Instituto Tecnológico de Aeronáutica

CE-265: Processamento Paralelo

Exercício 07: Paralelizar MPI o Jogo da Vida

Bruno Benjamim Bertucci - Turma 23.2

1 Introdução

A paralelização MPI (*Message Passing Interface*) é baseada em sistemas MIMD, ou seja, de memória distribuída. Portanto, constituí um paradigma bastante diferente de outros métodos de paralelização como OpenMP. Para demonstrar o uso de MPI, o programa que executa o Jogo da Vida, implementado na linguagem C, foi paralelizado usando MPI.

2 Implementação

A implementação foi feita da seguinte forma: na função `int main()`, foi feita a partição do domínio por linhas de acordo com a quantidade de processos criados. Disso resultaram duas variáveis que serão usadas nas próximas funções, que são o deslocamento de cada processo, ou seja, o índice no qual o domínio de um determinado processo se inicia, e o tamanho local, que é o número de linhas atribuído a um único processo. Além disso, o comando `MPI_Init()` foi adicionado ao início da função, conforme requisito da biblioteca usada.

Em seguida, a função `InitTabul()` foi levemente alterada para determinar quais elementos estarão vivos ou mortos usando índices globais, ou seja, os índices relativos ao tabuleiro completo.

Já a função `DumpTabul()` foi alterada significativamente, incluindo uma divisão entre os comandos executados pelo processo de ID zero e os demais processos. Em suma, quando o ID não é zero, o comando `MPI_Send()` é chamado para enviar ao processo de ID zero as linhas que estão contidas, em índices globais, no íntervalo entre os parâmetros *first* e *last*. Já o processo de ID zero escreve na tela as linhas pertencentes ao seu domínio, e recebe dos demais processos as outras linhas usando o comando `MPI_Recv()`. Vale notar que, pela dificuldade de se identificar de qual processo uma linha se originou, a opção `MPI_ANY_SOURCE` foi utilizada.

A função `UmaVida()` também foi alterada, principalmente para iterar somente sobre o domínio de um dado processo, e também para fazer a atualização das "bordas" dos domínios, denominadas *Ghost zones*, o que foi feito novamente usando comandos `MPI_Send()` e *MPI_Recv()*.

Finalmente, a função `Correto()` foi modificada para avaliar cada domínio segundo índices globais, mas principalmente para aplicar um comando de redução nos resultados de correção de cada processo, usando `MPI_Reduce()` junto com o operando `MPI_LAND`, de forma a garantir a correção de todo o tabuleiro ao final da execução. Logo antes de encerrar o programa, o comando `MPI_Finalize()` foi adicionado.

As funções citadas foram separadas em um arquivo *ModVida.c*, cujo código está no Anexo A.

3 Teste no SDumont

Para testar a eficácia da implementação feita, o programa foi executado no supercomputador SDumont, para um tamanho de tabuleiro de 2048 x 2048. Várias execuções foram feitas variando o número de processos entre 1, 2, 4, 8, 16, 24, 48, e 72. Os arquivos de saída para cada execução encontram-se no Anexo B.

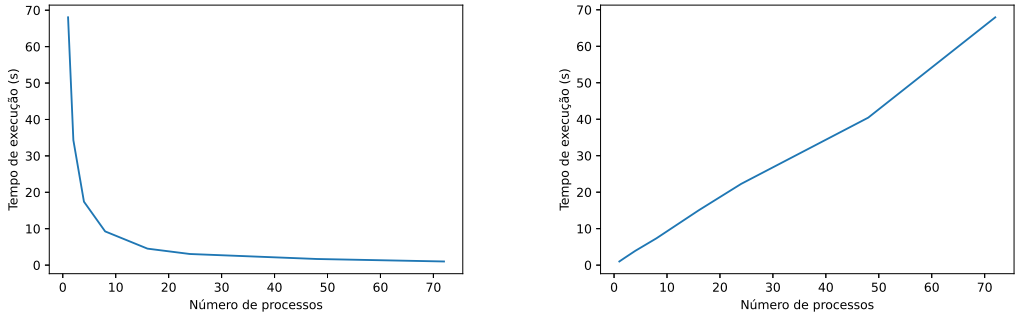
4 Tempos de execução

A partir dos arquivos de saída, os tempos de execução totais foram compilados na Tabela 1. Os *speed-ups* das execuções também foram calculados pela Equação 1, onde p é o número de processos, $T(1)$ é o tempo de execução com um único processo, e $T(p)$ é o tempo de execução para p processos.

$$S(p) = \frac{T(1)}{T(p)} \tag{1}$$

	Tempo (s)	Speed-up
1	68.048712	1.000000
2	34.324321	1.982522
4	17.421161	3.906095
8	9.263028	7.346271
16	4.525406	15.037040
24	3.055958	22.267555
48	1.683196	40.428276
72	1.001579	67.941432

Tabela 1: Resultados dos testes de execução.



