Trabalho 3 - Programação Paralela
Relatório de Experimentos

Geffté L. S. Caetano[1], **Arthur H. A. Farias**[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79.070-900 – Campo Grande – MS – Brazil

{geffte.caetano,arthur.farias}@ufms.br

Abstract. *In this study, the algorithm for counting arbitrary-sized clicks on undirected graphs was implemented and tested in two parallelized versions using CUDA and MPI. In Version 1 (CUDA), the click counting is parallelized on the GPU, using a global queue of vertices and efficient memory management strategies. The main difficulty in this approach is the control of local memory, since GPUs do not support efficient dynamic allocation. In Version 2 (MPI), the parallelization occurs in a distributed memory environment, with a manager process that distributes the vertices among the worker processes. This approach distributes the work among multiple processes and executes the click counting in a local environment, with inter-process communication via messages. The performance of each version was compared based on the execution time on graphs of different sizes, seeking to identify the advantages and limitations of each approach.*

Resumo. *Neste estudo, o algoritmo de contagem de cliques de tamanho arbitrário em grafos não orientados foi implementado e testado em duas versões paralelizadas utilizando CUDA e MPI. Na Versão 1 (CUDA), a contagem de cliques é paralelizada na GPU, utilizando uma fila global de vértices e estratégias de gerenciamento de memória eficiente. A principal dificuldade nessa abordagem é o controle da memória local, uma vez que as GPUs não suportam alocação dinâmica eficiente. Já na Versão 2 (MPI), a paralelização ocorre em um ambiente de memória distribuída, com um processo gerente que distribui os vértices entre os processos trabalhadores. Essa abordagem distribui o trabalho entre múltiplos processos e executa a contagem de cliques em um ambiente local, com comunicação entre processos via mensagens. O desempenho de cada versão foi comparado com base no tempo de execução em grafos de diferentes tamanhos, buscando identificar as vantagens e limitações de cada abordagem.*

1. Grafo

De acordo com [Feofiloff et al. 2011], um grafo é um par (V,A), onde V é um conjunto arbitrário e A é um subconjunto de V. V é chamado de **vértices** e seu subconjunto A é chamado de **arestas**. Ainda nessa linha, os elementos de A assumiram a forma v,w, onde v e w pertencem ao conjunto V.

1.1. Cliques, cliques máximas e de tamanhos arbitrários

Segundo [Feofiloff et al. 2011], dado um grafo G, uma **clique** (ou conjunto completo) de G é definido por ser um conjunto de vértices X dois a dois adjacentes, ou seja, deve haver pelo menos uma aresta que conecta todos os vértices de X.

Ainda nessa linha, se há uma clique máxima denotada por $\omega(G)$, é óbvio que haverá pelo menos uma clique de tamanho menor ou igual a k, onde $k \leq \omega(G)$. Assim, é possível concluir que, desde que haja uma clique máxima que atenda às condições acima, é possível obter cliques de tamanho menores que a máxima.

2. CUDA

CUDA é uma plataforma de computação paralela e uma API de programação desenvolvida pela NVIDIA que permite aos desenvolvedores utilizar GPUs para realizar cálculos de propósito geral. Além de renderizar gráficos e imagens, as GPUs podem ser usadas para acelerar algoritmos de computação científica, aprendizado de máquina, processamento de imagens, simulações e outros tipos de cálculos. O CUDA é projetado para aproveitar a arquitetura de múltiplos núcleos das GPUs, permitindo que milhares de threads sejam executadas em paralelo. Cada thread da GPU pode processar uma tarefa individualmente, permitindo um alto nível de paralelismo. Isso torna o CUDA particularmente eficaz em tarefas que podem ser divididas em muitas pequenas operações que podem ser realizadas simultaneamente. A programação em CUDA é feita em uma extensão do C/C++ e a comunicação entre a CPU e a GPU é gerenciada explicitamente pelo desenvolvedor, que precisa transferir dados entre a memória da CPU e a memória da GPU. O gerenciamento de memória é um aspecto crucial, já que a memória da GPU é separada da memória principal da CPU e não é dinâmica, o que exige uma alocação cuidadosa para maximizar a performance. CUDA é amplamente utilizado em áreas como computação científica, processamento de imagens e vídeo, aprendizado profundo e inteligência artificial.

