

UNIVERSIDADE DE BRASÍLIA  
Faculdade do Gama

Programação para Sistemas Paralelos e Distribuídos

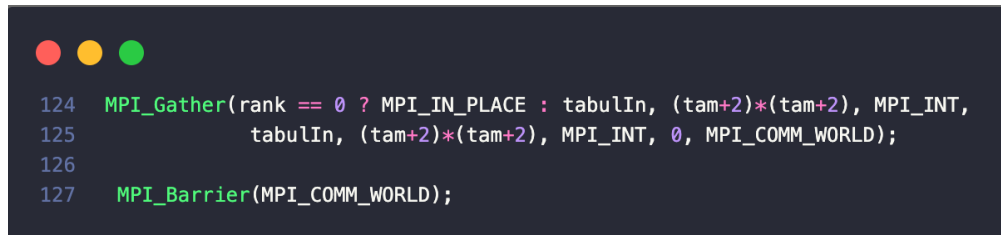
**Laboratório O jogo da vida**

Antônio Aldisio 20/2028211  
Fernando Miranda Calil 19/0106566  
Lorrany Oliveira Souza 18/0113992

Brasília, DF  
2023

## 1. MPI

Na figura 01 temos as seguintes linhas de código está utilizando a função `MPI_Gather` da biblioteca MPI (Message Passing Interface) para coletar dados de cada processo em um único processo. A função `MPI_Gather` é uma operação coletiva, o que significa que envolve a comunicação entre vários processos em um programa paralelo.

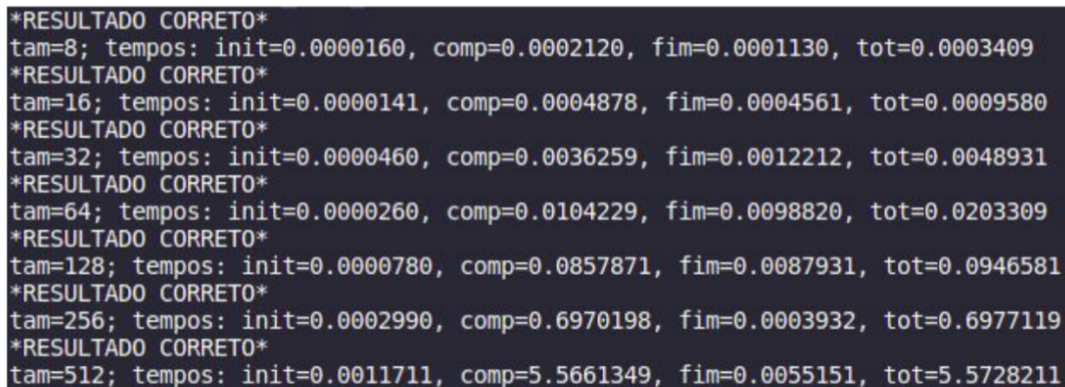


```
124 MPI_Gather(rank == 0 ? MPI_IN_PLACE : tabulIn, (tam+2)*(tam+2), MPI_INT,
125           tabulIn, (tam+2)*(tam+2), MPI_INT, 0, MPI_COMM_WORLD);
126
127 MPI_Barrier(MPI_COMM_WORLD);
```

Figura 01 - Trecho do código MPI

### 1.1. Resultado

Na figura 02 temos o resultado obtido na GPU1.



```
*RESULTADO CORRETO*
tam=8; tempos: init=0.0000160, comp=0.0002120, fim=0.0001130, tot=0.0003409
*RESULTADO CORRETO*
tam=16; tempos: init=0.0000141, comp=0.0004878, fim=0.0004561, tot=0.0009580
*RESULTADO CORRETO*
tam=32; tempos: init=0.0000460, comp=0.0036259, fim=0.0012212, tot=0.0048931
*RESULTADO CORRETO*
tam=64; tempos: init=0.0000260, comp=0.0104229, fim=0.0098820, tot=0.0203309
*RESULTADO CORRETO*
tam=128; tempos: init=0.0000780, comp=0.0857871, fim=0.0087931, tot=0.0946581
*RESULTADO CORRETO*
tam=256; tempos: init=0.0002990, comp=0.6970198, fim=0.0003932, tot=0.6977119
*RESULTADO CORRETO*
tam=512; tempos: init=0.0011711, comp=5.5661349, fim=0.0055151, tot=5.5728211
```

