Universidade de Brasília

Faculdade do Gama - FGA

Programação para Sistemas Paralelos e Distribuídos - Turma A

Prof.: Fernando W. Cruz

Projeto Final

Autor:

Gabriela da Gama Pivetta - 180052845

Murilo Gomes de Souza - 180025601

Brasília

24 de julho de 2023

Sumário

1 Introdução

O propósito deste projeto é desenvolver uma aplicação baseada em um código fornecido, que atenda aos requisitos de performance e elasticidade, permitindo que ela se comporte como uma aplicação de larga escala (*large-scale application*).

Esta aplicação deve conter alguns requisitos:

- **Requisito de Performance**: Deve ser feito utilizando o Apache Spark e MPI/OpenMP com o intuito de melhorar o desempenho do código. Essas opções de paralelismo são denominadas engines.
- **Requisito de Elasticidade**: Deve ser elaborado utilizando o orquestrador Kubernetes, para que a aplicação se possa adaptar às mudanças de carga sem afetar a performance ou a qualidade do serviço. oferecido

Além disso, os serviços da aplicação devem ser providos por uma interface de acesso via rede (Socket server), por onde são recebidas as requisições dos usuários. Os valores enviados ao Socket Server são referentes às variáveis POWMIN e POWMAX do código, que se referem ao intervalo do jogo da vida no qual o usuário quer calcular os estágios. Essas informações podem originar de um arquivo ou de um Socket cliente (onde as entradas são digitadas).

Ainda mais, deve ser configurada uma instância do banco de dados ElasticSearch/Kibana, para registrar e apresentar, graficamente as contabilizações de processamento dos engines.

Assim, podemos resumir a arquitetura deste projeto com a Figura 1.1.

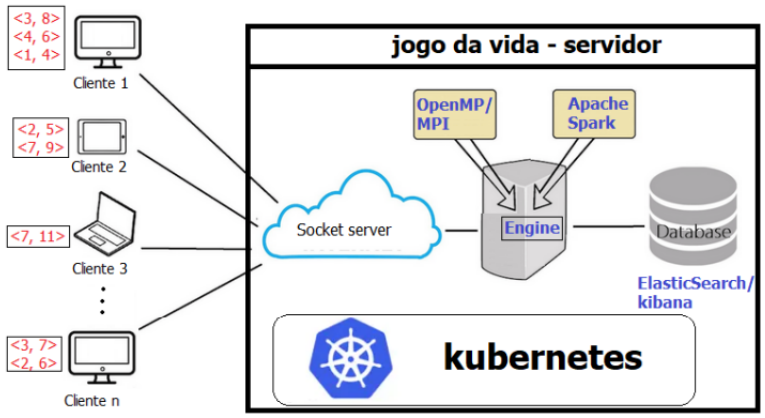

Figura 1.1: Arquitetura da aplicação. Fonte: Enunciado do trabalho

A seguir, o código original do Jogo da Vida fornecido e utilizado na resolução deste projeto:

Listing 1.1: Jogo da vida

```
1 #include <stdio.h>
```

```
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #define ind2d(i,j) (i)*(tam+2)+j
5  #define POWMIN 3
6  #define POWMAX 10
7
8  double wall_time(void) {
9      struct timeval tv; struct timezone tz;
10     gettimeofday(&tv, &tz);
11     return(tv.tv_sec + tv.tv_usec/1000000.0);
12 } /* fim-wall_time */
13
14 double wall_time(void);
15
16 void UmaVida(int* tabulIn, int* tabulOut, int tam) {
17     int i, j, vizviv;
18     for (i=1; i<=tam; i++) {
19         for (j= 1; j<=tam; j++) {
20             vizviv = tabulIn[ind2d(i-1,j-1)] + tabulIn[ind2d(i-1,j )
                   ] +
21             tabulIn[ind2d(i-1,j+1)] + tabulIn[ind2d(i ,j-1)] +
22             tabulIn[ind2d(i ,j+1)] + tabulIn[ind2d(i+1,j-1)] +
23             tabulIn[ind2d(i+1,j )] + tabulIn[ind2d(i+1,j+1)];
24             if (tabulIn[ind2d(i,j)] && vizviv < 2)
25                 tabulOut[ind2d(i,j)] = 0;
26             else if (tabulIn[ind2d(i,j)] && vizviv > 3)
27                 tabulOut[ind2d(i,j)] = 0;
28             else if (!tabulIn[ind2d(i,j)] && vizviv == 3)
29                 tabulOut[ind2d(i,j)] = 1;
30             else
31                 tabulOut[ind2d(i,j)] = tabulIn[ind2d(i,j)];
32         } /* fim-for */
33     } /* fim-for */
34 } /* fim-UmaVida */
35
36 void DumpTabul(int * tabul, int tam, int first, int last, char* msg)
       {
37     int i, ij;
38     printf("%s; Dump posi  es [%d:%d, %d:%d] de tabuleiro %d x %d\
          n", \
39     msg, first, last, first, last, tam, tam);
40     for (i=first; i<=last; i++)
41         printf("="); printf("=\n");
42     for (i=ind2d(first,0); i<=ind2d(last,0); i+=ind2d(1,0)) {
43         for (ij=i+first; ij<=i+last; ij++)
```

```
44              printf("%c", tabul[ij]? 'X' : '.');
45          printf("\n");
46      }
47      for (i=first; i<=last; i++)
48          printf("=");
49      printf("=\n");
50  } /* fim-DumpTabul */
51
52  void InitTabul(int* tabulIn, int* tabulOut, int tam){
53      int ij;
54      for (ij=0; ij<(tam+2)*(tam+2); ij++) {
55          tabulIn[ij] = 0;
56          tabulOut[ij] = 0;
57      } /* fim-for */
58      tabulIn[ind2d(1,2)] = 1; tabulIn[ind2d(2,3)] = 1;
59      tabulIn[ind2d(3,1)] = 1; tabulIn[ind2d(3,2)] = 1;
60      tabulIn[ind2d(3,3)] = 1;
61  } /* fim-InitTabul */
62
63  int Correto(int* tabul, int tam){
64      int ij, cnt;
65      cnt = 0;
66      for (ij=0; ij<(tam+2)*(tam+2); ij++)
67          cnt = cnt + tabul[ij];
68      return (cnt == 5 && tabul[ind2d(tam-2,tam-1)] &&
69      tabul[ind2d(tam-1,tam )] && tabul[ind2d(tam ,tam-2)] &&
70      tabul[ind2d(tam ,tam-1)] && tabul[ind2d(tam ,tam )]);
71  } /* fim-Correto */
72
73  int main(void) {
74      int pow, i, tam, *tabulIn, *tabulOut;
75      char msg[9];
76      double t0, t1, t2, t3;
77      // para todos os tamanhos do tabuleiro
78      for (pow=POWMIN; pow<=POWMAX; pow++) {
79          tam = 1 << pow;
80          // aloca e inicializa tabuleiros
81          t0 = wall_time();
82          tabulIn = (int *) malloc ((tam+2)*(tam+2)*sizeof(int));
83          tabulOut = (int *) malloc ((tam+2)*(tam+2)*sizeof(int));
84          InitTabul(tabulIn, tabulOut, tam);
85          t1 = wall_time();
86          for (i=0; i<2*(tam-3); i++) {
87              UmaVida(tabulIn, tabulOut, tam);
88              UmaVida(tabulOut, tabulIn, tam);
```

```
89          } /* fim-for */
90          t2 = wall_time();
91          if (Correto(tabulIn, tam))
92              printf("**Ok, RESULTADO CORRETO**\n");
93          else
94              printf("**Nok, RESULTADO ERRADO**\n");
95          t3 = wall_time();
96          printf("tam=%d; tempos: init=%7.7f, comp=%7.7f, fim=%7.7f,
               tot=%7.7f \n",
97          tam, t1-t0, t2-t1, t3-t2, t3-t0);
98          free(tabulIn); free(tabulOut);
99      }
100     return 0;
101 } /* fim-main */
```

2 Metodologia

Para a execução desse projeto, começamos com o requisito de Performance. Primeiro foram adicionados o MPI e o OpenMP ao código do Jogo da Vida. Em seguida e separadamente, fizemos a adaptação do código original para Python para então implementar o Spark.

Tendo a parte de Performance pronta, partimos para a elaboração do requisito de Elasticidade. Utilizamos o programa "minikube" para rodar o Kubernetes localmente. Em seguida, precisavamos criar o Socket Server, então desenvolvemos um algoritmo em C para ser esse servidor. Também criamos uma imagem Docker para executá-lo.

Embora tenhamos conseguido implantar a imagem no minikube, enfrentamos problemas ao tentar estabelecer a conexão da porta do socket, dentro do Docker, com as engines. Infelizmente, isso nos impediu de prosseguir com as demais etapas do trabalho.

