Universidade de São Paulo
Instituto de Física de São Carlos

SFI5822 - Introdução à Programação Paralela (2024)
Prof. Gonzalo Travieso

Trabalho B - Sequencial (Grafos)

Lucas Constante Mosquini - **NºUSP** 11858258

December 9, 2024

Contents

1 Introdução

Como descrito no projeto, receberemos um grafo com direção e sem peso. Também receberemos dados sobre ele, como número de arestas e vértices. Assim, dado a literatura, temos que calcular a eficiência do mesmo, e imrprimi-la, junto ao tempo de cálculo.

Como diretrizes principais: respeitar boas práticas, como visto em aula e especialmente pensar em código que possa ser paralelizado facilmente futuramente, ou seja, evitar usar estruturas complexas como árvores de armazenamento, mesmo que elas tenham um desempenho superior. Outro ponto importante, é que nos foi dito que todos os grafos usados seriam esparsos, o que influenciou a não adesão a vários métodos para gráfos densos.

Por fim, esse pdf se resume em justificar as escolhas visando o desempenho, assim como explicar o funcionamento e ideias bases por trás do código.

2 Version 1.0

Aqui vale fazer a distinção em relação ao trabalho anterior, em que foram feitas várias modificações visando o desempenho. Para o atual trabalho, como somos limitados a bibliotecas nativas do C e do fato de so utilizarmos grafos direcionados, esparsos e sem peso, algoritimos famosos como o de Floyd-Warshall, não ajudariam no ganho de desemepenho. Assim, a proposta foi simples: apenas usar estruturas simples de armazenamento e uma busca em largura, executada N vezes para calcular as distâncias dos vértices, com complexidade O(N(N+M)).

Como alteração visando a correção do trabalho anterior, abaixo apenas listarei a lógica do programa, sem copiá-lo.

O inícios padrão criando estruturas para nodes e para o graph em si. Os nodes são armazenados numa lista de adjacência, visando melhor desempenho, comparado ao uso de matrizes, uma vez que todos os grafos seriam esparsos. O graph em si, contém uma lista encadeada de nós para cada vértice, representando arestas que saem desses. O uso de uma lista encadeada facilita a manipulação das arestas.

Uma função específica para clean de memória do graph, algo que é essencial para grafos grandes, como vimos nos exemeplos.

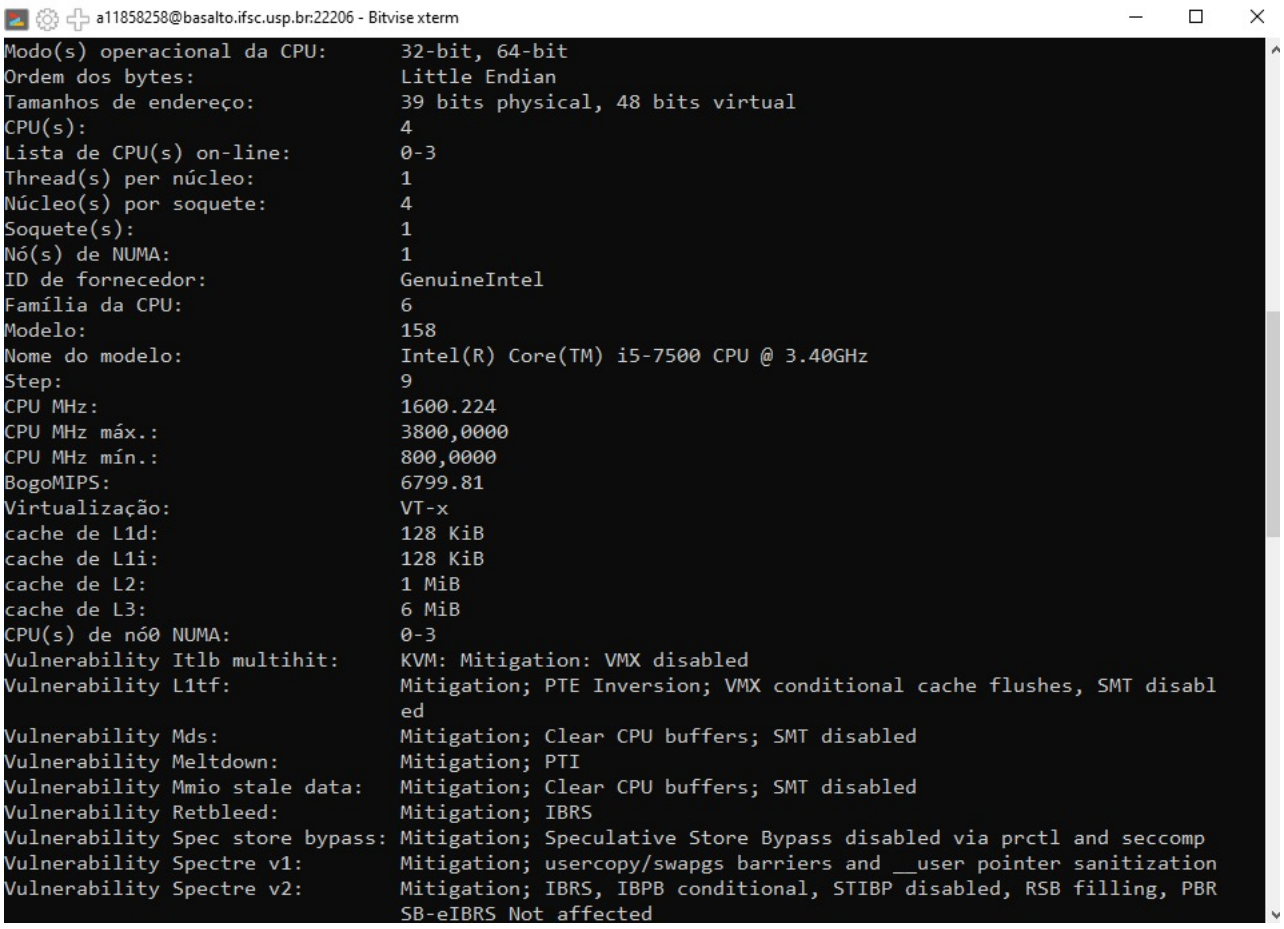
Finalmente a função mais vital do programa: a busca em largura (BFS), para calcular a distância, permitindo-nos calcular a efficiência do grafo a seguir. Ela é a escolha mais eficiente, visto que os grafos são esparsos e sem peso. Além disso, ele pode ser facilmente paralelizada usando MPI, como vimos em aula. Por fim, fala ressaltar que existe uma modificação possível para ela que pode aumentar o seu desempenho: ao invés de começarmos somente de cima para baixo, podemos começar com uma busca de cima para baixo, simultaneamente a uma de baixo para cima nos galhos, parando quando elas se encontram. Eu testei esse algortimo para grafos grandes, mas até o grafo maior que nos foi fornecido nos exemplos, não mudou em quase nada o desempenho. Assim, vou "guardar" essa proposta para quando paralelizarmos esse trabalho, no fim da disciplina. Essa abordagem provavelmente aumentará o desemepnho e também talvez trará vantagens em sincronização dos threads.

