Avaliação de desempenho do algoritmo PageRank sequencial e paralelo

THIAGO DOS SANTOS GONÇALVES*, Instituto de Informática - UFRGS, Brasil
JOÃO VITOR DO AMARAL SPOLAVORE*, Instituto de Informática - UFRGS, Brasil

Este trabalho apresenta uma avaliação de desempenho do algoritmo PageRank, utilizado por muito tempo pelo indexador do *Google*, testado com execução sequencial e paralela. Para isso, esse estudo analisa o comportamento do algoritmo sob diferentes configurações de execução em arquiteturas paralelas. A metodologia experimental consistiu na execução do algoritmo com 9 variações de contagem de *threads*, utilizando 7 grafos de entrada distintos, alternando o estado do *Hyper-Threading*, aplicando diferentes políticas de vinculação de *threads*, e fazendo diferentes análises com o *Intel Vtune Profiler*. Os resultados obtidos demonstram o impacto dessas configurações no tempo de execução e tentam correlacionar o tempo de execução com algumas métricas coletadas e com diferentes variações de configurações. Observamos valores de *speedup* de até 15× dependendo da estratégia de paralelismo adotada para execução.

Palavras Chave: PageRank, Google, Computação Paralela, Computação de Alto Desempenho, Algoritmos, Grafos, Avaliação de Desempenho

1 Introdução

1.1 Descrição do Objeto de Pesquisa

O algoritmo PageRank, originalmente proposto por Sergey Brin e Larry Page para o motor de busca do *Google*[6], é um método de análise de *links* que atribui pesos numéricos a cada elemento de uma coleção de documentos ligados, com o objetivo de medir a sua importância relativa. A internet seria modelada como um grafo direcionado $G = (V, E)$, onde os vértices V representam as páginas e as arestas E representam os *hyperlinks*. O conceito central do algoritmo se baseia na ideia de que uma página é importante se for citada por outras páginas importantes.

Matematicamente falando, o cálculo do valor do PageRank (PR) de uma página u é definido de forma recursiva. A equação que define a distribuição de probabilidade do "surfista aleatório" (*random surfer*) é dada por:

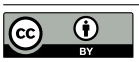
$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)} \tag{1}$$

onde d é o fator de amortecimento, N é o número total de páginas, B_u é o conjunto de páginas que apontam para u, e $L(v)$ é o número de links saindo da página v.

Do ponto de vista computacional, que é o foco deste trabalho, o cálculo do PageRank é resolvido através do "Método das Potências"[8]. Este processo envolve a multiplicação iterativa de um vetor de classificação por uma matriz que representa a estrutura do grafo até que os valores entrem em convergência dentro de uma margem de erro predefinida. Devido o enorme volume de dados da

*Ambos os autores contribuíram igualmente neste trabalho.

Informação de Contato dos Autores: Thiago dos Santos Gonçalves, tsgoncalves@inf.ufrgs.br, Instituto de Informática - UFRGS, Porto Alegre, Rio Grande do Sul, Brasil; João Vitor do Amaral Spolavore, joao.spolavore@inf.ufrgs.br, Instituto de Informática - UFRGS, Porto Alegre, Rio Grande do Sul, Brasil.



ACM /2025/12-ART
https://doi.org/

matriz esparsa da internet, este algoritmo demanda intenso processamento de ponto flutuante e acesso à memória dependendo da entrada. Essas características fazem com que o PageRank seja um bom algoritmo para estudo de escalabilidade paralela, justificando nosso estudo do comportamento da escolha de *threads*, afinidade e *Hyper-Threading* para otimizar o tempo de convergência.

1.2 Objetivos do Trabalho

Com isto definido, nós iremos analisar o desempenho do algoritmo PageRank em uma máquina do Parque Computacional de Alto Desempenho (PCAD) do Instituto de Informática (INF) da Universidade Federal do Rio Grande do Sul (UFRGS) [2]. Para realizar essa análise, utilizaremos o *Intel VTune Profiler* [3], para obter métricas mais aprofundadas de uso dos núcleos durante a execução do algoritmo, como também executaremos sem esse *profiler* pra obtermos o tempo de execução e calcularmos dados com base nele.

Mais especificamente, dividiremos nosso objetivo principal de analisar o algoritmo PageRank nesses sub-objetivos:

- Comparar tempos de execução/*speedup* entre diferentes configurações.
- Associar as métricas adquiridas com possíveis causas pra diferenças nos tempos de execução.

Assim conseguiremos obter mais informações sobre a escalabilidade do algoritmo PageRank e também descobrir o que limita o desempenho do nosso algoritmo dentro do sistema em que ele foi executado.

1.3 Revisão da Proposta e Metodologia

De início, nós planejamos apenas utilizar o *Intel Vtune Profiler* pra obter tanto o tempo de execução quanto as métricas que iríamos utilizar nesse trabalho. Porém, conforme fomos realizando os experimentos, percebemos que executar apenas com o *Intel Vtune Profiler* não nos entregaria os verdadeiros resultados de tempo devido ao *overhead* introduzido. Isso nos levou a realizar mais execuções sem esse *profiler*, realizando mais repetições por configuração pra obter maior precisão em nossos resultados de tempo de execução e dividindo nossa análise entre as execuções com o *profiler* e as execuções sem.

Fora isso, desde o início tínhamos confiança de que nosso projeto seria plausível, apenas não sabíamos exatamente como realizar essa análise do desempenho. Com a ajuda do professor, essa avaliação se tornou possível e seguiu critérios de maior rigor científico.

2 Fundamentação Teórica

O *Intel VTune Profiler* [3] é uma ferramenta pra análise de desempenho que ajuda os desenvolvedores e analistas de *hardware* a otimizar o desempenho de aplicações e sistemas, identificando gargalos em *softwares* que utilizam CPU, GPU ou FPGA. Ele analisa como uma aplicação usa os recursos de *hardware*, mostrando onde o código está mais lento, identificando problemas na memória *cache*, sincronização de *threads* e eficiência na utilização do processador. A ferramenta fornece informações detalhadas e ajuda a encontrar áreas para melhoria de desempenho em códigos seriais e paralelos.

O repositório *GAPBS* [1] é uma coleção de *benchmarks* de algoritmos de operações em grafos consolidada na comunidade acadêmica, citado em ao menos 20 artigos publicados em revistas e conferências revisadas por pares. Ele foi criado por estudantes da universidade de *Berkley* e possui algoritmos como busca por profundidade, contagem de triângulos, e o nosso objeto de pesquisa, o algoritmo PageRank. Esse repositório implementa seus *kernels* em C++ e OpenMP [7], com saídas e entradas simples pro usuário final.

