УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Марко Николић

Apache Giraph и дистрибуирана обрада графова

СЕМИНАРСКИ РАД

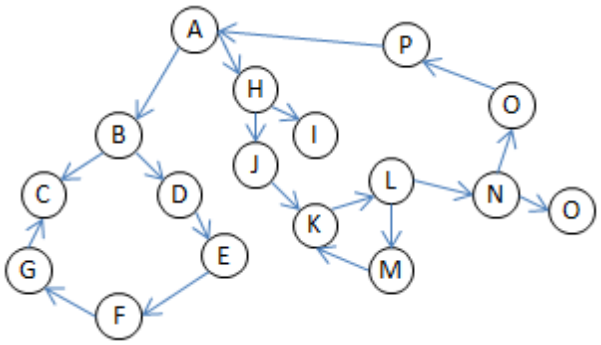
Нови Сад, 2025.

САДРЖАЈ

1. УВОД

1.1 Значај обраде графова

У савременим информационим системима количина података расте веома брзо. Велики део тих података поседује комплексне везе које се не могу представити једноставним структурама података. Многе области пословања као што су друштвене мреже, системи за препоруку, мапе или финансијски системи по природи генеришу податке који могу да се моделују као граф (слика 1). Графови омогућавају приказ ентитета (чворова) и односа између њих (грана) чиме се на интуитиван начин могу приказати подаци из оваквих система.



Слика 1 - Граф

Са брзим повећањем количине података у облику графа, расте и потреба за њиховом обрадом односно за алатима који омогућавају њихову обраду. Многе архитектуре и алгоритми који добро функционишу на малим скуповима података нису ефикасни у раду са графовима који садрже велики број чворова и веза. Због тога се развијају системи за дистрибуирану обраду графова, који искоришћавају паралелно и дистрибуирано програмирање како би омогућили обраду великих графова. Један од најпознатијих система за обраду великих графова је *Apache Giraph*, радни оквир заснован на *Google*-овом *Pregel* моделу, који омогућава обраду великих графова на *Apache Hadoop* инфраструктури.

1.2 Изазови при обради графова

Графови представљају неправилно структуриране податке, што их чини изазовним за обраду и складиштење. Постоји више разлога због којих граф проблеми постају тешки са повећањем самог графа:

1. Нелинеарна и неправилна структура података – Графови имају неуниформну структуру, за разлику од табеларних формата. Сваки чвор може имати различити број веза и сваки чвор и веза могу бити другачији.
2. Висока зависност међу подацима – Граф алгоритми при обради чворова често захтевају информације о суседним чворовима или скуповима чворова. Оваква

зависност спречава независну обраду над различитим деловима графа, због чега модели попут *MapReduce* модела нису погодни за обраду графова.

3. Меморијска захтевност – У савременим системима, графови могу бити веома велики, посебно у домену друштвених мрежа. Складиштење целог графа у *RAM* меморији једног рачунара постаје немогуће због чега је потребна дистрибуција података на рачунарски кластер.
4. Итеративна природа алгоритама – Многи алгоритми за обраду графова захтевају велики број итерација, где резултат једне итерације утиче на другу. Због тога системи који нису дизајнирани за итеративну обраду података нису погодни за обраду графова.
5. Потреба за паралелизацијом – Расподела графа по рачунарском кластеру је изазовна. Повезани чворови се често нађу на различитим рачунарима што повећава потребу за комуникацијом између рачунара у рачунарском кластеру. Смањење потребе за оваквом комуникацијом је изазован проблем.
6. Динамична структура графа – Графови се често мењају. Додају се нови чворови и везе, мењају се типови веза и додају се нови.

Због свих ових изазова, постоји потреба за специјализованим системима који могу ефикасно да обрађују велике и комплексне графове. Један од таквих система је *Apache Giraph*, који је дизајниран за дистрибуирану и паралелну обраду графова. *Apache Giraph* користи *vertex-centric* модел обраде, где сваки чвор обрађује своје поруке и комуницира са суседним чворовима, што омогућава рад са нелинеарним структурама и зависностима међу подацима. Такође, систем је дистрибуиран, што смањује проблем меморијске захтевности и омогућава скалабилну обраду графова који се не могу сместити у *RAM* меморију једног рачунара. Итеративни модел обраде (*Bulk Synchronous Parallel*) подржава алгоритме који захтевају више корака, а уграђена паралелизација и механизми за размештање података смањују комуникационе трошкове.

На овај начин, *Apache Giraph* директно адресира већину изазова при обради великих графова, што га чини погодним алатом за анализу друштвених мрежа и других сложених структура података.

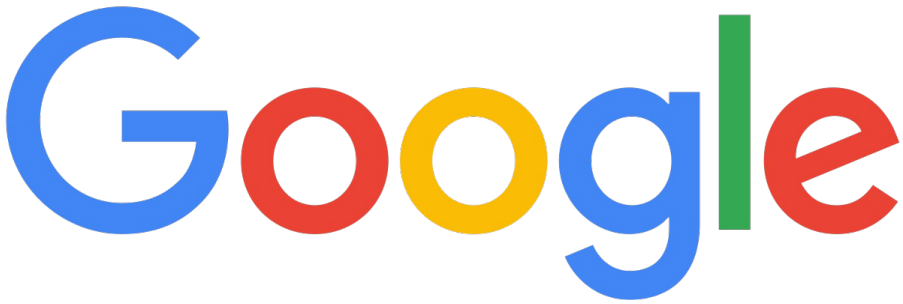
2. *APACHE GIRAPH* – ТЕОРИЈСКЕ ОСНОВЕ

2.1 Историјат

Apache Giraph (слика 2) је модел за дистрибуирану обраду великих графова. Омогућава паралелну и итеративну анализу графова са милијардама чворова и веза, користећи *Apache Hadoop* екосистем за складиштење података.

Слика 2 - *Apache Giraph* лого

Кључна инспирација за *Apache Giraph* потекла је од *Google*-овог (слика 3) *Pregel* модела, представљеног 2010. године. *Pregel* је увео *vertex-centric* парадигму, где сваки чвор самостално обрађује своје податке и комуницира са суседним чворовима.



***Слика 3 - Google* лого**

Први развој *Apache Giraph*-а започео је *Yahoo!* (слика 4) током 2011. године. *Yahoo!* је видео потребу за *open-source* алатом који би омогућио обраду великих графова на *Apache Hadoop* инфраструктури, која је већ била у широкој употреби за дистрибуирану обраду података. *Apache Giraph* је почео као имплементација *Pregel* концепта, али са фокусом на стабилност, ефикасност и скалабилност на *Apache Hadoop* кластерима.

Слика 4 - *Yahoo!* лого

Током лета 2011. године, *Apache Giraph* је преузет од стране *Apache Software Foundation*-а (слика 5), чиме је пројекат постао доступан широј *open-source* заједници. Ово је омогућило другим компанијама и истраживачима да доприносе развоју и тестирању система. 2012. године, *Apache Giraph* је промовисан у *Apache Top-Level Project*, што је значило да је пројекат достигао зрелост и стабилност која је погодна за продукцијску употребу.

Слика 5 - *The Apache Software Foundation* лого

Први корак у развоју *Apache Giraph*-а био је објављивање инкубационе верзије *0.1-incubating* у фебруару 2012. године. Након периода тестирања и развоја, маја 2013. године објављена је прва стабилна верзија *1.0.0*, која је потврдила стабилност система и богат скуп функционалности. Следеће верзије наставиле су да унапређују *Apache Giraph*, па је у новембру 2014. године објављена верзија *1.1.0*, затим у октобру 2016. године верзија *1.2.0*, док је најновија стабилна верзија *1.3.0* изашла у јуну 2020. године. Поред тога, 2015. године је објављена и књига *„Practical Graph Analytics with Apache Giraph“*, која је показала практичну примену и алгоритме за обраду графова у *Apache Giraph*-у.

