



Universidade do Estado do Rio de Janeiro
Campus Maracanã

Faculdade de Engenharia Departamento de
Engenharia de Sistemas e Computação

Filipe Rangel da Costa

Análise de desempenho do algoritmo paralelo "*The Game of Life*" implementado em linguagem de programação C utilizando OpenMP e MPI

Rio de Janeiro

2019

Filipe Rangel da Costa

Análise de desempenho do algoritmo paralelo "*The Game of Life*"
implementado em linguagem de programação C utilizando OpenMP e MPI

Monografia apresentada, como requisito parcial para obtenção do título de Bacharel, ao Faculdade de Engenharia Departamento de Engenharia de Sistemas e Computação, da Universidade do Estado do Rio de Janeiro.



Orientador: Profa. Dra. Cristiana Barbosa Bentes
Coorientador: BSc Alexandre Ribeiro Fernandes Azevedo

Rio de Janeiro
2019

Ficha elaborada pelo autor através do
Sistema para Geração Automática de Ficha Catalográfica da Rede Sirius - UERJ

C837 Costa, Filipe Rangel da
Análise de desempenho do algoritmo paralelo "The
Game of Life" implementado em linguagem de
programação C utilizando OpenMP e MPI / Filipe Rangel
da Costa. - 2019.
57 f.

Orientador: Cristiana Barbosa Bentes
Trabalho de Conclusão de Curso apresentado à
Universidade do Estado do Rio de Janeiro, Faculdade
de Engenharia, para obtenção do grau de bacharel em
Engenharia Elétrica.

1. Paralelização - Monografias. 2. OpenMP -
Monografias. 3. MPI - Monografias. 4. Análise de
desempenho - Monografias. I. Bentes, Cristiana
Barbosa. II. Universidade do Estado do Rio de
Janeiro. Faculdade de Engenharia. III. Título.

CDU 621.3

Filipe Rangel da Costa

**Análise de desempenho do algoritmo paralelo "The Game of Life"
implementado em linguagem de programação C utilizando OpenMP e MPI**

Monografia apresentada, como requisito parcial para obtenção do título de Bacharel, ao Faculdade de Engenharia Departamento de Engenharia de Sistemas e Computação, da Universidade do Estado do Rio de Janeiro.

Aprovada em 18 de Dezembro de 2019.

Banca Examinadora:

Otiane B. Bentes

Profa. Dra. Cristiana Barbosa Bentes (Orientador)
Faculdade de Engenharia Departamento de Engenharia de Sistemas
e Computação – UERJ

Alexandre N. Fernandes Azevedo
BSc Alexandre Azevedo (Coorientador)
Instituto de Matemática e Estatística – UERJ

Maria Clícia Stelling de Castro
Profa Dra Maria Clícia Stelling de Castro
Instituto de Matemática e Estatística – UERJ

Sheila - Rigua Murgel Veloso
Profa. Dra. Sheila Murgel Veloso
Faculdade de Engenharia Departamento de Engenharia de Sistemas
e Computação – UERJ

Rio de Janeiro

2019

DEDICATÓRIA

A minha família.

AGRADECIMENTOS

Aos meus familiares e namorada, por todo o suporte disponibilizado durante todo o período na faculdade.

Aos professores por acreditarem na educação.

Aos orientadores pelo tempo gasto no projeto.

A Deus por dar saúde e forças em todos os momentos difíceis.

Há três métodos para ganhar sabedoria: primeiro, por reflexão, que é o mais nobre; segundo, por imitação, que é o mais fácil; e terceiro, por experiência, que é o mais amargo.

Confúcio

RESUMO

RANGEL DA COSTA, F. R. C. *Análise de desempenho do algoritmo paralelo "The Game of Life" implementado em linguagem de programação C utilizando OpenMP e MPI.* 2019. 57 f. Monografia (Bacharelado em Engenharia Elétrica - Ênfase Sistemas e Computação) – Faculdade de Engenharia Departamento de Engenharia de Sistemas e Computação, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2019.

A paralelização é um dos meios de se utilizar a computação de alto desempenho, pois com ela poderemos aproveitar toda a gama de supercomputadores e *clusters*. Um código que roda serialmente no computador, perde as vantagens da programação em paralelo, que utiliza dois ou mais processadores para executá-lo. Com o aproveitamento dos diversos processadores que temos em *clusters*, nós podemos obter um ganho de tempo a cada vez que um código é executado. Com isso, o código "*The Game of Life*", escrito na linguagem de programação C, foi paralelizado para se obter um ganho de tempo em cada execução. Foram utilizados OpenMP e MPI para implementar paralelização e foram obtidos resultados satisfatórios, com o ganho de tempo desejado.

Palavras-chave: Paralelização. OpenMP. MPI. Análise de desempenho.

ABSTRACT

RANGEL DA COSTA, F. R. C. *Performance analysis of "The Game of Life" parallel algorithm implemented in C programming language using OpenMP and MPI.* 2019. 57 f. Monografia (Bacharelado em Engenharia Elétrica - Ênfase Sistemas e Computação) – Faculdade de Engenharia Departamento de Engenharia de Sistemas e Computação, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2019.

The parallelization is one way of using the high performance computing because we can take advantage of supercomputers and clusters. One code that runs serially loses the advantages of parallel programming, that uses two or more processors to run it. By taking advantage of the many processors we have in clusters, we can save time each time code is executed. Thus the code "The Game of Life", written in the C programming language, was parallelized to obtain a time saving in each execution. OpenMP and MPI were used to implement the parallelization, . Satisfactory results were obtained with the desired time gain.

Keywords: Parallelize. OpenMP. MPI. Performance analysis.

LISTA DE FIGURAS

Figura 1	- Entrada no arquivo life.in	14
Figura 2	- Programação serial e paralela	16
Figura 3	- Modelos de memória.	17
Figura 4	- Modelo Fork-Join.	18
Figura 5	- Exemplo de modelo de memória OpenMP	19
Figura 6	- Estrutura MPI_Send.	23
Figura 7	- Estrutura MPI_Recv.	23
Figura 8	- Operações de comunicação coletiva	24
Figura 9	- Exemplo de MPI_Bcast.	25
Figura 10	- Exemplo de MPI_Reduce.	26
Figura 11	- Exemplo de MPI_Allreduce.	26
Figura 12	- Exemplo de MPI_Scatter.	27
Figura 13	- Exemplo de MPI_Gather.	28
Figura 14	- Exemplo de MPI_Allgather.	28
Figura 15	- Máquina c5.12xlarge na nuvem AWS.	47
Figura 16	- Gráfico com os tempos de execução (em segundos) para diferentes números de <i>threads</i>	48
Figura 17	- Gráfico com o valor de <i>speedup</i> para diferentes números de <i>threads</i> . . .	50
Figura 18	- Gráfico da eficiência para determinado número de <i>threads</i>	51
Figura 19	- Gráfico com os tempos de execução (em segundos) para diferentes números de processos.	52
Figura 20	- Gráfico com o valor de speedup para diferentes números de processos. .	53
Figura 21	- Gráfico da eficiência para determinado número de <i>threads</i>	54

LISTA DE TABELAS

Tabela 1	- Média dos tempos e desvio padrão da análise de desempenho OpenMP.	48
Tabela 2	- <i>Speedup</i> do OpenMP, calculado a partir da média dos tempos de execução.	49
Tabela 3	- Tabela com a eficiência do OpenMP, calculado a partir da média dos tempos de execução.	50
Tabela 4	- Tabela com a média dos tempos e desvio padrão da análise de desempenho MPI.	51
Tabela 5	- <i>Speedup</i> do MPI, calculado a partir da média dos tempos de execução.	52
Tabela 6	- Tabela com a eficiência do MPI, calculado a partir da média dos tempos.	53

LISTA DE ABREVIATURAS E SIGLAS

UERJ	Universidade do Estado do Rio de Janeiro
DESC	Departamento de Engenharia de Sistemas e Computação
API	Application Programming Interface
MP	Multi-Processing
MPI	Message Passing Interface
AWS	Amazon Web Services

SUMÁRIO

1	INTRODUÇÃO	13
2	CONCEITOS BÁSICOS	15
2.1	Introdução a OpenMP	15
2.1.1	<u>Modelo de Execução</u>	18
2.1.2	<u>Modelo de Memória</u>	18
2.1.3	<u>Diretivas do OpenMP</u>	18
2.1.3.1	Regiões Paralelas	18
2.1.3.2	Diretivas <i>for/do</i>	19
2.1.3.3	Diretivas <i>Sections</i>	20
2.1.3.4	Construções de sincronizações	20
2.2	Introdução ao MPI	21
2.2.1	<u>Modelo de memória</u>	22
2.2.2	<u>Communicator</u>	22
2.2.3	<u>Comunicação ponto a ponto</u>	23
2.2.3.1	MPI_Send	23
2.2.3.2	MPI_Recv	23
2.2.4	<u>Comunicação coletiva</u>	24
2.2.4.1	MPI_Barrier	24
2.2.4.2	MPI_Bcast	24
2.2.4.3	MPI_Reduce	25
2.2.4.4	MPI_Allreduce	25
2.2.4.5	MPI_Scatter	25
2.2.4.6	MPI_Gather	25
2.2.4.7	MPI_Allgather	27
3	IMPLEMENTAÇÕES PARALELAS	29
3.1	Implementação Serial	29
3.1.1	<u>Código Serial</u>	29
3.2	Implementação OpenMP	32
3.2.1	<u>Código OpenMP</u>	32
3.3	Implementação MPI	35
3.3.1	<u>Código MPI</u>	36
3.4	Tutorial de execução	45
4	ANÁLISE DE DESEMPENHO	46
4.1	Execução Serial	47
4.2	Análise OpenMP	47
4.3	Análise MPI	49

5	CONCLUSÕES	55
	REFERÊNCIAS	56
	APÊNDICE A – O código utilizado neste trabalho está disponível no github: https://github.com/frangeldc/TCC	57

1 INTRODUÇÃO

The Game of Life é um autômato celular desenvolvido pelo matemático britânico John Horton Conway em 1970. O jogo foi criado de modo a reproduzir, através de regras simples, as alterações e mudanças em grupos de seres vivos, tendo aplicações em diversas áreas da ciência. As regras definidas são aplicadas a cada nova “geração”; assim, a partir de uma imagem em um tabuleiro bi-dimensional definida pelo jogador, percebem-se mudanças muitas vezes inesperadas e belas a cada nova geração, variando de padrões fixos a caóticos. Desde sua publicação, ele tem atraído muito interesse devido aos caminhos surpreendentes que pode tomar. *Life* é um exemplo de auto-organização (WIKIPEDIA, 2005).

Conway escolheu suas regras cuidadosamente, após um longo período de experimentos, para satisfazer três critérios:

- Não deve haver nenhuma imagem inicial para a qual haja uma prova imediata ou trivial de que a população pode crescer sem limite;
- Deve haver imagens iniciais que aparentemente cresçam sem limite;
- Deve haver imagens iniciais simples que cresçam e mudem por um período de tempo considerável antes de chegar a um fim das possíveis formas:
 - Sumindo completamente (por superpopulação ou por ficarem muito distantes);
 - Estacionando em uma configuração estável que se mantém imutável para sempre, ou entrando em uma fase de oscilação na qual são repetidos ciclos infinitos de dois ou mais períodos.

As regras do jogo da vida são simples:

- Qualquer célula viva com menos de dois vizinhos vivos morre de solidão.
- Qualquer célula viva com mais de três vizinhos vivos morre de superpopulação.
- Qualquer célula morta com exatamente três vizinhos vivos se torna uma célula viva.
- Qualquer célula viva com dois ou três vizinhos vivos continua no mesmo estado para a próxima geração.

O jogo tem sua evolução determinada pelo seu estado inicial, e foi disponibilizado para a 11^a maratona de programação paralela ocorrida no ano de 2016 (MARATONA..., 2016).

Um exemplo de uma entrada válida para o jogo, existente no arquivo "life.in", pode ser visto na Figura 1. Na entrada temos os valores do tamanho da matriz(*size*), do número

life.in		
1	11	7
2		
3		
4		x
5	x	x
6		x
7	xxx	xxx
8		x
9	x	x
10		x
11		
12		
13		

Figura 1 - Entrada no arquivo life.in

de passos que o jogo vai dar(*steps*) e as células da matriz definidas por espaços em branco e letras x.

