS_t{
    int id, i, j;
    int* gen;
};
typedef struct argsNCS_t argsNCS_t;

pthread_t pt[THREADS];
argsCF_t aCF[THREADS];
argsNCS_t aNCS[THREADS];
```

A função newGen, utilizada no código serial para atualizar a geração sofreu modificações a fim de dividir o processamento entre as Threads:

Figura 13 – newGen PThreads

```
void newGen(int *gen, int *auxGen){

    for (int t = 0; t < THREADS; t++){
        aNCS[t].id=t;
        aNCS[t].gen=gen;
        aNCS[t].auxGen=auxGen;
        pthread_create(&pt[t],NULL, newCellState, (void*)&aNCS[t]);
    }
    for (int t = 0; t < THREADS; t++){
        pthread_join(pt[t],NULL);
        for (int i = t*DIMENTION; i < DIMENTION/THREADS; i++){
            auxGen[i] = (int)aNCS[t].auxGen[i];
        }
    }
    for (int i=0;i<(DIMENTION*DIMENTION);i++){
        gen[i]=auxGen[i];
    }
}
```

No Main do código serial houve também atualizações a fim de criar as Threads:

Figura 14 – PThreads

```
for(int i = 1; i<=NGENERATIONS; i++){
    total=0;
    newGen(grid, auxGrid);

    for(int t = 0; t<THREADS; t++){
        aCF[t].id=t;
        aCF[t].gen = grid;
        pthread_create(&pt[t],NULL, countFinalCells, (void*)&aCF[t]);
    }
    for(int t = 0; t<THREADS; t++){
        pthread_join(pt[t],NULL);
        total+=aCF[t].n;
    }
    printf("Geracao %d: %d\n", i, total);

    //printf("Geracao %d: %d\n");
}
```

3.8 Implementação openMP

O código serial também foi usado na implementação da versão concorrente com openMP, pertencente a biblioteca <omp.h>.

Figura 15 – Constantes e Bibliotecas openMP

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <sys/time.h>

#define SRAND_VALUE 1985
#define DIMENTION 2048 //2048*2048 e um total de 2000 geracoes
#define NGENERATIONS 2000
#define THREADS 8
int b=0;
```

As funções onde foram paralelizados os processos com openMP são as seguintes:

Figura 16 – newGen openMP

```
void newGen(int *gen, int* auxGen){ //funcao que realiza a atualização da geracao

    int i,j;
    #pragma omp parallel shared(gen, auxGen) private(i,j) num_threads(THREADS)
    {
        #pragma omp for
        for(i=0; i < DIMENTION; i++){
            for (j = 0; j < DIMENTION; j++){
                auxGen[i * DIMENTION + j] =  newCellState(gen, i, j);
        }
    }
    }
    for (i=0;i<(DIMENTION*DIMENTION);i++){
        gen[i]=auxGen[i];
    }
}
```

Figura 17 – countFinalCells openMP

```
void countFinalCells(int *grid, int generation){ //Conta o total de celulas vivas no fim da i geração
    int n=0, i, j;
    #pragma omp parallel shared(grid) private(i,j) reduction(+:n) num_threads(THREADS)
    {
        #pragma omp for
        for(i=0; i < DIMENTION; i++){
            for (j = 0; j < DIMENTION; j++)
                if(grid[i * DIMENTION + j]==1) n++ ;
        }
    }
    printf("Geracao %d: %d\n", generation, n);
}
```

3.9 Implementação Java

O código serial também foi usado como base para a implementação da versão concorrente em linguagem java. Declarações:

Figura 18 – Declarações Java

```java
package JavaThread;

import java.util.Random;
import java.lang.Thread;

//Lembrar de compilar com javac JavaThread/jogoDaVidaJavaThread.java
// e exec com java JavaThread.jogoDaVidaJavaThread

public class jogoDaVidaJavaThread {

    static private int DIMENTION = 2048; //2048*2048 e um total de 2000 geracoes
    static private int NGENERATIONS = 2000;
    static private int THREADS = 2;

    static private countFinalCellsThread[] cFCT = new countFinalCellsThread[THREADS];
    static private newGenThread[] nGT = new newGenThread[THREADS];
    static private Thread[] th = new Thread[THREADS*2];

    static private void populate(int[] grid){
        Random gerador = new Random(1985);
        int i, j;
        for(i = 0; i<DIMENTION; i++) {
                for(j = 0; j<DIMENTION; j++){
                    grid[i * DIMENTION +j] = gerador.nextInt(2147483647) % 2;
            }
        }
    }
```

Figura 19 – Main Java

```java
    public static void main(String[] args) {

        long startTime = System.nanoTime();

        int[] grid = new int[DIMENTION * DIMENTION];
        int[] auxGrid = new int[DIMENTION * DIMENTION];

        populate(grid);

        //printField(grid,0);
        countFinalCells(grid, 0);

        for(int i=1; i<=NGENERATIONS; i++){
            grid = newGen(grid, auxGrid).clone();
            countFinalCells(grid,i);
            //printField(grid,i);
        }

        long endTime = System.nanoTime();
        long elapsed = (endTime - startTime) / 1000000;
        System.out.println("Tempo total: " + elapsed + "ms");
    }

}
```