Развој *Apache Giraph*-а долазио је из више значајних компанија и заједница. Поред *Yahoo!*-а као иницијатора пројекта, на систему су радили и доприносили програмери из *Facebook*-а (слика 6), *Twitter*-а (слика 6) и *LinkedIn*-а (слика 6), користећи *Apache Giraph* у продукцијским системима за анализу великих графова и тестирајући га на мрежама са стотинама милиона корисника и милијардама веза. *Facebook* је, на пример, активно користио *Apache Giraph* за анализу својих друштвених мрежа са стотинама милиона корисника и милијардама веза.

Слика 6 - *Facebook*, *Twitter* и *LinkedIn* лого

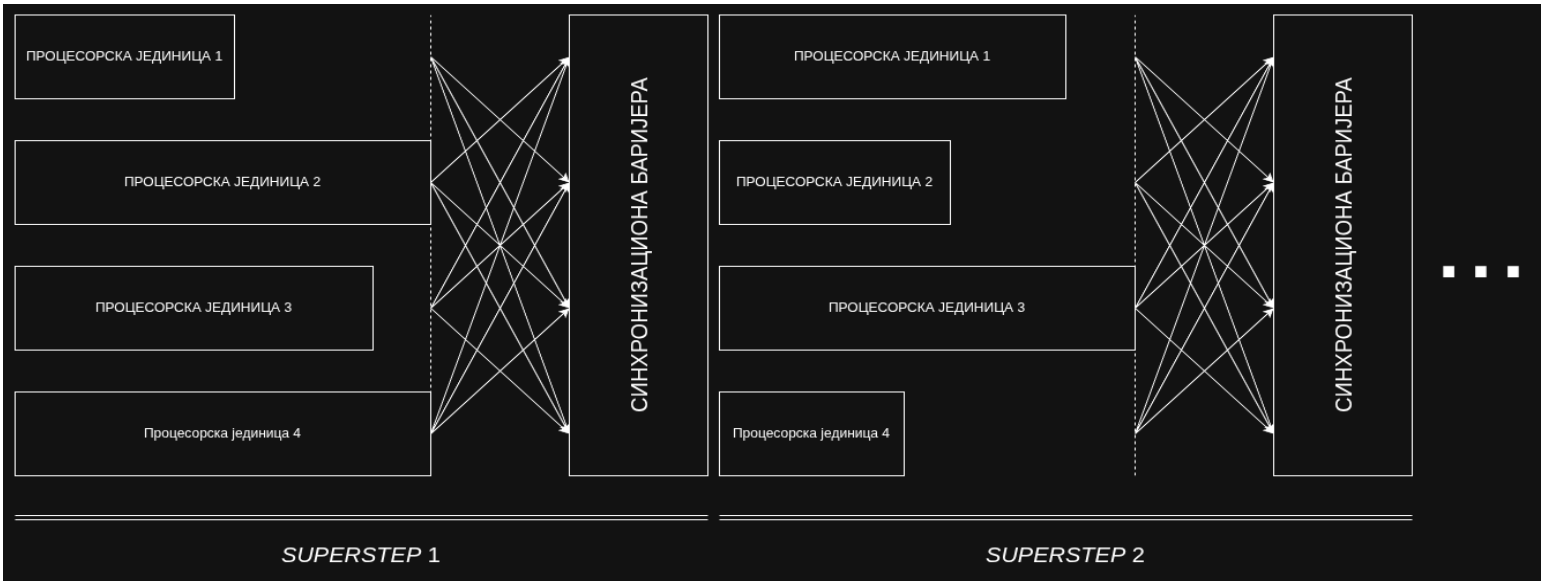
У септембру 2023. године, *Apache Giraph* је званично повучен из активног развоја и премештен у *Apache Attic*, а процес премештања је завршен у фебруару 2024. године. Премештање у *Apache Attic* означава да пројекат више није у активној фази развоја, али остаје доступан за коришћење.

2.2 Програмски модел

Apache Giraph изграђен је на основама *Bulk Synchronous Parallel* (*BSP*) модела, који представља један од приступа за пројектовање паралелних и дистрибуираних алгоритама. Он полази од идеје да се обрада велике количине података може поделити на више независних процесорских јединица које раде паралелно, при чему је ток извршавања организован у кораке.

Основни елемент *Bulk Synchronous Parallel* (*BSP*) модела је концепт *superstep*-а, који представља једну итерацију паралелног алгоритма и основну градивну јединицу извршавања. Сваки *superstep* се састоји из 3 фазе (слика 7):

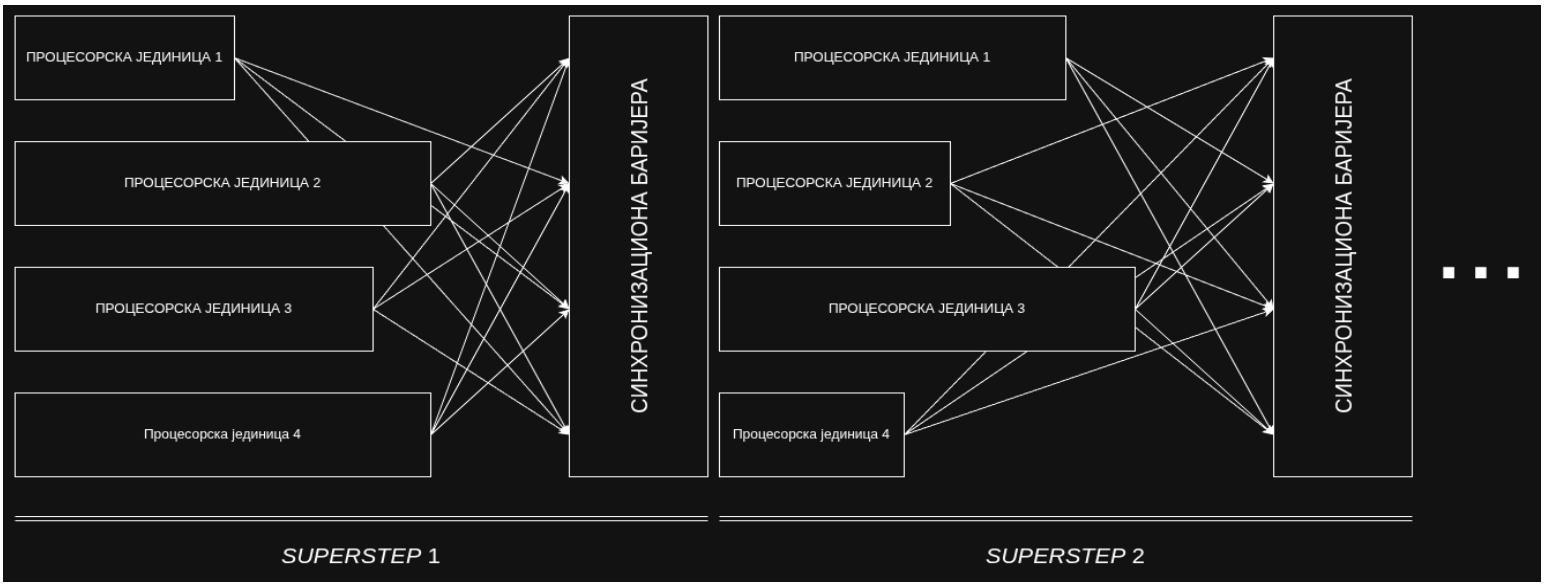
1. Локална обрада – Свака процесорска јединица располаже својим подскупом података и независно од других процесорских јединица обавља свој део посла над подацима. У овој фази не постоји комуникација између јединица, већ је нагласак на изолацији и паралелизму.
2. Комуникација – Након локалне обраде, процесорске јединице могу слати и примати поруке једна од друге. На овај начин преносе се резултати обраде који су неопходни за наставак обраде у следећим корацима (*superstep*-овима).
3. Синхронизациона баријера – На крају сваког *superstep*-а уводи се синхронизациона баријера која обезбеђује да све јединице заврше и обраду и размену порука пре него што било која од њих пређе на следећу итерацију.



Слика 7 - *Bulk Synchronous Parallel* (*BSP*) модел

Apache Giraph у великој мери примењује концептуалне основе *Bulk Synchronous Parallel* (*BSP*) модела. Програмер при писању програма не мора директно размишљати о *superstep*-овима и синхронизационим баријерама, сам механизам извршавања у *Apache Giraph*-у је дубоко укорењен у *Bulk Synchronous Parallel* (*BSP*) принципима. Граф је подељен на делове који се обрађују локално, чворови размењују поруке између *superstep*-ова, а синхронизација обезбеђује да се *superstep*-ови извршавају један за другим. На тај начин, *Apache Giraph* апстрахује низ техничких детаља, али у својој суштини функционише као граф-оријентисана специјализација *Bulk Synchronous Parallel* (*BSP*) модела, оптимизована за итеративне паралелне алгоритме над великим скупом чворова и веза.

Међутим, иако заснован на концептима *Bulk Synchronous Parallel* (*BSP*) модела, *Apache Giraph* у свом извршавању доноси неке специфичности. У *Apache Giraph*-у локална обрада чворова и комуникација могу да се преклапају (слика 8). Док *Bulk Synchronous Parallel* (*BSP*) модел подразумева да све јединице заврше локалну обраду пре него што започну комуникацију, *Apache Giraph* омогућава да се поруке шаљу одмах након што се генеришу од стране чворова. Ово смањује оптерећење мреже и омогућава ефикаснију паралелну обраду.



Слика 8 - Преклапање локалне обраде и комуникације чворова

Apache Giraph примењује *vertex–centric* модел обраде графа, у коме је основна идеја да сваки чвор (*vertex*) у графу самостално извршава своје прорачуне током сваког superstep-а. Уместо да се граф обрађује као целина, тај процес се разбија по чворовима. Овај приступ је оригинално идеја *Google*-овог *Pregel* модела који је први пут представљен 2010. године и који је послужио као инспирација за *Apache Giraph* и друге сличне системе.

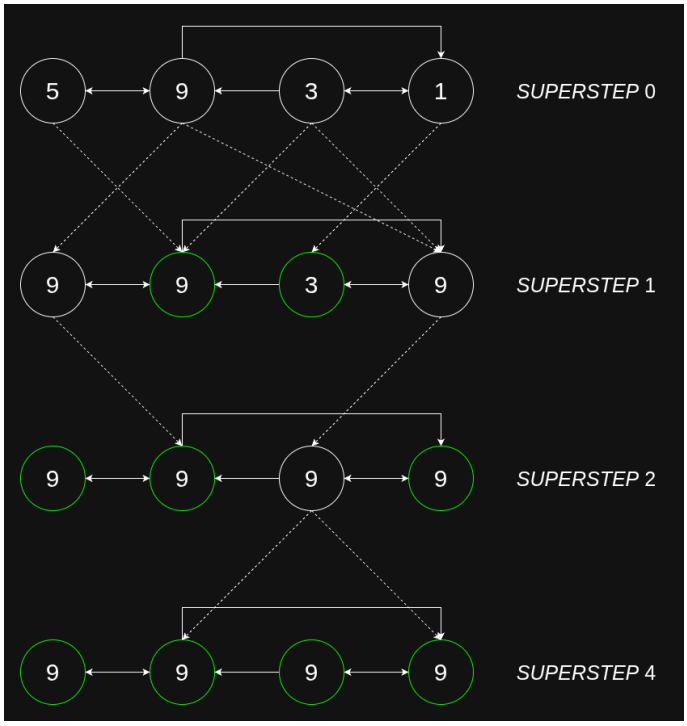
Током једног *superstep*-а дешава се следеће:

1. Чвор прима све поруке које су му други чворови послали у претходном *superstep*-у**.** Ове поруке представљају улазне податке потребне за израчунавање новог стања чвора.
2. Чвор ажурира своје стање на основу примљених порука и претходног стања. Ово је локална обрада која не зависи од целог графа, већ само од информација које чвор има и порука које је примио.
3. Чвор шаље поруке суседним чворовима. Ове поруке садрже резултате обраде који ће суседи користити у следећем *superstep*-у.
4. Чвор може да се деактивира ако више нема шта да ради. Ово представља механизам којим чвор сигнализира да је завршио своју улогу у алгоритму, али ако му стигне порука у неком каснијем *superstep*-у, он се аутоматски буди и наставља рад. У *Apache Giraph*-у алгоритам се завршава тек када сви чворови буду деактивирани и када нема нових порука.

Описани модел обраде података приказан је на слици 9 на примеру проналажења максимума:

1. Чворови графа у свакој итерацији (*superstep*-у) алгоритма својим суседима шаљу информацију о највећој вредности за коју тај чвор зна.
2. Чвор који прими поруку пореди добијену вредност са највећом вредношћу за коју он зна.

3. Уколико је нова вредност већа, ажурира своје стање и шаље поруку са новом вредношћу суседима.
4. Уколико је тренутна вредност ипак већа и не долази до промене, чвор гласа за прекид алгоритма и привремено се деактивира.
5. Алгоритам се извршава докле год има активних чворова или порука.



Слика 9 - *vertex-centric* алгоритам за проналажење максимума у графу

2.3 Архитектура

Архитектура *Apache Giraph*-а заснива се на скупу сервиса који заједно омогућавају обраду великих графова у оквиру *Bulk Synchronous Parallel* (*BSP*) модела. Три основна сервиса су: *master*, *worker* и координатор.

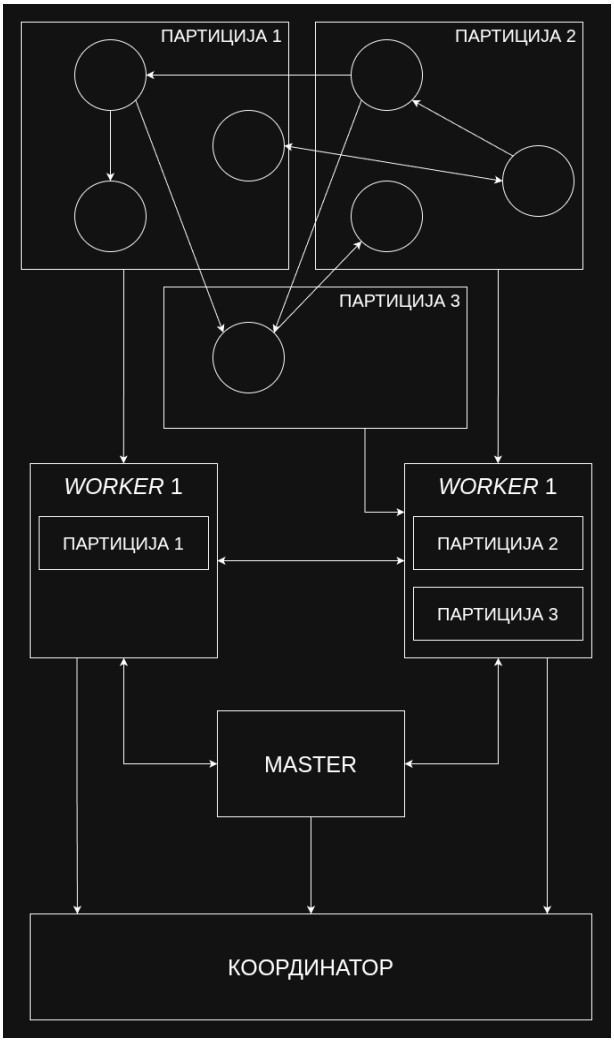
Master представља централни сервис који управља током извршавања *Apache Giraph* апликације. У сваком моменту постоји један активан *master* и неколико резервних *master* сервиса који чекају да преузму улогу у случају отказа активног. Када је активан, *master* контролише прелазак између *superstep*-ова, координира рад свих *worker*-а и покреће *master* прорачуне ако их програмер дефинише. Он пре сваког *superstep*-а додељује граф партиције појединим *worker* сервисима, а током извршавања прати њихово стање, прикупља статистике и реагује на отказ било ког *worker*-а. На тај начин, *master* сервис представља управљачку јединицу читавог *Apache Giraph* кластера, обезбеђујући да све компоненте раде синхронизовано у оквиру *Bulk Synchronous Parallel* (BSP) итерација.

Worker-и су радни део *Apache Giraph* архитектуре и уједно најзаступљенији. Сваки *worker* управља једним бројем граф партиција које му *master* додели. Током *superstep*-а, *worker* пролази кроз све чворове у својим партицијама и за сваки врши дефинисане прорачуне. Поред обраде локалних чворова, w*orker*-и размењују поруке путем интерних

remote procedure call (*RPC*) механизама, омогућавајући међусобну комуникацију чворова графа. Важно је нагласити да партиције нису трајно везане за одређеног *worker*-а, расподела се може мењати пре сваког *superstep*-а, зависно од статистика које *master* добија. Ово динамичко ребалансирање омогућава бољу искоришћеност ресурса и равномерније оптерећење. *Worker*-и такође периодично снимају *checkpoint* стања, што омогућава поуздан опоравак у случају пада појединачног сервиса.

Координатор сервиси омогућавају синхронизацију и управљање конфигурационим подацима који су неопходни за функционисање *master*-а и *worker*-а. Иако не учествују директно у обради графа, представљају кључни део архитектуре јер чувају метаподатке о кластеру, статусима сервиса и мапирању партиција. *Apache Giraph* најчешће користи *Apache ZooKeeper* као координаторски сервис.

На слици 10 приказана је архитектура *Apache Giraph*-а.



Слика 10 - *Apache Giraph* архитектура

3. *APACHE GIRAPH* – ПРИМЕНА

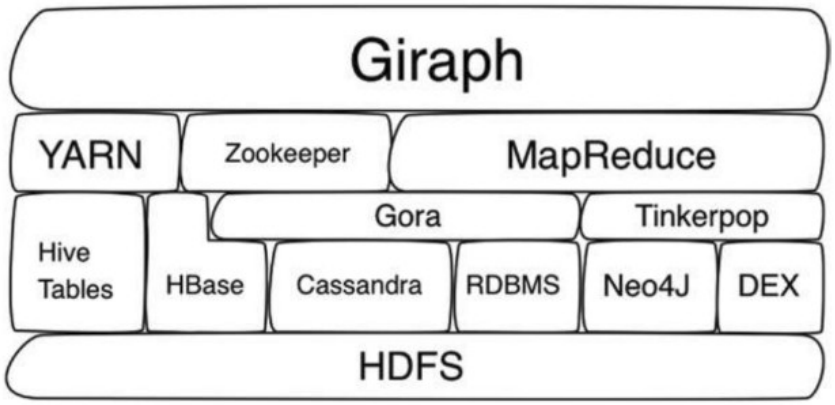
3.1 Технологије

Apache Giraph сам по себи не представља потпуно самосталну платформу. Напротив, *Apache Giraph* користи низ технологија и инфраструктурних компоненти, од којих свака игра улогу у систему.

Најважнија технологија коју *Apache Giraph* користи је *Apache Hadoop* (слика 11). Иако се у алгоритамском смислу *Apache Giraph* ослања на Pregel и Bulk Synchronous Parallel (*BSP*) модел, у практичном смислу он се ослања на Hadoop компоненте као што су *Hadoop YARN* (за управљање ресурсима у кластеру) и *HDFS* (за складиштење улазних и излазних података). Захваљујући томе, Giraph може да искористи постојећу *Apache Hadoop* инфраструктуру, што значајно поједностављује интеграцију у окружења у којима се већ обрађује велика количина података.

Слика 11 - *Apache Hadoop* лого

Apache Giraph се може интегрисати са различитим технологијама из *Apache Hadoop* екосистема (слика 12). Захваљујући томе, може да користи различите системе за складиштење података, координацију, машинско учење и обраду великих графова. Међу најзначајнијим технологијама са којима се *Apache Giraph* интегрише су: *MapReduce*, *HBase*, *Cassandra*, *Hive*, *HCatalog*, *Gora*, *Hama*, *Mahout* и *Nutch*.



Слика 12 - *Apache Giraph* интеграција са *Apache Hadoop* пројектима

Други кључни ослонац *Apache Giraph*-а је *Apache ZooKeeper* (слика 13). *Apache ZooKeeper* се користи као координаторски систем који обезбеђује синхронизацију између *master*-а, *worker*-а и других сервиса.

Слика 13 - *Apache ZooKeeper* лого

Apache Giraph је у потпуности изграђен у програмском језику *Java* (слика 14), што значи да се читав систем ослања на *Java Virtual Machine* (*JVM*). Пошто је и *Apache Hadoop* развијен у *Javi*, *Apache Giraph* се природно уклапа у његов екосистем. Употреба Јаве такође осигурава преносивост, *Apache Giraph* може да се извршава на различитим оперативним системима и конфигурацијама без измена у коду.

Слика 14 - *Java* лого

3.2 API

Apache Giraph API је заснован на *vertex–centric* програмском моделу, где се алгоритми дефинишу из перспективе појединачног чвора у графу. Основу *API*-ја чине класе које описују структуру графа (*Vertex*, *Edge*) и класе које управљају извршавањем алгоритма (*BasicComputation*, *Aggregator, MessageCombiner*). *API* је доступан у *Java* програмском језику.

Класа *Vertex* представља чвор графа. Методе које садржи су:

1. *getId()* – враћа јединствени идентификатор чвора.
2. *getValue()* – враћа вредност чвора, обично стање које алгоритам обрађује.
3. *setValue(value)* – поставља нову вредност чвора.
4. *getEdges()* – враћа колекцију свих излазних грана из чвора.
5. *getNumEdges()* – враћа укупан број излазних грана.
6. *getEdgeValue(targetId)* – враћа вредност гране која води ка одређеном чвору.
7. *setEdgeValue(targetId, value)* – мења вредност постојеће гране ка одређеном чвору.
8. *getAllEdgeValues(targetId)* – враћа све вредности грана ка одређеном чвору.
9. *voteToHalt()* – означава да чвор нема више посла, деактивација чвора.
10. *addEdge(edge)* – додаје нову излазну грану чвору.
11. *removeEdges(targetId)* – уклања све гране које воде ка одређеном чвору.

Класа *Edge* представља грану графа. Методе које садржи су:

1. *getTargetVertexId()* – враћа идентификатор циљног чвора на који грана води.
2. *getValue()* – враћа вредност гране.
3. *setValue(value)* – поставља или мења вредност гране.

Класа *BasicComputation* је основна класа за имплементацију алгоритама у *Apache Giraph*-у. Она садржи методе које омогућавају обраду чворова и порука током *superstep*-ова. Методе су:

1. *compute(vertex, messages)* – главна метода која се мора имплементирати, позива је *Apache Giraph* током извршавања *superstep*-а за сваки активни чвор и прослеђује поруке које је чвор примио.
2. *getSuperstep()* – враћа тренутни број *superstep*-а, корисно за праћење тока алгоритма.
3. *getTotalNumVertices()* – враћа укупан број чворова у графу.
4. *getTotalNumEdges()* – враћа укупан број грана у графу.
5. *sendMessage(targetId, message)* – шаље поруку од тренутног чвора ка циљном чвору.
6. *sendMessageToAllEdges(vertex, message)* – шаље исту поруку свим суседним чворовима.
7. *addVertexRequest(vertexId)* – захтев за додавање новог чвора у граф током извршавања.
8. *removeVertexRequest(vertexId)* – захтев за уклањање чвора из графа током извршавања.

Класа *MessageCombiner* омогућава комбиновање више порука које иду ка истом циљном чвору. Једина метода коју садржи је:

1. *combine(id, message1, message2)* – враћа комбиновану вредност две поруке које су послате ка чвору са датим идентификатором.

