

# Introdução à Programação Multithread com PThreads

## OBJETIVOS

- Introduzir o conceito de programação concorrente com múltiplas threads e seções críticas a serem tratadas.
- Analisar comparativamente o desempenho do algoritmo com diferentes números de threads.

## GRUPOS

Deverão ser formados grupos com 3 pessoas, a serem escolhidos livremente. **Em hipótese alguma este trabalho pode ser feito sozinho(a)!** Respeite as decisões do professor.

## PONTUAÇÃO

O referido trabalho será avaliado de 0 a 100 e corresponderá a 20% da nota semestral. A nota do trabalho corresponderá a média aritmética entre as notas do código e do relatório.

## IMPLEMENTAÇÃO

O objetivo final do algoritmo é: dada uma matriz de números naturais aleatórios (intervalo 0 a 31999) contabilizar quantos números primos existem e o tempo necessário para isso. No entanto, isso será feito de duas formas:

- De **modo serial**, ou seja, a contagem dos primos será feita um a um, um após o outro. Esse será o seu tempo de referência.
- De **modo paralelo**. Para tanto, o trabalho de verificar cada número e se for primo contabilizá-lo consistirá na subdivisão da matriz em “macroBlocos” (submatrizes), sem qualquer cópia, baseando-se apenas nos índices.

Tome, como exemplo (ao lado), uma matriz de 9 x 9. Cada macrobloco é composto por 9 elementos (3 x 3). O macrobloco 1 vai da coluna 0 a 2 e da linha 0 a 2, e assim sucessivamente. Os macroBlocos serão as unidades de trabalho de cada thread (*paralelismo de dados*, lembra?).  
 Atenção: **Nem a matriz nem os macroBlocos deverão ser obrigatoriamente quadradas.** A única exigência é que todos os macroBlocos tenham o mesmo tamanho. Além disso, você deve encontrar alguma forma de PARAMETRIZAR essa divisão (usando a diretiva `#define`) a fim de poder efetuar os testes para diferentes tamanhos de macroBlocos. **Os macroBlocos terão tamanhos que podem variar de desde um único elemento até a matriz toda** (equivalente ao caso serial).

1 2 3	4 5 6	7 8 9
10 11 12	13 14 15	16 17 18
19 20 21	22 23 24	25 26 27
28 29 30	31 32 33	34 35 36
37 38 39	40 41 42	43 44 45
46 47 48	49 50 51	52 53 54
55 56 57	58 59 60	61 62 63
64 65 66	67 68 69	70 71 72
73 74 75	76 77 78	79 80 81

Dito qual é o objetivo final da implementação, segue um passo a passo das diretrizes básicas a serem seguidas pelo algoritmo:

- Geração de uma matriz de números naturais aleatórios (intervalo 0 a 31999) usando uma semente pré-definida no código, a fim de sempre ter a mesma “matriz aleatória” para todos os testes. A geração de números aleatórios em C se dá com o uso das funções `srand()` e `rand()`. **Essa matriz e a variável que contabiliza o número de números primos encontrados na matriz deverão ser globais (logo, compartilhadas) e únicas. Além disso, a matriz deverá ser alocada dinamicamente (tamanho parametrizado via `#define`).** O tamanho da matriz deverá ser consideravelmente grande a fim de que possam ser efetuadas medidas de desempenho consistentes. O tamanho 10000 x 10000 é um bom começo para os computadores modernos. Mas, **atenção** a essas observações:

- a. Muito cuidado na escolha desse tamanho (dimensões da matriz)! Rode os testes no modo `debug` e, com informações do ambiente de programação (o Visual Studio é muito bom nisso!) e do Gerenciador de Tarefas (no caso do Windows), avalie a quantidade de memória consumida. É fundamental que você se atente para o S.O. não estar fazendo uso significativo do armazenamento secundário como extensão da memória principal (“Memória Virtual”), pois, caso isso aconteça, a medição de tempo não será consistente. Um tamanho razoável é algum que leve a um tempo de busca para o caso serial superior a 5 segundos.

- b. Ainda falando sobre as dimensões da matriz, a partir de certo tamanho a quantidade de memória principal consumida será maior que 2 GB e esse é o **limite** para processos de 32 bits no Windows (x86). Isso pode ser facilmente contornado compilando o código nativamente para 64 bits (x64). Novamente, no Visual Studio, isso é um ajuste muito simples feito na própria barra de ferramentas.
- Efetuar a **Busca Serial** pela quantidade de números primos da matriz gerada acima. Exiba a quantidade de números primos encontrados e o tempo decorrido nessa busca.