O cálculo da efficiência de acordo com o que nos foi pedido no projeto. Veja que ela também pode ser facilmente paralelizada por threads ou processos.

A main do progama, usando tudo que foi desenvolvido e seguindo o proposto no projeto. Veja que temos tratamento de excessões, assim como liberação de memória usada. Por fim, temos o tracking do timing de calculo pedido.

3 Desempenho

Para os cálculos, foi usado o processador abaixo:



Vale notar que na linha de saida, o tempo de cálculo sempre varia. Isso acontece por diversos fatores, como visto em aula: diferente tempos de latência entre outros hardwares com a CPU, localidades (principalmente temporal para as memórias na escrita e leitura), aquecimento e variação de frequência da CPU, distribuição de uso de outras aplicações e etc.

Assim, inicialmente é feito um "aquecimento" do código, rodando ele algumas vezes, com o intuito de minimizar esses efeitos. Em seguida, é tirado os valores da linha de saída 5 vezes. Disso, é feita a média dessas variáveis e então mostradas na tabela abaixo .

Diferentemente do primeiro trabalho, usar os parâmetros padrões de compilação da disciplina não mudou muito o desemepenho. Assim, testei alguns outros parâmetros como: -03, para maximização do desemepenho , O-fast, mais agressivo, -march=native para levar em conta a arquitetura e -funroll-loops para loops intensos. Assim a tabela abaixo para os valores medidos:

	Compilação Disciplina (s)	Compilação Maximizada (s)
Tempo de cálculo	0.1206	0.097435

Table 1: Tabela comparando compilações da disciplina e maximizada pelos parâmetros adcionais. Para cada caso, foi usado o arquivo $d_{52}.net$ dos exemplos.

Nos exemplos fornecidos pelo professor, temos diferentes tamanhos de grafos, mas sempre respeitando os tipos definidos anteriormente. Assim, deixo valores de desempenho para os diferentes arquivos:

	a01 (10,90)	b13 (100,1250)	c59 (10000, 5000000)	d52 (10000,10002)	d59 (10000,2555627)
Tempo	0.000009	0.000954	184.378409	0.097904	106.457077

Table 2: Tabela comparando os arquivos exemplos. Depois do nome do arquivo, entre parenteses, temos o número de vértices e arestas, respectivamente.

Por fim, abaixo deixo um gráfico para facilar a visualização dos dados da tabela: