Relatório da atividade de mecanismos de controle

Computação Escalável

Dominique de Vargas de Azevedo
Pedro Thomaz Conzatti Martins
Tatiana Lage
Thiago Franke Melchiors

Professor: Thiago Pinheiro de Araújo

Rio de Janeiro
2024

FGV EMAp
ESCOLA DE MATEMÁTICA APLICADA

Conteúdo

1 Introdução

A proposta do exercício é escrever um programa capaz de balancear a carga, utilizando mecanismos de controle para coordenar a execução das threads, protegendo, assim, as regiões críticas, com a finalidade de diminuir o tempo de processamento total. Para resolver o exercício, devemos criar um programa capaz de decidir se um número é primo ou não, com uma função inicial que recebe um número máximo, e, como retorno do programa, um arquivo com os números primos encontrados.

Após a execução do programa, precisamos exibir o número de threads, o tempo total de execução, a quantidade de números avaliados, a quantidade de números primos encontrados e os números primos encontrados. Posteriormente, devemos comparar a execução do programa não balanceado e balanceado, ambos com o número de threads variando de 1 à 10, produzir gráficos com o tempo versus número de threads e discutir os resultados entre os integrantes do grupo.

2 Metodologia

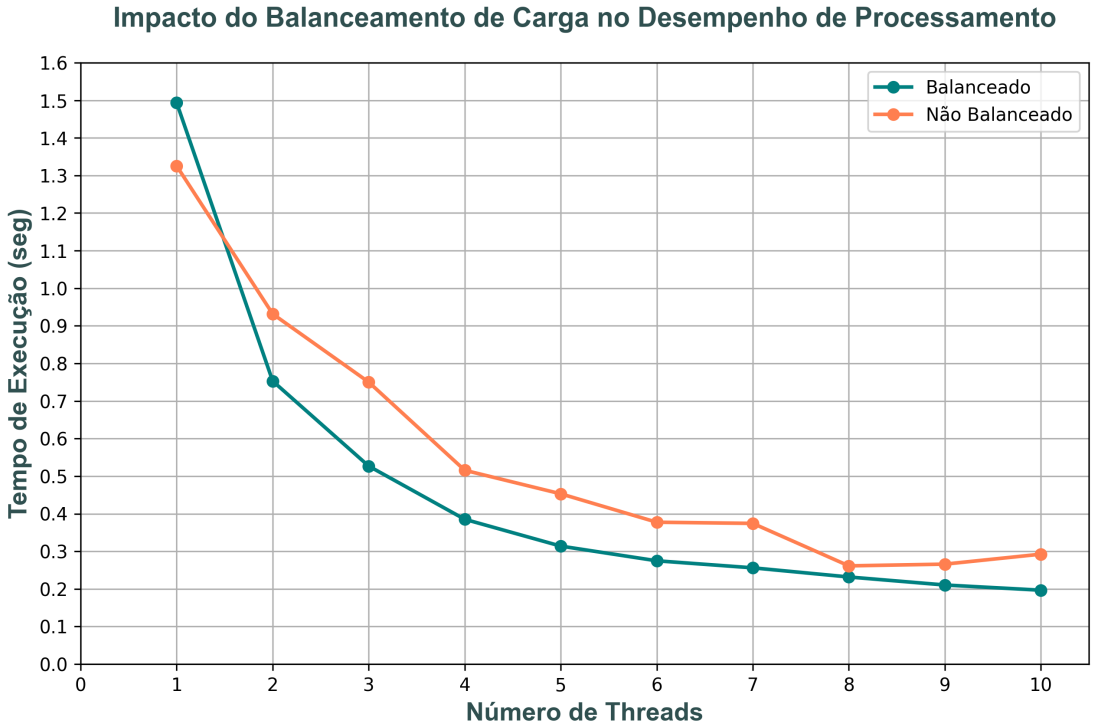
O programa começa incluindo diversas bibliotecas padrão do C++ que serão utilizadas para implementar funcionalidades específicas, tais como entrada e saída, manipulação de arquivos, operações matemáticas e gerenciamento de threads. Em seguida, são definidas algumas constantes que serão utilizadas no programa: 1) MAX_NUMBER Define o número máximo que será verificado se é primo; 2) MAX_THREADS Define o número máximo de threads que serão utilizadas para determinar se os números são primos; 3) REPETITIONS: Define o número de repetições que serão feitas para calcular o tempo médio; 4) UNBALANCED_OUTPUT_FILE: Define o nome do arquivo de saída do programa não balanceado; 5) BALANCED_OUTPUT_FILE Define o nome do arquivo de saída do programa balanceado.