Figura 02 - Resultado de MPI com GPU1

## 2. OMP

Na figura 03 temos a linha de código `"#pragma omp parallel for private(i, j, vizviv) shared(tabulIn, tabulOut)"` é uma diretiva de compilação para paralelizar um loop "for" usando o OpenMP. Ela permite que o loop seja executado em paralelo por várias threads. A cláusula "private" indica quais variáveis serão privadas para cada thread, evitando condições de corrida. A cláusula "shared" indica quais variáveis serão compartilhadas entre todas as threads. Essa abordagem pode melhorar o desempenho ao distribuir as

iterações do loop entre as threads, mas é importante garantir o acesso correto às variáveis compartilhadas para evitar problemas de concorrência.

```
34 #pragma omp parallel for private(i, j, vizviv) shared(tabulIn, tabulOut)
35 for (i=1; i<=tam; i++) {
36     for (j=1; j<=tam; j++) {
37         vizviv = tabulIn[ind2d(i-1,j-1)] + tabulIn[ind2d(i-1,j)] +
38             tabulIn[ind2d(i-1,j+1)] + tabulIn[ind2d(i,j-1)] +
39             tabulIn[ind2d(i,j+1)] + tabulIn[ind2d(i+1,j-1)] +
40             tabulIn[ind2d(i+1,j)] + tabulIn[ind2d(i+1,j+1)];
41         if (tabulIn[ind2d(i,j)] && vizviv < 2)
42             tabulOut[ind2d(i,j)] = 0;
43         else if (tabulIn[ind2d(i,j)] && vizviv > 3)
44             tabulOut[ind2d(i,j)] = 0;
45         else if (!tabulIn[ind2d(i,j)] && vizviv == 3)
46             tabulOut[ind2d(i,j)] = 1;
47         else
48             tabulOut[ind2d(i,j)] = tabulIn[ind2d(i,j)];
49     } /* fim-for */
50 } /* fim-for */
```

Figura 03 - Trecho do código OpenMP

## 1.2 Resultado

Na figura 04 temos os resultado na gpu2

```
**RESULTADO CORRETO**
tam=8; tempos: init=0.0000069, comp=0.0009410, fim=0.0000479, tot=0.0009959
**RESULTADO CORRETO**
tam=16; tempos: init=0.0000110, comp=0.0006831, fim=0.0000100, tot=0.0007041
**RESULTADO CORRETO**
tam=32; tempos: init=0.0000179, comp=0.0017571, fim=0.0000038, tot=0.0017788
**RESULTADO CORRETO**
tam=64; tempos: init=0.0000160, comp=0.0024400, fim=0.0000110, tot=0.0024669
**RESULTADO CORRETO**
tam=128; tempos: init=0.0000682, comp=0.0162148, fim=0.0000331, tot=0.0163162
**RESULTADO CORRETO**
tam=256; tempos: init=0.0003071, comp=0.1227109, fim=0.0001190, tot=0.1231370
**RESULTADO CORRETO**
tam=512; tempos: init=0.0012040, comp=0.9615560, fim=0.0004630, tot=0.9632230
**RESULTADO CORRETO**
tam=1024; tempos: init=0.0048752, comp=7.7059960, fim=0.0018330, tot=7.7127042
```

