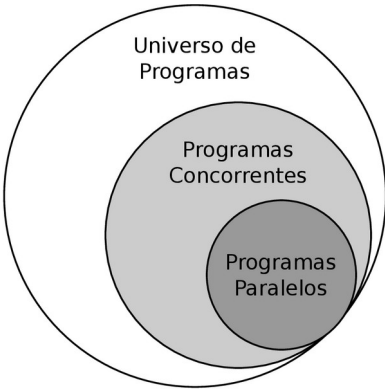
Atividade Prática encontra-se no final deste documento, páginas 7 e 8.

Programação Paralela

Considerando que processadores com memória compartilhada UMA e NUMA estão em todos os lugares, desde grandes servidores, computadores pessoais e chegando até os celulares, é importante que os programas escritos possam se aproveitar destes núcleos extras, com o objetivo de melhorar o desempenho e garantir a melhor eficiência durante sua execução. O grande porém é que não existe um compilador/software que seja capaz de identificar trechos paralelizáveis de código e automagicamente melhorar a eficiência deste, é o programador que deve fazer isso.

Considere a figura abaixo:



Dentre todos os programas que existem, ou ainda serão criados, uma parte é considerado programas concorrentes. Programas concorrentes são programas que permitem que várias ações sejam divididas e intercaladas em uma unidade de execução, passando a sensação de que todas estão ativas ao mesmo tempo, as aplicações de servidores webs são exemplos de aplicações concorrentes. E dentro os programas concorrentes, existem um grupo de programas que são naturalmente paralelos. Programas paralelos são programas que permitem que várias ações sejam divididas e executadas em unidades de execução diferentes, como exemplo, pode-se citar os algoritmos de solução de equações diferenciais parciais.

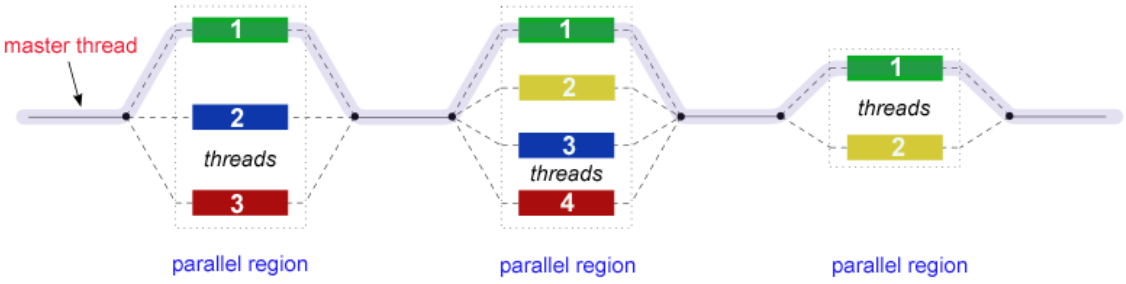
Desta forma, o programador deve criar seu código, normalmente serial, identificar a região que é concorrente e, a partir desta região, identificar o que pode ser paralelizado. E esta identificação é que determina se um programa paralelo será eficiente ou não, pois o ato de codificar é apenas a etapa final da construção de um programa paralelo.

Existem diversas APIs (Application Programming Interface) que ajudam os programadores a criarem códigos paralelos. Um exemplo é a OpenMP, voltada para sistemas UMA e NUMA.

OpenMP

OpenMP é uma API para programação paralela explícita constituída basicamente de bibliotecas de funções, diretivas de compilação e variáveis de ambiente. Tem o objetivo de ser um padrão na programação paralela, ser de fácil uso e portável.

A API OpenMP foi desenvolvida para funcionar em sistemas multiprocessador com memória compartilhada (UMA ou NUMA) e é baseada na metodologia FORK-JOIN.



