Desenvolvimento do Jogo da Vida de Conway utilizando a biblioteca OpenMP

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

3 de maio de 2018

Resumo

Esse relatório descreve o desenvolvimento do Jogo da Vida, definido pelo matemático John Conway, utilizando técnicas de processamento paralelo através da utilização da biblioteca OpenMP, que permite a identificação de trechos da aplicações que podem ser paralelizados e executados em múltiplas *threads*. O objetivo secundário é avaliar o ganho de desempenho conforme aumentamos o número de threads executando a aplicação.

1 Introdução

Este trabalho utilizou o Jogo da Vida, desenvolvido pelo matemático britânico John Conway, como plataforma de estudo da *Application Programming Interface* (API) OpenMP. O jogo não possui jogadores e avalia a evolução de células de acordo com uma série de regras, descritas na Seção 1.1.

1.1 Regras do Jogo da Vida

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.

1.2 OpenMP

OpenMP é uma API que permite a utilização de múltiplas threads através da adição de macros no código para definir quais áreas devem ser paralelizadas e quais não, bem como qual deve ser o tratamento das variáveis sendo processadas na região paralela. A biblioteca permite que controlemos o agendamento das threads e o escopo das variáveis, através de diretivas específicas.

O agendamento pode ser realizado das seguintes formas:

- Estático: a execução é dividia igualmente no número de threads disponível.
- Dinâmico: a alocação é feita a cada thread assim que a mesma está livre. Nesse caso é possível indicar o tamanho do trabalho a ser enviado para cada thread, sendo "um"por padrão. É um método útil quando o trabalho a ser executado por cada thread tem tempo de execução variável.
- Guiado: é um método similar a alocação dinâmica, mas que diminui a quantidade de trabalho enviada a cada thread conforme a execução prossegue. É possível indicar o tamanho mínimo do trabalho a ser enviado.
- Automático: o compilador é responsável por escolher o método mais adequado ao paralelismo de acordo com heurísticas da API.

As variáveis declaradas fora da região paralela tem seu escopo automaticamente definido como compartilhado, assim como aquelas declaradas dentro da região paralela são definidas como privadas (de acesso exclusivo a thread responsável pelo bloco). É importante ressaltar, dada a implementação deste trabalho, que os vetores dinamicamente alocados (através de *malloc*) são exclusivamente compartilhados. Os escopos permitidos são:

- Compartilhado: variáveis são acessíveis por todas as threads. Não existe controle implícito de concorrência, devendo o desenvolvedor fazer o controle, indicando a região crítica de acesso.
- Privado: as variáveis são acessíveis e tem seu valor conservado apenas no contexto da thread que as acessa. Nesse caso é possível definir se ela é inicializada com um valor antes

da execução (*firstprivate*) e se o valor da última thread que acessar a variável é mantido (*lastprivate*).

2 Implementação

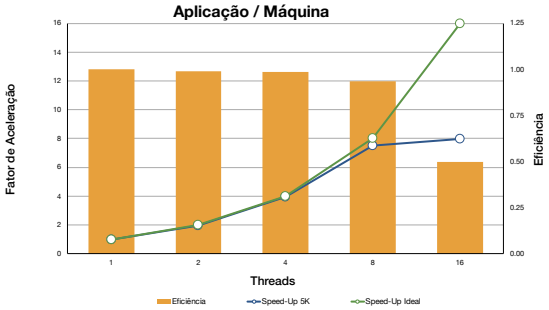
O desenvolvimento da aplicação se baseou no código (Apêndice A) disponível no site *Rosetta Code* (https://rosettacode.org/wiki/Conway%27s_Game_of_Life#C). O programa foi modificado para aceitar mais parâmetros para simplificar a execução. O laço principal, inicialmente feito de forma aninhada, foi expandido em um único laço de forma a simplificar a definição da região paralela, isto é, ao invés de fazer a iteração em $largura$ e $profundidade$ (aninhado), foi feita a iteração em termos de $largura \times profundidade$ (laço expandido). Duas regiões paralelas foram definidas, ambas no método *evolve*: a primeira no cáculo da evolução, onde uma matriz temporária armazena os resultados da próxima geração, e a cópia dessa matriz temporária para o universo atual (atualização da geração).