3. OpenMPI

OpenMPI, por outro lado, é uma implementação livre do padrão MPI (Message Passing Interface), que é uma especificação para comunicação entre processos em ambientes de computação paralela distribuída. O OpenMPI permite que múltiplos processos executem tarefas simultaneamente em diferentes máquinas ou núcleos de uma mesma máquina, comunicando-se entre si por meio de mensagens. Esse modelo de passagem de mensagens é fundamental para programar aplicações distribuídas, onde os processos precisam trocar informações para realizar um trabalho conjunto. O OpenMPI oferece suporte à comunicação de alto desempenho entre os processos e é projetado para ser altamente escalável, permitindo que um sistema execute milhares de processos distribuídos em um cluster de máquinas. Isso o torna ideal para aplicações em supercomputação, simulações científicas, modelagem matemática e análise de grandes volumes de dados. Ele permite que os desenvolvedores escolham entre diferentes formas de comunicação, como comunicação ponto a ponto ou coletiva, dependendo das necessidades do aplicativo, e oferece suporte para diferentes topologias de rede, como Ethernet e Infiniband. O OpenMPI é amplamente usado em ambientes de clusters e supercomputadores e é uma das bibliotecas mais populares para programação paralela em larga escala.

4. Algortimo de contagem de cliques em CUDA e OpenMPI

A implementação de algoritmos de contagem de cliques de tamanho k arbitrário em grafos não dirigidos de forma paralela utilizando CUDA e OpenMPI apresenta uma série de desafios técnicos, especialmente no que diz respeito à eficiência na distribuição de dados e à coordenação entre as unidades de processamento. Em CUDA, a principal dificuldade está no gerenciamento eficiente da memória, pois o grafo precisa ser particionado e carregado na memória global da GPU, que deve ser acessada de maneira otimizada para evitar latências. A paralelização da contagem de cliques exige a divisão do trabalho entre threads de forma equilibrada, minimizando a sobrecarga de sincronização entre eles, além de garantir a integridade dos dados ao lidar com operações concorrentes sobre as contagens de cliques. Já no OpenMPI, a comunicação entre processos distribuídos se torna um ponto crítico, uma vez que é necessário transferir informações sobre subgrafos entre nós distintos, o que pode gerar alta latência e overhead. Além disso, o particionamento do grafo para garantir uma divisão eficiente do trabalho e evitar redundância de dados, sem prejudicar a precisão da contagem de cliques, é um desafio importante. A escalabilidade do algoritmo, tanto na GPU quanto no ambiente distribuído, depende de estratégias de balanceamento de carga e da escolha adequada de parâmetros para otimizar o uso dos recursos computacionais, minimizando gargalos de comunicação e sincronização entre os processos ou threads.

4.1. Estratégias para implementação

As estratégias de CUDA e MPI buscam superar limitações de memória e maximizar desempenho em problemas de contagem de cliques. Em CUDA, utiliza-se buffers locais por thread para armazenar cliques parciais, evitando a necessidade de materializar todos os cliques na memória global. A expansão de cliques ocorre iterativamente, e uma fila global balanceia a carga entre threads. A comunicação é minimizada ao processar dados localmente na GPU, enquanto operações atomizadas registram contagens. Em MPI, a paralelização distribui subgrafos entre processos, permitindo que cada nó compute cliques em seus fragmentos locais. Reduz-se a comunicação ao compartilhar apenas informações essenciais, e os resultados são combinados ao final. Ambas abordagens priorizam a contagem direta, alocando memória de forma eficiente e utilizando estratégias como balanceamento dinâmico de carga e sincronização local para garantir escalabilidade e alta performance mesmo em cenários com grandes volumes de dados.

5. Resultados e discussões

Por fim, esta seção apresenta os resultados obtidos a partir dos testes de desempenho, comparando os tempos de execução e as métricas de eficiência para diferentes valores de k nas cliques e para diferentes datasets em cada algoritmo. O tempo de execução de cada implementação foi contabilizado para avaliar o impacto do paralelismo e verificar se houve ganho ou perda de desempenho.

