**BCC 447 - Programação Paralela**

**Descrição geral**

O objetivo desta disciplina é formar recursos humanos com capacidade de desenvolver aplicações paralelas escaláveis segundo os modelos de programação mais comuns da literatura, sejam eles o modelo de programação centrado nos dados (Data Intensive, Data Partitioning, etc.) ou o direcionado às tarefas (Pipelined, Task Intensive, Work Pool, etc.). Os alunos saberão mapear tarefas, decompor problemas, projetar/realizar experimentos e discutir resultados experimentais. Além disso, os interessados poderão ter contato com as tecnologias mais emergentes do mercado, tais como Docker, Kubernetes e ferramentas para mensageria, como o Kafka, RabbitMQ, Redis ou o Cloud Pub/Sub.

A disciplina terá conteúdo prático/teórico e tecnológico. O curso BCC 447 conta com a parceria do aluno Gabriel Ribeiro ([link](https://www.linkedin.com/in/riibeirogabriel/)), voluntário na disciplina para ensinar aspectos tecnológicos desta. Com encontros remotos com Gabriel, os alunos poderão montar cluster de máquinas virtuais e nestas um conjunto de contêineres.

**Estrutura**

São 3 trabalhos práticos

**1o trabalho ->** resolução de 4 problemas de baixa complexidade em qualquer linguagem de programação com suporte a threads. (Entrega no final do 1o mês de curso - Equipe de até 3 alunos)

**2o trabalho ->** resolução de 4 problemas de média a alta complexidade em qualquer linguagem de programação com suporte a threads. (Entrega no final do 2o mês de curso - Equipe de até 3 alunos)

**3o trabalho ->** tentativa de resolução de 2 problemas de alta complexidade e ainda em aberto na literatura. (Entrega no final do 4o mês de curso - Equipe de até 3 alunos)

Horários de aula prática/teórica:

Segunda-feira (10h10 - 11h50)

Quarta-feira (10h10 - 11h50)

Horários dos encontros da parte tecnológica do curso:

Normalmente aos sábados pela manhã

**Lista de problemas**

1o conjunto

1. Somar n números
2. Multiplicar matrizes
3. Contagem de frequência de termos em bases textuais
4. Ordenação de itens

2o conjunto

1. Problema dos N-corpos - (<https://en.wikipedia.org/wiki/N-body_problem>)
2. Menor caminho (shortest path) - (<https://en.wikipedia.org/wiki/Shortest_path_problem>)
3. Branch and Bound (<https://en.wikipedia.org/wiki/Branch_and_bound>)
4. Approximate string matching (<https://en.wikipedia.org/wiki/Approximate_string_matching>) e um survey (<https://users.dcc.uchile.cl/~gnavarro/ps/deb01.pdf>) e versões paralelas (<https://scholar.google.com.br/scholar?q=approximate+string+matching+parallel&hl=pt-BR&as_sdt=0&as_vis=1&oi=scholart>)

3o conjunto

1. Frequent Itemset ([Frequent itemset mining: A 25 years review](https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1329?casa_token=Wvhiqh8fL0gAAAAA:usmSbvCgopliZzmZDMgOTzGE6umiDby-7CDFNY03No6Kdi1K8Z9pNnHwU9dUBryIreL9ubjGO48) - [link](https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1329?casa_token=OTOfyy25q_YAAAAA:-Tv47P12bjzoXqE7ROpIQgc62ZH2nfl9mTWTQ7APSCHQ8dSIE8dBd7w5TufZqqMK16Rs1pr-gDw))
2. Caixeiro viajante (<https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante>) modelado como backtracking (<https://pt.wikipedia.org/wiki/Backtracking>)

