

**IFES - INSTITUIÇÃO FEDERAL DO ESPÍRITO SANTO**

# **Trabalho de Pthreads**

João Victor Ferrareis Ribeiro  
Renzo Fraga Loureiro Marinho

**Serra, ES**

**2023**

# Sumário

1. Introdução	3
2. Testes e Resultados	3
Teste 1	3
Teste 2	4
Teste 3	5
Teste 4	6
Teste 5	7
Teste 6	7
Analisar as diferenças de desempenho de x64 e x87	8
3. Conclusão	8

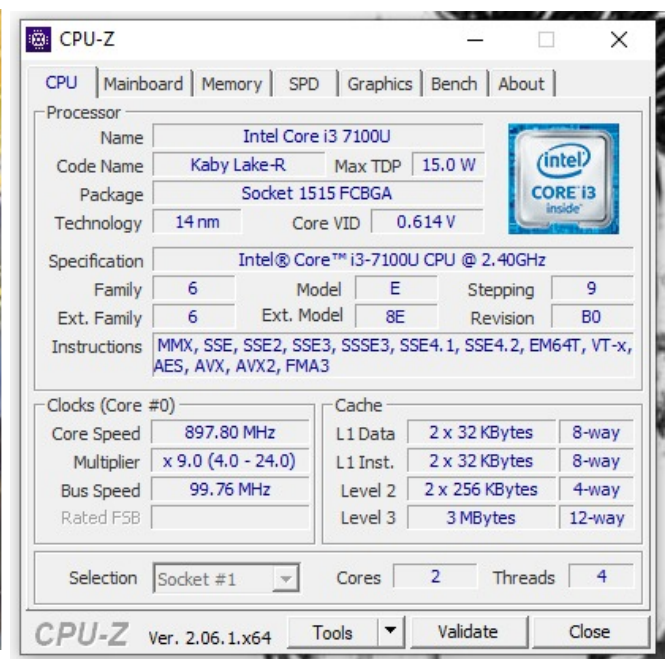
# 1. Introdução

Com o aumento da demanda por desempenho computacional em diversas áreas, como inteligência artificial, processamento de dados em larga escala e simulações complexas, a programação multi paralela se tornou fundamental para explorar o potencial máximo dos processadores multicore, aceleradores como GPUs e sistemas distribuídos.

O objetivo do trabalho realizado foi implementar um programa que fizesse a leitura de uma matriz de números naturais aleatórios (intervalo 0 a 31999) e contabilizar quantos números primos existem e o tempo necessário para isso. Isso seria feito tanto de modo serial quanto de modo paralelo, com o intuito de mostrar os efeitos da aplicação da programação Multithread.

## 2. Testes e Resultados

Os processadores usados nesses testes são:



### Teste 1: Rodar o Código Serial

Computador 1:

Consulta Serial	
Tempo marcado - x64	20,528000
Tempo marcado - x86	29,514000

Computador 2:

Consulta Serial	Matriz 15000x15000	
1x1	66.883000	
10x10	64.717000	
100x100	63.579000	
500x500	63.836000	
1000x1000	61.523.000	


Os dados de consulta serial coletados no Computador 1 e 2 atestam o seguinte:

Ambos os testes demonstram um enorme tempo de execução, devido ao fato de que utilizam-se de apenas um núcleo lógico da máquina para desempenhá-lo. No caso do Computador 1 esse tempo de execução é menor porque seu processador aguenta frequências maiores.


Já no Computador 2, foi feita uma coleta de dados baseada em tamanho de macrobloco, e foi percebido que o tamanho de macrobloco influencia diretamente no desempenho do programa. Muitos macroblocos implicam em mais consultas à memória, e, portanto, intensificam o problema do gargalo de Von Neumann.

## Teste 2: Rodar o Código Paralelamente (4 Threads)


Computador 1 - x64:

Num. Threads x Tamanho do Macrobloco									
	1 x 1	10 x 10	50 x 50	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
4	59,188000	6,468000	7,150000	6,650000	7,476000	7,205000	7,973000	8,171000	

Computador 1 - x86:

Num. Threads x Tamanho do Macrobloco									
	1 x 1	10 x 10	50 x 50	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
4	69,995000	7,984000	10,077000	8,450000	10,132000	10,088000	10,491000	11,691000	

Computador 2 - x86:

Num. Threads x Tamanho do Macrobloco								
	1 x 1	10 x 10	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
4	174,430000	30,458000	47,138000	29,803000	32,609000	29,869000	33,693000	

Logo de cara, a testa-se que Macroblocos 1x1 atrasam a execução do programa, já que o esforço computacional de se criar uma thread, fazer o mutex, alocar e desalocar recursos, e todo o resto que envolve rodar um trecho de código paralelizado não vale a pena para uma tarefa computacional tão pequena. Além

disso, consultas em macroblocos tão pequenos criam muitos acessos à memória, o que intensifica o gargalo de von Neumann, tudo isso colabora para um tempo de execução absurdo de lento. Esse atraso gerado pela criação exacerbada de threads para tarefas banais é chamado de Overhead.

