Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Programação Paralela — Projeto Final

# $\triangle$-Stepping Paralelo com Bucket Fusion

Alexander Thomas Mol Holmquist

1 de setembro de 2021

# 1 Introdução

O algoritmo de delta-stepping para o problema SSSP (single-source shortest path) foi sugerido por Ulrich Meyer e Prashantan Sanders em 1998 [1, 2]. Desde então, vem sendo utilizado na prática e em várias áreas de pesquisa. Além de servir para o SSSP, o princípio do delta-stepping (*priority coarsening*) pode ser aplicado em outros algoritmos para alcançar melhor paralelização.

Isso é possível porque este princípio é verdade para muitos algoritmos que não precisam seguir estritamente uma ordem para garantir *progresso*. Tal ordem pode ser relaxada sem prejudicar a corretude do procedimento. Por exemplo, o algoritmo de Dijkstra [3] requer que os vértices da fronteira de busca sejam tratados de acordo com a distância ao vértice de origem, em ordem crescente de distância. Porém, esta restrição não é necessária.

Delta-stepping é largamente baseado no algoritmo de Dijkstra. Talvez a única diferença considerável, que na verdade tem um impacto grande na prática, é o fato da ordem mencionada acima não ser respeitada. Vértices são alocados para "baldes" de acordo com a fórmula: $buckIdx = tentative\_distance/delta$. Como $tentative\_distance$ é um parâmetro altamente variável durante a execução do algoritmo, o "balde" a que o vértice pertence também muda de acordo.

No delta-stepping, trabalhamos sempre com o primeiro balde (balde que guarda os vértices mais próximos do vértice de origem). Assim, uma noção de ordem que é preservada de duas maneiras. Primeiro, a distância dos vértices ao vértice de origem só pode diminuir, da mesma forma que com Dijkstra. Em segundo lugar, uma vez que um vértice é inserido no primeiro balde, se em uma iteração posterior não houver reinserção, este vértice nunca mais será considerado pelo algoritmo.

Um esboço do algoritmo, em uma versão um pouco modificada da original, é dado abaixo. O algoritmo original contém complicações que julgamos desnecessárias, por isso não são incluídas. É assumido que os baldes foram previamente alocados na variável $bucks$, em um passo de pre-processamento. A função $getMinBuck$ percorre sequencialmente $bucks$ e retorna o primeiro balde não vazio, ou o valor especial $nullptr$ se não achar um.

**Algorithm 1:** DELTASTEPPING finds distances from source node

**Input:** Directed graph $G$ with nonnegative edges, $sourceNode$ and $delta$
**Output:** $dists = \{d_0, d_1, ..., d_n\}$

```
1 dists ← {∞, ∞, ..., ∞}
2 relax(sourceNode, 0, delta)
3 while minBuck = getMinBuck(bucks) do
4     for node n in minBuck do
5         remove(minBuck, n)
6         for e in outEdges(n) do
7             newDist ← dists[e.destNode] + e.weight
8             relax(e.destNode, newDist, delta)
```

**Algorithm 2:** RELAX is used to update the distance to $destNode$

**Input:** $destNode$, $newDist$, $delta$

1 **if** $newDist < dists[destNode]$ **then**

2 $\quad dists[destNode] = newDist$

3 $\quad appendTo(bucks[newDist \ / \ delta],\ destNode)$

O objetivo do presente trabalho é implementar uma versão paralela do algoritmo dado acima, mais a otimização *Bucket Fusion* como apresentada em [4]. A Seção 2 descreve o desenvolvimento do trabalho. A Seção 3 busca descrever os experimentos realizados, e contém uma breve discussão. A Seção 4 mostra as instruções para compilar o código localmente. O repositório com controle de versão do projeto pode ser encontrado em [5].

# 2 Jornada de Paralelização

O processo de paralelizar este algoritmo pode ser descrito como uma jornada. Várias tentativas de otimização falharam, mas o resultado final é satisfatório. Nesta seção, cobrimos os aspectos principais de tal jornada. A linguagem objetivo é C++, conforme definido em seu padrão de 2014 [6]. Utilizamos a interface OpenMP para criação de manipulação do ambiente paralelizado. Detalhes sobre a máquina sobre a qual os programas foram executados podem ser encontrados no Apêndice A.