Neste trabalho, o objetivo é realizar a implementação da aplicação *The Game of Life* em paralelo utilizando os paradigmas de memória compartilhada e troca de mensagens, e a partir dos resultados obtidos, fazer a análise de desempenho e demonstrar se há ganho de velocidade com a paralelização do programa. A implementação foi realizada utilizando-se OpenMP e MPI, e para o caso do MPI é utilizada a distribuição OpenMPI.

Foram realizados experimentos na nuvem da Amazon (AWS) com até 24 processadores e os resultados mostram os ganhos de tempo da paralelização em cada um dos modelos, comprovando que quanto mais processadores, mais rápido o jogo da vida paralelo pode rodar. Dependendo da aplicação, tem-se um limite em que a partir de um número de processadores o ganho torna-se irrisório.

2 CONCEITOS BÁSICOS

Os *hardwares* paralelos estão presentes em quase todos os computadores de mesa, *notebooks*, servidores, celulares. Para aproveitar ao máximo esses *hardwares* tem-se a programação em paralelo. Um programa serial utiliza apenas um processador, mesmo que o *hardware* tenha vários, subutilizando os processadores. Tradicionalmente, os *softwares* são escritos para executar serialmente, sendo as instruções executadas em apenas um processador. Um exemplo de problema serial se encontra na Figura 2a, onde o problema é dividido em várias instruções que rodam uma por vez em um processador.

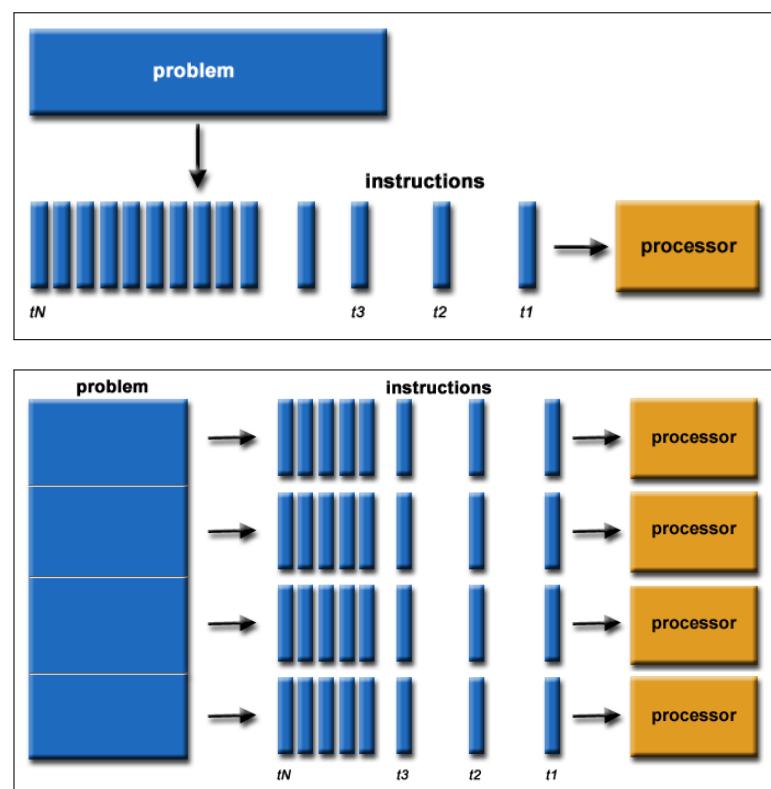
No sentido mais simples, a programação em paralelo utiliza os múltiplos recursos de um computador para resolver os problemas computacionais, sendo assim, as instruções são divididas e executadas em diversos processadores paralelamente. Na figura 2b temos o exemplo de um problema sendo resolvido paralelamente. Primeiro ocorre a divisão das instruções do problema em partes, em seguida as instruções são executadas paralelamente, cada processador com seu conjunto de instruções.

A programação paralela possui dois modelos de memória que definem certos aspectos da sua aplicabilidade. O modelo de memória compartilhada, que na prática é a disponibilidade da mesma memória para todos os processadores, ou seja, a não ser que seja explicitamente definido de forma diferente, toda a memória é sempre acessível por cada um dos processadores. O segundo modelo de memória é o modelo de memória distribuída em que cada processador tem sua própria memória privativa e o compartilhamento de dados deve ser feito pela troca de mensagens. Tem-se na Figura 3 os modelos de memória compartilhada e distribuída.

2.1 Introdução a OpenMP

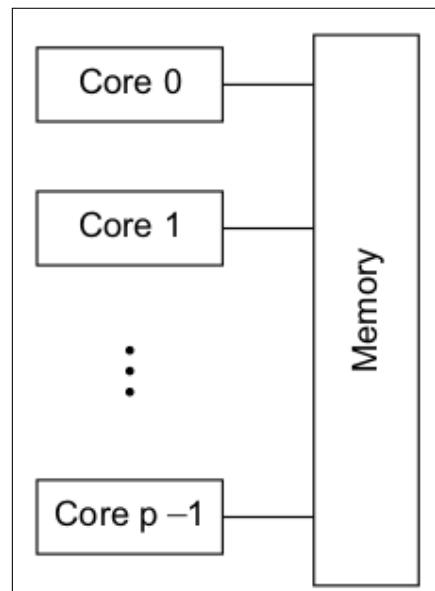
O OpenMP é uma *Application Programming Interface* (API) que suporta o modelo de programação de memória paralela de memória compartilhada multiplataforma em C/C++ e Fortran. Podemos ver na Figura 3a o modelo de memória compartilhada. A API do OpenMP define um modelo portátil e escalável, com uma interface simples e flexível para o desenvolvimento de aplicativos paralelos em plataformas, de computadores comuns a supercomputador (JUNIOR, 2004).

Figura 2 - Programação serial e paralela

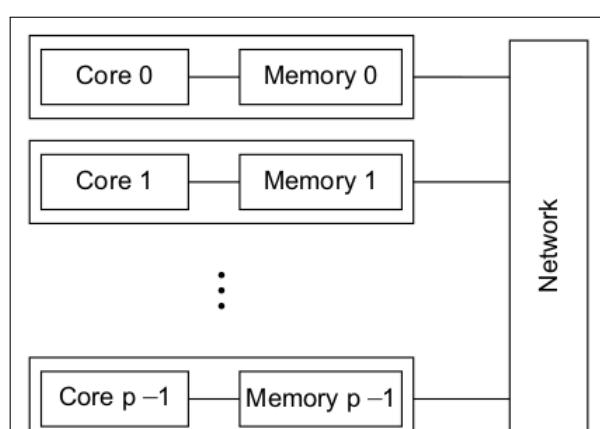


Legenda: Problema em computação serial e paralela: (a) serial;
(b) paralela;

Figura 3 - Modelos de memória.



(a)



(b)

Legenda: Modelos de memória compartilhada e memória distribuída: (a) Modelo de memória compartilhada; (b) modelo de memória distribuída;

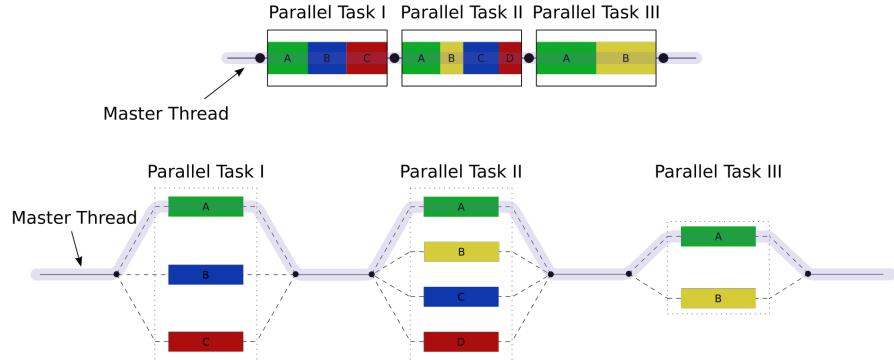


Figura 4 - Modelo Fork-Join.

2.1.1 Modelo de Execução

O *OpenMP* trabalha internamente com um modelo *FORK-Join* onde, a partir de uma *thread master* (*thread* que iniciou o programa), cria-se um time de *threads* para a realização de tarefas em paralelo. Depois é realizado um *join* onde todas as *threads* são sincronizadas e somente a *thread master* não é colocada em modo de espera.

2.1.2 Modelo de Memória

O modelo de memória utilizado em OpenMP é o modelo de memória compartilhada. Na Figura 4 tem-se o modelo *Fork-Join* utilizado no OpenMP, onde a escalabilidade é limitada pela arquitetura de memória, pois a arquitetura vai influenciar no número de threads disponíveis.

2.1.3 Diretivas do OpenMP

2.1.3.1 Regiões Paralelas

O *OpenMP* utiliza diretivas do compilador para definir as regiões paralelas, criando as *threads* do programa a partir dessas diretivas.

As diretivas são definidas pela estrutura "#pragma omp parallel [cláusula] ...". O bloco de código dentro de uma região paralela é executado por diferentes *threads*.

Cláusulas:

- num_threads() - define o número de *threads* que rodarão em paralelo;
- if() - cria uma condição para o paralelismo;

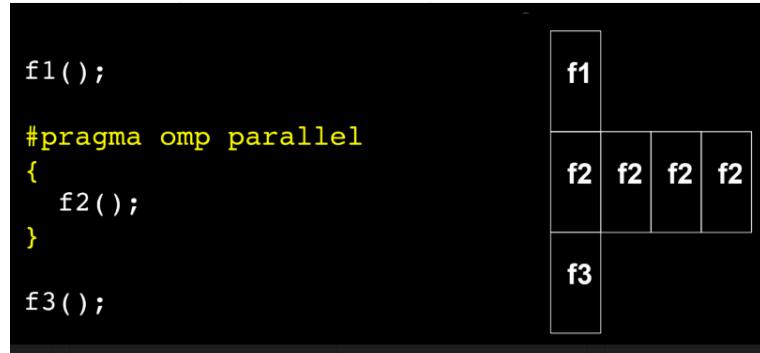


Figura 5 - Exemplo de modelo de memória OpenMP

- private(list) - variáveis privadas as *threads*;
- firstprivate(list) - são iniciadas fora da *thread* e não são compartilhadas;
- shared(list) - variáveis compartilhadas por todas as *threads*;
default(shared | none);
- reduction - fórmula para combinar as variáveis de todas as *threads* nas variáveis da *thread master*;
- omp_get_thread_num() – Função básica do OpenMP que retorna o identificador da *thread* corrente. As “N” *threads* a executar numa região paralela são numerados de 0 a “N-1” e a *thread master* é sempre identificado pelo número 0;
- omp_get_num_threads() - Função básica do OpenMP que retorna o número de *threads* momentaneamente ativas. Se for chamada a partir de uma região sequencial (executada apenas pela *thread master*) retorna 1.

2.1.3.2 Diretivas *for/do*

A diretiva *for* permite com que uma instrução do tipo laço *for* ou *do* sejam realizadas em paralelo por *threads*, definindo uma relação de variáveis privadas e compartilhadas pelas *threads*.

Cláusulas:

- private(list) - variáveis privadas as *threads*;
- firstprivate(list) - são iniciadas fora da *thread* e não são compartilhadas;
- lastprivate(list) - atualiza o valor da variável ao final da região;

- reduction - fórmula para combinar as variáveis de todas as *threads* nas variáveis da *thread master*;
- nowait() - Evita barreira implícita ao final de uma construção *worksharing*;
- schedule() - Determina como as iterações são executadas em paralelo;
- ordered() - Força que eventos aconteçam em determinada ordem;

2.1.3.3 Diretivas *Sections*

A diretiva *section* permite definir seções de código distintas que serão executadas em paralelas (paralelismo heterogêneo), com exemplo do código:

```
#pragma omp sections [clause ...] newline
    shared (list)
    private (list)
    firstprivate (list)
    lastprivate (list)
    reduction (operator: list)
    nowait
{

#pragma omp section newline
    structured_block1
#pragma omp section newline
    structured_block2
}
```

A diretiva *sections* indica o início e dentro dela pode ter diversas seções *section*. Cada seção marca uma área de código diferente. A diretiva *sections* distribui as tarefas entre as *threads* existentes e o único requisito é que as seções sejam independentes. Depois, cada *thread* executa uma tarefa por vez. Cada seção é executada apenas por uma *thread*.

2.1.3.4 Construções de sincronizações

As construções de sincronização são utilizadas para evitar situações de corrida. As condições de corrida ocorrem quando a saída do programa muda quando as *threads* são escalonadas de forma diferente.