A verificação se um número é primo ou não deve ser feita por uma função com protótipo `int ehPrimo(int n)`, onde `n` é o número a ser verificado. Se ele for primo, a função retorna 1 (true), caso contrário, 0 (false). Atenção: evite fazer essa função de modo “muito otimizado”. Não se esqueça: Nós precisamos de uma carga de trabalho grande para testarmos o impacto da paralelização do código. Por outro lado, não implemente essa função de modo tão grosseiro. Dica: teste se há divisores até a raiz quadrada do número. No primeiro momento que encontrar um divisor, já conclua que o número não é primo. Se chegar até o teste `sqrt (número)` e não houver divisor, é porque ele é primo. #Obrigado #DeNada.

- Efetuar a **Busca Paralela** pela quantidade de números primos na **mesma matriz** usada na busca serial (e utilizando, também, a mesma função de verificação). Cabem, aqui, várias observações:

- a. A escolha do número de threads para o teste principal deverá levar em conta o **número de núcleos físicos** do processador (CPU) que equipa o computador (Se for o caso de *Multiprocessamento Heterogêneo*, como os núcleos P/E da Intel, considere apenas os núcleos P já que você terá que calcular o *speedup* também). Se não souber dessa informação, procure saber de antemão (Você fez AOC com o Giraldeli né? Favor não decepcionar!). Caso o processador possua SMT/HT (*Simultaneous Multithreading*), teste também a quantidade de threads igual a quantidade de núcleos lógicos/virtuais. Faça uma análise dessa comparação: Quantidade de Threads igual ao número de **núcleos físicos** x Quantidade de Threads igual ao número de **núcleos lógicos/virtuais**, estimando, assim, o ganho proporcionado pelo SMT. Você aprendeu isso em AOC se estudou com o Giraldeli (caso contrário, procure o professor e tire suas dúvidas). Ex: Ryzen 7 5700X é um 8N (16T). Então teste para 1, 8 e 16 threads. 8, neste caso, é a base para calcular o speedup (em relação a 1).
- b. A atribuição de qual macrobloco será processado em cada momento deverá ser da seguinte forma: Suponha que foram criadas 4 threads. A thread 1 deverá começar a busca no macrobloco 1, a thread 2 no macrobloco 2, a thread 3 no macrobloco 3 e a thread 4 no macrobloco 4. Devido à natureza aleatória dos números (consequentemente do tempo de verificação se o número é primo ou não depender da magnitude do número) e do escalonador do sistema operacional, nada se pode afirmar sobre que thread terminará sua busca primeiro. No entanto, digamos que a thread 2 termine sua busca primeiro. Ela deverá: somar o número de primos encontrado à variável global que esteja contabilizando o número total de primos da matriz (ou seja, a thread usará uma variável temporária para contar o número de primos e, após terminada a busca no macrobloco, somará esse valor à variável global) e reiniciar a busca no próximo macrobloco que ainda não foi atribuído a nenhuma thread. **A variável que controla quais macroblocos estão livres/allocated deverá ser global (compartilhada)**. Exemplo: Thread 2 termina. Logo, ela verificará se o macrobloco 5 já foi atribuído a alguma thread. Se não, ela “marca-o” como já atribuído e reinicia seus trabalhos nele. Caso contrário, busca pelo próximo macrobloco “livre”, até que por fim se esgotem os macroblocos a serem buscados. Neste ponto, a thread termina e a thread principal fica esperando que as demais threads terminem.

Aqui cabe uma observação importante: Alguns grupos optam por criar uma outra estrutura de dados armazenando as coordenadas de início de fim de blocos. Isso é uma decisão válida em termos de programação. PORÉM, raciocina comigo: a matriz de testes é gigantesca (vários GB de RAM) e o tamanho dos macroblocos varia, certo? Em testes com macroblocos muito pequenos, guardar as coordenadas dos macroblocos vai, nesse caso, aumentar muito a quantidade de RAM consumida pelo programa, chegando ao ponto (no pior caso, para macroblocos de 1 x 1) da matriz de controle ocupar mais memória que a própria matriz principal, inviabilizando o teste em diversos computadores! Uma solução para isso é não guardar as coordenadas previamente, mas sim calculá-las na hora de atribuir um macrobloco a uma thread.

- c. **ATENÇÃO:** É proibido criar um vetor para armazenar o número de primos encontrados em cada macrobloco e depois somá-los (ao fim). **Como já mencionado, as variáveis que armazenam o número de primos total, a que controla a alocação do macroblocos e matriz principal deverão ser globais e o acesso compartilhado deverá ser controlado. Não fuja das seções críticas.** Trate-as!