Funções com Threads:

Figura 20 – newGen Java

```
static private int[] newGen(int[] gen, int[] auxGen){

    for(int t = 0; t< THREADS; t++){
        nGT[t] = new newGenThread(gen, auxGen, t, DIMENTION, THREADS);
        th[t*2] = new Thread(nGT[t]);
        th[t*2].start();
    }
    for(int t = 0; t < THREADS; t++){
        try {
            th[t*2].join();
            for(int i=t; i < DIMENTION; i+=THREADS){
                for (int j = 0; j < DIMENTION; j++){
                    auxGen[i * DIMENTION +j] = nGT[t].getNauxGen()[i * DIMENTION +j];
                }
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
    return auxGen;
}
```

Figura 21 – countFinalCells Java

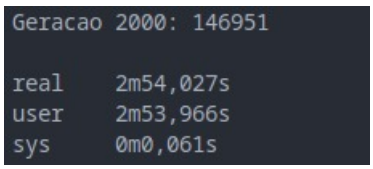
```
    static private void countFinalCells(int[] grid, int generation){
        int n=0;

        for(int t = 0; t < THREADS; t++){
            cFCT[t] = new countFinalCellsThread(grid, t, DIMENTION, THREADS)
            th[t] = new Thread(cFCT[t]);
            th[t].start();
        }
        for(int t = 0; t < THREADS; t++){
            try {
                th[t].join();
                n+=cFCT[t].getN();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
        System.out.println("Geracao " + generation + ": " + n);
    }
```

4 Resultados Obtidos e Discussões

Os códigos acima foram testados na seguinte máquina: Intel(R) Core(TM) i5-8265U CPU @8x1.80 GHz 8,00 GB (7,85 GB usable) 64-bit operating system, x64-based processor nVidea MX230 - 2GB.

4.1 Código Serial

Figura 22 – Tempo de Processamento do Código Serial



4.2 Código utilizando *PThreads*.

Figura 23 – Tempo de Processamento Pthreads com 2 Threads

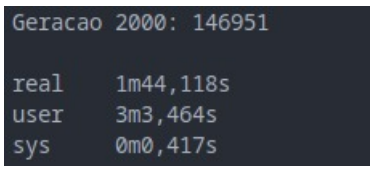


Figura 24 – Tempo de Processamento Pthreads com 4 Threads

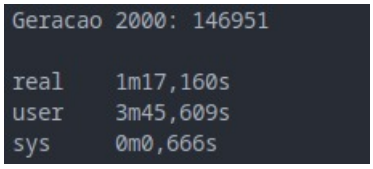


Figura 25 – Tempo de Processamento Pthreads com 8 Threads

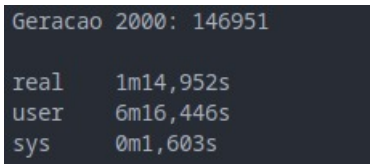
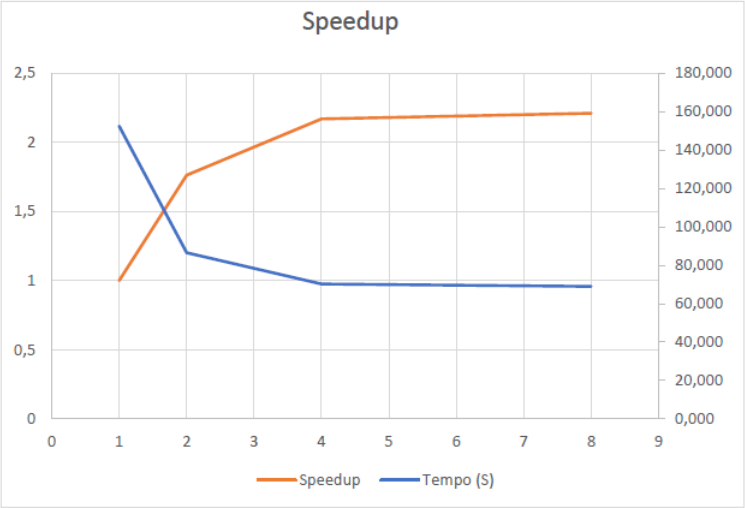


Figura 26 – Tabela Pthreads

N threads	Tempo (S)	Speedup	Eficiência
1	152,416	1	100,000%
2	86,471	1,763	88,131%
4	70,269	2,169	54,226%
8	68,971	2,210	27,623%

Figura 27 – Gráfico Pthreads



4.3 Código utilizando *OpenMP*.

Figura 28 – Tempo de Processamento OpenMp com 2 Threads

```
Geracao 2000: 146951

real    1m51,957s
user    3m43,692s
sys     0m0,072s
```

Figura 29 – Tempo de Processamento OpenMp com 4 Threads

```
Geracao 2000: 146951

real    1m18,342s
user    5m12,558s
sys     0m0,068s
```