O repositório *Stanford Large Network Dataset Collection* (SNAP) [5] é uma coleção de grafos de entrada da universidade de *Stanford*. Também consolidado academicamente e com vários artigos já

publicados [4], os grafos disponibilizados que representam relações entre páginas da internet ou entre usuários de aplicativos foram essenciais para nosso trabalho.

3 Metodologia

Nós consideramos os seguintes grafos de entrada com as seguintes características:

Table 1. Grafos utilizados nos experimentos

Grafo	Vértices	Arestas
Friendster	65.608.366	1.806.067.135
LiveJournal	3.997.962	34.681.189
Orkut	3.072.441	117.185.083
BerkStan	685.230	7.600.595
Google	875.713	5.105.039
NotreDame	325.729	1.497.134
Stanford	281.903	2.312.497

Utilizamos a implementação do algoritmo PageRank do repositório *GAPBS*, já mencionado anteriormente. Nesta implementação, ele utiliza uma margem de erro de 1^{-4}, e um fator de amortecimento de $0,85$.

Executamos nosso algoritmo na máquina blaise do GPPD. Essa máquina possui 2 processadores *Intel(R) Xeon(R) E5-2699 v4* com $256GB$ de memória RAM DDR4, totalizando 88 *threads* e 44 núcleos operando a $2{,}2GHz$ e com suporte a *Turbo Boost*, levando os núcleos a operarem a até $3{,}6GHz$ com $55MB$ de memória *cache* compartilhada. Então, para nossos experimentos, consideramos apenas 44 *threads* e 22 núcleos. Os experimentos foram executados no *kernel* Linux versão 4.19.0-25, com a versão do gcc 8.3.0, e as métricas foram coletadas com o *Intel Vtune Profiler* versão 2021.1.

Essas foram as configurações consideradas em nossos testes:

Quantidade de threads: 1, 4, 12, 22, 28, 36, 44, 66 e 88 *threads*.

Tipos de análise do *Intel VTune Profiler*: hpc-performance; hotspots; performance-snapshot; nenhuma.

Políticas de vinculação de *threads*: *close*; *spread*.

Estado do *Hyperthreading*: ligado (**HT_ON**); desligado (**HT_OFF**).

Totalizando 144 combinações de configurações diferentes, com nossas entradas possuindo relacionamentos não-lineares entre si. Além disso, para configurações utilizando o *Intel VTune Profiler* ele executa 5 vezes, e para configurações sem o *Intel VTune Profiler* ele executa 10 vezes. Essa escolha foi devido ao *overhead* do *Intel VTune Profiler*, para dar tempo de realizarmos nossos experimentos dentro de um único *batch* da ferramenta slurm.

As métricas que decidimos analisar estatísticamente em nosso trabalho utilizando o *profiler* foram: *Memory Bound*, *CPI Rate*, e *Average CPU Frequency*. A métrica *Memory Bound* é uma porcentagem dos ciclos da execução do programa onde o processador fica esperando operações na memória serem concluídas. A métrica *CPI Rate* é a quantidade de ciclos por instrução média calculada durante toda a execução da aplicação. Por fim, a métrica *Average CPU Frequency* é uma média simples da frequência de operação do processador durante a execução.

Por fim, versionamos nosso código em um repositório *git* disponível em https://github.com/thgdsg/perf-analysis.

4 Resultados

4.1 Métricas Brutas do *Intel VTune Profiler*

Vamos iniciar comentando sobre as métricas coletadas pelo *Intel Vtune Profiler*. O gráfico abaixo é um agrupamento da frequência média de execução do programa, separando por grafo de entrada, quantidade de *threads*, estado do *Hyperthreading*, e política de vinculação de *threads*.

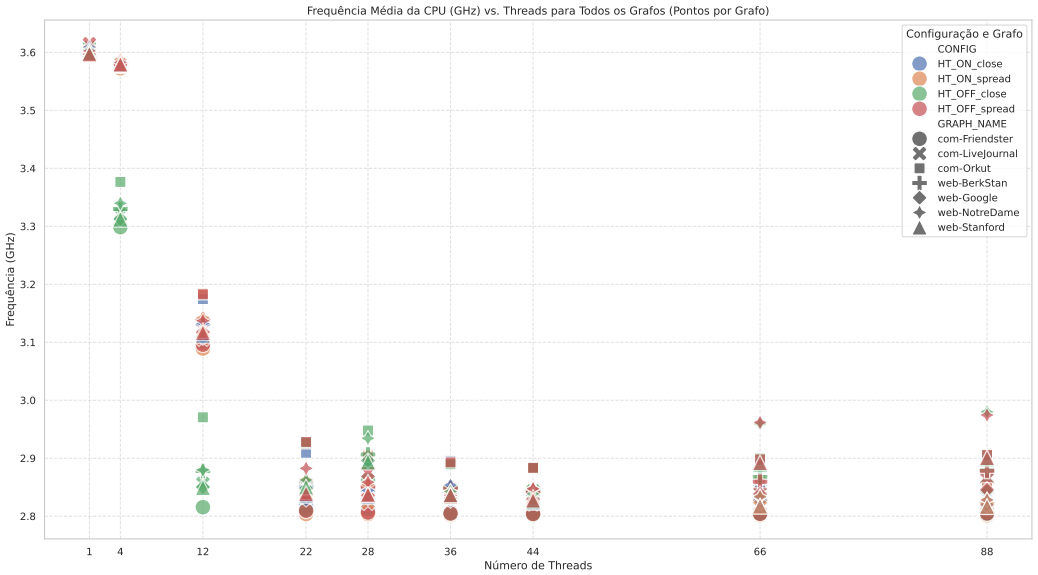


Fig. 1. Gráfico da Frequência Média da CPU por quantidade de *threads* por configuração

No gráfico 1 podemos observar um comportamento similar a uma função exponencial decrescente, com grandes quedas na frequência média nos primeiros aumentos na quantidade de *threads* de execução. Além disso, observa-se que executando com apenas 1 *thread* em todos os casos a frequência média fica próxima a $3.6GHz$, o que é a frequência esperada de *Turbo Boost* do processador. E a partir de 22 *threads*, o processador executa em frequências abaixo de $3.0GHz$. Como ele possui 22 núcleos físicos, essa configuração utiliza todos os núcleos, e por isso atinge limites de gasto energético (Watts) e de temperatura (Celsius), tendo que limitar a frequência de operação dos núcleos para evitar desgastes no processador. Esse métrica nos confirma que o processador estava operando com o máximo de desempenho possível durante a execução de nossa aplicação, e valida nossos outros resultados obtidos.