- Barrier - A diretriz *Barrier* sincroniza todas as *threads* correntes esperando a finalização de todas para retornar somente a execução da *thread master*.
- Critical - Define uma região critica que poderá ser executada somente por uma *thread* por vez, bloqueando o acesso de outras *threads*.
- Master - A diretiva *master* define uma região que será executada somente pela *thread master*, sendo tal região ignorada pelas outras *threads*.
- Atomic - Define que a atribuição de uma variável será realizada somente por uma *thread* de cada vez impedindo o acesso simultâneo de outras *threads* ao mesmo tempo.
- Ordered - A diretiva *ordered* define que as interações em um determinado *loop* serão executados de modo sequencial (JUNIOR, 2004).

2.2 Introdução ao MPI

O *Message Passage Interface*(MPI) é um padrão para comunicação de dados em computação paralela. Existem vários modelos de programação paralela, como memória compartilhada, *threads*, passagem de mensagens, paralelismo de dados, paralelismo de objetos, híbrido, e dependendo do problema que se está tentando resolver, pode ser necessário passar informações entre os vários processadores ou nós de um *cluster*, e o MPI oferece uma infraestrutura para essa tarefa.

No padrão MPI, uma aplicação é constituída por uma ou mais tarefas (as quais podem ser processos, ou *threads*, dependendo da implementação) que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém, esses processos podem executar diferentes programas.

Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga.

A implementação da biblioteca MPI utilizada foi a OpenMPI. O *Open MPI Project* é uma implementação *open source* de troca de mensagens que é desenvolvida e mantida por um consórcio de acadêmicos, pesquisadores e parceiros do setor. O *Open MPI* é, portanto, capaz de combinar o conhecimento, as tecnologias e os recursos de toda a comunidade de computação de alto desempenho. O *Open MPI* oferece vantagens, como alta performance em diversas plataformas, API's documentadas e suporte a falhas de rede, para fornecedores de sistemas e *software*, desenvolvedores de aplicativos e pesquisadores de ciência da computação (OPEN-MPI-ORG, 2004).

A biblioteca MPI contém 125 funções, sendo que 6 funções são consideradas principais, que são elas:

- MPI_INIT – inicializa os processos do MPI;
- MPI_Finalize – Finaliza a computação;
- MPI_Comm_Size – Retorna o número de processos;
- MPI_Comm_Rank – Retorna o *id* dos processos;
- MPI_Send – Envia uma mensagem;
- MPI_Recv – Recebe uma mensagem.

Inicialização na linguagem C: MPI_Init (&argc, &argv); Finalização na linguagem C: MPI_Finalize();

Temos um esqueleto de um programa MPI na linguagem C:

```
#include <mpi.h>
main( int argc , char** argv )
{
    MPI_Init( &argc , &argv );
    /* código do programa */
    MPI_Finalize();
}
```

2.2.1 Modelo de memória

O modelo de memória utilizado no MPI é o modelo distribuído. Na Figura 3b temos um exemplo de modelo de memória distribuída.

2.2.2 Communicator

Communicator é o universo de processos envolvido numa comunicação. MPI_COMM_WORLD é o *communicator* padrão utilizado em MPI, que inclui todos os processos. MPI permite criação de novos *communicators* definidos por grupo (conjunto ordenado de processos) e contexto (*tag* de uma mensagem), sendo que, processos do mesmo grupo e usando o mesmo contexto podem se comunicar.

```

int MPI_Send(
    void*      msg_buf_p    /* in */,
    int        msg_size     /* in */,
    MPI_Datatype msg_type   /* in */,
    int        dest         /* in */,
    int        tag          /* in */,
    MPI_Comm   communicator /* in */);

```

Figura 6 - Estrutura MPI_Send.

```

int MPI_Recv(
    void*      msg_buf_p    /* out */,
    int        buf_size     /* in */,
    MPI_Datatype buf_type   /* in */,
    int        source       /* in */,
    int        tag          /* in */,
    MPI_Comm   communicator /* in */,
    MPI_Status* status_p    /* out */);

```

Figura 7 - Estrutura MPI_Recv.

2.2.3 Comunicação ponto a ponto

2.2.3.1 MPI_Send

É a função de envio de mensagens, com ela é possível que um determinado processo envie pacotes de mensagens explicitamente a outro processo.

Tem-se na Figura 6 a sintaxe do MPI_Send. Os três primeiros argumentos, *msg_buf_p*, *msg_size*, e *msg_type*, determinam o conteúdo da mensagem. Os outros três argumentos, *dest*, *tag*, *communicator* determinam o destino da mensagem.

2.2.3.2 MPI_Recv

É a função de recebimento de mensagens, é necessária para que os processos saibam que uma mensagem vai ser enviada por outro processo.

Tem-se na Figura 7 a sintaxe do MPI_Recv. Os seis primeiros argumentos correspondem aos seis primeiros argumentos do MPI_Send. Os três primeiros argumentos especificam a memória disponível para receber a mensagem e os outros três identificam a mensagem. O tipo MPI_Status é uma estrutura com os três membros MPI_SOURCE, MPI_TAG e MPI_ERROR.

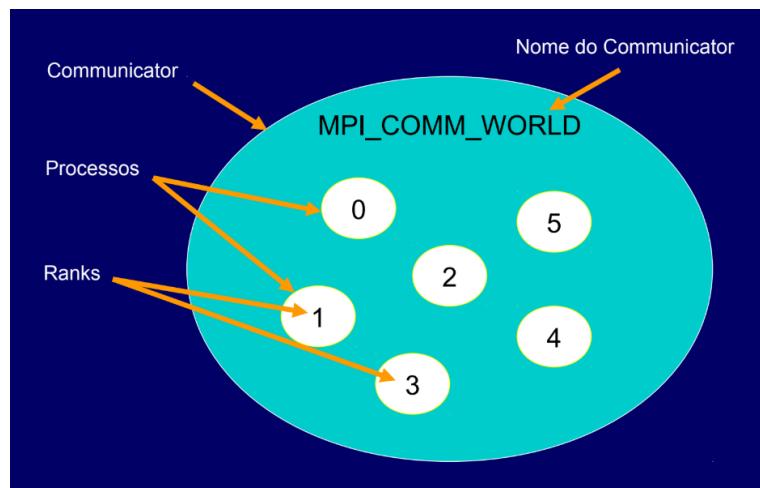


Figura 8 - Operações de comunicação coletiva

Fonte: (QUINN, 2015)

2.2.4 Comunicação coletiva

Tipos de comunicação coletiva:

- Sincronização – os processos esperam todos os membros do grupo chegarem ao ponto de sincronização.
- Movimentação de dados – broadcast, scather/gather;
- Computação coletiva – um membro do grupo coleta dados dos outros membros e faz uma operação com os dados.

2.2.4.1 MPI_Barrier

É uma operação de sincronização. Cria uma barreira que impede que os processos avancem até que todos os processos estejam naquele mesmo local da chamada da barreira.

2.2.4.2 MPI_Bcast

É uma operação de movimentação de dados. O processo principal envia a mesma mensagem para todos os outros processos do grupo. Podemos ver na Figura 9 um exemplo de broadcast.

MPI_Bcast

Broadcasts a message from one task to all other tasks in communicator

```
count = 1;           task1 contains the message to be broadcast
source = 1;
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

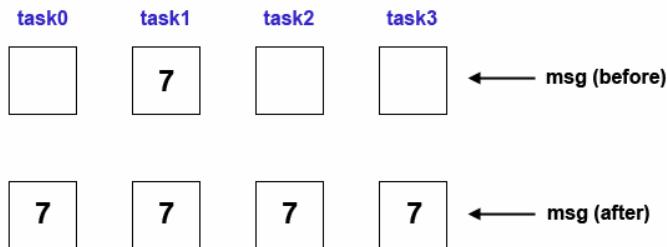


Figura 9 - Exemplo de MPI_Bcast.

Fonte: (BARNEY, 2019)

2.2.4.3 MPI_Reduce

É uma operação de computação coletiva. A operação de redução aplica uma operação de redução em todos os dados recebidos de cada um dos processos e coloca o resultado em apenas um processo. Podemos ver na Figura 10 um exemplo de *reduce*.

2.2.4.4 MPI_Allreduce

É uma operação de computação coletiva e movimentação de dados. É como um MPI_Reduce seguido de um MPI_Bcast. Podemos ver na Figura 11 um exemplo de *allreduce*.

2.2.4.5 MPI_Scatter

É uma operação de movimentação de dados. Distribui mensagens distintas entre os processos do grupo. Podemos ver na Figura 12 um exemplo de *scatter*.

2.2.4.6 MPI_Gather

É uma operação de movimentação de dados. Junta mensagens distintas de cada processador do grupo em um processador. Podemos ver na Figura 13 um exemplo de

MPI_Reduce

Perform reduction across all tasks in communicator and store result in 1 task

```
count = 1;
dest = 1;                                task1 will contain result
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT,
           MPI_SUM, dest, MPI_COMM_WORLD);
```

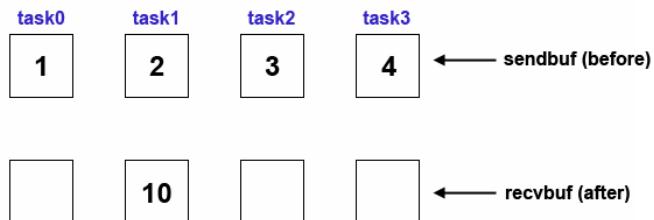


Figura 10 - Exemplo de MPI_Reduce.

Fonte: (BARNEY, 2019)

MPI_Allreduce

Perform reduction and store result across all tasks in communicator

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);
```

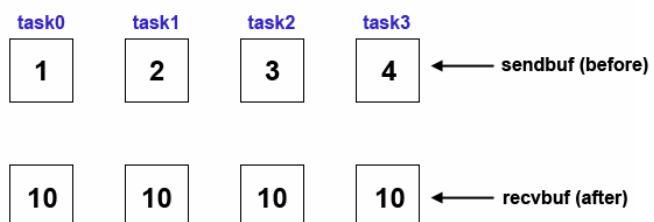


Figura 11 - Exemplo de MPI_Allreduce.

Fonte: (BARNEY, 2019)

MPI_Scatter

Sends data from one task to all other tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
src = 1;                                task1 contains the data to be scattered
MPI_Scatter(sendbuf, sendcnt, MPI_INT
            recvbuf, recvcnt, MPI_INT
            src, MPI_COMM_WORLD);
```

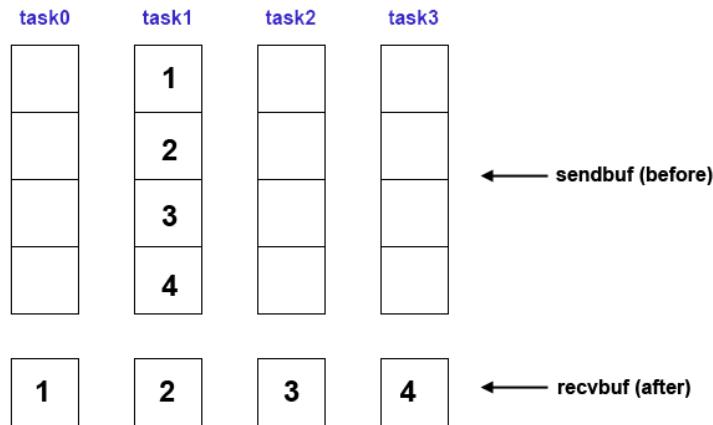


Figura 12 - Exemplo de MPI_Scatter.

Fonte: (BARNEY, 2019)

gather.

2.2.4.7 MPI_Allgather

É uma operação de movimentação de dados. Podemos ver na Figura 14 um exemplo de *allgather*.

MPI_Gather

Gathers data from all tasks in communicator to a single task

```
sendcnt = 1;
recvcnt = 1;
src = 1;                                message will be gathered into task1
MPI_Gather(sendbuf, sendcnt, MPI_INT
           recvbuf, recvcnt, MPI_INT
           src, MPI_COMM_WORLD);
```

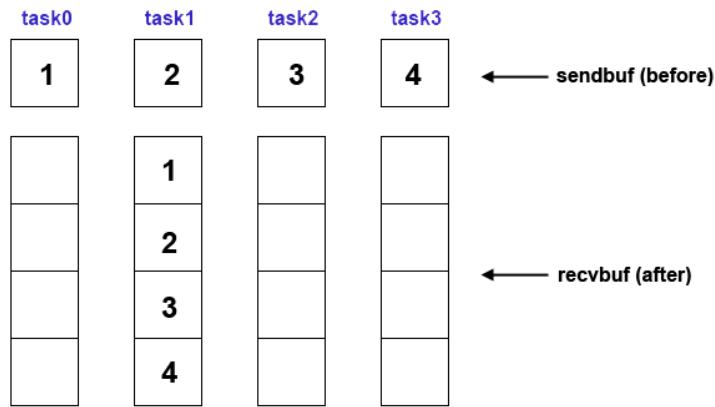


Figura 13 - Exemplo de MPI_Gather.

Fonte: (BARNEY, 2019)

MPI_Allgather

Gathers data from all tasks and then distributes to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT
              recvbuf, recvcnt, MPI_INT
              MPI_COMM_WORLD);
```

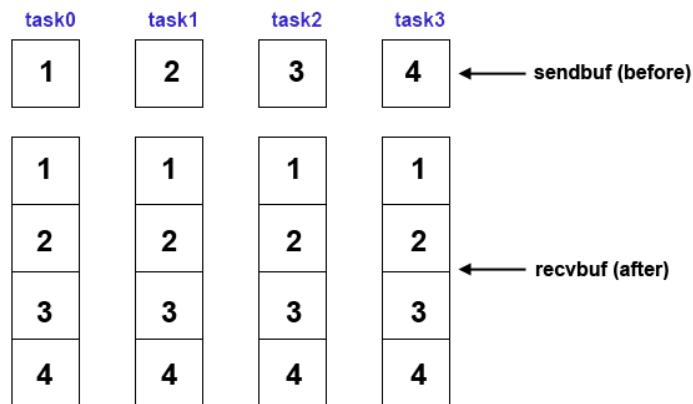


Figura 14 - Exemplo de MPI_Allgather.

Fonte: (BARNEY, 2019)

3 IMPLEMENTAÇÕES PARALELAS

3.1 Implementação Serial

Na implementação serial, foram iniciados os primeiros testes para se obter o local adequado para a paralelização do código. Foi feita a métrica dos tempos de execução de cada laço *for* existente no código, e a partir dessa métrica, foi detectada a área do código que mais demorava a executar. O laço mais demorado é o laço *for* principal, linha 102, que tem dentro dele a chamada da função *play*, linha 39, e as impressões em tela, linhas 106 e 107. Já o segundo laço *for* mais demorado, encontra-se dentro da função *play*, linha 42, que é chamada pelo laço principal, e este sim pode ser paralelizado. Foi utilizada a função *clock_t* da biblioteca *clock()* para se obter os tempos de execução, tendo início na linha 101 e fim na linha 113.

3.1.1 Código Serial

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef unsigned char cell_t;
4
5
6 #define DEBUG = 1
7
8 cell_t ** allocate_board (int size) {
9     cell_t ** board = (cell_t **) malloc(sizeof(cell_t *)*size);
10    int i;
11    for (i=0; i<size; i++)
12        board[i] = (cell_t *) malloc(sizeof(cell_t)*size);
13    return board;
14 }
15
16 void free_board (cell_t ** board, int size) {
17     int i;
18     for (i=0; i<size; i++)
19         free(board[i]);
20     free(board);
21 }
22
23
24 /* return the number of on cells adjacent to the i,j cell */

```

```

25 int adjacent_to (cell_t ** board, int size, int i, int j) {
26     int k, l, count=0;
27     int sk = (i>0) ? i-1 : i;
28     int ek = (i+1 < size) ? i+1 : i;
29     int sl = (j>0) ? j-1 : j;
30     int el = (j+1 < size) ? j+1 : j;
31
32     for (k=sk; k<=ek; k++)
33         for (l=sl; l<=el; l++)
34             count+=board[k][l];
35     count-=board[i][j];
36     return count;
37 }
38
39 void play (cell_t ** board, cell_t ** newboard, int size) {
40     int i, j, a;
41     /* for each cell, apply the rules of Life */
42     for (i=0; i<size; i++)
43         for (j=0; j<size; j++) {
44             a = adjacent_to (board, size, i, j);
45             if (a == 2) newboard[i][j] = board[i][j];
46             if (a == 3) newboard[i][j] = 1;
47             if (a < 2) newboard[i][j] = 0;
48             if (a > 3) newboard[i][j] = 0;
49         }
50 }
51
52 /* print the life board */
53 void print (cell_t ** board, int size) {
54     int i, j;
55     /* for each row */
56     for (j=0; j<size; j++) {
57         /* print each column position... */
58         for (i=0; i<size; i++)
59             printf ("%c", board[i][j] ? 'x' : ' ');
60         /* followed by a carriage return */
61         printf ("\n");
62     }
63 }
64
65 /* read a file into the life board */
66 void read_file (FILE * f, cell_t ** board, int size) {
67     int i, j;
68     char *s = (char *) malloc(size+10);
69     char c;
70     for (j=0; j<size; j++) {
71         /* get a string */

```

```

72     fgets (s, size+10,f);
73     /* copy the string to the life board */
74     for (i=0; i<size; i++)
75     {
76         //c=fgetc(f);
77         //putchar(c);
78         board [i][j] = s[i] == 'x';
79     }
80     //fscanf(f, "\n");
81 }
82 }
83
84 int main () {
85     int size , steps;
86     FILE *f;
87     f = fopen("judge.in" , "r");
88 //    f = stdin;
89     fscanf(f, "%d %d" , &size , &steps);
90     cell_t ** prev = allocate_board (size);
91     read_file (f , prev , size);
92     fclose(f);
93     cell_t ** next = allocate_board (size);
94     cell_t ** tmp;
95     int i , j;
96 #ifdef DEBUG
97     printf("Initial \n");
98     print (prev , size);
99     printf("-----\n");
100 #endif
101    clock_t begin = clock();
102    /* Laco for principal */
103    for (i=0; i<steps; i++) {
104        play (prev , next , size);
105        #ifdef DEBUG
106            printf("%d ----- \n" , i);
107            print (next , size);
108        #endif
109        tmp = next;
110        next = prev;
111        prev = tmp;
112    }
113    clock_t end = clock();
114    double spent = (double)(end - begin) / CLOCKS_PER_SEC;
115    printf("Tempo de execucao do laco
116    for principal:%.3f\n" , spent);
117    print (prev , size);
118    free_board (prev , size);

```

```

119     free_board(next, size);
120 }
```

Listing 3.1 - life.c

3.2 Implementação OpenMP

A partir das métricas da implementação serial, foi feita a implementação paralela utilizando-se OpenMP. No Open MP há a necessidade de sincronização para evitar as condições de corrida. Como vantagens para a implementação, o OpenMP é simples e rápido, possui construções de alto nível, e ainda éativamente desenvolvido.

A implementação ocorreu através da declaração das variáveis “i”, “j” e “a” como variáveis privadas, na linha 43, pois estas devem ser únicas dentro de cada *thread*, enquanto as outras variáveis devem ser compartilhadas.

3.2.1 Código OpenMP

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef unsigned char cell_t;
4 #include <omp.h>
5 #include <time.h>
6 #define DEBUG = 1
7
8 cell_t ** allocate_board (int size) {
9     cell_t ** board = (cell_t **) malloc(sizeof(cell_t *) * size);
10    int i;
11    for (i=0; i<size; i++)
12        board[i] = (cell_t *) malloc(sizeof(cell_t) * size);
13    return board;
14 }
15
16 void free_board (cell_t ** board, int size) {
17     int i;
18     for (i=0; i<size; i++)
19         free(board[i]);
20     free(board);
21 }
22
23
24 /* return the number of on cells adjacent to the i,j cell */
```

```

25 int adjacent_to (cell_t ** board , int size , int i , int j) {
26     int k , l , count=0;
27
28     int sk = (i>0) ? i-1 : i ;
29     int ek = (i+1 < size) ? i+1 : i ;
30     int sl = (j>0) ? j-1 : j ;
31     int el = (j+1 < size) ? j+1 : j ;
32
33     for (k=sk; k<=ek; k++)
34         for (l=sl; l<=el; l++)
35             count+=board [k] [l];
36     count-=board [i] [j];
37     return count;
38 }
39
40 void play (cell_t ** board , cell_t ** newboard , int size) {
41     int i , j , a;
42     /* for each cell , apply the rules of Life */
43 #pragma omp parallel for private(i , j , a)
44     for (i=0; i<size; i++)
45         for (j=0; j<size; j++) {
46             a = adjacent_to (board , size , i , j);
47             if (a == 2) newboard [i] [j] = board [i] [j];
48             if (a == 3) newboard [i] [j] = 1;
49             if (a < 2) newboard [i] [j] = 0;
50             if (a > 3) newboard [i] [j] = 0;
51         }
52 }
53
54 /* print the life board */
55 void print (cell_t ** board , int size) {
56     int i , j;
57     /* for each row */
58     for (j=0; j<size; j++) {
59         /* print each column position... */
60         for (i=0; i<size; i++)
61             printf (" %c " , board [i] [j] ? 'x' : ' ');
62         /* followed by a carriage return */
63         printf ("\n");
64     }
65 }
66
67 /* read a file into the life board */
68 void read_file (FILE * f , cell_t ** board , int size) {
69     int i , j;
70     char *s = (char *) malloc (size+10);
71     char c;

```

```

72     for (j=0; j<size ; j++) {
73         /* get a string */
74         fgets (s, size+10,f);
75         /* copy the string to the life board */
76         for (i=0; i<size ; i++)
77         {
78             //c=fgetc(f);
79             //putchar(c);
80             board [ i ][ j ] = s [ i ] == 'x';
81         }
82         //fscanf(f, "\n");
83     }
84 }
85
86 int main () {
87     time_t start_t1 , end_t1;
88     double diff_t1;
89     int size , steps;
90     FILE *f;
91     f = fopen( "judge.in" , "r" );
92 //    f = stdin;
93     fscanf(f, "%d %d" , &size , &steps);
94     cell_t ** prev = allocate_board ( size );
95     read_file ( f , prev , size );
96     fclose(f);
97     cell_t ** next = allocate_board ( size );
98     cell_t ** tmp;
99     int i ,j ;
100 #ifdef DEBUG
101 //    printf(" Initial \n");
102 //    print(prev ,size );
103 //    printf("-----\n");
104 #endif
105     double begin = omp_get_wtime();
106     /* Laco for principal */
107     for ( i=0; i<steps ; i++) {
108         play ( prev ,next ,size );
109         #ifdef DEBUG
110 //            printf("%d ----- \n" , i );
111 //            print ( next ,size );
112 #endif
113         tmp = next ;
114         next = prev ;
115         prev = tmp ;
116     }
117     double end = omp_get_wtime();
118     double spent = (end - begin);

```

```

119   printf( "Tempo de execucao do laco
120   for principal:%.3f\n" , spent );
121 //  print ( prev , size );
122   printf( "%f\n" ,diff_t1 );
123   printf( "Tempo demorado pelo laco for com varios plays:
124   %2fs\n" ,diff_t1 );
125   free_board( prev , size );
126   free_board( next , size );
127 }
```

Listing 3.2 - lifeOpenMP.c.

3.3 Implementação MPI

A partir das métricas da implementação serial, foi feita a implementação paralela utilizando-se MPI. A implementação MPI é mais difícil de usar que a OpenMP e é necessário reestruturar o código serial. Como não há memória compartilhada, apenas distribuída, toda a comunicação é feita através da troca de mensagens.

O início da implementação MPI ocorre na função *main*, onde tem a inicialização do MPI e do *communicator* com *MPI_Init*, a determinação do tamanho do grupo associado ao *communicator* com *MPI_Comm_size*, e *MPI_Comm_rank* que define a ID do processo.

Após o início, temos a leitura do arquivo de entrada e a distribuição dos valores de *size* e *steps* para todos os processos com *MPI_Bcast*.

Na função *play* ocorre a troca de bordas, onde cada processo manda a sua borda para os processos vizinhos, já que é necessário que cada processo tenha a borda para realizar o seu próximo passo. Para o primeiro e último processos, temos apenas uma borda a ser enviada e uma a ser recebida, enquanto para os outros temos 2 bordas a serem enviadas e 2 a serem recebidas.