3 Requisito de performance

3.1 OpenMP/MPI

Tomando como base o código do Jogo da Vida, primeiro o executamos como fornecido, sem paralelização com OpenMP ou MPI. Obtivemos os seguintes resultados de performance:

Listing 3.1: Log Jogo da Vida original

```
1          **Ok, RESULTADO CORRETO** tam=8;
2          tempos: init=0.0000141, comp=0.0000451, fim=0.0000198, tot
               =0.0000789
3          **Ok, RESULTADO CORRETO** tam=16;
4          tempos: init=0.0000050, comp=0.0004020, fim=0.0000041, tot
               =0.0004110
5          **Ok, RESULTADO CORRETO** tam=32;
6          tempos: init=0.0000110, comp=0.0035961, fim=0.0000172, tot
               =0.0036242
7          **Ok, RESULTADO CORRETO** tam=64;
8          tempos: init=0.0000479, comp=0.0297630, fim=0.0000410, tot
               =0.0298519
9          **Ok, RESULTADO CORRETO** tam=128;
10         tempos: init=0.0001879, comp=0.1201141, fim=0.0000439, tot
               =0.1203458
11         **Ok, RESULTADO CORRETO** tam=256;
12         tempos: init=0.0003171, comp=0.8231528, fim=0.0001521, tot
               =0.8236220
13         **Ok, RESULTADO CORRETO** tam=512;
14         tempos: init=0.0012391, comp=6.2695191, fim=0.0005629, tot
               =6.2713211
15         **Ok, RESULTADO CORRETO** tam=1024;
16         tempos: init=0.0049760, comp=51.7422149, fim=0.0022430, tot
               =51.7494340
```

Em seguida, começamos adicionando um pragma OpenMP, como é mostrado a seguir, acima do for da linha 18 do código do Jogo da Vida para paralelizar os dois laços seguintes. As variáveis *i*, *j* e *vizviv* foram privadas e as variáveis *tabulIn* e *tabulOut* foram tratadas como globais.

Listing 3.2: Alterações de OpenMP

```
1 #pragma omp parallel for private(i, j, vizviv) shared(tabulIn,
      tabulOut)
```

Executamos o código na máquina cm1 da Chococino, com 12 threads e o resultado está no Log a seguir, onde é possível notar que já houve melhora no desempenho:

Listing 3.3: Log Jogo da Vida + OpenMP

```
**Ok, RESULTADO CORRETO** tam=8;
tempos: init=0.0000451, comp=0.0000858, fim=0.0000551, tot
    =0.0001860
**Ok, RESULTADO CORRETO** tam=16;
tempos: init=0.0000110, comp=0.0007579, fim=0.0000110, tot
    =0.0007799
**Ok, RESULTADO CORRETO** tam=32;
tempos: init=0.0000191, comp=0.0036299, fim=0.0000031, tot
    =0.0036521
**Ok, RESULTADO CORRETO** tam=64;
tempos: init=0.0000150, comp=0.0099549, fim=0.0000100, tot
    =0.0099800
**Ok, RESULTADO CORRETO** tam=128;
tempos: init=0.0001199, comp=0.0808120, fim=0.0000322, tot
    =0.0809641
**Ok, RESULTADO CORRETO** tam=256;
tempos: init=0.0003560, comp=0.6609581, fim=0.0001211, tot
    =0.6614351
**Ok, RESULTADO CORRETO** tam=512;
tempos: init=0.0011411, comp=5.3329060, fim=0.0006120, tot
    =5.3346591
**Ok, RESULTADO CORRETO** tam=1024;
tempos: init=0.0046771, comp=43.7467699, fim=0.0018570, tot
    =43.7533040
```

Depois, adicionamos ao código já modificado com OpenMP, a parte de MPI. Para isso alteramos a função *main*. Fizemos com que um Mestre distribua as tarefas entre os Escravos de acordo com o número de processos.

