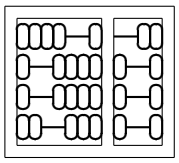

Universidade Estadual de Campinas - UNICAMP
Instituto de Computação - IC

MO644a - Relatório Final

Problema do *k*-plex Máximo

ITALOS ESTILON DA S. DE SOUZA - RA192864
MAURO ROBERTO COSTA DA SILVA - RA192800

Campinas
2017

1 Descrição do Problema

O Problema da Clique Máxima (PCM) pode ser apresentado da seguinte maneira: dado um grafo simples não-direcionado $G = (V, E)$, encontre $C \subseteq V$ com cardinalidade máxima tal que $\forall u, v \in C, \{u, v\} \in E$ (ou seja, C é uma clique). O PCM é um problema NP-difícil. Em termos práticos, o PCM possui um grande número de aplicações em áreas como: bioinformatica, processamento de imagens, design de circuitos quânticos e no projeto de sequências de DNA e RNA para a computação biomolecular [Tomita and Seki, 2003].

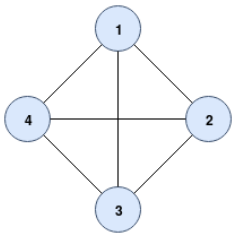

Figura 1: Exemplo de um k-plex para $k = 1$. É fácil ver que toda clique é um 1-plex.

O problema do k-plex máximo é uma relaxação do problema da clique máxima [Trukhanov et al., 2013] e pode ser apresentado da seguinte maneira: dado um grafo não-direcionado $G = (V, E)$, encontre $S \subseteq V$ com cardinalidade máxima tal que $\forall v \in S, |\Gamma(v) \cap S| \geq |S| - k$, em que $\Gamma(v)$ representa o conjunto de vértices adjacentes de v.

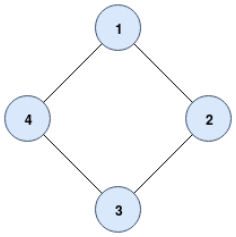

Figura 2: Exemplo de um k-plex para $k = 2$.

2 Descrição da Região Sequencial

O algoritmo sequencial é o BITPLEX, descrito no artigo [Silva et al., 2017]. BITPLEX (chamado de basicPlexBranching no código) executa um *Branch-and-Bound* em profundidade, semelhante a uma *Depth-first search* (DFS). Em cada chamada recursiva do *Branch-and-Bound* as funções para gerar conjuntos candidatos (generate, make_saturated_list e isPlex) e a função de cálculo de limite superior (branching) são executadas. O *profiling* da Tabela 1 deixa claro que essas são as funções de maior custo no algoritmo. Dessa forma, podemos paralelizar as chamadas recursivas do *Branch-and-Bound* (basicPlexBranching) com o objetivo de reduzir o tempo total do algoritmo.

O *profiling* foi executado com gprof para a instância "brock_200-2"e com $k = 3$.

Tabela 1: Resultado do *profiling* usando gprof

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
41.26	1123.41	1123.41	3157887317	355.75	682.09	generate
37.85	2153.97	1030.56	3157887317	326.35	326.35	make_saturated_list
17.53	2631.28	477.31	3157887318	151.15	151.15	branching
3.41	2724.20	92.92				basicPlexBranching
0.03	2725.04	0.84				isPlex
0.02	2725.58	0.54				degree_order
0.00	2725.61	0.04				frame_dummy
0.00	2725.61	0.00	1			_GLOBAL__sub_I_Vnbr

Foi utilizada uma máquina na *Azure* para realização dos testes e do *profiling*. A descrição da máquina está na Tabela 2.

Tabela 2: Saída do comando *lscpu* na máquina da utilizada na *Azure*

Architecture	x86_64	**NUMA node(s)**	2	**Hypervisor vendor**	Microsoft
CPU op-mode(s)	32-bit, 64-bit	**Vendor ID**	GenuineIntel	**Virtualization type**	full
Byte Order	Little Endian	**CPU family**	6	**L1d cache**	32k
CPU(s)	16	**Model**	63	**L1i cache**	32k
On-line CPU(s) list	0-15	**Model name**	Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40GHz	**L2 cache**	256k
Thread(s) per core	1	**Stepping**	2	**L3 cache**	30720k
Core(s) per socket	8	**CPU MHz**	2397.206	**NUMA node0 CPU(s)**	0-7
Socket(s)	2	**BogoMIPS**	4794.37	**NUMA node1 CPU(s)**	8-15

3 Como o problema foi paralelizado

Duas abordagens paralelas foram desenvolvidas, visando identifar qual a melhor. As duas utilizam *pthreads*, pois asssim, é possível ter maior controle das threads o que facilitou o desenvolvimento, dado que o *Branch-and-Bound* é recursivo.

3.1 Pool de Threads e criação de Tarefas (Stack)

Nessa abordagem foi desenvolvido um pool de threads, onde cada chamada recursiva era adicionada à uma pilha de tarefas e as threads do pool ficavam responsáveis por executar essas chamadas. A Figura 3.1 apresenta graficamente como foi realizada a paralelização nessa abordagem.

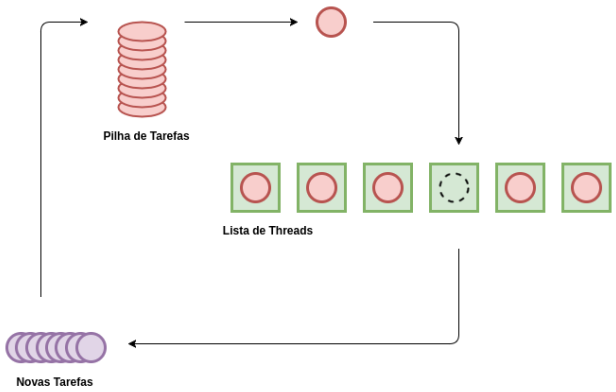

Figura 3: Funcionamento do Stack

Quando uma thread executa uma tarefa ela também adiciona novas tarefas na pilha, que são as chamadas recursivas que devem ser feitas a partir da tarefa executada.

Foi utilizada uma pilha para tentar aproximar essa solução de uma busca em profundidade, pois uma fila iria realizar uma busca em largura e isso faria com que o problema demorasse muito para convergir para a solução ótima (que está no vértice de maior profundidade da árvore de recursão).

Inicialmente tentamos usar uma fila de prioridade, que ordenaria as tarefas por profundidade na árvore de recursão. Essa ideia foi descartada, pois o custo de inserção e remoção dos elementos fazia muita diferença quando o número de tarefas crescia.

3.2 Recursão e Balanceamento de Carga (Load)

Essa solução é semelhante a primeira, pois também possuímos um pool de threads e adicionamos chamadas recursivas nesse pool. Porém, nessa abordagem cada tarefa é retirada do pool e executada recursivamente por uma thread.

Observe que é possível que algumas threads terminem a execução e que outras continuem executando por muito tempo. Para resolver esse problema, antes de fazer uma chamada recursiva cada thread verifica se existe alguma thread desocupada, caso exista, as chamadas recursivas são adicionadas à pilha de tarefas, caso todas estejam trabalhando, a chamada recursiva é executada pela própria thread.

A Figura 3.2 apresenta graficamente como foi realizada a paralelização nessa abordagem.

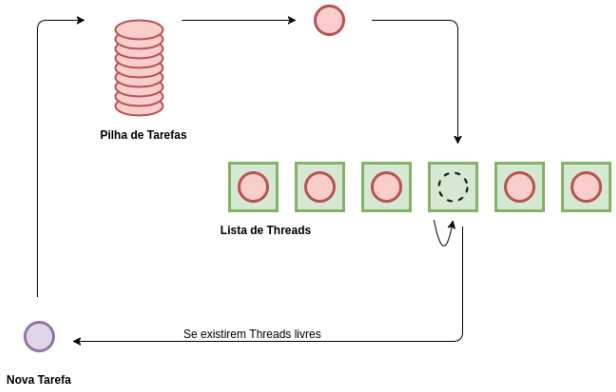

Figura 4: Funcionamento do Load

Essa solução permitiu diminuir o *overhead* de inserir e remover da pilha, pois grande parte das tarefas geradas são executadas pela própria thread que as gerou.