Como a matriz está na variável “*prev*”, criou-se a variável *local_matrix* para pegar o conteúdo inicial e distribuir os dados localmente entre todos os processos, cada processo tendo seus dados na sua matriz local, que é uma variável chamada “*local_matrix*”. Fazemos assim o “*MPI_send*” para enviar os dados do processo 0 para todos os outros processos e o “*MPI_Recv*” de todos os outros processos para receber seu dado local que o 0 mandou.

A seguir temos o laço *for* em que ocorre a série de chamadas na função *play*, sendo esse o laço que realiza todos os passos, pegando o dado inicial e entregando com o dado final em cada matriz local. Tendo os dados finais alocados localmente em cada processo, precisamos juntá-los no processo 0 para a impressão do resultado final. Sendo assim, após a junção dos dados, o dado final completo se localiza na *local_matrix* do processo 0.

3.3.1 Código MPI

```

1 #include <mpi.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 /* inclui a biblioteca time para calcular
5 o tempo do laco for principal */
6 #include <time.h>
7 typedef unsigned char cell_t;
8
9 #define DEBUG = 1
10
11 int my_size; /* size do MPI */
12 int my_rank; /* rank do MPI */
13 int i_bar, i_bar_max, resto_i_bar;
14
15 cell_t ** allocate_board (int size) {
16     cell_t ** board = (cell_t **)
17         malloc( sizeof(cell_t *) * size );
18     int i;
19     for (i=0; i<size; i++)
20         board[ i ] = (cell_t *)
21             malloc( sizeof(cell_t ) * size );
22     return board;
23 }
24
25 void free_board (cell_t ** board, int size) {
26     int i;
27     for (i=0; i<size; i++)
28         free( board[ i ] );
29     free( board );
30 }
31
32
33 /* return the number of on cells adjacent to the i,j cell */
34 int adjacent_to (cell_t ** board, int size, int i, int j) {
35     int k, l, count=0;
36     int sk = (i>0) ? i-1 : i;
37     int ek = (i+1 < size) ? i+1 : i;
38     int sl = (j>0) ? j-1 : j;
39     int el = (j+1 < size) ? j+1 : j;
40
41     for (k=sk; k<=ek; k++)
42         for (l=sl; l<=el; l++)
43             count+=board[ k ][ l ];

```

```

44     count=board[ i ][ j ];
45     return count;
46 }
47
48 void play ( cell_t ** board , cell_t ** newboard , int size ) {
49     int i , j , a;
50     MPI_Status status;
51
52 /* ----- Troca de bordas ----- */
53     if ( my_rank == 0 ){
54         /* Envio da ultima linha para o rank de baixo */
55         MPI_Send( ( &board [ i_bar - 1 ] [ 0 ] ) , ( size ) , MPI_CHAR ,
56             ( my_rank + 1 ) , 0 , MPI_COMM_WORLD );
57         /* Recebimento da borda inferior */
58         MPI_Recv( ( &board [ i_bar ] [ 0 ] ) , ( size ) , MPI_CHAR ,
59             ( my_rank + 1 ) , 0 , MPI_COMM_WORLD , &status );
60     }
61     else if (( my_rank != 0 ) && ( my_rank != ( my_size - 1 ) ) )
62     && ( i_bar == i_bar_max ) ){
63         /* Recebimento da borda inferior */
64         MPI_Recv( ( &board [ my_rank * i_bar - 1 ] [ 0 ] ) , ( size ) ,
65             MPI_CHAR , ( my_rank - 1 ) ,
66             0 , MPI_COMM_WORLD , &status );
67         /* Envio da primeira linha para o rank de cima */
68         MPI_Send( ( &board [ my_rank * i_bar ] [ 0 ] ) , ( size ) , MPI_CHAR ,
69             ( my_rank - 1 ) , 0 , MPI_COMM_WORLD );
70         /* Envio da ultima linha para o rank de baixo */
71         MPI_Send( ( &board [ ( my_rank + 1 ) * i_bar - 1 ] [ 0 ] ) ,
72             ( size ) , MPI_CHAR ,
73             ( my_rank + 1 ) , 0 , MPI_COMM_WORLD );
74         /* Recebimento da borda superior */
75         MPI_Recv( ( &board [ ( my_rank + 1 ) * i_bar ] [ 0 ] ) ,
76             ( size ) , MPI_CHAR ,
77             ( my_rank + 1 ) , 0 , MPI_COMM_WORLD , &status );
78     }
79     else if (( my_rank != 0 ) && ( my_rank != ( my_size - 1 ) ) )
80     && ( i_bar < i_bar_max ) ){
81         /* Recebimento da borda superior */
82         MPI_Recv( ( &board [ ( i_bar_max * resto_i_bar - 1 ) +
83             ( ( my_rank - resto_i_bar ) * i_bar ) ] [ 0 ] ) ,
84             ( size ) , MPI_CHAR , ( my_rank - 1 ) ,
85             0 , MPI_COMM_WORLD , &status );
86         /* Envio da primeira linha para o rank de cima */
87         MPI_Send( ( &board [ ( i_bar_max * resto_i_bar ) +
88             ( ( my_rank - resto_i_bar ) * i_bar ) ] [ 0 ] ) ,
89             ( size ) , MPI_CHAR , ( my_rank - 1 ) , 0 , MPI_COMM_WORLD );
90         /* Envio da ultima linha para o rank de baixo */

```

```

91     MPI_Send((&board[(i_bar_max*resto_i_bar+i_bar-1) +
92                     ((my_rank-resto_i_bar)*i_bar)][0]), 
93                 (size), MPI_CHAR, (my_rank+1), 0, MPI_COMM_WORLD);
94     /* Recebimento da borda inferior */
95     MPI_Recv(&board[i_bar_max*resto_i_bar+i_bar][0]),
96             (size), MPI_CHAR, (my_rank+1),
97             0, MPI_COMM_WORLD, &status);
98 }
99 else if (my_rank == (my_size-1)){
100    /* Recebimento da borda superior */
101    MPI_Recv(&board[size-i_bar-1][0]), (size), MPI_CHAR,
102            (my_rank-1), 0, MPI_COMM_WORLD, &status);
103    /* Envio da primeira linha para o rank de cima */
104    MPI_Send(&board[size-i_bar][0]), (size), MPI_CHAR,
105            (my_rank-1), 0, MPI_COMM_WORLD);
106 }
107
108 /* for each cell, apply the rules of Life */
109 if (i_bar == i_bar_max){
110    for (int i = (my_rank * i_bar_max);
111         ( i < ((my_rank + 1) * i_bar_max) ); i++){
112        for (j=0; j<size; j++) {
113            a = adjacent_to (board, size, i, j);
114            if (a == 2) newboard[i][j] = board[i][j];
115            if (a == 3) newboard[i][j] = 1;
116            if (a < 2) newboard[i][j] = 0;
117            if (a > 3) newboard[i][j] = 0;
118        }
119    }
120 }
121 else if ( (i_bar < i_bar_max) && (my_rank != (my_size-1)) ){
122    for (int i = ((my_rank * i_bar)+1);
123         ( i < ((my_rank * i_bar) + i_bar) ); i++){
124        for (j=0; j<size; j++) {
125            a = adjacent_to (board, size, i, j);
126            if (a == 2) newboard[i][j] = board[i][j];
127            if (a == 3) newboard[i][j] = 1;
128            if (a < 2) newboard[i][j] = 0;
129            if (a > 3) newboard[i][j] = 0;
130        }
131    }
132 }
133 else if (my_rank == (my_size-1)){
134    for (int i = (size-i_bar); ( i < (size) ); i++){
135        for (j=0; j<size; j++) {
136            a = adjacent_to (board, size, i, j);
137            if (a == 2) newboard[i][j] = board[i][j];

```

```

138         if (a == 3) newboard[i][j] = 1;
139         if (a < 2) newboard[i][j] = 0;
140         if (a > 3) newboard[i][j] = 0;
141     }
142 }
143 }
144 }
145
146 /* print the life board */
147 void print (cell_t ** board, int size) {
148     int i, j;
149     /* for each row */
150     for (j=0; j<size; j++) {
151         /* print each column position... */
152         for (i=0; i<size; i++)
153             printf ("%c", board[i][j] ? 'x' : ' ');
154         /* followed by a carriage return */
155         printf ("\n");
156     }
157 }
158
159 /* read a file into the life board */
160 void read_file (FILE * f, cell_t ** board, int size) {
161     int i, j;
162     char *s = (char *) malloc(size+10);
163     char c;
164     for (j=0; j<size; j++) {
165         /* get a string */
166         fgets (s, size+10,f);
167         /* copy the string to the life board */
168         for (i=0; i<size; i++) {
169             //c=fgetc(f);
170             //putchar(c);
171             board[i][j] = s[i] == 'x';
172         }
173         //fscanf(f, "\n");
174     }
175 }
176
177 int main (int argc, char** argv) {
178     /* Inicio do MPI */
179     MPI_Init(&argc, &argv);
180     MPI_Comm_size (MPI_COMM_WORLD, &my_size);
181     MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
182     MPI_Status status;
183     time_t start_t1, end_t1;
184     double diff_t1;

```

```

185 int size , steps;
186 FILE *f;
187 if (my_rank == 0) {
188     f = fopen("judge.in" , "r");
189 //     f = stdin;
190     fscanf(f , "%d %d" , &size , &steps);
191 }
192 /* Envio do valor de size e steps da entrada
193 para todos os processadores */
194 MPI_Bcast(&size , 1 , MPI_INT , 0 , MPI_COMM_WORLD);
195 MPI_Bcast(&steps , 1 , MPI_INT , 0 , MPI_COMM_WORLD);
196 cell_t ** prev = allocate_board (size);
197 if (my_rank == 0) {
198     read_file (f , prev , size);
199     fclose(f);
200 }
201
202 cell_t ** next = allocate_board (size);
203 cell_t ** tmp;
204 cell_t ** local_matrix = allocate_board (size);
205 int i,j;
206 /* i_bar e a divisao do size da matriz
207 pelo size de processadores */
208 i_bar = size/my_size;
209 /* resto_i_bar e o resto da divisao entre size da matriz
210 pelo size de processadores */
211 int resto_i_bar = size%my_size;
212 if (my_rank < resto_i_bar) {
213     /* i_bar_max e o maximo de linhas por rank
214     e i_bar e o numero de linhas local do rank */
215     i_bar_max = i_bar + 1;
216     i_bar = i_bar_max;
217 }
218 else if (resto_i_bar != 0) {
219     i_bar_max = i_bar + 1;
220 }
221 else if (resto_i_bar == 0) {
222     i_bar_max = i_bar;
223 }
224
225 #ifdef DEBUG
226 /* apenas o my_rank 0 imprime o initial */
227 // if (my_rank == 0) {
228 //     printf("Initial \n");
229 //     print(prev , size);
230 //     printf("-----\n");
231 // }

```

```

232 #endif
233
234 /* A matriz esta na variavel prev! */
235
236 /* ----- Inicializa a local_matrix do rank 0 ----- */
237 if ( my_rank == 0 ) {
238     for ( int i = 0; i < size; i++){
239         for ( int j = 0; j < size; j++){
240             local_matrix[ i ][ j ] = prev[ i ][ j ];
241         }
242     }
243     free_board( prev , size );
244     /*
245     if ( my_rank != 0 ) {
246         for ( int i = 0; i < size; i++){
247             for ( int j = 0; j < size; j++){
248                 local_matrix[ i ][ j ] = 0;
249             }
250         }
251     }
252 }
253 */
254 /* ----- Send do rank master para os outros ranks ----- */
255 if ( (my_rank == 0) && (resto_i_bar != 0) ) {
256     /* Esse for funciona para mandar para
257     os outros 'i' ranks */
258     for ( int i = 1; i < (resto_i_bar); i++) {
259         /* Esse for funciona para mandar
260         para as 'j' linhas da matriz */
261         for ( int j=(i*i_bar_max);
262              j < ( (i+1)*i_bar_max ); j++){
263             MPI_Send( (&local_matrix[ j ][ 0 ]) , ( size ) ,
264                     MPI_CHAR, i , 0 ,MPI_COMM_WORLD );
265         }
266     }
267     /* Esse for funciona para mandar para
268     os outros 'i' ranks */
269     for ( int i = resto_i_bar; i < (my_size); i++) {
270         /* Esse for funciona para mandar
271         para as 'j' linhas da matriz */
272         for ( int j= ( (i_bar_max*resto_i_bar) +
273                      ((i-resto_i_bar)*(i_bar_max-1)) ) ;
274              j < ( (i_bar_max*resto_i_bar) +
275                      ((i-resto_i_bar+1)*(i_bar_max-1)) ) ; j++) {
276             MPI_Send( (&local_matrix[ j ][ 0 ]) , ( size ) ,
277                     MPI_CHAR, i , 0 ,MPI_COMM_WORLD );
278         }
}

