Lista de Tarefas utilizando *threads*

Thiago I. Yasunaka RA:103069

***Resumo*—Sistemas operacionais executam muitos processos simultaneamente, isso só é possível pois computadores atuais possuem a capacidade de processar dados ou tarefas de forma paralela, ou seja, eles são capazes de executarem diversas tarefas ou processar dados ao mesmo tempo. Uma forma de trabalhar estes conceitos é por meio da construção de algoritmos paralelos. Por exemplo, a linguagem C possui uma biblioteca de *threads* conhecida como *pthreads*, a partir dessa biblioteca é possível executar programas paralelos tratando de possíveis problemas com acessos de dados simultâneos para escrita em um mesmo endereço de memória. A maior dificuldade para executar estes algoritmos de forma correta são os problemas de concorrência que podem acabar ocorrendo, devido a isso é necessário analisar o problema, arquitetar e somente depois iniciar sua construção. O trabalho realizado neste artigo apresenta os benefícios dessa abordagem utilizando um algoritmo que foi implementado e executado para resolver uma simulação de diversos processos simultâneos a partir de uma fila de tarefas, sendo que cada tarefa possui uma característica diferente na hora de ser executado.**

I. Introdução

EM tempos de pandemia, organizações da saúde em todo o mundo fazem pesquisas para combater determinada doença que ameaça de forma global todos as pessoas do planeta. Uma forma de combater a disseminação da doença fica por conta da produção de vacinas, por exemplo, durante a escrita desse relatório, está sendo aplicado a vacina contra o vírus da *covid-19*, um vírus que aterrorizou todo o planeta por sua rápida disseminação e ameaças quanto a saúde da população.

No entanto, qual é a relação de uma pandemia com o assunto abordado sobre paralelismo? A princípio nenhuma, porém é possível fazer uma analogia com programação paralela.

Quando a vacinação é disponibilizada pelo estado, geralmente o que acaba ocorrendo é que muitas pessoas ao mesmo tempo vão até a unidade mais próxima de saúde para receber a dose da vacina, e quando muitas pessoas vão ao mesmo tempo, acaba gerando uma fila de pessoas para receber a vacina. Supõem-se que para todas essas pessoas que vão tomar a vacina, exista somente uma enfermeira para aplicá-la. Então, ocorre que apenas uma única enfermeira tenha que ser capaz de vacinar uma fila inteira de pessoas, de forma que o processo de vacinação ocorre de uma em uma pessoa. Entrando em uma visão computacional, esse tipo de situação poderia ser classificada como um modelo de programação sequencial, na qual ocorre que apenas uma pessoa é vacinada por vez, pois só existe uma enfermeira no local, ou seja, a execução do algoritmo ocorre de passo em passo, uma instrução após a outra. Portanto, a pessoa seria análoga ao processo a ser executado em um determinado momento e a enfermeira seria análoga a quantidade de *threads* disponíveis no algoritmo. Voltando ao exemplo da fila da vacinação, é possível listar algumas vantagens e desvantagens. A grande desvantagem de ter apenas uma enfermeira é que o tempo para vacinar todas as pessoas da fila será muito maior, em contrapartida, caso existir mais enfermeiras e elas estiverem disponíveis para o trabalho, o tempo de vacinação total será muito mais eficiente. Outra desvantagem é que as pessoas ficariam ociosas sem poder fazer nada a não ser esperar, fazendo com que elas ocupem espaço desnecessário no ambiente em que elas ocupam. No entanto, apesar dessas desvantagens, é possível listar algumas vantagens. Fazer a logística da vacina quando há apenas uma enfermeira fica muito mais simples, por exemplo, se na sala de vacinação tivermos 1000 doses, fica óbvio que essas 1000 doses ficarão a cargo da única enfermeira que está aplicando a dose. Um outro ponto a destacar é contar quantas doses já foram aplicadas, para isso bastaria olhar quantas doses a única enfermeira aplicou e fazer uma subtração do total de doses que existia no início. É possível listar diversas outras vantagens e desvantagens, mas em resumo é para quando existe apenas uma enfermeira, o processo não é tão otimizado, ou seja, leva mais tempo para finalizar tudo o que se deseja, porém o controle da fila (analogia aos processos) e vacinas (analogia aos dados) fica mais intuitivo, simples de organizar e tratar os dados.