Класа Aggregator је механизам у *Apache Giraph*-у који омогућава сабирање или обраду вредности са више чворова као једне глобалне вредности. На пример, ако желимо да нађемо највећу вредност у целом графу, сваки чвор може послати своју вредност агрегатору, који ће израчунати максимум. Методе које садржи су:

1. *aggregate(value)* – додаје нову вредност у агрегацију.
2. *getAggregatedValue()* – враћа резултат агрегације.
3. *setAggregatedValue(value)* – поставља резултат агрегације на одређену вредност.
4. *reset()* – почиње агрегацију од почетка, без старих вредности.

Класа *TextVertexInputFormat* служи за дефинисање начина на који *Apache Giraph* чита улазне податке из текстуалног фајла и претвара их у чворове и гране графа. Најчешће се користи када су подаци записани у текстуалном облику, при чему свака линија представља један чвор и његове везе.

Класа *TextVertexOutputFormat* дефинише начин на који *Apache Giraph* уписује резултате израчунавања у излазни текстуални фајл.

Поред наведених, постоје и бројне друге класе и механизми у *Apache Giraph*-у који омогућавају флексибилну обраду графова. За потребе овог рада описане су само најзначајније компоненте.

3.3 Пример – локално покретање *Apache Giraph*-а

У овом примеру приказано је локално покретање Apache Giraph-а. Инфраструктура је реализована помоћу *Docker*-а и *Docker Compose*-а и обухвата све неопходне компоненте за покретање *Apache Giraph* апликација, укључујући *Apache Hadoop* и *Apache Giraph*. Примери и изворни код доступни су на *GitHub* репозиторијуму https://github.com/marko-nikolic01/apache-giraph-graph-processing.

Основу чини *docker-compose.yml* фајл (слика 15) који дефинише један сервис, *apache-giraph*. Тај сервис се гради из локалног директоријума који садржи *Dockerfile* са *Apache Giraph* и *Apache Hadoop* окружењем.

```yaml
version: "3.8"

services:
  apache-giraph:
    build: ./apache-giraph
    container_name: apache-giraph
    volumes:
      - ./apache-giraph/input:/data/input
      - ./apache-giraph/output:/data/output
    tty: true
```

Слика 15 - *docker-compose.yml*

Dockerfile (слика 16) инсталира *Apache Hadoop* и *Apache Giraph*, копира конфигурационе фајлове и *Apache Giraph* апликације, затим их компајлира помоћу *Maven*-а.

```
FROM maven:3.9.0-eclipse-temurin-11

ENV HADOOP_VERSION=2.10.1
ENV GIRAPH_VERSION=1.3.0
ENV GIRAPH_HADOOP_CLASSIFIER=hadoop2

WORKDIR /opt

RUN apt-get update && apt-get install -y wget ssh rsync && rm -rf /var/lib/apt/lists/*

RUN wget https://archive.apache.org/dist/hadoop/common/hadoop-${HADOOP_VERSION}/hadoop-${HADOOP_VERSION}.tar.gz \
    && tar -xzf hadoop-${HADOOP_VERSION}.tar.gz \
    && rm hadoop-${HADOOP_VERSION}.tar.gz

COPY giraph-dist-${GIRAPH_VERSION}-${GIRAPH_HADOOP_CLASSIFIER}-bin.tar.gz /opt/
RUN tar -xzf giraph-dist-${GIRAPH_VERSION}-${GIRAPH_HADOOP_CLASSIFIER}-bin.tar.gz -C /opt/ \
    && rm giraph-dist-${GIRAPH_VERSION}-${GIRAPH_HADOOP_CLASSIFIER}-bin.tar.gz

ENV HADOOP_HOME=/opt/hadoop-${HADOOP_VERSION}
ENV HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
ENV GIRAPH_HOME=/opt/giraph-${GIRAPH_VERSION}-${GIRAPH_HADOOP_CLASSIFIER}
ENV PATH=$PATH:$HADOOP_HOME/bin
ENV CLASSPATH=$CLASSPATH:$GIRAPH_HOME/lib/*

COPY ./hadoop/core-site.xml $HADOOP_CONF_DIR/core-site.xml
COPY ./hadoop/mapred-site.xml $HADOOP_CONF_DIR/mapred-site.xml
COPY ./hadoop/yarn-site.xml $HADOOP_CONF_DIR/yarn-site.xml
COPY ./../apps /opt/apps
RUN mkdir -p /data/input /data/output

WORKDIR /opt/apps/max-value
RUN mvn clean package -DskipTests

WORKDIR /opt/apps/shortest-distance-to-capital-city
RUN mvn clean package -DskipTests

WORKDIR /opt/apps/movie-recommendation
RUN mvn clean package -DskipTests

WORKDIR /opt

CMD ["bash"]
```

Слика 16 - *Dockerfile*

Контејнер се након покретања користи у интерактивном режиму. *Apache Giraph* апликације се покрећу тако што се проследе: име апликације, име класе у којој је дефинисана *compute* метода, име класе која конвертује улазни фајл у граф, име класе која конвертује граф у излазни фајл, као и имена улазног и излазног фајла. Описане команде налазе се на слици 17 и коришћене су за покретање примера из овог рада.

```
docker compose up --build

docker exec -it apache-giraph bash

cd /opt/apps/{app-name} && \
OUT="test-$(date +%Y%m%d-%H%M%S)" && \
hadoop jar target/{jar-name} \
  org.apache.giraph.GiraphRunner \
  {ComputationClassName} \
  -vif {InputFormatClassName} \
  -vof {OutputFormatClassName} \
  -vip /data/input/{input-file-name} \
  -op /data/output/$OUT \
  -w 1 \
  -ca giraph.SplitMasterWorker=false
```

Слика 17 - Команде за покретање *Apache Giraph* апликација

3.4 Пример – проналажење највеће вредности

У следећем примеру представљена је једноставна *Apache Giraph* апликација којом се у графу проналази највећа вредност помоћу слања порука.

Улазни подаци (слика 18) налазе се у текстуалном фајлу где је сваки чвор представљен као један ред у ком су редом написани идентификатор чвора, иницијална вредност чвора и низ идентификатора чворова ка којима постоји грана из тренутног чвора. Улазни формат података описан је и програмски (слика 19).

```
0 5 1
1 9 0 3
2 3 1 3
3 1 2
```

Слика 18 - Проналажење највеће вредности – улазни подаци

```java
package mv;

import org.apache.giraph.io.formats.TextVertexInputFormat;
import org.apache.giraph.edge.Edge;
import org.apache.giraph.edge.EdgeFactory;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class MaxValueVertexInputFormat extends TextVertexInputFormat<LongWritable, LongWritable, NullWritable> {

    @Override
    public TextVertexReader createVertexReader(InputSplit split, TaskAttemptContext context) {
        return new MaxValueVertexReader();
    }

    public class MaxValueVertexReader extends TextVertexReaderFromEachLineProcessed<String[]> {

        @Override
        protected String[] preprocessLine(Text line) throws IOException {
            return line.toString().split("\\s+");
        }

        @Override
        protected LongWritable getId(String[] tokens) throws IOException {
            return new LongWritable(Long.parseLong(tokens[0]));
        }

        @Override
        protected LongWritable getValue(String[] tokens) throws IOException {
            return new LongWritable(Long.parseLong(tokens[1]));
        }

        @Override
        protected Iterable<Edge<LongWritable, NullWritable>> getEdges(String[] tokens) throws IOException {
            List<Edge<LongWritable, NullWritable>> edges = new ArrayList<>();

            for (int i = 2; i < tokens.length; i++) {
                edges.add(EdgeFactory.create(new LongWritable(Long.parseLong(tokens[i])), NullWritable.get()));
            }

            return edges;
        }
    }
}
```