A tabela 2 mostra a métrica *Memory Bound* pra todos os grafos e configurações combinados. Por exemplo, **H_OFF_C** se refere a configuração *Hyperthreading* desligado com a política de vinculação de *threads close*, podendo variar de 1 a 88 *threads*. A tabela 3 é organizada similarmente a tabela 2.

Como é possível observar na tabela, nenhuma única configuração pra todos os grafos de entrada obteve a melhor ou pior taxa de ciclos ociosos esperando por operações na memória. Isso é particularmente interessante para nossa pesquisa pois uma das nossas hipóteses iniciais era que certas configurações poderiam aumentar bastante a quantidade de uso da memória, o que não foi o caso. Vale acrescentar que o coeficiente de variação médio entre todos os casos foi de aproximadamente 8%, com pouca variação dessa métrica no geral entre *runs* diferentes na mesma configuração.

GRAPH_NAME	CONFIG	THREADS								
		1	4	12	22	28	36	44	66	88
BerkStan	H_OFF_C	9.56	14.38	13.60	14.00	15.08	14.70	16.12	25.90	23.86
	H_OFF_S	9.32	17.98	18.14	16.08	15.26	13.70	15.92	25.06	23.16
	H_ON_C	10.12	4.32	6.14	9.00	9.72	10.24	10.52	12.28	13.30
	H_ON_S	9.04	18.90	19.08	16.70	14.96	14.82	14.94	10.72	13.14
Friendster	H_OFF_C	29.20	33.08	41.74	56.46	66.58	70.16	73.02	73.30	76.84
	H_OFF_S	29.26	40.62	52.08	62.50	67.56	70.90	73.06	73.66	76.20
	H_ON_C	33.50	21.56	19.72	23.86	31.62	37.00	39.58	52.86	68.36
	H_ON_S	28.34	37.18	53.38	61.76	64.26	69.26	71.50	61.56	68.00
Google	H_OFF_C	30.14	28.22	25.26	26.56	32.86	30.52	27.36	51.34	54.28
	H_OFF_S	30.12	41.28	37.56	30.20	35.78	30.50	28.56	50.56	53.44
	H_ON_C	31.86	32.06	29.26	24.34	25.66	22.06	23.30	27.16	24.06
	H_ON_S	31.06	41.20	36.14	31.02	34.06	29.24	30.60	27.24	26.76
LiveJournal	H_OFF_C	17.92	23.26	28.86	35.18	34.40	24.70	37.14	53.06	55.00
	H_OFF_S	18.78	39.52	37.72	27.12	19.16	27.22	38.18	52.30	63.32
	H_ON_C	25.98	17.50	17.42	14.18	19.50	29.40	26.22	25.10	28.94
	H_ON_S	34.28	41.40	32.40	32.06	39.12	41.84	45.62	35.38	34.00
NotreDame	H_OFF_C	6.68	10.30	11.44	19.28	8.84	14.08	17.48	39.74	46.60
	H_OFF_S	6.58	21.50	17.32	16.22	13.88	14.58	15.98	36.82	47.60
	H_ON_C	5.48	8.06	9.00	5.78	6.72	10.58	6.90	8.66	8.34
	H_ON_S	6.30	22.28	17.08	19.20	13.28	13.24	14.68	8.02	7.78
Orkut	H_OFF_C	33.36	33.82	32.70	33.66	44.78	43.50	37.40	60.12	60.00
	H_OFF_S	33.00	50.20	44.56	43.42	43.38	40.50	38.38	59.42	62.92
	H_ON_C	39.28	35.38	32.34	30.94	33.76	28.48	29.16	32.42	34.46
	H_ON_S	32.14	48.22	45.40	44.42	44.94	40.30	37.74	29.70	28.26
Stanford	H_OFF_C	29.60	32.08	26.22	20.92	28.74	24.20	18.94	38.24	41.08
	H_OFF_S	29.96	40.52	35.34	26.64	26.84	23.54	19.88	40.86	40.48
	H_ON_C	29.18	31.00	24.68	24.00	18.82	15.64	18.88	18.82	16.84
	H_ON_S	29.68	39.54	33.78	30.18	30.50	25.06	25.22	14.14	15.64

Table 2. Tabela Combinada de Memory Bound (%) (Valores Ótimos Destacados)

Diferentemente da taxa de *Memory Bound*, a taxa de ciclos por instrução na tabela 3 possui um padrão bem claro, atingindo seu menor valor executando na configuração *single-threaded*, e atingindo seu maior valor executando com 44 ou 88 *threads*. Essa diferenciação entre maior valor em alguns casos está associada a configuração de *Hyperthreading*, pois com ele desligado ele atinge o máximo de *threads* de execução em 44, mas com o *Hyperthreading* ligado o máximo de *threads* é atingido em 88.

Isso apenas não acontece no grafo Friendster, e nossa hipótese é que isso pode ser devido ao grande tamanho do grafo e altos volumes de transferência de memória nessas quantidades de *threads*, algo que pode ser visto devido aos altos valores (>70%) de *Memory Bound* desse caso na tabela 2.

GRAPH_NAME	CONFIG	THREADS								
		1	4	12	22	28	36	44	66	88
BerkStan	H_OFF_C	0.539	0.563	0.678	0.838	0.955	0.996	1.113	0.673	0.687
	H_OFF_S	0.538	0.609	0.719	0.862	0.963	1.003	1.113	0.674	0.689
	H_ON_C	0.549	0.731	0.859	1.029	1.148	1.184	1.300	1.565	1.783
	H_ON_S	0.536	0.601	0.710	0.859	0.957	0.992	1.107	1.501	1.791
Friendster	H_OFF_C	0.663	0.697	0.861	1.314	2.020	2.590	3.104	2.975	3.233
	H_OFF_S	0.667	0.850	1.173	1.692	2.058	2.560	3.113	2.962	3.226
	H_ON_C	0.792	1.148	1.202	1.330	1.592	1.925	2.302	3.830	5.187
	H_ON_S	0.612	0.730	1.101	1.539	1.843	2.375	2.995	4.993	5.845
Google	H_OFF_C	0.936	0.960	0.995	1.100	1.362	1.464	1.591	1.412	1.484
	H_OFF_S	0.935	1.246	1.248	1.306	1.369	1.458	1.626	1.363	1.413
	H_ON_C	0.973	1.747	1.701	1.712	1.742	1.823	1.926	2.153	2.271
	H_ON_S	0.954	1.227	1.228	1.286	1.335	1.425	1.682	2.015	2.288
LiveJournal	H_OFF_C	0.589	0.692	0.926	1.262	1.340	1.479	1.539	1.514	1.400
	H_OFF_S	0.586	0.856	0.974	1.094	1.170	1.408	1.554	1.505	1.399
	H_ON_C	0.633	1.073	1.139	1.336	1.525	1.790	2.026	2.127	2.411
	H_ON_S	0.698	0.871	0.986	1.181	1.382	1.675	1.900	2.649	3.154
NotreDame	H_OFF_C	0.484	0.534	0.789	1.001	1.179	1.328	1.422	0.752	0.793
	H_OFF_S	0.485	0.632	0.855	1.053	1.176	1.326	1.432	0.756	0.797
	H_ON_C	0.487	0.752	1.064	1.272	1.414	1.561	1.705	1.992	2.146
	H_ON_S	0.483	0.628	0.856	1.053	1.176	1.320	1.429	1.897	2.189
Orkut	H_OFF_C	0.706	0.835	0.873	1.057	1.322	1.392	1.465	1.371	1.416
	H_OFF_S	0.712	1.136	1.172	1.170	1.250	1.327	1.469	1.395	1.393
	H_ON_C	0.830	1.301	1.364	1.432	1.496	1.430	1.611	2.051	2.415
	H_ON_S	0.688	1.073	1.134	1.163	1.254	1.343	1.472	1.842	2.301
Stanford	H_OFF_C	0.883	0.953	1.075	1.182	1.465	1.562	1.665	1.217	1.262
	H_OFF_S	0.886	1.178	1.286	1.341	1.451	1.568	1.660	1.222	1.267
	H_ON_C	0.889	1.586	1.627	1.663	1.765	1.890	2.002	2.291	2.392
	H_ON_S	0.883	1.174	1.273	1.333	1.451	1.563	1.657	2.154	2.419