HelloWorld em OpenMP:

```
01.  /* Olá Mundo não tão simples
02.         exemplo 01.c
03.
04.     compilação:
05.        gcc -fopenmp 01.c -o 01.x
06.  */
07.  #include <stdio.h>
08.  #include "omp.h"
09.
10.  int main()
11.  {
12.      //omp_set_num_threads(8);
13.      #pragma omp parallel
14.      { // (fork implícito)
15.          int ID = omp_get_thread_num();
16.          // ID é variável privada de cada thread
17.
18.          printf(" Olá %d", ID);
19.          printf(" Mundo %d\n", ID);
20.      } // (join implícito)
21.      return 0;
22.  }
```

Quadro 01: Exemplo 01.c

No exemplo do Quadro 01 destaca-se as seguintes linhas:

`linha08` → #include “omp.h” → inclusão da API
`linha13` → #pragma omp parallel → diretiva de compilação que indica seção paralela (FORK)
`linha15` → omp_get_thread_num() → chamada a função da API que retona o número da thread
`linha20` → existe uma sincronização implícita entre as threads no final do bloco “}”

Para compilar o código em C apresentado no Quadro 01 em ambiente Linux faça:

```
$: gcc -fopenmp 01.c -o 01.x
```

A saída para a execução pode variar entre máquinas, pois não foi configurado o número de threads, desta foma, a API sempre irá configurar para o número máximo de núcleos lógicos.

Caso deseja-se alterar o número de threads, basta chamar a função antes da seção paralela:

```
12.      //omp_set_num_threads(8);
```

Um exemplo de saída é apresentado no Quadro 02 abaixo, a máquina em questão contém 4 núcleos lógicos, porém o número de threads foi configurado para 8.

```
$: ./01.x
Olá 1 Mundo 1
Olá 0 Mundo 0
Olá 6 Mundo 6
Olá 4 Mundo 4
Olá 5 Mundo 5
Olá 2 Mundo 2
Olá 3 Mundo 3
Olá 7 Mundo 7
$:
```

Quadro 02: Exemplo de saída para 01.c configurado para 8 threads.

Vale observar que a ordem de execução das threads não foi sequencial, pois depende do escalonador de processos do Sistema Operacional e também do número limitado de núcleos da máquina teste.

A ideia do OpenMP é dividir o trabalho de processamento entre as threads, isso implica em separar os dados a serem processados. Uma informação importante do Quadro 01 está na linha 15: a definição da variável inteira ID é local para cada thread, pois foi declarada dentro do bloco paralelo. As variáveis declaradas anteriormente ao bloco paralelo são todas compartilhadas entre as threads por padrão.

Caso o programador desejar alterar o comportamento das variáveis, decidindo quais são privadas ou compartilhadas, é possível usar algumas cláusulas na diretiva #omp, são elas:

- `private(list)` → todas as variáveis listadas se tornam privadas e não inicializadas
- `firstprivate(list)` → todas as variáveis listadas se tornam privadas e inicializadas com o valor atual
- `shared(list)` → todas as variáveis listadas são compartilhadas (configuração padrão)

Exemplo para private e firstprivate é apresentado no Quadro 03:

```
01.  /* Olá Mundo não tão simples
02.         exemplo 02.c
03.
04.     compilação:
05.        gcc -fopenmp 02.c -o 02.x
06.  */
07.  #include <stdio.h>
08.  #include "omp.h"
09.
10.  int main()
11.  {
12.      int v5 = 5; // inicialmente compartilhada
13.      int v55 = 55;
14.      omp_set_num_threads(8);
15.      #pragma omp parallel private(v5) firstprivate(v55)
16.      {
17.          int ID = omp_get_thread_num();
18.          // ID é var privada de cada thread
19.
20.          printf(" thread %d, v5=%10d\tv55=%10d\n", ID, v5, v55);
21.          v5 = ID;
22.          v55 = ID;
23.      }
24.      printf(" JOIN:     v5=%10d\tv55=%10d\n", v5, v55);
25.      return 0;
26.  }
```

Quadro 03: Exemplo para private e firstprivate, arquivo 02.c

No Quadro 3, entre as linhas 16 e 23, as variáveis v5 e v55 são privadas, sendo essa última inicializada com o valor anterior da seção paralela, isso significa que as duas variáveis são copiadas para as regiões de memória local de cada thread, e a v55 é inicializada com o valor 55.

