
Институт за математику и информатику
Природно-математички факултет
Универзитет у Крагујевцу

Мастер рад

# MPI-2 стандард и Lustre фајл систем

*Студент:*
Ненад Ацковић

*Професор:*
др Милош Ивановић

Септембар 2015.

# Садржај

# Скраћенице

| | |
|---|---|
| ACL | Access Control Lists. 9 |
| API | Application Programming Interface. 5 |
| HPC | High-Performance Computing. 4 |
| LNET | Lustre Networking. 1, 5 |
| LVM | Logical Volume Management. 13 |
| MDC | Metadata Client. 5 |
| MDS | Metadata Server. 5 |
| MDT | Metadata Target. 5 |
| MGC | Management Client. 5 |
| MPI | Message-Passing Interface. 23 |
| NAL | Network Abstraction Layers. 5 |
| NFS | Network File Sistem. 4 |
| NTP | Network Time Protocol. 10 |
| OSC | Object Storage Client. 5 |
| OSS | Object Storage Servers. 5 |
| OST | Object Storage Target. 5 |
| PDSH | Parallel Distributed SHell. 10 |
| PVM | Parallel Virtual Machine . 41 |
| RMA | Remote Memory Access. 37 |

SELinux Security-Enhanced Linux. 11

SNMP Simple Network Management Protocol. 15

# Глава 1

# *Увод*

Овај мастер рад као примарни циљ поставља проучавање могућности *Lustre* фајл система у унапређењу рада кластера високих перформанси, као и користи од паралелних улазно излазних операција дефинисаних MPI-2 стандардом. *Lustre* паралелни фајл систем има предност над осталим (секвенцијалним) системима услед могућности обављања улазно-излазних операција у паралелном маниру. Са друге стране, MPI-2 представља индустријски стандард за развој паралелних програма, а његове имплементације подржавају паралелно читање и писање на уређаје масовне меморије. У циљу сагледавања начина да се побољшају укупне перформансе локалног кластера *Medflow*, упоређују се брзине читања и писања података у фајлове који се налазе како на *Lustre*, тако и на NFS фајл систему.

Кластер високих перформанси (*High Performance Computing Cluster*) представља скуп рачунара умрежених коришћењем локалне мрежне инфраструктуре, помоћу које међусобно комуницирају. Коришћење специфичне програмске подршке даје висок степен интеграције рачунара, омогућава њихов координирани заједнички рад и претвара их ефективно у јединствен вишепроцесорски систем који користи дистрибуирану меморију.

За истраживачке радове који захтевају обимне математичке прорачуне данас се углавном користе паралелни програми који на кластеру високих перформанси рашчлањују проблем на више рачунарских процесора. Тиме се време добијања резултата знатно смањује у односу на време потребно за добијање резултата на једном процесору, понекад за више редова величине. Најефективнији начин је употреба поменутог MPI стандарда за развој паралелних програма. На Универзитету у Крагујевцу постоји неколико HPC кластера, али ни један од њих не поседује паралелни фајл систем. Та чињеница у појединим случајевима употребе може да доведе до озбиљног пада перформанси. На срећу, први кластер са паралелним фајл системом постављен је недавно у истраживачко развојном центру за биоинжењеринг у Крагујевцу. Након иницијалне инсталације HPC кластера, дошло се на идеју да се истраже реалне могућности и употребна вредност новоинсталираног *Lustre* система, што и представља основну мотивацију за израду овог мастер рада.

Рад је подељен у пет поглавља, а у наставку је дат кратак садржај. У првом делу рада се описују компоненте *Lustre* фајл система, као и тестна инсталација система на скупу виртуалних машина. Затим се објашњава специфично дељење фајлова у систему и улога појединих сервиса. Управо је дељење фајлова и паралелни приступ оно што разликује *Lustre* фајл систем од осталих. Најподобније је описана инсталација система услед значајне разлике у односу на инсталације класичних фајл система који су подржани директно од стране *Linux* кернела. У поглављу 2.7 су описани алати помоћу којих се најоптималније врше подешавања *Lustre* фајл система. У другом делу рада се описује MPI-2 стандард и његове могућности везане за паралелне улазно/излазне операције. Испитују се начини преко којих је могуће вршити паралелне улазно/излазне операције и њихове особености. У трећем делу рада су дата тестирања брзина извршавања операција фајл система помоћу програма *Iozone*, *dd* и *Game of Life*. Под претпоставком да постоји разлика у брзини ула-

зно/излазних операција ових фајл система, анализирају се добијени резултати тестирања и на основу њих се утврђује који је погоднији за који домен употребе.

# Глава 2

# *Lustre*

## 2.1 Увод

*Lustre* систем је бесплатан, дистрибуиран, паралелни фајл систем развијен од стране *Sun Microsystems Inc.* *Lustre* систем је дизајниран и имплементиран тако да смањи или потпуно неутралише уска грла претходних паралелних фајл система. Централна компонента *Lustre* система је дељени фајл систем за кластере. *Lustre* је тренутно доступан за Linux оперативне системе и омогућава **UNIX®** фајл систем окружење. *Lustre* архитектура се користи за више врста кластера. Користи се у седам од десет највећих кластера високих перфоманси у свету који имају хиљаде клијента, петабајте складишта и хиљаде гигабајта у секунди У/И операција. Скалабилност коју нуди *Lustre* учинила га је погодним алатом у оквиру истраживања нафте и гаса, у производњи као и у финансијском сектору. Са корисним побољшањима подршци у оквиру *Lustre* мреже (*LNET*) и софтверима који управљају *Lustre* фајл системом, коришћење *Lustre* фајл система би требало да буде још шире. Скалабилност *Lustre* система смањује потребу за одвојеним фајл системима, на пример креирање једног фајл система за један кластер или још неефикасније један фајл систем за сваки NFS фајл сервер. Ово доводи до предности управљања складиштем података, избегавајући вишеструке копије података на више фајл система. Главни HPC центри због тога захтевају много мање складишта података са *Lustre* фајл системом него са другим системима. Проток или капацитет може се лако подесити после инсталације кластера, уколико се дода нови сервер. Како је *Lustre* бесплатан систем, усвојен је од стране једног броја рачунарских компанија и интегрисан је њиховим понудама. У циљу једноставне инсталације *Lustre* фајл система, *Red Hat* и *Novell* (*SUSE*) нуде закрпе кернела својих дистрибуција. *Lustre* архитектура први пут је развијена 1999. године, а 2003. године је објављена верзија 1.0 и одмах је искоришћена на великом броју кластера високих перформанси. Коришћење *Lustre* система такође је утицало и на побољшање перформанси *Linux ext3* фајл система.

## 2.2 Компоненте *Lustre* фајл система

### 2.2.1 Основне компоненте

*Lustre* фајл систем се састоји од следећих основних компоненти(слика 2.1):

- **Management Server(MGS)** Чува информације о конфигурацијама за све *Lustre* системе у кластеру. Сваки *Lustre* клијент контактира MGS да пружи информације. MGS захтева сопствени диск за складиштење. Међутим, постоји одредба која омогућава дељење MGS диска („ко-лоцирање") са једним MDT-ом. MGS се не сматра „делом" индивидуалног фајл система, већ пружа информације о конфигурацији за *Lustre* компоненте.

- **Metadata Server (MDS)** MDS сервер чине метаподаци ускладиштени у један или више MDT сервера који су на располагању *Lustre* клијентима. Сваки MDS управља именима фајлова и директоријумима *Lustre* система и обезбеђује мрежни захтев за руковање једним или више локалних MDT-а.

- **Metadata Target (MDT)** MDT чува метаподатаке (као што су имена фајлова, директоријума, дозвола и распоред фајлова) на MDS-у. Сваки фајл систем има по један MDT. MDT на дељивом фајл систему може бити доступан многим MDS серверима, мада само један заправо треба да га користи. Уколико дође до грешке на MDT-у, MDS може аутоматски преузети његову улогу и ставити се на располагање клијентима. Ова могућност се назива MDS *failover*.

- **Object Storage Servers (OSS)** OSS обезбеђују улазно/излазне операције, као и захтеве за један или више локалних OST-а. Обично OSS опслужује између 2 и 8 OST-а, од којих један OST може имати до 16 TB складишног простора. MDT, OST и *Lustre* клијенти могу се покренути истовремено на једном чвору. Међутим, најзаступљенија конфигурација је да се MDT инсталира на одвојеном чвору, са једним или више OST-а на сваком OSS чвору.

- **Object Storage Target (OST)** OST чува податке (делове корисничних фајлова) на једном или више OSS-а. Један *Lustre* фајл систем може имати више OST-а, од којих сваки опслужује део фајла. Није нужно да се један фајл налази на једном OST-у. У циљу оптимизације перформанси, фајл може бити расподељен на много OST-а. Logical Object Volume (LOV) управља деловима датотека на вишеструким OST.

- **Lustre клијенти** Lustre клијенти су рачунари који по покретању *Lustre* програма омогућавају подизање партиције *Lustre* система. *Lustre* програм се састоји од окружења између *Linux Virtul File System-a* и *Lustre* сервера. Сваки клијент се састоји од: *Metadata Client (MDC)*, *Object Storage Client (OSC)* и *Management Client (MGC)*. Група OSC се налази у једном LOV. Радећи здружено, OSC омогућава транспарентан приступ фајл систему. Клијенти који подижу *Lustre* фајл систем виде један кохерентан и синхронизован фајл систем све време. Различити клијенти могу да уписују различите делове истог фајла истовремено, док остали клијенти могу да читају из фајла.

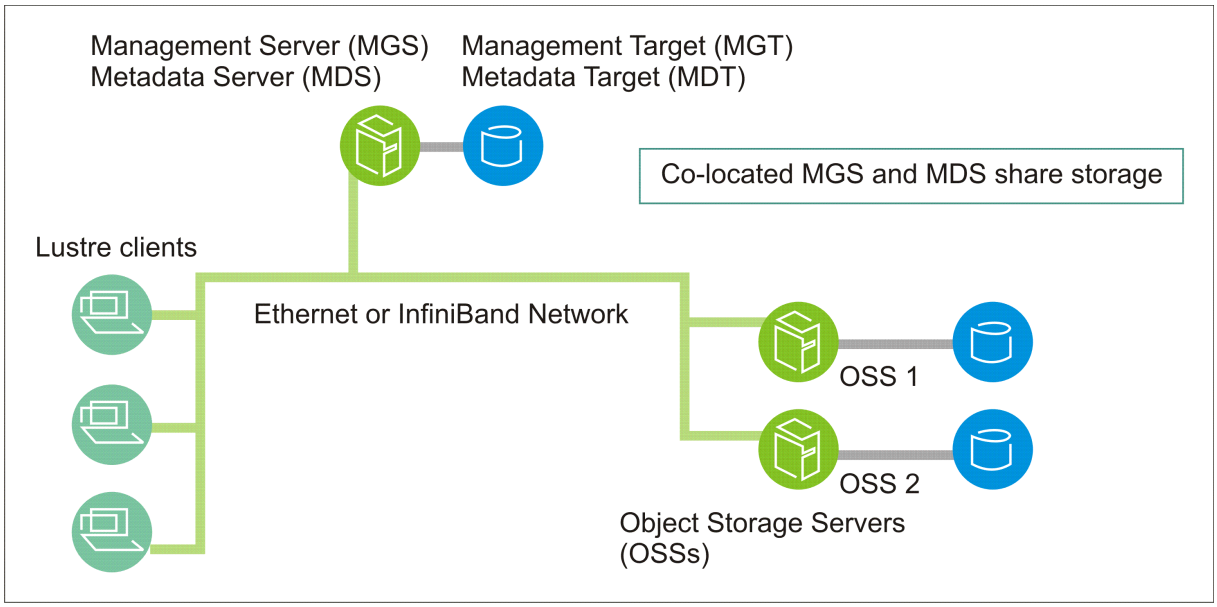
### 2.2.2 *Lustre* Умрежавање (LNET)

Lustre умрежавање (LNET) је *API* који управља мета подацима и улазно/излазним податацима за системске сервере и клијенте. LNET подржава више хетерогених окружења на клијентима и серверима. LNET комуницира са више врста мрежа преко *Network Abstraction Layers (NAL)*. Lustre систему су на располагању већи број мрежа, укључујући *Infiniband*, *TCP/IP*, *Qyadrics Elan*, *Myrinet (MX and GM)* и *Cray*. Код кластера са *Lustre* системом,

сервери и клијенти комуницирају са другом врстом мреже која се зове *Lustre Networking (LNET)*, док су складишта MDS and OSS повезана традиционалним SAN технологијама.

**Кључне карактеристике *Lustre* мреже**

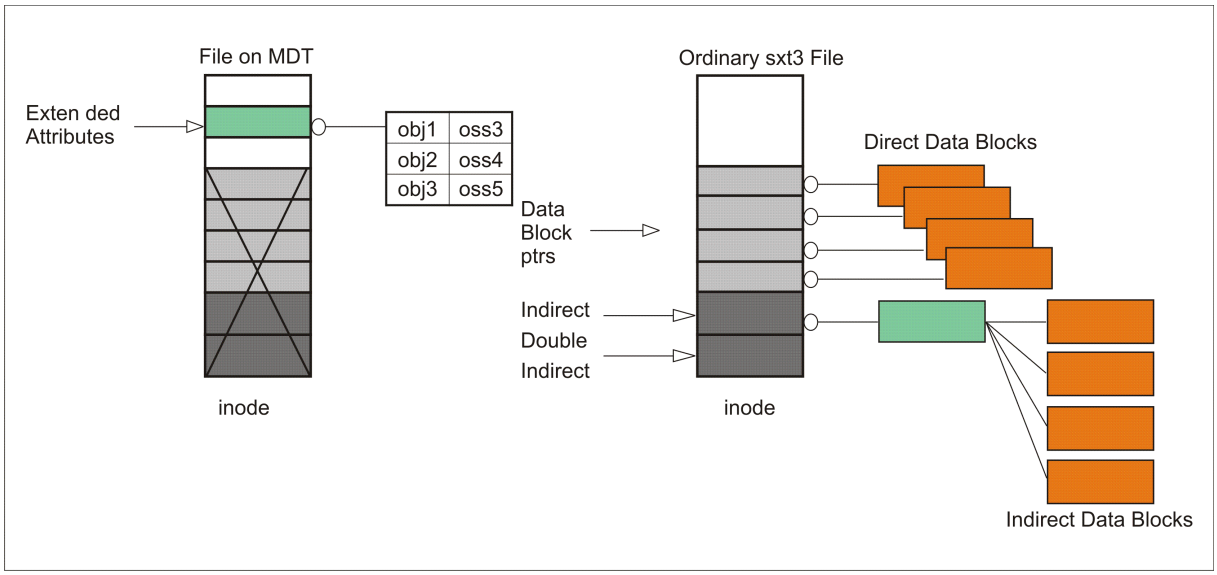
- RDMA - подршка мрежа као што су *Elan*, *Myrinet* и *InfiniBand*.
- Подршка за много чешће коришћене типове мрежа попут *InfiniBand* и *TCP/IP*.
- Својства висока доступности и опоравка омогућавају транспарентан опоравак сервера.
- Истовремена доступност више мрежних типова са рутирањем између њих.