```

```

279     }
280 }
281 else if ( (my_rank == 0) && (resto_i_bar == 0) ) {
282     /* Esse for funciona para mandar para
283     os outros 'i' ranks */
284     for (int i = 1; i < (my_size); i++) {
285         /* Esse for funciona para mandar
286         para as 'j' linhas da matriz */
287         for (int j=(i*i_bar_max);
288              j < ( (i+1)*i_bar_max ); j++) {
289             MPI_Send( (&local_matrix[j][0]) , (size),
290                      MPI_CHAR, i , 0 ,MPI_COMM_WORLD );
291         }
292     }
293 }
294
295 /* ----- Receive dos ranks slaves ----- */
296 else if ( (my_rank != 0) && (i_bar == i_bar_max) ) {
297     /* Esse for funciona para
298     receber as 'i' linhas da matriz */
299     for (int i = (my_rank * i_bar_max);
300          i < ((my_rank + 1)*i_bar_max); i++ ) {
301         MPI_Recv( (&local_matrix[i][0]) , (size),
302                  MPI_CHAR, 0 , 0 , MPI_COMM_WORLD, &status );
303     }
304 }
305 else if ( (my_rank != 0) && (i_bar < i_bar_max) ) {
306     /* Esse for funciona para
307     receber as 'i' linhas da matriz */
308     for (int i = ( (i_bar_max*resto_i_bar) +
309                  ((my_rank-resto_i_bar)*(i_bar_max-1)) );
310          i < ((i_bar_max*resto_i_bar) +
311                  ((my_rank-resto_i_bar+1)*(i_bar))) ; i++ ) {
312         MPI_Recv( (&local_matrix[i][0]) , (size),
313                  MPI_CHAR, 0 , 0 , MPI_COMM_WORLD, &status );
314     }
315 }
316
317 /* ----- Etapa em que ocorre a serie de plays ----- */
318 MPI_Barrier(MPI_COMM_WORLD);
319 localbegin = MPI_Wtime();
320 /* Laco for principal */
321 for ( i=0; i<steps; i++) {
322     play (local_matrix,next,size);
323 #ifdef DEBUG
324 //     printf("%d ----- My_rank = %d\n", i , my_rank);
325 //     if (my_rank == 0 ) {

```

```

326 //      print( next , size );
327 //    }
328 #endif
329 tmp = next;
330 next = local_matrix;
331 local_matrix = tmp;
332 }
333 localend = MPI_Wtime();
334 localspent = (localbegin - localend);
335 MPI_Reduce(&localspent , &spent , 1 , MPI_DOUBLE,
336 MPI_MAX, 0 , MPI_COMM_WORLD );
337 if ( my_rank ==0 ){
338   printf("Tempo de execucao do laco
339   for principal:%.3f\n" , spent );
340 }
341 /* _____ Send dos slaves para o rank 0
342 para juncao da matriz final _____ */
343 if ( (my_rank != 0) && (i_bar == i_bar_max) ) {
344   /* Esse for funciona para mandar
345   para as 'i' linhas da matriz */
346   for (int i = (my_rank * i_bar_max);
347 ( i < ((my_rank + 1)*i_bar_max) ); i++ ){
348     MPI_Send( (&local_matrix[ i ][ 0 ]) , (size) ,
349     MPI_CHAR, 0 , 0 ,MPI_COMM_WORLD );
350   }
351 }
352 else if ( (my_rank != 0) && (i_bar < i_bar_max) ) {
353   /* Esse for funciona para mandar
354   para as 'i' linhas da matriz */
355   for (int i = ((i_bar_max*resto_i_bar) +
356 ((my_rank-resto_i_bar)*i_bar)) ;
357 ( i < ((i_bar_max*resto_i_bar+i_bar) +
358 ((my_rank-resto_i_bar)*(i_bar))) ); i++ ) {
359     MPI_Send( (&local_matrix[ i ][ 0 ]) , (size) ,
360     MPI_CHAR, 0 , 0 ,MPI_COMM_WORLD );
361   }
362 }
363 /* _____ Receive do rank 0 para juncao da matriz final _____ */
364 else if ((my_rank == 0) && (resto_i_bar != 0)) {
365   /* Esse for funciona para receber
366   dos outros 'i' ranks */
367   for (int i = 1; i < (resto_i_bar); i++) {
368     /* Esse for funciona para receber
369     as 'j' linhas da matriz */
370     for (int j=(i*i_bar_max); j < ( (i+1)*i_bar_max ) ;
371 j++) {
372       MPI_Recv( (&local_matrix[ j ][ 0 ]) , (size) ,

```

```

373         MPI_CHAR, i , 0 , MPI_COMM_WORLD, &status ) ;
374     }
375 }
376 /* Esse for funciona para receber
377 dos outros 'i' ranks */
378 for ( int i = resto_i_bar; i < (my_size); i++) {
379     /* Esse for funciona para receber
380     as 'j' linhas da matriz */
381     for ( int j= ( (i_bar_max*resto_i_bar) +
382 ((i-resto_i_bar)*(i_bar_max-1)) );
383 j < ( (i_bar_max*resto_i_bar) +
384 ((i-resto_i_bar+1)*(i_bar_max-1)) ); j++){
385         MPI_Recv( (&local_matrix [ j ][ 0 ]), (size),
386 MPI_CHAR, i , 0 , MPI_COMM_WORLD, &status );
387     }
388 }
389 }
390 else if ((my_rank == 0) && (resto_i_bar == 0)) {
391     /* Esse for funciona para receber
392 dos outros 'i' ranks */
393     for ( int i = 1; i < (my_size); i++) {
394         /* Esse for funciona para receber
395         as 'j' linhas da matriz */
396         for ( int j=(i*i_bar_max); j < ( (i+1)*i_bar_max );
397 j++) {
398             MPI_Recv( (&local_matrix [ j ][ 0 ]), (size),
399 MPI_CHAR, i , 0 , MPI_COMM_WORLD, &status );
400         }
401     }
402 }
403 /*
404 /* apenas o my_rank 0 imprime o final */
405 if (my_rank == 0){
406     print (local_matrix ,size );
407 }
408 */
409 free_board(next ,size );
410 free_board(local_matrix ,size );
411 MPI_Finalize(); // finalizando o MPI
412 }
413 }
```

Listing 3.3 - lifeMPI.c

3.4 Tutorial de execução

As subseções a seguir contém os códigos utilizados para executar cada programa escrito na linguagem de programação C. Os arquivos “life.c”, “lifeopenmp.c”, “lifempi.c” devem estar todos na mesma pasta que o arquivo “judge.in” para o devido funcionamento do programa. Temos a seguir os passos para compilar e executar os códigos.

- Passos para compilar e executar o código serial em uma máquina linux:
 - Abrir o terminal;
 - entrar na pasta que contém os arquivos através do comando cd;
 - compilar o arquivo life.c usando o gcc e colocar na saída life;
 - * "gcc life.c -o life"
 - executar o arquivo compilado life;
 - * "./life"
- Passos para compilar e executar o código OpenMP em uma máquina linux:
 - Abrir o terminal;
 - entrar na pasta que contém os arquivos através do comando cd;
 - exportar o número de *threads* que deseja utilizar;
 - * export OMP_NUM_THREADS=2
 - compilar o arquivo lifeopenmp.c usando o gcc e colocar na saída lifeopenmp;
 - * "gcc lifeopenmp.c -o lifeopenmp -fopenmp"
 - executar o arquivo compilado lifeopenmp;
 - * "./lifeopenmp"
- Passos para compilar e executar o código MPI em uma máquina linux:
 - Abrir o terminal;
 - entrar na pasta que contém os arquivos através do comando cd;
 - compilar o arquivo lifempi.c usando o mpicc e colocar na saída lifempi;
 - * "mpicc lifempi.c -o lifempi"
 - executar o arquivo compilado lifempi com o número de processadores desejado;
 - * "mpirun -np 2 lifempi"

4 ANÁLISE DE DESEMPENHO

Para se fazer uma comparação justa, todos os testes são feitos em uma mesma máquina, de modelo Intel Xeon Platinum 8275CL, com tipo de instância c5.12xlarge, com **24 processadores** e 48 VCPUs, Figura 15. Foram gastos em média \$32 dólares e mais de 12 horas de uso da nuvem *Amazon Web Services* (AWS), para se ter certeza que o código executa independente do número de processos, e para toda a análise de desempenho. Foram utilizados números de *threads/processos* aleatórios definidos pelo autor para a análise de desempenho.

Em nenhum dos casos está sendo impresso o resultado da execução na tela, pois no MPI os dados estão separados entre os processos e ocorre apenas a troca de bordas entre eles, a junção se dá somente no final. Imprime-se apenas o tempo total de execução do laço principal do código. Temos o dado completo em um processo somente no início e final, ou seja, só se tem uma impressão válida da matriz inicial que tem os dados do arquivo "judge.in" e da matriz final, que é o resultado ao fim de todos os passos. Como a máquina tem 24 processadores, não há sentido em colocar mais de 24 processos ou *threads*, pois há um *overhead* grande que faz com que o tempo de execução aumente.

A análise se dá pela execução do código por 10 vezes, e é tirada a média e o desvio padrão dessas execuções. A partir da média e do desvio padrão tem-se a criação de suas devidas tabelas e gráficos.

A partir dos dados da análise pode-se calcular o *speedup* e a eficiência paralela. O *speedup* é calculado através da equação $S = \frac{T(s)}{T(p)}$, onde S é o *speedup*, "T(s)" é a média do tempo de execução do código serialmente enquanto "T(p)" é a média do tempo de execução do código paralelamente.

A partir do *speedup* pode-se calcular a eficiência, através da equação $E = \frac{S}{p}$, onde "E" é a eficiência, "S" é o *speedup* e "p" é o número de *threads*.

Pode-se perceber que a eficiência vai diminuindo com o aumento do número de processos ou *threads*, e isto ocorre devido ao fato de haver um *overhead* quando se utiliza a programação paralela. Esse *overhead* ocorre por alguns motivos como:

- sincronizações;
- comunicação de dados;
- tempo de início de *threads* e processos.

```
ubuntu@ip-172-31-41-7:~/final$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                48
On-line CPU(s) list:  0-47
Thread(s) per core:   2
Core(s) per socket:   24
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz
Stepping:               7
CPU MHz:               1363.370
BogoMIPS:              6000.00
Hypervisor vendor:    KVM
Virtualization type:  full
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              36608K
NUMA node0 CPU(s):    0-47
```

Figura 15 - Máquina c5.12xlarge na nuvem AWS.

4.1 Execução Serial

A análise do código rodando serialmente se dá pelo cálculo do tempo que é demorado no laço *for* principal, que envolve a série de chamadas da função *play*. A análise na máquina c5.12xlarge computou 323,51 segundos.

4.2 Análise OpenMP

A análise do código executando com OpenMP se dá pelo cálculo do tempo que é demorado no laço *for* que envolve a série de chamadas da função *play*. Sendo essa análise feita várias vezes, cada vez para um determinado número de *threads*.

Pode-se ver pela análise pela Tabela 1, que o menor tempo de execução do código foi de 14,85 segundos para 24 *threads*.

Pela análise do gráfico 16 percebe-se um enorme ganho de tempo de execução até um certo número de *threads*. Para o caso do código executando com o arquivo judge.in temos que o ganho se deu até o valor de 24 *threads*, que a máquina possui.

A partir dos tempos de execução do código, foi calculado o *speedup*, como mostra a Tabela 2. Colocando os dados da Tabela 2 em um gráfico, tem-se a Figura 17, que mostra um ganho quase linear até as 24 *threads*, sugerindo que com uma máquina com

Theads	Tempo (s)	Desvio padrão
1	323,51	0,14
2	165,08	0,16
4	85,77	0,03
6	57,77	0,03
8	43,32	0,05
10	34,69	0,08
12	28,97	0,04
16	21,74	0,01
20	17,39	0,00
24	14,85	0,01
30	22,47	0,05

Tabela 1 - Média dos tempos e desvio padrão da análise de desempenho OpenMP.