Começamos nossa implementação de delta-stepping com uma versão que seguia estritamente a versão sequencial do algoritmo apresentado em [1], escrita em Python. Partiu-se daí para a implementação equivalente na linguagem C++, que requer muito maior atenção a detalhes. Não houveram grandes empecilhos, exceto que na implementação dos baldes, foi necessário inventar o meio de reaproveitar que os autores mencionam em sua publicação, pois não é especificado (a implementação final ignora esse reaproveitamento, e simplesmente mantém um grande vetor com todos os baldes necessários—$maxDist/delta$ baldes).

Chegado o momento de implementar a versão paralela, surgiram muitas dúvidas que não são respondidas em nenhum dos artigos que estávamos utilizando como fonte. Por exemplo, o artigo original sugere que cada balde seja tratado paralelamente. Ao tentar colocar esta ideia em prática, todavia, se tornou muito difícil sincronizar a modificação e utilização de baldes por diferentes threads. O problema enraíza do fato de qualquer thread $n$ tem o potencial de interferir em estruturas de dados utilizadas por todos os threads $0..n$.

Desistimos de atribuir um thread para cada balde. Como resultado de um pouco de pesquisa (especialmente [7]), passamos a pensar em criar porções de memória privadas para cada thread, para que possam trabalhar individualmente. Novamente, surge a pergunta: será que permitimos que cada thread manipule um balde diferente? A resposta é não: complicado demais. Apesar de inicialmente parecer uma má ideia, cada iteração do algoritmo final distribui somente os vértices do balde mínimo global entre os threads.

Este método apresenta um sério perigo, em termos de performance. Se alocarmos números de vértices diferentes para cada thread, corremos o risco de perder no quesito *load balancing*. Além disso, não sabemos quantas arestas de saída cada vértice tem. É razoavelmente provável que, se distribuirmos os vértices uniformemente, a qualidade do algoritmo também se deteriore. Dito isto, decidiu-se utilizar a cláusula $schedule(dynamic, 64)$ para a diretiva OpenMP do laço principal do algoritmo. O número *64* é baseado em [7], apesar de não ter se mostrado tão bom nos experimentos que executamos.

Como era o objetivo inicial de exploração do trabalho, implementamos ainda a técnica

*Bucket Fusion*. Uma descoberta interessante deste trabalho é que ela não provê significativa melhoria da performance do código, para problemas pequenos (veja Seção 3).

Outras tentativas interessantes de otimização podem ser encontradas no Apêndice B. Abaixo apresentamos uma descrição em alto nível do algoritmo paralelizado, em sua forma final. Algumas complicações para evitar corrida de dados foram omitidas. Micro-otimizações, como regiões marcadas com `#pragma omp single`, são também omitidas. O importante a notar é que o algoritmo requer sincronização em dois pontos distintos, e essas regiões são dividas pelas tarefas que realizam.

**Algorithm 3:** Parallel DELTASTEPPING with bucket fusion

**Input:** Directed graph G with nonegative edges, $sourceNode$ and $delta$
**Output:** $dists = \{d_0, d_1, ..., d_n\}$

```
1  for t in threads do
2      local lBucks
3      while gMinBuck = getGlobalMinBuck(bucks) do
4          for i = gMinBuckStartIdx to size(gMinBuck) do
5              relaxEdgesPrl(gMinBuck[i], lBucks)
6          bucketFusion(lBucks)
7          copyToGBuck(bucks, lBucks)
8          clear(lBucks)
9          OMP barrier
10         updateIdxs(bucks)
11         OMP barrier
```

**Algorithm 4:** BUCKETFUSION

**Input:** Local buckets $lBucks$

```
1  if lBucks is not empty then
2      lbIdx = getMinBuckIdx(lBucks)
3      while lBucks[lbIdx] is not empty and size(lBucks[lbIdx]) < threshold do
4          lBuckCopy = copy(lBucks[lbIdx])
5          clear(lBucks[lbIdx])
6          for srcNode in lBuckCopy[lbIdx] do
7              relaxEdges(srcNode, lBucks)
```

# 3 Avaliação

Os experimentos realizados neste projeto fizeram uso dos bancos de dados a seguir. Cada problema é executado 10 vezes pelo algoritmo de Dijkstra, delta-stepping sem bucket fusion, e delta-stepping com bucket fusion. Calculamos a média do tempo de execução dos 10 despachos.

