Trabalho

- Implementar o programa das próximas transparências usando MPI em C. Os programas devem ser executados no cluster do IME.
- Depositar os arquivos fontes no classroom e um relatório.
- O relatório deve conter a descrição dos algoritmos desenvolvidos e gráficos de speedup, que é um gráfico de linha (x = número de processadores, y = speedup).
- Data de entrega: **21 NOV**

Trabalho

- Nos próximos slides é apresentado um programa sequencial que calcula a quantidade de números primos entre 1 e um valor inteiro *N*. O programa verifica verifica se *N* é divisível por algum número ímpar entre 1 e a raiz quadrada de *N*, sendo que os números pares são descartados.

Programa Números primos

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int primo (long int n) {
long int i;

   for (i=3; i < (int)(sqrt(n)+1); i+=2)
      if ((n%i) == 0)
         return 0;
   return 1;
}

int main(int argc, char *argv[]) {

int total=0;
long int i, n;
clock_t inicio, fim;
```

Programa Números primos

```
   if (argc < 2) {
      printf("Valor invalido! Entre com o valor do maior
inteiro\n");
      return 0;
   } else {
      n = strtol(argv[1], (char **) NULL, 10);
   }

   inicio = clock();
   for (i=3; i <= n; i += 2)
      if (primo(i) == 1) total++;

   total += 1; /* Contabiliza o 2 que é primo */
   fim = clock();
   printf("Quant. de primos entre 1 e n: %d\nTempo: %.2lf
mseg\n", total, ((double) (fim-inicio)) / CLOCKS_PER_SEC);
   return 0;
}
```

Trabalho

- Desenvolver duas versões paralelas do programa.
- As duas versões devem ser escritas na linguagem C usando MPI.
- Deve-se medir o tempo de execução utilizando a função MPI_Wtime.
- As duas versões devem ser executadas variando o número de processadores de 2 a 16 processadores (passo 1).
- A versão sequencial deve ser executada para medir o tempo a fim de calcular o *speedup*.
- Executar para os dois valores de *N:* 100.000.000 e 1.000.000.000.
- Gerar um gráfico de *speedup* para cada versão do programa. Cada gráfico deve conter duas linhas correspondendo ao *speedup* das duas entradas.

Trabalho

- Primeira versão paralela:
  - Uma distribuição em lote, passando uma faixa contínua de valores a serem verificados por cada processo, deve ser descartada pois gera desbalanceamento de carga.
  - Solução: cada processo começa verificando um número ímpar diferente, saltando (*num_procs* * *2*) números entre cada número ímpar verificado. Assim, todos os processos irão avaliar tanto números pequenos quanto números grandes.
  - Utilize MPI_Reduce para obter o total de números primos calculado por cada processo.

Trabalho

- Segunda versão paralela:
  - A execução da primeira versão com um número de processos que não é uma potência exata de 2, gera uma perda de *speedup*, devido a desbalanceamento de carga.
  - Solução: usar um método de balanceamento dinâmico denominado “saco de tarefas”. Um processo (mestre) fica responsável por enviar as tarefas para os demais processos (trabalhadores). Assim que uma tarefa é terminada e o resultado é enviado para o mestre, uma outra tarefa é alocada para o trabalhador.

Trabalho

- Segunda versão paralela:
  - Para evitar uma sobrecarga devido a comunicação, é necessário que cada tarefa tenha um tamanho mínimo. Tente distribuir 500.000 números por vez.
  - Nesta versão é necessário utilizar MPI_Send e MPI_Recv.
  - O mestre deve informar aos trabalhadores quando as tarefas terminarem para que todos os processos terminem.