Bacharelado em Ciência da Computação
Colegiado de Ciência da Computação - CCET
UNIOESTE – Universidade Estadual do Oeste do Paraná Campus Cascavel - Rua Universitária, 1619

Disciplina: Organização e Arquitetura de Computadores **Data**: 06/02/2017
Série: 3 **Relatório**: Prática 05
Nome: Welcome to parallel programming using openMP – Introdução

---

Objetivos:

Discutir conceitos básicos sobre processamento paralelo;
Discutir conceitos básicos da programação paralela em ambientes SMP;
Apresentar a API openMP.

INTRODUÇÃO

Tradicionalmente, o computador tem sido visto como uma máquina sequencial. A maioria das linguagens de programação de computadores requer que o programador especifique algoritmos como uma sequências de instruções, onde os processadores as executarão obedecendo essa sequência. A medida que a tecnologia computacional evoluiu e o custo de hardware reduziu, os projetistas procuraram mais e mais oportunidades para paralelismo, normalmente para melhorar o desempenho e também a disponibilidade.

TIPOS DE SISTEMAS DE PROCESSADORES PARALELOS (Taxonomia básica de FLYNN)

A taxonomia de FLYNN é a forma mais comum de categorizar sistemas com capacidades paralelas:

- SISD (*Single Instruction, Single Data*): Um processador único (PU) executa uma única sequência de instruções (IP) para operar nos dados armazenados (DP) em uma única memória.
  - Exemplo: todos os uniprocessadores (qualquer processador pessoal do século XX)
- SIMD (*Single Instruction, Multiple Data*): Uma única instrução de máquina controla a execução simultânea de uma série de elementos de processamento em operações básicas. Cada elemento de processamento possui uma memória de dados associada, então cada instrução é executada em um conjunto diferente de dados por processadores diferentes.
  - Exemplo: processadores gráficos
- MISD (*Multiple Instruction, Single Data*): Modelo teórico. Uma sequência de dados é transmitida para um conjunto de processadores, onde cada um executa uma sequência de instruções diferentes.
  - Exemplo: Nenhum exemplo comercial aplicável, pois é um modelo teórico.
- MIMD (*Multiple Instruction, Multiple Data*): Um conjunto de processadores que executam sequências de instruções diferentes simultaneamente em diferentes conjuntos de dados.
  - SMP (*Simetric Multi Processor*): Memória Compartilhada entre os processadores.
    - Exemplo: Intel Xeon E7 8890v4 (12c24t - $ 7.100,00)
  - NUMA (*Non-Uniform Memory Access*) vs UMA (*Uniform Memory Access*)

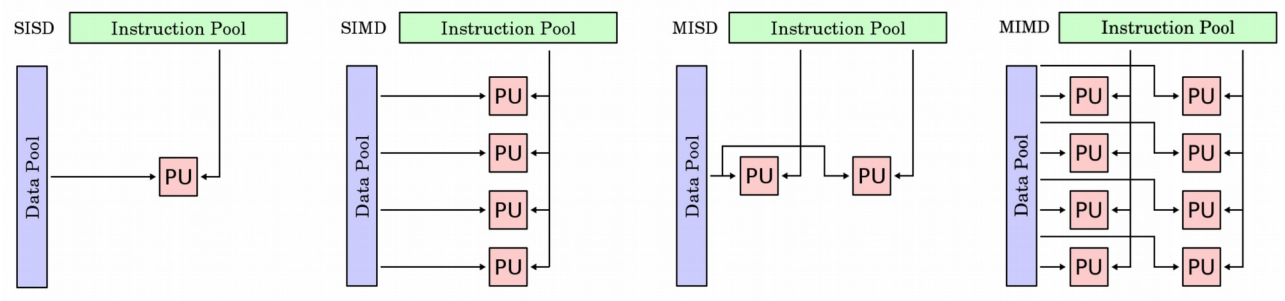


Figura 1: Taxonomia de FLYNN ; Wikipedia

CARACTERÍSTICAS DOS MULTIPROCESSADORES SIMÉTRICOS SMP:

- Há dois ou mais processadores semelhantes que são capazes de desempenhar as mesmas funções;
- Compartilhamento de memória principal (tempo de acesso semelhante) e recursos de E/S;
- Processadores usam os mesmos canais de comunicação entre dispositivos;
- Todo o sistema é controlado por um SO integrado que fornece interação entre processadores e seus programas em nível de trabalhos, tarefas e arquivos ou elementos de dados.