Essas considerações valem para qualquer caso de teste paralelo com macrobloco 1x1.

Dito isso, o SpeedUp previsto pela lei de Amdahl para essa paralelização considerando 99% do código como paralelizado, é de 3,88.

Speedup Previsto =  $1 / [(1 - 0,99) + (0,99 / 4)]$

Speedup Previsto =  $1 / [0,01 + 0,2475]$

Speedup Previsto =  $1 / 0,2575$

Speedup Previsto  $\approx 3,88x$


O SpeedUp real observado no computador 1 foi de 3.08x

O SpeedUp real observado no computador 2 para 100 de tamanho de macrobloco foi de 1.34x


Essa diferença de SpeedUp pode se dar por vários fatores, por exemplo, o Computador 2 não tem de fato 4 núcleos, ele tem apenas 2 núcleos físicos, que simulam 4 núcleos lógicos.

## Teste 3: Rodar o Código Paralelamente (8 Threads)


Computador 1 - x64:

Num. Threads x Tamanho do Macrobloco									
	1 x 1	10 x 10	50 x 50	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
8	120,879000	4,525000	4,273000	3,660000	4,090000	4,316000	4,102000	5,865000	


Computador 1 - x86:

Num. Threads x Tamanho do Macrobloco									
	1 x 1	10 x 10	50 x 50	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
8	154,219000	5,862000	5,034000	5,812000	5,684000	5,829000	6,193000	8,294000	

Computador 2 - x64:

Num. Threads x Tamanho do Macrobloco								
	1 x 1	10 x 10	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
8	242,159000	21,058000	20,667000	20,762000	21,308000	21,001000	24,021000	

Computador 2 - x86:

Num. Threads x Tamanho do Macrobloco								
	1 x 1	10 x 10	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
8	270,371000	28,960000	28,501000	28,964000	28,867000	28,867000	32,705000	

Voltando a Analisar as amostras de Macrobloco 1x1, percebe-se que com o aumento das Threads também se intensifica o Overhead, afinal, o sistema operacional precisa gerenciar uma quantidade absurda de Threads, junto da quantidade avassaladora de macroblocos.

Em 8 threads o Computador 2 não entregou um desempenho muito diferente da amostra de 4 threads, e o mesmo comportamento se repetiria caso fosse feita a coleta de uma amostra de 16 threads. Isso acontece porque acima de 4 Threads não importa para o Computador 2 quantas threads você vai colocar, afinal, ele só consegue processar 4 simultaneamente. Já o computador 1 conseguiu entregar um SpeedUp, abaixo segue uma análise disso tomando como referência a lei de Amdahl:

$$\text{Speedup Previsto} = 1 / [(1 - 0,99) + (0,99 / 8)]$$

$$\text{Speedup Previsto} = 1 / [0,01 + 0,12375]$$


$$\text{Speedup Previsto} = 1 / 0,13375$$

$$\text{Speedup Previsto} \approx 7,49$$


Tomando como base a amostra x64 serial, e a amostra x64 paralela de tamanho de macrobloco=100, o SpeedUp real foi de 5,6x

## Teste 4: Rodar o Código Paralelamente (16 Threads)

Computador 1 - x64:

Num. Threads x Tamanho do Macrobloco									
	1 x 1	10 x 10	50 x 50	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
16	200,962000	3,471000	3,090000	3,036000	3,098000	3,162000	3,588000	3,455000	

Computador 1 - x86:

Num. Threads x Tamanho do Macrobloco									
	1 x 1	10 x 10	50 x 50	100 x 100	500 x 500	1000 x 1000	2500 x 2500	5000 x 5000	Graficos
16	285,113000	4,366000	3,749000	3,751000	3,829000	3,826000	4,421000	5,333000	

O Computador 1 mais uma vez mostra um ganho grande de desempenho.

$$\text{Speedup Previsto} = 1 / [(1 - 0,99) + (0,99 / 16)]$$

$$\text{Speedup Previsto} = 1 / [0,01 + 0,061875]$$

Speedup Previsto =  $1 / 0,071875$

Speedup Previsto  $\approx 13,91$

Tomando como base os dados da Consulta Serial x64 e da Consulta paralela (16 threads) x64 de tamanho de macrobloco 100x100, o SpeedUp Real foi de 5,47x.

