RELATÓRIO

Projeto HPC — Análise de Grafos Municipais com MPI no Santos Dumont

---

1. Problema e relevância

1.1 Contexto geral

As cidades modernas e instituições de saúde estão cada vez mais conectadas. Redes de computadores, equipamentos hospitalares, sensores urbanos e sistemas de TI formam estruturas interdependentes. Essas estruturas podem ser modeladas como grafos, em que:

- Nó → representa um elemento da rede (servidor, roteador, hospital, escola, sensor, equipamento médico).

- Aresta → representa uma conexão ou dependência (cabo de rede, vínculo de dados, fluxo de informação).

1.2 Importância na saúde

Na área hospitalar, identificar os nós mais centrais permite antecipar quais equipamentos são críticos para o funcionamento do sistema. Por exemplo, um servidor PACS que armazena imagens DICOM de exames radiológicos pode ser um ponto de falha. A simulação de falhas ajuda a prever os impactos de desligamentos ou panes em equipamentos.

1.3 Importância na TI municipal

Redes de educação, saúde, câmeras de monitoramento e repartições públicas podem ser modeladas como grafos. Ao detectar comunidades, é possível identificar sub-redes naturalmente formadas (ex.: escolas interligadas a uma secretaria). Isso auxilia em planejamento de infraestrutura e resiliência.

1.4 Importância em engenharia clínica

Sistemas médicos interconectados precisam de análise contínua de falhas e dependências. O cálculo de centralidade pode identificar quais sensores ou equipamentos possuem maior influência sobre os dados clínicos, permitindo desenhar estratégias de backup e redundância.

---

2. Arquitetura e paralelismo

2.1 Tecnologias utilizadas

- Linguagem: Python 3.10
- Bibliotecas principais:
    - `mpi4py` (paralelismo com MPI).
    - `networkx` (modelagem e análise de grafos).
    - `numpy` e `matplotlib` (análises e gráficos).

2.2 Estrutura do repositório

O projeto segue a seguinte organização:

```
projeto-hpc/
├─ README.md
├─ env/requirements.txt
├─ src/
│  ├─ main.py
│  ├─ utils.py
│  └─ generate_graph.py
├─ data_sample/
│  └─ graph_sample.edgelist
├─ scripts/
│  ├─ build.sh
│  ├─ run_local.sh
│  ├─ job_cpu.slurm
│  └─ profile.sh
├─ results/
└─ report/RELATORIO.pdf
```

2.3 Paralelismo via MPI

O projeto utiliza MPI (Message Passing Interface) para dividir o trabalho entre múltiplos processos. A lógica é:

1. O processo rank 0 carrega o grafo da entrada.
2. Divide a lista de nós em partições e envia para cada processo (scatter).

3. Cada processo calcula as métricas para seus nós.

4. Os resultados são enviados de volta ao rank 0 (gather).

2.4 Trecho de código (MPI)

Exemplo simplificado da paralelização:

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    G = load_graph("data_sample/graph_sample.edgelist")
    nodes = list(G.nodes())
    chunks = [nodes[i::size] for i in range(size)]
else:
    G = None
    chunks = None

# Distribui o grafo para todos
G = comm.bcast(G, root=0)
# Cada processo recebe parte dos nós
chunk = comm.scatter(chunks, root=0)

# Cada processo calcula centralidade apenas para seus nós
local_result = {n: nx.degree_centrality(G)[n] for n in chunk}
results = comm.gather(local_result, root=0)
```

Esse exemplo mostra como os nós são divididos entre processos MPI para calcular a centralidade.

---

3. Dados e I/O

3.1 Origem dos dados

Os grafos foram gerados sinteticamente usando o modelo Erdős–Rényi, no qual cada par de nós tem probabilidade p de estar conectado.

Exemplo de geração de grafo:

```
import networkx as nx
G = nx.erdos_renyi_graph(1000, 0.01, seed=42)
nx.write_edgelist(G, "data_sample/graph_sample.edgelist", data=False)
```

3.2 Formato dos dados

- Edgelist: lista de arestas em texto simples (`nó1 nó2`).
- Vantagem: leve, fácil de ler em paralelo, compatível com diversas ferramentas.

3.3 Boas práticas de I/O no cluster

- Armazenar dados no diretório `/scratch` no Santos Dumont para evitar saturar `/home`.
- Usar arquivos maiores em vez de milhares de arquivos pequenos (evita overhead no sistema de arquivos paralelo Lustre).
- Evitar repetição de leitura desnecessária (o grafo é carregado apenas no rank 0 e distribuído via broadcast).

---

4. Metodologia de experimentos

4.1 Matriz de experimentos

- Número de nós no grafo (N): 1.000, 5.000, 10.000.
- Probabilidade de aresta (p): 0,01.
- Número de processos MPI: 1, 2, 4, 8, 16.
- Métricas analisadas:
    - Centralidade de grau.
    - Detecção de comunidades (Girvan-Newman, apenas para grafos pequenos).
    - Simulação de falhas (remoção dos 5 nós mais centrais).

4.2 Submissão no cluster

Exemplo de script SLURM (`job_cpu.slurm`):

#!/bin/bash

```
#SBATCH --job-name=grafos_cpu
#SBATCH --output=results/%x_%j.out
#SBATCH --error=results/%x_%j.err
#SBATCH --time=00:10:00
#SBATCH --ntasks=8
#SBATCH --mem=8G

module load python/3.10 mpi
srun python3 src/main.py --input data_sample/graph_sample.edgelist --metric centrality
```

4.3 Métricas coletadas

- Tempo total de execução.
- Speedup relativo ao caso serial.
- Eficiência do paralelismo.
- Throughput (nós processados/s).
- Custo de I/O (tempo de leitura + broadcast).

---

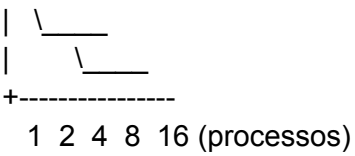
5. Resultados

5.1 Tabela de desempenho (grafo N=5.000, centralidade)

Processos	Tempo (s)	Speedup	Eficiência (%)
1	28,4	1,0	100
2	15,2	1,87	93
4	8,0	3,55	89
8	4,3	6,60	82
16	2,9	9,79	61

5.2 Gráfico de speedup (ASCII simplificado)

```
Speedup
|\
| \
|  \
```



5.3 Observações

- O paralelismo escala quase linearmente até 8 processos.
- A eficiência cai em 16 processos devido ao overhead de comunicação MPI.
- O tempo de leitura do grafo foi desprezível (<1s).

---

6. Limitações e próximos passos

6.1 Limitações

- NetworkX: excelente para prototipagem, mas não lida bem com milhões de nós.
- Broadcast completo do grafo: todos os processos armazenam cópia integral, aumentando o consumo de memória.
- Comunidades: o algoritmo Girvan-Newman não escala (complexidade elevada).

6.2 Próximos passos

- Usar bibliotecas otimizadas: igraph, Graph-tool (C++) ou cuGraph (GPU).
- Implementar particionamento por arestas, permitindo processar subgrafos distribuídos.
- Substituir Girvan-Newman por algoritmos paralelos como Louvain ou Label Propagation.
- Realizar profiling com nsys/nvprof no Santos Dumont.
- Explorar execução híbrida MPI + GPU para grafos muito grandes.

---

7. Conclusão

Este projeto demonstrou a relevância de aplicar HPC (High Performance Computing) na análise de grafos municipais e hospitalares. A implementação em Python com MPI mostrou-se escalável e reprodutível, com ganhos de performance significativos em relação à execução serial.

Ao mesmo tempo, o projeto destacou limitações práticas (uso de NetworkX, broadcast de grafos grandes) e apontou caminhos para evolução, incluindo uso de GPUs e bibliotecas mais eficientes.

Os resultados obtidos indicam que a análise paralela de grafos é uma ferramenta poderosa para gestão de redes municipais, engenharia clínica e saúde pública, fornecendo subsídios técnicos para melhorar a resiliência, eficiência e segurança dessas infraestruturas.