VANTAGENS POTENCIAIS DE UM SMP:

- **Desempenho**: Se o trabalho a ser feito por um computador pode ser organizado de tal forma que algumas partes do trabalho possam ser feitas em paralelo, então um sistema com vários processadores vai atingir um desempenho melhor do que um uniprocessador.
- **Disponibilidade**: Se existir uma falha em um processador, a existência de um outro processador simétrico fará o sistema continuar, porém com desempenho reduzido.
- **Crescimento** Incremental: Pode-se melhorar o desempenho adicionando novos processadores simétricos.
- **Escalabilidade**: Fornecedores podem disponibilizar uma série de produtos com diferentes preços e características de desempenho com base no número de processadores configurados.

ORGANIZAÇÃO GERAL DE UM SMP:

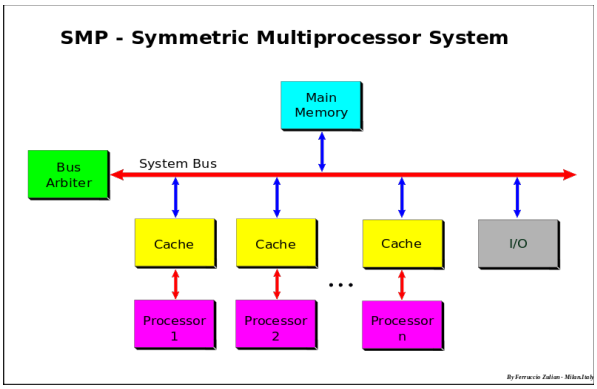


Figura 2: Organização SMP, Wikipedia

- Dois ou mais processadores autosuficientes
  - UC, ULA, Registradores e caches
- Cada Proc tem acesso à memória compartilhada e aos dispositivos de E/S
  - Algum mecanismo de interconexão
- Comunicação entre processadores diretamente ou via memória
- A memória é organizada de tal forma que seja o possível acessos simultâneos aos seus blocos.
- Normalmente usa-se o barramento de tempo compartilhado
  - Problema de desempenho. Barramento comum para transferência de dados.

POR QUE SMP?

Para tentar responder essa pergunta, considere a tabela a abaixo:

Processador	Litografia	Núcleos/ Threads	Clock	Potência	Tensão máx.	Tamanho	Transistores	Data
Pentium 4 511	90nm	1C/1T	2.8 GHz	84W	1.4V	112mm^2	125 M	2005 Q1
Pentium D 820	90nm	2C/2T	2.8 GHz	95W	1.4V	206mm^2	230 M	2005 Q2

Fonte: ark.intel

Que tipo de informação pode-se tirar desta tabela?

TL;DR: dois núcleos mais simples é mais fácil de construir e gerenciar e com o bonus de consumir menos se comparado com um núcleo grande e complexo. (fim TL;DR)

O processador Pentium D 820 é fabricado no mesmo processo de litografia do Pentium 4 511, com a mesma frequência, mesma tensão máxima, porém com quase o dobro de transistores e, consequentemente, quase o dobro do tamanho, porém, com apenas 11% a mais de potência consumida.

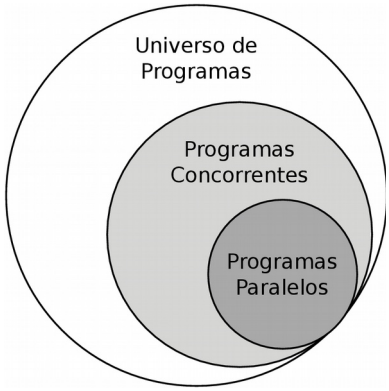
Provavelmente, os projetistas da Intel chegaram à um teto de potência com a família de processadores Pentium 4, e viram que não era viável continuar aumentando sua complexidade. Alternativamente, esses projetistas optaram por construir um processador com dois núcleos mais simples, que consomem uma quantidade menor de potência, e mesmo com o *overhead* de hardware para os controladores dos núcleos, chegou-se em uma potência mais baixa do que um provável Pentium 4 teórico com 230 M de transistores em um núcleo mais complexo.

E isso virou uma tendência.

PROGRAMAÇÃO PARALELA

Considerando que os SMPs 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.

