GCC177 - Trabalho 2

Nome: *Arthur H. S. Cruz* ***Matrícula:*** *201611484*
Turma: *10A*

Nome: *Eduardo F. de Lima* ***Matrícula:*** *201611003*
Turma: *10A*

Nome: *João Pedro T. Silva* ***Matrícula:*** *201610990*
Turma: *10A*

Programação Paralela e Concorrente
Prof. Marluce Pereira
2018/02

Universidade Federal de Lavras
Departamento de Ciência da Computação

1 Descrição do problema

O Algoritmo de Prim é um algoritmo ganancioso desenvolvido pelo checo Vojtech Jarník em 1930 e republicado por Robert C. Prim em 1957, tendo como objetivo encontrar uma árvore geradora mínima em um grafo ponderado e unidirecional.

O algoritmo busca achar um conjunto de arestas que forme uma árvore que inclua todos os vértices do grafo, onde o peso de todas as arestas seja minimizado. O Algoritmo de Prim, originalmente, constrói a árvore adicionando um vértice de cada vez e iniciando de um vértice arbitrário. O vértice adicionado é aquele mais próximo da árvore geradora mínima, sendo então necessário percorrer as areas da árvore para encontrar esse vértice.

Visto que originalmente o Algoritmo de Prim foi concebido para ser executado de forma sequencial, o objetivo deste trabalho é apresentar uma implementação paralela utilizando a biblioteca MPI [1] e a linguagem de programação Python

2 Descrição da implementação paralela

Na implementação do trabalho, foi utilizada uma matriz de adjacência de tamanho $N \times N$ para representar o grafo. A estratégia de paralelização foi dividir o grafo em P conjuntos (Onde p é o número de processos) de tamanho $\frac{N}{P}$

Entretanto, podem existir momentos no qual o número de vértices não será divisível pelo número de processos. A estratégia para contornar esse problema é aumentar em +1 o número de colunas para cada processo caso o índice do processo seja menor que o resto da divisão $\frac{N}{P}$

O algoritmo irá iniciar dividindo o grafo em matrizes de tamanho $N \times \frac{N}{P}$ entre os P processos. O vértice 0 é inserido na árvore e o algorítimo procede da seguinte forma:

1º O processo com rank igual à zero envia em broadcast para todos os processos a atual árvore geradora mínima

2º Cada processo indica qual vértice está mais próximo da árvore geradora mínima em sua partição

3º O processo com rank igual à zero recebe informação de todos os processos através da função gather e determina qual é o melhor vértice no contexto global. O melhor vértice e sua aresta são então adicionados na árvore geradora mínima.

4º Se todos os vértices estiverem na árvore geradora mínima então o algoritmo termina. Se não, volta ao primeiro passo.

3 Resultados

Foram realizados testes com diversas instâncias para um, dois e quatro processos. Cada instância foi executada dez vezes e a média de tempo foi calculada. Os testes foram realizados em um computador com processador Core I7 e 8gb de memória RAM. A média dos

resultados são apresentados na tabela abaixo.

Média dos tempos dos testes realizados			
Num vértices/Num arestas	1 processo	2 processos	4 processos
9/13	0.075	0.083	0.761
20/50	0.077	0.081	0.126
20/49	0.075	0.083	0.124
50/135	0.082	0.084	0.123
50/136	0.083	0.085	0.124
100/286	0.138	0.120	0.178
100/284	0.140	0.158	0.168
200/579	0.563	0.362	0.441
200/583	0.565	0.359	0.462
250/729	0.983	0.639	0.753
250/733	0.995	0.604	1.225
500/1479 (1)	7.798	4.355	5.078
500/1479 (2)	7.834	4.217	4.600
1000/2978	62.251	35.997	36.470
1000/2977	62.832	35.364	35.666

Speedup e Eficiência obtidos nos testes			
Num vértices/Num arestas	*Speedup*/Eficiência 1 processo	*Speedup*/Eficiência 2 processos	*Speedup*/Eficiência 4 processos
9/13	1.0/1.0	0.903/0.452	0.098/0.025
20/50	1.0/1.0	0.944/0.472	0.612/0.153
20/49	1.0/1.0	0.909/0.455	0.611/0.153
50/135	1.0/1.0	0.973/0.487	0.667/0.167
50/136	1.0/1.0	0.982/0.491	0.673/0.168
100/286	1.0/1.0	1.557/0.578	0.779/0.195
100/284	1.0/1.0	0.888/0.444	0.836/0.209
200/579	1.0/1.0	1.557/0.778	1.277/0.319
200/583	1.0/1.0	1.573/0.786	1.224/0.306
250/729	1.0/1.0	1.537/0.769	1.305/0.326
250/733	1.0/1.0	1.647/0.823	0.812/0.203
500/1479 (1)	1.0/1.0	1.790/0.895	1.536/0.384
500/1479 (2)	1.0/1.0	1.858/0.929	1.703/0.426
1000/2978	1.0/1.0	1.729/0.865	1.707/0.427
1000/2977	1.0/1.0	1.777/0.888	1.762/0.440

4 Conclusão

É possível observar que o algoritmo obteve uma melhora considerável com o aumento de processos em grafos grandes, contudo, em grafos menores pode se observar, até mesmo, piora no valor do *speedup*. Isso se dá por devido ao *overhead* paralelo gerado pela comunicação de processos e espera por respostas (O *gather* para processo 0 faz com que ele necessite esperar pela mensagem de todos os outros processos). Em grafos pequenos o tempo gasto no *overhead* somado ao próprio tempo de execução do algoritmo supera o tempo da execução em um único processo.

Com isso, a paralelização do algoritmo de Prim só se mostra benéfica caso seja aplicada em um grafo com muitos vértices e arestas. Para grafos com poucos vértices e arestas, o mais recomendável é que seja executada em apenas por um processo.

As instâncias utilizadas e a implementação propriamente dita pode ser encontrada no Github [2]

Referências

[1] GONINA,E. ; Kalé, Laxmikant V. Parallel Prim's algorithm on dense graphs with a novel extension. Departament of Computer Science; University of Illinois at Urbana-Champaign, 2007.

[2] Código fonte da implementação disponível em: `https://github.com/LimaEduardo/prim-python`