Table 3. Tabela Combinada de CPI Rate (Valores Ótimos Destacados)

4.2 Análise do *Speedup*

Agora iremos comentar sobre o *speedup* obtido executando sem o *Intel Vtune Profiler*, coletado diretamente da aplicação.

Nas figuras 2 a 5 é possível observar o *speedup* obtido conforme o número de *threads* foi aumentando, pra cada grafo de entrada. Além disso, nesse gráfico é possível ver o intervalo de confiança de 95% de cada *speedup* obtido. Na maioria dos gráficos, o intervalo de confiança é tão pequeno que é difícil de perceber, mas em grafos como web-Google e com-Orkut ele é perceptível.

Primeiramente, é possível observar que em algumas das figuras (2, 3, 4, e 5) o maior *speedup* obtido foi com a configuração **HT_OFF_close**, e nos casos dos grafos com-Friendster, web-NotreDame, e web-Stanford essa configuração foi executada com 22 *threads*. Acreditamos que o *Hyperthreading* foi o maior responsável por isso, mas que a política de vinculação de *threads close* também pode ter contribuído. Primeiro, executar com 22 *threads* com o *Hyperthreading* desligado está utilizando 100% da capacidade de um processador físico, sem utilizar *threads* lógicas, que poderiam dividir a memória *cache* do núcleo. Além disso, a política de vinculação de *threads close* nesses casos foi eficaz, conseguindo aproveitar a localidade espacial para obter ganhos no desempenho.

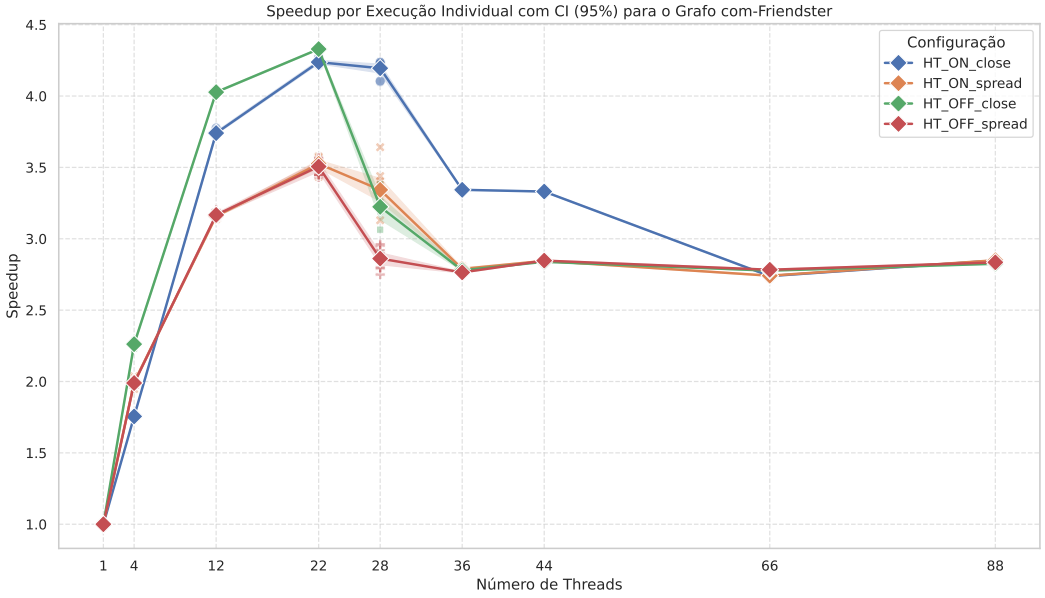


Fig. 2. Análise de Speedup para o grafo com-Friendster.

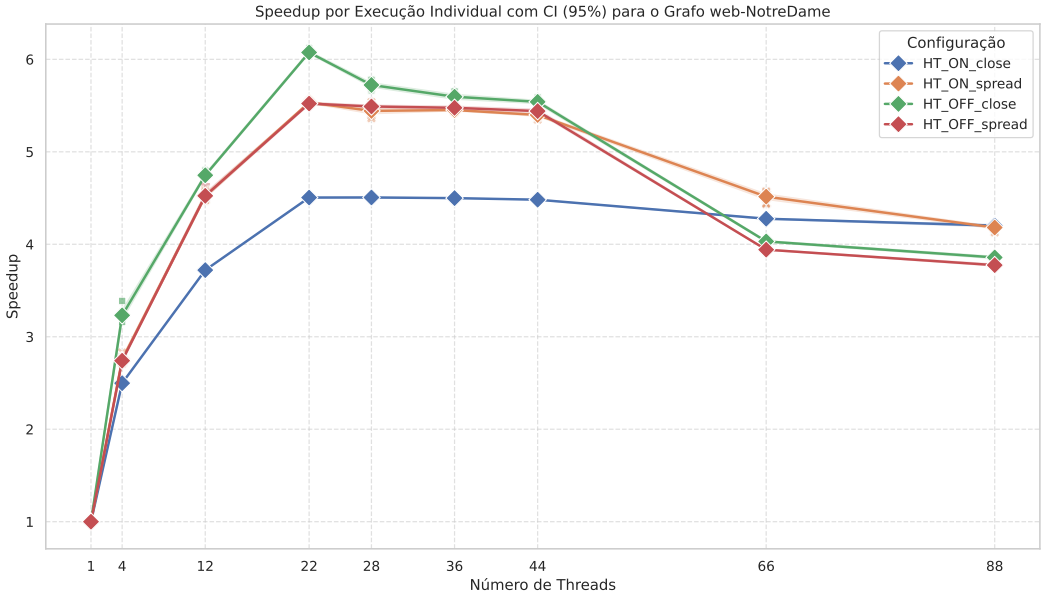


Fig. 3. Análise de Speedup para o grafo web-NotreDame.