Слика 19 - Проналажење највеће вредности – дефиниција улазног формата података

У току алгоритма (слика 20), у оквиру сваког *superstep*-а, сваки чвор прати највећу вредност за коју зна. У *superstep*-у 0, сваки чвор шаље своју вредност ка суседним чворовима. Чворови обрађују добијене поруке и уколико наиђу на већу вредност, ажурирају своје стање и шаљу ту информацију својим суседима за обраду у наредном *superstep*-у. Овај процес се понавља док у систему постоје поруке и прекида се у оном *superstep*-у у ком сваки чвор задржи претходно стање и када сваки чвор гласа за прекид алгоритма.

```java
package mv;

import org.apache.giraph.graph.BasicComputation;
import org.apache.giraph.graph.Vertex;
import org.apache.giraph.edge.Edge;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;

import java.io.IOException;

public class MaxValueComputation extends BasicComputation<LongWritable, LongWritable, NullWritable, LongWritable> {

    @Override
    public void compute(Vertex<LongWritable, LongWritable, NullWritable> vertex, Iterable<LongWritable> messages) throws IOException {
        long currentMax = vertex.getValue().get();

        for (LongWritable msg : messages) {
            currentMax = Math.max(currentMax, msg.get());
        }

        if (currentMax > vertex.getValue().get() || getSuperstep() == 0) {
            vertex.setValue(new LongWritable(currentMax));
            for (Edge<LongWritable, NullWritable> edge : vertex.getEdges()) {
                sendMessage(edge.getTargetVertexId(), new LongWritable(currentMax));
            }
        }

        vertex.voteToHalt();
    }
}
```

Слика 20 - Проналажење највеће вредности – алгоритам

Излазни подаци (слика 21) уписују се у текстуални фајл где је сваки чвор представљен као један ред у ком су редом написани идентификатор чвора и највећа вредност. Излазни формат података описан је и програмски (слика 22).

```
0   9
1   9
2   9
3   9
```

Слика 21 - Проналажење највеће вредности – излазни подаци

```java
package mv;

import org.apache.giraph.io.formats.TextVertexOutputFormat;
import org.apache.giraph.io.formats.IdWithValueTextOutputFormat;
import org.apache.giraph.edge.Edge;
import org.apache.giraph.graph.Vertex;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

import java.io.IOException;

public class MaxValueVertexOutputFormat extends TextVertexOutputFormat<LongWritable, LongWritable, NullWritable> {

    @Override
    public TextVertexWriter createVertexWriter(TaskAttemptContext context) throws IOException, InterruptedException {
        return new MaxValueVertexWriter();
    }

    public class MaxValueVertexWriter extends TextVertexWriter {
        @Override
        public void writeVertex(Vertex<LongWritable, LongWritable, NullWritable> vertex) throws IOException, InterruptedException {
            getRecordWriter().write(new Text(vertex.getId().toString()), new Text(vertex.getValue().toString()));
        }
    }
}
```

Слика 22 - Проналажење највеће вредности – дефиниција излазног формата података

3.5 Пример – проналажење најкраћег пута до главног града

У следећем примеру представљена је *Apache Giraph* апликација којом се у графу градова и путева проналази дужина најкраћег пута до главног града за сваки град у графу.

Градови представљају чворове графа, а путеви заједно са својом дужином представљају гране. Сви путеви су двосмерни.

Улазни подаци (слика 23) налазе се у текстуалном фајлу где је сваки чвор представљен као један ред у ком су редом написани идентификатор чвора (назив града), почетна вредност (0 за главни град и велика вредност за остале градове) и низ парова где је први елемент сваког пара назив града ка коме постоји пут, а други елемент пара је дужина пута. Улазни формат података описан је и програмски (слика 24).

```
Belgrade 0 Novi_Sad 100 Nis 200 Subotica 300 Kragujevac 150
Novi_Sad 99999 Belgrade 100 Nis 100 Subotica 200 Pancevo 120
Nis 99999 Belgrade 200 Novi_Sad 100 Kragujevac 180 Leskovac 90 Vranje 120
Subotica 99999 Belgrade 300 Novi_Sad 200 Zrenjanin 80 Sombor 150
Kragujevac 99999 Belgrade 150 Nis 180 Pancevo 140
Pancevo 99999 Novi_Sad 120 Kragujevac 140 Zrenjanin 110
Leskovac 99999 Nis 90 Vranje 70
Vranje 99999 Leskovac 70 Nis 120
Zrenjanin 99999 Subotica 80 Pancevo 110 Sombor 130
Sombor 99999 Subotica 150 Zrenjanin 130
```

Слика 23 - Проналажење најкраћег пута до главног града – улазни подаци

```java
package sdtcc;

import org.apache.giraph.io.formats.TextVertexInputFormat;

import org.apache.giraph.edge.Edge;
import org.apache.giraph.edge.EdgeFactory;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class ShortestDistanceToCapitalCityVertexInputFormat extends TextVertexInputFormat<Text, LongWritable, LongWritable> {

    @Override
    public TextVertexReader createVertexReader(InputSplit split, TaskAttemptContext context) {
        return new ShortestDistanceToCapitalCityVertexReader();
    }

    public class ShortestDistanceToCapitalCityVertexReader extends TextVertexReaderFromEachLineProcessed<String[]> {

        @Override
        protected String[] preprocessLine(Text line) throws IOException {
            return line.toString().split("\\s+");
        }

        @Override
        protected Text getId(String[] tokens) throws IOException {
            return new Text(tokens[0]);
        }

        @Override
        protected LongWritable getValue(String[] tokens) throws IOException {
            return new LongWritable(Long.parseLong(tokens[1]));
        }

        @Override
        protected Iterable<Edge<Text, LongWritable>> getEdges(String[] tokens) throws IOException {
            List<Edge<Text, LongWritable>> edges = new ArrayList<>();

            for (int i = 2; i < tokens.length; i += 2) {
                edges.add(EdgeFactory.create(new Text(tokens[i]), new LongWritable(Long.parseLong(tokens[i + 1]))));
            }

            return edges;
        }
    }
}
```

Слика 24 - Проналажење најкраћег пута до главног града – дефиниција улазног формата података

У току алгоритма (слика 25), у оквиру сваког *superstep*-а, сваки чвор прати најкраћи пут за који зна. У *superstep*-у 0, сваки чвор шаље своју вредност ка суседним чворовима. Чворови обрађују добијене поруке и уколико наиђу на мању вредност, ажурирају своје стање, а свим суседима шаљу нови најкраћи пут сабран са дужином пута до суседа. Овај процес се понавља док у систему постоје поруке и прекида се у оном *superstep*-у у ком сваки чвор задржи претходно стање и када сваки чвор гласа за прекид алгоритма.

```java
package sdtcc;

import org.apache.giraph.graph.BasicComputation;
import org.apache.giraph.graph.Vertex;
import org.apache.giraph.edge.Edge;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

import java.io.IOException;

public class ShortestDistanceToCapitalCityComputation extends BasicComputation<Text, LongWritable, LongWritable, LongWritable> {

    @Override
    public void compute(Vertex<Text, LongWritable, LongWritable> vertex, Iterable<LongWritable> messages) throws IOException {
        long currentDistance = vertex.getValue().get();
        long minDistance = currentDistance;

        for (LongWritable msg : messages) {
            minDistance = Math.min(minDistance, msg.get());
        }

        if (minDistance < currentDistance || getSuperstep() == 0) {
            vertex.setValue(new LongWritable(minDistance));

            for (Edge<Text, LongWritable> edge : vertex.getEdges()) {
                long distanceToNeighbor = minDistance + edge.getValue().get();
                sendMessage(edge.getTargetVertexId(), new LongWritable(distanceToNeighbor));
            }
        }

        vertex.voteToHalt();
    }
}
```

Слика 25 - Проналажење најкраћег пута до главног града – алгоритам

Излазни подаци (слика 26) уписују се у текстуални фајл где је сваки чвор представљен као један ред у ком су редом написани идентификатор чвора (назив града) и дужина најкраћег пута до главног града. Излазни формат података описан је и програмски (слика 27).

```
Pancevo 220
Sombor 450
Leskovac 290
Kragujevac 150
Nis 200
Zrenjanin 330
Belgrade 0
Vranje 320
Novi_Sad 100
Subotica 300
```

