Análise do algoritmo mestre-trabalhador paralelo utilizando MPI

Felipe Diniz Tomás - RA:110752

***Abstract*—O presente estudo realiza uma análise do algoritmo mestre-trabalhador implementado de forma paralela utilizando MPI, com o objetivo de analisar e mensurar o desempenho do mesmo. Dessa forma, foi coletado o tempo de execução de determinados testes com diferentes quantidades de tarefas, buscando calcular o *speedup*[1] em cada caso. Os resultados mostraram que o algoritmo performa bem em determinados tipos de testes. Podendo variar conforme o número de tarefas, tempo de cada uma e ordem. A razão disso é a influência da carga desbalanceada entre os processos trabalhadores.**

I. Introdução

Quando tratamos de realizar tarefas de processamento, cada tarefa consumirá um determinado tempo para ser processada, podendo ser mais ou menos complexa levando consequentemente mais ou menos tempo para ser resolvida. Tradicionalmente, os softwares são escritos para serem executados sequencialmente, sendo implementados como um fluxo serial de instruções. Tais instruções são executadas por uma unidade central de processamento de um computador, onde somente uma instrução pode ser executada por vez, e após sua execução, a próxima então é executada.

Essa abordagem, aplicada a sistemas onde há uma demanda de processamento maior, com instruções complexas que consomem muito tempo para serem produzidas, irá naturalmente escalonar seu tempo de execução de acordo com cada instrução, já que uma é executada uma após a outra. Imagine então, se em vez de executarmos cada instrução por vez, poderiamos executar várias instruções paralelamente, ou seja, enquanto a primeira tarefa é realizada a próxima também está sendo executada paralelamente, em teoria reduziriamos drasticamente o tempo de execução.

Sendo assim, entra em ação a computação paralela, que faz uso de múltiplos elementos de processamento simultaneamente para resolver um problema. Isso é possível ao quebrar um problema em partes independentes de forma que cada elemento de processamento pode executar sua parte do algoritmo simultaneamente com outros, possibilitando assim a redução do tempo de execução [1].

Uma ferramenta que possibilita a troca de informação que desenvolve a programação paralela é a MPI (*Message Passing Interface*) que nada mais é que um padrão para comunicação de dados em computação paralela. Existem várias modalidades de computação paralela, e dependendo do problema que se está tentando resolver, pode ser necessário passar informações entre os vários processadores ou nodos de um cluster, e o MPI oferece uma infraestrutura para essa tarefa.

Existem várias implementações de MPI de código aberto, que fomentaram o desenvolvimento de uma indústria de software paralela e encorajaram o desenvolvimento de aplicativos paralelos portáteis e escaláveis em grande escala.

Portanto, o objetivo do trabalho é realizar a implementação do código paralelo do algoritmo mestre-trabalhador com a utilização da biblioteca *MPI*, entender concenceitos de trocas de mensagens entre processos, além de produzir uma análise da métrica de *speedup* considerando diferentes casos de uso, a fim de esclarecer os resultados encontrados e estabelecer um padrão de análise de código.

II. Descrição do problema

O problema consiste em simular o processamento de instruções, a partir da leitura de uma base de dados. A base de dados é um arquivo contendo uma lista de tarefas. Cada tarefa por sua vez, é um código de caractere que indica uma ação e um número, podendo ter como ação “p” (para processar) e “e” (para esperar).

Processar uma tarefa com número n significa esperar n segundos e então atualizar algumas variáveis agregadas globais. A ação “e” simula uma pausa nas entradas de tarefas, ou seja, pausa-se por determinado tempo as novas entradas.

As váriavés globais agregadas que são atualizadas ao processar uma tarefa, consistem na soma de todos os números, quantidade de números ímpares, maior número e menor número.

Observa-se que o problema é de paralelismo de tarefas e segue o padrão mestre-trabalhador, que nada mais é quando a thread original mestre divide um problema em vários subproblemas e os despacha para as threads de trabalho, ou seja, Isso permite que a thread mestre se concentre em receber trabalhos, enquanto a thread de trabalho se concentra em fazer o trabalho real.

Neste caso a base de dados contendo a lista de tarefas é lida pela thread mestre e distribuída para as demais, assim é possível diminuir o tempo de execução processando diferentes tarefas ao mesmo tempo.

Por exemplo, em uma lista com 4 ações, sendo três delas de processos com tempo 1, 2 e 3 respectivamente, e uma ação de espera de dois segundos entre o primeiro e segundo processo, sequencialmente levaria 8 segundos para executar toda a operação. Já se utilizarmos 2 threads de trabalho, o tempo pode ser diminuído para 5 segundos, tendo em vista que nossas tarefas são independentes.

[1] O *speedup* é um valor que mede a melhoria na velocidade de execução de um processo executado em duas arquiteturas semelhantes com recursos distintos.

