Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Aula Prática 07 - *OpenMP*

Identificação da Disciplina

Código da Disciplina	Nome da Disciplina	Turma	Professor	Período
117935	Programação Concorrente	A	Jeremias Moreira Gomes	2019/2

Objetivo da Aula Prática

O objetivo desta aula prática é apresentar técnicas básicas de paralelismo, por meio da utilização do *OpenMP*.

Detalhes Acerca das Aulas Práticas

0.1. Restrições deste Trabalho Prático

Durante a escrita de quaisquer códigos, tentar não utilizar bibliotecas exóticas que não sejam as disponíveis no UNIX como biblioteca padrão. Se o fizer (pois ajuda na realização de testes, por exemplo), lembrar que no momento da correção o sistema padrão a ser utilizado é similar a uma instalação padrão das máquinas do LINF e nada além do código será instalado apenas para uma correção individual. Então o uso de uma `<gtest.h>`, por exemplo, pode acabar fazendo com que o aluno perca pontos, porque seu código não pôde ser testado, em virtude da falta desta no computador de testes.

O atraso da entrega da atividade, incorre no desconto de 0.5 pontos a cada 60 minutos de atraso, em relação ao prazo de entrega da atividade e da nota final do aluno nesta atividade.

Esta atividade pode ser feita em dupla.

0.2. Por onde entregar os exercícios das aulas práticas?

As atividades envolvendo as aulas práticas deverão ser entregues via Aprender (https://aprender.ead.unb.br), na disciplina de Programação Concorrente. As informações para ingresso são as seguintes:

- URL: https://aprender.ead.unb.br/course/view.php?id=6775
- Chave de Acesso: `s3nhaD3ss3semestre201902`

Após o ingresso na disciplina, haverá uma atividade chamdada "Aula Prática 07 - *OpenMP*", para submissão dos exercícios.

0.3. O que deverá ser entregue, referente as aulas práticas?

Deverão ser entregues respostas referentes a todos os tópicos das **Seções de Atividades** ao longo deste documento. Essas atividades (dessa aula prática) estão divididas em duas categorias:

- Questionários (pergunta e resposta).
- Elaboração de Códigos.

Todos os dois tipos de atividades deverão ser entregues em um documento único contendo todas as respostas. Esse documento deverá ter identificação do aluno (nome e matrícula), identificação da disciplina, identificação da aula prática e as respostas identificadas de maneira igual as numerações em que aparecem neste documento. Para auxiliar na elaboração desse documento, pode-se utilizar esse documento (clique aqui) de referência.

Além disso, as questões de elaboração de código também deverão ser entregues em arquivos (.c) separados, sendo um arquivo para cada código elaborado. O início desse código deverá vir com comentários fazendo referência ao autor do código, nome do arquivo e a identificação da atividade, da seguinte forma:

```
// autor: Jeremias Moreira Gomes
// arquivo: exemplo-arquivo.c
// atividade: 0.0.0

#include <stdio.h>

int main()
{
    printf("\n");
    return 0;
}
```

Assim, os códigos elaborados irão estar em arquivos separados e no relatório.

0.4. Como entregar as atividades das aulas práticas?

A submissão das atividades deverá ser feita em um arquivo único comprimido no tipo zip (`Zip archive data, at least v1.0 to extract`) contendo um diretório com o relatório e os códigos elaborados durante a atividade. Além disso, para garantir a integridade do conteúdo entregue, o nome do arquivo comprimido deverá possuir duas informações (além da extensão `.zip`):

- A matrícula do aluno (ou dos alunos separadas por hífen, em caso de dupla).
- O *hash* md5 do arquivo `.zip`.