Слика 2.1: *Lustre* компоненте

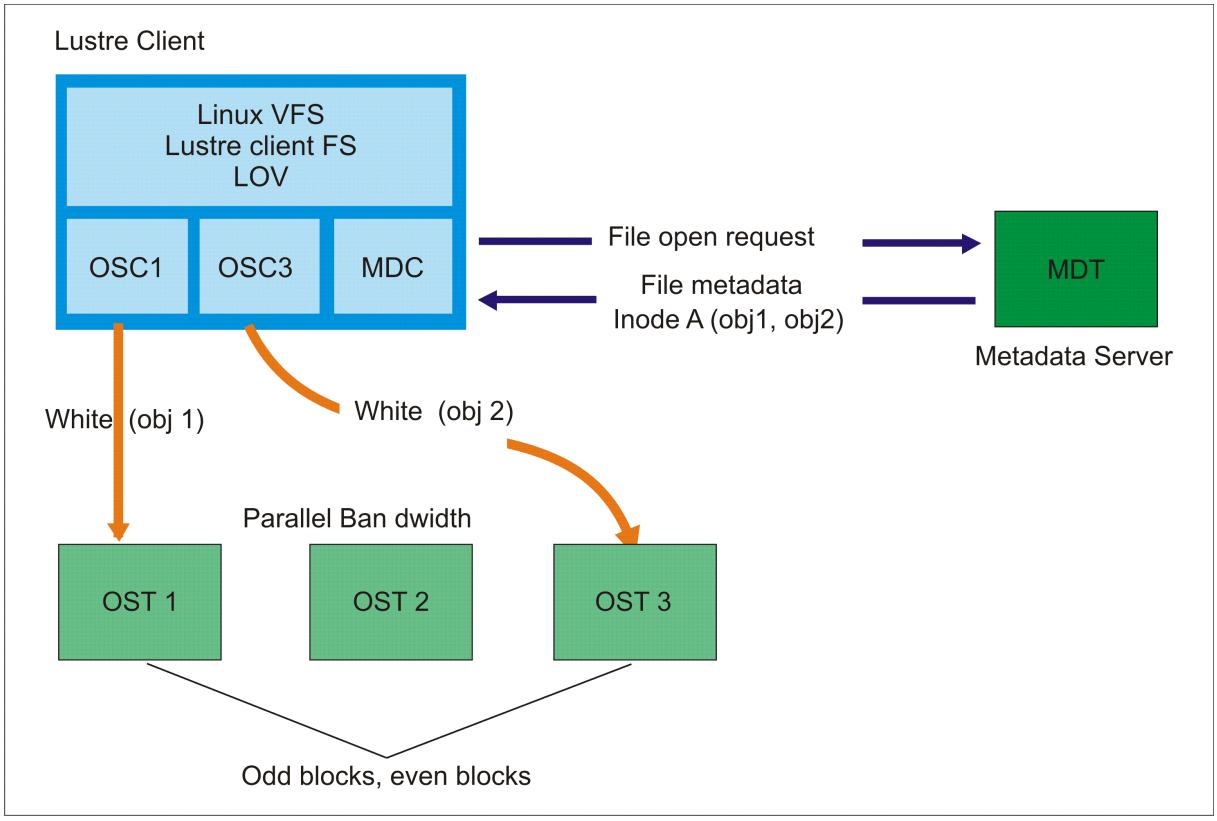
## 2.3 Фајлови у *Lustre* систему

Традиционални UNIX фајл системи користе чворове, који садрже спискове редних бројева блокова где се чувају подаци о фајлу за дати чвор. Слично томе, за сваки фајл у *Lustre* систему фајлова, један чвор постоји на MDT-у. Међутим, у *Lustre* систему чвор на MDT није показивач на блок података, већ показује на један или више објеката коју су у вези са фајловима(слика 2.2). Ови објекти су датотеке на OST и садрже податке.




Слика 2.2: Разлика између MDS и ext3 чворова

Уколико је само један објекат повезан са MDS чвор, тај објекат садржи све податке у том систему. Када је више од једног објекта повезано, подаци у фајлу су подељени широм објекта. MDS зна распоред сваког фајла, број и локацију дела фајла. Клијенти добијају изглед фајла из MDS. Када клијент изврши У/И операцију на делу фајла, фајл комуницира директно са релевантним OST-ом(слика 2.3).

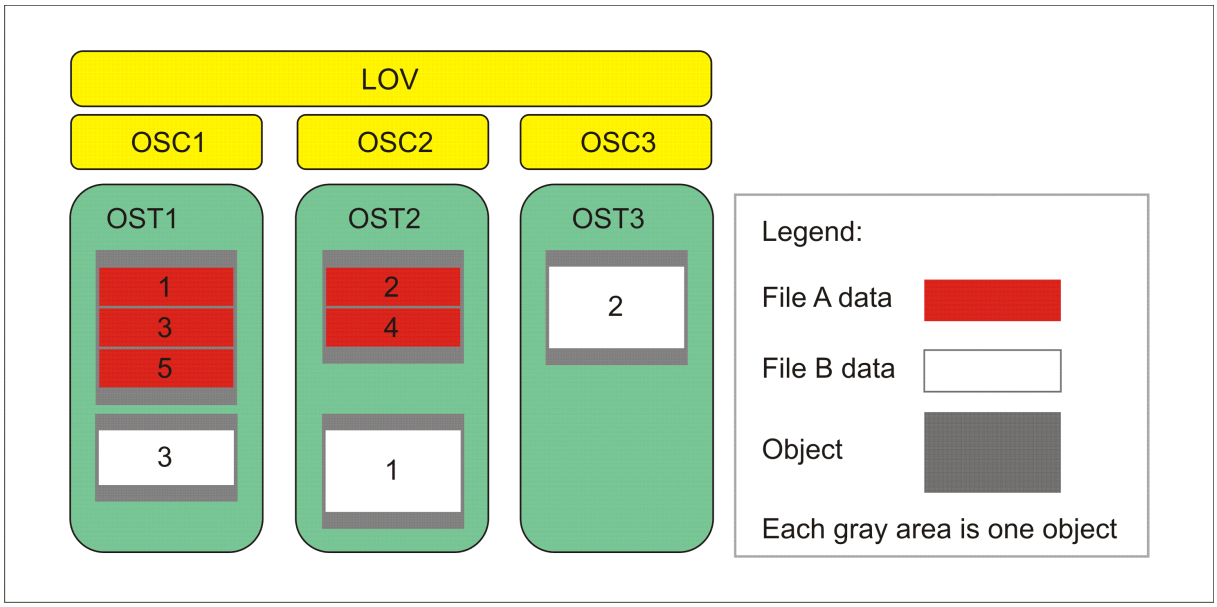



Слика 2.3: Lustre улазно/излазне операције

### 2.3.1 Дељење фајлова у *Lustre* систему

Дељење фајлова омогућава да се делови фајлова чувају на различитим OST-има(слика 2.4). У RAID 0 нивоу подаци су подељени на већем броју објеката.

Број објеката се назива *stripe_ count*. Сваки објекат садржи део података. Када део података који треба да се упише на одређени објекат прелази *stripe_ count*, следећи део података у датотеци се чува на следећем објекту. Дељење фајлова има неколико предности. Једна је да максимална величина датотеке није ограничена величином једног OST. *Lustre* може имати преко 160 подељених делова, и сваки део може да подржи максималну величину од 8TB. То доводи до максималне количине фајла од чак 1,48 PB. Још једна корист од дељења фајлова је та да је улазно/излазни пропусни опсег у једном фајлу збир улазно/излазних пропусних опсега за објекте од којих се фајл састоји. То у крајњем случају може бити проближно збиру пропусних опсега 160 сервера.




Слика 2.4: Дељење фајлова

### 2.3.2 *Lustre* складиштење

Складиштење у серверима је подељено, опционо организовано помоћу *Logical volume management (LVM)* система и форматирано као фајл систем. Lustre OSS и MDS читају, пишу и мењају податке у формату који захтевају ови фајл системи.

#### OSS складиштење

Сваки OSS може управљати са више циљева за складиштење објеката (OST-има), по један за сваки *volume*; Улазно/излазни саобраћај је избалансиран између OSS-а и OST-а. OSS такође треба да уравнотежи пропусни опсег мреже између система мреже и складиштења и да спречи појаву уских грла. У зависности од карактеристика хардвера сервера, OSS обично служи између 2 и 25 OST-а, при чему је капацитет сваког од њих до 8 TB.

#### MDS складиштење

За MDS, складиштење мора бити везано за *Lustre* метаподатке, за које је потребно 1-2 % капацитета фајл система. Приступ подацима за MDS складиштење се разликује од приступа подацима за OSS складиштења.

*Metadata* приступ подразумева рад са више захтева и читање-писање мале количине података, док улазно/излазни приступ подразумева пренос великих количина података. Велики проток података за MDS није превише битан, па се зато препоручује да се користе врсте складишта попут FC или SAS дискова, који обезбеђују малу потражњу података. Са ниским нивоима улазно/излазних операција, RAID 5/6 није оптималан, док RAID 0+1 даје много боље резултате. *Lustre* користи и „journaling" фајл система. За MDS се понекад могу добити и до 20 % бољи резултати уколико се „journaling" фајл систем постави на различите дискове. Поред тога, MDS захтева велику снагу процесора и препоручује се барем четири процесорска језгра за оптималне перформансе.

## 2.4 Кључне карактеристике

- **Скалабилност** - *Lustre* перформансе зависе од броја клијентских чворова, складишта и пропусног опсега. Тренутно највећа инсталација *Lustre* система покренута у продукцији ради са 26000 клијента, на кластерима који имају између 10000-20000 клијената. Неколико *Lustre* система има капацитет од 1 PB или више, омогућавајући складиште за 2 милијарде фајлова.

- **Перформансе** - *Lustre* у продукцији има проток од око 100 GB/s. У тест окружењима перформансе су око 130 GB/s и 13,000 creates/s. *Lustre* клијент има проток око 2 GB/s и OSS проток од 2.5 GB/s (максимално). Такође, постоје подаци да је покренут на 240 GB/s на *Spider* фајл систему у *Oak Ridge National Laboratories*.

- ***POSIX* сагласност** - Потпуни *POSIX* стандарди су испуњени на *Lustre* клијентима. Код кластера, *POSIX* сагласност значи да су све операције једноставне и да клијенти увек приступају свежим подацима.

- **Висока доступност** - *Lustre* нуди дељено складиште OSS (за OST) и дељено складисте MDS (за MDT).

- **Сигурност** - У *Lustre* систему, TCP конекција се одвија само преко привилегованих портова. Припадност групама се одређује на серверу. POSIX листе за контролу приступа (*Access control lists - ACL*) су такође подржане.

- **Бесплатан** - *Lustre* је лиценциран под GNU GPL.

- **Интероперативност** - *Lustre* је покренут на више врста процесорских архитектура и на више верзија *Lustre* система истовремено.

- **Листе за контролу приступа** - Тренутно, *Lustre* безбедносни модел дозвољава UNIX фајл систем побољшан са *POSIX ACLs (Access control list)*. Додатне функције укључују *root squash* и конекцију само са привилегованих потрова.

- **Квоте** - Корисничке и групне квоте су доступне.

- **OSS додатак** - Капацитет фајл система и проток кластера може бити повећан додавањем новог OSS са OST, и то без икаквих прекида услуге.

- **Контролисано дељење фајлова** - Подразумевани *stripe count* и *stripe size* може бити контролисан на неколико начина. Фајл систем добија стандардна подешавања приликом инсталације. Такође, директоријумима може бити додат атрибут који ће означавати начин дељења. Велики број библиотека и програма омогућавају једноставно контролисање дељења индивидуалних фајлова у *Lustre* системима.

- **Тренутни снимак - *Snapshot*** - *Lustre* фајл сервери користе *volumes* који се налазе на серверским чворовима. У пакету *Lustre* програма се налази и програм који омогућава креирање снимака свих *volumes*.

- **Алати за прављење резервних копија** - *Lustre* подржава 2 услужна алата: Један алат скенира фајл систем и проналази измењене фајлове у датом временском периоду. Овај алат прави списак путања до измењених фајлова, а затим се ти фајлови обрађују паралелно користећи други алат. Веома користан алат је и измењена верзија GNU tar (*gtar*) која прави резервне копије и врши повратак проширених атрибута (за дељење фајла и дозволама за приступ).

## 2.5 Инсталација

### 2.5.1 Потребни *Lustre* пакети

За инсталацију *Lustre* фајл система потребни су следећи пакети:

- ***Linux* кернел** закрпљен специфичним *Lustre* закрпама (потребан само за MDS и OSS)

- ***Lustre* модули** компајлирани за Linux kernel

- ***Lustre* кориснички програми** потребни за конфигурацију

- ***Lustre* алати (*e2fsck* and *lfsck*)** који се користе за опоравак фајл система, доступни у пакету под називом *e2fsprogs*

- **(Опционо) Мрежни кернел модули и библиотеке** (на пример, кернел модули и библиотеке потребни за *InfiniBand* мрежу)

- ***e2fsprogs***: *Lustre* захтева сопствену верзију *e2fsprogs*. *e2fsprogs* мора се инсталирати само на чворовима које подижу *ldiskfs* фајл системе, као што су OSS, MDS и MGS чворови. Није потребно инсталирати их на клијентима.

- ***Perl*** - Разни кориснички програми за *Lustre* су писани у *Perl-y*.

### 2.5.2 Захтеви окружења

- Сви чворови *Lustre* фајл система треба да да имају *remote shell* приступ. Иако није стриктно потребно за покретање система, препоручује се да сви чворови имају овакав приступ, због олакшавања конфигурације *Lustre-a* и скрипти за праћење рада система. *Parallel Distributed SHell (PDSH)* се препоручује, али и *Secure SHell (SSH)* је прихватљив.

- Обезбедити синхронизацију сатова. Lustre користи сатове за timestamps. Уколико сатови нису синхронизовани, доћиће до проблема код праћења рада система. Биће отежано отклањање грешака и корелисање дневника. Препоручује се *Network Time Protocol (NTP)*.

- Користити јединствени приступ фајловима на свим кластер чворовима. Користити исти користички ID (UID) и групни ID (GID).

- Искључити *Security-Enhanced Linux (SELinux)* на серверима и клијентима. *Lustre* не подржава SELinux. Зато је потребно искључити *SELinux* на свим *Lustre* чворовима, као и остале безбедносне екстензије, на пример *Novell AppArmor* и *network packet filtering tools (iptables)*.

### 2.5.3 Захтеви у вези са меморијом

#### Захтеви у вези са меморијом за клијенте

Препоручљиво је да клијенти имају најмање 2 GB RAM.

#### Захтеви у вези меморије за MDS

Захтеви меморије за MDS зависе од следећих фактора:

- Броја клијента
- Величине директоријума
- Обима оптерећења

Количина меморије за MDS је функција броја клијената на систему и броја фајлова које користе приликом покретања операција. То је пре свега, број закључавања клијента у једном временском тренутку. Стандардни максимум броја закључавања по чвору је 100*(број језгара), а интерактивни клијенти могу да држе више од 10000 закључавања у тренутку. За MDS, ово значи приближно 2 KB по фајлу. Стандардно је 400 MB за фајл систем дневника и додатна употреба меморије за кеширање фајлова за велике радне скупове који се тренутно не користе од стране клијената. Имати постојање великих података у кешу може побољшати metadata перформансе 10x или и више ако се упоређује са читањем са диска. Приближно 1.5 KB за фајл је потребно за чување фајла у кешу.

На пример, за MDT на MDS са 1000 клијената, 16 интерактивних чворовима и са 2 милиона радних скупова (од којих су 400000 у кешу), потребно је 4GB меморије(Листинг 2.1)

**Листинг 2.1: Количина меморије за MDT**

```
File system journal = 400 MB
1000 * 4-core clients * 100 files/core * 2kB = 800 MB

16 interactive clients * 10,000 files * 2kB = 320 MB

1,600,000 file extra working set * 1.5kB/file = 2400 MB
```

Повећање меморије аутоматски значи и боље перформансе.

Ако постоје директоријуми који садрже милион или више фајлова, можда ће бити потребно значајно више меморије. На пример, у окружењу где клијенти насумично приступају једном од 10 милиона фајлова, постоји и додатна меморија за кеш.

#### Захтеви у вези са меморијом за OSS

Приликом планирања хардвера за OSS чвор, треба размотрити употребу меморије у *Lustre* систему (нпр. дневник, сервисне нити, фајл систем метаподатака, итд.). Такође, треба размислити колико је битно да OSS кешира податке.

- **Величина дневника** - Стандардно, сваки *Lustre ldiskfs* фајл систем има 400 MB за дневник. Ово може бити једнако количини меморије на OSS чвору по једном фајл систему.
- **Сервисне нити** - Нити на OSS чвору алоцирају око 1 MB за улазно/излазни бафер за сваку OST сервисну нит, тако да ове бафере није потребно алоцирати за сваки улазно/излазни захтев.

- **Метаподаци фајл система** - Разумна количина меморије би требало да буде доступна за фајл систем метаподатака. Уколико је меморија доступна, онда се улазно/излазне операције на диску одвијају брже.
- **Мрежни транспорт** - Уколико се користи TCP или неки други протокол, треба да се има у виду меморија за бафере за слање/пријем.
- **Конфигурација у случају отказа** - Ако се OSS чвор користи за *failover* са другог чвора, онда RAM меморија за сваки дневник треба бити дуплирана тако да сервер за резервне копије може да реши проблем уколико матични сервер откаже.
- **OSS читање кеша** - омогућава само читање кешираних података на OSS, користећи регуларану *Linux* кеш страну за чување података. Исто као кеширање са регуларног фајл система на *Linux* оперативном систему, OSS чита кеш користећи што више физичке меморије.

#### Израчунавање количине меморије за OSS

Минимална препоручена количина RAM меморије OSS-а са 2 OST-а је 4 GB(Листинг 2.2).

**Листинг 2.2: Препоручена количина RAM меморије**

```
1.5 MB per OST IO thread * 512 threads = 768 MB

e1000 RX descriptors, RxDescriptors=4096 for 9000 byte MTU = 128 MB

Operating system overhead = 512 MB

400 MB journal size * 2 OST devices = 800 MB

600 MB file system metadata cache * 2 OSTs = 1200 MB
```

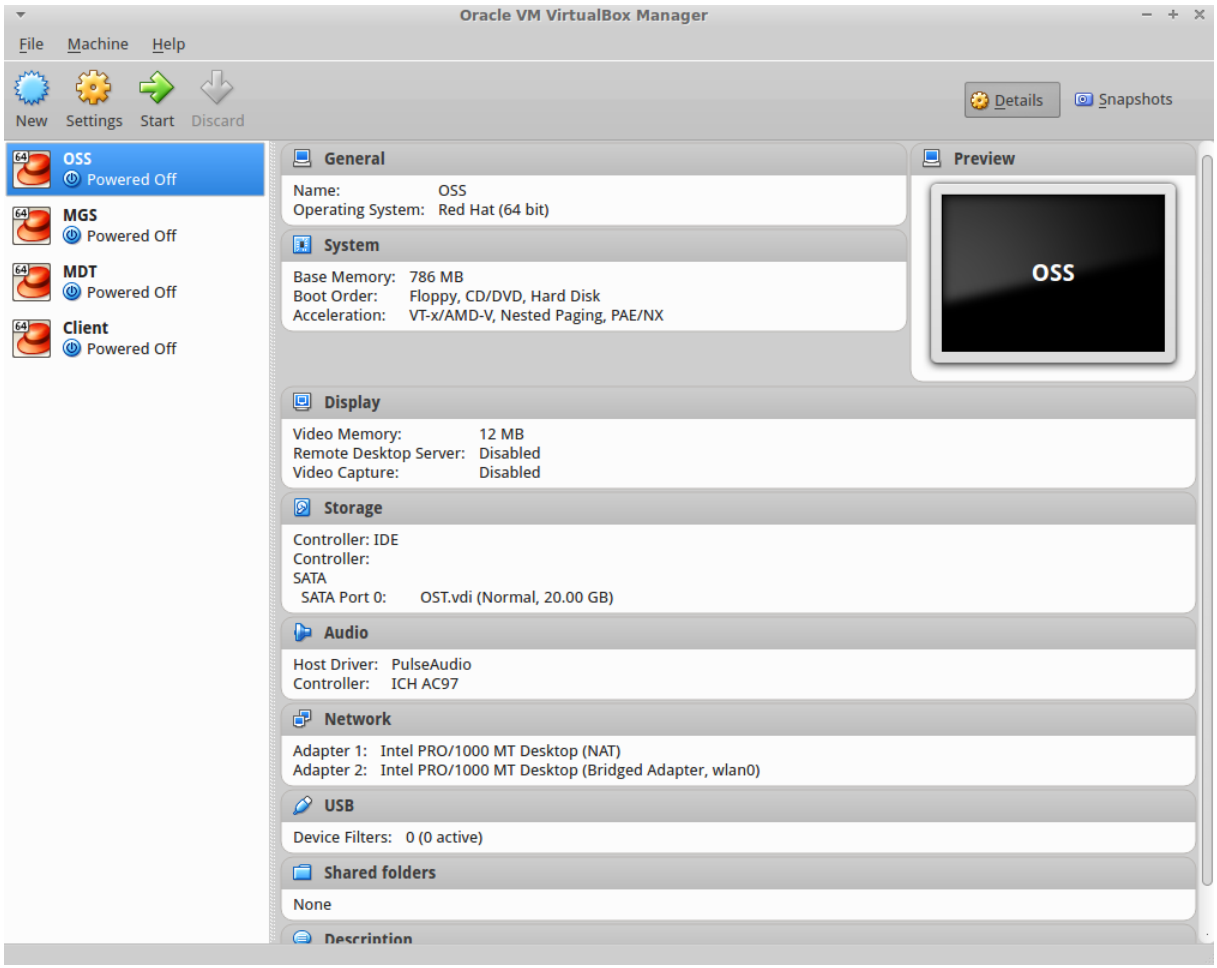
1700 MB је за алокацију бафера и додатна 2GB за минималан фајл систем и кернел. Значи за стандардну конфигурацију минимум меморије је 4 GB за сваки OSS чвор са 2 OST. Иако то није стриктно речено додавањем више меморије на OSS долази до побољшања перформансе читања мањих често посећиваних фајлова. За *failover* конфигурацију, минимална количина RAM меморије је 6 GB. За 4 OST-а на сваком OSS-у, у *failover* конфигурацији потребно је 10GB.

## 2.6 Инсталација *Lustre* фајл система

### 2.6.1 Инсталација оперативног система

Да би се инсталирао *Lustre* фајл систем, потребно је имати најмање 3, а пожељно је имати 4 чвора.

За ову инсталацију користи се *Xubuntu* 14.04 оперативни систем на којем је инсталиран *VirtualBox* 4.3.10, помоћу којег се креирају 4 виртуалне машине(слика 2.5). Свакој машини је потребно доделити 768MB RAM меморије и 1 процесор. Да би машина имала приступ интернету, потребно је у подешавањима, у секцији *Network* поставити први адаптер на NAT, а други на *Bridged Adapter*. *Bridged Adapter* је потребан да би машине комуницирале у локалној мрежи.




Слика 2.5: Виртуалне машине

На свим чворовима биће инсталиран *Scientific Linux 6.5* и сваком чвору бити додељена по једна статичка IP адреса(Табела 2.1).

Табела 2.1: Тестна конфигурација *Lustre* чворова

| **Оперативни систем** | **Назив** | **IP** | **Функција** |
|---|---|---|---|
| Scientific Linux 6.5 | mgs | 192.168.1.200 | Managment Server |
| Scientific Linux 6.5 | mdt | 192.168.1.201 | Metadata Server |
| Scientific Linux 6.5 | oss | 192.168.1.202 | OS Server |
| Scientific Linux 6.5 | client | 192.168.1.203 | Client |

На MGS-у, MDT-у и OSS-у чврсти диск је подељен на:

- *Boot* партиција – 10 GB (*root*, *home* директоријум)
- *Swap* партиција – 2 GB
- Logical volume partition – 8 GB ( LVM партиција за *Lustre* фајл систем)

Након инсталације оиперативног система, потребно је урадити следеће кораке:

1. Обезбедити SSH приступ између свих машина. SSH daemon се стартује командом

```
/etc/init.d/sshd start
```

док следећом командом подешавамо да се SSH сервис покреће приликом подизања оперативног система

```
chkconfig sshd on
```

2. Инсталација потребних програма

   Да бисмо касније компајлирали кернел, потребни су пакети попут *gcc, make* ... Њих инсталирамо командом:

   ```
   yum -y groupinstall "Development Tools"
   ```

3. Додавање IP адреса у */etc/hosts* фајл. На свим чворовима додати следеће линије у */etc/hosts* фајл:

   ```
   192.168.1.200   mgs
   192.168.1.201   mdt
   192.168.1.202   ost
   192.168.1.203   client
   ```

4. Искључити *Linux Firewall*

   ```
   chkconfig iptables off
   ```

5. Потребно је скинути *Lustre* пакете(Листинг 2.3).

**Листинг 2.3: *Lustre* пакети**

```
cd  /home/mgs/Downloads/

wget http://downloads.whamcloud.com/public/lustre/
   latest-maintenance-release/el6/server/RPMS/x86_64/
kernel-2.6.32-358.23.2.el6_lustre.x86_64.rpm

wget http://downloads.whamcloud.com/public/lustre/
   latest-maintenance-release/el6/server/RPMS/x86_64/
kernel-firmware-2.6.32-358.23.2.el6_lustre.x86_64.rpm

wget http://downloads.whamcloud.com/public/lustre/
   latest-maintenance-release/el6/server/RPMS/x86_64/
lustre-2.4.2-2.6.32_358.23.2.el6_lustre.x86_64.x86_64.rpm

wget http://downloads.whamcloud.com/public/lustre/
   latest-maintenance-release/el6/server/RPMS/x86_64/
lustre-ldiskfs-4.1.0-2.6.32_358.23.2.el6_lustre.x86_64.x86_64.
   rpm

wget http://downloads.whamcloud.com/public/lustre/
   latest-maintenance-release/el6/server/RPMS/x86_64/
lustre-modules-2.4.2-2.6.32_358.23.2.el6_lustre.x86_64.x86_64.
   rpm

wget http://downloads.whamcloud.com/public/lustre/
   latest-maintenance-release/el6/server/RPMS/x86_64/
lustre-osd-ldiskfs-2.4.2-2.6.32_358.23.2.el6_lustre.x86_64.
   x86_64.rpm

wget http://downloads.whamcloud.com/public/e2fsprogs/latest/el6/
   RPMS/i686/e2fsprogs-1.42.7.wc2-7.el6.i686.rpm
```

```
wget http://downloads.whamcloud.com/public/e2fsprogs/latest/el6/
    RPMS/i686/e2fsprogs-libs-1.42.7.wc2-7.el6.i686.rpm

wget http://downloads.whamcloud.com/public/e2fsprogs/latest/el6/
    RPMS/i686/libss-1.42.7.wc2-7.el6.i686.rpm

wget http://downloads.whamcloud.com/public/e2fsprogs/latest/el6/
    RPMS/i686/libcom_err-1.42.7.wc2-7.el6.i686.rpm
```

6. Ископирати пакете на OSS i MDT помоћу команде `scp`:

```
scp -r /home/mgs/Downloads/ ost:/home/ost/
scp -r /home/mgs/Downloads/ mdt:/home/mdt/
```

7. Креирати фајл */etc/modprobe.d/lustre.conf* и подесити мрежни протокол и мрежно окружење додавањем следеће линије:

```
options lnet networks=tcp0(eth1)
```

8. Такође, ископирати *lustre.conf* на OSS i MDT:

```
scp  /etc/modprobe.d/lustre.conf mdt:/etc/modprobe.d/lustre.conf
scp  /etc/modprobe.d/lustre.conf ost:/etc/modprobe.d/lustre.conf
```

9. Инсталирати *Lustre* кернел:

**Листинг 2.4: Инсталација *Lustre* кернела**

```
rpm -ivh kernel-2.6.32-358.23.2.el6_lustre.x86_64.rpm
    kernel-firmware-2.6.32-358.23.2.el6_lustre.x86_64.rpm
```

10. Инсталирати *e2fsprogs* *e2fsprogs* је скуп програма за одржавање *Linux* фајл система. *e2fsprogs* садржи неколико програма од којих су најпознатији:

- ***e2fsck*** - Проверава и поправља несугласице.
- ***mke2fs*** - Креира ext2, ext3 и ext4 фајл системе.
- ***resize2fs*** - Промена величине фајл система.
- ***tune2fs*** - Подешавање параметара фајл система.
- ***logsave*** - Снимање логова.
- ***e2label*** - Промена лабеле фајл система.
- ***findfs*** - Претрага фајл система по лабели или UUID.
- ***badblocks*** - Претрага лоших сектора.
- ***blkid*** - Штампа атрибуте блокова.
- ***chattr*** - Промена атрибута фајлова.

Инсталирати *e2fsprogs* (Листинг 2.5).

**Листинг 2.5: Инсталација *e2fsprogs***

```
rpm -Uvh e2fsprogs-1.42.7.wc2-7.el6.x86_64.rpm  e2fsprogs-libs-1
    .42.7.wc2-7.el6.x86_64.rpm libcom_err-1.42.7.wc2-7.el6.x86_64
    .rpm libss-1.42.7.wc2-7.el6.x86_64.rpm
```

11. Инсталирати *Lustre* кернел модула(Листинг 2.6).

**Листинг 2.6: Инсталација кернел модула**

```
rpm -ivh lustre-modules-2.4.2-2.6.32_358.23.2.el6_lustre.x86_64.
    x86_64.rpm lustre-ldiskfs-4.1.0-2.6.32_358.23.2.el6_lustre.
    x86_64.x86_64.rpm
```

12. Како би се *Lustre* лакше надгледао, потребно је инсталирати *net-snmp-libs (Simple Network Management Protocol (SNMP) )* (Листинг 2.7). SNMP је протокол за надгледање мрежне опреме. Net-SNMP је пакет апликација које се користе за имплементацију SNMP користећи IPv4, као и IPv6.

**Листинг 2.7: Инсталација Net-SNMP-а**

```
yum install net-snmp-libs

rpm -ivh lustre-osd-ldiskfs-2.4.2-2.6.32_358.23.2.el6_
lustre.x86_64.x86_64.rpm

rpm -ivh lustre-2.4.2-2.6.32_358.23.2.el6_lustre.x86_64.x86_64.
    rpm
```

13. Искључивање *SELinux*-а Да би се *Lustre* покренуо потребно је искључити *Security-Enhanced Linux (SELinux)* (Листинг 2.8). *SELinux* је безбедоносни кернел модул. У фајл */boot/grub/grub.conf* додати selinux=0.

**Листинг 2.8: Команде за искључивање *SELinux*-а**

```
kernel /boot/vmlinuz-2.6.32-358.23.2.el6\_lustre.x86\_64 ro root
    =UUID=2c8a0af6-545a-4761-9e41-74cfe026385e
rd\_NO\_LUKS rd\_NO\_LVM LANG=en\_US.UTF-8 rd\_NO\_MD SYSFONT=
    latarcyrheb-sun16 crashkernel=auto  KEYBOARDTYPE=pc
 KEYTABLE=us rd\_NO\_DM rhgb quiet selinux=0

reboot
```

14. Креирати физичких *volumes* за LVM систем на MGS, MDT, OSS(Листинг 2.9).

**Листинг 2.9: Излаз команде *fdisk -l***

```
fdisk -l

/dev/sda1   *           1         1306    10485760   83  Linux
/dev/sda2            1306         2350     8387584   83  Linux
/dev/sda3            2350         2611     2097152   82  Linux
    swap/Solaris
```

```
pvcreate /dev/sda2
```

Команда *pvs* даје излаз приказан на Листингу 2.10.

**Листинг 2.10:**

```
  PV         VG   Fmt  Attr PSize PFree
  /dev/sda2       lvm2 a--  8.00g 8.00g
```

15. Креирати групу *volumes*

```
vgcreate lustre /dev/sda2
```

Команда *vgs* даје следећи приказан на Листингу 2.11.

**Листинг 2.11: Излаз команде *vgs***

```
  VG      #PV #LV #SN Attr   VSize VFree
  lustre    1   1   0 wz--n- 8.00g 8.00g
```

У зависности од типа сервисног чвора покренути команду и креирати логичке *volumes*(Листинг 2.12).

**Листинг 2.12: Креирање логичких *volumes***

```
lvcreate -L 7.9G -n MGS lustre
lvcreate -L 7.9G -n MDT lustre
lvcreate -L 7.9G -n OST lustre
```

16. Конфигурисати MGS

    Креирати MGS фајл систем, а затим га и подигнути(Листинг 2.13).

**Листинг 2.13: Креирање MGS фајл систем**

```
mkfs.lustre --mgs /dev/lustre/MGS
mkdir -p /mnt/MGS/
mount -t lustre /dev/lustre/MGS /mnt/MGS/
```

Команда *df* даје следећи излаз приказан на Листингу 2.14.

**Листинг 2.14: Излаз *df* команде**

```
Filesystem           1K-blocks      Used Available Use% Mounted
    on
/dev/sda1             10321208   2615424   7600960  26% /
tmpfs                   388408        72    388336   1% /dev/shm
/dev/mapper/lustre-MGS
                       8156088    347176   7394604   5% /mnt/MGS
```

17. Конфигурисати MDT Креирати MDT фајл систем, а затим га и подигнути(Листинг 2.15).

**Листинг 2.15: Креирање MDT фајл систем**

```
mkfs.lustre --fsname=lustre --mdt  --mgsnode=mgs@tcp0 /dev/
    lustre/MDT
mkdir -p /mnt/MDT/
mount -t lustre /dev/lustre/MDT /mnt/MDT/
```

18. Конфигурисати OSS Креирати OSS фајл систем, а затим га и подигнути(Листинг 2.16).

**Листинг 2.16: Креирање OSS фајл систем**

```
mkfs.lustre --fsname=lustre --ost  --index=1 --mgsnode=mgs@tcp0
    /dev/lustre/OST
mkdir -p /mnt/OST/
mount -t lustre /dev/lustre/OST /mnt/OST/
```

19. Конфигурисати клијента На клијенту је потребно инсталирати *Lustre* клијент пакете(Листинг 2.17).

**Листинг 2.17: Команде за инсталацију *Lustre* клијент пакета**

```
yum -y groupinstall "Development Tools"

yum install net-snmp-libs

wget http://downloads.whamcloud.com/public/lustre/
    latest-maintenance-release/el6/client/RPMS/x86_64/
lustre-client-modules-2.4.2-2.6.32_358.23.2.el6.x86_64.x86_64.
    rpm

http://downloads.whamcloud.com/public/lustre/
    latest-maintenance-release/el6/client/RPMS/x86_64/
lustre-client-2.4.2-2.6.32_358.23.2.el6.x86_64.x86_64.rpm

yum localinstall  lustre-client-2.4.2-2.6.32_358.23.2.el6.x86_64
    .x86_64.rpm
lustre-client-modules-2.4.2-2.6.32_358.23.2.el6.x86_64.x86_64.
    rpm

mkdir /lustre

reboot
```

Подизање *Lustre* фајл система врши се командом:

```
mount -t lustre mgs@tcp0:/lustre /lustre
```

Уколико је све конфигурисано као што је назначено, команда *df* треба да да следећи излаз приказан на Листингу 2.18.

**Листинг 2.18: Излаз команде *df***

```
Filesystem            1K-blocks       Used Available Use% Mounted
    on
/dev/sda1              18577148    2735112  15653360  15% /
tmpfs                    388408         72    388336   1% /dev/shm
mgs@tcp0:/lustre        8156088     365396   7376384   5% /lustre
```

## 2.7 Кориснички алати за подешавање система

### 2.7.1 Алати за откривање грешака

*Lustre* је сложен систем и може се наићи на разнолике проблеме при коришћењу. Треба имати алат за откривање грешака при руци, који може помоћи у схватању проблема и разлога његовог настанка. Разноврсни алати за дијагностику и анализе су доступни приликом отклањања проблема са *Lustre* софтвером. Неки од њих се налазе у *Linux* дистрибуцијама, док су други развијени и доступни су у оквиру *Lustre* пројекта. Следећи *in-kernel* механизми за отклањање грешака су уграђени у *Lustre* софтвер:

- *Debug logs*
- *Debug* позадински сервис
- */proc/sys/lnet/debug*

Следећи програми су такође укључени у *Lustre* :

- *lctl*

- *Lustre subsystem asserts*
- *lfs*

Поред њих, ту су и општи алати који су уграђени у стандардне *Linux* дистрибуције:

- *strace*
- */var/log/messages*
- *Crash dumps*
- debugfs

Следећи *logging* алати могу се користити за скупљање информација приликом откривања *Lustre kernel* проблема.

- *kdump*
- *netconsole*
- *netdump*

За отклањање грешака *Lustre* система у развојном окружењу користи се *leak_finder.pl*, док се за отклањање грешака и анализу користе

- *kgdb*
- *crash.*

Кориснички програми:

- *mkfs.lustre*
- *tunefs.lustre*
- *lctl*
- *mount.lustre*
- *lustre_rsync*
- Додатни кориснички програми за подешавање конфигурације

### 2.7.2 mkfs.lustre

*mkfs.lustre* је кориснички програм који форматира диск за *Lustre* фајл систем.
Синтакса ове команде је:

```
mkfs.lustre <target_type> [options] device
```

где је <*target_ type*> један од следећих типова:

- **–ost** *Object Storage Target (OST)*
- **–mdt** *Metadata Storage Target (MDT)*
- **–network=net** врста мреже којом су повезани OST и MDT
- **–mgs** *Configuration Management Service (MGS)* може бити комбинован са –mdt

После форматирања диска, диск може бити коришћен од стране *Lustre* сервиса. Када је диск креиран, истом командом се могу подесити параметри одговарајућег диска. Параметри се додају као *–param* опција *mkfs.lustre* команде.

Параметри `mkfs.lustre` команде су:

- ***–backfstype=fstype*** Поставља посебан формат за резервне копије фајл система (као што су *ext3*, *ldiskfs*).
- ***–comment*** Поставља кориснички коментар.
- ***–device-size=KB*** Поставља величину уређаја.
- ***–dryrun*** Штампа шта ће бити учињено. Не утиче на операције са диском.
- ***–failnode=nid*** Поставља Ид партнера за отказе.
- ***–fsname=filesystem_name*** Поставља назив *Lustre* фајл система којег ће бити део. Подразумевани је „lustre”. Назив може имати највише 8 карактера.
- ***–index=index*** Поставља OST или MDT индекс.
- ***–mkfsoptions=opts*** Поставља опције за резервне копије. Опције могу бити постављене овде су ext3.
- ***–mountfsoptions=opts*** Поставља опције које се користе приликом монтирања диска.
- ***–mgsnode=nid*** Постављају NID за MGS чвор.
- ***–param sys.timeout=40*** Поставља тајм-аут система.
- ***–param lov.stripesize=2M*** Поставља stripe количину меморије.
- ***–param lov.stripecount=2*** Поставља број stripe делова.
- ***–param failover.mode=failout*** Враћа грешке диска, уместо да чека на опоравак.
- ***–reformat*** Реформатира диск.

### 2.7.3 *tunefs.lustre*

Кориснички програм који мења конфигурацију на *Lustre* диску је *tunefs.lustre*. Синтакса ове команде је:

```
tunefs.lustre [options] device
```

Овом командом се може инсталирати најновија верзија *Lustre* система, с тим што се приликом извршавања ове команде не брише садржај диска. Измене после извршавања команде се могу уочити након следећег подизања диска.

*tunefs.lustre* поставља параметре додавањем нових или мењањем старих. За брисање старих параметара и додавање нових потребно је покренути:

```
tunefs.lustre --erase-params --param=<new parameters>
```

Параметри *tunefs.lustre* команде су:

- ***–comment=comment*** Поставља кориснички коментар.

- ***–dryrun*** Штампа шта ће бити учињено. Не утиче на операције са диском.
- ***–erase-params*** Брише све претходне параметре.
- ***–failnode=nid*** Поставља Ид партнера за отказе
- ***–fsname=filesystem_name*** Поставља назив *Lustre* фајл система којег ће бити део. Подразумевани је „*lustre*”.
- ***–index=index*** Поставља OST или MDT индекс.
- ***–mkfsoptions=opts*** Поставља опције за резервне копије. Опције које могу бити постављене овде су ext3.
- ***–network=net*** Поставља мрежу преко које су повезани OST/MDT чворови.
- ***–mgs*** Додаје сервис за управљање конфигурацијом.
- ***–msgnode=nid,...*** Поставља NID(s) за MGS чвор
- ***–nomgs*** Уклања сервис за управљање конфигурацијом.
- ***–quiet*** Штампа основних информације.
- ***–verbose***

  Штампа више информације.
- ***–writeconf*** Брише све конфигурационе дневнике за фајл систем на коме је MDT, а затим их регенерише. Приликом извршавања ове команде може доћи до престанка рада сервера и клијената. У општем случају, ова команда треба да буде извршена само на MDT.

***lctl***

Кориснички програм који се користи за *root* контролу и конфигурацију назива се *lctl*. Са *lctl* може се директно контролисати *Lustre* , омогућавајући различите конфигурације.

```
lctl
lctl --device <OST device number> <command [args]>
```

Најчешће *lctl* команде су:

- ***dl*** Излистава *Lustre* уређаје по називу и броју. Команда штампа и UUID уређаја. На серверу, UUID је различит за све уређаје, док код клијента UUID је исти за све уређаје који су део исте тачке подизања фајл система.
- ***device*** Селектује задати OBD уређај. Све команде после селектовања уређаја зависиће од њега
- ***network <up/down>*** Укључује или искључује LNET. Овом командом је могуће и одређивање типа мреже за остале *lctl* LNET команде.
- ***list_nids*** Штампа све NIDs на локалном чвору.
- ***ping nid*** Проверава LNET умреженост користећи LNET ping.
- ***help*** Штампа листу свих могућих lctl команди.
- ***conn_list*** Штампа све умрежене чворове.
- ***route_list*** Штампа комплетну табелу рутирања.

### 2.7.4 *mount.lustre*

Кориснички програм који покреће *Lustre* клијента или други сервис је *mount.lustre*.

```
mount -t lustre [-o options] directory
```

Овај програм не треба позивати директно. Ово је помоћни програм програма *mount*. Командом *umount* стопирамо *Lustre* сервис. Постоје 2 облика подизања фајл система:

- ***<mgsspec>:/<fsname>*** Подиже *Lustre* фајл систем контактирајући MGS *<mgsspec>* на директоријуму *<directory>*. Подигнути фајл систем постоји у *fstab-y* и он је доступан као и остали локални фајл системи.
- ***<disk_device>*** Покреће сервис дефинисан *mkfs.lustre* командом на физичком диску *<disk_device>*. Подигнути фајл систем је доступан само за *df* операције. Постоји у *fstab-y*, показујући да је он у употреби.

### 2.7.5 *lustre_rsyncl*

*lustre_rsync* програм синхронизује *Lustre* фајл систем. Дизајниран је тако да синхронизује систем са другим фајл системом. Тај фајл систем може бити било који други фајл систем. Операција синхронизације је ефикасна и не захтева претрагу измена, већ користи MDT логове измена да идентификује измене у систему.

Опције овог програма су:

- ***–source=<src>*** Путања до *root-a Lustre* система који ће бити синхронизован.
- ***–target=<tgt>*** Путања до фајл система где се синхронизује *root Lustre* система.
- ***–mdt=<mdt>*** Извршити синхронизацију MDT-а.
- ***–user=<user id>*** Ид корисника логова измена. Да би се вршила синхронизација, корисник мора бити регистрован.
- ***–statuslog=<log>*** Лог фајл у коме ће се чувати статус синхронизације.
- ***–dry-run*** Штампа излаз *lustre_rsyncl* команде (*cp*, *mkdir*, итд.).
- ***–abort-on-err*** Уколико дође до грешке прекида операцију синнхронизације.

### 2.7.6 Додатни кориснички програми за подешавање конфигурације

***lustre_rmmod.sh***

Брише све *Lustre* и LNET модуле.

**e2scan**

Кориснички програм који претражује фајл систем у циљу проналаска последње измењених фајлова назива се *e2scan*. *e2scan* користи *libext2fs* да пронађе фајлове са новијим *mtime* и *ctime* од задатог. Користећи овај програм, веома се ефикасно може добити листа фајлова који су измењени.

```
e2scan [options] [-f file] block_device
```

e2scan скенира све чворове на уређају, проналази измењене фајлове, а затим штампа њихове бројеве.

