Programação p Sistemas Paralelos e Distribuídos / Prof.: Fernando W. Cruz

Lab sobre programação paralela/distribuída

A) Objetivos do laboratório

O objetivo deste laboratório é permitir que o aluno experimente a programação de aplicações paralelas usando memória distribuída (com MPI) e compartilhada (OpenMP), além dos recursos de interface gráfica, com CUDA e OpenCL em GPUs.

B) Roteiro do laboratório

Considere o texto a seguir:

O Jogo da Vida, criado por John H. Conway (GARDNER, 1970), utiliza um autômato celular para simular gerações sucessivas de uma sociedade de organismos vivos. Este jogo é composto por um tabuleiro bidimensional, infinito em qualquer direção, de células idênticas. Cada célula tem exatamente oito células vizinhas (todas as células que compartilham uma aresta ou um vértice com a célula original). Para identificar essa vizinhança, fixe os olhos em uma célula de um tabuleiro de xadrez e perceba as casas vizinhas); cada célula pode estar em um de dois estados: viva ou morta.

Uma geração da sociedade é representada pelo conjunto dos estados das células do tabuleiro. Sociedades evoluem de uma geração para a próxima aplicando simultaneamente, a todas as células do tabuleiro, regras que estabelecem o próximo estado de cada célula. As regras são:

- Células vivas com menos de 2 vizinhas vivas morrem por abandono
- Células vivas com mais de 3 vizinhas vivas morrem de superpopulação
- Células mortas com exatamente 3 vizinhas vivas tornam-se vivas
- As demais células mantêm seu estado anterior.

A simulação desse jogo pode ser feita pelo uso de um tabuleiro NxN, orlado por células eternamente mortas, no qual o "veleiro" (uma configuração de células vivas, permeadas por células mortas) sai do canto superior esquerdo e chega no canto inferior direito do tabuleiro, de modo a simular as mudanças geracionais desta sociedade de organismos.

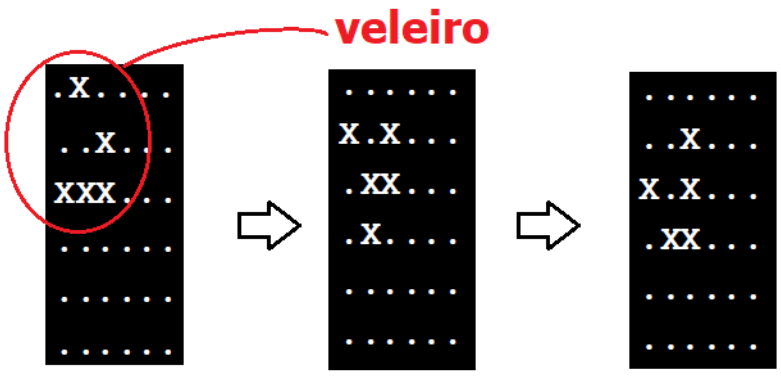

Fig 01 – Tabuleiro com matriz quadrada 6 x 6 e a configuração do "veleiro"

A Fig 01, é um exemplo de tabuleiro 6x6 com células vivas (1, expressas por um X) e mortas (0, expressa por um ponto). Neste exemplo, aparecem três gerações da sociedade de organismos,

pela aplicação das regras citadas. Vale ressaltar que um tabuleiro 6x6 orlado por células mortas obriga a definição de uma matriz 8 por 8 (as células de orla não aparecem na Fig 01).

Com base no programa `jogodavida.c` entregue junto com esta especificação, realize os seguintes passos:

1. Uma vez entendido o programa `jogodavida.c`, gere as seguintes versões deste código no cluster chococino, em linguagem C:
   - Uma versão MPI com o nome `jogodavidampi.c`, de modo a distribuir o trabalho de simulação da evolução da sociedade de organismos vivos entre os processos *workers*, numa proporção igual (ou próxima) ao número de *workers* instanciados.
   - Uma versão OpenMP com o nome `jogodavidaomp.c`, de modo a distribuir o trabalho de simulação da evolução da sociedade de organismos vivos entre as *threads*, numa proporção igual (ou próxima), em função do parâmetro OMP_NUM_THREADS. Por exemplo, supondo 4 *threads*, o programa irá calcular a evolução da sociedade numa proporção de 25% para cada thread.
   - Uma versão CUDA com o nome `jogodavida.cu`, de modo que o cálculo de evolução da sociedade seja feito com uso de uma das GPUs do cluster chococino.
   - Uma versão OpenMP do jogo da vida para acesso aos recursos da GPU (usando, por exemplo, pragmas como o *#pragma omp target*. Nesse caso, os alunos devem fazer pesquisa[1] sobre as principais diretivas e apresentar texto e exemplos de uso no relatório desse experimento. Na versão final, a aplicação deve realizar o cálculo de evolução da sociedade com uso de uma das GPUs do servidor.
2. Montar um experimento que permita comparar os códigos acima, de modo a se identificar qual deles apresenta melhor performance (menor tempo de execução), sob as mesmas condições (dimensões da sociedade, porções do código paralelizadas, número de *threads*, núcleos, etc.).

C) Questões de Ordem

- O experimento pode ser feito por grupos de, no mínimo, 3 e, no máximo, 4 alunos para turma com menos de 50 alunos e 5 alunos para turmas acima de 50 alunos. Nesse caso, basta que um dos alunos faça a postagem das entregas no Moodle da disciplina.
- Este laboratório deve ter os artefatos entregues no Moodle (arquivo zipado) e apresentado em data estabelecida pelo professor, se solicitado (os alunos devem estar preparados, caso sejam convidados à apresentação do trabalho).
- A entrega é composta por *(i)* um relatório, cuja estrutura e conteúdo está descrito a seguir, *(ii)* códigos, instruções de uso e todas as informações necessárias para esclarecimento e uso dos programas entregues, *(iii)* um vídeo gravado pelos membros participantes, com apresentação do projeto. Nesse caso, considerar uma média de 4 a 6 minutos por aluno para que possam demonstrar como participaram e conhecimentos adquiridos *(iv)* uma documentação sobre os códigos, comparações e configurações entregues – essa documentação deve fazer parte do relatório a ser entregue.

[1] Alguns exemplos em https://web.inf.ufpr.br/erad2022/wp-content/uploads/sites/35/2022/08/Minicurso-OpenMP-GPU-V-17_04_22_compressed.pdf e https://www.openmp.org/updates/openmp-accelerator-support-gpus/

- Em qualquer das versões, se o programa estiver certo, o veleiro (configuração inicial do tabuleiro com indicação de células vivas) deve sair do canto superior esquerdo e chegar no canto inferior direito do tabuleiro. Portanto, as comparações de desempenho só serão válidas se essa condição for satisfeita durante as execuções.
- Os alunos podem realizar o experimento em qualquer plataforma, mas devem considerar que a correção do experimento será feita no *host* 164.41.20.252 (códigos que não rodarem neste *host* não serão validados).
- O relatório deve conter o seguinte:
  i) Identificação da disciplina/turma, do grupo (matrícula e nome) e o nome do laboratório
  ii) Listagem dos códigos com comentários e indicações sobre qual parte foi paralelizada, instruções de execução e comentários sobre dificuldades e soluções encontradas em cada uma das versões do `jogodavida.c`
  iii) Descrição do experimento – Apresentar os cenários de teste que foram feitos (número de execuções realizadas, resultados encontrados, etc.), considerando parâmetros idênticos para cada um dos códigos. O resultado dos testes deve ser apresentado na forma de uma tabela comparativa, apresentando os de tempos de execução e melhoria de performance em relação à versão sequencial. Obs.: Sugere-se utilizar funções da linguagem que consigam calcular os tempos de execução para melhorar a qualidade da resposta.
  iv) Conclusão – Gerar um texto conclusivo sobre o experimento, indicando, por exemplo, *(i)* qual o percentual de ganho de uma solução em relação a outra, *(ii)* qual das GPUs do cluster chococino é mais eficiente para executar essa aplicação e outros achados que julgar pertinentes.

D) Referências

[1] Gardner, M. “Mathematical Games – The fantastic combinations of John Conway´s new solitaire game life”, Scientific American 223, Oct, 1970, pp 120-123. Disponível em http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm.