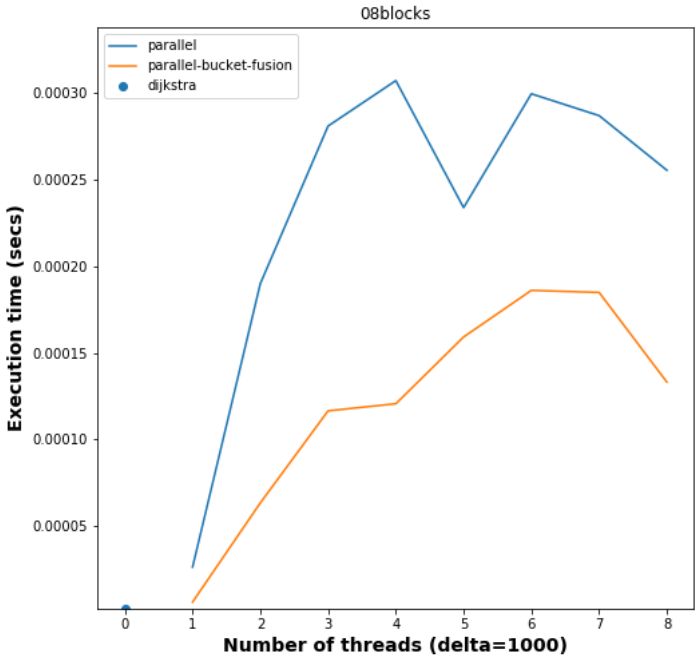
- 08blocks [8]—300 vértices, 592 arestas.

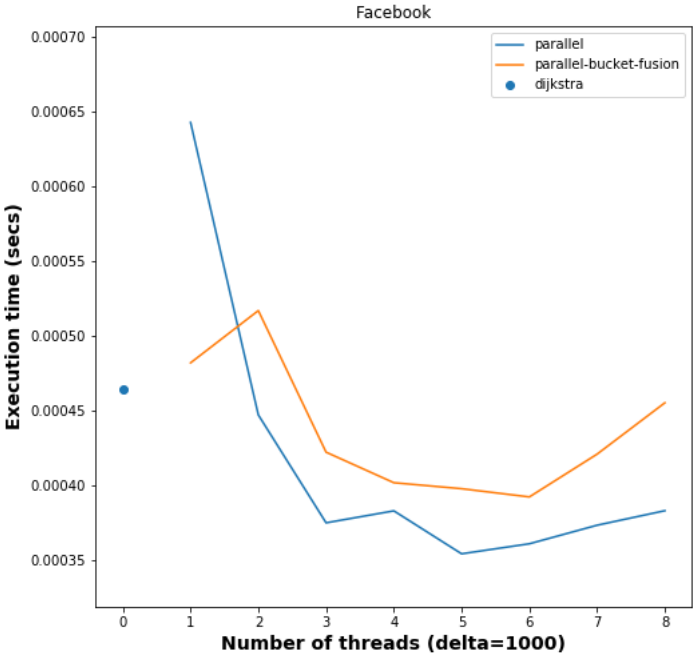
- Facebook-like social network [9, 10]—1899 vértices, 20296 arestas.
- Political blogosphere [11]—82144 vértices, 549202 arestas.