Ao final da seção paralela, linhas 21 e 22, as duas variáveis são alteradas, porém, após a seção paralela (linha 23), elas são destruídas e não afetam os valores das variáveis v5 e v55 declaradas nas linhas 12 e 13.

Um exemplo de saída para o código do Quadro 03, executado com 2 threads e apresentado no Quadro 04 abaixo:

```
$: ./02.x
thread 0, v5=         0 v55=        55
thread 1, v5=     32587 v55=        55
JOIN:     v5=         5 v55=        55
$:
```

Quadro 04: Exemplo de saída para o código do Quadro 03.

Próximo passo é realmente dividir uma tarefa entre as threads. O código mostrado no Quadro 05 é uma inicialização de vetor de inteiros com os “nomes” (IDs) das theads trabalhadoras.

```
01.  /* Olá Mundo não tão simples
02.         exemplo 03.c
03.
04.     compilação:
05.        gcc -fopenmp 03.c -o 03.x
06.  */
07.  #include <stdio.h>
08.  #include <stdlib.h>
09.  #include "omp.h"
10.
11.  #define MAX 20
12.
13.  int main(){
14.      int *vet = malloc(10 * sizeof(int));
15.      int i;
16.
17.      omp_set_num_threads(3);
18.      #pragma omp parallel shared(vet, i)
19.      {
20.          int ID = omp_get_thread_num();
21.          #pragma omp for
22.          for(i = 0; i < MAX; i++)
23.              vet[i] = ID;
24.      }
25.      for(i = 0; i < MAX; i++)
26.          printf("%d ", vet[i]);
27.      printf("\n");
28.      return 0;
29.  }
```

Quadro 05: código exemplo para divisão de tarefas entre threads, arquivo 03.c

Importante notar o uso da diretiva #pragma omp for na linha 21, dentro da seção paralela (linhas 18 até 24). Essa diretiva indica que o laço FOR em seguida deve ter sua variável de controle e dados compartilhados entre as threads trabalhadoras (dividir para conquistar). A saída do código do Quadro 05 é apresentado abaixo:

```
$: ./03.x
0 0 0 0 0 0 0 1 1 1 1 1 1 1 2 2 2 2 2 2
$:
```

Quadro 06: saída do código do Quadro 05 para 3 threads.

Importante: Como a divisão não é exata (20/3 = 6,66), as primeiras duas threads assumem 7 elementos cada uma, e a terceira thread processa apenas 6, sendo esta divisão feita pela própria API.

Um problema que pode acontecer com variáveis shared é a escrita indiscriminada das threads, por exemplo, a soma dos elementos de um vetor. Considere o Quadro 07, que apresenta o trecho de código adicional ao código apresentado no Quadro 05 (entre as linhas 29 e 34).

```
29.      int sEscalar = 0;
30.      #pragma omp parallel for shared(vet, i) // reduction(+:sEscalar)
31.      for(i = 0; i < MAX; i++)
32.          sEscalar += vet[i];
33.
34.      printf("sEscalar %d\n", sEscalar);
```

Quadro 07: Trecho de código do arquivo 03b.c

Os resultados variam de execução para execução, pois existe um problema de acesso à variável compartilhada sEscalar. Para evitar o problema, usa-se a cláusula reduction, que recebe como parâmetros o operador da redução, seguido do caractere ':' e da variável que se deseja atribuir o resultado. Remova o comentário da cláusula reduction(+:sEscalar) e tudo estará funcionando corretamente.

O Quadro 08 apresenta um trecho de código de inicialização de matriz utilizando threads:

```
21.      omp_set_num_threads(3);
22.      #pragma omp parallel shared(vet, i) private(j)
23.      {
24.          int ID = omp_get_thread_num();
25.          #pragma omp for
26.          for(i = 0; i < MAX; i++)
27.              for(j = 0; j < MAX; j++)
28.                  vet[i][j] = ID;
29.      }
```

Quadro 08: trecho de código para inicialização de matriz, arquivo 04.c.