III. METODOLOGIA

As execuções realizadas foram feitas no seguinte ambiente computacional:

- Processador Intel® Core™ i5-4210U
  - Número de núcleos: 2;
  - Número de threads: 4;
  - Frequência baseada em processador: 1.70 GHz;
  - Frequência turbo max[2]: 2.70 GHz;
  - Cache: 3 MB Intel® Smart Cache[3]
    - Tamanho da memória cache:
      - L1 - 128 KB;
      - L2 - 512 KB;
      - L3 - 3 MB.
- 8 Gb de memória RAM DDR3
- Sistema Operacional: Ubuntu, sendo ele:
  - Ubuntu 18.04 LTS;
  - Kernel: 4.15.0.43;
  - Compilador: gcc 9.3.0.

A. Casos de testes

Os casos de testes utilizados variam em seu número de tarefas ("p" ou "e") a serem executadas, tempo de cada tarefa e ordem. Foram 6 aquivos de testes, planejados da seguinte forma:

- "test1": Poucas tarefas com tempo elevado[4].
- "test2": Tarefas ordenadas de forma crescente de acordo com o tempo.
- "test3": Mesmas tarefas do "test2" porém em ordem aleatória.
- "test4": Muitas tarefas com tempo elevado[4].
- "test5": Muitas tarefas com tempo baixo[5].
- "test6": Poucas tarefas com tempo baixo [5].

Para cada arquivo foi realizado um total de 6 execuções, sendo a primeira delas descartada para evitar *compulsory miss*[6], e calculado a média de tempo das demais.

IV. RESULTADOS

O tempo do código sequencial para os seus respectivos arquivos de testes foram:

TABLE I: Tempo em segundos do código sequencial.

test1	test2	test3	test4	test5	test6
329s	284s	347s	1542s	232s	71s

Os resultados de tempo do código paralelo, considerando os processos trabalhadores, para os seus respectivos arquivos de testes foram:

[2]Frequência turbo máxima é a frequência máxima de núcleo único, à qual o processador pode funcionar, usando a Tecnologia Intel® Turbo Boost.

[3]Cache inteligente refere-se à arquitetura que permite que todos os núcleos compartilhem dinamicamente o acesso ao cache de último nível.

[4]10 segundos ou mais.

[5]9 segundos ou menos.

[6]O *compulsory miss* é uma falha que ocorre quando um bloco é trazido pela primeira vez a memória cache.

TABLE II: Tempo em segundos do código paralelo.

Threads	test1	test2	test3	test4	test5	test6
1	274s	311s	311s	1364s	201s	56s
2	151s	164s	168s	733s	113s	34s
3	116s	117s	122s	515s	85s	26s
4	100s	85s	92s	415s	72s	25s

Com o tempo de execução de cada base de dados, tanto do código paralelo como sequencial, é aplicado a fórmula abaixo para o cálculo de *Speedup*.

$$S(t) = \frac{Tempo\ de\ execução\ Sequencial}{Tempo\ de\ execução\ com\ t\ Threads}$$

A figura 1 a seguir indica o *Speedup* alcançado para cada base de dados executadas pelo algoritmo mestre-trabalhador paralelo.

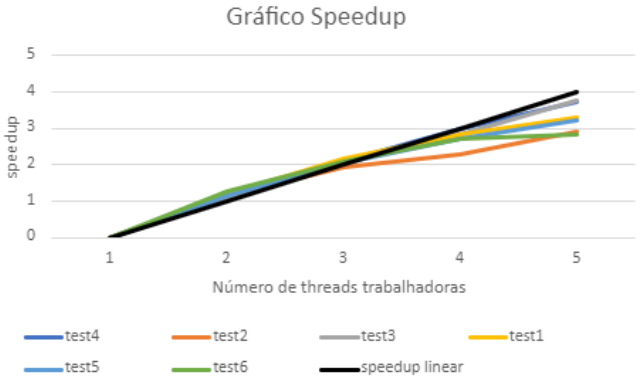

Fig. 1: Gráfico de *Speedup* do algoritmo mestre-trabalhador.

TABLE III: Valores da métrica de Speedup.

Threads	test1	test2	test3	test4	test5	test6
1	1,20	1,14	1,11	1,13	1,15	1,26
2	2,17	1,95	2,06	2,10	2,05	2,04
3	2,83	2,28	2,84	2,99	2,72	2,72
4	3,29	2,92	3,77	3,71	3,22	2,83

Primeiramente é importante pontuar que o nosso problema não possui carga balanceada entre as threads trabalhadoras, mas sim uma distribuição de carga irregular, logo um determinado gráfico de *Speedup* é específico a um determinado dado, ou seja, a volatilidade da métrica irá depender de como as tarefas são listadas no arquivo e o tempo das mesmas. Uma configuração diferente das tarefas a serem executadas resultará em um *Speedup* também diferente.

Assim, é interessante que haja uma diversidade de configurações de entradas. Nesse caso, como explicado anteriormente na seção de metodologia, os arquivos de entrada possuem uma lógica de diversidade para testar diferentes possibilidades de base de dados.

Nota-se que houve um *Speedup superlinear*, que apesar de ocorrer nos testes, não garante que todos os casos serão assim. Além dos testes específicos, outra das possíveis razões para acontecer um *Speedup superlinear* se dá pelo efeito cache,

que ocorre quando todo o conjunto de trabalho pode caber em cache então o tempo de acesso à memória é reduzido drasticamente, o que causa a aceleração extra além daquela do cálculo real [2].