Também é possível observar nas figuras 3, 4, 5, 6, e 8 que a política *close* resultou nos menores *speedups* executando com entre 4 a 44 *threads*. Como as *threads* foram alocadas preferencialmente nos mesmos núcleos, isso gerou uma competição por acessos a memória *cache* L2 e L3 (nosso

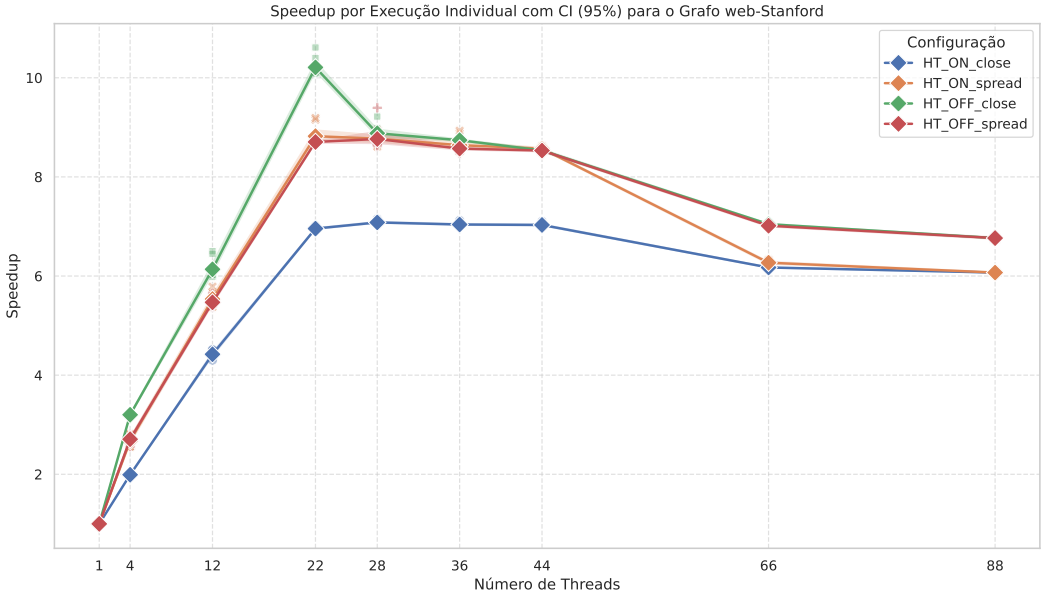


Fig. 4. Análise de Speedup para o grafo web-Stanford.

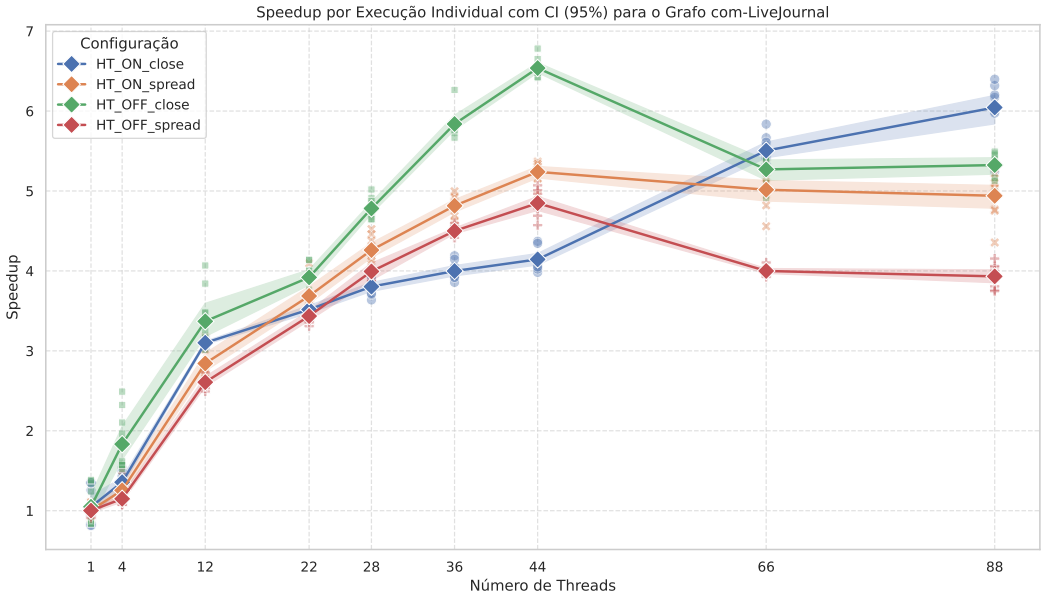


Fig. 5. Análise de Speedup para o grafo com-LiveJournal.

processador não possui memória L1), diminuindo o desempenho atingido por essa configuração. Isto não aconteceu com o grafo web-Friendster e com-Orkut, e nossa hipótese é que foi devido a

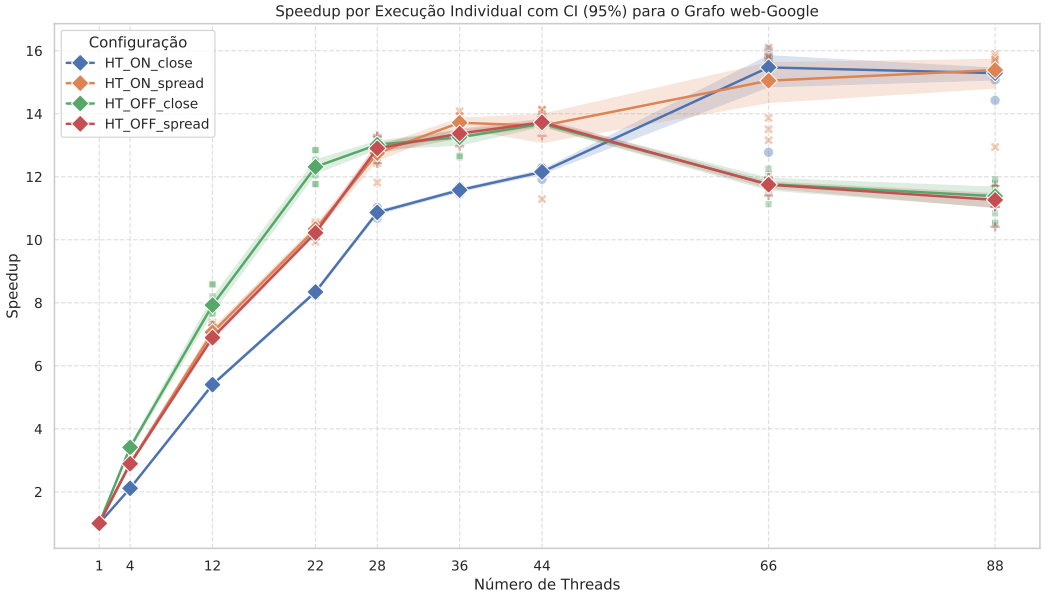


Fig. 6. Análise de Speedup para o grafo web-Google.