Para gerar o md5 do arquivo comprimido, utilize o comando `md5sum` do Linux e em seguida faça o renomeamento utilizando o *hash* coletado. Exemplo:

```
[6189] j3r3mias@tardis:aula-01-processos|master > zip -r aaa.zip atividade-01/
  adding: atividade-01/ (stored 0%)
  adding: atividade-01/relatorio.docx (stored 0%)
  adding: atividade-01/01-hello-3-fork.c (deflated 34%)
  adding: atividade-01/03-exemplos.c (deflated 47%)
  adding: atividade-01/02-arvore.c (deflated 35%)
  adding: atividade-01/02-pid_t.c (deflated 39%)
  adding: atividade-01/01-hello-fork.c (deflated 21%)
  adding: atividade-01/03-processos-e-ordens.c (deflated 55%)
[6190] j3r3mias@tardis:aula-01-processos|master > ls -lha aaa.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 2,5K set  4 23:26 aaa.zip
[6191] j3r3mias@tardis:aula-01-processos|master > md5sum aaa.zip
fe36b6799eae103a464cbc4857fce404  aaa.zip
[6192] j3r3mias@tardis:aula-01-processos|master > mv aaa.zip 160068444-fe36b6799eae103a464cbc4857fce404.zip
[6193] j3r3mias@tardis:aula-01-processos|master > ls -lha 160068444-fe36b6799eae103a464cbc4857fce404.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 2,5K set  4 23:26 160068444-fe36b6799eae103a464cbc4857fce404.zip
[6194] j3r3mias@tardis:aula-01-processos|master > md5sum 160068444-fe36b6799eae103a464cbc4857fce404.zip
fe36b6799eae103a464cbc4857fce404  160068444-fe36b6799eae103a464cbc4857fce404.zip
[6195] j3r3mias@tardis:aula-01-processos|master >
```

1. Coleta e Experimentação

Ao lidar com paralelismo, o fator experimentação e coleta de tempos está sempre envolvido. Para esta seção, o código base para testes será esse:

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

long long int somavalores(int *valores, int n)
{
    long long int soma = 0;

    for (int i = 0; i < n; i++) {
        soma = soma + valores[i];
    }

    return soma;
}

int main()
{
    long long int i, n, soma;
    int *valores;

    // scanf("%lld", &n);
    n = 100000000;

    valores = (int *)malloc(n * sizeof(int));

    for (i = 0; i < n; i++) {
        valores[i] = 1;
    }

    soma = somavalores(valores, n);

    printf("Soma: %lld - %s\n", soma, soma == n ? "ok" : "falhou");

    return 0;