Figura 04 - Resultado do código OpenMP

### 3. CUDA

Na figura 05 temos o trecho de código contém um loop "for" que chama duas vezes a função "UmaVida", utilizando a sintaxe do CUDA, para processar matrizes na GPU. A função "UmaVida" é chamada alternando entre os buffers d\_tabulIn e d\_tabulOut, possivelmente para aplicar um algoritmo iterativo. A função "cudaDeviceSynchronize()" é usada para garantir a sincronização entre as chamadas de kernel. Esse código possibilita o processamento paralelo e eficiente dos dados na GPU.

```
113 for (i=0; i<2*(tam-3); i++) {
114     UmaVida<<<numBlocks, threadsPerBlock>>>(d_tabulIn, d_tabulOut, tam);
115     cudaDeviceSynchronize();
116     UmaVida<<<numBlocks, threadsPerBlock>>>(d_tabulOut, d_tabulIn, tam);
117     cudaDeviceSynchronize();
118 }
```

Figura 05 - Trecho do código CUDA

Na figura 06 temos a função \_\_global\_\_ que será invocada a partir da CPU e será executada pela GPU.

```
31 __global__ void UmaVida(int* tabulIn, int* tabulOut, int tam) {
32     int i = blockIdx.x * blockDim.x + threadIdx.x + 1;
33     int j = blockIdx.y * blockDim.y + threadIdx.y + 1;
34     int vizviv;
35
36     if (i <= tam && j <= tam) {
37         vizviv = tabulIn[ind2d(i-1,j-1)] + tabulIn[ind2d(i-1,j)] +
38         tabulIn[ind2d(i-1,j+1)] + tabulIn[ind2d(i,j-1)] +
39         tabulIn[ind2d(i,j+1)] + tabulIn[ind2d(i+1,j-1)] +
40         tabulIn[ind2d(i+1,j)] + tabulIn[ind2d(i+1,j+1)];
41         if (tabulIn[ind2d(i,j)] && vizviv < 2)
42             tabulOut[ind2d(i,j)] = 0;
43         else if (tabulIn[ind2d(i,j)] && vizviv > 3)
44             tabulOut[ind2d(i,j)] = 0;
45         else if (!tabulIn[ind2d(i,j)] && vizviv == 3)
46             tabulOut[ind2d(i,j)] = 1;
47         else
48             tabulOut[ind2d(i,j)] = tabulIn[ind2d(i,j)];
49     }
50 }
```

Figura 06 - Trecho do código que é paralelizado

## 2.3 Resultados

Na figura 07 temos o resultado da gpu2

```

**RESULTADO CORRETO**
tam=8; tempos: init=0.0000010, comp=0.1340041, fim=0.0000000, tot=0.1340051
**RESULTADO CORRETO**
tam=16; tempos: init=0.0000010, comp=0.0003600, fim=0.0000000, tot=0.0003610
**RESULTADO CORRETO**
tam=32; tempos: init=0.0000081, comp=0.0007350, fim=0.0000000, tot=0.0007432
**RESULTADO CORRETO**
tam=64; tempos: init=0.0000219, comp=0.0014491, fim=0.0000000, tot=0.0014710
**RESULTADO CORRETO**
tam=128; tempos: init=0.0000820, comp=0.0032001, fim=0.0000000, tot=0.0032821
**RESULTADO CORRETO**
tam=256; tempos: init=0.0002751, comp=0.0083330, fim=0.0000000, tot=0.0086081
**RESULTADO CORRETO**
tam=512; tempos: init=0.0009780, comp=0.0338831, fim=0.0000000, tot=0.0348611
**RESULTADO CORRETO**
tam=1024; tempos: init=0.0039670, comp=0.2199140, fim=0.0000000, tot=0.2238810

