Desenvolvimento do Jogo da Vida de Conway utilizando MPI

Raphael Lopes Baldi
raphael.baldi@acad.pucrs.br

23 de junho de 2018

1 Introdução

Este trabalho utilizou o Jogo da Vida, desenvolvido pelo matemático britânico John Conway, como plataforma de estudo da biblioteca MPI (*Message Passing Interface*). O jogo não possui jogadores e avalia a evolução de células de acordo com uma série de regras, descritas a seguir.

O universo do jogo é definido por uma matriz bidimensional finita, sendo que cada célula da mesma descreve o estado de uma "entidade"do jogo. Tal estado é definido como um valor verdade: verdadeiro, se a célula estiver viva; falso, do contrário. A cada iteração o universo evolui de acordo com as seguintes regras:

- Uma célula viva que tenha menos de dois vizinhos, morre (regra da solidão).
- Uma célula viva que tenha dois ou três vizinhos, se mantém viva.
- Uma célula com mais de três vizinhos, morre (regra da superpopulação).
- Uma célula morta que tenha exatamente três vizinhos se torna uma célula viva (regra da reprodução).

A vizinhança da célula compreende as oito células adjacentes a mesma, sendo que, tradicionalmente, as bordas são consideradas periódicas, ou seja, uma célula na borda do universo é considerada vizinha de outra célula em oposição a mesma em relação a borda a qual está conectada.

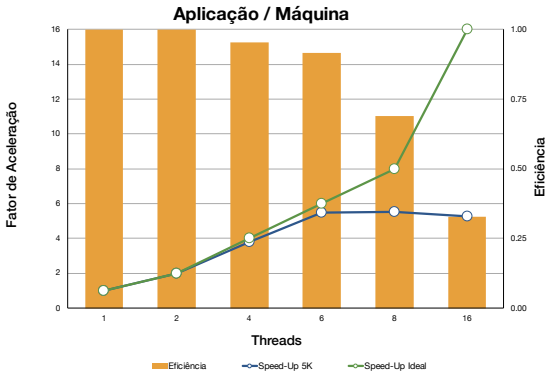
2 Implementação

O desenvolvimento da aplicação (Apêndice A) se baseou no código disponível no site *Rosetta Code* (https://rosettacode.org/wiki/Conway%27s_Game_of_Life#C). O programa foi modificado para aceitar mais parâmetros para simplificar a execução e permitir maior flexibilidade dos dados de teste. Uma segunda modificação envolveu normalizar a matriz que representa o tabuleiro - convertendo-a sem um vetor -, com o objetivo de transferir dados entre os processos de forma mais simples, utilizando-se matemática de ponteiros para as transferências.

O processo principal (mestre) tem três objetivos: obter os dados de teste (de um arquivo ou os gerando de forma aleatória); particionar os dados, enviando as partição para cada um dos demais processos (escravos); e finalmente receber os resultados dos escravos e agrupá-los. Os escravos, por sua vez, recebem uma partição do problema (conjunto de linhas) e o número de ciclos a executar. A cada ciclo, o escravo envia sua primeira linha para o processo com rank[1] inferior ao seu e a sua última linha para o escravo com rank superior ao seu. Em seguida recebe a linha dos processos anterior e posterior, de forma que cada processo tenha informações da vizinhança (necessário para o cálculo do ciclo). Por fim, o escravo avança um passo no ciclo. Após executar todas as iterações cada escravo envia sua partição - resultado - para o mestre.

3 Resultados

Os testes foram executados com universos gerados de forma aleatória e tamanho 1500×1500. O número de processos foi configurado em 2, 3, 5, 7, 9 e 17. Note-se que ignoramos, para avaliação do desempenho, um dos nodos, uma vez que o nodo mestre não realiza processamento, mas sim combina os resultados e tem impacto não significativo na execução. As simulações foram avaliadas no cluster *Grad*, com 1 nodo alocado em modo exclusivo (8 núcleos físicos, 16 com hyper-threading).



A implementação resulta em muitas trocas de mensagens entre os processos ($n\_msg = (n\_proc - 1) \times 2$), e como cada escravo precisa esperar pelos dados da vizinhança para executar a simulação, o mais lento dentre eles ditará a velocidade de processamento de cada etapa da simulação. Quanto maior a quantidade de processos, maior a quantidade de mensagens trocadas. Um teste adicional que executamos envolveu alocar mais nodos no cluster: nessa situação percebemos um ganho maior de eficiência do que com todos os processos sendo executados na mesma máquina.