Слика 26 - Проналажење најкраћег пута до главног града – излазни подаци

```java
package sdtcc;

import org.apache.giraph.io.formats.TextVertexOutputFormat;
import org.apache.giraph.graph.Vertex;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

import java.io.IOException;

public class ShortestDistanceToCapitalCityVertexOutputFormat extends TextVertexOutputFormat<Text, LongWritable, LongWritable> {

    @Override
    public TextVertexWriter createVertexWriter(TaskAttemptContext context) throws IOException {
        return new ShortestDistanceToCapitalCityVertexWriter();
    }

    public class ShortestDistanceToCapitalCityVertexWriter extends TextVertexWriter {

        @Override
        public void writeVertex(Vertex<Text, LongWritable, LongWritable> vertex) throws IOException, InterruptedException {
            String outputLine = vertex.getId().toString() + " " + vertex.getValue().get();
            getRecordWriter().write(key: null, new Text(outputLine));
        }
    }
}
```

Слика 27 - Проналажење најкраћег пута до главног града – дефиниција излазног формата података

3.6 Пример – препорука филмова

У следећем примеру представљена је комплекснија *Apache Giraph* апликација којом се у графу корисника и филмова проналазе препоруке филмова за све кориснике у систему на основу оцена које су корисници који су гледали исте филмове давали осталим филмовима.

Корисници и филмови представљају два типа чворова графа у овом проблему. Корисници су повезани са филмовима које су гледали и то чини један тип гране у графу који такође садржи и оцену коју је корисник дао филму. Други тип гране повезује филмове са корисницима који су их гледали.

Улазни подаци (слика 28) налазе се у текстуалном фајлу где је сваки чвор представљен као један ред. За чворове који представљају кориснике су редом написани идентификатор чвора (идентификатор корисника) и низ парова где је први елемент сваког пара идентификатор филма ког је корисник гледао, а други елемент пара је оцена коју је корисник дао том филму. За чворове који представљају филмове су редом написани идентификатор чвора (идентификатор филма) и низ идентификатора корисника који су гледали тај филм. Улазни формат података описан је и програмски (слика 29).

```
U1 M1 5 M2 3 M5 4
U2 M2 4 M3 5 M6 3
U3 M1 4 M4 2 M7 5
U4 M5 3 M8 4 M9 5
U5 M2 5 M6 4 M10 3
U6 M3 5 M7 4 M11 2
U7 M1 3 M8 5 M12 4
U8 M4 5 M9 3 M13 4
U9 M5 4 M10 3 M14 5
U10 M6 2 M11 5 M15 4
U11 M7 5 M12 3 M16 4
U12 M8 4 M13 5 M17 3
U13 M9 3 M14 4 M18 5
U14 M10 4 M15 5 M19 3
U15 M11 5 M16 3 M20 4
U16 M12 3 M17 5 M1 4
U17 M13 4 M18 3 M2 5
U18 M14 5 M19 4 M3 3
U19 M15 4 M20 5 M4 3
U20 M16 3 M1 4 M5 5
M1 U1 U3 U7 U16 U20
M2 U1 U2 U5 U17
M3 U2 U6 U18
M4 U3 U8 U19
M5 U1 U4 U9 U20
M6 U2 U5 U10
M7 U3 U6 U11
M8 U4 U7 U12
M9 U4 U8 U13
M10 U5 U9 U14
M11 U6 U10 U15
M12 U7 U11 U16
M13 U8 U12 U17
M14 U9 U13 U18
M15 U10 U14 U19
M16 U11 U15 U20
M17 U12 U16
M18 U13 U17
M19 U14 U18
M20 U15 U19
```

Слика 28 - Препорука филмова – улазни подаци

```java
package mr;

import org.apache.giraph.io.formats.TextVertexInputFormat;
import org.apache.giraph.edge.Edge;
import org.apache.giraph.edge.EdgeFactory;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class MovieRecommendationVertexInputFormat extends TextVertexInputFormat<Text, Text, LongWritable> {

    @Override
    public TextVertexReader createVertexReader(InputSplit split, TaskAttemptContext context) {
        return new MovieRecommendationVertexReader();
    }

    public class MovieRecommendationVertexReader extends TextVertexReaderFromEachLineProcessed<String[]> {

        @Override
        protected String[] preprocessLine(Text line) throws IOException {
            return line.toString().split("\\s+");
        }

        @Override
        protected Text getId(String[] tokens) throws IOException {
            return new Text(tokens[0]);
        }

        @Override
        protected Text getValue(String[] tokens) throws IOException {
            return new Text(string: "");
        }

        @Override
        protected Iterable<Edge<Text, LongWritable>> getEdges(String[] tokens) throws IOException {
            List<Edge<Text, LongWritable>> edges = new ArrayList<>();

            if (tokens[0].startsWith("U")) {
                for (int i = 1; i < tokens.length; i += 2) {
                    String movie = tokens[i];
                    long rating = Long.parseLong(tokens[i + 1]);
                    edges.add(EdgeFactory.create(new Text(movie), new LongWritable(rating)));
                }
            } else if (tokens[0].startsWith("M")) {
                for (int i = 1; i < tokens.length; i++) {
                    edges.add(EdgeFactory.create(new Text(tokens[i]), new LongWritable(value: 0)));
                }
            }

            return edges;
        }
    }
}
```

Слика 29 - Препорука филмова – дефиниција улазног формата података

У току алгоритма (слика 30) издвајају се тачно 4 *superstep*-а. Ово значи да је број *superstep*-ова у овом примеру познат, за разлику од претходних примера где је број *superstep*-ова зависио од улазних података. Четврти *superstep* служи за завршетак алгоритма, док претходна *superstep*-ови извршавају јединствене кораке у оквиру алгоритма.

```java
package mr;

import org.apache.giraph.graph.BasicComputation;
import org.apache.giraph.graph.Vertex;
import org.apache.giraph.edge.Edge;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

import java.io.IOException;
import java.util.*;

public class MovieRecommendationComputation extends BasicComputation<Text, Text, LongWritable, Text> {

    @Override
    public void compute(Vertex<Text, Text, LongWritable> vertex, Iterable<Text> messages) throws IOException {
        if (getSuperstep() == 0) {
            sendUserRatings(vertex);
        } else if (getSuperstep() == 1) {
            aggregateMovieInfo(vertex, messages);
        } else if (getSuperstep() == 2) {
            computeUserRecommendations(vertex, messages);
        }

        vertex.voteToHalt();
    }
```

Слика 30 - Препорука филмова – алгоритам

У *superstep*-у 0 (слика 31) учествују само чворови корисника. Они свим филмовима које су гледали шаљу поруке у којима су информације о свим филмовима које су гледали и о оценама које су дали тим филмовима. На тај начин филмови добијају информације о корисницима који су их гледали.

```java
    private void sendUserRatings(Vertex<Text, Text, LongWritable> vertex) {
        if (!vertex.getId().toString().startsWith("U")) {
            return;
        }

        for (Edge<Text, LongWritable> fromEdge : vertex.getEdges()) {
            String message = vertex.getId() + "," + fromEdge.getTargetVertexId() + "," + fromEdge.getValue().get();
            sendMessageToAllEdges(vertex, new Text(message));
        }
    }
```

Слика 31 - Препорука филмова – *superstep* 0

У *superstep*-у 1 (слика 32) учествују само чворови филмова. На основу порука које су добили од корисника, филмови агрегирају оцене тако што за сваки филм о ком су добили информацију бележе збир оцена, као и број оцена. Ове информације се затим шаљу назад ка корисницима који су гледали филм из тренутног чвора. На тај начин корисници добијају информације о филмовима које су гледали корисници који су гледали исти филм као и тренутни корисник.

```java
private void aggregateMovieInfo(Vertex<Text, Text, LongWritable> vertex, Iterable<Text> messages) {
    if (!vertex.getId().toString().startsWith("M")) {
        return;
    }

    Map<String, int[]> ratings = new HashMap<>();

    for (Text message : messages) {
        String[] parts = message.toString().split(",");
        String userId = parts[0];
        String movieId = parts[1];
        int rating = Integer.parseInt(parts[2]);

        ratings.putIfAbsent(movieId, new int[]{0, 0});
        ratings.get(movieId)[0] += rating;
        ratings.get(movieId)[1] += 1;
    }

    for (Map.Entry<String, int[]> rating : ratings.entrySet()) {
        String movieId = rating.getKey();
        int sum = rating.getValue()[0];
        int count = rating.getValue()[1];
        sendMessageToAllEdges(vertex, new Text(movieId + "," + sum + "," + count));
    }
}
```