4 Experimento

Os testes foram executados para as instâncias "c_fat500-1", "c_fat500-2", "p_hat300_1"e "brock200_2". Essas instâncias foram obtidas do *benchmark* do DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) e são instâncias amplamente utilizadas na literatura.

Como o problema do k-plex máximo é NP-difícil, encontrar a solução ótima pode levar muito tempo, assim, os testes foram executados com tempo limite de 1 hora (3600 segundos).

As Tabelas 3 e 4 apresentam os resultados dos testes. A notação $\omega_k(G)$ representa o maior k-plex encontrado e NT é o número de threads utilizadas. Observe que $\omega_k(G)$ pode não ser a solução ótima quando o tempo limite de 3600 segundos é alcançado.

Tabela 3: Resultado dos Testes com $k = 2$

		Load		**Stack**		**Serial**	
Instância	**NT**	$\omega_2(G)$	**Tempo**	$\omega_2(G)$	**Tempo**	$\omega_2(G)$	**Tempo**
brock200_2	2	13	8.99s	13	18.32s	13	13.68s
	4	13	4.94s	13	12.83s		
	8	13	3.50s	13	11.48s		
	16	13	13.29s	13	13.08s		
c-fat500-1	2	14	0.22s	14	0.20s	14	0.31s
	4	14	0.14s	14	0.14s		
	8	14	0.09s	14	0.10s		
	16	14	0.06s	14	0.06s		
c-fat500-2	2	26	2.00s	26	2.68s	26	2.14s
	4	26	2.79s	26	1.76s		
	8	26	0.54s	26	1.31s		
	16	26	1.04s	26	1.24s		
p_hat500-1	2	12	19.75s	12	36.09s	12	27.10s
	4	12	10.13s	12	26.96s		
	8	12	5.31s	12	20.05s		
	16	12	13.68s	12	13.58s		

Tabela 4: Resultado dos Testes com $k = 3$

		Load		Stack		Serial	
Instância	**NT**	$\omega_3(G)$	**Tempo**	$\omega_3(G)$	**Tempo**	$\omega_3(G)$	**Tempo**
brock200_2	2	16	2359.96s	16	2984.80s	16	3008.67s
	4	16	1220.78s	16	2224.23s		
	8	16	622.32s	16	1696.26s		
	16	16	1090.00s	16	1797.82s		
c-fat500-1	2	14	21.67s	14	24.40s	14	39.66s
	4	14	11.14s	14	13.47s		
	8	14	5.69s	14	7.55s		
	16	14	2.81s	14	3.68s		
c-fat500-2	2	26	23.34s	26	29.48s	26	39.75s
	4	26	11.90s	26	19.77s		
	8	26	6.53s	26	12.14s		
	16	26	3.17s	26	10.00s		
p_hat500-1	2	13	$\geq$ 3600s	13	$\geq$ 3600s	14	$\geq$ 3600s
	4	14	2351.40s	13	$\geq$ 3600s		
	8	14	1005.80s	14	2102.94s		
	16	14	656.97s	14	2076.30s		

Os resultados mais expressivos foram alcançados com $k = 3$ (Tabela 4). Observe, por exemplo, os resultados encontrados para a instância "p_hat500-1", em que o algoritmo serial não conseguiu resolver todas as instâncias dentro do tempo limite.

4.1 Comparação de SpeedUp

O speedup foi calculado dividindo o tempo serial pelo tempo paralelo. Em algumas execuções o tempo limite foi atingido antes de o problema ser resolvido. Nesses casos, foi utilizado o tempo de 3600 segundos para calcular o speedup. As cores nos gráficos a seguir representam o número de threads.

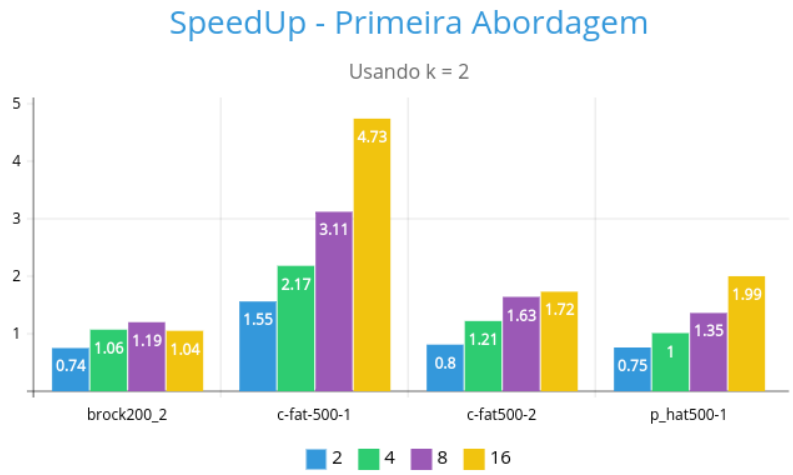

Figura 5: SpeedUp coma primeira abordagem para $k = 2$.

Na Figura 4.1 é apresentado o speedup conseguido pela abordagem Stack com $k = 2$ para as instâncias "brock200_2", "c-fat-500-1", "c-fat-500-2"e "p_hat500-1". Exceto para a primeira

instância, aumentar o número de threads aumentou o speedup, especialmente para a instância "c-fat-500-1".

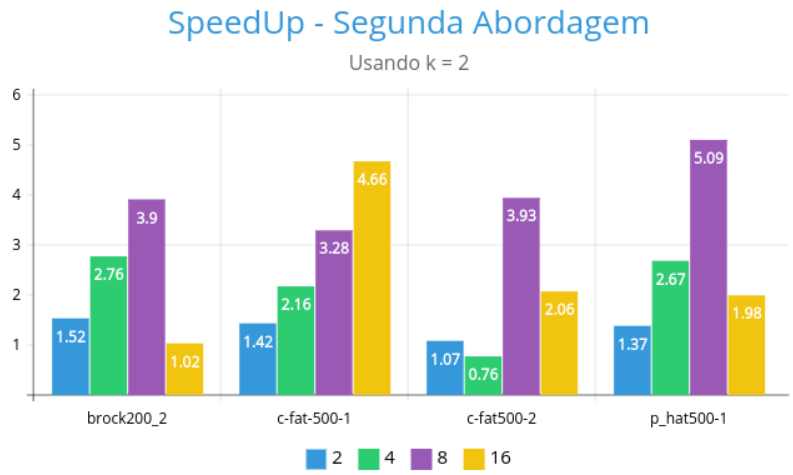

Figura 6: SpeedUp coma segunda abordagem para $k = 2$.

Na Figura 4.1 é apresentado o speedup conseguido pela abordagem Load com $k = 2$ para as instâncias "brock200_2", "c-fat-500-1", "c-fat-500-2"e "p_hat500-1". Exceto para a segunda instância, aumentar o número de threads para mais de 8 threads não aumentou o speedup. Talvez isso tenha acontecido pelo fato de que as árvores de busca dessas instâncias terem ramos muito longos.