Tendo ciência disso, é correto analisar o desempenho de *Speedup* de acordo com a configuração do arquivo usado. Por exemplo, poucas tarefas com tempo elevado obteve um *Speedup* em geral melhor comparado a muitas tarefas com baixo tempo.

O teste de muitas tarefas com tempo elevado alcançou um *Speedup* pior comparado a poucas tarefas com tempo baixo, isso mostra que para problemas relativamente menores com menos carga o algoritmo desempenha melhor do que para menos tarefas sem grande consumo de tempo.

Quando se trata da ordem das tarefas, o teste com as tarefas em ordem aleatória obteve um *Speedup* melhor comparado ao teste ordenado de forma crescente. Lembrando que ambos possuem as mesmas tarefas. Ou seja a ordem das tarefas também podem influenciar no desempenho do algoritmo. Isso ocorre pelo fato das distribuições de cargas não balanceadas paras as threads trabalhadoras, que podem ter tempo ocioso adicional dependendo da sequencia de tarefas na fila.

Os testes obtiveram um aumento de performance comparados ao sequencial. Isso mostra o quanto interessante pode ser a utilização do padrão mestre-trabalhador e como ele pode ser aplicado a determinados tipos de problemas dependendo de como são as tarefas necessárias. Porém devido ser um problema que depende muito das instruções/tarefas a serem executadas, é difícil analisá-lo de forma precisa.

A utilização do MPI tornou o código mais prático em comparação a outros métodos. Nele você não precisa se preocupar com a memória compartilhada, ou seja, não há risco de um processo alterar valores de outro processo, evitando problemas que muitas vezes são complicados de resolver. Além disso, trabalhar com memória compartilhada exige muita atenção e cuidado para que outros processos não alterem o que não devem. De forma geral o código mostrou-se eficiente, mas ainda sim deve-se ter uma preocupação quanto ao balanceamento de carga.

V. Conclusão

As formas "tradicionais" de multithreading costumam compartilhar a memória entre todas threads. Isso pode ser um tanto perigoso, já que torna-se muito fácil modificar acidentalmente dados que outra thread está usando, resultando em algum problema. Dessa forma, a responsabilidade recai sobre o programador para proteger cuidadosamente os dados contra acesso inseguro. Isso também implica que todos os processos estejam rodando na mesma máquina, com a mesma memória.

Utilizar processos independentes com *message-passing interface (MPI)* é possível ter mais controle sobre qual dados são compartilhados e quais são exclusivos em cada processo, assim a chance que um processo modifique o outro é muito menor. Além disso, essa troca de mensagens possibilita que máquinas diferentes troquem informação entre processos via rede.

Os resultados obtidos para os testes mostraram-se satisfatórios, reduzindo bastante o tempo em cada caso. Dependendo do problema em que queira aplicar o padrão mestre-trabalhador, terá resultados eficientes.

Considerando projetos futuros, seria interessante a utilização de um processador mais moderno, com mais núcleos físicos, afim de estudar o comportamento do código com mais threads trabalhadoras. Também seria importante testar mais listas de tarefas diversificadas, examinando o comportamento para demais casos possíveis

Quanto ao código, existem implementações que fazem o uso de uma combinação de threads OpenMP/pthread e processos MPI, que é conhecido como Programação Híbrida [3]. É uma baordagem mais complexa em comparação ao uso de MPI puro, no entanto com a recente redução nas latências com OpenMP, faz muito sentido usar MPI híbrido. Essa abordagem pode evitar a replicação de dados, já que uma vez em que as threads podem compartilhar dados dentro de um nó, se algum dado precisar ser replicado entre os processos, isso pode ser evitado. Threads ficam mais "leves" e, portanto, metadados associados aos processos são reduzidos, além de que como as threads se comunicam usando memória compartilhada, você pode evitar o uso de comunicação MPI ponto a ponto dentro de um nó [4].

Também já existem estudos que buscam modificar o algoritmo mestre trabalhador, combinando algoritmos estáticos e dinâmicos para melhorar o balanceamento de carga durante as partes críticas da execução da tarefa [5], portanto uma revisão bibliográfica a cerca de melhoramentos ao padrão é sempre essencial para estabelecer futuros projetos sobre o tema.

References

[1] G. E. Blelloch and B. M. Maggs, “Parallel algorithms,” *Communications of the ACM*, vol. 39, pp. 85–97, 1996.

[2] D. Benzi, “Parallel three dimensional direct simulation monte carlo for simulating micro flows,” *Springer*, p. p. 95, 2007.

[3] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, “Mpi + mpi: A new hybrid approach to parallel programming with mpi plus shared memory,” *Computing*, vol. 95, 12 2013.

[4] H. Zhou, J. Gracia, N. Zhou, and R. Schneider, “Collectives in hybrid mpi+mpi code: Design, practice and performance,” *Parallel Computing*, vol. 99, p. 102669, 2020.

[5] L. Filipovic and B. Krstajic, “Modified master-slave algorithm for load balancing in parallel applications,” *ETF Journal of Electrical Engineering*, vol. 20, pp. 74–83, 10 2014.