Relatório
Programação Paralela (PP)
Prof. Marcelo Veiga Neves

Alunos: David Cainã Araujo Vieira
Exercício: Pattern Matching em OpenMP
Entrega: 15/10/2018

1) Implementação

A fim de avaliar o ganho obtido em paralelização através de OpenMP, foi solicitado o desenvolvimento de uma solução para o problema Pattern Matching (Correspondência de Padrões). O problema em questão trata-se do ato de verificação da presença de um padrão em um determinado conjunto de dados. Esse problema trabalha com a utilização de dois parâmetros inicias, sendo eles: o padrão a ser procurado e o conjunto de dados, a onde as funções de pesquisa serão aplicadas. Explicando melhor o problema solicitado, existem, dentre as condições de desenvolvimento, dois casos especiais (Curingas), sendo eles: o *"?"*, que representa uma casa com qualquer caractere único e *"*"*, que representa uma sequência de caracteres (incluindo sequencias vazias). Ou seja, tendo-se o conjunto de dados "baaabab", por exemplo, e os padrões: "*ba*ab", "*baaa?ab", "ba*a?" e "a*ab", teríamos, respectivamente, os resultados: padrão encontrado, encontrado, encontrado e não encontrado.

Em uma primeira aproximação do desenvolvimento, foi planejado dividir o problema em duas classes separadas, tratando as execuções sequencias e paralelas separadamente. Ainda, foi escolhido dividir o conjunto de dados em palavras separadas, afim de aplicar o padrão para cada palavra separadamente. Após alguns testes, foi notado que a "carga de processamento" passada para as threads (na execução paralela) estava muito "leve", o que influenciava na precisão do tempo de execução. Ou seja, a falta e/ou nenhuma mudança ao alterar a quantidade de threads em execução. Com essa conclusão, foi decidido alterar o jeito como era divido o conjunto de dados, passando a uma divisão "linha por linha" ao em vez de uma divisão de "palavra por palavra". Dessa forma, a "carga de processamento" das threads se mostrou muito maior, e a oscilação de tempo, conforme alterado o número de threads, comprovou isso. Ainda, para evitar repetição de código, foi decidido unificar as duas classes, anteriormente separadas, em uma única execução, tornando a execução e a visualização de dados muito mais compreensível.

Explicando as execuções sequencias e paralelas, respectivamente, foi utilizado um buffer para armazenar, dinamicamente, o conteúdo do conjunto de dados, evitando assim que para cada execução fosse aberto e percorrido todo o conjunto de dados. Utilizando a função *"strtok", foi iterado entre todas as linhas e aplicado a verificação de padrões, contabilizando e/ou atualizando a quantidade quando encontrado. Para a execução paralela, a única diferença desenvolvida foi a abertura da seção de paralelização em conjunto com a função "reduction", que evita que as threads sobrescrevam valores de outras threads. Após ambas as execuções serem finalizadas, foi decidido calcular a média dos resultados obtidos das 'n' execuções, dessa forma, adquirindo uma maior precisão nos dados.*

2) Dificuldade Encontradas

A maior dificuldade encontrada foi a falta de praticidade com a linguagem C, linguagem essa utilizada para o desenvolvimento do projeto. Mesmo que já conhecida, exigiu algum tempo do desenvolvimento para ser estudada/relembrada. Ainda, a "biblioteca" OpenMP gerou alguns desafios, entre eles, o "pensamento correto" para se paralelizar um problema sequencial, visto que, não ouve planejamento inicial para o projeto e, dessa forma, ocorreram várias inevitabilidades de mudar e replanejar a aproximação para o problema.

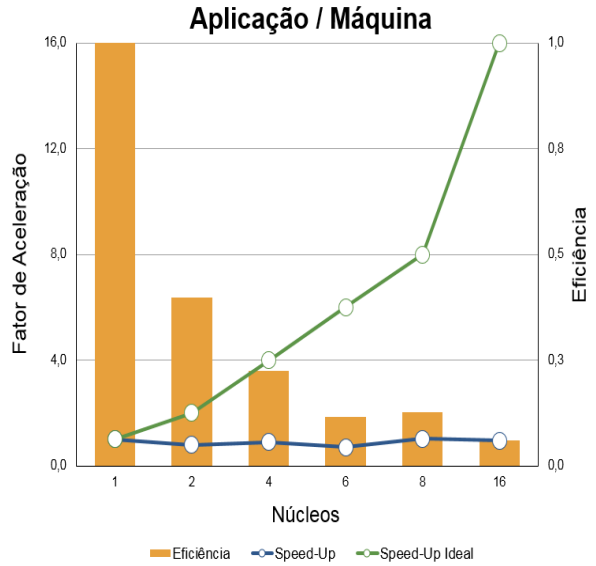
3) Testes

Pela união do projeto em uma única classe, ouve a necessidade de criar uma pequena interface para a interpretação dos dados. Além disso, foi desenvolvido um laço para automatizar 'n' execuções o que ajudou na execução dos testes. Para a

obtenção dos dados, foram efetuados um total de seis execuções por texto, alterando-se o número de threads disponíveis. Sendo esses números 1,2,4,6,8 e 16 threads do cluster, respectivamente. Como já mencionado anteriormente, houve a ocorrência de vários *inputs* de textos, o que levou a um grande número de padrões utilizados. Todavia, todos os padrões utilizados contavam com algum dos curingas em sua estrutura, dessa forma, gerando-se um maior volume de dados e maximizando a utilização e o processamento do programa. Ainda, os textos utilizados foram do autor Gutenberg, que totalizaram uma média de 6000 linhas por texto. O que foi muito útil, visto que textos com um baixo número de caracteres não teriam nenhum impacto no tempo de execução.

4) Analise do Desempenho

Como esperado, a versão paralela da Correspondência de Padrões obteve um desempenho superior a versão sequencial, especialmente quando o número de theads ficavam entre 2 e 6. Curiosamente, para a maioria dos textos testados, uma vez que os números de threads ultrapassavam o número 6, começava-se a ocorrer pequenas perdas de processamento, dessa forma, ganhando-se algum tempo a mais em comparação a um número menor de theads. Como pode ser visto no gráfico, calculado a partir das medias dos tempos de todas as execuções e testes, teve-se uma maior eficiência em torno de 6 threads, considerando o spee-up obtido e o ideal.



5) Observações Finais

Diferente de suposições iniciais, um maior número de theads/poder de processamento, não gerou o melhor desempenho, e nem pelo contrário. O melhor aproveitamento de processamento encontrado foi o balanceamento entre o número de theads e o tamanho da entrada recebida. Dito isso, a paralelização do problema de Wildcard Pattern Matching se mostrou possível com o algoritmo desenvolvido. Ainda, mesmo com o objeto alcançado, é possível realizar aperfeiçoamentos para o mesmo, utilizando melhores práticas e com um maior conhecimento da biblioteca utilizada.