- d. Uma vez criadas as N threads, outras não deverão ser criadas. As mesmas threads deverão “buscar trabalho” em outros macroblocos livres, conforme mencionado acima. Quando não houver mais trabalho a ser feito pela thread, permita ela seja encerrada naturalmente.
- e. Faça testes com macroblocos de tamanhos diferentes, entre os extremos: um único elemento e a matriz toda. Anote esses valores, pois serão usados no relatório.
- f. Forneça uma forma prática e fácil de testar o código em *single-thread*, *multi-thread* ou ambos. Você pode fazer isso com um menu simples dentro do código ou mesmo com chamadas muito claras e fáceis de comentar (desativando-as) tais como `BuscaSerial()` e `BuscaParalela()`. O professor precisará fazer todos esses testes ao corrigir seu trabalho. Ademais, permita que os testes *multi* e *single* aconteçam numa única “rodada” do programa, ok? (Ou seja, que eu possa rodar tanto o serial quanto o paralelo na mesma execução). De posse desses tempos, já exiba o **speedup** (Lei de Amdahl) pois ele é o principal parâmetro a ser avaliado no quesito desempenho.

#### Outras observações gerais:

- **Atenção ao fazer a alocação dinâmica de matrizes!** Há um jeito correto de se fazer isso e eu explico tudo na minha apostila de Programação em C (página 113). Eu vou ser rígido quanto a isso pois é o jeito certo a ser feito.
- É **obrigatória** a parametrização das dimensões da matriz principal via `#define`. Ou seja, a **largura** e **altura** da matriz devem ser facilmente modificáveis. O professor precisa ter acesso fácil a esses parâmetros a fim de que os testes possam ser rapidamente feitos. Não complique a vida do professor! (Ou a sua se complicará, como já ocorreu antes). Nos interessa o tamanho dos macroblocos (carga de trabalho das threads) e não exatamente a quantidade deles.
- Forneça uma forma **fácil e direta** de o professor rodar o código serial e paralelo de maneira direta, **um após o outro**.
- **Evite ignorar os warnings dados pelo Visual Studio.** Tente entender do que ele está reclamando e corrigir isso. Essa observação vale especialmente no caso de não verificação de sucesso/insucesso de alocação dinâmica (se o ponteiro é diferente de `NULL`) e em problemas de indireção múltipla (“ponteiro para ponteiro”).
- **Aprenda a usar a ferramenta de debug do Visual Studio.** Ela é muito simples de ser usada e útil demais! Para com esse negócio de ficar usando `printf` (“passou aqui”) em tudo quanto é lugar! (Tá, em algumas situações é interessante sim...).
- **Sempre exiba a quantidade de números primos encontrados.** Isso serve para verificar se o processamento foi correto.
- **Evite colocar cálculos muito complexos dentro** de loops se eles não variarem ao longo do loop (na hora você vai saber a que me refiro). Faça-os fora do loop (`#sqrt`). Em situações como a desse trabalho, a diferença de desempenho é enorme!
- Especificamente neste trabalho, modularize, mas não divida em .c e .h. **Coloque tudo dentro de um único .c**, ok?

## VAMOS FALAR DE PLÁGIO E IA?

Então... todo professor honesto deseja que seus(suas) alunos(as) também tenham honestidade no processo de aprendizagem. Essa honestidade é fundamental para que o professor identifique possíveis deficiências na aprendizagem e possa intervir para tentar solucionar. Em termos mais simples, eu, Giraldeli, preciso saber se você aprendeu ou não os conceitos abordados nesse trabalho. E estou mensurando isso através de uma tarefa (construção de um algoritmo que solucione um problema). Se você abre mão de exercitar seu cérebro e busca soluções mais “fáceis”, perde-se completamente o sentido do trabalho.

Hoje há duas formas básicas de você cometer essa desonestade nesse trabalho: buscando trabalhos de turmas anteriores e/ou consultando diretamente ferramentas de IA (como o chatGPT). Sobre os trabalhos anteriores, é bem simples: **eu tenho TODOS os trabalhos anteriores arquivados e possuo ferramentas de verificação**. Já a questão do uso indiscriminado de **ferramentas de IA**, a coisa é mais complicada, já que é difícil afirmar que você a **usou extensivamente no seu trabalho**, o que, acho quase desnecessário dizer, é **proibido**! O que farei nesse caso é também simples: **caso eu desconfie**, por alguma razão (experiência na disciplina, talvez?), que a sua solução não partiu do seu esforço direto, a dupla será **convocada para uma entrevista presencial**. Nessa entrevista eu farei perguntas e possivelmente solicitarei algumas modificações “ao vivo” no código apresentado. Caso seja observada a incapacidade do grupo diante desse cenário, a nota será considerada zero. Caso o grupo se negue a comparecer nessa entrevista, a mesma decisão será tomada: nota zero.