E se, em vez de uma única enfermeira, existirem 2..n enfermeiras aplicando as doses? Supondo que existirá 4 enfermeiras, ocorrerá que, dessas 1000 doses iniciais, cada enfermeira ficaria responsável por 250 doses. Isso quer dizer que o processo de vacinação ficaria 4x mais rápido? Em uma situação perfeita sim, porém como cada enfermeira possui uma velocidade diferente de aplicação, quer dizer que uma enfermeira poderá aplicar mais doses do que a outra, por exemplo, ao final de todas as aplicações, a enfermeira 1 poderia aplicado 200, a enfermeira 2 poderia ter aplicado 300, a enfermeira 3 poderia ter aplicado 225 e por fim, a enfermeira 4 poderia ter aplicado 275, portanto, provavelmente existiria uma diferença entre uma situação prática para uma situação perfeita. A principal vantagem de se dividir em diversas enfermeiras é o ganho de performance na velocidade das aplicações das doses, porém, como todas as doses deverão ser divididas em várias enfermeiras, é necessário trabalhar possíveis concorrências nas doses. Para isso, vamos supor um exemplo, suponha que a enfermeira 1 e a enfermeira 2 finalizou a aplicação de uma dose ao mesmo tempo, então ambas irão pegar a próxima dose de um mesmo "recipiente", disso fica a dúvida, quem irá pegar primeiro a próxima dose? Pois uma mesma dose não pode ser acessado por duas enfermeiras distintas.

É claro que, no mundo real a enfermeira 1 iria esperar a enfermeira 2 pegar a dose para só então, a enfermeira 1 pegar a próxima dose, ou vice-versa. Agora, fazendo uma analogia com a computação, poderia ocorrer o seguinte: A enfermeira 1 poderia acabar acessando uma dose que já não existia mais, pois a enfermeira 2 já teria pego a dose que a enfermeira 1 iria pegar, ou seja, a enfermeira 1 iria "acessar um dado antigo, depreciado". Este é um problema clássico na programação concorrente em que dois fluxos distintos acessam o mesmo dado (no nosso exemplo são as doses) ao mesmo tempo. Para resolver estes problemas, é necessário bloquear estes acessos simultâneos, para isso existem "cadeados"que bloqueiam acessos simultâneos por *threads* diferentes.

12 de setembro de 2021

II. Descrição do Problema

O trabalho realizado para a escrita desse relatório visa simular a execução de diversas tarefas paralelamente, enquanto possam existir chamadas externas em relação a tarefa principal. A entrada desse problema é um arquivo que contém duas informações em cada linha, a primeira informação é uma flag que pode ser o caractere ‘p‘ de processar ou então o caractere ‘e‘ de esperar. O caractere ‘p‘ representa um processo para ser executado pelo algoritmo construído enquanto o caractere ‘e‘ é um processo de espera aleatória que é executada pelo sistema, ou seja, quando um caractere ‘e‘ é lido, todos os fluxos devem esperar até que o tempo seja finalizado antes de pegar a próxima tarefa. O segundo parâmetro da linha é um número que representa o tempo para que cada tarefa necessita para ser executado, além disso, quando for uma tarefa do tipo ‘p‘, será preciso calcular a soma, quantidade de número ímpar, o valor máximo e o valor mínimo para estes valores.

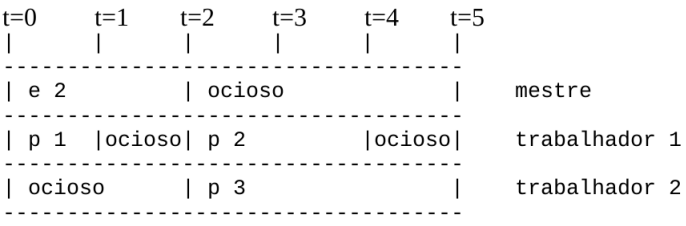

Figura 1: Exemplo de Execução

A figura 1 mostra um exemplo de execução para um arquivo que possui o seguinte conteúdo:

```
p 1
e 2
p 2
p 3
```