**llobdstat**

llobdstat програм приказује OST статистику за задати OST и временски интервал.

```
llobdstat ost_name [interval]
```

**llstat**

Кориснички програм за приказивање статистике је *llstat*.

```
llstat [-c] [-g] [-i interval] stats_file
```

Опције овог програма су:

- ***-c*** Брише фајл у коме се чувају резултати статистике.
- ***-i*** Подешава временски интервал.
- ***-g*** Подешава графички приказ статистике.
- ***stats_file*** Подешава путању до фајла статистике или кратку референцу MDS или OST.

# Глава 3

# MPI-2 стандард

## 3.1 MPI

Као и *Lustre* фајл систем, и MPI има за циљ да побољша паралелизам. Најоптималнији учинак се постиже истовременом применом MPI стандарда и *Lustre* фајл система. MPI (Message-Passing Interface) је стандард за писање паралелних програма. MPI је развијан у две фазе, од стране произвођача паралелних рачунара, писаца библиотека и програмера апликација. Прва фаза је била 1993-1994 и резултат ове фазе је прва верзија MPI стандарда, названа MPI-1. Један број важних тема у паралелном рачунарству је намерно изостављен из MPI-1, како би се убрзао излазак нове верзије ове библиотеке. MPI форум се састао 1995. да би се размотриле ове теме као и извршавање мањих исправки и појашњења која су се појавила у MPI-1. Верзија стандарда MPI-2 изашла је у лето 1997.

Почевши са радионицама 1992. године, MPI форум формално је организован 1992. године. MPI стандард је успео да се развије захваљујући привлачењу пажње широког спектра заједнице паралелног рачунарства. На окупљањима, произвођачи паралелних компјутера су слали најбоље техничко особље. MPI форум се одржавао сваких шест недеља, почевши од јануара 1993, а прва верзија MPI је изашла већ у лето 1994.

Прва акција форума је била да исправи грешке и разјасни низ питања која су изазивала неспоразуме у оригиналном документу из јула 1994, који је означен као MPI-1.0. Све ове измене су заокружене у целину и у мају 1995. изашао је MPI-1.1. Исправке и појашњења су се наставила и следеће две године. Резултат тог рада је MPI-2 документ, који као поглавље садржи и верзију MPI-1.2. У наредним поглављима описан је стандардни метод за покретање MPI програма, а затим паралелне улазно/излазне операције у MPI-2. Значајно је напоменути да MPI 2 омогућава да процес директно приступа подацима другог процеса. Да би се увидела разлика између *Lustre* и NFS фајл система, покретани су програми који мере брзине улазно/излазних операција. Поређењем резултата уочава се да је *Lustre* фајл систем погоднији за паралелне програме.

## 3.2 Преносни процес покретања

Мали, али веома користан додатак MPI-2 стандарда је стандардни метод за покретање MPI програма, који у ранијем стандарду није био специфициран. Најједноставнији пример овог метода је

```
mpiexec -n 16 myprog
```

за покретање програма *myprog* на 16 процеса. MPI спецификација стриктно не говори како се покреће MPI програм, већ утиче на писање самог програма. Од MPI програма се захтева покретање на широком спектру окружења, различитим оперативним системима, менаџерима процеса итд. Све ово доводи до чињенице да механизам за мулти-процесно покретање није могућ.

Међутим, корисници желе да програме са једне машине покрену на другој машини без икаквих додатних подешавања. Неколико савремених MPI имплементација користи *mpirun* за покретање MPI послова. Команда *mpirun* се разликује од имплементације до имплементације и захтева различите аргументе. То доводи до конфузијем поготово када су различите MPI имплементације инсталиране на истој машини. Да би прекинули све недоумице, MPI форум је одлучио да се позабави и овим проблемом у верзији стандарда MPI-2. Она препоручује да је *mpiexec* једини програм за покретање MPI апликација и да су аргументи овог програма тачно одређени и јединствени. Команда

```
mpiexec -n 32 myprog
```

стартује 32 MPI процеса, где је величина `MPI_COMM_WORLD` комуникатора 32. Назив *mpiexec* је изабран да се избегну сукоби са различитим варијантама mpirun програма.

Поред *-n <numprocs>* argumenta, mpiexec има и један мали број аргумената који су одређени MPI стандаром. У сваком случају формат за аргументе је -<назив> вредност. Неки од осталих аргумената су:

- *soft*
- *host*
- *arch*
- *wdir*
- *path*
- *file*

```
mpiexec -n 32 -soft 16 myprog
```

Значи да уколико се због ограничења распоређивања процеса програм не може покренути на 32 процеса, онда да се покрене на 16 процеса.

```
mpiexec -n 4 -host denali -wdir /home/me/outfiles myprog
```

Значи покренути 4 процеса на машини под називом denali и притом поставити радни директоријум на */home/me/outfiles*.

```
mpiexec -n 12 -soft 1:12 -arch sparc-solaris \
-path /home/me/sunprogs myprog
```

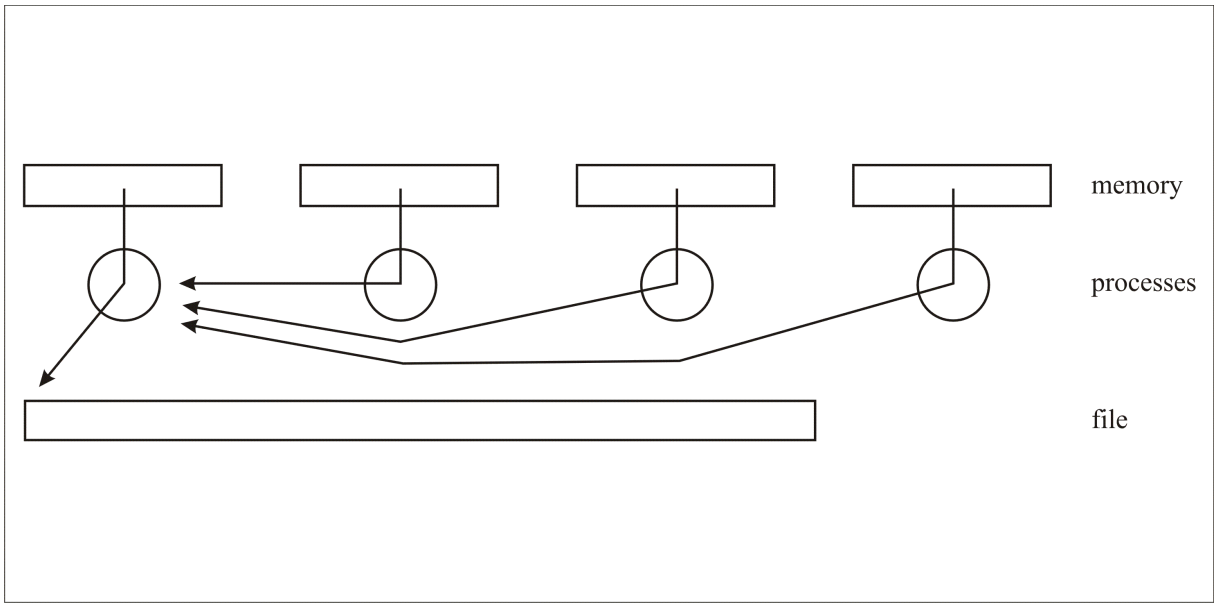
Значи да уколико се не може покренути програм на 12 процеса, покренути програм на било ком броју процеса од 1 до 12, на *sparc-solaris* архитектури, с тим што се програм *myprog* налази у директоријуму */home/me/sunprogs*.

```
mpiexec -file myfile
```

Значи да *mpiexec* погледа *myfile* за следеће инструкције. Формат фајла зависи од MPI имплементације.

## 3.3 Паралелне улазно/излазне операције

Паралелне улазно/излазне операције у MPI програму се обављају функцијама које су сличне стандардним „језичким" улазно/излазним функцијама и библиотекама. MPI има неколико додатних функција које побољшавају учинак и портабилност програма. Основна карактеристика MPI је да може изразити паралелизам у овим операцијама. Секвенцијалне улазно/излазне операције паралелног програма приказане су на слици 3.1.




Слика 3.1: Секвенцијалне улазно/излазне операције паралелног програма

### 3.3.1 MPI програм - непаралелне У/И операције

MPI-1 нема никакву експлицитну подршку за паралелне У/И операције, док су MPI апликације развијене у последњих неколико година морале да имају и свој У/И део програма. Ти делови програма су писани ослањајући се на карактеристике које пружа оперативни систем, најчешће UNIX. Најједноставније је имати један процес који извршава све У/И операције, док други процеси извршавају операције са учитаним подацима. Ако се узме пример писања низа бројева у фајл дужине 100, онда дужина података коју обрађује сваки процес зависи од укупног броја процеса. Програм почиње иницијализацијом дела низа за сваки процес. Сви процеси осим процеса 0 шаљу своје делове низа процесу 0. Процес 0 уписује свој део низа у фајл, а затим прихвата делове низа од других процеса. Ранг сваког процеса је одређен у `MPI_Recv` функцији, тако да се зна редослед пристизања делова низа. Ово је најчешћи начин да се непаралелне У/И операције врше у паралелном

програму који је конвертован из секвенцијалног програма, јер промене нису направљене на У/И делу програма.

Уколико је $numprocs = 1$, онда нема MPI комуникације.

Неки од разлога зашто се У/И операције пишу на овај начин су:

- Паралелни компјутери на којима је покренут програм можда подржавају У/И операције само са једног процеса.
- Могу се користити софистициране У/И библиотеке које су можда писане као део високог нивоа слоја за управљање података, а које не подржавају паралелне У/И операције.
- Резултујући фајл је погодан за руковање изван програма (нпр. *mv*, *cp*, или *ftp*).

Учинак програма може бити побољшан омогућавањем процеса да складишти велики блок података. Уколико процес 0 има довољан бафер за податке, он може акумулирати податке других процеса у јединствен бафер за једну велику операцију писања(Листинг 3.1). Разлог због кога не треба писати У/И операције на овај начин је недостатак паралелизма који ограничава учинак и скалабилност, нарочито ако основни систем фајлова омогућава паралелне У/И операције.

**Листинг 3.1: MPI програм**

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
  int i, myrank, numprocs, buf[BUFSIZE];
  MPI_Status status;
  FILE *myfile;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  for (i=0; i<BUFSIZE; i++)
  {
    buf[i] = myrank * BUFSIZE + i;
  }
  if (myrank != 0)
  {
    MPI_Send(buf, BUFSIZE, MPI_INT, 0, 99, MPI_COMM_WORLD);
  }
  else
  {
    myfile = fopen("testfile", "w");
    fwrite(buf, sizeof(int), BUFSIZE, myfile);
    for (i=1; i<numprocs; i++)
    {
      MPI_Recv(buf, BUFSIZE, MPI_INT, i, 99, MPI_COMM_WORLD,
      &status);
      fwrite(buf, sizeof(int), BUFSIZE, myfile);
    }
    fclose(myfile);
  }
  MPI_Finalize();
  return 0;
}
```

### 3.3.2 MPI програм - без MPI улазно/излазних операција

У циљу решавања овог недостатка, следећи корак у миграцији секвенцијалног програма ка паралелном је да се за сваки процес оперише са посебним фајлом, што омогућава паралелни пренос података(Листинг 3.2). Овде су У/И операције сваког процеса потпуно

независне од У/И операција других процеса. Тако, је сваки програм секвенцијалан у односу на У/И операције. Наиме, процес отвара свој фајл, уписује податке у њега, а затим га и затвара. Најбоље је да се у називу излазног фајла налази и ранг процеса. Предност овог приступа је да се У/И операције могу одвијати паралелно, а и даље се могу користити секвенцијалне У/И библиотеке. Основни недостатак оваквог приступа је креирање више фајлова уместо једног. Поред тога, недостаци овакве шеме су:

- Фајлови се морају спојити пре него што буду коришћени као улаз у другом програму.
- Може се десити да програм који чита ове фајлове мора бити паралелни и стартован са истим бројем процеса.
- Тешко је држати скуп фајлова као групу, ради копирања, премештања и слања путем мреже.

**Листинг 3.2: MPI програм без MPI улазно/излазних операција**

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
  int i, myrank, buf[BUFSIZE];
  char filename[128];
  FILE *myfile;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  for (i=0; i<BUFSIZE; i++)
  {
    buf[i] = myrank * BUFSIZE + i;
  }
  sprintf(filename, "testfile.%d", myrank);
  myfile = fopen(filename, "w");
  fwrite(buf, sizeof(int), BUFSIZE, myfile);
  fclose(myfile);
  MPI_Finalize();
  return 0;
}
```

Учинак може бити мањи уколико имамо велики број процеса. То ће довести до великог броја У/И операција са малим бројем података.

### 3.3.3 MPI У/И операције са одвојеним фајловима

MPI У/И програм је сличан као и претходни програм, с тим што се све У/И операције извршавају MPI функцијама(Листинг 3.3). Овакав програм има неколико предности и мана.

Прва разлика је у овим фајловима је та што је декларација `FILE` замењена са `MPI_File` као типом *myfile*. Сада је *myfile* променљива типа MPI_File, уместо показивач на објекат типа FILE.

**Листинг 3.3: MPI програм са одвојеним фајловима**

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
  int i, myrank, buf[BUFSIZE];
  char filename[128];
  MPI_File myfile;
  MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for (i=0; i<BUFSIZE; i++)
    {
      buf[i] = myrank * BUFSIZE + i;
    }
    sprintf(filename, "testfile.%d", myrank);
    MPI_File_open(MPI_COMM_SELF, filename,
    MPI_MODE_WRONLY | MPI_MODE_CREATE,MPI_INFO_NULL, &myfile);
    MPI_File_write(myfile, buf, BUFSIZE, MPI_INT,
    MPI_STATUS_IGNORE);
    MPI_File_close(&myfile);
    MPI_Finalize();
    return 0;
  }
```

MPI функција која замењује функцију fopen назива се `MPI_File_open`.