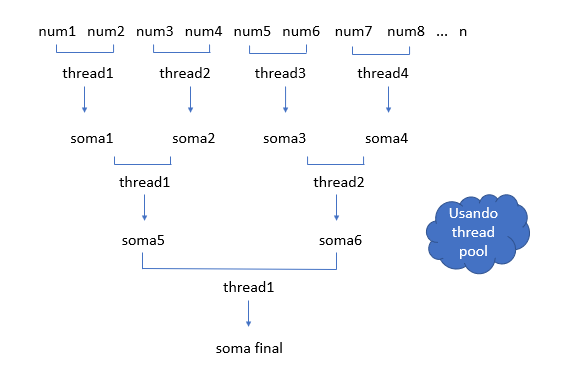
**Detalhes dos problemas**

**1o Conjunto**

**Problema 1 - Somar n números:**

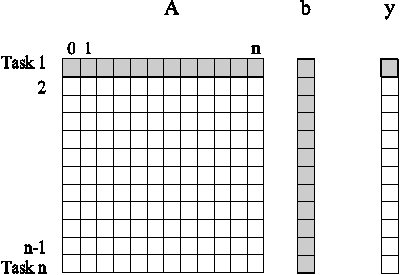
1. Entrada - 1 conjunto de n números (inteiros ou ponto flutuante);
2. Saída - a soma destes n itens;
3. Detalhar a alternativa de solução escolhida (modelo de programação utilizado - work pool, pipeline, data partition, master slave, híbrido) e suas razões;
4. Apresentar os experimentos e resultados experimentais, incluindo discussões;
5. Parametrize sua solução para aceitar diferentes entradas;

O algoritmo sequencial possui complexidade O(n) e resolve o problema somando cada item ao resultado já existente de forma serial. Entretanto, há como alocar n/2 threads ou processos ou máquinas e cada um destes recursos somar apenas dois números, retornando n/2 resultados parciais que podem ser executados em paralelo por n/4 threads ou processos ou máquinas e assim sucessivamente até que uma thread ou processo ou máquina efetue a soma final. Este conjunto de reduções paralelas possui complexidade O(lgn). Note que esta ideia é o cerne do que existe em ferramentas como Mapreduce, Spark e muitas outras. Em linguagens de programação, tal como o Java, também há o operador reduce, incluindo sua execução em paralelo ([link](https://docs.oracle.com/javase/tutorial/collections/streams/index.html)). Diversas outras operações, tais como união de conjuntos, ordenação de itens, detecção de maior/menor em conjuntos não ordenados e muito mais, podem ser resolvidas com a mesma modelagem.

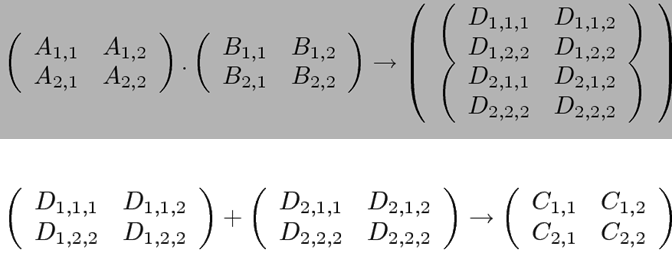


**Problema 2 - Multiplicar 2 matrizes ou 1 matriz por um vetor:**

1. Entrada- Matriz 1; Matriz 2; ([links](https://www.dropbox.com/s/4u6hm2h3yadscm6/matrizes.rar?dl=0))
2. Saída - Matriz resultante;
3. Detalhar a alternativa de solução escolhida (modelo de programação utilizado - work pool, pipeline, data partition, master slave, híbrido) e suas razões;
4. Apresentar os experimentos e resultados experimentais, incluindo discussões;
5. Parametrize sua solução para aceitar diferentes entradas;
6. Material extra - <https://onlinemathtools.com/generate-random-matrix>