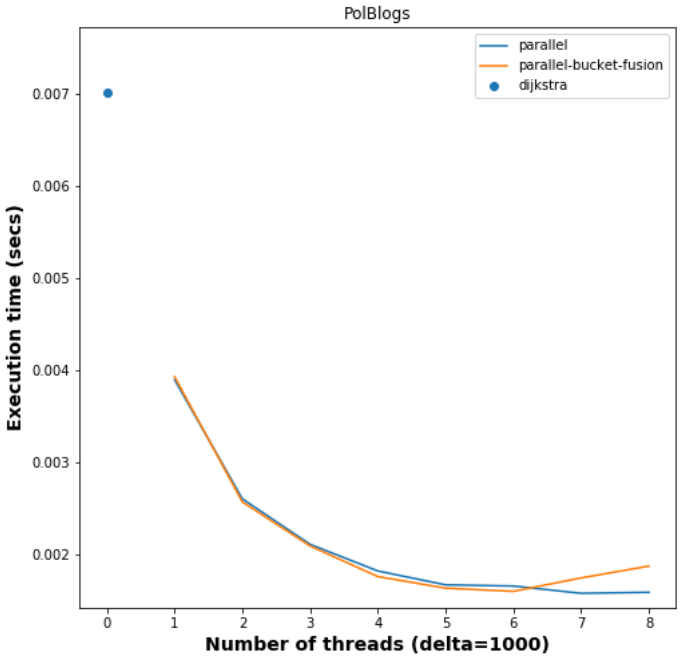
Não conseguimos expandir o algoritmo para tamanhos muito maiores (a partir de 1000000 vértices), pois ele passa a entrar em um estado de deadlock cuja causa não detectamos ainda. Nas seções a seguir, iremos avaliar o comportamento do tempo de execução desses problemas ao variar o número de threads (com delta fixo), ou o delta (com o número de threads fixo).

## 3.1 Comportamento ao variar número de threads

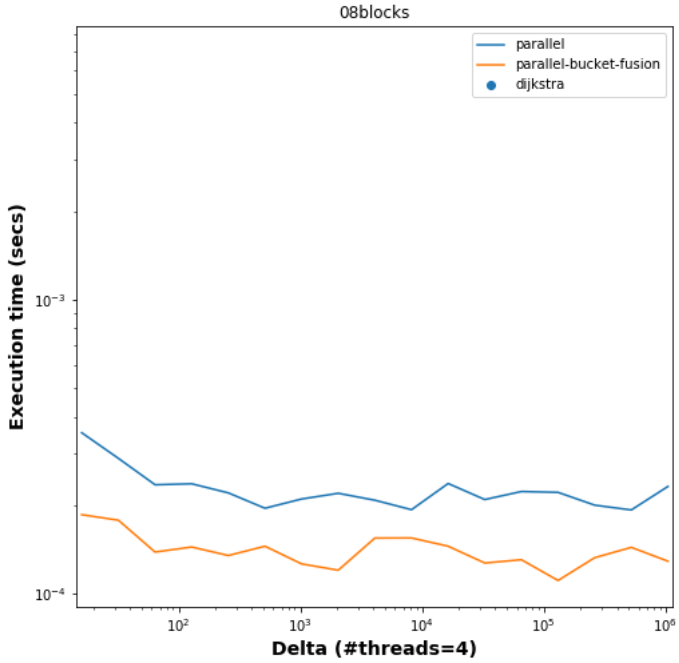
As figuras as seguir ilustram este comportamento. As figuras estão ordenadas crescentemente com o tamanho do problema.