```
MPI_File_open(MPI_COMM\_SELF, filename,MPI_MODE_CREATE | MPI_MODE_WRONLY,
MPI_INFO_NULL, &myfile);
```

Аргументи ове функције су:

- **Комуникатор** - Ово је најзначајнија компонента У/И операција у MPI. Фајлови су отворени скупом процесора идентификованих од стране комуникатора. Ово обезбеђује да процеси раде на фајлу заједно омогућујући и комуникацију између процеса. Пошто сваки процес отвара свој фајл, онда се користи комуникатор `MPI_COMM_SELF`.
- **Назив фајла** - Други аргумент је стринг који представља назив фајла као и у функцији *fopen*.
- **Тип мода** - Трећи аргумент је мод у коме је фајл отворен. У овом програму значи да је креиран или преписан ако већ постоји, као и да ће писање у фајл бити извршено само од стране овог програма. Константе `MPI_MODE_CREATE` и `MPI_MODE_WRONLY` представљају заставице.
- **MPI_INFO_NULL** - MPI_INFO_NULL је предефинисана константа која представља лажну вредност за инфо аргумент `MPI_File_open`.
- **Фајл променљива** - Као последњи аргумент је адреса `MPI_File` променљиве, коју ће функција `MPI_File_open` отворити. Као и све MPI функције у C програму, `MPI_File_open` има повратну вредост. Уколико је фајл успешно отворен повратна вредност је `MPI_SUCCESS`.

Следећа функција која је важна за функционисање за MPI У/И је:

```
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT,MPI_STATUS_IGNORE);
```

Податак који се уписује мора бити одређен адресом, величином и типом. Овим начином се описује бафер који ће бити коришћен за писање(Листинг 3.4). То омогућава да се несуседни подаци у меморији запишу само једним позивом. Конкретно, овде се уписују `BUFSIZE` целих бројева са почетком у адреси *buf*. Последњи аргумент функције је статус, који је истог типа као и код `MPI_Recv`. У овом сличају занемариће се повратна вредност. MPI-2 је одредио специјалну вредност статуса `MPI_STATUS_IGNORE`. Ова вредност може бити послата ако аргумент било којој MPI функцији у циљу игнорисања повратне вредности одговарајуће функције. Технички, ово упрошћење може побољшати учинак програма уколико нам статус није потребан.

Функција `MPI_File_close(&myfile)` служи за затварање фајла. Послата адреса *myfile* биће преписана са `MPI_FILE_NULL` уколико се затварање фајла не обави успешно. Тако се могу идентификовати неважећи фајлови.

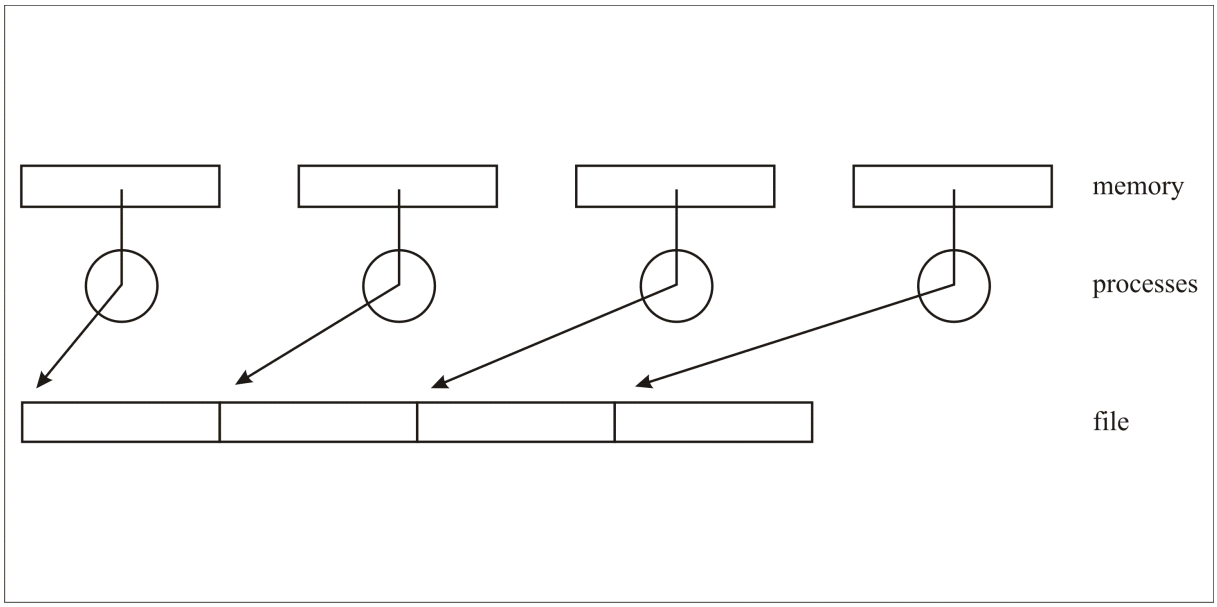
**Листинг 3.4: MPI програм за паралелним MPI функцијама**

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
  int i, myrank, buf[BUFSIZE];
  MPI_File thefile;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  for (i=0; i<BUFSIZE; i++)
  {
    buf[i] = myrank * BUFSIZE + i;
  }
  MPI_File_open(MPI_COMM_WORLD, "testfile",
  MPI_MODE_CREATE | MPI_MODE_WRONLY,
  MPI_INFO_NULL, &thefile);
  MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
  MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
  MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
  MPI_STATUS_IGNORE);
  MPI_File_close(&thefile);
  MPI_Finalize();
  return 0;
}
```

### 3.3.4 Паралелне MPI У/И операције са једним фајлом

Да би се добио још бољи учинак MPI У/И операција, потребно је изменити програм тако да процеси деле један фајл, уместо да пишу у више њих(слика 3.2). Тако се отклањају све мане код уписа у више фајлова и постиже се потпуни паралелизам.

Прва разлика између програма који уписује у више различитих фајлова је први аргумент функције `MPI_File_open`. Пошто сада сваки процес не приступа свом фајлу, већ једном дељеном за све процесе, уместо комуникатора `MPI_COMM_SELF` користи се комуникатор `MPI_COMM_WORLD`. Тиме се постиже да сви процеси отварају исти фајл.




Слика 3.2: Паралелне MPI У/И операције са једним фајлом

Ово је колективна операција на комуникатору, тако да сви процеси који учествују позивају `MPI_File_open`, при чему се, као што је напоменуто, отвара се само један фајл. Део фајла може се видети у процесу, што се назива **поглед фајла**. Поглед фајла се поставља функцијом `MPI_File_set_view`.

```
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int), MPI_INT,
 MPI_INT, "native", MPI_INFO_NULL);
```

Први аргумент идентификује фајл. Други аргумент је место (у бајтовима) у фајлу од кога почиње део фајла асоциран датом процесу. Овде множимо величину података (BUFSIZE * sizeof (int)) по рангу процеса, тако да поглед на сваки процес почиње на одговарајућем месту у фајлу. Тај аргумент је типа `MPI_Offset`, који на системима који подржавају велике фајлове очекује 64-битни цели број. Следећи аргумент се назива **etype погледа**. То је скуп свих типова података који се налазе у фајлу. У овом случају то је `MPI_INT`, што значи да ће се у фајл увек уписати цели број. Следећи аргумент се назива *filetype*, и то је веома флексибилан начин да се опишу дисконтинуални погледи у фајлу. У овом случају ради се само о типу `MPI_INT`, тако да нема дисконтинуалних података за упис. Генерално, *etype* и *filetype* могу да буду било који предефинисани MPI типови података. Аргумент који означава представљање података у фајлу и најчешће је типа стринг назива се „природни". Нативно представљање значи да се подаци уписују у фајл тачно онако како су представљени у меморији. Ова шема чува податке и сумарни учинак програма, јер се не губи време ни на какве додатне конверзије. Остала представљања су *унутрашња* и *external32*, која омогућавају различите врсте преноса између машина са различитим архитектурама и типовима представљања. Последњи аргумент је инфо аргумент као и у функцији `MPI_File_open`.

Неке MPI функције у програмском језику C се налазе у Листингу 3.5:

**Листинг 3.5: MPI функције**

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info
    info,
MPI_File *fh)

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype
   etype,
MPI_Datatype filetype, char *datarep, MPI_Info info)

int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype
   datatype,
MPI_Status *status)
int MPI_File_close(MPI_File *fh)

int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype
   datatype,
MPI_Status *status)
```

Овим начином, написани програм је независан од броја процеса на којима је покренут. Укупна величина датотеке се добија тако да сваки процес ради са скоро истом величином података. MPI функција која се користи за добијање величине фајла је `MPI_File_get_size`. Први аргумент ове функције је отворени фајл, а другі је адреса где треба сместити израчунату величину фајла у бајтовима. Пошто многи системи сада могу управљати датотекама чије су дужине превелике да би биле представљене као 32-битни цео број, MPI дефинише тип, `MPI_Offset`, који може да а садржи величину фајла у 64 бита. То је тип који се користи за аргументе MPI функција који се односе на померање у фајловима. У супротном, програм који се користи за читање фајла је веома сличан оном који пише. Разлика између писања и читања је да процес не зна увек тачно колико ће података буде прочитано.

### 3.3.5 Коришћење појединачних фајл показивача

MPI програм са улазно/излазним операцијама је могуће писати и са појединачним фајл показивачима(Листинг 3.6). Сваки од ових програма има део за У/И операције које отварају, читају и на крају затварају фајл. `MPI_File_open` је функција која отвара фајл. Први аргумент је комуникатор који идентификује групу процеса којима је потребан приступ фајлу. `MPI_COMM_WORLD` се користи зато што је свим процесима потребан приступ фајлу */pfs/datafile*. MPI стандард не одређује формат за назив фајла. Свака од MPI имплементација има слободу да дефинише формат који они подржавају. Може се очекивати да ће имплементација подржати познате конвенције именовања. Имплементације које се покрећу на Unix окружењу подржавају Unix конвенцију именовања. */pfs/datafile* је фајл који се налази у директоријуму *pfs*. У имплементацијама је назив директоријума опциони део назива фајла. Уколико не постоји, имплементација ће користити директоријум у коме се процес тренутно налази. Трећи аргумент функције `MPI_File_open` је начин приступа. У овом случају то је `MPI_MODE_RDONLY`, зато што је довољно да програм само чита из фајла. Четврти аргумент је инфо аргумент. Последњи аргумент је показивач на фајл који враћа функција `MPI_File_open`.

**Листинг 3.6: MPI програм са појединачним фајл показивачима**

```
#include "mpi.h"
#define FILESIZE (1024 * 1024)
int main (int argc, char **argv)
{
int *buf, rank, nprocs, nints, bufsize;
MPI_File fh;
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
bufsize = FILESIZE/nprocs;
buf = (int *) malloc (bufsize);
nints = bufsize/sizeof (int);
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,
MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
free (buf);
MPI_Finalize();
return 0;
}
```

Овај C програм извршава улазно/излазне операције користећи појединачне показиваче на фајл. После отварања фајла, сваки процес копира глобални фајл показивач у локални фајл показивач који показује на локацију у фајлу од које сваки процес чита свој део фајла. Први аргумент функције `MPI_File_seek` је показивач на фајл који је отворила функција `MPI_File_open`. Други аргумент одређује део фајла који сваки процес чита. `MPI_SEEK_SET` значи да се почетак локације за читање рачуна од почетка фајла. У C програмском језику за ово се користи предефинисан тип `MPI_Offset`. Имплементација дефинише `MPI_Offset` као цео број који је довољно велики да подржи највећу дужину фајла. Део фајла сваког процеса се одређује производом ранга процеса и величине података у бајтовима. Величина података може се одредити и функцијама `MPI_Get_count` или `MPI_Get_elements`, користећи статус објекат који враћа функција `MPI_File_read`.

### 3.3.6 Употреба експлицитних одступања

`MPI_File_read` и `MPI_File_write` се називају појединачним фајл показивачима због тога што користе показивач на локацију у фајлу од које сваки процес чита фајл(Листинг 3.7).

MPI такође специфицира неколико функција које се називају **експлицитним функцијама одступања** (`MPI_File_read_at` и `MPI_File_write_at`). Ове функције не користе појединачне фајл показиваче. У њима, локација у фајлу се директно прослеђује функцији као аргумент. Уколико више нити процеса приступају истом фајлу, онда се морају користити појединачни фајл показивачи због безбедности нити.

**Листинг 3.7: MPI функције**

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int
    count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int
    count,
MPI_Datatype datatype, MPI_Status *status)
```

### 3.3.7 Писање у фајл

Уколико је потребно уписати податке у фајл, онда се користе функције `MPI_File_write` и `MPI_File_write_at`. Уместо заставице `MPI_MODE_RDONLY` која је служила за читање фајла, за упис података у фајл се користе заставице `MPI_MODE_CREATE` и `MPI_MODE_WRONLY`. `MPI_MODE_CREATE` се користи за креирање фајла уколико он већ не постоји. `MPI_MODE_WRONLY` означава да је фајл отворен за писање. У C програмском језику, обе заставице могу се користити битовним или оператором : `MPI_MODE_CREATE | MPI_MODE_WRONLY`. Да би функција `MPI_File_open` креирала фајл, потребно је да постоји директоријум који је наведен у називу фајла.

### 3.3.8 Неконтинуални приступи и колективне У/И операције

У великом броју реалних паралелних апликација, сваком процесу је потребно да приступи малим деловима података који су смештени у фајлу неконтинуално [4, 17, 65, 77, 78, 85]. Један начин да се приступи неконтунуалним подацима је користећи функције за читање/писање малих континуалних делова, као у Unix системима. Због велике латенције улазно/излазних операција, приступање малим деловима података захтева много времена. Предност MPI на Unix системима је могућност приступа неконтинуалним деловима података позивајући само једну функцију.

#### Неконтинуални приступи

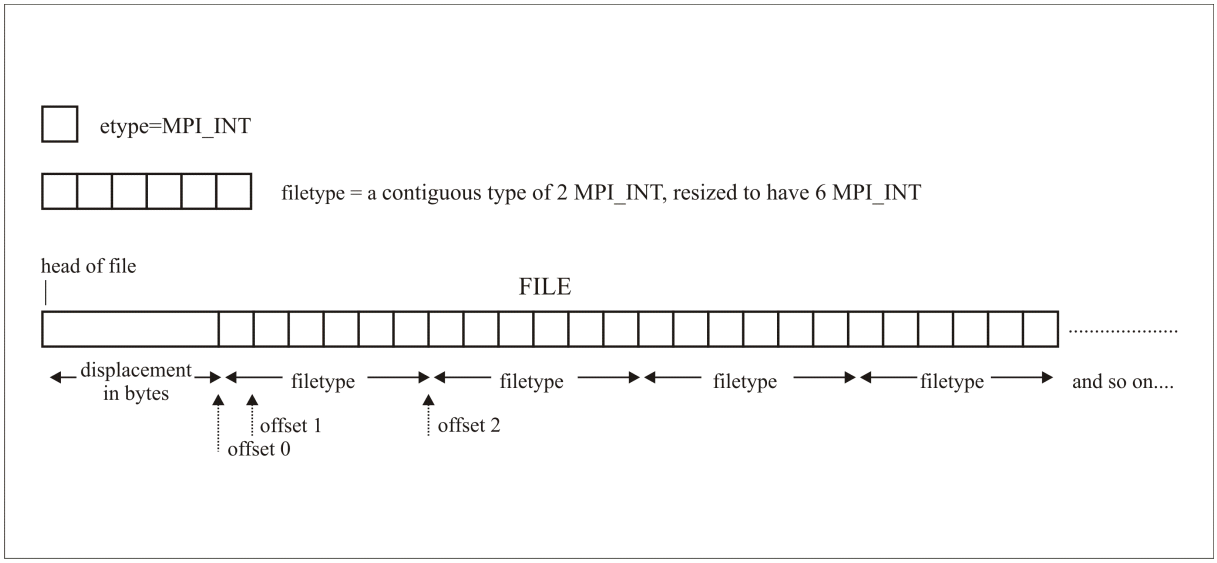
MPI програм поседује концепт погледа на фајл. Поглед фајла у MPI-у је дефинисан као део фајла коме процес има приступ(слика 3.3). Користећи поглед на фајл, функције читања и писања могу приступити само том делу фајла. Сви остали подаци се прескачу. Када се отвори, фајл је доступан процесу и MPI третира фајл као скуп бајтова (не као цели бројеви, реални бројеви итд.). Приликом покретања програма, сваки појединачни фајл показивач је постављен на 0. Ово се може променити помоћу функције `MPI_File_set_view`(Листинг 3.8). Најчешће се то ради из два разлога:

- Да се одреди тип податка коме је потребно приступити, нпр. целим или децималним бројевима. Ово је нарочито неопходно за преносивост фајла уколико корисник жели да приступи фајлу са друге машине и са различитим представљањем фајла.
- Да се одреде делови фајла који ће бити прескочени, тј. одређивање неконтунуалног приступа фајлу. За појединачне фајл показиваче или експлицитна одступања, сваки процес може користети различит тип погледа.

