

# Módulo 2 – Trabalho de implementação (Relatório)

Aluno: Harley Gomes Seixas Junior

DRE: 118161413

Período: 2020.1 Remoto

Disciplina: Computação Concorrente UFRJ

Professora: Silvana Rossetto

Entrega: 17/02/2021

## *Introdução*

Dado que o tema Quick Sort recursivo Concorrente já foi abordado anteriormente por mim no primeiro trabalho, dediquei esse segundo para tentar melhorá-lo ao máximo que eu pude. Irei demonstrar algumas alterações que foram aplicadas para que haja um melhor funcionamento do programa em si, bem como outras mudanças que foram feitas baseadas na avaliação da professora sobre o primeiro projeto.

Gostaria de ressaltar que, visto que eu já tinha abordado esta proposta de programa concorrente de forma muito similar à que foi proposta agora (eu já havia implementado um programa baseado na bolsa de tarefas e pool de threads para o Quick Sort concorrente) e que a avaliação dessa minha “primeira versão” do programa saiu justamente no mesmo dia da entrega deste trabalho, eu não consegui implementar tudo o que eu realmente desejava. De toda forma, creio que o programa foi melhorado de forma considerável, assim como será este novo relatório um pouco mais detalhado que o anterior.

Por fim, pode parecer, a princípio, que este relatório é parecido com o do primeiro trabalho; de fato é, contudo, contém detalhadamente as alterações realizadas no código. Por isso, peço que seja dado um voto de confiança para que tudo seja lido, mesmo que pareça ser o mesmo relatório de antes. Vamos ao algoritmo.

## *Quick Sort*

O Quick Sort, algoritmo escolhido para ser implementado de forma concorrente em meu trabalho, é um algoritmo de ordenação que se baseia na filosofia do “dividir para conquistar”. Um pivô é escolhido e todos os elementos menores são colocados à esquerda dele enquanto todos os elementos maiores são colocados à sua direita; ao final o pivô estará em sua devida posição se comparado com os outros elementos, todavia esses outros elementos não estão necessariamente na ordem correta se comparados consigo mesmos. Note que agora há a necessidade de se ordenar o lado esquerdo do pivô e o lado direito do mesmo. Assim, a função que implementa o Quick Sort é chamada recursivamente tanto para ordenar o lado esquerdo quanto o direito.

Pelo fato de apresentar complexidade de caso médio  $O(n \log n)$  é um algoritmo muito conhecido e utilizado. Pode apresentar no pior caso complexidade  $O(n^2)$ , contudo é um cenário raro de acontecer. Além disso, como todo algoritmo recursivo, pode ser escrito na sua forma não recursiva, mas neste trabalho foi optado pela opção recursiva pois me pareceu ser a mais fácil de tornar concorrente.

# A solução concorrente

Para obter uma implementação concorrente elegante e ao mesmo tempo eficiente, pensei em criar um algoritmo que empregasse o Quick Sort junto com a técnica de bolsa de tarefas, que consiste em deixar um conjunto de tarefas esperando até que haja uma thread disponível para realizá-la. Para fazer tal implementação e ao mesmo tempo economizar o máximo possível de memória, empreguei uma lista encadeada para funcionar como uma fila que armazena as “tarefas” que ficarão esperando pelas threads.

Dado o resumo feito no parágrafo acima, vamos detalhar melhor o que eu implementei. Como o algoritmo padrão é recursivo, percebi que a quantidade de tarefas necessárias para serem processadas aumenta com o tempo (cada execução da tarefa resulta na necessidade de execução de mais duas tarefas). Assim, percebi que seria um algoritmo ideal para empregar a concorrência, visto que em um certo ponto teremos muitas chamadas para a tarefa e as threads poderão cuidar disso executando-as ao mesmo tempo (na medida do possível, tendo a limitação do processador e Sistema Operacional).

Para ter um controle inteligente dessas tarefas, resolvi “quebrar” a recursão e armazenar cada tarefa nova que surge na minha fila que implementei. Nesse contexto, antes de criar as threads eu inicializo a minha primeira tarefa na fila. Quando eu digo tarefa estou me referindo, na verdade, aos argumentos que a função Quick Sort deve receber. Assim, após colocar os primeiros argumentos na fila, eu começo a criar as threads (o número de threads e tamanho do vetor são escolhidos pelo usuário na linha de comando na hora de chamar o programa para execução). Uma vez que as threads foram criadas, teremos uma pool de threads disponíveis para agir, que ficarão esperando até que uma tarefa esteja disponível para elas. Logo após o começo de criação das threads alguma delas provavelmente já começará a agir. Essa thread encontrará o argumento que há na fila e irá trabalhar com ele.