```

Figura 07 - Resultado do código CUDA

## 4. OpenCL

Na figura 08 temos o código é um kernel CUDA chamado "UmaVida", que é executado em paralelo na GPU. Ele recebe duas matrizes, tabulIn e tabulOut, representadas como ponteiros globais, e um tamanho tam. O kernel calcula a próxima iteração de um jogo da vida, onde cada célula na matriz de entrada (tabulIn) é atualizada com base nas células vizinhas. As células são verificadas dentro dos limites ( $i \leq \text{tam}$  e  $j \leq \text{tam}$ ) e o número de vizinhos vivos é calculado. Em seguida, as regras do jogo da vida são aplicadas para determinar o estado atualizado da célula na matriz de saída (tabulOut).

```

1  __kernel void UmaVida(__global int* tabulIn, __global int* tabulOut, const int tam) {
2      size_t i = get_global_id(0) + 1;
3      size_t j = get_global_id(1) + 1;
4      int vizviv;
5
6      if (i <= tam && j <= tam) {
7          vizviv = tabulIn[ind2d(i-1,j-1)] + tabulIn[ind2d(i-1,j)] +
8                  tabulIn[ind2d(i-1,j+1)] + tabulIn[ind2d(i,j-1)] +
9                  tabulIn[ind2d(i,j+1)] + tabulIn[ind2d(i+1,j-1)] +
10                 tabulIn[ind2d(i+1,j)] + tabulIn[ind2d(i+1,j+1)];
11
12         if (tabulIn[ind2d(i,j)] && vizviv < 2)
13             tabulOut[ind2d(i,j)] = 0;
14         else if (tabulIn[ind2d(i,j)] && vizviv > 3)
15             tabulOut[ind2d(i,j)] = 0;
16         else if (!tabulIn[ind2d(i,j)] && vizviv == 3)
17             tabulOut[ind2d(i,j)] = 1;
18         else
19             tabulOut[ind2d(i,j)] = tabulIn[ind2d(i,j)];
20     }
21 }
22

```

Figura 08 - Kernel utilizado pelo OPENCL

### 3.4 Resultado

Na figura 09 temos o resultado obtido na gpu2

```

**RESULTADO CORRETO**
tam=8; tempos: init=0.0000141, comp=0.1743190, fim=0.0000179, tot=0.1743510
**RESULTADO CORRETO**
tam=16; tempos: init=0.0000081, comp=0.1093230, fim=0.0000129, tot=0.1093440
**RESULTADO CORRETO**
tam=32; tempos: init=0.0000131, comp=0.0962379, fim=0.0000122, tot=0.0962632
**RESULTADO CORRETO**
tam=64; tempos: init=0.0000250, comp=0.0968099, fim=0.0000200, tot=0.0968549
**RESULTADO CORRETO**
tam=128; tempos: init=0.0000589, comp=0.0976110, fim=0.0000482, tot=0.0977180
**RESULTADO CORRETO**
tam=256; tempos: init=0.0001981, comp=0.0947869, fim=0.0001452, tot=0.0951302
**RESULTADO CORRETO**
tam=512; tempos: init=0.0007629, comp=0.0985060, fim=0.0005910, tot=0.0998600
**RESULTADO CORRETO**
tam=1024; tempos: init=0.0035279, comp=0.1132841, fim=0.0023069, tot=0.1191189

```

Figura 09 - Fractal do código OpenCL

#### 4. Comparação

Tabela tempo(s) para o tamanho de 8

Host	Sem pareamento	MPI	OMP	CUDA	OPENCL
cm1 (Intel i7-8700)	<b>0.0001590</b>	<b>0.0001819</b>	<b>0.0005181</b>	-	-
pos1(Intel i7-9700)	<b>0.0001490</b>	<b>0.0002990</b>	<b>0.0003982</b>	-	-
gpu1(Ryzen 7 2700 + RTX 3060)	<b>0.0000892</b>	<b>0.0001860</b>	<b>0.0003421</b>	<b>0.1447031</b>	<b>0.1763320</b>
gpu2(Ryzen 7 2700 + GTX 1650)	-	-	-	<b>0.1477170</b>	<b>0.2611122</b>