O programa contém as seguintes funções: 1) isPrimeSlow(): Determina se um número n é primo de forma lenta, iterando através de todos os números menores que n e verificando se n é divisível por algum deles. Antes de cada verificação, a função dorme por 1 milissegundo para simular um processo lento e retorna verdadeiro se n for primo e falso caso contrário; 2) findPrimesBlock(): Encontra números primos em um bloco específico de números, iterando através dos números dentro do bloco e chamando a função isPrimeSlow() para cada um deles. Os resultados são armazenados em um vetor de booleanos results. 3) findPrimesThread(): Similar à função findPrimesBlock(), esta função também encontra números primos, com o diferencial que é projetada para ser chamada por threads separadas e utiliza um mutex para controlar o acesso ao índice do próximo número a ser verificado. Cabe ressaltar que, no contexto da programação concorrente, para garantir a consistência dos dados e evitar condições de corrida, é essencial proteger as regiões críticas, que são seções de código onde recursos compartilhados são acessados e modificados. Isso é feito através do uso de mecanismos de sincronização, como mutex (mutual exclusion). A ideia básica é que apenas uma thread por vez tenha permissão para entrar na região crítica, enquanto outras threads aguardam sua vez. 4) findPrimesUnbalanced(): Encontra os números primos de 0 até um número passado como parâmetro, utilizando o número de threads de forma desbalanceada. O intervalo de números em blocos é dividido e cada bloco é atribuído à uma thread diferente, sem garantir uma distribuição igualitária de trabalho entre elas. Os resultados são armazenados em um arquivo de saída; 5) findPrimesBalanced(): Similar à função findPrimesUnbalanced(), esta função encontra números primos de 0 até um número passado como parâmetro, porém, de forma balanceada. As threads compartilham um índice, cujo acesso e incremento residem em uma região crítica, protegida por um mutex. Os resultados também são armazenados em um arquivo de saída.

Na função main(), iniciamos o teste da abordagem desbalanceada criando um arquivo para armanezar os resultados, com as colunas "Threads "e "Tempo". Em um loop, o programa itera sobre diferentes quantidades de threads, variando de 1 a MAX_THREADS, para encontrar os números primos através da função findPrimesUnbalanced(). O cálculo é realizado executando "REPETITIONS"e medindo o tempo decorrido, desse modo, o tempo médio é calculado e escrito no arquivo CSV junto com o número de threads correspondente. Além disso, para cada quantidade de threads, o programa imprime na saída padrão informações sobre o tempo médio de execução, a quantidade de números avaliados, a quantidade de números primos encontrados e

quais são esses números. Depois de imprimir os resultados da abordagem desbalanceada, o programa repete o mesmo processo para a abordagem balanceada, utilizando a função findPrimesBalanced().

3 Resultados



O gráfico ilustra os resultados obtidos pelo nosso algoritmo, desenhado para salvaguardar regiões críticas contra condições de corrida. Podemos observar o contraste entre duas abordagens distintas: uma que segmenta a carga de trabalho de forma balanceada, com o intuito de minimizar o tempo total de processamento, e outra que não implementa essa segmentação. Através dessa visualização, é possível constatar que o tempo de execução é menor no programa balanceado comparado ao programa desbalanceado em todos os números de threads.

Ao aumentar o número de threads, de uma para duas, obtivemos uma redução no tempo de processamento de 49,58% no programa balanceado, enquanto no programa desbalanceado a redução foi de 29,71%. A média da redução no tempo com o acréscimo de uma thread para o programa balanceado é de 18,84%, enquanto no programa desbalanceado é de 14,38%.

Existem diferentes explicações para a redução de tempo no programa balanceado e no programa desbalanceado, no primeiro deles, como a carga total é dividida entre as threads de forma mais equilibrada, ocorre a redução no tempo de processamento, no programa desbalanceado, embora a thread com maior carga de trabalho possa reduzir sua carga, as outras threads não conseguem aumentar sua eficiência devido à falta de distribuição equitativa de trabalho. Assim, podemos concluir que o balanceamento adequado da carga entre as threads resulta em uma diminuição no tempo total de processamento.