**OU**



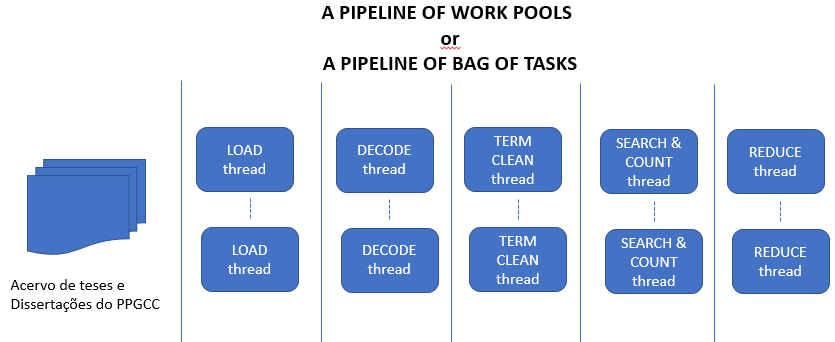
No exemplo acima há n tarefas, onde cada tarefa realiza n multiplicações sequencialmente. Após as multiplicações, há n somas sequenciais, resultando em uma célula do vetor Y. Há como ter n3 tarefas e não apenas n, onde cada tarefa realiza apenas uma multiplicação de duas células. De forma análoga, após as multiplicações há n somas a serem feitas para cada célula de Y. Até o momento, ambos os algoritmos possuem complexidade O(n), entretanto o nível de concorrência pode ser n no primeiro algoritmo ou n3 no segundo, sendo a segunda opção mais promissora para ambientes computacionais maiores, ou seja, com mais máquinas ou processadores. Por fim, há como fazer as n somas de cada célula de Y também em paralelo em múltiplas reduções, ou seja, com n/2 tarefas que realizam a soma de dois números, depois n/4 tarefas e assim sucessivamente até que se tenha uma tarefa que realiza a soma final, gerando uma célula de Y em paralelo. Este algoritmo é um dos mais eficientes para multiplicação de matrizes em paralelo e possui complexidade O(lgn).

No trabalho implemente as diferentes versões explicadas anteriormente e as teste com as mesmas entradas. Utilize como célula das matrizes ou dos vetores um conjunto de itens e não apenas um inteiro ou um número real. Pense que uma matriz representa uma porção do Cosmos ou do cérebro ou da Amazônia, ou seja, cada célula pode ter centenas ou milhões de itens internamente, portanto somar ou multiplicar duas células pode ser uma tarefa custosa computacionalmente.

**Problema 3 - Contagem da frequência de termos em bases textuais;**

1. Entrada: n teses e dissertações do PPGCC UFOP (<https://www.repositorio.ufop.br/handle/123456789/596>); conjunto de termos a serem contabilizados;
2. Saída: contagem da frequência de cada termo;
3. Detalhar a alternativa de solução escolhida (modelo de programação utilizado - work pool, pipeline, data partition, master slave, híbrido) e suas razões;
4. Apresentar os experimentos e resultados experimentais, incluindo discussões;
5. Parametrize sua solução para aceitar diferentes entradas;

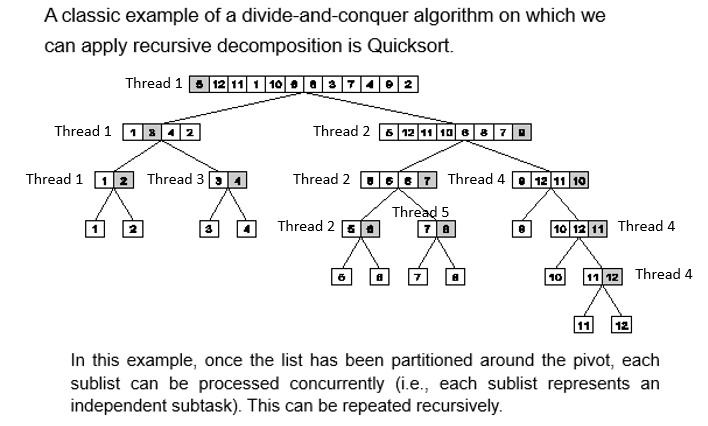
A solução pode ser modelada como um pipeline de estágios complementares, onde cada arquivo (tese ou dissertação) é carregada ou obtida da Web (LOAD), depois decodificada (DECODE - o pdf é aberto e transformando em uma matriz de termos), depois se elimina os termos desnecessários (TERM CLEAN - stop words e outros termos), depois se efetua a contagem do termo sendo pesquisado no acervo de termos restantes (SEARCH & COUNT) e, por fim, se efetua as reduções das contagens em paralelo, similar ao que foi feito para o problema 2 onde efetuamos a soma de n números em paralelo (REDUCE). Notem que em cada estágio podemos ter um conjunto de threads ou processos modelados como *thread pool* ou *bag of tasks*. Cada estágio do pipeline pode estar em uma máquina ou vários estágios podem estar compartilhados numa mesma máquina. Até mesmo um único estágio do pipeline pode estar em mais de uma máquina. Em linhas gerais, a modelagem proposta aceita inúmeros *deploys*.