Enquanto a primeira thread (não necessariamente a primeira criada) trabalha com o primeiro argumento que estava na fila, há a possibilidade de que alguma thread tente acessar a fila de argumentos. Contudo, ao encontrá-la vazia, essa thread se bloqueará pela variável de condição chamada cond\_ex (abreviação de “condição de execução”). Toda thread que encontrar a fila vazia se encontrará nessa situação até que ela seja desbloqueada por essa variável de condição. Tal implementação é uma melhoria realizada encima da primeira versão de meu código que deixava as threads em espera ocupada, o que desperdiçava o uso do processador. Além disso, eu tive que pensar em uma forma de saber quando não há mais nenhuma tarefa para as threads realizarem e voltar para o fluxo de execução da main. Para tanto, utilizei duas variáveis de controle, uma de controle interno na thread e outra de controle global para saber quando realmente deveria finalizar a execução das threads e permitir que elas alcançassem o pthread\_exit(NULL).

Finalmente, vale ressaltar que empreguei o mecanismo de sincronização por escalonamento através do uso do lock para que não ocorresse inconsistências quando alguma variável global era alterada/lida pelas threads (evitar a situação de

corrida). Voltando a descrever o funcionamento do bloqueio e desbloqueio de threads, algumas alterações no código original foram feitas para o pleno funcionamento do mecanismo da variável de condição. Uma delas foi a implementação de um loop while na linha 141, que garante que uma thread só execute se de fato houver um argumento na fila. Tal implementação impede que alguma thread seja desbloqueada ao receber um signal, mas seja escalonada no processador apenas após outra thread consumir o argumento da fila. Como consequência, na ausência do while a thread poderia acabar progredindo com um argumento NULL, o que acarretaria em erros.

Finalizando a discussão sobre o controle de bloqueio de threads, toda vez que uma thread chega ao fim do seu loop de execução, são gerados mais dois argumentos que são depositados na fila. Para cada argumento é dado um signal para desbloquear threads para tratarem desses respectivos argumentos. Além disso, quando a última thread identifica que não há mais argumentos na fila (identifica que o algoritmo terminou) ela gera um broadcast para desbloquear todas as threads, as quais irão atingir o pthread\_exit() e assim o fluxo de execução voltará para a função main.

Há outras alterações realizadas no código, como por exemplo a exclusão de uma linha redundante que checava se o argumento retirado da fila era diferente de NULL, sendo que aquele trecho de código só era atingido se tivesse conseguido sair do while(arg == true). Em adição, foi implementado, baseado na sugestão que a professora deu na avaliação do primeiro trabalho, uma pequena função que avalia se a ordenação foi realizada de forma devida (muito útil para evitar ter que analisar exaustivamente vetores enormes). Também foi implementado um cálculo de aceleração obtida com a execução concorrente, resultado que é encontrado calculando-se o tempo de execução do programa em seu formato sequencial. Tal funcionalidade pode ajudar na hora de avaliar melhor o desempenho do programa. Eu sei que até agora não é o ideal, pois me foi sugerido fazer um programa a parte para criar casos de teste com vetores variados e depois gerar um outro arquivo com os resultados, contudo não foi possível realizar tal implementação em tão pouco tempo, visto que essas recomendações foram feitas no dia da entrega do trabalho. Eu até tentei implementar, mas tive problemas com manipulação de arquivos em C e desisti por medo de não dar tempo, sem falar que tenho também a disciplina de física.

Voltando ao programa, finalmente, a última alteração que julgo interessante ressaltar é sobre o tratamento de erros na execução, que agora são tratados com funções exit() para que a execução possa ser interrompida mesmo dentro de funções que não sejam a main, se você comparar verá que o código original não fazia isso caso ocorresse erros durante execuções distantes da main. Creio que até aqui eu tenha detalhado bem o funcionamento do meu programa, caso haja alguma explicação faltando é provável que esteja explicado de forma comentada no código. Espero que tenha gostado da descrição de meu programa, tentei detalhar bem e ao mesmo tempo deixar de uma forma fácil de entender, mantendo tanto as explicações clássicas da primeira versão do programa quanto demonstrando as novas alterações.