“Professor... agora você me deixou preocupado(a)!“ 😬 Se você não está sendo desonesto(a), você não tem com que se preocupar. Prometo! Por fim... uma última coisa: isso não é um desafio. Mas, se assim você quiser, eu não vou fugir.

## RELATÓRIO

**Seja criativo!** Monte tabelas, gráficos, etc. Avalie/critique o máximo possível os resultados dos testes. Faça conjecturas e verifique, com testes, se elas estão certas. E, POR FAVOR, mantenha a clareza e organização do relatório! Facilite a correção do professor.

Elabore gráficos relativo ao **tempo de processamento** versus diferentes **números de threads** e **tamanhos de macroblocos**, conforme citado anteriormente. Você encontrará resultados bem interessantes quando o tamanho dos macroblocos for muito grande ou muito pequenos! ;-) Mas, lembre-se de testar isoladamente a influência de cada um desses parâmetros. Igualmente, verifique se há algum ganho/perda quando o número de threads for maior que o número de núcleos de processamento.

Não há necessidade de, no relatório, você ficar “explicando código”. Se tiver que explicar algo do código, opte por fazer isso com comentários no próprio código fonte. Foque-se no que mais importa: nos testes e análises! Mas, para que eu possa avaliar seus testes, **não se esqueça de incluir as configurações básicas dos computadores**, especialmente em termos de CPU e RAM.

### Reforçando: Testes e Análises obrigatórias

- **Fixe o número de threads igual a quantidade de núcleos físicos do processador.** Faça múltiplos testes variando o tamanho dos macroblocos (desde 1x1 até as dimensões da matriz, algo como: 1x1, 10x10, 100x100, 1000x1000, ???, “matriz completa”). Faça um gráfico correlacionando o tamanho dos macroblocos com o tempo de execução multithread. Você verá que existe uma “faixa” de tamanhos de macroblocos que levam a um tempo de execução bastante semelhante. Você pode usar outros tamanhos que eu não falei, mas não se esqueça de testar os muito pequenos, os muito grandes, e uns dois tamanhos intermediários cujo tempo de execução não mude muito (com testes você verá isso facilmente).
- Usando o tamanho de macrobloco adequado (nem muito grande, nem muito pequeno, conforme teste anterior), mensure o *speedup* ao rodar em múltiplas threads. **Fixe esse tamanho de macrobloco** e faça testes com...
  - 2, 4 e 8 threads se o processador for 4 (8) núcleos.
  - 2, 4, 6 e 12 threads se o processador for 6 (12) núcleos.
  - 2, 4, 8 e 16 threads se o processador for 8 (16) núcleos.
  - Ou adapte segundo essa mesma lógica se o seu processador for diferente disso. converse com o professor se tiver um processador Intel com multiprocessamento heterogêneo (você sabe o que é isso, né?).

Analise se esse resultado está coerente com o que a Lei de Amdahl prevê. Mas atenção a diferença entre núcleos físicos e núcleos lógicos (ou virtuais)! O *speedup* deve ser calculado em cada caso e claramente identificado no relatório do trabalho. Adicionalmente, teste em dois computadores bem diferentes, tal qual feito em AOC. Compare o desempenho.

- Aumente **muito** o número de threads (algumas centenas ou mais) a fim de que o overhead possa realmente ficar crítico e analise os resultados. Nesse caso, mantenha fixo o tamanho do macrobloco.
- Remova os mutexes que protegem as Regiões Críticas. Isso mesmo... remova as proteções temporariamente, rode o programa com macroblocos pequenos e macroblocos grandes (por quê?) e observe os resultados.

## CONCLUSÃO

Pronto, chegou a hora de você responder da maneira mais abrangente possível a simples questão:

***O que você pode aprender com esse trabalho?***

Capriche na elaboração da resposta.

## ENTREGA

A data limite para entrega do trabalho, via seção correspondente no AVA, é o dia **01/11/2025 (sábado)** até as **23:59**. Total de dias letivos disponíveis desde essa Especificação: 18 dias (em linha com semestres anteriores).

O trabalho deverá ser entregue comprimido em zip/rar (Relatório em PDF + Arquivo fonte único em .c), com a nomenclatura **[SO 2025-2] Trabalho 1 - Aluno1, Aluno2, Aluno3**. Atenção: É necessário apenas o código fonte, não o projeto todo.

**Siga estritamente as regras acima, ou seu trabalho poderá ser desclassificado. Principalmente as normas de nomenclatura. O objetivo é facilitar a correção/análise por parte do professor.**

*Bom Trabalho!*