Em ambos os casos optou-se por manter o agendamento padrão (estático), uma vez que o tempo de execução de cada iteração varia pouco (laço que percorre os 8 vizinhos e cópia de valores). Apesar de ser o valor padrão, optou-se por explicitar que as matrizes são controladas como variáveis compartilhadas.

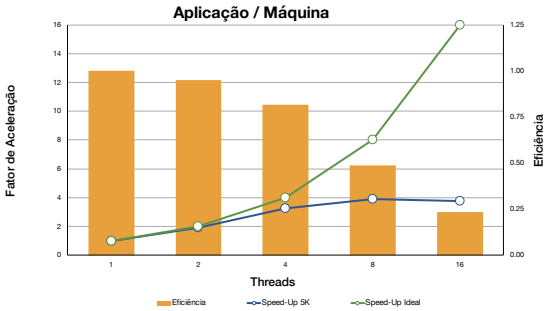
3 Resultados

A implementação final permite que se escolha entre utilizar arquivos ou gerar o universo de forma aleatória. Para os testes foram utilizados universos randômicos de tamanhos 50×50, 500×500 e 5000×5000. Além da utilização do cluster *Grad*, com 16 núcleos alocados, foi utilizado um computador pessoal equipado com processador Intel i7 4790K executando a 4.2GHz (4 núcleos físicos, 8 processadores lógicos) e 32GB de memória executando a 1600MHz.

Executando a aplicação no cluster *Grad*, com 16 núcleos alocados, percebemos que o desempenho cai rapidamente quando o número de threads alocadas é superior ao número de núcleos físicos da nó. Esse resultado é compreensível, uma vez que os núcleos executando em *hyper threading* compartilham recursos e só apresentam ganho significativo de desempenho quando as próprias aplicações tem desempenho melhor quando compartilhando recursos. No caso do Jogo da Vida cada thread tem seu próprio conjunto de dados, o que faz com que o desempenho degrade com mais threads, uma vez que temos apenas a sobrecarga do HT e não seus benefícios.



Ao executar a aplicação no computador pessoal tivemos resultados similares, com impacto de desempenho perceptível a partir de 8 threads.



4 Conclusão

Quando estamos desenvolvendo aplicações com o objetivo de obter o máximo desempenho de nossos computadores devemos pensar as implementações para tal. Na maioria das vezes ajustes serão necessários para que seja possível executar uma aplicação em vários núcleos paralelamente. Além disso, foi possível perceber, a partir dos resultados apresentados, que pensar no particionamento dos dados pode auxiliar na obtenção de ganhos de performance quando utilizando *hyper threading*, uma vez que o compartilhamento de recursos deve ser levado em consideração nessa situação.

Finalmente, a utilização da biblioteca OpenMP simplifica o desenvolvimento de aplicações paralelas. O algoritmo apresentado pode ser expandido para executar em múltiplos nodos se introduzirmos uma nova biblioteca que permita o controle da comunicação e cooperação entre os mesmos - como MPI.