(a) Tempos de execução.

(b) Valores de *speed-up*.

Figura 1: Gráficos dos resultados dos testes.

Observando a tabela acima e os gráficos da Figura 1, pode-se observar um ganho considerável no desempenho, principalmente porque os ganhos relativos de velocidade não diminuem com o aumento do número de processos, ao menos no leque de valores testados, o que indica uma escalabilidade boa da paralelização MPI. Além disso, os valores de *speed-up* tendem a ter valores semelhantes ao número de processos usado. Isso pode ser observado pelo fato de o gráfico de *speed-ups* ser próximo de uma reta. Curiosamente, entre 48 e 72 processos, os ganhos de velocidade aparentam aumentar a uma taxa maior do que o que foi observado entre os demais números testados.

Pode-se notar também que a implementação de paralelização MPI é de dificuldade elevada em comparação com outros métodos, necessitando de alterações mais profundas no algoritmo. Apesar disso, a capacidade demonstrada pelo MPI de melhorar o desempenho de um programa faz com que as dificuldades de implementação sejam proveitosas.

Anexo A – Código do arquivo ModVida.c

```
#include "ModVida.h"


// UmaVida: Executa uma iteracao do Jogo da Vida
//          em tabuleiros de tamanho tam. Produz o tabuleiro
//          de saida tabulOut a partir do tabuleiro de entrada
//          tabulIn. Os tabuleiros tem (tam, tam) celulas
//          internas vivas ou mortas. Tabuleiros sao orlados
//          por celulas eternamente mortas.
//          Tabuleiros sao dimensionados tam+2 x tam+2.



void UmaVida(int* restrict tabulIn, int* restrict tabulOut, const int tam,
  const int tamLocal,const int desloc, const int myId, const int numProc) {

  MPI_Status status;

  for (int i=1; i<=tamLocal; i++) {
    for (int j= 1; j<=tam; j++) {
      const int vizviv =
        tabulIn[ind2d(i-1,j-1)] +
        tabulIn[ind2d(i-1,j  )] +
        tabulIn[ind2d(i-1,j+1)] +
        tabulIn[ind2d(i  ,j-1)] +
        tabulIn[ind2d(i  ,j+1)] +
        tabulIn[ind2d(i+1,j-1)] +
        tabulIn[ind2d(i+1,j  )] +
        tabulIn[ind2d(i+1,j+1)];
      if (tabulIn[ind2d(i,j)] && vizviv < 2)
        tabulOut[ind2d(i,j)] = 0;
      else if (tabulIn[ind2d(i,j)] && vizviv > 3)
        tabulOut[ind2d(i,j)] = 0;
      else if (!tabulIn[ind2d(i,j)] && vizviv == 3)
        tabulOut[ind2d(i,j)] = 1;
      else
        tabulOut[ind2d(i,j)] = tabulIn[ind2d(i,j)];
    }
  }

  if(myId != numProc - 1)
    MPI_Send(&tabulOut[ind2d(tamLocal,0)], tam+2,MPI_INT,
    myId+1, tam*(desloc + tamLocal), MPLCOMM_WORLD);

  if(myId != 0)
    MPI_Send(&tabulOut[ind2d(1,0)], tam+2, MPI_INT,
    myId-1, tam*(desloc + 1), MPLCOMM_WORLD);

  if(myId != 0)
    MPI_Recv(&tabulOut[0], tam+2, MPI_INT,
    myId-1, tam*desloc, MPLCOMM_WORLD, &status);

  if(myId != numProc-1)
    MPI_Recv(&tabulOut[ind2d(tamLocal+1,0)], tam+2, MPI_INT,
    myId+1, tam*(desloc + tamLocal + 1), MPLCOMM_WORLD, &status);
```

```
}


// DumpTabul: Imprime trecho do tabuleiro entre
//            as posicoes (first,first) e (last,last)
//            X representa celula viva
//            . representa celula morta


void DumpTabul(int* restrict tabul, const int tam,
               const int first, const int last,
               char* restrict msg, int tamLocal, int desloc, int myId){

  int line[tam+2];
  MPI_Status status;
  int indi;

  if(myId == 0) {
    printf("%s: Dump posicoes [%d:%d, %d:%d] de tabuleiro %d x %d\n",
    msg, first, last, first, last, tam, tam);
    for (int i=first; i<=last; i++) printf("="); printf("=\n");

    for(int i=first; i<=last; ++i) {
      indi = ind2d(i, 0);
      if(i < desloc || i > desloc + tamLocal + 1) {
        MPI_Recv(&line, tam+2, MPI_INT,
        MPI_ANY_SOURCE, i, MPI_COMM_WORLD, &status);
        for (int j=1; j<=tam; j++)
          printf("%c", line[j]? 'X' : '.');
        printf("\n");
      } else {
        for (int ij=indi+first; ij<=indi+last; ij++)
          printf("%c", tabul[ij]? 'X' : '.');
        printf("\n");
      }
    }

    for (int i=first; i<=last; i++) printf("="); printf("=\n");

  } else {
    for(int i=1; i<=tamLocal; ++i) {
      if(i + desloc > first && i + desloc <= last) {
        MPI_Send(&tabul[ind2d(i,0)], tam+2,
        MPI_INT, 0, i + desloc, MPI_COMM_WORLD);
      }
    }
  }
}


// InitTabul: Inicializa dois tabuleiros:
//            tabulIn com um veleiro no canto superior esquerdo
//            tabulOut com celulas mortas
```

```
void InitTabul(int* restrict tabulIn, int* restrict tabulOut,
const int tamLocal , const int tam, const int desloc){

int vivos[5] = {ind2d(1,2), ind2d(2,3), ind2d(3,1),
    ind2d(3,2), ind2d(3,3)};
int estaVivo;

  for (int ij=0; ij<(tam+2)*(tamLocal+2); ij++) {
    estaVivo = 0;
    for(int k=0; k<5; ++k) {
      if(ij + desloc*(tam+2) == vivos[k]) {
        estaVivo = 1;
        break;
      }
    }

    if(estaVivo) {
      tabulIn[ij] = 1;
    } else {
      tabulIn[ij] = 0;
    }

    tabulOut[ij] = 0;
  }
}



// Correto: Verifica se a configuracao final do tabuleiro
//          estah correta ou nao (veleiro no canto inferior direito)



bool Correto(int* restrict tabul,
    const int tam, const int tamLocal, int desloc){

  bool resul = true;
  bool finalResul = true;
  int globLine = desloc;

  // percorre todas as linhas do tabuleiro

  for (int lin=0; lin<tamLocal+2; lin++) {

    // conta celulas vivas na linha

    int cnt=0;
    for (int col=0; col<tam+2; col++) {
      cnt += tabul[ind2d(lin,col)];
    }
    // confere a contagem para linhas totalmente mortas

    if (globLine < tam-2)
      resul &= (cnt == 0);
```

```
    // confere a contagem e a posicao na primeira linha do veleiro

    else if (globLine == tam-2)
      resul &= (cnt == 1) &
        (tabul[ind2d(lin,tam-1)] == 1);

    // confere a contagem e a posicao na segunda linha do veleiro

    else if (globLine == tam-1)
      resul &= (cnt == 1) &
        (tabul[ind2d(lin,tam)] == 1);

    // confere a contagem e a posicao na terceira linha do veleiro

    else if (globLine == tam)
      resul &= (cnt == 3) &
        (tabul[ind2d(lin,tam-2)] == 1) &
        (tabul[ind2d(lin,tam-1)] == 1) &
        (tabul[ind2d(lin,tam  )] == 1);

    // confere a contagem na borda inferior

    else if (globLine == tam+1)
      resul &= (cnt == 0);

    ++globLine;

  }
  MPI_Reduce(&resul, &finalResul, 1, MPI_C_BOOL,
    MPI_LAND, 0, MPI_COMM_WORLD);

  return(resul);
}
```