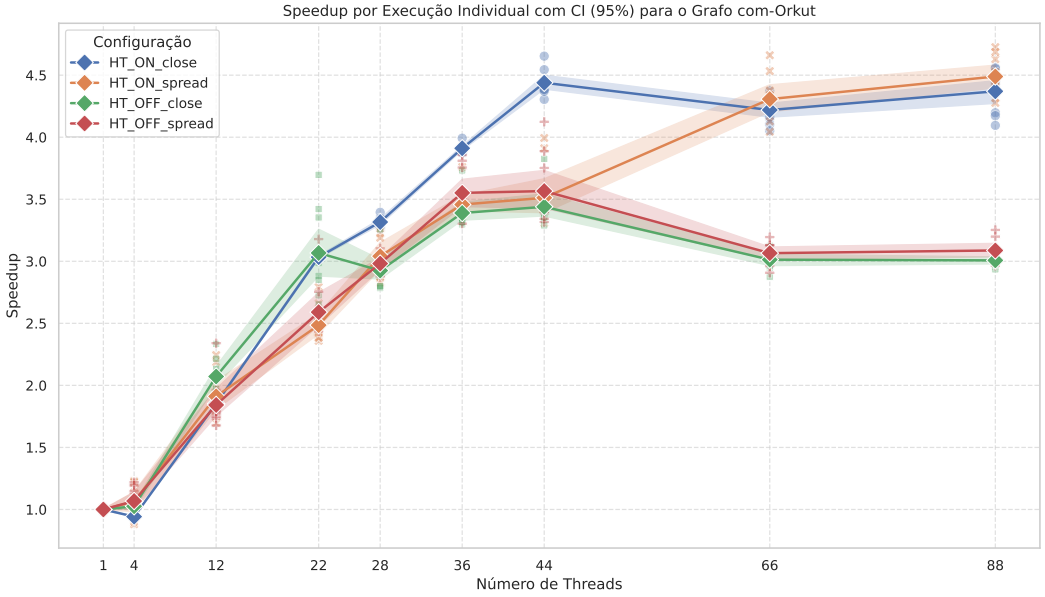


Fig. 7. Análise de Speedup para o grafo com-Orkut.

alta quantidade de arestas proporcionais aos vértices, com o grafo web-Friendster possuindo quase 28× a quantidade de arestas, e o grafo com-Orkut possuindo pouco menos de 39×.

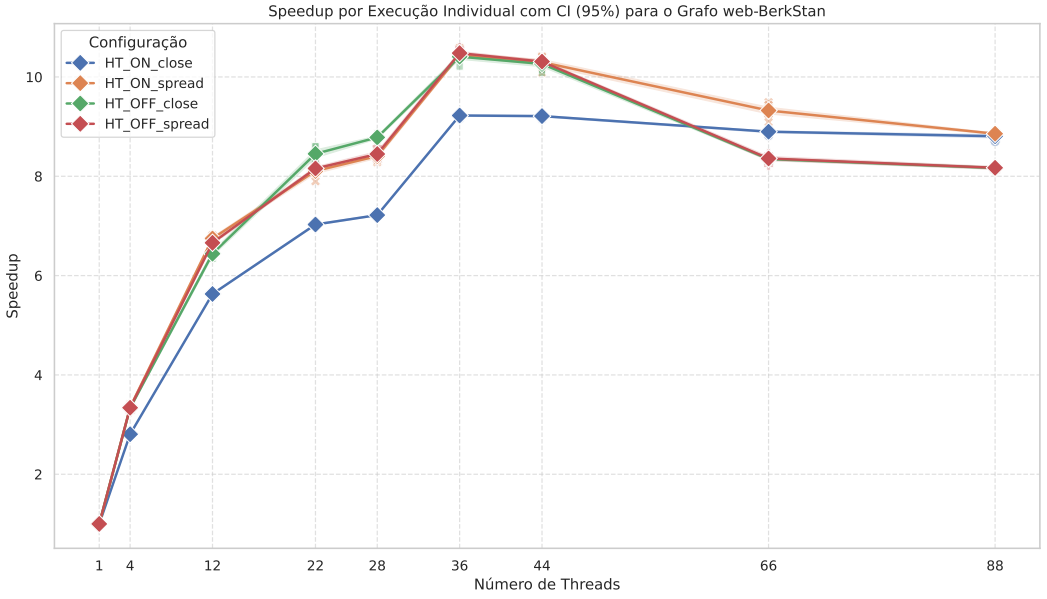


Fig. 8. Análise de Speedup para o grafo web-BerkStan.

4.3 Análise estatística

Por fim, faremos a análise estatística dos nossos resultados, tentando correlacionar algumas variáveis de saída entre si e algumas das nossas variáveis de entrada com as de saída.