ANEXO – Código jogodavida.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define ind2d(i,j) (i)*(tam+2)+j
#define POWMIN 3
#define POWMAX 10

double wall_time(void) {
  struct timeval tv;
  struct timezone tz;

  gettimeofday(&tv, &tz);
  return(tv.tv_sec + tv.tv_usec/1000000.0);
```

```
} /* fim-wall_time */

double wall_time(void);

void UmaVida(int* tabulIn, int* tabulOut, int tam) {
  int i, j, vizviv;

  for (i=1; i<=tam; i++) {
    for (j= 1; j<=tam; j++) {
      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)];
    } /* fim-for */
  } /* fim-for */
} /* fim-UmaVida */

// DumpTabul: Imprime trecho do tabuleiro entre
//            as posicoes (first,first) e (last,last)
void DumpTabul(int * tabul, int tam, int first, int last, char* msg){
  int i, ij;

  printf("%s; Dump posições [%d:%d, %d:%d] de tabuleiro %d x %d\n", \
   msg, first, last, first, last, tam, tam);
  for (i=first; i<=last; i++) printf("="); printf("=\n");
  for (i=ind2d(first,0); i<=ind2d(last,0); i+=ind2d(1,0)) {
    for (ij=i+first; ij<=i+last; ij++)
      printf("%c", tabul[ij]? 'X' : '.');
    printf("\n");
  }
  for (i=first; i<=last; i++) printf("="); printf("=\n");
} /* fim-DumpTabul */

// InitTabul: Inicializa tabulIn com veleiro no canto superior esquerdo e
//            tabulOut com celulas mortas
void InitTabul(int* tabulIn, int* tabulOut, int tam){
  int ij;
  for (ij=0; ij<(tam+2)*(tam+2); ij++) {
    tabulIn[ij] = 0;
    tabulOut[ij] = 0;
  } /* fim-for */
```

```
  tabulIn[ind2d(1,2)] = 1; tabulIn[ind2d(2,3)] = 1;
  tabulIn[ind2d(3,1)] = 1; tabulIn[ind2d(3,2)] = 1;
  tabulIn[ind2d(3,3)] = 1;
} /* fim-InitTabul */

// Correto: Verifica se a configuracao final do tabuleiro
//          está correta ou não (veleiro no canto inferior direito)
int Correto(int* tabul, int tam){
  int ij, cnt;

  cnt = 0;
  for (ij=0; ij<(tam+2)*(tam+2); ij++)
    cnt = cnt + tabul[ij];
  return (cnt == 5 && tabul[ind2d(tam-2,tam-1)] &&
      tabul[ind2d(tam-1,tam  )] && tabul[ind2d(tam  ,tam-2)] &&
      tabul[ind2d(tam  ,tam-1)] && tabul[ind2d(tam  ,tam  )]);
} /* fim-Correto */

int main(void) {
  int pow, i, tam, *tabulIn, *tabulOut;
  char msg[9];
  double t0, t1, t2, t3;
  // para todos os tamanhos do tabuleiro
  for (pow=POWMIN; pow<=POWMAX; pow++) {
    tam = 1 << pow;
    // aloca e inicializa tabuleiros
    t0 = wall_time();
    tabulIn  = (int *) malloc ((tam+2)*(tam+2)*sizeof(int));
    tabulOut = (int *) malloc ((tam+2)*(tam+2)*sizeof(int));
    InitTabul(tabulIn, tabulOut, tam);
    t1 = wall_time();
    for (i=0; i<2*(tam-3); i++) {
      UmaVida(tabulIn, tabulOut, tam);
      UmaVida(tabulOut, tabulIn, tam);
    } /* fim-for */
    t2 = wall_time();

    if (Correto(tabulIn, tam))
      printf("**RESULTADO CORRETO**\n");
    else
      printf("**RESULTADO ERRADO**\n");
    t3 = wall_time();
    printf("tam=%d; tempos: init=%7.7f, comp=%7.7f, fim=%7.7f, tot=%7.7f \n",
     tam, t1-t0, t2-t1, t3-t2, t3-t0);
    free(tabulIn); free(tabulOut);
  }
  return 0;
} /* fim-main */
```