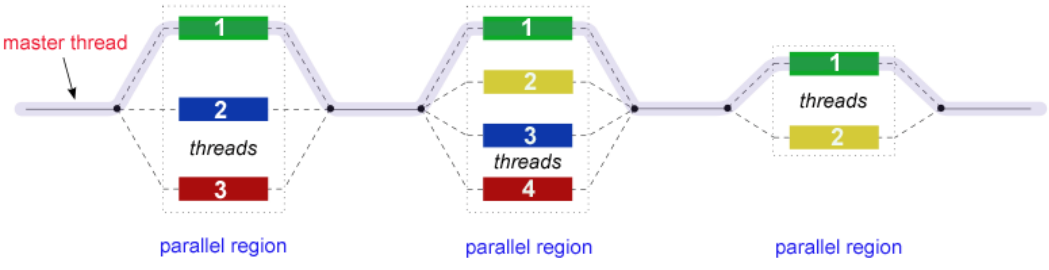
Desta forma, o programador deve criar seu código, normalmente serial, identificar a região que é concorrente e, a partir desta região, identificar o que pode ser paralelizado. E esta identificação é que determina se um programa paralelo será eficiente ou não, pois o ato de codificar é apenas a etapa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

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          #pragma omp parallel
12          { // (fork)
13
14          int ID = omp_get_thread_num();
15          // ID é var privada de cada thread
16                printf(" Olá %d", ID);
17                printf(" Mundo %d\n", ID);
18          }     // (join)
19          return 0;
20    }
```

Quadro 01: Exemplo 01.c

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

`l08` → `#include “omp.h”` → inclusão da API

`l11` → `#pragma omp parallel` → diretiva de compilação que indica seção paralela (FORK)

`l14` → `omp_get_thread_num()` → chamada a função da API que retona o número da thread

`l18` → `}` → Existe uma sincronização implícita entre as threads

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:

```
omp_set_num_threads(int nthreads);
```

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    $: ./01.x
02     Olá 1 Mundo 1
03     Olá 0 Mundo 0
04     Olá 6 Mundo 6
05     Olá 4 Mundo 4
06     Olá 5 Mundo 5
07     Olá 2 Mundo 2
08     Olá 3 Mundo 3
09     Olá 7 Mundo 7
10    $:
```

Quadro 02: Exemplo de saída para 01.c

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 idéia 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 14: 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    #include <stdio.h>
02    #include "omp.h"
03
04    int main(){
05       int v5  = 5;
06       int v55 = 55;
07       omp_set_num_threads(2);
08          #pragma omp parallel private(v5) firstprivate(v55)
09          {
10                int ID = omp_get_thread_num();
11                // ID é var privada de cada thread
12
13                printf(" thread %d, v5=%10d\tv55=%10d\n", ID, v5, v55);
14                v5 = ID;
15                v55 = ID;
16          }
17          printf("JOIN: v5=%10d\tv55=%10d\n", v5, v55);
18          return 0;
19    }
```

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

No Quadro 3, entre as linhas 09 e 16, 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 14 e 15, as duas variáveis são alteradas, porém, após a seção paralela (linha 16), elas são destruídas e não afetam os valores das variáveis v5 e v55 declaradas nas linhas 05 e 06.

Um exemplo de saída para o código do Quadro 03, executado com 2 threads e apresentado abaixo:

```
01    $: ./02.x
02     thread 0, v5=         0  v55=        55
03     thread 1, v5=     32587  v55=        55
04     JOIN:     v5=         5  v55=        55
05    $:
```

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    #include <stdio.h>
02    #include <stdlib.h>
03    #include "omp.h"
04
05    #define MAX 20
06
07    int main(){
08       int *vet = malloc(10 * sizeof(int));
09       int i;
10
11       omp_set_num_threads(3);
12           #pragma omp parallel shared(vet, i)
13           {
14                  int ID = omp_get_thread_num();
15                  #pragma omp for
16                  for(i = 0; i < MAX; i++)
17                         vet[i] = ID;
18           }
19       for(i = 0; i < MAX; i++)
20          printf("%d ", vet[i]);
21       printf("\n");
22       return 0;
23    }
```

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

Importante notar o uso da diretiva #pragma omp for na linha 15, dentro da seção paralela (linhas 12 até 18. 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:

```
01    $: ./03.x
02    0 0 0 0 0 0 0 1 1 1 1 1 1 1 2 2 2 2 2 2
03    $:
```

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

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 21 e 22).

```
22    int sEscalar = 0;
23    #pragma omp parallel for shared(vet, i) //reduction(+:sEscalar)
24    for(i = 0; i < MAX; i++)
25          sEscalar += vet[i];
26
27    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:

```
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();
      ...   // operação 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://computing.llnl.gov/tutorials/openMP/

ANÁLISE DE DESEMPENHO

Um dos cálculos mais básico para avaliação de desempenho é o *speedup*:

$$speedup = \frac{tempo\ de\ execução\ antes\ da\ melhoria}{tempo\ de\ execução\ depois\ da\ melhoria}$$

Por exemplo, um código sequencial de multiplicação de matrizes leva 5 segundos para executar. Após uma melhoria no código que aproveita melhor a memória cache do sistema, o código passa a levar 2 segundos para executar, assim:

$$speedup = \frac{5}{2} = 2{,}5$$

O valor do speedup = 2,5 para esse exemplo indica que o código com a melhoria é 2,5x mais rápido do que o código sem a melhoria.