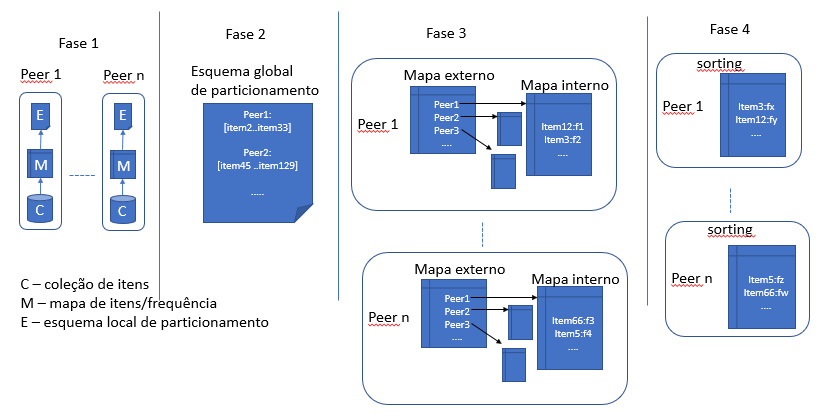
**Problema 4 - Ordenação de itens;**

1. Entrada: um conjunto não ordenado de itens;
2. Saída: o conjunto ordenado (ordem crescente ou não crescente) dos itens;
3. Detalhar a alternativa de solução escolhida (modelo de programação utilizado - work pool, pipeline, data partition, master slave, híbrido) e suas razões;
4. Apresentar os experimentos e resultados experimentais, incluindo discussões;
5. Parametrize sua solução para aceitar diferentes entradas;

Modelagem 1: Usando a recursividade podemos elencar um pivô, particionando os itens em esquerda e direita. A thread ou processo ou máquina da partição à esquerda elege novamente um pivô. O mesmo ocorre com a partição à direita. Este método aloca novas threads ou processos numa mesma máquina ou em um cluster até que se tenha apenas dois itens a serem ordenados, fazendo com que o método Quicksort escolha por manter os itens como estão ou os inverter. Uma thread ou processo nunca cria dois novos processos ou threads, pois uma partição dos itens não ordenados é sempre controlada pela thread ou processo em execução. Neste sentido, apenas uma thread ou processo é criada a cada nova sub-lista de itens com um novo pivô devidamente eleito. Concorrentemente, as sub-listas de itens podem ser processadas, conforme ilustra a figura a seguir.