Listing 3.4: Alterações de MPI

```
int main(int argc, char *argv[])
{
    int pow = 0, i, tam, *tabulIn, *tabulOut, numtasks, taskid;
    char msg[9];
    double t0, t1, t2, t3;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    // np 4
    if (taskid == 0)
```

```
11      {
12          for (pow = POWMIN; pow <= POWMAX; pow++)
13          {
14              int proc = (pow - POWMIN) % numtasks;
15                          if (proc == 0) proc++;
16              MPI_Send(&pow, 1, MPI_INT, proc, 0, MPI_COMM_WORLD);
17          }
18
19          for (int i = 1; i < numtasks; i++)
20          {
21              int pow = -1;
22              MPI_Send(&pow, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
23          }
24      }
25      else
26      {
27          while (pow != -1)
28          {
29              MPI_Recv(&pow, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
30
31              if (pow == -1)
32                  break;
33
34              tam = 1 << pow;
35              // aloca e inicializa tabuleiros
36              t0 = wall_time();
37              tabulIn = (int *)malloc((tam + 2) * (tam + 2) * sizeof(
                    int));
38              tabulOut = (int *)malloc((tam + 2) * (tam + 2) * sizeof(
                    int));
39              InitTabul(tabulIn, tabulOut, tam);
40              t1 = wall_time();
41              for (i = 0; i < 2 * (tam - 3); i++)
42              {
43                  UmaVida(tabulIn, tabulOut, tam);
44                  UmaVida(tabulOut, tabulIn, tam);
45              } /* fim-for */
46              t2 = wall_time();
47              if (Correto(tabulIn, tam))
48                  printf("**Ok, RESULTADO CORRETO**\n");
49              else
50                  printf("**Nok, RESULTADO ERRADO**\n");
51              t3 = wall_time();
52              printf("tam=%d; tempos: init=%7.7f, comp=%7.7f, fim=%7.7
```

```
                f, tot=%7.7f \n",
53                  tam, t1 - t0, t2 - t1, t3 - t2, t3 - t0);
54              free(tabulIn);
55              free(tabulOut);
56          }
57      }
58
59      MPI_Finalize();
60      return 0;
61  }
```

Ao executar o código, obtivemos o resultado a seguir. Nesse caso houve uma queda no desempenho.

Listing 3.5: Log Jogo da Vida + OpenMP + MPI

```
1           **Ok, RESULTADO CORRETO**
2           tam=8; tempos: init=0.0000050, comp=0.4391639, fim
                =0.0039990, tot=0.4431679
3           **Ok, RESULTADO CORRETO**
4           tam=16; tempos: init=0.0000050, comp=1.2719800, fim
                =0.0000520, tot=1.2720370
5           **Ok, RESULTADO CORRETO**
6           tam=32; tempos: init=0.0000319, comp=2.6392059, fim
                =0.0000122, tot=2.6392500
7           **Ok, RESULTADO CORRETO**
8           tam=64; tempos: init=0.0000720, comp=5.8830879, fim
                =0.0001140, tot=5.8832738
9           **Ok, RESULTADO CORRETO**
10          tam=128; tempos: init=0.0000870, comp=11.9198620, fim
                =0.0000968, tot=11.9200459
11          **Ok, RESULTADO CORRETO**
12          tam=256; tempos: init=0.0003691, comp=24.4154499, fim
                =0.0001731, tot=24.4159920
13          **Ok, RESULTADO CORRETO**
14          tam=512; tempos: init=0.0012319, comp=45.4470401, fim
                =0.0004740, tot=45.4487460
15          **Ok, RESULTADO CORRETO**
16          tam=1024; tempos: init=0.0128441, comp=51.8523459, fim
                =0.0019290, tot=51.8671191
```

Por fim, para otimizar a performance do MPI (junto do OpenMP), o dividimos em quatro hosts no momento da execução e então, o resultado foi melhor do que todas as versões anteriores. O log está a seguir:

Listing 3.6: Log Jogo da Vida + OpenMP + MPI com hosts

```
        **Ok, RESULTADO CORRETO**
        tam=8; tempos: init=0.0000088, comp=0.0010130, fim
           =0.0002611, tot=0.0012829
        **Ok, RESULTADO CORRETO**
        tam=16; tempos: init=0.0000129, comp=0.0006340, fim
           =0.0001440, tot=0.0007908
        **Ok, RESULTADO CORRETO**
        tam=32; tempos: init=0.0000200, comp=0.0016940, fim
           =0.0001290, tot=0.0018430
        **Ok, RESULTADO CORRETO**
        tam=64; tempos: init=0.0000451, comp=0.0045581, fim
           =0.0001109, tot=0.0047140
        **Ok, RESULTADO CORRETO**
        tam=128; tempos: init=0.0000761, comp=0.0154891, fim
           =0.0000660, tot=0.0156312
        **Ok, RESULTADO CORRETO**
        tam=256; tempos: init=0.0003111, comp=0.1179428, fim
           =0.0001571, tot=0.1184111
        **Ok, RESULTADO CORRETO**
        tam=512; tempos: init=0.0012279, comp=0.9280231, fim
           =0.0004671, tot=0.9297180
        **Ok, RESULTADO CORRETO**
        tam=1024; tempos: init=0.0054169, comp=7.4560680, fim
           =0.0019190, tot=7.4634039
```