# Testes de desempenho

No tangente ao desempenho, vale a pena ressaltar em qual ambiente computacional os testes foram realizados (como a professora me recomendou na avaliação da primeira versão do programa/relatório). Os testes foram realizados no processador Ryzen 7 2700X que possui **8 núcleos** e 16 threads, rodando no clock base de 3.7GHz. Além disso, o sistema possui **16 GB de memória ram, 2666 MHz**. O hardware foi bem detalhado para que possíveis discrepâncias entre o desempenho listado aqui e o desempenho obtido pela professora possam ser justificados e também porque foi pedido que eu explicitasse os mesmos.

No tangente aos testes de performance, vale a pena comparar a antiga tabela de desempenho do programa concorrente em sua versão anterior (que deixava threads em espera ocupada) com a versão mais nova (que impede a espera ocupada por meio do bloqueio de threads).

Seguem os seguintes cenários de teste do programa original (antigo) e o desempenho obtido comparando o código sequencial (versão sequencial mesmo, sem nenhuma thread) com a versão concorrente. A aceleração foi calculada comparando o melhor caso concorrente com o sequencial. Perceba o impacto do overhead das threads para vetores pequenos, bem como o ganho espetacular da execução concorrente para vetores muito grandes. Vale citar que os resultados exibidos foram os melhores obtidos em 4 execuções de cada respectivo cenário (me foi recomendado explicitar o número de testes realizados na avaliação do primeiro trabalho da professora e eu tinha deixado de fazer isso na primeira versão do relatório, mas eu lembro perfeitamente o número de testes e eles foram 4).

| Dimensão | Sequencial | 2 threads | 4 threads | 8 threads | Aceleração       |
|----------|------------|-----------|-----------|-----------|------------------|
| 500      | 0.000053   | 0.003404  | 0.002904  | 0.003022  | 0,01825068870... |
| 1000     | 0.000123   | 0.009064  | 0.009300  | 0.012527  | 0,01357016769... |
| 100000   | 0.526423   | 0.470793  | 0.970857  | 1.247780  | 1,11816233461... |
| 1000000  | 54.468701  | 27.927257 | 19.076291 | 12.155635 | 4,48094246001... |

Agora, seguem os cenários de teste do programa em sua versão mais nova, que trata a espera ocupada por meio do bloqueio de threads pela variável de condição. Vale ressaltar que será mantido o mesmo padrão da tabela do programa anterior para que a comparação seja justa (4 execuções pegando a de melhor desempenho para cada caso e mantendo o padrão da tabela, destacando a melhor aceleração obtida).

| Dimensão | Sequencial | 2 threads | 4 threads | 8 threads | Aceleração       |
|----------|------------|-----------|-----------|-----------|------------------|
| 500      | 0.000053   | 0.002912  | 0.003436  | 0.003702  | 0,01820054945... |
| 1000     | 0.000123   | 0.008628  | 0.007764  | 0.009531  | 0,01584234930... |
| 100000   | 0.526423   | 0.522839  | 1.079699  | 1.226475  | 1,00685488266... |
| 1000000  | 54.468701  | 25.868756 | 14.021733 | 12.875457 | 4,23042855876... |

A partir da análise das duas tabelas e de comparações realizadas entre os diferentes cenários encontramos um fato curioso: ao compararmos os casos isolados, vemos que realmente a implementação mais nova (com bloqueio de threads) obteve vantagem em 7 casos, enquanto a implementação antiga (por espera ocupada) obteve vantagem em apenas 5 casos. Vale ressaltar que a coluna de threads em que a implementação por espera ocupada obteve mais ganhos (em quantidade de cenários) foi justamente na de 2 threads (perdendo em apenas 1 cenário para a execução por espera ocupada), enquanto nas outras duas colunas (4 e 8 threads) cada coluna perdeu em 2 cenários para a espera ocupada.

Como aqui foram feitos simples testes não é possível afirmar com certeza, contudo os dados me levam a crer que a coluna de 2 threads se saiu melhor do que as outras por conta do menor número de threads, que acarreta num menor overhead de bloqueio de threads, que acaba “levando menos tempo” para gerenciar esses bloqueios e desbloqueios. Em adição, podemos observar que nos melhores casos de tempo de cada linha, a implementação por bloqueio de threads obteve a vantagem, perdendo apenas para o caso de vetores com 1000 elementos, o que não me parece fazer muito sentido. Provavelmente o tempo “desvantajoso” é devido à gestão de escalonamento de threads do Sistema Operacional.

Por fim, podemos concluir que, com base na coluna de 2 threads, quanto maior o número de threads maior será o overhead de gerenciamento das mesmas, porém também será menor o tempo desperdiçado por evitar a espera ocupada. Nesse contexto, observa-se que em alguns casos o modo como o Sistema Operacional gerencia a execução das threads acaba influenciando no tempo total de execução.