É possível observar que o programa inicia com um trabalhador que recebe um processo de 1 segundo de duração e logo depois o mestre trava os próximos fluxos por 2 segundos, então todos os trabalhadores devem aguardar o mestre liberar para que possam receber novos processamentos. Portanto, o mestre fica responsável por enviar processos para os trabalhadores enquanto ele analisa se existe algum comando de espera para ser executado. A entrada do algoritmo é um arquivo e a quantidade de *threads* (trabalhadores) enquanto a sua saída são os valores obtidos de todos os processamentos. Para o exemplo da figura 1, a saída deverá ser:

```
6 1 2 3
```

Na qual cada valor é representado por soma, #ímpar, mínimo e máximo, respectivamente.
Para solucionar o problema, foi utilizado a biblioteca *pthreads* disponível para a linguagem C, dessa forma é possível implementar processos paralelos incluindo validações para possíveis problemas com concorrêcia de dados ou tarefas. Além da biblioteca *pthreads*, foi implementado uma fila de tarefas para que o gerenciamento de cada tarefa fosse mais simples de ser manuseado. Os principais pontos da implementação do algoritmo são listados abaixo:

- Fila de tarefas para controlar quais tarefas devem ser realizadas, incrementado um novo item (chamado de *Task* no código fonte) quando uma nova linha do arquivo de entrada é lido, supondo que o comando seja ‘p‘ e decrementado quando um trabalhador consegue iniciar um novo processamento.
- *Locks* para cada variável agregada (soma, #impar, min, max) para não ocorrer condição de corrida na escrita de um novo dado.
- O número de *threads* definida ao executar o algoritmo representa a quantidade de trabalhadores que existirá durante a execução do algoritmo. Por exemplo, para a figura 1, a quantidade de *threads* seria dois.
- *Lock* e variável condicional para controlar entrada e saída de dados da fila. Por exemplo, caso a fila estiver vazia e o arquivo ainda não estiver totalmente lido, o trabalhador deve aguardar até que um novo processo seja inserido na fila de tarefas. Sempre que um novo processo ‘p‘ é lido, o principal papel do mestre é inserir esse novo processo na fila e então, "acordar"todos os trabalhadores para que possam realizar o novo trabalho.

III. Metodologia

Para a implementação da fila de tarefas e para os experimentos realizados, foram utilizados algumas ferramentas que serão descritas na tabela e tópicos abaixo:

- Versão GCC: gcc (Ubuntu 10.3.0-1ubuntu1) 10.3.0
- Biblioteca para threads: *pthread*
- Makefile para compilação e execução do algoritmo

Tabela I
Configurações da máquina

S.O	Ubuntu 21.04
Processador	11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz × 4
Memória RAM	16 GB’s
Placa de Vídeo	GeForce MX330/PCIe/SSE2

Observação para compilar o código: No projeto existe um diretório */utils*, nele contém a implementação da fila, *queue.c*, utilizada no código principal *par_sum.c* e também o arquivo de interface *queue.h*. Certifique-se de que existe

estes três arquivos para que o programa seja compilado com sucesso.
Para simplificar todo o processo de compilar, executar e passar os parâmetros, existe no arquivo Makefile a configuração que faz todo esse processo de forma mais rápida. para isso execute este comando no terminal:

make clean && make && time make exec-par-sum

Este comando irá limpar os arquivos compilados, compilá-los novamente e depois exibir o tempo de execução juntamente com a execução do programa. A execução padrão do programa será exatamente o mesmo programa exibido na figura 1, ou seja, 2 *threads* de execução e um arquivo de entrada presente em *./examples/simple.txt*

IV. Resultados e Discussão

Os resultados adquiridos utilizando a arquitetura de programação paralela resultaram em uma melhora significativa se comparada com a execução do algoritmo de forma sequencial. Diversos arquivos de entrada que simulam o processamento foram testados. Dentre eles, para um arquivo de 35 processos, foi possível obter o resultado que é apresentado na tabela abaixo:

Tabela II
Valores obtidos para um teste com 35 processos

n° threads	tempo(s)
sequencial	91.01
1	79.01
2	42.01
3	30
4	23
8	17

O teste realizado para a tabela acima possui o seguinte resultado: 79 19 1 6. Sendo que os valores representam a soma, quantidade de valores ímpares, mínimo e máximo, respectivamente. Além desse teste, um arquivo maior foi testado, dessa vez com 78 processos, para este caso de teste, foram obtidos os seguintes resultados:

Tabela III
Valores obtidos para um teste com 78 processos

n° threads	tempo(s)
sequencial	205.01
1	190.01
2	97.01
3	67.01
4	51
8	28

Para os resultados obtidos na tabela três, foi adquirido o resultado 190 46 1 6.
A partir dos dados obtidos e de outros casos testados que não foram demonstrados nas tabelas é que, quanto menor for a entrada do algoritmo, ou seja, menor for o arquivo com os processos a serem executados, menor será o impacto das *threads* no tempo de execução de todos as tarefas. Por exemplo, se a fila possuir algumas unidades de processos e executá-la com várias *threads*, o impacto das *threads* no tempo de execução não será tão efetivo. Porém em uma situação real e com os computadores atuais, dificilmente esse caso ocorrerá, pois geralmente o que acontece é que centenas de processos são executados simultaneamente. Em contrapartida, executando o programa com entradas maiores, o ganho de performance começa a ter grande relevância, isto pode ser observado tanto na tabela II quanto na tabela III. Os tempos de execução que foram adquiridos poderiam ser ainda melhores caso não existissem impedimentos das chamadas do comando 'e' de espera, pois as tarefas de espera fazem com que todos os trabalhadores fiquem ociosos até que o mestre dê algum sinal de trabalho.
Analisando os resultados das duas tabelas acima e executando outros testes, é possível gerar o seguinte gráfico da figura 1. Como o gráfico gerado pode ter muitas variações em suas informações devido o fato de que cada linha representa um arquivo de entrada e também, arquivos distintos podem ter dados muito dissipados, algumas medidas foram tomadas. As linhas do gráfico não possui relação uma com as outras, isso acontece pois são execuções de arquivos diferentes com chamadas de espera e processos distintos. Para a construção deste gráfico, a medida adotada foi inserir processos ou comandos de espera entre valores de 1 segundo até 10 segundos, aproximadamente.

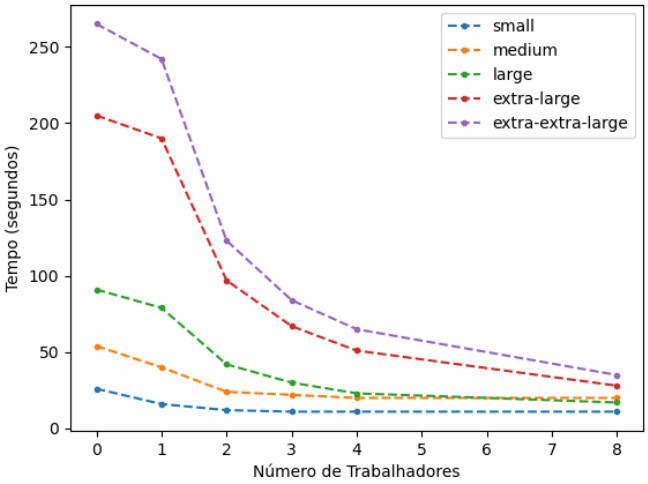

Figura 1: *Número de trabalhadores* x *Tempo (s)*

As curvas presentes no gráfico da figura I representa o tempo de execução do algoritmo enquanto o número de trabalhadores vai aumentando. Perceba que o eixo x representa o número de trabalhadores e quando este valor é 0, significa que o algoritmo foi executado sequencialmente, ou seja, sem o uso de mestre e trabalhadores. Como foi explicado acima, cada linha da figura I simboliza um arquivo de entrada, sendo que os arquivos foram nomeados conforme o tamanho de suas entradas, ou seja, a curva representada por *small* é também o nome dado ao arquivo com a menor entrada, com 10 processos. Por sua vez, a linha representada pela *extra-extra-large* é o nome dado ao arquivo com a maior entrada testada, com 103 processos a serem executados.

De acordo com a figura I e tendo conhecimento de que não existe relações entre uma curva e outra, é possível da mesma forma observar um padrão que ocorre entre eles.
Independentemente da quantidade de processos a serem executados, a performance para quando existe palelismo sempre é melhor se comparada sem paralelismo. Além disso, para todos os casos de testes, existe um ganho de performance até determinada quantidade de trabalhadores. Estes detalhes são explicados nos tópicos abaixos:

- Para a curva *small* e *medium* o ganho de desempenho não é muito expressivo pois para esses dois casos específicos existe uma limitação na quantidade de processos para serem executados, isto significa que aumentar a quantidade de trabalhadores não resolve o problema pois os trabalhadores ficariam ociosos de qualquer forma. Observe que o melhor caso são para 2 trabalhadores, e a partir disso o tempo de execução fica estagnado.
- Os ganhos mais espressivos ficam para as duas maiores entradas testadas, ou seja, para *extra-extra-large* e *extra-large*, pois nesses dois casos os trabalhadores conseguem se ocupar durante grande parte da execução, otimizando assim, o tempo de execução.