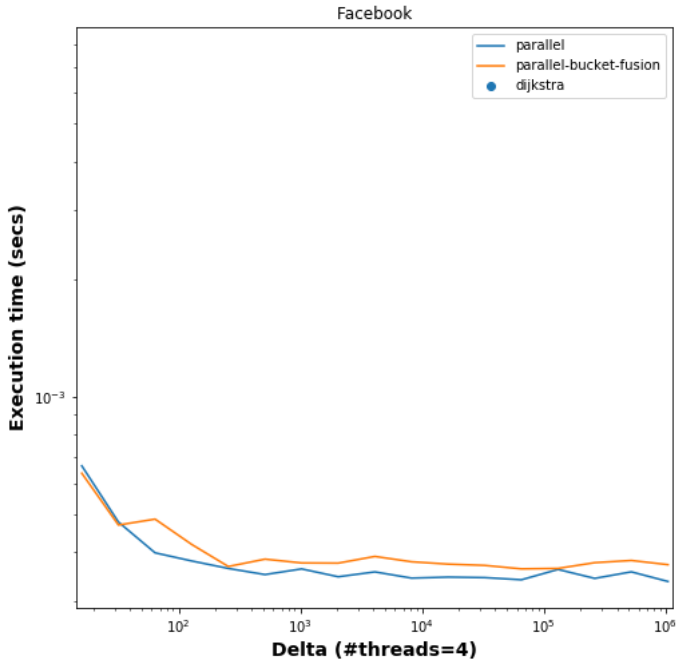


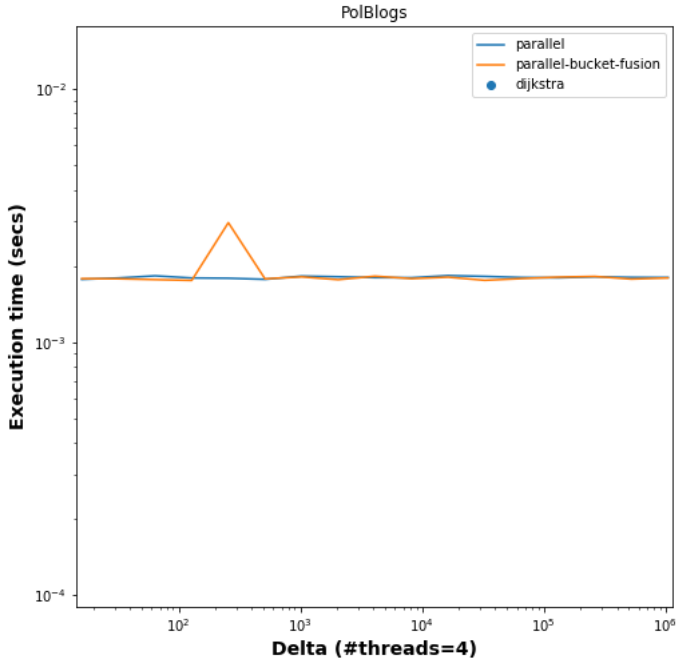

Facebook
parallel
parallel-bucket-fusion
dijkstra
Execution time (secs)
0.00070
0.00065
0.00060
0.00055
0.00050
0.00045
0.00040
0.00035
0 1 2 3 4 5 6 7 8
Number of threads (delta=1000)


PolBlogs
parallel
parallel-bucket-fusion
dijkstra
Execution time (secs)
0.007
0.006
0.005
0.004
0.003
0.002
0 1 2 3 4 5 6 7 8
Number of threads (delta=1000)

## 3.2 Comportamento ao variar delta






### 3.3 Discussão e Trabalho Futuro

Os gráficos mostram um *speedup* de cerca de 100% quando passando de 1 a 4 threads. Isso equivale a uma eficiência de cerca de 50%. Tal eficiência é um passo considerável na direção certa, mas não é muito alta, em termos gerais. Esse fenômeno pode ser justificado quando se considera o tamanho pequeno dos problemas dados como entrada do algoritmo.

Pretendemos no futuro próximo estender o arcabouço de testes realizados sobre o algoritmo, e executá-lo em servidores remotos disponibilizados da Google Cloud [12]. Este esforço tem o simples objetivo de confirmar empiricamente o que foi dito em [4]: que *bucket fusion* funciona melhor em redes de tráfego com diâmetro largo.

## 4 Instruções de Compilação

### 4.1 Requerimentos

O código foi testado com o compilador GCC, versão 9.3.0. É necessário que o compilador tenha uma implementação da API OpenMP. A versão da API testada foi a de novembro de 2015. A API é suportada por distintos sistemas operacionais com semântica intacta. Veja o Apêndice A para mais detalhes do ambiente utilizado nos experimentos.

### 4.2 Comando para compilar

Para compilar o código, basta digitar `make` enquanto no diretório principal do projeto. Deve ser gerado o executável `build/delta-stepping`, que pode ser utilizado seguindo o padrão:

```
<program> <in-file> <mode> <delta-step> [<num-threads>]
```

### 4.3 Configurações

É possível ajustar diversos parâmetros nos arquivos `lib/make/global-var/global-var-general.mk` e `lib/main/header/Utils/defs.hpp`. Por exemplo, no segundo arquivo pode-se configurar o programa para imprimir as distâncias computadas. Basta atribuir o valor 1 a `INTERFACE_INIT_PRINT_DISTS`.

