Arquitetura de computadores (2021). Semestre 2. Prof. Juan G. Colonna

Objetivo: se familiarizar com conceitos de arquiteturas paralelas e threads usando OpenMP em C.

Descrição geral

Devem ser escritos dois programas em C que permitam executar os seguintes experimentos:

1) Uma multiplicação matricial sequencial e uma distribuída, variando o tamanho da matriz, mas mantendo constante o número de threads. Neste caso, o número de threads deve ser constante e igual ao número de cores da arquitetura específica de cada um. Calcular os speedups. Graficar tamanho da matriz vs speedup. Valendo 5 pts.
2) Uma multiplicação matricial distribuída seguida de um código sequencial que soma todos os elementos da matriz resultante. Neste caso, o tamanho da matriz deve ser constante e variar os threads. Graficar número de threads vs tempo total. Valendo 5pts.

São fornecidos dois códigos fontes *speedup.c* e *amdahl.c*. Os dois códigos devem ser completados conforme os comentários dentro dos arquivos. O primeiro programa deve ser utilizado para responder o ponto 1), enquanto que o segundo programa será utilizado para o ponto 2).

Observação geral

Antes de executar os experimentos e obter os resultados colocar no relatório uma breve descrição da arquitetura utilizada. Para isso, pode ser utilizado o programa hardinf. Incluir informações relevantes como por exemplo: marca e modelo do microprocessar, quantidade de cores, quantidade de threads físicos que permite por core, indicar se possui hyper-threading ou alguma tecnologia similar, hierarquia de memória (L1, L2, L3, etc) e os tamanhos das memórias em cada nível, etc.

Descrições específicas

1) Fazer uma tabela com as seguintes colunas: tamanho da matriz, tempo sequencial (seg), tempo distribuído (seg), speedup.
    a) Utilizar o número de threads igual ao número de cores. Variar o tamanho da matriz entre 10 e 1000 com intervalos de 10 valores.
    b) Repetir cada experimento no mínimo 10 vezes. Ou seja, os valores de cada linha da tabela devem ser a média de dez execuções do experimento. Por exemplo: executar o programa 10 vezes com N=200, mensurar os 10 tempos sequenciais e os 10 tempos distribuídos e calcular a média de cada um. Colocar na tabela o valor médio. Valendo 1pt.
    c) Fazer um plot de “N” vs “speedup”. Valendo 2pt.
    d) Escrever conclusões relacionadas ao tamanho do problema em relação ao paralelismo de threads comparando a abordagem sequencial vs distribuída. Valendo 2pt.

2) Fazer uma tabela com as seguintes colunas: número de threads, tempo dist (seg), tempo soma (seg) e tempo total (seg).
    a) Utilizar tamanho de matriz constante N=1000 e variar a quantidade de threads da variável de ambiente OMP_NUM_THREADS entre 1 e 30 com intervalos de 1. (configurar a variável antes de compilar cada nova rodada do experimento, ficar atento(a) para não obter resultados errados).

b) Repetir cada experimento no mínimo 10 vezes. Ou seja, os valores de cada linha da tabela deve ser a média de dez execuções do experimento. Por exemplo: executar o programa 10 vezes com OMP_NUM_THREADS=1, mensurar os 10 tempos e calcular a média. Colocar na tabela o valor médio. Valendo 1pt.
c) Fazer um plot de OMP_NUM_THREADS vs tempo total. Reparar que o gráfico de seguir uma curva com decaimento exponencial aproximado com uma assíntota inferior igual ao tempo do programa que executa de forma sequencial, ou seja ao tempo da soma. (está no livro). Valendo 1pt.
d) Graficar o speedup usando a equação teórica da lei de Amdahl considerando o valor médio de "tempo dist(seg)" obtido com OMP_NUM_THREADS=1, a partir desses dados variar a quantidade de melhoria entre 1 e 100. Reparar que o gráfico deve seguir uma curva monotonicamente crescente logarítmica. Valendo 1pt.
e) Comparar as curvas teóricas e práticas e discutir as conclusões. Escrever conclusões que relacionam o comportamento da curva do experimento com a lei de amdahl. Valendo 2pt.

O trabalho deve ser enviado num arquivo .zip com o nome completo do aluno(a) por exemplo: pepa_pig.zip, contendo os dois códigos em C, mais o relatório em PDF (nunca word). Prazo 19/11/2021 até 26/11/2021 (Colabweb).

Informações adicionais

- Recomendação: utilizar linux e instalar OpenMP.
- Utilizar nível de otimização do compilador -O0.
- compilar usando o argumento -fopenmp. Por exemplo: $ gcc -O0 speedup.c -fopenmp
- antes de compilar definir a quantidade de threads com a variável de ambiente $ export OMP_NUM_THREADS=4 (observar que o número de threads de respeitar o que foi pedido no enunciada de cada caso)
- Exemplo de execução $ ./a.out 100, onde o segundo argumento é o tamanho da matriz.

Observações específicas

- O objetivo de calcular a média dos tempos é porque outros processos executados no mesmo computador podem atrapalhar a medição quando o SO escalona esses processos.
- Perceba que gerar as tabelas de forma incorreta, implica em gráficos incorretos, que por sua vez implicam em conclusões erradas.
- O professor se reserva o direito de chamar o aluno(a) para uma defesa oral do trabalho que pode incluir perguntas teóricas. Com agendamento prévio.
- Os professores no geral não gostam de alunos(as) com a atitude "se não está no enunciado, então não faço". Se tiverem dúvidas perguntem!
- Não deixar para última hora, não haverá extensão de prazo.
- Um bom gráfico deve ser legível, o tamanho da fonte e das linhas deve ser razoável, os eixos devem estar devidamente identificados e a legenda deve ser explicativa. Se tiver que fazer uma marcação em algum ponto da curva, essa marcação deve estar identificada.
- Os programas devem compilar e executar pelo menos na versão gcc 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)

Arquivo speedup.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int** create_matrix(int rows) {
   /* retorna ponteiro para vetor de ponteiros
   utilizar dois mallocs para alocar uma matriz quadrada*/
   return mat;
}

void init_matriz(int **M, int v, int N){
   /* inicializa os valores da matriz com um número inteiro v */
}

void print_matriz(int **M){
   /* função útil para verificar se os cálculos estão corretos */
   for (int i=0; i<10; i++) {
       for (int j=0; j<10; j++)
           printf("%d ", M[i][j]);
       printf("\n");
   }
}

void mat_mul_sequential(int **X, int **Y, int **Z, int N){
/* escrever o código de multiplicação sequencial de matrizes (sem paralelização) */
}


void mat_mul_dist(int** X, int** Y, int** Z, int N) {
   int i,j,k; // contadores dos loops
   // inserir aqui q diretiva do OpenMP com as variáveis compartilhadas (X,Y,Z,N) e
variáveis particulares (i,j,k)
   {
       #pragma omp for schedule(static)
       /* inserir aqui o código para multiplicar matrizes */
   }
}

int main(int argc, char** argv) {
   /* o tamanho da matriz é passado como argumento na hora de executar */
   if(argc!=2) {
       printf("unexpected number of arguments\n");
       return -1;
   }
   char *a = argv[1];
   int N = atoi(a); // quantidade de linhas e colunas da matriz
   printf("Tamanho da matriz: %d\n", N);

   /* informações sobre a varíavel do ambiente export OMP_NUM_THREADS=8 */
```

```
   int thread_num = // função do OpenMP que retorna o número maximo de threads
   printf("Quantidade de threads disponíveis: %d\n", thread_num);

   /* informações sobre a arquitetura */
   int procs_num = // função do OpenMP que retorna o número cores da arquitetura
   printf("Quantidade de processadores disponíveis: %d\n", procs_num);

   int **X = create_matrix(N);
   int **Y  = create_matrix(N);
   int **Z  = create_matrix(N);

   init_matriz(X, 2, N);
   init_matriz(Y, 1, N);

   double wtime = omp_get_wtime(); // função que retorna o tempo decorrido em
segundos.
   mat_mul_sequential(X, Y, Z, N);
   wtime = omp_get_wtime() - wtime;
   printf("Tempo sequencial: %g\n", wtime);
   printf("\n");
   // print_matriz(Z);

   wtime = omp_get_wtime();
   mat_mul_dist(X, Y, Z, N);
   wtime = omp_get_wtime() - wtime;
   printf("Tempo distribuido: %g\n", wtime);
   printf("\n");
   // print_matriz(Z);

   return 0;
}
```

Arquivo amdahl.c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int** create_matrix(int rows) {
   /* retorna ponteiro para vetor de ponteiros
   utilizar dois mallocs para alocar uma matriz quadrada*/
   return mat;
}

void init_matriz(int **M, int v, int N){
   /* inicializa os valores da matriz com um número inteiro v */
}

void print_matriz(int **M){
   /* função util para verificar se os calculos estão corretos */
```

```
    for (int i=0; i<10; i++) {
        for (int j=0; j<10; j++)
            printf("%d ", M[i][j]);
        printf("\n");
    }
}

int soma_acumulada(int **Z, int N){
    int soma = 0;
    /* função que soma todos os valores da matriz e atualiza a variável soma */
    return soma;
}

void mat_mul_dist(int** X, int** Y, int** Z, int N) {
    int i,j,k; // contadores dos loops
    // inserir aqui q diretiva do OpenMP com as variáveis compartilhadas (X,Y,Z,N) e
variáveis particulares (i,j,k)
    {
        #pragma omp for schedule(static)
        /* inserir aqui o código para multiplicar matrizes */
    }
}

int main(void) {
    int N = ?; // deixar fixo respeitando o valor do enunciado da atividade
    int thread_num = omp_get_max_threads();
    printf("Quantidade de processadores disponíveis: %d\n", omp_get_num_procs());
    printf("Quantidade de threads disponíveis: %d\n", thread_num);

    int **X = create_matrix(N);
    int **Y  = create_matrix(N);
    int **Z  = create_matrix(N);

    init_matriz(X, 2, N);
    init_matriz(Y, 1, N);

    double wtime1 = omp_get_wtime();
    // Multiplicação distribuída
    wtime1 = omp_get_wtime() - wtime1;
    printf("Tempo distribuido: %g\n", wtime1);

    double wtime2 = omp_get_wtime();
    // soma acumulada
    wtime2 = omp_get_wtime() - wtime2;
    printf("Tempo sequencial: %g\n", wtime2);

    printf("Tempo total: %g\n", // tempo total (dist + soma));
    printf("\n");
    return 0;
}
```