Todos os cinco casos possuem uma tendência de queda para zero, porém intuitivamente eles nunca chegarão realmente a zero, pois um processo nunca levará um tempo menor ou igual a 0 para ser executado. No entanto, é possível afirmar que o resultado gerado se assemelha a uma função exponencial decrescente, na qual o eixo Y será sempre um valor acima de 0 e o eixo X será um valor maior ou igual a 0, onde o eixo Y representa o tempo de execução e o eixo X a quantidade de trabalhadores utilizados no programa.
É claro que, para os casos testados, a quantidade de tarefas para os trabalhadores é muito maior do que as chamadas de espera para o mestre. Considerando que existam mais chamadas de espera do que tarefas, então possivelmente a curva do gráfico não irá ter essa semelhança.

V. Conclusão e Trabalhos Futuros

Após o desenvolvimento do trabalho, é possível apresentar algumas conclusões. Construir algoritmos paralelos exige mais detalhamento para resolver um problema, pois é necessário analisar o seu comportamento de uma perspectiva diferente, é uma quebra de paradigma se comparado com o que é usualmente feito durante o dia a dia, e além disso, se construído de maneira correta, é evidente que o desempenho utilizando *multi-threads* é melhor.
Para o problema solucionado neste trabalho, uma decisão tomada de grande relevância foi a implementação da fila de tarefas dinâmica, dessa forma não houve alocação de memória desnecessária como ocorreria caso a fila fosse armazenado em um vetor de tamanho fixo. Outro fator importante a destacar é que, apesar da fila dinâmica ter exigido um esforço maior de implementação no início, o ganho de produtividade, seja em usabilidade e manutenibilidade durante o desenvolvimento do *core* do algoritmo, foi bastante expressivo.
Apesar do projeto resolver o problema corretamente, uma melhoria que é possível realizar em termos de organização de código e possivelmente de desempenho, é fazer com que o mestre seja atribuído a uma *thread* separada, pois da forma como a primeira versão foi implementada, o trabalho realizado pelo mestre é feito dentro da função principal do programa. Dessa forma, seria possível abstrair responsabilidades da função *main* para funções externas, utilizando referências via ponteiro. Com isso, a função *main* ficaria responsável somente com o gerenciamento das *threads*, alocação de memória e quando necessário, encerramento, destruição de *threads*, locks e por fim, a finalização do programa. Fazendo dessa forma, a execução do programa passaria a ter um comportamento um pouco diferente, pois sempre que o programa fosse executado, seria necessário passar a quantidade de trabalhadores acrescentado de uma unidade para o mestre, por exemplo, para executar o programa com dois trabalhadores, seria necessário executá-lo com três *threads*, pois a terceira *thread* seria alocado para o trabalho do mestre. São pequenas alterações que podem ser realizadas e testadas, porém o resultado do programa deverá ser sempre o mesmo.
Um outro trabalho que poderia ser aprimorado é referente a análise da performance do algoritmo. Para obter um relatório mais completo tanto em nível de *hardware* quanto a nível de código implementado, é possível utilizar ferramentas que fazem essas análises. Dessa maneira, caso haja comportamentos diferentes em algumas ocasiões, é bem provável que esses monitoramentos possam ser de grande importância.
Vale ressaltar que o principal objetivo destas implementações *mult-threads* são o ganho de performance na execução dos algoritmos, então é sempre importante analisar com cuidado se determinada alteração no código é relevante a ponto de melhorar a sua performance.

Referências

[1] CONDITIONAL wait and signal in multi-threading. 2021. Disponível em: https://www.geeksforgeeks.org/condition-wait-signal-multi-threading/. Acesso em: 14 set. 2021.

[2] PTHREADS Cond. Disponível em:https://pubs.opengroup.org /onlinepubs/007904975/functions/pthread_cond_broadcast.html. Acesso em: 11 set. 2021.