Modelagem 2: Nesta modelagem o foco é a frequência dos itens, assumindo que em grandes coleções há sempre repetições. Esta abordagem é muito útil para múltiplas máquinas organizadas em cluster. Inicialmente, assumimos que cada máquina ou thread ou processo possui uma partição dos itens não ordenados. A primeira fase varre cada partição em paralelo e efetua a contagem da frequência dos itens, portanto nesta fase há a possibilidade de carregar para memória RAM grandes volumes de dados armazenados em disco. Ao final da primeira fase há um mapa com itens como chave e as frequências como valores. Cada máquina ou processo ou thread deverá saber a priori o número de outros peers que participam da execução em paralelo para que se possa efetuar o particionamento dos itens de acordo com tal número. Ainda na primeira fase se cria um esquema de particionamento dos itens local, ou seja, cada máquina ou thread ou processo avalia a frequência de seus itens e o número de peers do cluster, por exemplo. Como há a frequência total dos itens de uma partição, assim como o número de itens da mesma, há como montar inúmeras formas de esquema de particionamento (Ex. particionar com base na frequência dos itens ou com base apenas na quantidade de itens). Após o esquema de particionamento ser montado para cada partição dos itens não ordenados, cada thread ou processo ou máquina possui uma ideia de como dividir os itens entre os peers, mesmo que ainda de forma local. Para que se tenha um esquema global, há necessidade de um dos peers (uma thread, um processo ou uma máquina) receber todos os esquemas de particionamento locais e montar um esquema global de particionamento de itens com base nisto. Esta fase de montagem de um esquema global é a segunda fase da solução. Após o esquema global montado, este é enviado aos peers para que todos saibam como proceder o particionamento de seus itens locais, os enviando aos seus respectivos destinatários. A fase três se inicia com a varredura do mapa de itens-frequências criado na fase um. Para cada item se sabe com base no esquema de particionamento global qual peer é o destinatário deste item, portanto após a varredura do mapa, todos os itens e suas respectivas frequências já estão sob o controle de seus devidos destinatários. Como todas as fases ocorrem concorrentemente ou em paralelo, na fase três poderá haver muitas sincronizações para o recebimento de itens, pois um mesmo item poderá ser enviado por inúmeros remetentes para um mesmo destinatário e ao mesmo tempo. Para se evitar sincronizações, há como criar em cada peer um mapa onde a chave é o identificador do peer, seja este uma thread ou um processo ou uma máquina de um cluster. Os valores de tal mapa são outros mapas, onde a chave destes são os itens e os valores são suas respectivas frequências. Como exemplo, um peer 1, que possui os itens i1, i23, i34, sendo i1 atribuído pelo esquema de particionamento global ao peer 12, item i23 ao peer 3 e o item i34 ao peer 10, deverá enviar o item i1 ao peer 12 e este o armazenará no mapa com chave peer 1, valor sendo o mapa com chave item i1 e valor sendo a frequência de i1. O mesmo deverá ocorrer para os itens i23 e i34. Ao final da fase três todos os itens estarão em seus peers destinos e todo o procedimento de envio de itens poderá ocorrer concorrentemente sem qualquer demanda de sincronização. Na fase quatro cada peer (thread ou processo ou máquina de um cluster) varre seu mapa de mapas, ordena localmente seus itens segundo qualquer solução sequencial (quicksort, heapsort, etc..) e gera as frequências finais. Em linhas gerais, após a fase quatro os itens ficarão particionados e ordenados, portanto esta modelagem para ordenação de itens é extremamente útil para clusters de máquinas, pois mantém os itens ordenados e distribuídos. Outra vantagem ocorre no envio dos itens, pois apenas o item e sua frequência são enviados, o que gera para itens numéricos a transferência de apenas 64 ou 128 bits (float ou int de 32 bits ou double de 64 bits) para cada item, independentemente do quanto este se repete na coleção de itens. Assumimos que os itens não numéricos possuem alguma forma de cálculo de hash e também possuem o operador clone.



**2o Conjunto**

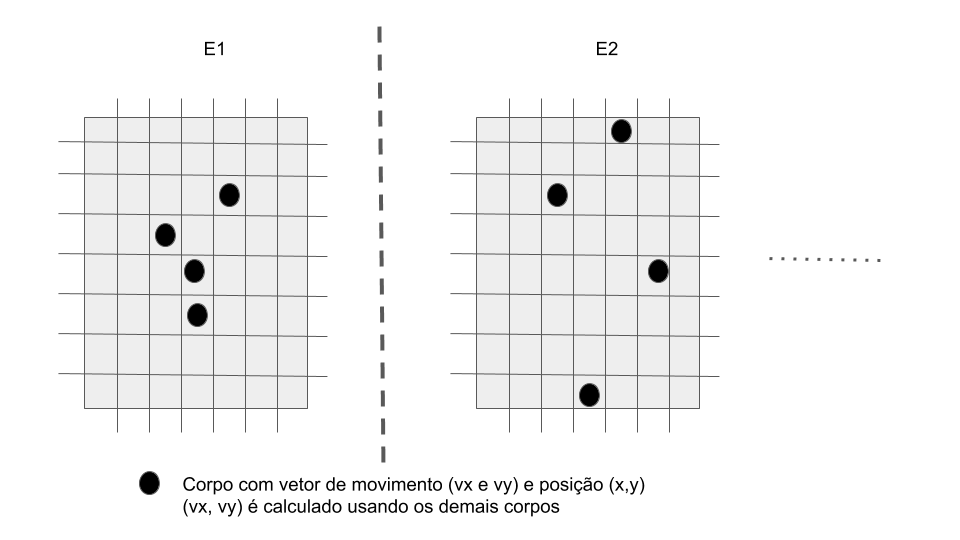
**Problema 1 - N-corpos;**

1. Entrada: uma matriz NxM com n corpos dispostos em tal matriz e cada corpo com n -1 valores de interferência em outros corpos;
2. Saída: uma série de eventos E ocorridos. Uma matriz NxM com n corpos dispostos em tal matriz e cada corpo com n-1 valores de interferência em outros corpos após E eventos;
3. Detalhar a alternativa de solução escolhida (modelo de programação utilizado - work pool, pipeline, data partition, master slave, híbrido) e suas razões;
4. Apresentar os experimentos e resultados experimentais, incluindo discussões;
5. Parametrize sua solução para aceitar diferentes entradas;