A Código Fonte da Aplicação

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

unsigned char** new = NULL;

unsigned char** empty_univ(int w, int h) {
  unsigned char** univ = (unsigned char **) malloc(h * sizeof(unsigned char*));
  for(int i = 0; i < h; i++) {
    univ[i] = (unsigned char *)malloc(w * sizeof(unsigned char));
    for (int j = 0; j < w; j++) {
      univ[i][j] = 0;
    }
  }

  return univ;
}

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][x] ? "\033[07m  \033[m" : "  ");
    }
    printf("\033[E");
  }
  fflush(stdout);
}

void evolve(unsigned char ** univ, int w, int h) {
  if (NULL == new) {
    new = empty_univ(w, h);
  }

  // Nested for converted to expanded for in order to support nested loop
  #pragma omp parallel for shared(univ, new)
  for (int xy = 0; xy < w * h; ++xy) {
    int x = xy / h;
    int y = xy % h;

    int n = 0;
    for (int y1 = y - 1; y1 <= y + 1; y1++) {
      for (int x1 = x - 1; x1 <= x + 1; x1++) {
        if (univ[(y1 + h) % h][(x1 + w) % w]) {
          n++;
        }
      }
    }

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

  #pragma omp parallel for shared(univ, new)
  for (int xy = 0; xy < w * h; ++xy) {
    int x = xy / h;
    int y = xy % h;
    univ[y][x] = new[y][x];
  }
}

void game(int w, int h, unsigned char** univ, int cycles, int print_result, int display_timer) {
  int c = 0;
  double starttime, stoptime;

  starttime = omp_get_wtime();
  while (c < cycles) {
    if (display_timer > 0) {
      show(univ, w, h);
      usleep(display_timer);
    }
    evolve(univ, w, h);
    c++;
  }
  stoptime = omp_get_wtime();

  // Should we print the universe when simulation is over?
  if (1 == print_result) {
    show(univ, w, h);
  }
  printf("Simulation completed after %d cycles using %d threads. Environment size: width=%d, height=%d. Execution time: %4.5f seconds.\n\n", cycles, omp_get_max_threads(), w, h, stoptime-starttime);
}

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

  int seed = time(NULL);
  srand(seed);

  #pragma omp parallel for
  for (int xy = 0; xy < w * h; ++xy) {
    int x = xy / h;
    int y = xy % h;
    if (rand() < (RAND_MAX / 5)) {
      univ[y][x] = 1;
    } else {
      univ[y][x] = 0;
    }
  }
  return univ;
}

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][x] = 1;
      }
    }
  }

  fclose(file);

  return univ;
}

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

  int w = 0, h = 0, cycles = 10, display_timer = 0, print_result = 0, num_threads = 0;
  unsigned char** univ;
```

```
153
154  if (0 == strcmp(v[1], "random")) {
155    printf("Using randomly generated pattern.\n");
156
157    if (c > 2) w = atoi(v[2]);
158    if (c > 3) h = atoi(v[3]);
159    if (c > 4) cycles = atoi(v[4]);
160    if (c > 5) num_threads = atoi(v[5]);
161    if (c > 6) print_result = atoi(v[6]);
162    if (c > 7) display_timer = atoi(v[7]);
163
164    if (w <= 0) w = 30;
165    if (h <= 0) h = 30;
166
167    univ = random_univ(w, h);
168  } else if (0 == strcmp(v[1], "file")) {
169    if (c < 3) {
170      printf("Missing file path.\n");
171      return -1;
172    }
173
174    univ = read_from_file(v[2], &w, &h);
175
176    if (NULL == univ) {
177      printf("Invalid input file.\n");
178      return -1;
179    }
180
181    if (c > 3) cycles = atoi(v[3]);
182    if (c > 4) num_threads = atoi(v[4]);
183    if (c > 5) print_result = atoi(v[5]);
184    if (c > 6) display_timer = atoi(v[6]);
185  } else {
186    printf("Missing arguments.\n");
187    return -1;
188  }
189
190  if (cycles <= 0) cycles = 10;
191  if (display_timer < 0) display_timer = 0;
192
193  if (print_result < 0) print_result = 0;
194  else if (print_result > 1) print_result = 1;
195
196  if (num_threads <= 0) num_threads = 1;
197  omp_set_num_threads(num_threads);
198
199  game(w, h, univ, cycles, print_result, display_timer);
200
201  return 0;
202 }
```