O speedup é genérico e pode ser usado para avaliar diversos tipos de melhorias em códigos.

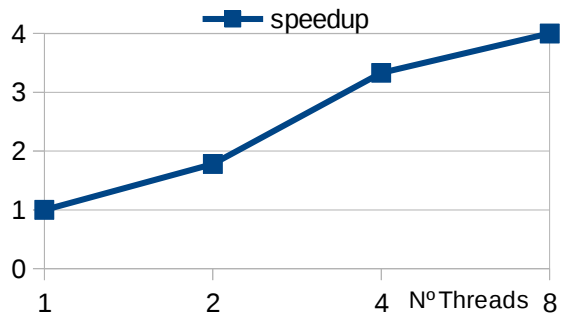
Para avaliar a escalabilidade de um programa paralelo, isto é, se um programa se beneficia de maneira aceitável de várias threads, usa-se o cálculo da *eficiência*:

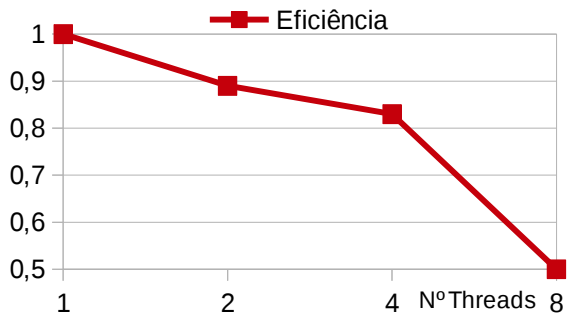
$$Ef = \frac{speedup}{número\ de\ threads}$$

Considere a tabela abaixo que mostra os tempo de execução de um programa qualquer utilizando 1, 2, 4 e 8 threads, speedup e eficiência:

Num Threads:	1	2	4	8
Tempo (s):	10	5,6	3	2,5
Speedup:	1	1,78	3,33	4
Ef:	1	0,89	0,83	0,50

Os valores de eficiência indicam que para 2 e 4 threads, o programa escala razoavelmente bem, com aproximadamente 85%, já com 8 threads, a escalabilidade é ruim, pois foi de apenas 50%. Indicando que, ou existe algum problema no algoritmo, ou o problema executado com o grupo de dados não se beneficia de 8 threads, ou ainda, o pior caso, com 8 threads existe a competição por recurso da máquina, por exemplo, um processador de 4 núcleos executando com 8 threads.





EXERCÍCIO

Grupo de no máximo 3 alunos.

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

O que deve ser feito:

- Inicialmente deve ser criado o código sequencial.
- Testado e validado com matrizes pequenas (4x4, 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 $\mathbf{m_x m}$ 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 6 execuções utilizando 1, 2 e 4 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.

CRONOGRAMA:

A aula do dia 20/02 13h20m é destinada para tirar dúvidas do exercício.

A entrega deve acontecer até dia 21/03.

REFERÊNCIAS

A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey and R. W. Brodersen, "Optimizing power using transformations," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, no. 1, pp. 12-31, Jan 1995.

Barney, Blaise. OpenMP. Lawrence Livermore National Laboratory. Acessado em 10/02/2017. Endereço: https://computing.llnl.gov/tutorials/openMP/#Introduction

Intel. Especificação de Produtos
Pentium 4 511. Acessado em 10/02/2017. Endereço: http://ark.intel.com/pt-br/products/27452/Intel-Pentium-4-Processor-511-1M-Cache-2_80-GHz-533-MHz-FSB
Pentium D 820. Acessado em 10/02/2017. Endereço: http://ark.intel.com/Pt-Br/products/27512/Intel-Pentium-D-Processor-820-2M-Cache-2_80-GHz-800-MHz-FSB

Mattson, Tim. Introduction do Parallel Programming. Videos Educacionais. Acessado em 10/02/2017. Endereço: https://youtu.be/nE-xN4Bf8XI?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG. Intel University Program Office.

Silva, Fernando. Introdução ao OpenMP. Departamento de Ciência de Computadores. Universidade do Porto. Portugal. Acessado em 10/02/2017. Endereço: https://www.dcc.fc.up.pt/~fds/aulas/PPD/0708/intro_openmp-1x2.pdf

Weingaertner, Daniel. CI316 – Programação Paralela – Disciplina de graduação/pós-graduação. 2014. Acessado em 10/02/2017. Endereço: http://web.inf.ufpr.br/danielw/cursos/ci316-20141/ci316-20141

Stallings, William. Arquitetura e Organização de Computadores. 8ª Ed. São Paulo. Person Pratice Hall. 2010