Material de apoio - <https://www.repositorio.ufop.br/bitstream/123456789/5664/1/DISSERTA%c3%87%c3%83O_TerraMEHPAArquitetura.pdf>

A modelagem do problema possui duas fases: os vetores de movimento dos corpos ou interferência dos corpos em outros corpos; e a atualização da posição dos corpos no espaço. Isto se repete por E eventos numa matriz NxM que representa o espaço, onde cada célula da matriz pode ser habitada por um ou vários corpos, totalizando n corpos. Podemos modelar os corpos em espaços tridimensionais ou bidimensionais. Por questão de simplicidade, optamos por espaços bidimensionais.

A primeira etapa consiste em calcular o vetor de movimento de cada corpo (vx, vy) considerando os demais corpos do espaço e propriedades do mesmo. Terminada a etapa de atualização dos vetores de movimento de todos os corpos, inicia-se a segunda etapa. Na segunda etapa, as posições (x, y) de todos os corpos devem ser atualizadas, considerando o vetor de movimento.



O método mais eficiente se chama FMM (Fast Multipole Method - <https://en.wikipedia.org/wiki/Fast_multipole_method>), mas não iremos nos concentrar nele.

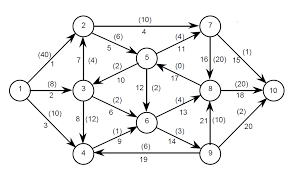
A fase de calcular o vetor de movimento de cada corpo considerando os demais corpos pode ter complexidade O(n) para n corpos, pois em paralelo cada corpo consegue realizar o cálculo em O(1) se este não dependesse dos demais corpos. Como depende, usando seus n-1 vizinhos e ele mesmo, a complexidade de cada execução em paralelo sobe para O(n) quando há n processadores disponíveis. Tal abordagem pode ser reduzida para O(lgn) se não considerarmos todos os n vizinhos e somente os que “mais influenciam” um determinado corpo. Outra alternativa é a adoção de mesh, hypercube ou ring para soluções distribuídas e com o intuito de reduzir a utilização de redes de comunicação de longa distância, pois com isto conseguimos mensagens enviadas num cluster para que todos os corpos recebam todos os vetores de movimento dos demais corpos, por exemplo. Caso contrário, há na versão ingênua n2 mensagens para obter os vetores de movimentos dos n corpos a cada evento da simulação.

A segunda fase de movimentação no espaço pode ser feita em O(1) com os mesmos n processadores disponíveis, pois basta cada corpo calcular para onde se movimenta na matriz NxM usando o vetor de movimento calculado previamente. Note que na modelagem escolhida, o espaço onde os corpos residem possuem propriedades (gravidade, nível de CO2 e qualquer outro índice não pertencente aos corpos) particulares ao mesmo. Para evitar que múltiplos corpos se movimentem para uma mesma célula do espaço num determinado evento da simulação, causando necessidade de sincronização, sugerimos modelar o espaço ou o ambiente onde os corpos residem como um cubo NxMxn, onde n é no pior caso o número de corpos. Em suma, cada célula do espaço ou ambiente é um vetor com até n posições e cada posição do vetor é reservada a cada corpo para que este resida. É claro que não há necessidade de todas as NxM células terem um vetor de n posições a priori, portanto há como montar tal estrutura sob demanda, ou seja, apenas quando um corpo reside numa célula da matriz há sua posição reservada em tal célula de forma a evitar qualquer sincronização no processo de movimentação dos corpos. A ideia é similar à fase 3 do algoritmo de ordenação explicado anteriormente, onde há como todos os peers enviarem os números para todos os peers sem qualquer bloqueio.