[1] Em MPI o rank de um processo é o seu índice na rede, começando em 0.

A Código Fonte da Aplicação

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include "mpi.h"

int tag = 42; /* Message tag */
int my_rank;  /* Process identifier */
int proc_n;   /* # of running processes */
unsigned char* new = NULL;

// Displays the board
void show(unsigned char* univ, int w, int h) {
  system("clear");
  printf("\033[H");
  for (int y = 0; y < h; ++y) {
    for (int x = 0; x < w; ++x) {
      printf(univ[y * w + x] ? "\033[07m  \033[m" : "  ");
    }
    printf("\033[E");
  }
  fflush(stdout);
}

// Crate an empty board with size w x h
unsigned char* empty_univ(int w, int h) {
  unsigned char* univ = (unsigned char*) malloc(w * h * sizeof(unsigned char));
  for(int xy = 0; xy < w * h; ++xy) {
    univ[xy] = 0;
  }
  return univ;
}

// Advance a step on the simulation
void evolve(unsigned char* univ, int w, int h) {
  if (NULL == new) {
    new = empty_univ(w, h);
  }

  for (int y = 0; y < h; ++y) {
    for (int x = 0; x < w; ++x) {
      int n = 0;
      for (int y1 = y - 1; y1 <= y + 1; ++y1) {
        for (int x1 = x - 1; x1 <= x + 1; ++x1) {
          // Any cell outside the bounding box is considered to be dead.
          if (univ[((y1 + h) % h) * w + ((x1 + w) % w)]) {
            n++;
          }
        }
      }

      if (univ[y * w + x]) {
        n--;
      }
      new[y * w + x] = (n == 3 || (n == 2 && univ[y * w + x]));
    }
  }

  for (int xy = 0; xy < w * h; ++xy) {
    univ[xy] = new[xy];
  }
}

// Generate a random board
unsigned char* random_univ(int w, int h) {
  unsigned char* univ = empty_univ(w, h);

  int seed = time(NULL);
  srand(seed);
  for (int xy = 0; xy < w * h; ++xy) {
    if (rand() < (RAND_MAX / 5)) {
      univ[xy] = 1;
    } else {
      univ[xy] = 0;
    }
  }
  return univ;
}

// Read the board from a file.
// First line describes the size of the board
// Subsequential lines contain each of the lines of the board to play
unsigned char* read_from_file(char* filename, int* w, int* h) {
  FILE* file = fopen(filename, "r");
  char line[1024];

  if (file == NULL) {
    return NULL;
  }

  if(fgets(line, sizeof(line), file)) {
    char *currnum;
    int numbers[2], i = 0;

    while ((currnum = strtok(i ? NULL : line, " ,")) != NULL && i < 2) {
      numbers[i++] = atoi(currnum);
    }

    *h = numbers[0];
    *w = numbers[1];
  }

  unsigned char* univ = empty_univ(*w, *h);

  for (int y = 0; y < *h; ++y) {
    fgets(line, sizeof(line), file);
    for (int x = 0; x < *w; ++x) {
      if (line[x] == '1') {
        univ[y * *w + x] = 1;
      }
    }
  }

  fclose(file);
  return univ;
}

void run_master(int c, char** v) {
  int w = 0, h = 0, cycles = 10, print_result = 0;
  unsigned char* univ;

  // Decide if we're running the game from a file or generating it randomly
  if (0 == strcmp(v[1], "random")) {
    printf("Using randomly generated pattern.\n");

    if (c > 2) w = atoi(v[2]);
    if (c > 3) h = atoi(v[3]);
    if (c > 4) cycles = atoi(v[4]);
    if (c > 5) print_result = atoi(v[5]);

    if (w <= 0) w = 30;
    if (h <= 0) h = 30;

    univ = random_univ(w, h);
  } else if (0 == strcmp(v[1], "file")) {
    if (c < 3) {
      printf("Missing file path.\n");
      return;
    }

    univ = read_from_file(v[2], &w, &h);

    if (NULL == univ) {
      printf("Invalid input file.\n");
      return;
    }

    if (c > 3) cycles = atoi(v[3]);
    if (c > 4) print_result = atoi(v[4]);
  } else {
    printf("Missing arguments.\n");
    return;
  }
```

```
  if (cycles <= 0) cycles = 10;

  if (print_result < 0) print_result = 0;
  else if (print_result > 1) print_result = 1;