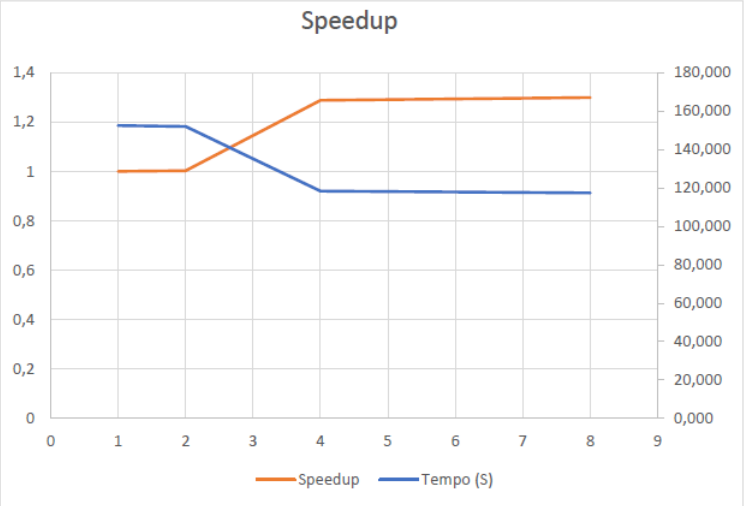
Figura 30 – Tempo de Processamento OpenMp com 8 Threads

```
Geracao 2000: 146951

real    1m17,360s
user    9m55,337s
sys     0m0,499s
```

Figura 31 – Tabela OpenMP

N threads	Tempo (S)	Speedup	Eficiência
1	152,416	1	100,000%
2	151,957	1,003	50,151%
4	118,342	1,288	32,198%
8	117,360	1,299	16,234%

Figura 32 – Gráfico OpenMP



4.4 Código utilizando *Java Threads*.

Figura 33 – Java Serial

```
Geracao 2000: 149488

real    1m41,626s
user    1m47,133s
sys     0m0,440s
```

Figura 34 – Java 2 Threads

```
Geracao 2000: 149488

real    1m17,570s
user    2m23,547s
sys     0m1,206s
```

Figura 35 – Java 4 Threads

```
Geracao 2000: 149488

real    1m7,108s
user    3m27,349s
sys     0m1,931s
```

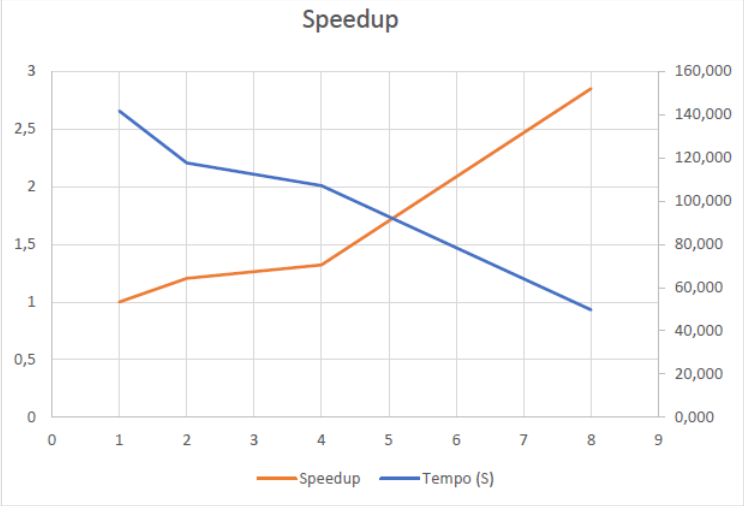
Figura 36 – Java 8 Threads

```
Geracao 2000: 149488

real    0m49,717s
user    3m45,216s
sys     0m3,926s
```

Figura 37 – Tabela Java

N threads	Tempo (S)	Speedup	Eficiência
1	141,626	1	100,000%
2	117,570	1,205	60,231%
4	107,108	1,322	33,057%
8	49,717	2,849	35,608%

Figura 38 – Gráfico Java



5 Considerações Finais

A implementação dessa atividade ajudou a concretizar o conhecimento sobre programação concorrente distribuída, o funcionamento de threads, limitações de hardware, problemas lógicos no contole das threads e até com relação a perda de eficiência do programa, quando neste há o uso de muitas threads, pois o programa acaba tendo que manejar uma quantidade muito grande de threads, o que faz a eficiência cair. Por fim, os códigos foram implementados em reuniões a distancia pelo grupo em questão, o que proporcionou a troca de conhecimentos e experiências e contribuiu para o entendimento da matéria.

Referências

MPI Reference (online): http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

OpenMP (online resources): https://wwVidiaw.openmp.org/

NVidia resources: https://developer.nvidia.com/cuda-education

[1] – Martin Gardner, "Mathematical Games – The fantastic combinations of John Conway's new solitaire game life", Scientific American 223, Oct. 1970, pp 120-123.

http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis$_p rojekt/proj_g amelife/ConwayScien$