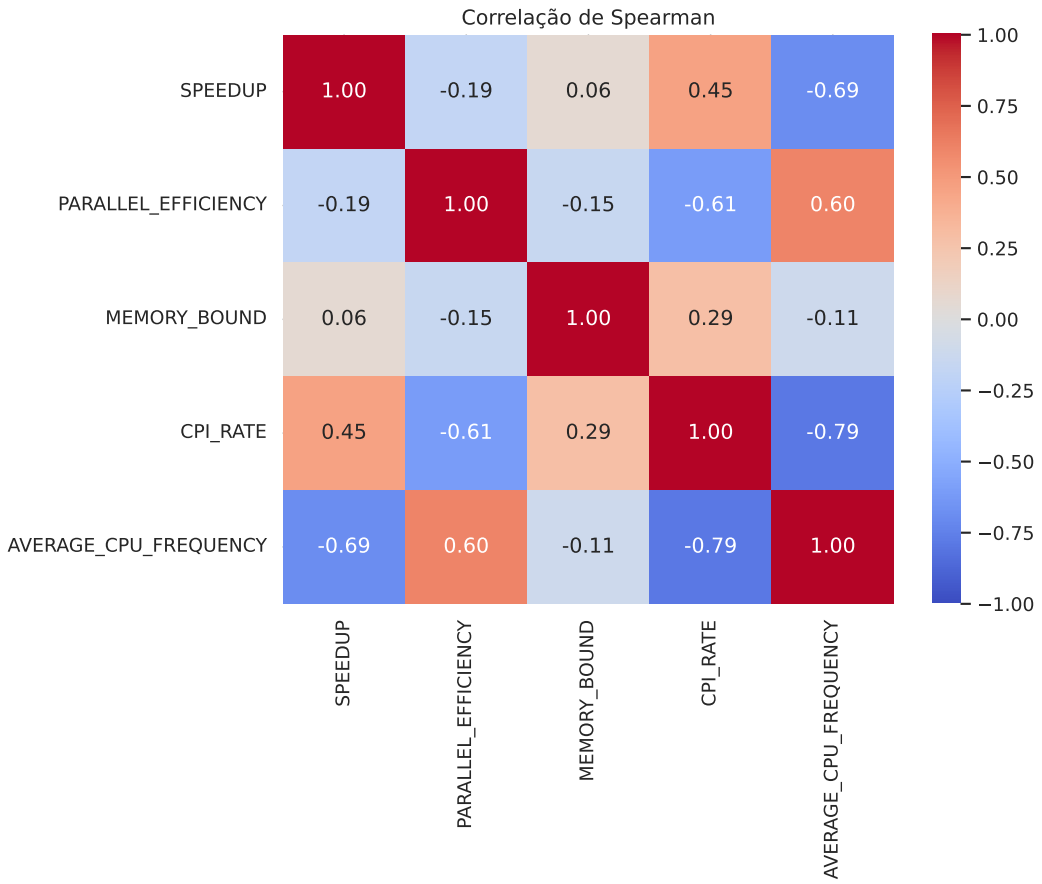


Fig. 9. Gráfico da Correlação de Spearman entre as variáveis

Primeiramente, na figura 9 podemos observar a correlação de *Spearman* entre as variáveis de saída. Consideramos o *Speedup*, *Memory Bound*, *CPI Rate*, *Average CPU Frequency*, e *Parallel Efficiency*. A eficiência paralela é calculada dividindo o *Speedup* pela quantidade de *threads*.

Podemos ver que a taxa de ciclos por instrução possui uma forte correlação positiva com o *speedup* e negativa com a eficiência paralela, condizente com os dados apresentados anteriormente na tabela 3, pois conforme a quantidade de *threads* aumentava, a taxa de ciclos por instrução também aumentava, com o *speedup* aumentando junto até certo ponto (normalmente 22 *threads*). Similarmente, o *memory bound* possuiu uma fraca correlação com a taxa de ciclos por instrução, relacionado com a fraca tendência da aplicação em aumentar o tempo perdido em transferência de memória conforme a quantidade de *threads* aumentava.

Dito isso, tentamos realizar um cálculo da importância das variáveis de saída, exceto a eficiência paralela, pra tentar ver o quanto elas estão relacionadas com o *Speedup* adquirido e o quanto elas estão possívelmente limitando o desempenho do programa. Ou seja, relacionando o comportamento

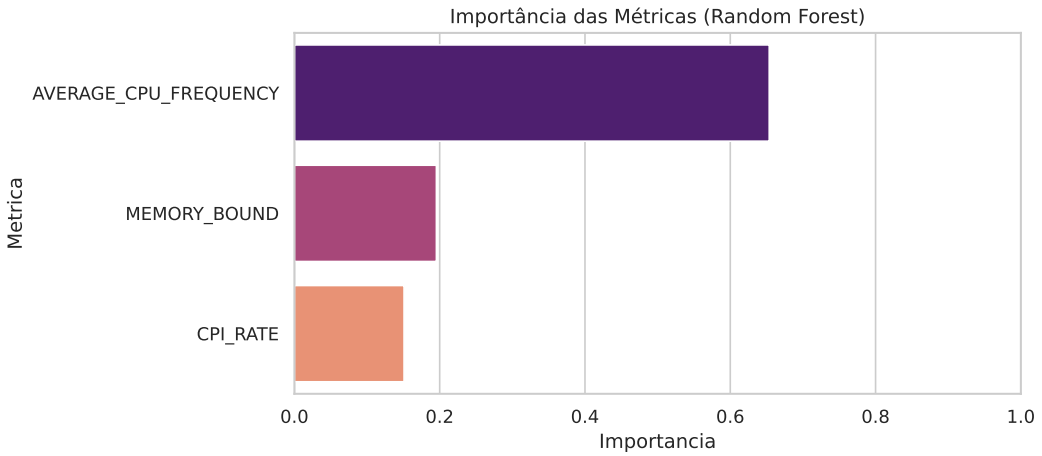


Fig. 10. Gráfico da *Feature Importance* das variáveis calculada por *Random Forest*

da máquina com o desempenho final. Pra isso, utilizamos *Random Forest* com 100 estimadores e semente de execução fixa, e chegamos no resultado visível na figura 10.

Assim, é possível observar que a frequência da CPU tem uma fortíssima importância pro *Speedup* da aplicação, e também que o tempo gasto esperando a memória impacta mais do que os ciclos por instrução, indicando que otimizações na memória podem ser cruciais para melhorar a escalabilidade na aplicação.

Por fim, aplicamos ANOVA pra relacionar nossas variáveis de entrada com as de saída, tanto as métricas, quanto o *speedup* obtido. Calculamos pras variáveis *threads* como controle, *Hyperthreading*, *Binding*, e a interação entre *Hyperthreading* e *Binding*. A coluna *sum_sq* mostra a magnitude do efeito, a coluna *F* é uma razão que mostra o quanto as diferenças são devido a variação na entrada (maior = melhor). A coluna *p-value* mostra a probabilidade desse resultado ser aleatório, e está marcado em verde caso seja menor que 0,05 (5%).

	sum_sq	F	p-value
C(THREADS)	21.194140	764.778120	0.000000
C(HT_OFF)	0.009423	2.720265	0.100390
C(BINDING)	0.040890	11.804060	0.000696
C(HT_OFF):C(BINDING)	0.041835	12.076720	0.000606
Residual	0.831384	NaN	-

Table 4. ANOVA para AVERAGE_CPU_FREQUENCY (GHz)

Na tabela 4 podemos ver que a política de vinculação de *threads* afeta a frequência média. Uma possível razão disso é que manter a política em *close* pode aumentar a temperatura do *chip*, pois núcleos mais próximos fisicamente serão utilizados, atingindo temperaturas máximas mais rápido. E além disso, há interação entre *hyperthreading* e a política escolhida, ou seja, provavelmente ligar o *hyperthreading* e manter a política em *close* seria o pior resultado térmico.