Tabela tempo(s) para o tamanho de 16

Host	Sem pareamento	MPI	OMP	CUDA	OPENCL
cm1 (Intel i7-8700)	<b>0.0006950</b>	<b>0.0001819</b>	<b>0.0006158</b>	-	-
pos1(Intel i7-9700)	<b>0.0008008</b>	<b>0.0008800</b>	<b>0.0004089</b>	-	-
gpu1(Ryzen 7 2700 + RTX 3060)	<b>0.0004458</b>	<b>0.0007401</b>	<b>0.0002871</b>	<b>0.0003910</b>	<b>0.1078110</b>
gpu2(Ryzen 7 2700 + GTX 1650)	-	-	-	<b>0.0003569</b>	<b>0.0819550</b>

**Tabela tempo(s) para o tamanho de 32**

<b>Host</b>	<b>Sem pareamento</b>	<b>MPI</b>	<b>OMP</b>	<b>CUDA</b>	<b>OPENCL</b>
cm1 (Intel i7-8700)	<b>0.0037310</b>	<b>0.0001819</b>	<b>0.0021770</b>	-	-
pos1(Intel i7-9700)	<b>0.0039358</b>	<b>0.0045660</b>	<b>0.0021610</b>	-	-
gpu1(Ryzen 7 2700 + RTX 3060)	<b>0.0036209</b>	<b>0.0065482</b>	<b>0.0013940</b>	<b>0.0007749</b>	<b>0.0934620</b>
gpu2(Ryzen 7 2700 + GTX 1650)	-			<b>0.0007172</b>	<b>0.0929210</b>

**Tabela tempo(s) para o tamanho de 64**

<b>Host</b>	<b>Sem pareamento</b>	<b>MPI</b>	<b>OMP</b>	<b>CUDA</b>	<b>OPENCL</b>
cm1 (Intel i7-8700)	<b>0.0114081</b>	<b>0.0001819</b>	<b>0.0029740</b>	-	-
pos1(Intel i7-9700)	<b>0.0098040</b>	<b>0.0101750</b>	<b>0.0036771</b>	-	-
gpu1(Ryzen 7 2700 + RTX 3060)	<b>0.0292430</b>	<b>0.0367610</b>	<b>0.0086570</b>	<b>0.0015609</b>	<b>0.0971930</b>
gpu2(Ryzen 7 2700 + GTX 1650)	-			<b>0.0016000</b>	<b>0.0929210</b>



**Tabela tempo(s) para o tamanho de 128**

<b>Host</b>	<b>Sem pareamento</b>	<b>MPI</b>	<b>OMP</b>	<b>CUDA</b>	<b>OPENCL</b>
cm1 (Intel i7-8700)	<b>0.0848541</b>	<b>0.0001819</b>	<b>0.0222540</b>		
pos1(Intel i7-9700)	<b>0.0796161</b>	<b>0.0825529</b>	<b>0.0212641</b>	-	-
gpu1(Ryzen 7 2700 + RTX 3060)	<b>0.0982659</b>	<b>0.1236541</b>	<b>0.0484979</b>	<b>0.0034189</b>	<b>0.0972788</b>
gpu2(Ryzen 7 2700 + GTX 1650)	-			<b>0.0043209</b>	<b>0.0815048</b>

**Tabela tempo(s) para o tamanho de 256**

<b>Host</b>	<b>Sem pareamento</b>	<b>MPI</b>	<b>OMP</b>	<b>CUDA</b>	<b>OPENCL</b>
cm1 (Intel i7-8700)	<b>0.6962080</b>	<b>0.0001819</b>	<b>0.1774480</b>		
pos1(Intel i7-9700)	<b>0.6514828</b>	<b>0.6642358</b>	<b>0.1698730</b>	-	-
gpu1(Ryzen 7 2700 + RTX 3060)	<b>0.7251849</b>	<b>0.7831461</b>	<b>0.2359381</b>	<b>0.0088100</b>	<b>0.0932159</b>
gpu2(Ryzen 7 2700 + GTX 1650)	-			<b>0.0180919</b>	<b>0.0814011</b>

