Entrega: Trabalho 1 (Soma dos Termos da Progressão Harmônica usando OpenMP) [Peso: 50% do G1; Data de entrega: 07/10/2021]

O objetivo principal deste trabalho é paralelizar, usando a ferramenta OpenMP, um programa para calcular, com determinado número de casas decimais, a soma dos termos da seguinte progressão harmônica:

1/1, 1/2, 1/3, 1/4, ...

Sendo a soma dessa progressão determinada, portanto, por:

Sn = ∑ i=1n (1/i)

Por exemplo, S7, com 12 dígitos de precisão após a vírgula (ou após o ponto na convenção inglesa/americana), será igual a 2.592857142857.

Este programa está entre os desafios propostos na 1ª Maratona de Programação Paralela do SBAC-PAD - 2006 (<http://lspd.mackenzie.br/marathon/06/problems.html>).

Ao mesmo tempo que serão usadas as diretivas de OpenMP vistas em aula sobre um problema real, será possível também fazer uma reflexão sobre as questões que devem ser levadas em consideração quando um código sequencial é paralelizado.

Considerando uma precisão de 1000 dígitos após a vírgula (o que, no código sequencial disponibilizado a seguir, corresponde à constante DIGITS), sua tarefa é paralelizar a versão sequencial (código-fonte listado a seguir) que calcula os 1000 dígitos dos somatórios dos termos da progressão harmônica com n variando de 1000000 (um milhão) até 10000000 (dez milhões), com um passo de 1000000 (um milhão). Este código sequencial calcula assim esses 10 somatórios com 1000 dígitos. Os somatórios são impressos na saída pardão (*stdout*) e os tempos gastos no cálculo de cada somatório são impressos na saída de erro (*stderr*). Com as informações mostradas na saída padrão será possível verificar se o programa está mostrando os resultados corretos; e com as informações mostradas na saída de erro será possível traçar um gráfico que mostra o desempenho da aplicação à medida que o número de termos da progressão aumenta.

Sua tarefa é **paralelizar de forma correta o algoritmo para cálculo do somatório que está implementado na função sum com as diretivas de OpenMP**, que devem ser colocadas nos pontos corretos e com as cláusulas adequadas para melhorar o desempenho da aplicação.

Após executar a versão sequencial e paralelizá-la, os programas devem ser executados em um nodo do *cluster* grad com 2, 4, 8 e 16 *threads* - lembre-se de que o próprio programa se encarrega de gerar resultados variando o valor de *n*. É fortemente recomendável que cada aplicação seja executada mais de uma vez e que sejam tomados os melhores resultados individualmente para compor gráficos de tempos, *speed-up* e eficiência (nestes gráficos deve-se mostrar a variação destes parâmetros à medida que o valor de n varia).

Em relação à paralelização, todos os laços da função sum devem ser analisados quanto à viabilidade de execução paralela, sendo importante observar o seguinte: condições de corrida podem ocorrer e poderiam ser evitadas usando-se a diretiva para seções críticas de OpenMP, porém o uso de uma seção crítica em um laço paralelo prejudica o desempenho da aplicação em função dos bloqueios. Uma solução mais interessante para melhorar o desempenho é identificar a variável (ou variáveis) que está (ou estão) gerando a condição de corrida, dando a cada *thread* uma cópia desta variável (ou variáveis) e agregando os valores desta variável (ou variáveis) ao final do laço (de forma semelhante ao que é feito com reduções).

Como o programa gera resultados na saída padrão (*stdout*) e tempos na saída de erro padrão (*stderr*), será possível separar estas duas informações. Na linha de comandos, pode-se, por exemplo, usar:

./sum 1> saida.txt 2> tempos.txt

Assim, cada execução gerará um arquivo com o resultado (saida.txt) e um arquivo com os tempos (tempos.txt).

Observe que **cada resultado de uma execução paralela deve gerar resultados iguais à respectiva execução sequencial**, porém com tempos de execução provavelmente menores.

Cada aluno ou grupo (de no máximo 3 integrantes) deve entregar um relatório em formato PDF (*Portable Document Format*) com: descrição dos testes realizados, descrição do processador, resultados obtidos (gráficos mostrando os resultados de várias execuções), análise dos resultados e conclusões.