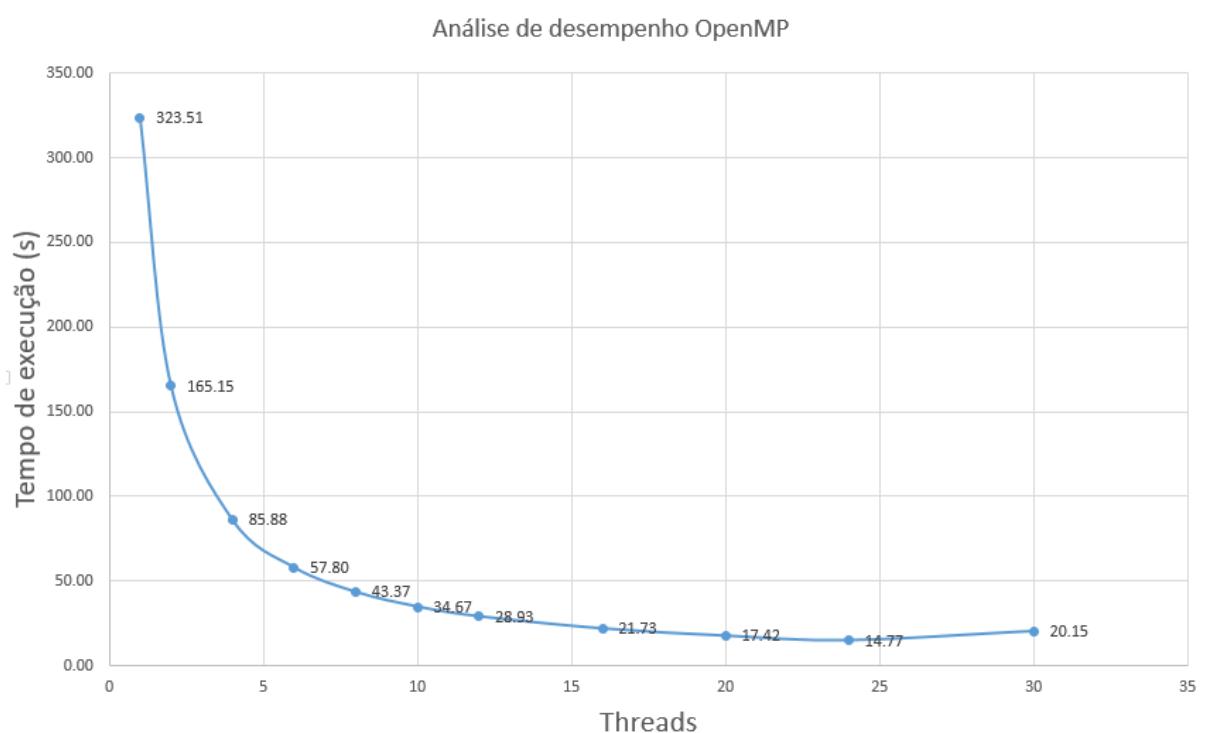


Figura 16 - Gráfico com os tempos de execução (em segundos) para diferentes números de threads.

Threads	Speedup
1	1
2	1,960
4	3,772
6	5,600
8	7,468
10	9,326
12	11,168
16	14,883
20	18,560
24	21,783
30	14,395

Tabela 2 - *Speedup* do OpenMP, calculado a partir da média dos tempos de execução.

mais processadores ainda poderia obter-se um ganho de tempo.

A partir do *speedup*, foi calculada a eficiência paralela, como mostra a Tabela 2. Colocando os dados da Tabela 3 em um gráfico, tem-se a Figura 18, que mostra que para esse código executando nessa máquina com 24 *threads* o *overhead* ainda é pequeno, mas demonstrando que a para 30 *threads* a eficiência cai bastante.

4.3 Análise MPI

A análise do código rodando MPI se dá pelo cálculo do tempo de execução no laço *for* que envolve a série de chamadas da função *play*. Sendo essa análise feita várias vezes, cada vez para um determinado número de processos.

Pode-se ver pela análise pela Tabela 4 que o menor tempo de execução do código foi de 14,77 segundos para 24 processos.

Pela análise do gráfico 19 percebe-se um enorme ganho de tempo de execução até um certo número de processos. Para o caso do código executando com o arquivo *judge.in* temos que o ganho se deu até o valor de 24 processos, que a máquina possui.

A partir dos tempos de execução do código, foi calculado o *speedup*, como mostra a Tabela 5. Colocando os dados da Tabela 5 em um gráfico, tem-se a Figura 20, que mostra um ganho quase linear até os 24 processos, sugerindo também para o MPI que com uma máquina com mais processadores ainda poderia obter-se um ganho de tempo.

A partir do *speedup*, foi calculada a eficiência paralela, como mostra a Tabela 5. Colocando os dados da Tabela 6 em um gráfico, tem-se a Figura 21, que mostra que

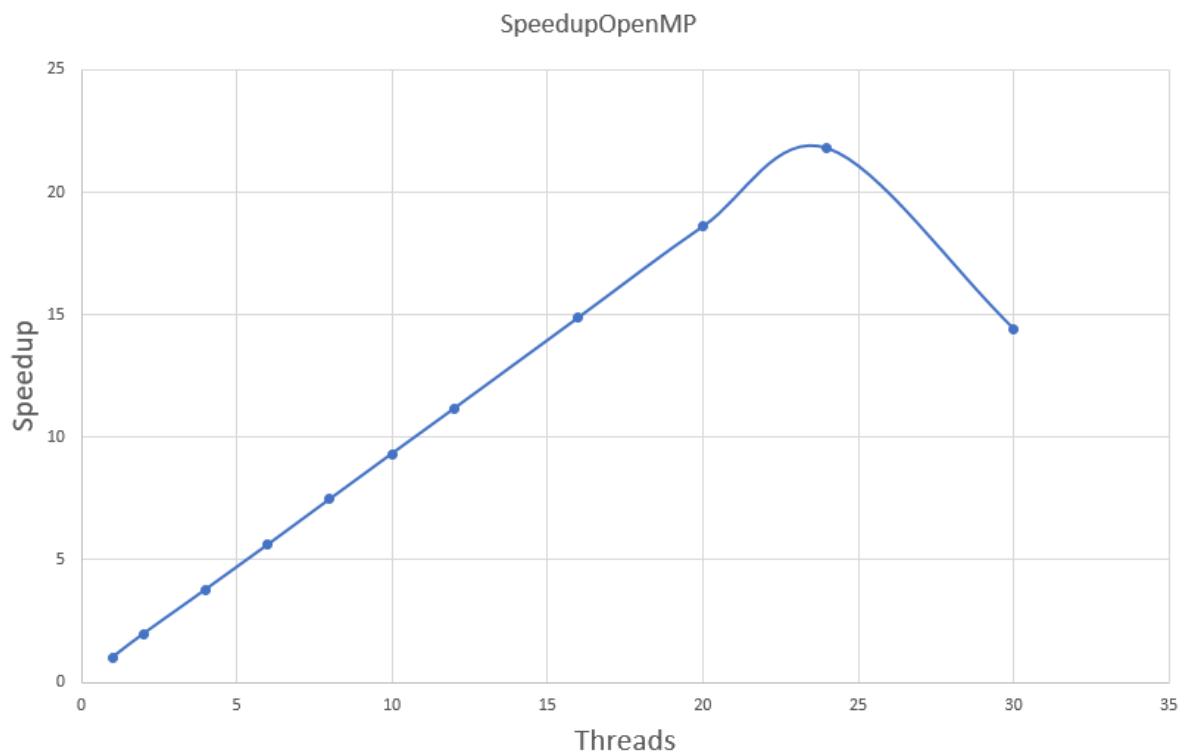


Figura 17 - Gráfico com o valor de *speedup* para diferentes números de *threads*.

Threads	Eficiência
1	1,00
2	1,980
4	1,943
6	1,933
8	1,933
10	1,933
12	1,930
16	1,930
20	1,930
24	1,907
30	0,480

Tabela 3 - Tabela com a eficiência do OpenMP,
calculado a partir da média dos
tempos de execução.

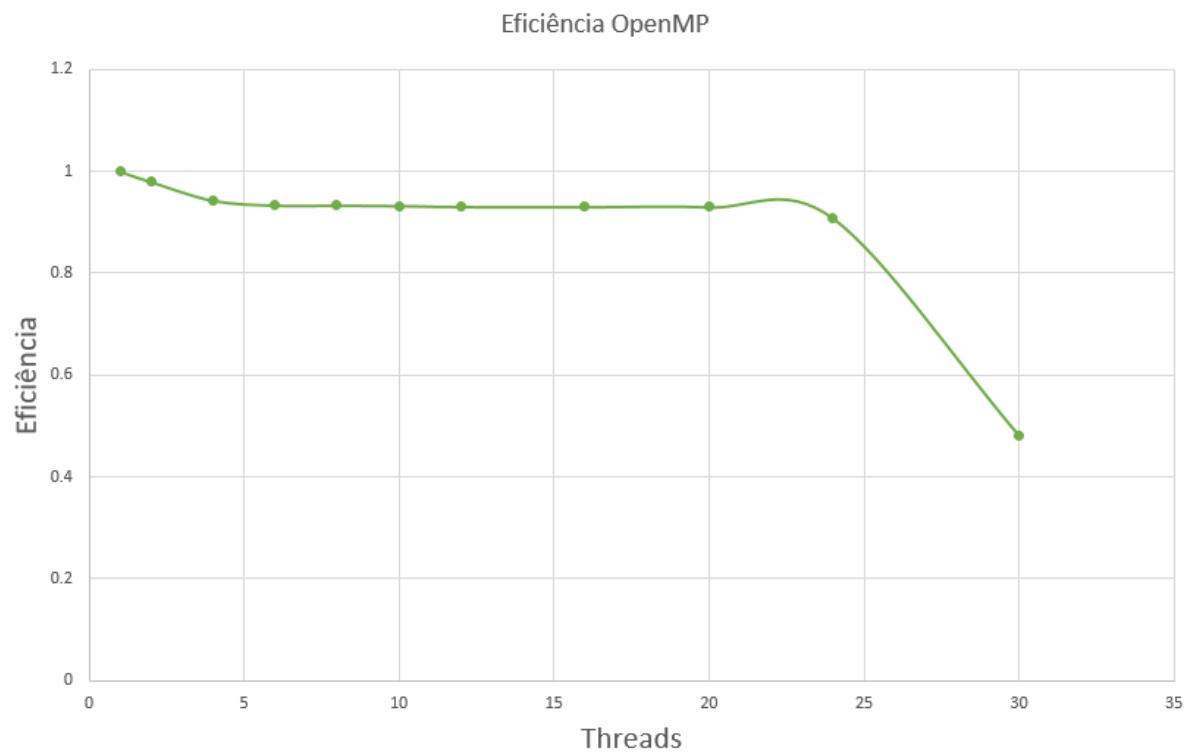


Figura 18 - Gráfico da eficiência para determinado número de *threads*.

Processos	Tempo (s)	Desvio padrão
1	323,51	0,14
2	165,15	0,07
4	85,88	0,34
6	57,80	0,01
8	43,37	0,08
10	34,67	0,01
12	28,93	0,02
16	21,73	0,04
20	17,42	0,03
24	14,77	0,10
30	20,15	0,18

Tabela 4 - Tabela com a média dos tempos e desvio padrão da análise de desempenho MPI.