# A Detalhes da máquina local

- **Sistema operacional:** Ubuntu 20.04.3 LTS.
- **CPU:** Intel(R) Core(TM) i7-1065G7 @ 1.30GHz-3.90GHz. 1 processador físico, 4 núcleos, 8 threads virtuais.
- **RAM:** 16GB tecnologia DDR4 a 3200MHz.

# B Outras tentativas de otimização

No processo de otimização do algoritmo, nos pareceu que a fase de copiar os baldes locais para os globais *copyToGBuck* era a que gastava mais tempo de execução. O que levantou tal suspeita é o fato de que essa região do código consistia em um laço sobre uma região crítica, o que é um pesadelo em termos de overhead de sincronização. Contudo, a cópia de elementos para os baldes globais não pode ser feita de maneira completamente liberal. Por exemplo, utilizando

$$bucks.insert(bucks.end(), lBucks.begin(), lBucks.end())$$

é possível que um thread modifique o iterador interno de `bucks`, deixando-o em um estado inválido para os demais threads. É preciso ter cuidado. Contudo, mesmo respeitando regras de segurança entre threads, é possível melhorar muito a eficiência de tal código. Nossa primeira tentativa de otimização foi retirar a região crítica, e inserir "cadeados" no código. Cada balde global tem seu próprio cadeado. Quando um thread quer copiar seu balde para um balde $i$ global, ele tranca o cadeado $i$, impossibilitando o acesso dos demais threads ao mesmo balde global. Note, porém, que se trata cada balde global separadamente, o que entra em contraste com o método anterior, que só permitia um thread por vez acessar qualquer balde, em um dado momento.

Tal otimização, apesar de promissora, não alterou o tempo de execução médio que observávamos nos experimentos. Isso nos levou a uma tentativa desesperada: randomizar a cópia dos baldes locais para os globais. Cada thread copiaria seus baldes em uma ordem aleatória. Pensávamos que com isso disputas por cadeados se tornassem menos frequentes. Na verdade, essa mudança aumentou significativamente o tempo de execução (quase dobrou, mesmo para grafos pequenos), e rapidamente voltamos atrás.

Por fim, decidimos inserir medidas de tempo de execução de menor granularidade no código (algo que deveríamos ter feito desde o início desta exploração), dando a cada fase uma medida separada. Tais medidas revelaram que a maior parte do tempo do algoritmo estava sendo gasta na fase de relaxação das arestas, e não de cópia dos baldes locais. A fase de cópia consistia em cerca de *1%* do tempo total gasto, para os experimentos. Desta forma, paramos de tentar otimizar a fase *copyToGBucks*.

# Referências


[1] P. Sanders U. Meyer. Delta-stepping: A parallel single source shortest path algorithm. 1998.

[2] U. Meyer and P. Sanders. Delta-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003. 1998 European Symposium on Algorithms.

[3] EW Dijkstra. A note on two problems on connexion with graphs. *Numer. Math*, 1:269–271, 1959.

[4] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 158–170, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Git repository. https://github.com/Yowgf/delta-stepping. Acessado em 01 de setembro de 2021.

[6] C++14. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf. Acessado em 31 de agosto de 2021.

[7] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[8] 08blocks graph. https://networkrepository.com/08blocks.php. Acessado em 31 de agosto de 2021.

[9] Weighted static one-mode facebook-like social network (weighted by number of characters). https://toreopsahl.com/datasets/. Acessado em 31 de agosto de 2021.

[10] Tore Opsahl and Pietro Panzarasa. Clustering in weighted networks. *Social Networks*, 31:155–163, 05 2009.

[11] Polblogs weighted graph. https://networkrepository.com/polblogs.php. Acessado em 31 de agosto de 2021.

[12] Google cloud. https://cloud.google.com/gcp. Acessado em 13 de junho de 2021.