**Tabela tempo(s) para o tamanho de 512**

<b>Host</b>	<b>Sem pareamento</b>	<b>MPI</b>	<b>OMP</b>	<b>CUDA</b>	<b>OPENCL</b>
cm1 (Intel i7-8700)	<b>5.4734631</b>	<b>0.0001819</b>	<b>1.4199839</b>	-	-
pos1(Intel i7-9700)	<b>5.2480640</b>	<b>5.3370869</b>	<b>1.3586020</b>	-	-
gpu1(Ryzen 7 2700 + RTX 3060)	<b>6.6382332</b>	<b>7.3078589</b>	<b>1.7391150</b>	<b>0.0354180</b>	<b>0.0983570</b>
gpu2(Ryzen 7 2700 + GTX 1650)	-	-	-	<b>0.1164341</b>	<b>0.0820150</b>

**Tabela tempo(s) para o tamanho de 1024**

<b>Host</b>	<b>Sem pareamento</b>	<b>MPI</b>	<b>OMP</b>	<b>CUDA</b>	<b>OPENCL</b>
cm1 (Intel i7-8700)	<b>45.9842451</b>	<b>0.0001819</b>	<b>11.7785630</b>	-	-
pos1(Intel i7-9700)	<b>43.0631351</b>	<b>44.6066740</b>	<b>11.1561251</b>	-	-
gpu1(Ryzen 7 2700 + RTX 3060)	<b>53.9471290</b>	<b>58.9955010</b>	<b>14.8778419</b>	<b>0.2230780</b>	<b>0.1153791</b>
gpu2(Ryzen 7 2700 + GTX 1650)	-	-	-	<b>0.7246370</b>	<b>0.0875449</b>

## 5. Conclusão

Analisando as tabelas de tempos para diferentes tamanhos de problemas e diferentes configurações de hardware e tecnologias de programação, podemos observar algumas informações interessantes.

Observando as tabelas, podemos fazer as seguintes análises:

1. Comparando os tempos nas colunas "Sem pareamento", "MPI", "OMP", "CUDA" e "OPENCL" nos diferentes tamanhos de problema, podemos ver que, em geral, o uso de paralelismo (MPI, OMP, CUDA e OPENCL) leva a uma redução significativa nos tempos em comparação com a execução sequencial (Sem pareamento).
2. Comparando os tempos nas colunas "Sem pareamento", "MPI", "OMP", "CUDA" e "OPENCL" para um mesmo tamanho de problema em diferentes configurações de hardware, podemos observar que os tempos podem variar significativamente dependendo do processador ou da placa de vídeo utilizados. Por exemplo, para o tamanho de 8, o tempo utilizando CUDA no "gpu1" é muito menor do que nos outros casos.
3. Para os tamanhos de problema menores (8 e 16), os tempos são bastante baixos em todas as configurações, o que indica que o paralelismo não traz benefícios significativos para esses casos. Entretanto, para tamanhos maiores, o uso de técnicas de paralelismo se mostra vantajoso, resultando em tempos menores.
4. Em geral, as configurações "gpu1" (Ryzen 7 2700 + RTX 3060) e "gpu2" (Ryzen 7 2700 + GTX 1650) apresentam tempos menores em comparação com as configurações "cm1" (Intel i7-8700) e "pos1" (Intel i7-9700). Isso ocorre porque as GPUs são projetadas para acelerar operações paralelas, o que é especialmente útil para problemas de grande escala.
5. A tecnologia CUDA mostra bons resultados em termos de desempenho, especialmente nas configurações "gpu1" e "gpu2". Isso está relacionado ao fato de que a CUDA é uma plataforma de computação paralela desenvolvida pela NVIDIA, otimizada para aproveitar a capacidade de processamento das GPUs NVIDIA.

Em resumo, a análise das tabelas mostra que o uso de técnicas de paralelismo, como MPI, OMP, CUDA e OPENCL, pode trazer benefícios significativos em termos de desempenho para problemas de grande escala. Além disso, o uso de GPUs pode resultar em tempos ainda menores devido à sua capacidade de processamento paralelo. No entanto, é importante considerar que a escolha da tecnologia e configuração de hardware adequadas depende do tipo de problema e dos recursos disponíveis.