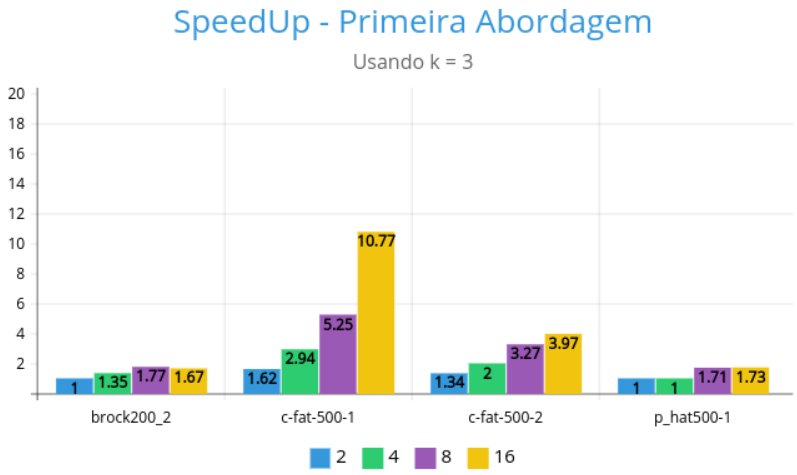

Figura 7: SpeedUp coma primeira abordagem para $k = 3$.

Na Figura 4.1 é apresentado o speedup conseguido pela abordagem Stack com $k = 3$ para as instâncias "brock200_2", "c-fat-500-1", "c-fat-500-2"e "p_hat500-1". Exceto para a segunda instância, o speedup aumentou pouco com o aumento de threads. Com $k = 3$ o árvore de busca fica muito grande, isso deve ter causado um *overhead* de acesso à pilha.

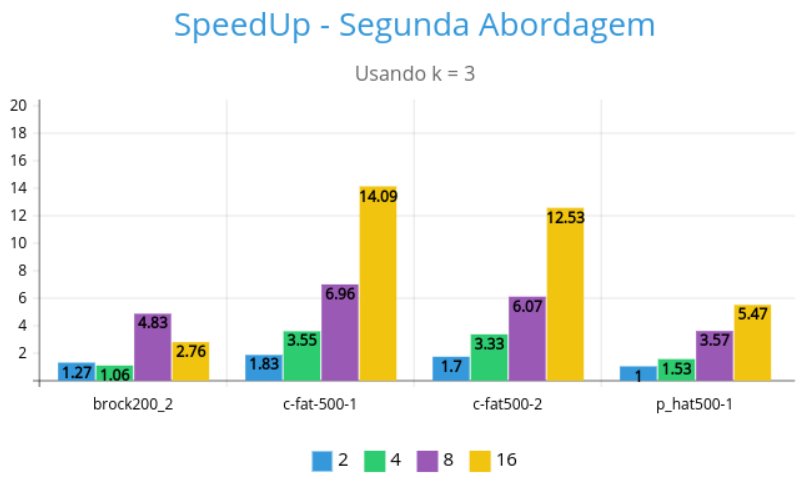

Figura 8: SpeedUp coma segunda abordagem para $k = 3$.

Na Figura 4.1 é apresentado o speedup conseguido pela abordagem Load com $k = 3$ para as instâncias "brock200_2", "c-fat-500-1", "c-fat-500-2"e "p_hat500-1". Esses foram os melhores speedups conseguidos, as árvores de buscas para essas instâncias, embora grandes, devem ter ramos que permitiam que e o trabalho fosse melhor divido entre as threads.

5 Dificuldades Encontradas

O gerenciamento das threads foi a principal dificuldade encontrada, pois, durante boa parte do trabalho, encontrou-se muitos problemas para notificar as threads de forma eficiente o momento de parar a execução. Reproduzir os erros foi trabalhoso, porque foi difícil determinar os estados que causavam falhas.

Referências

[Silva et al., 2017] Silva, M. R. C., Tavares, W., Dias, F., and Neto, M. (2017). Algoritmo branch-and-bound para o problema do k-plex máximo.

[Tomita and Seki, 2003] Tomita, E. and Seki, T. (2003). An efficient branch-and-bound algorithm for finding a maximum clique. In *Discrete mathematics and theoretical computer science*, pages 278–289. Springer, Dijon, France.

[Trukhanov et al., 2013] Trukhanov, S., Balasubramaniam, C., Balasundaram, B., and Butenko, S. (2013). Algorithms for detecting optimal hereditary structures in graphs, with application to clique relaxations. *Computational Optimization and Applications*, 56(1):113–130.