Anexo B – Arquivos de saída para os testes realizados

1 processo

```
**RESULTADO CORRETO**
numProc=1; tam=2048; tempos: init=0.013498, comp=68.033930, fim=0.001284, tot=68.048712
```

2 processos

```
**RESULTADO CORRETO**
numProc=2; tam=2048; tempos: init=0.007168, comp=34.316455, fim=0.000698, tot=34.324321
```

4 processos

```
**RESULTADO CORRETO**
numProc=4; tam=2048; tempos: init=0.003941, comp=17.416340, fim=0.000880, tot=17.421161
```

8 processos

```
**RESULTADO CORRETO**
numProc=8; tam=2048; tempos: init=0.002243, comp=9.260390, fim=0.000395, tot=9.263028
```

16 processos

```
**RESULTADO CORRETO**
numProc=16; tam=2048; tempos: init=0.001427, comp=4.523706, fim=0.000273, tot=4.525406
```

24 processos

```
**RESULTADO CORRETO**
numProc=24; tam=2048; tempos: init=0.000989, comp=3.054338, fim=0.000631, tot=3.055958
```

48 processos

```
**RESULTADO CORRETO**
numProc=48; tam=2048; tempos: init=0.000505, comp=1.623541, fim=0.059150, tot=1.683196
```

72 processos

```
**RESULTADO CORRETO**
numProc=72; tam=2048; tempos: init=0.000360, comp=0.948955, fim=0.052264, tot=1.001579
```