}
```

Este programa cria um vetor de tamanho n, insere o valor 1 em todo ele e a função principal realiza a soma desses valores. Dessa forma, o primeiro caso a se levar em consideração, é como realizar as coletas de tempo. A um primeiro momento, o Sistema Operacional Linux provê o comando `time` que serve para mensurar três tipos de tempo envolvendo processos:

- `Real` - tempo decorrido entre a execução e a conclusão do comando.
- `User` - tempo gasto pelo usuário no processador.
- `Sys` - tempo usado pelo sistema (kernel) para executar o comando.

Assim, é possível mensurar o tempo gasto pelo processo referente ao programa, analisando tempo gasto pelo sistema (chamadas no modo kernel), tempo de processamento (modo usuário) e tempo total ou real gasto do início ao fim da chamada do programa (também chamado de *wall time*).

1.1. Atividade - Mensure o tempo gasto pelo programa base, utilizando o comando *time*.

Como a intenção é medir apenas o tempo utilizado pelo programa, tentando filtrar o máximo possível de interferências externas, a mensuração mais comum utiliza chamadas dentro do próprio programa para realizar a medição de tempo. O *OpenMP* disponibiliza a função `omp_get_wtime()` que retorna o tempo de relógio em segundos.

Como um ponto de partida, a intenção dessa aula é paralelizar somente o processamento da função `somavalores`, desconsiderando entrada e inicialização de dados[1].

1.2. Atividade - Mensure o tempo gasto pelo programa base, utilizando a função *omp_get_w_time*, medindo desde o início do programa até o término (liberação da memória alocada) e medindo apenas o tempo de processamento. Como a intenção é reduzir o tempo de interferência do SO (principalmente no segundo caso), realize a medição no mínimo três vezes para verificar a média dos tempos decorridos.

A partir desse ponto, temos em mãos o tempo base como um ponto de partida para paralelizarmos o código. Foi apresentado em aula o `#pragma omp parallel`, para criar uma região paralela. O que deve ser observado para esse programa, é que como os resultados estão sendo acumulados em uma variável chamada `soma`, não é possível apenas utilizar a diretiva citada, pois todas as *threads* irão realizar o mesmo trabalho. Dessa forma, deve-se alterar os limites da variável de iteração (`i`) para iniciar pela id da *thread* e iterar na quantidade de *threads* por vez. Além disso, a variável soma é compartilhada e somente essa alteração pode levar a uma condição de corrida em seu acesso. Para tentar, em um primeiro momento, resolver o impasse. O *OpenMP* disponibiliza o `pragma omp critical` que ao ser utilizado, informa que apenas uma *thread* por vez irá executar a instrução de seu contexto. O problema dessa instrução é que apesar da existência de todas as *threads*, a principal instrução estará sendo executa uma por vez e não de forma paralelizada.

1.3. Atividade - Utilize a diretiva *pragma omp parallel* para paralelizar o trecho do código base que realiza as somas. O laço deve ser modificado de forma a variável *i* iterar em cima da quantidade de *threads*, ao invés de apenas incrementar. E a diretiva *critical* do *OpenMP* para sanar a condição de corrida envolvendo a variável *soma*. Mensure os tempos utilizando 1, 2 e 4 *threads*, que coincidem com a quantidade de núcleos dos computadores utilizados no laboratório.

O `pragma omp critical` garantiu a exclusão mútua, mas não auxiliou em termos de paralelismo do código, pois apenas uma *thread* realiza a soma por vez. Para tentar ajudar, o *OpenMP* possui a diretiva `atomic` onde ao invés de criar por software a exclusão mútua, ele utiliza de instruções de hardware para reduzir o *overhead* envolvendo as operações.

[1] Para as atividades seguintes, sempre que for pedido para mensurar e comparar o tempo, calcule o *speedup* comparando-o com o código base (isso para as versões paralelas)

1.4. Atividade - Troque a diretiva *critical* por *atomic* e mensure a diferença entre os tempos de ambos.

Como a intenção é paralelizar e reduzir o tempo, utilizar abordagens com regiões críticas não são o melhor caminho (deve utilizar quando não há outra opção). Uma mudança de abordagem no código, para evitar regiões críticas, é utilizar uma variável local chamada `soma_parcial` onde cada *thread* poderia trabalhar livremente realizando suas somas e somente ao final do trabalho principal, cada *thread* irá inserir o seu conteúdo somado para a soma final. Dessa forma, a região crítica só será chamada uma vez por *thread*, permitindo alcançar maior paralelismo.

1.5. Atividade - Altere o programa base para utilizar somas locais por *thread* antes de acumular na variável soma os resultados do trabalho individual de cada uma. Lembre-se que acumular as somas parciais na soma total é uma região critica e deve ser tratada (*critical ou atomic*) para evitar condições de corrida. Mensure os tempos e compare com o código base.

Continuando, como apresentado em aula, ao invés de se realizar individualmente a alteração do laço for para manipular a divisão do trabalho, a diretiva `{#pragma omp for}` dentro da região paralela faz todo o trabalho da divisão do laço entre as *threads* criadas.

1.6. Atividade - A partir do código da atividade anterior (somas parciais separadas e em seguida acúmulo na soma total), altere o código para utilizar a diretiva *pragma omp for* (lembre-se que a região crítica ainda precisa ser tratada)

.

Por último, envolvendo o código base, deve-se perceber que colocar cada *thread* ainda executa uma operação em serial (o acúmulo na soma final), utilizando uma região crítica. A combinação de variáveis locais de *threads* em uma variável única em paralelismo é uma ação bem comum e é chamada de **redução**. A maioria dos ambientes de programação paralela provê suporte a realização de reduções de forma rápida, e a própria diretiva `omp parallel for` contém uma cláusula chamda `reduction` para realizar essa operação. Assim, pode-se utilizar:

```
#pragma omp paralle for private(...) reduction(opercao: variavel)
```

Dentro da região paralela a divisão de trabalho:

- Fará uma cópia local de carda variável da lista de *reduction*.
- Inicializará essas variáveis locais (0 se for adição e 1 se for multiplicação).
- Atualizará a partir das cópias locais.
- Irá acumular, ao final, na variável original (variável global).

1.7. Atividade - Altere o programa base para utilizar a redução provida pelo *OpenMP* e mensure os tempos comparando com o código base.