За приступ подацима са подељеним фајл показивачем потребно је да сви процеси користе исти поглед. Поглед фајла се може мењати небројано пута. MPI типови података се користе за одређивање погледа фајла. Поглед је одређен са:

- премештање
- *etype*
- *filetype*

**Премештање** одређује број бајтова који ће бити прескочени на почетку фајла. Ово се користи када је потребно да се прескочи читање заглавља фајла. **Etype** је основна јединица за приступ подацима. Може бити основни или изведени MPI тип података. Сви приступи фајлу се врше преко јединица типа etype. Сва одступања фајла се дефинишу као број *etype-ова*. Уколико је *etype* MPI_INT, појединачни и подељен фајл показивач биће померен за одређени број целих бројева. **Filetype** је основни или изведени тип података који дефинише који део фајла је доступан процесу и ког типа су подаци. *Filetype* мора бити исти као etype или изведен од типа који се заснива на *etype*. Поглед фајл почиње од премештања и састоји се од више суседних *etype*. Приликом отварања фајла, премештање има вредност 0 и *etype* и *filetype* су типа `MPI_BYTE`. Ово је познато као подразумевани поглед фајла.




Слика 3.3: Поглед фајла

На слици 3.3 је приказан суседни изведени тип података који је представљен као два цела броја. Уколико се поставе још 4 цела броја на крај овог типа података, функцијом `MPI_Type_create_resized` се добија тип податка који је величине шест целих бројева. `Etype` је типа `MPI_INT`, а премештање је 5*sizeof (int). У MPI-1 верзији ово се ради са функцијом `MPI_Type_struct`.

**Листинг 3.8: MPI функције**

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype
    etype,
MPI_Datatype filetype, char *datarep, MPI_Info info)
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
MPI_Aint extent, MPI_Datatype *newtype)
```

Аргументи који се прослеђују функцији `MPI_File_set_view` су:

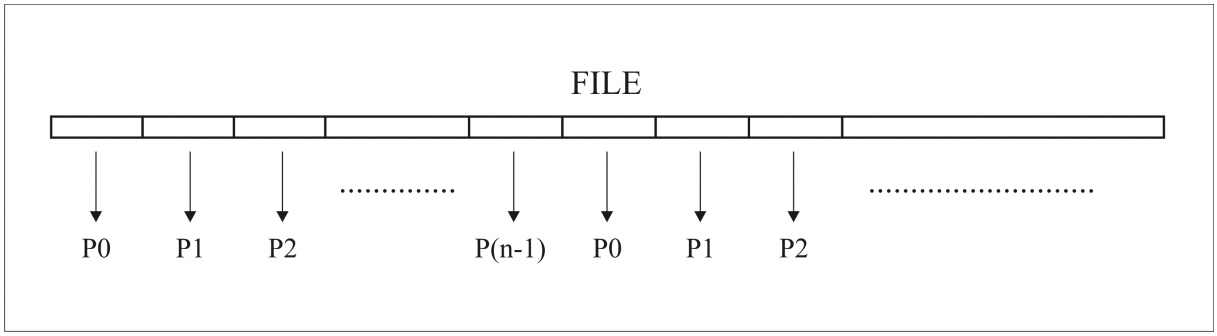
- показивач на фајл

- премештање
- *etype*
- *filetype*
- представљање података
- инфо аргумент

Подразумевано представљање је нативно, док је подразумевани инфо аргумент `MPI_INFO_NULL`.

### Колективне У/И операције

Разлика између колективних У/И операција са неконтинуалним приступом и других У/И операција је у томе што код колективних операција сваки процес чита мале блокове података, који се налазе у фајлу по принципу *round-robin* распоређивања(слика 3.4). Са Unix У/И операцијама, једини начин да се читају подаци је читање сваког блока одвојено, због тога што Unix функције омогућавају приступ само једном континуалном делу података.




Слика 3.4: Колективне У/И операције

Код MPI-а, уместо позивања функције за читање више пута, може се дефинисати поглед на неконтунуални фајл сваког процеса, како би се прочитао фајл позивајући функцију само једном(Листинг 3.9). Други начин је употреба колективног читања. MPI имплементација даје много бољи учинак у односу на Unix У/И функције.

`FILESIZE` одређује величину фајла у бајтовима. `INTS_PER_BLK` је величина сваког блока који процес треба да прочита (број целих бројева у блоку). Сваки процес треба да прочита неколико блокова у цикличном распореду.

**Листинг 3.9: MPI програм са погледом фајла**

```
#include "mpi.h"
#define FILESIZE
#define INTS_PER_BLK 104857616

int main(int argc, char **argv)
{
  int *buf, rank, nprocs, nints, bufsize;
  MPI_File fh;
  MPI_Datatype filetype;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  bufsize = FILESIZE/nprocs;
  buf = (int *) malloc(bufsize);
  nints = bufsize/sizeof(int);
```

```
    MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,
    MPI_INFO_NULL, &fh);
    MPI_Type_vector(nints/INTS_PER_BLK, INTS_PER_BLK,
    INTS_PER_BLK*nprocs, MPI_INT, &filetype);
    MPI_Type_commit(&filetype);
    MPI_File_set_view(fh, INTS_PER_BLK*sizeof(int)*rank, MPI_INT,
    filetype, "native", MPI_INFO_NULL);
    MPI_File_read_all(fh, buf, nints, MPI_INT, MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
    MPI_Type_free(&filetype);
    free(buf);
    MPI_Finalize();
    return 0;
}
```

Помоћу `MPI_File_open` отвара се фајл и поставља се комуникатор MPI_COMM_WORLD пошто сваки процес треба да има приступ фајлу */pfs/datafile*. Следећи корак је дефинисање погледа. За filetype, креира се изведени тип типа вектор, користећи функцију `MPI_Type_vector`. Први аргумент ове функције је број блокова који сваки процес треба да прочита. Други аргумент је број целих бројева у сваком блоку, док је трећи број целих бројева између полазних елемената два узастопна блока које процес чита. Четврти аргумент је тип податка, у овом случају MPI_INT. Пети аргумент је повратна вредност функције `MPI_Type_vector`. Након креирања овог типа, нови тип се може користити као *filetype* аргумент функције `MPI_File_set_view`.

*Etype* је `MPI_INT`. Улазно/излазне операције се извршавају користећи колективну верзију функције `MPI_File_read`, која се назива `MPI_File_read_all`. У позивима ових функција нема разлике. Једина разлика је што се колективна функција позива од стране сваког процеса у комуникатору који је прослеђен функцији `MPI_File_open`. Овај комуникатор је имплицитно прослеђен функцији `MPI_File_read_all`. Функција `MPI_File_read`, може се позивати независно од стране процеса.

### 3.3.9 Приступни низови смештени у фајловима

Велики број паралелних програма има један или више вишедимензионих низова подељених између процеса. Локални низ сваког процеса није контунуално смештен у фајл. Сваки ред низа сваког процеса је одвојен редовима локалних низова других процеса. MPI омогућава погодан начин за опис улазно/излазних операција и извршава их преко једног позива функције. Уколико корисник користи колективне У/И функције, MPI имплементација омогућава бољи учинак користећи овакав приступ, иако је приступ дисконтунуални. У MPI-2 је дефинисано два нова типа конструктора података: **darray** и **subarray**. Ове функције олакшавају креирање изведених типова података, описујући локацију локалних низова спојених у један глобални низ. Ови типови података могу бити коришћени као filetype да опишу дисконтинуални приступ фајлу, када се обавља У/И операција за подељене низове.

### 3.3.10 Подељени низови

Конструктор типа података *darray* омогућава лак начин креирања изведеног типа података, који описује мултидимензионални глобални низ који се састоји од локалних низова(слика 3.5). Низ може бити било којих димензија и свака димензија може бити дистрибуирана на било који начин. Аргументи *darray* конструктора су величина низа, опис расподеле и ранг процеса чији је локални низ описан. Излаз је изведен тип података који описује распоред локалних низова у глобалном низу. Постоје и други начини за креирање изведених типова података, али су они нешто сложенији. Први аргумент функције `MPI_Type_create_darray` је број процеса којима је низ дистрибуиран. Други аргумент је ранг процеса чији је локални низ описан. Трећи аргумент су димензије низа, док је четврти аргумент сам низ.

**Листинг 3.10: Део MPI програма са подељеним низовима**

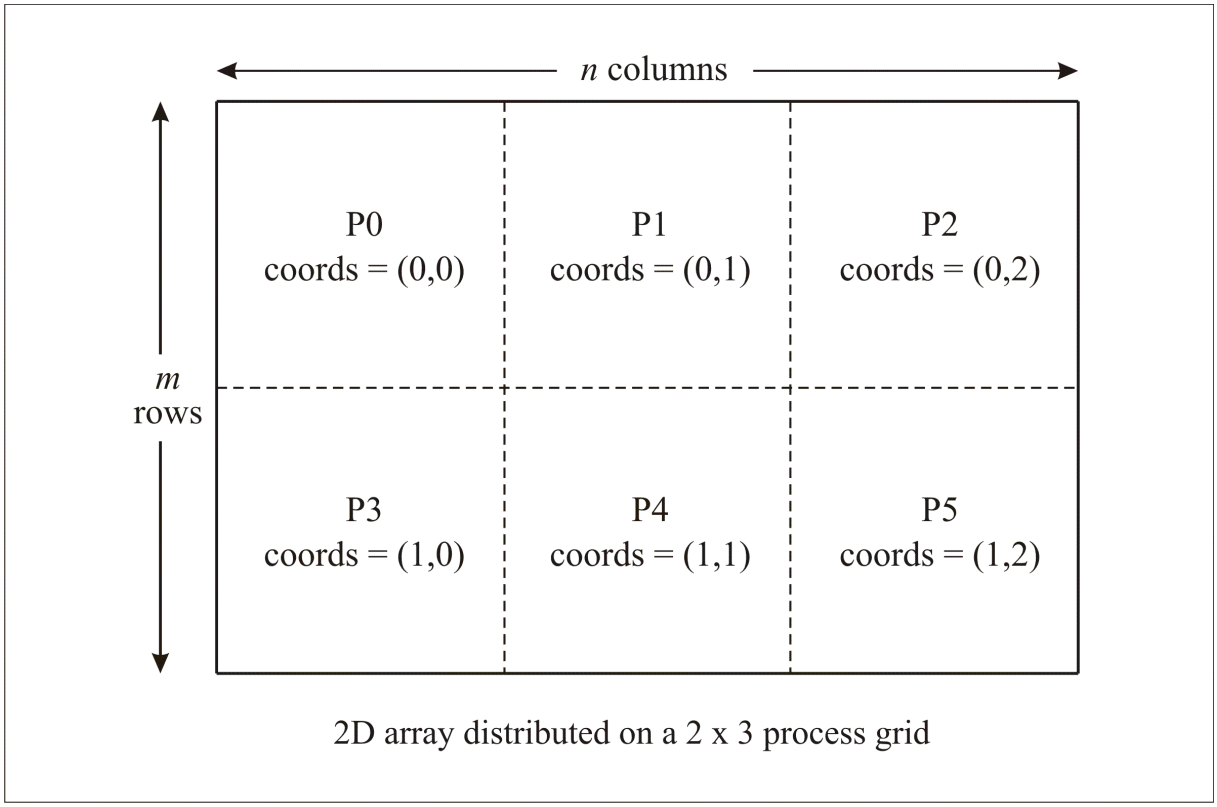
```
gsizes[0] = m;
gsizes[1] = n;

distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;