Os itens para avaliação são:

* formação de um grupo de no máximo 3 integrantes em até uma semana após a divulgação do enunciado do trabalho;
* execução da versão sequencial (para viabilizar o cálculo de *speed-up* e eficiência);
* implementação da versão paralela do algoritmo em C e OpenMP (considerando a estratégia e a complexidade da solução paralela);
* cálculo de *speed up* e de eficiência para os testes paralelos executados;
* análise do balanceamento da carga na execução do programa paralelo;
* relatório escrito usando *template* de artigo disponibilizado no moodle (em formato .PDF), com no mínimo 2 páginas de texto e com gráficos;
* clareza do código-fonte da versão paralela (utilização de comentários e nomes de variáveis adequadas), que deverá ser entregue compactado juntamente com o artigo;
* arquivo compactado no formato ZIP incluindo artigo em PDF, código-fonte e demais arquivos necessários à compilação e execução;
* submissão através do moodle até o dia e o horário definidos na respectiva sala de entrega;
* análise do número de horas máquina usadas pelo grupo no LAD (Laboratório de Alto Desempenho) da PUCRS durante o desenvolvimento do trabalho.

**NÃO serão aceitos artigos que NÃO tenham feito a execução no *cluster* grad do LAD da PUCRS. Da mesma forma, artigos com dados forjados (inventados ou copiados) ou cuja execução tenha sido feita de forma incorreta (por exemplo, na máquina grad.lad.pucrs.br, em vez de em um dos nodos do *cluster* grad), serão avaliados com nota ZERO.**

Serão adotados os seguintes padrões na avaliação:

* nota 0 (zero) para trabalhos: copiados, com dados forjados, com dados que não foram obtidos nas estações de trabalho do *cluster* grad (cuidado para não executar aplicações na máquina grad), com integrantes não registrados no grupo dentro do prazo, sem artigo, ou sem resultados de execução devidamente documentados no artigo;
* nota máxima 3 (três) para trabalhos com erros de implementação que gerem resultados incorretos;
* nota máxima 9 para trabalhos que tenham feito a paralelização apenas inserindo primitivas de OpenMP sem criar uma forma mais elegante de melhorar o desempenho do que simplesmente usar seções críticas.

**Arquivo: sum.c**

/\* sum.c (Roland Teodorowitsch; 16 set. 2021)

\* Compilation: gcc sum.c -o sum -fopenmp

\* Adapted from: http://lspd.mackenzie.br/marathon/06/sum.zip

\*/

#include <stdio.h>

#include <string.h>

#include <omp.h>

#define START 1000000

#define STEP 1000000

#define END 10000000

#define DIGITS 1000

void sum(char\* output, const long unsigned int d, const long unsigned int n) {

long unsigned int digit, i, remainder, div, mod;

long unsigned int digits[d + 11];

for (digit = 0; digit < d + 11; ++digit) {

digits[digit] = 0;

}

for (i = 1; i <= n; ++i) {

remainder = 1;

for (digit = 0; digit < d + 11 && remainder; ++digit) {

div = remainder / i;

mod = remainder % i;

digits[digit] += div;

remainder = mod \* 10;

}

}

for (i = d + 11 - 1; i > 0; --i) {

digits[i - 1] += digits[i] / 10;

digits[i] %= 10;

}

if (digits[d + 1] >= 5) {

++digits[d];

}

for (i = d; i > 0; --i) {

digits[i - 1] += digits[i] / 10;

digits[i] %= 10;

}

sprintf(output,"%lu.",digits[0]);

unsigned long int t = strlen(output);

for (i = 1; i <= d; ++i)

output[t++] = digits[i]+'0';

output[t] = '\0';

}

int main() {

long unsigned int n;

double start, finish;

char output[DIGITS + 10]; // extra chars to avoid error

for (n=START; n<=END; n+=STEP) {

start = omp\_get\_wtime();

sum(output, DIGITS, n);

finish = omp\_get\_wtime();

fprintf(stdout,"%s\n",output);

fprintf(stderr,"%lu %lf\n",n,finish-start);

}

return 0;

}