  // Partition the dataset and send to all nodes
  double t1,t2;
  t1 = MPI_Wtime();
  int temp[4];
  temp[0] = w;
  temp[1] = cycles;
  int partitions = proc_n - 1;
  int lines_per_partition = h / partitions;

  int current_line = 0;
  for (int i = 1; i < proc_n; i++) {
    temp[2] = current_line;
    current_line += lines_per_partition;
    if (current_line > h || i == proc_n - 1) {
      current_line = h;
    }
    temp[3] = current_line;

    // FIRST MESSAGE
    //  temp[0] = width of each line
    //  temp[1] = cycles to run
    //  temp[2] = start line the slave will get
    //  temp[3] = last line the slave will get
    MPI_Send(temp, 4, MPI_INT, i, tag,
        MPI_COMM_WORLD);

    // SECOND MESSAGE
    //  partition
    MPI_Send(univ + temp[2] * w, (temp[3] - temp[2]) * w
        , MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }

  MPI_Status status; /* MPI message status */

  // Receive data from all nodes
  current_line = 0;
  for (int i = 1; i < proc_n; i++) {
    int initial_line = current_line;
    current_line += lines_per_partition;
    if (current_line > h || i == proc_n - 1) {
      current_line = h;
    }
    MPI_Recv(univ + initial_line * w, (current_line -
        initial_line) * w, MPI_CHAR, i, tag,
        MPI_COMM_WORLD, &status);
  }
  t2 = MPI_Wtime();

  // Display the final result
  if (print_result) {
    show(univ, w, h);
  }

  printf("Simulation completed in [%f] after %d cycles.\n\n
      ", t2 - t1, cycles);
}

void run_slave() {
  int universe_data[4];
  MPI_Status status; /* MPI message status */

  MPI_Recv(universe_data, 4, MPI_INT, 0, tag,
      MPI_COMM_WORLD, &status);

  // Alocate two additional lines
  int h = universe_data[3] - universe_data[2];
  int w = universe_data[0];
  int cycles = universe_data[1];
  unsigned char* local_univ = (unsigned char *)malloc ((h
      + 2) * w * sizeof(unsigned char));
  for (int i = 0; i < h * w; ++i) {
    local_univ[i] = 0;
  }

  // Start wrting at the second line
  MPI_Recv(local_univ + w, h * w, MPI_CHAR, 0, tag,
      MPI_COMM_WORLD, &status);

  int c = 0;
  while (c < cycles) {
    if (my_rank > 1) {
      // Send first line to up
      MPI_Send(local_univ + w, w, MPI_CHAR, my_rank -
          1, tag, MPI_COMM_WORLD);
    }

    if (my_rank < proc_n - 1) {
      // Send last line to the down
      MPI_Send(local_univ + h * w, w, MPI_CHAR,
          my_rank + 1, tag, MPI_COMM_WORLD);
    }

    if (my_rank > 1) {
      // Receive from up
      MPI_Recv(local_univ, w, MPI_CHAR, my_rank - 1,
          tag, MPI_COMM_WORLD, &status);
    } else {
      // My first line should be empty if I'll not get it from
          up
      for (int i = 0; i < w; ++i) {
        local_univ[i] = 0;
      }
    }

    if (my_rank < proc_n - 1) {
      // Receive from down
      MPI_Recv(local_univ + (h + 1) * w, w, MPI_CHAR,
          my_rank + 1, tag, MPI_COMM_WORLD, &status);
    } else {
      // My last line should be empty, if I'll not get it from
          down
      for (int i = (h + 1) * w; i < (h + 2) * w; ++i) {
        local_univ[i] = 0;
      }
    }

    // Got all data the slave needs, run the evolution
    evolve(local_univ, w, (h + 2));
    ++c;
  }

  // Send results to master
  MPI_Send(local_univ + w, h * w, MPI_CHAR, 0, tag,
      MPI_COMM_WORLD);
}

int main(int c, char** v) {
  if (c < 2) {
    printf("Invalid number of arguments.\n");
    return -1;
  }

  // Ensure we're not buffering data on stdout (print
      everything right away)
  setbuf(stdout, NULL);

  MPI_Init (&c, &v);

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &proc_n);

  if (my_rank == 0) {
    run_master(c, v);
  } else {
    run_slave();
  }

  MPI_Finalize();

  return 0;
}
```