Слика 32 - Препорука филмова – *superstep* 1

У *superstep*-у 2 (слика 33) учествују само чворови корисника. На основу порука које су добили од филмова, корисници агрегирају оцене тако што за сваки филм о ком су добили информацију бележе збир оцена, као и број оцена, при чему се води рачуна да се избаце филмови које је корисник већ погледао и оценио. Након агрегације рачунају се просечне оцене филмова тако што се подели збир оцена са бројем оцена. Након сортирања филмова на основу израчунатих просечних оцена, бирају се највише 3 најбоље оцењена филма и стављају се у листу препоручених филмова за тог корисника. Ово представља крај алгоритма.

У *superstep*-у 3 се не дешава ништа осим што сви чворови гласају за крај извршавања алгоритма.

```java
    private void computeUserRecommendations(Vertex<Text, Text, LongWritable> vertex, Iterable<Text> messages) {
        if (!vertex.getId().toString().startsWith("U")) {
            return;
        }

        Set<String> seen = new HashSet<>();

        for (Edge<Text, LongWritable> edge : vertex.getEdges()) {
            seen.add(edge.getTargetVertexId().toString());
        }

        Map<String, int[]> ratings = new HashMap<>();

        for (Text message : messages) {
            String[] parts = message.toString().split(",");
            String movieId = parts[0];
            int sum = Integer.parseInt(parts[1]);
            int count = Integer.parseInt(parts[2]);

            ratings.putIfAbsent(movieId, new int[]{0, 0});
            ratings.get(movieId)[0] += sum;
            ratings.get(movieId)[1] += count;
        }

        Map<String, Double> averageRatings = new HashMap<>();

        for (Map.Entry<String, int[]> rating : ratings.entrySet()) {
            if (!seen.contains(rating.getKey())) {
                averageRatings.put(rating.getKey(), (double) rating.getValue()[0] / rating.getValue()[1]);
            }
        }

        List<Map.Entry<String, Double>> sortedMovies = new ArrayList<>(averageRatings.entrySet());
        sortedMovies.sort((a, b) -> Double.compare(b.getValue(), a.getValue()));

        List<String> top3 = new ArrayList<>();

        for (int i = 0; i < Math.min(3, sortedMovies.size()); i++) {
            top3.add(sortedMovies.get(i).getKey());
        }

        vertex.setValue(new Text(String.join(" ", top3)));
    }
}
```

Слика 33 - Препорука филмова – *superstep* 2

Излазни подаци (слика 34) уписују се у текстуални фајл где је сваки чвор корисника представљен као један ред у ком су редом написани идентификатор чвора (идентификатор корисника) и низ идентификатора препоручених филмова. Излазни формат података описан је и програмски (слика 35).

```
U4 M18 M4 M14
U5 M14 M1 M3
U6 M14 M15 M1
U7 M7 M9 M13
U8 M2 M20 M7
U9 M15 M18 M1
U10 M3 M2 M20
U12 M2 M4 M9
U11 M17 M3 M5
U14 M14 M2 M20
U13 M2 M4 M19
U16 M7 M13 M8
U15 M3 M5 M7
U18 M15 M18 M2
U17 M1 M3 M4
U19 M7 M11 M1
U20 M14 M17 M7
U1 M3 M7 M9
U2 M14 M1 M15
U3 M17 M3 M20
```

Слика 34 - Препорука филмова – излазни подаци

```java
package sdtcc;

import org.apache.giraph.io.formats.TextVertexOutputFormat;
import org.apache.giraph.graph.Vertex;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

import java.io.IOException;

public class ShortestDistanceToCapitalCityVertexOutputFormat extends TextVertexOutputFormat<Text, LongWritable, LongWritable> {

    @Override
    public TextVertexWriter createVertexWriter(TaskAttemptContext context) throws IOException {
        return new ShortestDistanceToCapitalCityVertexWriter();
    }

    public class ShortestDistanceToCapitalCityVertexWriter extends TextVertexWriter {

        @Override
        public void writeVertex(Vertex<Text, LongWritable, LongWritable> vertex) throws IOException, InterruptedException {
            String outputLine = vertex.getId().toString() + " " + vertex.getValue().get();
            getRecordWriter().write(key: null, new Text(outputLine));
        }
    }
}
```

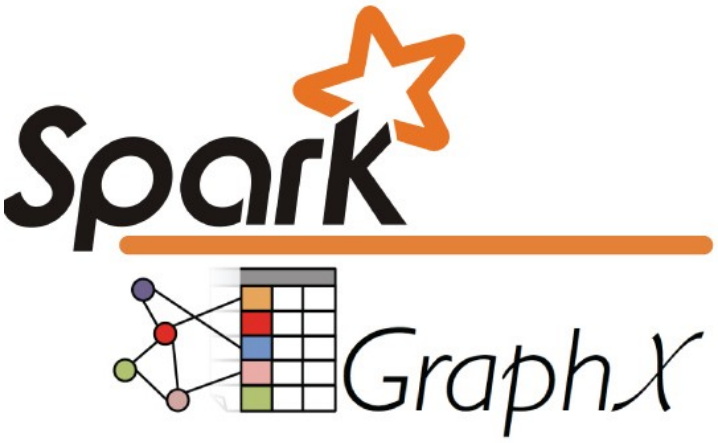
Слика 35 - Препорука филмова – дефиниција излазног формата података

4. ЗАКЉУЧАК

У раду је преглед обраде графова и њене важности у савременим системима, као и изазова који се јављају при раду са великим и сложеним графовима. Посебан фокус стављен је на *Apache Giraph* као представника *vertex-centric* модела обраде, кроз објашњење његовог програмског модела, архитектуре, начина рада и технологија које користи.

Практични део рада демонстрира примену *Apache Giraph*-а кроз више примера, укључујући проналажење највеће вредности, најкраћег пута између градова и систем препоруке филмова. Ови примери укључују начин покретања *Apache Giraph*-а на локалној машини и показују како се теоријски концепти Giraph-а могу применити на конкретне проблеме.

Apache Giraph је данас архивиран пројекат и више се активно не развија, што значи да се не препоручује за нове продукционе системе. Ипак, он остаје значајан са академског и едукативног аспекта. Његову улогу у пракси су у великој мери преузели новији алати и оквири, као што су *Apache Spark Graph* (слика 36) и *Apache Flink Gelly* (слика 37).



Слика 36 - *Apache Spark GraphX* лого

Слика 37 - *Apache Flink* лого

5. РЕФЕРЕНЦЕ

1. *Shaposhnik, R., Martella, C., & Logothetis, D. (2015). Practical Graph Analytics with Apache Giraph (1st ed.). Packt Publishing.*
2. *Apache Giraph. (n.d.). Apache Giraph. https://giraph.apache.org/*
3. *Apache Giraph. (n.d.). Apache Giraph (Attic). https://attic.apache.org/projects/giraph.html*
4. *Ching, A. (2013, August 14). Scaling Apache Giraph to a trillion edges. Facebook Engineering. https://engineering.fb.com/2013/08/14/core-infra/scaling-apache-giraph-to-a-trillion-edges/*
5. *Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In Proceedings of the 2010 International Conference on Management of Data (pp. 135–146). ACM. https://research.google/pubs/pregel-a-system-for-large-scale-graph-processing/*
6. Марко Николић *(n.d.). apache-giraph-graph-processing [GitHub* репозиторијум*]. https://github.com/marko-nikolic01/apache-giraph-graph-processing*