O detalhe desse código é que somente a variável de controle do laço externo é compartilhada e será dividia entre as threads devido a cláusula #pragma omp for (que está dentro da seção parallel), a variável de controle j é privada de cada thread. O exemplo de saída para o código do Quadro 08 utilizando 3 threads em uma matriz 6x6 é apresentado no Quadro 09 abaixo:

```
$: ./04.x
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
1 1 1 1 1 1
2 2 2 2 2 2
2 2 2 2 2 2
$:
```

Quadro 09: exemplo de saída para o código do Quadro 08, arquivo 04.c

A captura do tempo de processamento de um programa executando pode ser feita utilizando o comando time do linux. Porém, para melhor análise de resultados, é preferível (altamente recomendado) a utilização da função de captura de tempo do próprio OpenMP:

```
double omp_get_wtime(void);
```

O Quadro 10 apresenta um trecho de código que calcula o tempo parcial de uma operação usando threads.

```
27.      double start_time, run_time;
         ...
33.      start_time = omp_get_wtime();
         ... // região de códido que se deseja capturar tempo
44.      run_time = omp_get_wtime() - start_time;
45.      printf("Tempo de alocação: %lf s\n", run_time);
```

Quadro 10: captura de tempo de execução utilizando omp_get_wtime(), código 05.c

Importante, OpenMP é muito mais do que o apresentado neste documento, que é apenas uma leve introdução à paralelismo.

Tudo o que você precisa saber sobre OpenMP pode ser encontrado no link:

https://hpc-tutorials.llnl.gov/openmp/

Lawrence Livermore National Laboratory - HPC

Atividade Prática

Criar um algoritmo de execute a soma das diferenças dos quadrados de 2 matrizes A e B, carinhosamente chamado aqui de SDQM, com valores em double.

Exemplo de SDQM para duas matrizes 2x2:

Matriz A:

5.0	8.0
9.0	3.0

Matriz B:

8.0	5.0
2.0	2.0

Passo 01: Calcular os quadrados de cada elemento:

Matriz A^2:

25.0	64.0
81.0	9.0

Matriz B^2:

64.0	25.0
4.0	4.0

Passo 02: Calcular as diferenças e armazenar na matriz D = A^2 - B^2:

Matriz D:

-39.0	39.0
77.0	5.0

Passo 03: Somar as diferenças da matriz D

Resultado:	82.0

Requisitos:

- Inicialmente deve ser criado o código sequencial.
- Testado e validado com matrizes pequenas (4x4 ou 6x6). A validação pode ser realizada utilizando saída para arquivo .csv em conjunto com a função matricial SOMAX2DY2 encontrado no excel do LibreOffice (Calc).
- Após a validação, deve ser executado com matrizes de dimensões $m_x m$ que necessitem um tempo de execução maior do que 10 segundos.
- Após encontrar o valor de m, crie o código paralelo.
  - Teste e valide com matrizes pequenas.
- Após validado o código paralelo, é hora de tomar o tempo médio de 10 execuções utilizando 1, 2, 4 e 8 threads, podendo ser realizado com número maior de threads.
- Importante: Os testes devem ser executados em uma mesma máquina com no mínimo 4 núcleos reais.
  - Tanto o sequencial, quanto o paralelo!
- Calcule o speedup e eficiência de todos as médias dos testes.
- Escreva um arquivo .pdf contendo uma discussão sobre os resultados obtidos
  - não esqueçam dos gráficos de speedup e eficiência.

Entrega e apresentação:

- equipes com 1, 2 ou 3 alunos.
- por e-mail para edmar.bellorini@gmail.com
  - relatório .pdf e arquivos fontes
- prazo: dia 11/08, 12h
- cada apresentação para os integrantes do grupo.
- Importante: alunos que elaboraram o trabalho final da disciplina de Programação Paralela poderá entregá-lo, substituindo este trabalho.