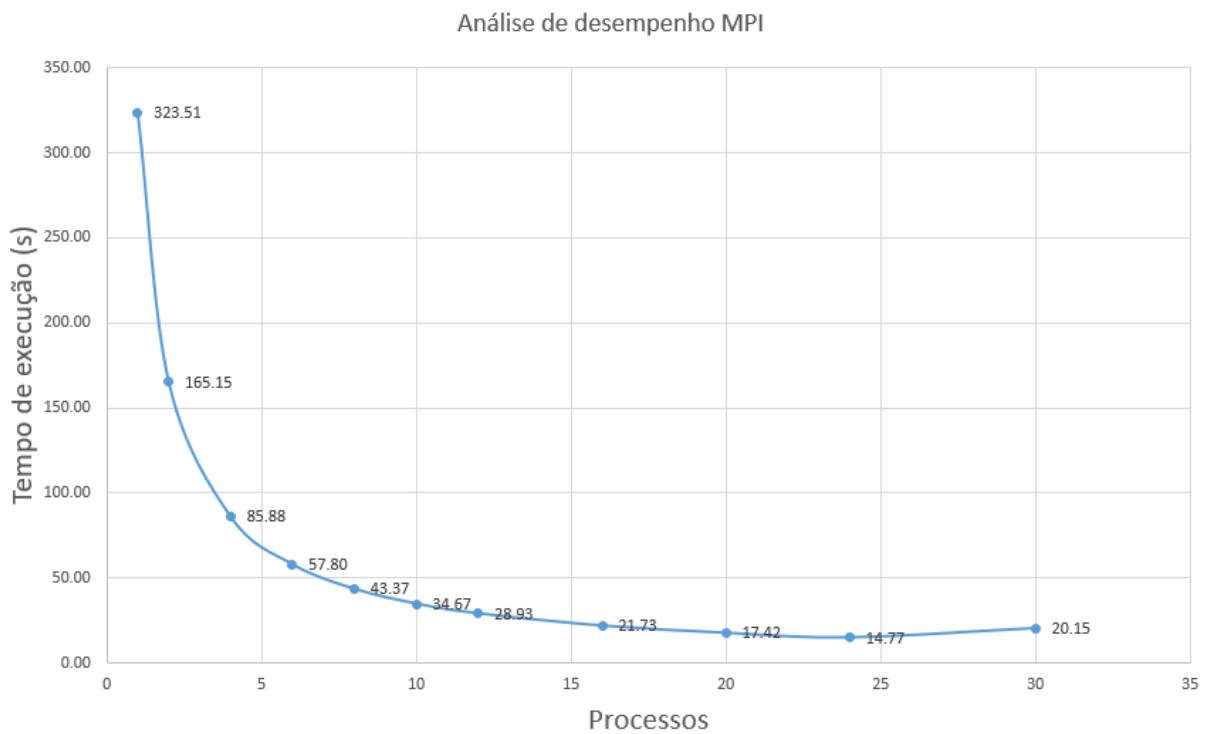


Figura 19 - Gráfico com os tempos de execução (em segundos) para diferentes números de processos.

Processos	Speedup
1	1
2	1,959
4	3,767
6	5,597
8	7,459
10	9,331
12	11,184
16	14,887
20	18,574
24	21,900
30	16,054

Tabela 5 - Speedup do MPI, calculado a partir da média dos tempos de execução.

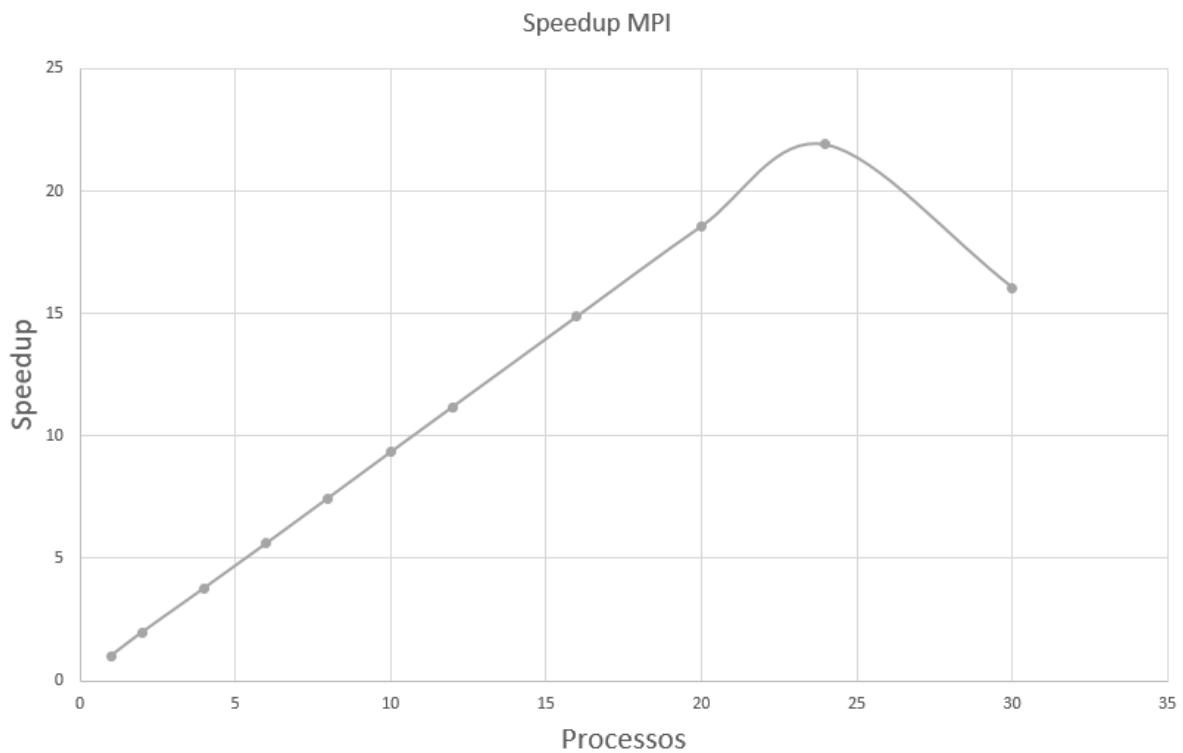


Figura 20 - Gráfico com o valor de speedup para diferentes números de processos.

Processos	Eficiência
1	1,00
2	0,979
4	0,942
6	0,933
8	0,932
10	0,933
12	0,932
16	0,930
20	0,929
24	0,913
30	0,535

Tabela 6 - Tabela com a eficiência do MPI, calculado a partir da média dos tempos.

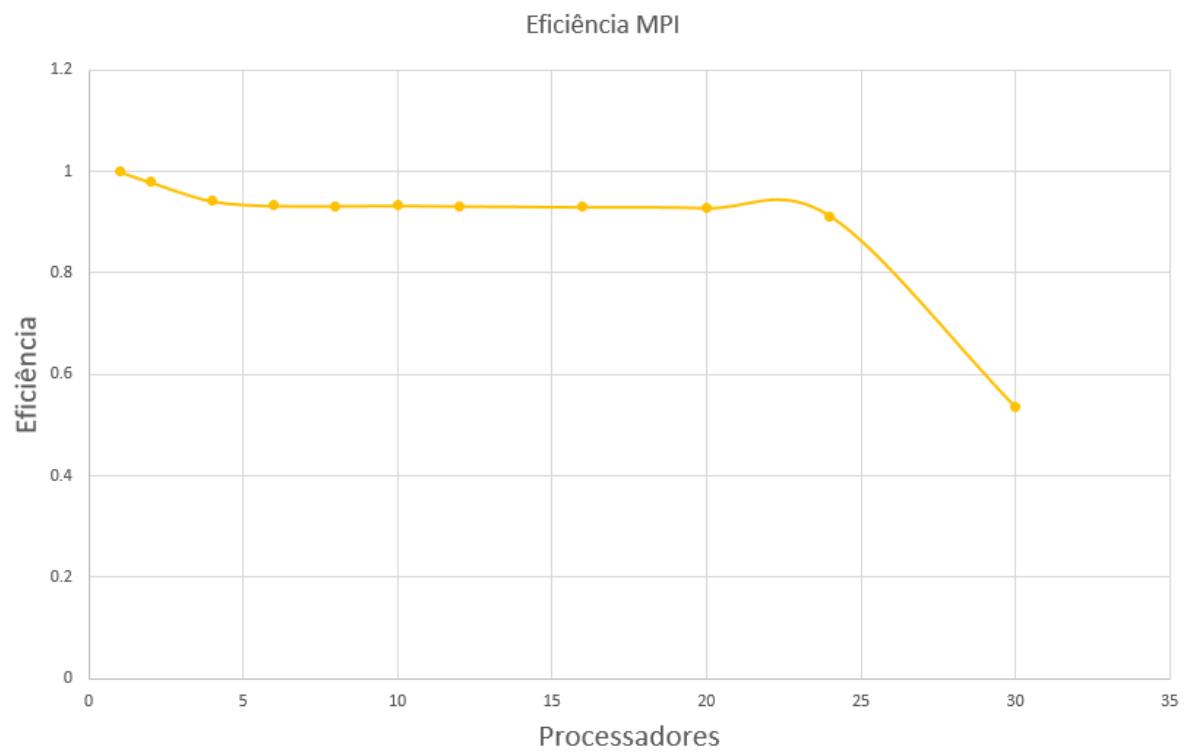


Figura 21 - Gráfico da eficiência para determinado número de *threads*.

para esse código executando nessa máquina com 24 processos, assim como no OpenMP, o *overhead* ainda é pequeno, mas demonstrando que para 30 processos a eficiência cai bastante.

5 CONCLUSÕES

Conclui-se por meio de todos os testes realizados que a programação paralela possibilita o ganho de tempo, pois com a elevada quantidade de processadores que se tem nos dias de hoje, um programa executando serialmente deixa essa grande quantidade de processadores ociosos. A utilização de máquinas com grande quantidade de processadores e *clusters* se torna algo fundamental para quem busca um alto desempenho em computação.

Analizando a Tabela 4 e Figura 19, observa-se que não há mais ganho de velocidade plausível a partir de determinado número de processos, assim como ocorre com as *threads* no OpenMP. Como a máquina utilizada tem 24 processadores, não há sentido em colocar mais do que 24 processos/*threads*, pois eles terão pouco trabalho a fazer e a sincronização e comunicação passam a ter um peso maior no tempo de execução.

Para o código que executa serialmente, com um tempo de execução médio de 323,51 segundos para a entrada "judge.in", viu-se que esse tempo de execução pode ser reduzido em mais de 90% paralelizando-o. Através do cálculo de *speedup* se percebe um aumento de velocidade, que chega a ser aproximadamente vinte e duas vezes mais rápido para 24 processos ou *threads*.

Comparando o desempenho OpenMP com MPI, cada um tem um particularidade que acabou deixando os tempos muito parecidos. Enquanto o OpenMP tem que criar as *threads* todas as vezes em que a função *play* é chamada, ocasionando assim uma perda do desempenho, o MPI tem a criação apenas uma vez dos processos no início, mas tem de realizar diversas trocas de mensagem entre seus processos.

Por fim, tem-se que paralelizar tanto por OpenMP quanto MPI nos dá um ganho de tempo de execução enorme comparado ao mesmo código rodando serialmente, como mostra o *speedup*.

Como perspectiva de trabalhos futuros, pode ser utilizado para uma adaptação de códigos seriais utilizados em outras áreas, já que *The Game of Life* é um jogo interessante para biólogos, matemáticos, estatísticos, economistas, filósofos e artistas por permitir a observação do modo como imagens complexas podem surgir de implementações de regras muito simples.

REFERÊNCIAS

- BARNEY, Blaise. *Introduction to parallel computing*. 2004. Disponível em: <https://computing.llnl.gov/tutorials/parallel_comp/#Whatis>. Acesso em: 05 dez. 2019.
- _____. *MPI Tutorial*. 2019. Disponível em: <<https://computing.llnl.gov/tutorials/mpi>>. Acesso em: 26 nov. 2019.
- BOSCO Sobral. 2010. Disponível em: <http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/Conceitos_OpenMP.pdf>. Acesso em: 21 nov. 2019.
- FORK-JOIN. 2007. Disponível em: <https://pt.wikipedia.org/wiki/Modelo_Fork-Join#/media/Ficheiro:Fork_join.svg>. Acesso em: 22 nov. 2019.
- JUNIOR, Leonardo Mattes Claudio Penasio. *Análise Comparativa de Desempenho de um algoritmo Paralelo implementado nas Linguagens de Programação CPAR e OpenMP: Programação paralela e distribuída*. 2004. 20 f. Monografia — Universidade de São Paulo, São Paulo, 2004.
- MARATONA de programação paralela. 2016. Disponível em: <<http://lspd.mackenzie.br/marathon/16/problems.pt.html>>. Acesso em: 21 nov. 2019.
- OPEN-MPI-ORG. 2004. Disponível em: <<https://www.open-mpi.org/>>. Acesso em: 26 nov. 2019.
- PACHECO, Peter S. *An Introduction to Parallel Programming*. 1. ed. [S.l.]: Morgan Kaufmann, 2011.
- QUINN, Michael J. *MPI Tutorial*. 2015. Disponível em: <<https://slideplayer.com/slide/5088918/>>. Acesso em: 26 nov. 2019.
- WIKIPEDIA. 2005. Disponível em: <https://pt.wikipedia.org/wiki/Jogo_da_vida>. Acesso em: 26 nov. 2019.

APÊNDICE A – O código utilizado neste trabalho está disponível no github:
<https://github.com/frangeldc/TCC>