Os testes em CUDA foram realizados em um notebook com uma placa RTX 3070 Ti de 8 GB, enquanto os testes em OpenMPI foram conduzidos em um notebook com processador Intel i7-12H. A tabela a seguir apresenta os valores de cliques obtidos e os tempos de execução em segundos para a implementação paralela utilizando CUDA e OpenMPI, considerando o balanceamento de carga entre os processos distribuídos. A

comparação também inclui os tempos das implementações utilizando Thread, permitindo uma análise detalhada do impacto do paralelismo. Todos os dados foram organizados em uma tabela única para facilitar a comparação entre os métodos e os diferentes ambientes de execução. Abaixo segue a tabela com os resultados.

Table 1. Resultados de desempenho com CUDA, OpenMPI, OpenMP e Threads.

Dataset	Cliques (k)	Tempo (s) - CUDA	Tempo (s) - OpenMPI	Tempo (s) - OpenMP	Tempo (s) - Thread
citeseer	3	-	0.0147	0.0115	0.0297
	4	-	0.0185	0.0155	0.0303
	5	-	0.0215	0.0161	0.0406
	6	-	0.0241	0.0183	0.0407
ca_astroph	3	-	4.9841	3.0360	10.6048
	4	-	69.5167	48.6461	158.1690
	5	-	750.989	822.0420	1694.7100
	6	-	7931.4012	7364.4700	14962.4000
	7	-	+4hrs	+4hrs	+4hrs
dblp	3	-	10.0341	5.8966	18.2536
	4	-	114.4010	67.6689	175.1290
	5	-	1729.6000	2148.23	1723.2500
	6	-	+4hrs	+4hrs	+4hrs

Notas: **Tempo (s) - CUDA** refere-se ao tempo de execução com aceleração CUDA (não disponível neste experimento); **Tempo (s) - OpenMPI** refere-se ao tempo de execução com paralelismo via OpenMPI; **Tempo (s) - OpenMP** refere-se ao tempo de execução com paralelismo via OpenMP; **Tempo (s) - Thread** refere-se à implementação de paralelismo em CPU usando Threads de forma balanceada.

Os resultados indicam que a implementação utilizando OpenMPI geralmente apresenta tempos maiores em comparação com as outras abordagens, principalmente devido à natureza distribuída do OpenMPI, que simula um ambiente de execução em múltiplas máquinas. Essa simulação de ambiente distribuído pode ser mais lenta, especialmente em cenários de paralelismo local, como aqueles onde o OpenMP se destaca.

Por outro lado, o OpenMP, que é otimizado para paralelização em sistemas de múltiplos núcleos locais, apresenta tempos de execução significativamente menores, com uma vantagem clara sobre o OpenMPI na maioria dos casos. Ao comparar OpenMP e OpenMPI, percebe-se que o OpenMPI leva praticamente o dobro do tempo para completar as mesmas tarefas, evidenciando a eficiência do OpenMP em tarefas de paralelismo local, enquanto o OpenMPI pode sofrer perdas devido à sobrecarga de comunicação e gerenciamento em um ambiente distribuído.

Além disso, a implementação utilizando Threads balanceadas (via OpenMP) também mostra-se mais eficiente do que o OpenMPI, reforçando a ideia de que a paralelização local é mais vantajosa para os cenários analisados, enquanto o OpenMPI se destaca em tarefas que realmente exigem uma arquitetura distribuída.

Embora o problema em questão não seja regular, o que levanta dúvidas sobre o

potencial desempenho superior da GPU, a falha na implementação foi atribuída a aspectos técnicos relacionados à codificação e configuração. Dessa forma, não foi possível avaliar o desempenho da GPU em comparação com as outras abordagens, como OpenMP e OpenMPI.

6. Conclusões

Concluímos que, para paralelização em larga escala em um ambiente distribuído, o OpenMPI tende a apresentar um desempenho superior. No entanto, quando a execução ocorre de forma local, o desempenho do OpenMPI é significativamente reduzido. Além disso, o impacto do uso da GPU no problema de cliques ainda não é conclusivo, sendo necessário mais estudo para avaliar seu desempenho de maneira definitiva.

References

Feofiloff, P., Kohayakawa, Y., and Wakabayashi, Y. (2011). Uma introdução sucinta à teoria dos grafos.