3.1.1 Instrução de execução

Para executar o algoritmo a fim de usar o MPI e o OpenMP com múltiplos hosts, devem ser executados os seguintes comandos:

```
    $ mpicc -o jogodavida jogodavida_mpi_openmp.c -fopenmp
    $ mpirun jogodavida -n 4 -host cm1,cm2,cm3,cm4
```

Os hosts podem ser alterados de acordo com a necessidade e a localização da aplicação, nesse caso foram usadas todas as máquinas “cm” da chococino. Para rodar sem os hosts basta retirar a parte “-host cm1,cm2,cm3,cm4” e para rodar sem o OpenMP basta retirar a flag “-fopenmp”.

3.2 Apache Spark

Foi implementada uma versão do algoritmo usando Apache Spark por meio da biblioteca pyspark do python, nesse caso, houve uma queda enorme no desempenho devido ao constante uso da instrução “collect” do Spark que pode demorar muito tempo.

Listing 3.7: Log Jogo da Vida + Spark

```
        tam=8 **Ok, RESULTADO CORRETO**
```

```
        time=8.283813238143921

        tam=16 **Ok, RESULTADO CORRETO**
        time=8.375973463058472

        tam=32 **Ok, RESULTADO CORRETO**
        time=18.700819969177246

        tam=64 **Ok, RESULTADO CORRETO**
        time=40.72215127944946

        tam=128 **Ok, RESULTADO CORRETO**
        time=86.22166776657104

        tam=256 **Ok, RESULTADO CORRETO**
        time=234.15412831306458

        tam=512 **Ok, RESULTADO CORRETO**
        time=954.4769561290741

        tam=1024 **Ok, RESULTADO CORRETO**
        time=5257.697530269623
```

3.2.1 Instrução de execução

Para executar o algoritmo em python, é necessário abrir o arquivo "jogodavida_spark.ipynb" dentro de um jupyter notebook que contenha um ambiente pyspark, nesse caso, também foi usada a maquina "cm1" da Chococino para executar o jupyter. O notebook pode ser aberto pelo comando:

```
$ python3 -m notebook
```

Para abrir o notebook em uma máquina local, foi necessário fazer a configuração da extensão "FoxyProxy" do navegador "Firefox". Com o notebook aberto, basta rodar as células em ordem que o código deve funcionar.

4 Requisito de elasticidade

Para atender à necessidade de escalabilidade e flexibilidade, optamos por utilizar o Kubernetes, um orquestrador de contêineres de código aberto (https://kubernetes.io/). Esse orquestrador é responsável por gerenciar um servidor de socket, que por sua vez recebe as requisições. Ao receber essas requisições, o servidor executa os algoritmos desenvolvidos em Spark e OpenMP/MPI, buscando fornecer os resultados com máxima eficiência e agilidade.

4.1 Kubernetes

Kubernetes, ou K8s, é um sistema de código aberto projetado para automatizar o deploy, escalonamento e gerenciamento de aplicações em contêineres. Para a implementação deste projeto, utilizamos o programa "minikube" (https://minikube.sigs.k8s.io/), uma ferramenta que possibilita a utilização do Kubernetes em uma máquina local, eliminando a necessidade de depender de serviços em nuvem ou de outras máquinas externas.

4.2 Aplicação

Inicialmente, a proposta da aplicação do Kubernetes era criar um programa servidor capaz de receber conexões via socket, a fim de gerenciar a execução dos algoritmos do jogo da vida. Para isso, desenvolvemos um algoritmo em C que atuaria como esse servidor, e em seguida, criamos uma imagem Docker para executá-lo.

Aqui está o Dockerfile utilizado para criar a imagem:

Listing 4.1: Dockerfile

```
FROM gcc:latest
WORKDIR /app
# Copiar o c digo fonte do Socket Server para o cont iner
COPY socket_server.c /app/
# Compilar o c digo fonte
RUN gcc -o socket_server socket_server.c
# Definir o comando de inicializa  o do cont iner (executar o
    Socket Server)
CMD ["./socket_server"]
```

Para executar a imagem Docker no minikube, criamos um arquivo YAML que descreve a configuração do serviço "Job" no Kubernetes:

Listing 4.2: socket-server.yml

```
    apiVersion: batch/v1
    kind: Job
    metadata:
        name: socket-server
    spec:
```

```
    template:
    metadata:
        name: socket-server-pod
    spec:
        containers:
        - name: socket-server
          image: docker.io/trab/socket-server:latest
          imagePullPolicy: Never
          ports:
          - containerPort: 8080
        restartPolicy: OnFailure
```

O deploy da imagem no minikube pode ser realizado usando o seguinte comando:

```
kubectl create -f socket-server.yml
```

No entanto, mesmo com o Kubernetes e a imagem funcionando corretamente, encontramos um obstáculo ao tentar conectar à porta 8080 do pod da aplicação a partir de uma máquina externa. Essa dificuldade na conexão inviabilizou a continuação do experimento, pois a elasticidade do sistema dependia do servidor socket funcionando corretamente para executar os algoritmos e enviar os dados para o Kibana/ElasticSearch. Infelizmente, a falha na conexão impossibilitou o prosseguimento do projeto.

5 Análise dos resultados

Para realizar a análise dos resultados obtidos, comparamos o desempenho dos algoritmos em ambientes idênticos. Para isso, executamos todos os algoritmos em máquinas do tipo "cm" (cm1, cm2, cm3 e cm4) dentro do cluster "chococino". Essas máquinas possuem 12 núcleos de processamento e 16GB de memória RAM.

A seguir, apresentamos duas tabelas. A primeira tabela atribui uma sigla a cada algoritmo, enquanto a segunda tabela demonstra o tempo de execução (em segundos) de cada algoritmo para um tamanho específico de população.

Tabela 5.1: Tabela de referência para a sigla dos algoritmos

Sigla	**Algoritmo**
Seq	Sequencial em C
OMP	Usando apenas OpenMP
OMP/MPI 1h	Usando OpenMP + MPI em apenas 1 host
OMP/MPI 4h	Usando OpenMP + MPI em 4 hosts diferentes
Spark	Usando PySpark

Tabela 5.2: Tabela de tempo de execução (s) para cada algoritmo

Tamanho	**Seq**	**OMP**	**OMP/MPI 1h**	**OMP/MPI 4h**	**Spark**
8	0.00007	0.00018	0.44316	0.00128	8.2838
16	0.00041	0.00077	1.27203	0.00079	8.3759
32	0.00362	0.00365	2.63925	0.00184	18.70081
64	0.02985	0.00998	5.88327	0.00471	40.72215
128	0.12034	0.08096	11.92	0.01563	86.22166
256	0.82362	0.66143	24.416	0.11841	234.15412
512	6.27132	5.33465	45.44874	0.92971	954.47695
1024	51.74943	43.7533	51.86711	7.4634	5257.69753

Tabela 5.3: Caption

Com base nos resultados apresentados acima, podemos fazer algumas análises e conclusões relevantes:

- Ao utilizar apenas OpenMP, já é possível observar uma considerável otimização nos tempos de execução para tamanhos maiores de população.
- Ao combinar OpenMP e MPI sem a utilização de hosts adicionais, nota-se uma perda de desempenho para quase todos os tamanhos de população. No entanto, essa diferença em relação à execução sequencial diminui drasticamente na população de tamanho 1024, onde ambos os algoritmos praticamente têm o mesmo tempo de execução.

- A utilização de OpenMP e MPI com múltiplos hosts (no caso, 4) resulta em um ganho significativo de desempenho em relação a qualquer outro algoritmo. Nesse cenário, o procedimento é finalizado em apenas 7 segundos para a população de tamanho 1024.
- Por outro lado, a utilização do Spark mostrou uma piora considerável no desempenho em relação à execução sequencial, tornando o algoritmo praticamente inutilizável. Esse resultado se deve às repetidas chamadas do comando "collect()" no PySpark durante a execução do algoritmo, uma vez que esse comando é muito custoso e demorado.

6 Conclusão

Em geral, o projeto foi bastante interessante para entender como integrar as tecnologias envolvidas, porém também bastante complexo. Na fase de Performance, enfrentamos algumas dificuldades com o Spark, pois não conseguimos melhorar seu desempenho; na verdade, ocorreu uma queda na performance. Além disso, o cluster Chococino não possuía o Kubernetes instalado, o que nos obrigou a encontrar uma solução para executar o projeto localmente utilizando o "minikube", conforme mencionado anteriormente.

Outra questão significativa foi a dificuldade em conectar corretamente os hosts e as portas de cada componente da arquitetura (engines), como o Spark ou MPI/OpenMP com o Socket Server. Infelizmente, esse problema impediu nosso progresso com o projeto a partir dessa etapa.