Na tabela 5, é possível ver pelo valor *F* de 21 que o *hyperthreading* possui um impacto tangível nos ciclos por instrução, provavelmente devido ao comportamento dos ciclos por instrução ser

	sum_sq	F	p-value
C(THREADS)	48.361651	18.586491	0.000000
C(HT_OFF)	7.106524	21.849600	0.000005
C(BINDING)	0.033739	0.103734	0.747674
C(HT_OFF):C(BINDING)	0.294290	0.904820	0.342450
Residual	78.059354	NaN	-

Table 5. ANOVA para CPI_RATE

fortemente dependente da quantidade de *threads* e da disputa por recursos dentro do núcleo, caso o *Hyperthreading* esteja ativado. Além disso, não há interação visível entre o *hyperthreading* e a política de vinculação pra essa métrica.

	sum_sq	F	p-value
C(THREADS)	5300.901527	2.754524	0.006290
C(HT_OFF)	3510.528229	14.593494	0.000170
C(BINDING)	2063.090006	8.576399	0.003733
C(HT_OFF):C(BINDING)	672.672057	2.796342	0.095783
Residual	57733.038537	NaN	-

Table 6. ANOVA para MEMORY_BOUND

Na tabela 6 vemos que ambos *hyperthreading* e a política de vinculação impactam a métrica de *Memory Bound*, sendo isso especialmente evidente pelo valor *F* do *hyperthreading*. Isso é condizente com nossas figuras 3, 4, 5, e 6, onde o estado do *Hyperthreading* causou uma diminuição grande no *speedup* da aplicação nesses casos.

	sum_sq	F	p-value
C(THREADS)	191.385561	17.909454	0.000000
C(HT_OFF)	0.508229	0.380472	0.537936
C(BINDING)	1.135873	0.850341	0.357382
C(HT_OFF):C(BINDING)	0.734943	0.550195	0.458963
Residual	320.588597	NaN	-

Table 7. ANOVA Speedup

Por fim, na tabela 7 podemos observar que tanto o *hyperthreading* quanto a política de vinculação não possuem impacto significativo no *speedup* da aplicação num contexto global. E além disso, elas não possuem interação estatísticamente relevante entre si nesse contexto.

Embora essa análise de variância global não tenha demonstrado nenhuma significância estatística do estado do *hyperthreading* e da política de vinculação de *threads* no *speedup*, os resultados anteriores demonstram que há uma grande mudança no comportamento da máquina, e os resultados de cada grafo de entrada dependem da sensibilidade da entrada ao comportamento. Uma análise crítica com mais métricas seria necessária para gerar um modelo com maior certeza sobre o que afeta o desempenho da aplicação.

5 Conclusão

5.1 Conclusões

Este trabalho apresentou uma análise detalhada do desempenho do algoritmo PageRank em arquiteturas modernas, explorando o impacto de configurações de *multithreading*, afinidade de processador e microarquitetura no tempo de execução. Através de uma metodologia experimental combinando métricas de tempo real com instrumentação via *Intel VTune Profiler* e validação estatística via ANOVA, conseguimos identificar alguns gargalos que limitam a escalabilidade de algoritmos baseados em grafos.

Nossa análise demonstrou que para algoritmos limitados por memória como o PageRank a estratégia padrão de usar o máximo de *threads* lógicas disponíveis nem sempre resulta no melhor desempenho. Observamos que configurações utilizando apenas núcleos físicos com *Hyper-Threading* desligado (**HT_OFF**) frequentemente superaram o uso de todas as *threads* lógicas.

E com isso, nós chegamos a três conclusões principais:

- Observamos um aumento na métrica do *CPI Rate* e *Memory Bound* ao ligarmos o *hyperthreading*. Isso indica que o *hyperthreading* aumentou a disputa por recursos de memória e *caches* L2/L3, sem oferecer ganho de vazão suficiente para compensar a latência adicional. Na maioria dos grafos, executar com o *hyperthreading* desativado foi a melhor configuração considerando o *speedup*.
- A política de vinculação *close* mostrou-se superior em diversos cenários, especialmente quando o número de *threads* se limitava a um único soquete físico (menor que 44). Isso sugere que a comunicação entre processadores diferentes na mesma placa-mãe pode causar uma latência significativa na execução. No entanto, em grafos com estruturas de alta densidade (como o *com-Orkut*), a política *spread* pode ser necessária para evitar limitações térmicas do processador.
- E por fim, a análise estatística juntamente com os gráficos de *speedup* nos demonstraram que não existe uma única configuração ideal independente da entrada. A eficiência da configuração nesse algoritmo depende altamente da sensibilidade da entrada ao estado atual do sistema.

5.2 Aprendizados e Dificuldades

Nesse trabalho, desenvolvemos as seguintes habilidades:

- Aprendemos a fazer análises estatísticas criteriosas.
- Aprendemos a organizar nossos dados, e a inseri-los em uma plataforma reprodutível
- Aprendemos a fazer pesquisa de maneira séria, considerando variações estatísticas e construindo modelos que minimizem o impacto dessas variações.

Além disso, essas foram nossas principais dificuldades:

- Organizar os dados gerados pelo *Intel Vtune Profiler*. Ele gera muitos dados, e a maioria não era de interesse do nosso grupo.
- A nossa máquina nem sempre estava disponível para ser alocada.
- Desligar o *hyperthreading* na máquina foi difícil, pois é necessário fazer manualmente, não havendo nenhum programa ou comando pronto que possa fazer isso.
- Não estávamos acostumados a fazer pesquisa dessa maneira, utilizar *notebooks* é algo novo pro grupo.

References

[1] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC] https://arxiv.org/abs/1508.03619

[2] Instituto de Informática. [n. d.]. Laboratório de Processamento Paralelo e Distribuído (LPPD). https://gppd-hpc.inf.ufrgs.br/ Acesso em: 23-11-2025.
[3] Intel. 2025. Intel® VTune™ Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. Acesso em: 23-11-2025.
[4] Jure Leskovec. 2025. Stanford Large Network Dataset Collection. https://snap.stanford.edu/papers.html. Acesso em: 23-11-2025.
[5] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.
[6] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford infolab.
[7] Wikipedia. 2025. OpenMP — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=OpenMP&oldid=1322630707. Acesso em: 23-11-2025.
[8] Wikipedia. 2025. Power iteration — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Power%20iteration&oldid=1322057966. Acesso em: 23-11-2025.