Para maiores informações sobre comunicação de todos com todos num cluster: Ring, Linear Array, Mesh e Hypercube - <http://users.atw.hu/parallelcomp/ch04lev1sec2.html>

**Problema 2 - Menor caminho (shortest path);**

1. Entrada: um grafo completo com V vértices E arcos entre os vértices. Em cada Ei há o custo de locomoção entre dois vértices em uma única direção. Além do grafo, há o vértice de origem e destino (Vo e Vd, respectivamente) para o cálculo do menor caminho;
2. Saída: um de vértices que representa o menor caminho entre Vo e Vd;
3. Detalhar a alternativa de solução escolhida (modelo de programação utilizado - work pool, pipeline, data partition, master slave, híbrido) e suas razões;
4. Apresentar os experimentos e resultados experimentais, incluindo discussões;
5. Parametrize sua solução para aceitar diferentes entradas;



Modelagem 1

Parallel All Pairs - <https://en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm#:~:text=A%20central%20problem%20in%20algorithmic,%2Dpaths%20(APSP)%20problem.>

Modelagem 2

Parallel single source - <https://en.wikipedia.org/wiki/Parallel_single-source_shortest_path_algorithm>

**3o Conjunto**

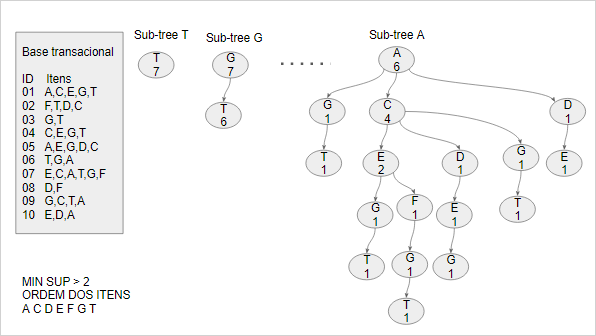
**Problema 1. Frequent Itemset**

Dada uma base de itens com inúmeras transações, queremos saber quais os conjuntos de itens com suporte mínimo maior que X (min\_sup > X). Vejamos:

Um bom paper - Frequent itemset mining: A 25 years review (de 2019 - [link](https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1329?casa_token=OTOfyy25q_YAAAAA:-Tv47P12bjzoXqE7ROpIQgc62ZH2nfl9mTWTQ7APSCHQ8dSIE8dBd7w5TufZqqMK16Rs1pr-gDw))

Modelagem:

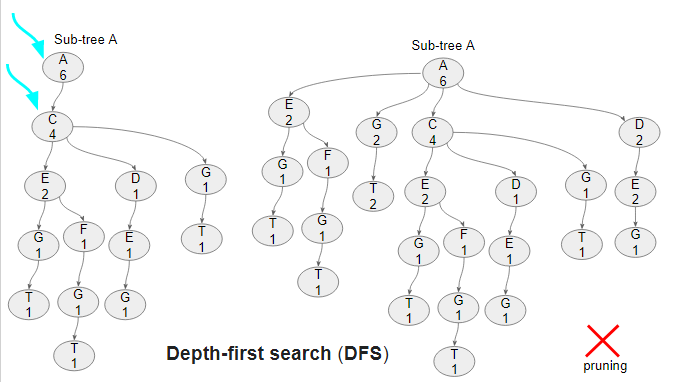
Construir um pipeline onde cada estágio é responsável por montar uma SUB-TREE de um item. A tupla com os itens passa por todos os estágios do pipeline. Vejamos o exemplo a seguir:



No exemplo da figura acima teremos A,C,D,E,F,G,T estágios no pipeline, ou seja, 7 estágios. Além dos sete estágios, há um oitavo que é o primeiro estágio, sendo responsável por ordenar os itens de cada tupla de acordo com a regra lexicográfica {A,C,D,E,F,G,T}, portanto todas as tuplas passam por uma ordenação de itens seguindo a regra explicada anteriormente. Cada nó de cada SUB-TREE possui seu label {A,C,D,E,F,G,T}, uma lista de outros nós chamada descendentes e um COUNT responsável por contabilizar a frequência daquele item.