## Teste 5: Rodar o Código Paralelamente (500 Threads)

O Computador 2 demonstrou um desempenho análogo aos outros exemplos paralelizados acima de 4 threads, não demonstrou grande aumento de desempenho. Isso se dá porque o processador não aguenta processar mais do que 4 threads simultaneamente, assim, não importa quantas forem criadas, o speedup será o mesmo.

```
C:\Users\usuario\source\repos\trabgigi\x64\Debug\trabgigi.exe
Quantidade de elementos primos na matriz por busca paralela: 24493479
Tamanho da matriz: 15000 X 15000
Tamanho do macrobloco: 100 X 100
Número de threads: 500
Tempo total de contagem: 22.957000 segundos

Pressione qualquer tecla para continuar. . . _
```

O Computador 1 teve comportamento análogo ao Computador 2.

Consulta Paralela - 500 Threads - Macrobloco 100 x 100	
Tempo marcado - x64	3,781000
Tempo marcado - x86	4,804000

## Teste 6: Rodar o Código paralelamente com Mutexes removidos:

```
C:\prog\ws-C\ProjetoSO\x64\
Quantidade de elementos primos na matriz por busca paralela: 24498887
Tamanho da matriz: 15000 X 15000
Tamanho do macrobloco: 100 X 100
Número de Threads: 16
Tempo total de contagem: 3.107000 segundos

Pressione qualquer tecla para continuar. . . |
```

```
C:\prog\ws-C\ProjetoSO\Deb X + v
Quantidade de elementos primos na matriz por busca paralela: 24495708
Tamanho da matriz: 15000 X 15000
Tamanho do macrobloco: 100 X 100
Numero de Threads: 16
Tempo total de contagem: 3.824000 segundos

Pressione qualquer tecla para continuar. . . |
```

Consulta Paralela - 16 Threads - Mutexs desabilitados - Macrobloco 100 x 100	
Tempo marcado - x64	3,107000
Tempo marcado - x86	3,824000

Quando se remove os mutexes, ocorre o que chamamos de Inconsistência de dados, que é quando mais de uma Thread entra em sua região crítica ao mesmo tempo, isso ocorre porque enquanto um thread está lendo a memória, outra a sobrescreve, assim, os dados ficam inconsistentes. Assim, o número de elementos primos calculados está errado, pois ele é armazenado na RC do processo, e seus dados foram comprometidos pela falta de mutexes.

## Analisar a diferença de desempenho entre x86 e x64:

Durante os testes observamos uma diferença entre os resultados em x86 e x64. Isso pode se dar pelo seguinte fator:

-Computadores x86 tem no máximo 4GB de RAM.

Esse fator pode ser melhor observado nos testes do Computador 1, a diferença entre memória dedicada para o processo é gritante entre x86 e x64. (4GB e 8GB respectivamente). No computador 2 esse fator não pode ser contemplado, já que a RAM total dele já é 4GB.

Essa limitação dos computadores x86 se dá pelo fato de que 32 bits correspondem a  $2^{32}$  endereços de memória, isso resulta em 4GB de endereços de memória.

## 3. Conclusão

Em conclusão, a programação multi paralela mostra ser uma grande abordagem para otimizar o desempenho de programas computacionais em arquiteturas modernas. Ao longo do trabalho, vimos como um programa que levaria até minutos para ser realizado e levar poucos segundos para terminar.

Contudo, é importante se considerar o modo em que realizaremos a paralelização, as dependências de dados e a sincronização entre as threads. Um exemplo



presente no trabalho foi quando se teve que usar de macroblocos de pequenas dimensões, o acesso à memória junto ao grande uso dos semáforos acarretou num péssimo desempenho do programa. Em suma, a programação Multithread é muito importante para um melhor desempenho nos processadores modernos, tendo em vista a melhora do desempenho e a eficiência dos programas computacionais em cenários de processamento intensivo e altamente escaláveis.

Muito obrigado por ler, boa noite! 😊