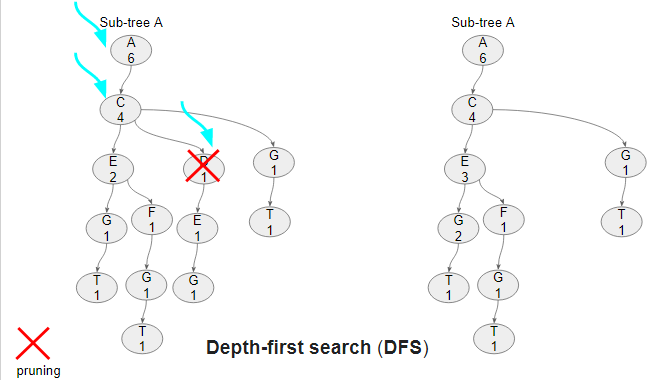
Note que cada SUB-TREE é uma estrutura prefixada onde os nós mais próximos da raiz são considerados nós agregados ou generalizados, pois possuem valores de COUNT sem muitos prefixos. Após uma única varredura da base, as inúmeras SUB-TREEs estão devidamente construídas em paralelo e a fase II do algoritmo se inicia.

A segunda fase realiza uma busca em profundidade, conforme ilustra a figura a seguir com a SUB-TREE com label A.



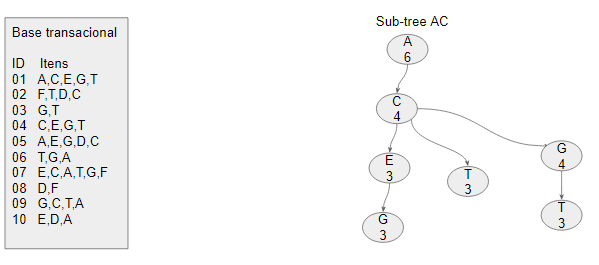
Inicialmente, se percorre o caminho AC e o algoritmo copia os descendentes de C para o nó A, conforme ilustra a figura acima. A SUB-TREE com raiz E é criada como descendente de A. A SUB-TREE com raiz D é fundida, pois A já possui um descendente D. Neste caso, os nós D e E passam a ter COUNT igual a 2 e o nó G é criado. Por fim a SUB-TREE com raiz G é fundida em A, pois este nó já possui descendente G.

Após varrer o nó C a operação DFS passa ao nó D, formando o caminho ACD. O algoritmo já percebe que o MIN\_SUP é inferior ao estipulado inicialmente, portanto o nó D não o satisfaz e nenhum descendente de D também o satisfará. Tal propriedade se chama Apriori Property. A ramificação a partir do nó D é punida, o nó D eliminado e seus descendentes copiados para formar um possível caminho a partir do nó C. A figura a seguir ilustra tal operação. Os nós E e G são fundidos aos descendentes de C, aumentando seus valores de COUNT para 3 e 2, respectivamente.



O algoritmo continua na operação de poda e DFS, assim como a cópia de SUB-TREEs. Maiores detalhes de como a SUB-TREE com label A é construída se encontra em: <https://docs.google.com/presentation/d/1PHLwjZBlmgqWzZ2w0A-VaHRBpYqB-yXOGvcOzV3_pPY/edit?usp=sharing>

Ao final do procedimento teremos:



Agora vejamos: como modelar a segunda fase de tal algoritmo de forma paralela? Uma primeira alternativa seria atribuir uma thread para cada SUB-TREE e a deixar proceder conforme explicado anteriormente, ou seja, em cada thread o algoritmo sequencial passaria a ser executado. Esta estratégia sofre de desbalanceamento de carga, fazendo com que, por exemplo, a thread com a SUB-TREE com label T acabe bem antes das demais e fique ociosa. Outra limitação: se houver mais threads ou cores do que SUB-TREES haverá ociosidade também. Dito isto, há alguma outra forma de modelarmos o problema da segunda fase em paralelo?

**Bibliografia**

GRAMA, Ananth. Introduction to parallel computing. 2. ed. Harlow, England: Addison Wesley, 2003.

KLEPPMANN, Martin. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. 1. ed. Nova York: Oreilly & Assoc, 2015.