dargs[0] = MPI_DISTRIBUTE_DFLT_DARG; /* default block size */
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG; /* default block size */
psizes[0] = 2;
psizes[1] = 3;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs,
psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
MPI_MODE_CREATE | MPI_MODE_WRONLY,
MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
MPI_INFO_NULL);
local_array_size = num_local_rows * num_local_cols;
MPI_File_write_all(fh, local_array, local_array_size,
MPI_FLOAT, &status);
MPI_File_close(&fh);
```

Пети аргумент је низ који одређује начин на који је глобални низ подељен. На овом примеру, ту улогу има `MPI_DISTRIBUTE_BLOCK`. Шести аргумент одређује дистрибуциони параметар за сваку димензију, у овом случају у цикличној($k$) расподели. За блок и цикличне расподеле којима није потребан овај аргумент, подразумевана вредност је `MPI_DISTRIBUTE_DFLT_DARG`. Седми аргумент је низ који одређује број процеса дуж сваке димензије глобалног низа. Грид процеса увек има димензије као и глобални низ.




Слика 3.5: Подељени низ

Осми аргумент функције `MPI_Type_create_darray` одређује редослед складиштења локалног низа у меморији, као и глобалног низа у фајлу. Девети аргумент је *datatype* који описује ког је типа елемент низа, у овом програму `MPI_FLOAT`. Повратна вредност функције је изведени тип података *&filetype*. После комитовања типа података, нови тип може се користити у постављању погледа(Листинг 3.10).

### 3.3.11 Неблокирајуће У/И операције и подељене колективне У/И операције

MPI подржава неблокирајућу верзију независних функција за писање и читање. MPI механизам омогућава ове функције, слично као и неблокирајуће интерпроцесне комуникације. Неблокирајуће функције почињу са `MPI_File_ixxx`, нпр. `MPI_File_iread` и `MPI_File_iwrite_at`. Неблокирајуће функције враћају `MPI_Request` објекат. Могу се користити уобичајене MPI_Test и MPI_Wait операције. Користећи ове функције, може доћи до преклапања улазно/излазних операција са осталим комуникацијама у програму. Ова преклапања зависе од квалитета имплементације. За колективне операције, MPI подржава ограничен облик неблокирајућих операција, које се називају подељеним колективним У/И операцијама. Да би се користиле подељене колективне функције, корисник мора позвати „почетак" функције (`MPI_File_read_all_begin`) да би се покренула колективна У/И, као и "крај" функције (`MPI_File_read_all_end`) да би се она завршила. Ограничење је да корисник у исто време може имати само једну подељену колективну операцију над једним фајлом. Подељене колективне функције не враћају `MPI_Request` објекат.

### 3.3.12 Подељени фајл показивачи

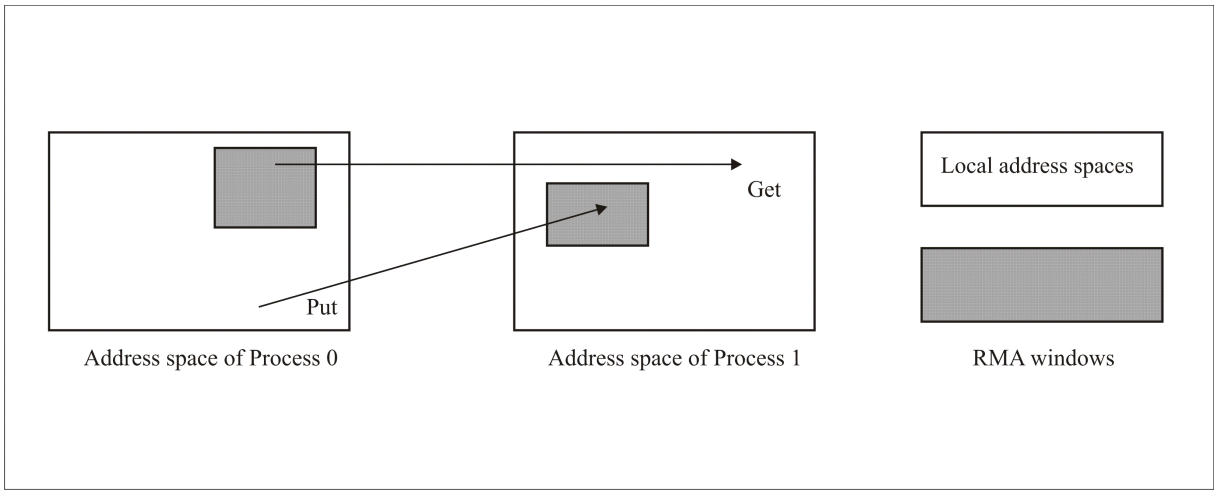
Постоји три начина да се одреди локација у фајлу са које је потребно прочитати или уписати податке: појединачни фајл показивачи, експлицитна одступања и подељени фајл показивачи. **Подељени фајл показивач** је фајл показивач чија вредност је подељена између процеса који се налазе у комуникатору функције `MPI_File_open`. MPI пружа функције `MPI_File_read_shared` и `MPI_File_write_shared` које читају/уписују податке са почетком у тренутној локацији подељеног фајл показивача. После позивања ових функција, подељени фајл показивач биће освежен новом количином података који су уписани или прочитани. Следећи позив ових функција ће радити са освеженим подељеним фајл показивачем. Процес може експлицитно померити показивач у *etypes* јединици помоћу функције `MPI_File_seek_shared`. MPI захтева да сви процеси имају исти фајл поглед када користе подељене фајл показиваче. За остала два начина могу се користити различити фајл погледи.

## 3.4 Даљински приступ меморији

MPI-2 омогућава да процес директно приступи подацима другог процеса. Ове операције које омогућавају читање и писање тих података називају се *remote memory access* (RMA) операције. Главна карактеристика MPI имплементације је слање података између процеса помоћу операција за слање и примање података. Треба приметити да MPI-2 не омогућава реални подељени меморијски модел. Даљинске меморијске операције MPI-2 омогућавају велику флексибилност подељене меморије. Даљински приступ меморији је пројектован да ради на машинама са дељеном меморијом и на окружењима која немају дељену меморију, као што су мреже радних станица које користе TCP/IP протокол за комуникацију. Њихова главна предност је флексибилност коју нуде у пројектовању алгоритама. Крајњи програми су преносиви кроз све MPI имплементације и биће ефикасни на свим платформама које омогућавају приступ меморији других процеса.

### 3.4.1 Меморијски оквири

У строгом прослеђивању порука, бафери за слање и примање су одређени MPI типовима података који представљају делове адреса процеса који се шаљу другим процесима у случају слања, или адреса где ће други процеси уписати податке у случају примања. У MPI-2 имплементацији, појам комуникацијске меморије је генерализован на појам оквира за даљински приступ меморији. Сваки процес може одредити део адресног простора који је доступан другим процесима за писање и читање. Операције писања и читања, које су покренуте од стране другог процеса називају се *get* и *put* операције за даљински приступ меморији(слика 3.6). Трећи тип операција је accumulate. У MPI-2, оквир представља део меморије једног процеса који чини дистрибуиран објекат, који се назива оквирни објекат. Оквирни објекат је направљен од више оквира, од којих се сваки састоји од локалне меморијске области која је изложена другим процесима.




Слика 3.6: Меморијски оквир

### 3.4.2 Перформансе програма са даљинским приступом меморији

Програм за рачунање $\pi$ рачуна вредност броја помоћу нумеричке интеграције. У класичној верзији постоји 2 типа комуникације. Процес 0 комуницира са корисником и захтева број интервала за интеграцију. Помоћу функције `MPI_Bcast` процес 0 шаље тај број другим процесима. Сваки процес затим израчунава парцијалну суму и све суме се сумирају помоћу колективне `MPI_Reduce` операције.

У једностраној верзији овог програма, процес 0 снима вредност броја интервала као део RMA оквирног објекта, одакле га други процеси могу једноставно прочитати(Листинг 3.11). После израчунавања парцијалне суме, сви процеси додају своју вредност у други оквирни објекат помоћу accumulate операције. Сваки оквирни објекат садржи само један број у меморији. Оквирни објекти су представљени као променљиве типа `MPI_Win`. Функције за креирање оквира су следеће:

```
MPI_Win_create (&n, sizeof(int), 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
```

Позив са процеса 0 треба бити упарен са осталим процесима, иако они не доприносе никакву меморију за оквирни објекат, пошто је `MPI_Win_create` колективна операција над процесима који се налазе у комуникатору. Комуникатор одређује који процеси могу приступити оквирном објекту. Прва два аргумента функције `MPI_Win_create` су адреса и дужина оквира у бајтовима.

**Листинг 3.11: MPI програм са даљинским приступом меморији**

```
#include "mpi.h"
#include <math.h>

int main(int argc, char *argv[])
{
  int n, myid, numprocs, i;
  double PI25DT = 3.14159265358979323846264 3;
  double mypi, pi, h, sum, x;
  MPI_Win nwin, piwin;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  if (myid == 0)
  {
    MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL,
    MPI_COMM_WORLD, &nwin);
    MPI_Win_create(&pi, sizeof(double), 1, MPI_INFO_NULL,
    MPI_COMM_WORLD, &piwin);
  }
  else
  {
    MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
    MPI_COMM_WORLD, &nwin);
    MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
    MPI_COMM_WORLD, &piwin);
  }
  MPI_Win_fence(0, nwin);
  while (1)
  {
    if (myid == 0)
    {
      printf("Enter the number of intervals: (0 quits) ");
      fflush(stdout);
      scanf("%d",&n);
      pi = 0.0;
    }
    MPI_Win_fence(0, nwin);
    if (myid != 0)
    MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
    MPI_Win_fence(0, nwin);
    if (n == 0)
    {
      break;
    }
    else
    {
      h = 1.0 / (double) n;
      sum = 0.0;
      for (i = myid + 1; i <= n; i += numprocs)
      {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
      }
      mypi = h * sum;
      MPI_Win_fence( 0, piwin);
      MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE,
      MPI_SUM, piwin);
      MPI_Win_fence(0, piwin);
      if (myid == 0)
      printf("pi is approximately %. 16f, Error is %. 16f/n",
      pi, fabs(pi - PI25DT));
    }
  }
  MPI_Win_free(&nwin);
  MPI_Win_free(&piwin);
  MPI_Finalize();
  return 0;
}
```

Следећи аргумент је displacement unit који се користи да одреди одступање локације у меморији. Сваки оквирни објекат садржи једну променљиву, код којих је одступање нула, тако да одступање у овом примеру може да се занемари. Четврти аргумент је `MPI_Info` који се може користити да побољша учинак RMA операција. Пети аргумент је комуникатор који одређује скуп процеса који ће имати приступ меморији оквирног објекта. MPI имплементација враћа `MPI_Win` објекат као последњи аргумент. После позива функције `MPI_Win_create`, сваки процес који се налази у комуникатору има приступ података *nwin* помоћу операција *put*, *get* и *accumulate*. За меморију оквира није потребно алоцирати посебну меморију, већ се користи меморија самог процеса, којој остали процеси приступају. MPI имплементација омогућава алоцирање посебне меморије позивом функције `MPI_Alloc_mem`.

Други позив функције `MPI_Win_create` креира оквирни објекат piwin дозвољавајући сваком процесу да приступи променљивој $\pi$ првог процеса, где ће бити смештена израчуната вредност броја $\pi$. У следећем делу програма, процес са рангом 0 захтева број интервала, а затим остали процеси рачунају број $\pi$. Петља се завршава када корисник унесе нулу. Процеси којима ранг није нула, вредност броја $n$ узимају директно из оквирног објекта без икакве додатне акције. Пре позива функције MPI_ Get или било које функције за даљински приступ меморији, потребно је позвати функцију `MPI_Win_fence` да одвоји операције. MPI имплентација омогућава специјални механизам синхронизације за операције дељене меморије - *three of them*. *Fence* операција је изазвана функцијом `MPI_Win_fence` која захтева два аргумента. Први аргумент је потврдни аргумент за дозвољавање оптимизације. Увек исправан потврдни аргумент је 0. Други аргумент је оквир на коме се операција извршава. `MPI_Win_fence` се може тумачити као баријера која одваја локалне операције на оквиру од скупа даљинских операција на оквиру.

У овом програму одваја читање вредности променљиве $n$ од осталих даљинских операција које следе. Вредност променљиве $n$ остали процеси добијају помоћу позивом:

```
MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin)
```

Аргументи ове функције су слични аргументима функција које примају или шаљу податак. Get операција је слична операцији примања, па су зато прва три аргумента опис податка који се прима (адреса, количина и тип податка). Следећи аргумент је ранг процеса чијој меморији приступамо. У овом случају је ранг 0, јер сви процеси приступају меморији првог процеса. Следећа три аргумента дефинишу бафер за слање (адреса, количина и тип податка). Овде се адреса даје као одступање од почетка локације у дељеној меморији. У овом случају је 0, зато што се приступа само једној вредности. Последњи аргумент је објекат оквира. `MPI_Get` је неблокирајућа операција. После позива ове функције не може се гарантовати да је вредност смештена у променљивој $n$. Зато је потребно позвати `MPI_Win_fence`. Сваки процес рачуна свој део суме *mypi*. Сада се позива `MPI_Win_fence`, али на оквирном објекту piwin, како би се покренуо други RMA приступ. Позивом функције `MPI_Accumulate`, сабирају се све суме процеса у глобалну суму.

```
MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE, MPI_SUM, piwin)
```

Прва три аргумента одређују локалну променљиву (адреса, количина и тип податка), док је четврти аргумент ранг процеса. Следећа три аргумента описују променљиву коју је потребно изменити. Аргумент који следи је операција коју је потребно извршити. Пошто нам је потребна глобална сума, у овом случају то је `MPI_SUM`. Последњи аргумент је објекат оквира. Програм се завршава штампањем вредности броја $\pi$ и ослобађањем меморије објекта помоћу функције `MPI_Win_free`. `MPI_Win_free` је колективна функција над комуникатором прослеђеног објекта оквира.

Неке MPI функције у програмском језику C дате су у Листингу 3.12:

**Листинг 3.12: MPI функције**

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
    MPI_Info info,MPI_Comm comm, MPI_Win *win)
int MPI_Win_fence(int assert, MPI_Win win)
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
    origin_datatype,
int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)
int MPI_Accumulate(void *origin_addr, int origin_count,
MPI_Datatype origin_datatype, int target_rank,
MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
int MPI_Win_free(MPI_Win *win)
```

## 3.5 Управљање процесима

Процес модел који се користи у MPI-1 имплементацијама користи фиксиран број процеса током MPI рачунања. Ово је концептуално једноставан модел, јер ставља све сложености интеракције са оперативним системом (који мора бити укључен у стварање процеса) потпуно изван оквира апликације. Када се изврши `MPI_Init`, процеси су покренути и комуникатор `MPI_COMM_WORLD` има коначан број процеса. Они могу међусобно да комуницирају преко комуникатора. Други комуникатори имају своје групе, која је подгрупа `MPI_COMM_WORLD` комуникатора. Динамичнији приступ управљањем процеса пролазилази из PVM (*Parallel Virtual Machine*) заједнице, где се процеси покрећу под контролом апликације. Интеркомуникатор служи да повеже две групе процеса. Интеркомуникатори омогућавају природан начин да опишу *spawning* процесе. Интеркомуникатори се могу спојити помоћу функције MPI_Intercomm_merge, чије је повратна вредност нови интракомуникатор.

### 3.5.1 *Spawning* процеса

У MPI-2 имплементацијама, процес се креира помоћу функције `MPI_Comm_spawn`. Кључна предности `MPI_Comm_spawn` су:

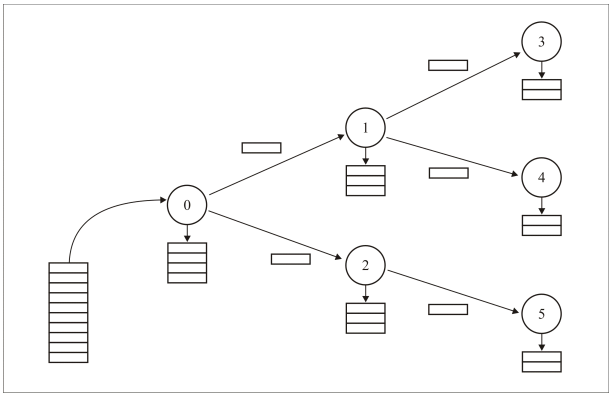
- Ово је колективна операција над новим процесима.
- Нови процеси имају свој властити `MPI_COMM_WORLD`.
- Функција `MPI_Comm_parent`, позвана од стране детета процеса, као повратну вредност има интеркомуникатор који садржи децу процеса као локалну групу и родитеље као даљинску групу.

### 3.5.2 Пример паралелног копирања

Једноставан услужни програм који извршава паралелно копирање тако што копира фајл са локалног диска машине на локалне дискове других машина(Листинг 3.13). Са MPI имплементацијом може се на скалабилан начин смањити време извршавања програма. Основни начин слања фајла помоћу MPI-а је користећи функцију `MPI_Bcast` за слање са *root* процеса(слика 3.7). Да би се покренуо програм, потребно је на свакој машини имати извршни фајл.

Процес са рангом 0 чита фајл, а затим користећи `MPI_Bcast` шаље блок по блок фајла другим процесима. Овај начин садржи 3 облика паралелизма:

- Сви процеси извршавају паралелне улазно/излазне операције са фајлом.
- Већи део слања порука између процеса се одвија паралелно.




Слика 3.7: MPI remote

- Подела фајла у блокове се одвија у *pipeline* паралелизму.

Први део програма је парсирање листе свих машина на које је потребно ископирати фајл, а затим и креирање празног фајла на тим машинама. Функција makehostlist парсира први аргумент и штампа списак машина у фајл чији је назив прослеђен као други аргумент. Број машина је повратна вредност ове функције.

```
makehostlist( argv[1], "targets", &num_hosts );
```

Да би сви процеси знали име фајла у коме се налази списак машина, потребно је да се назив фајла проследи функцији која покреће нове процесе `MPI_Comm_spawn` помоћу инфо објекта. Креира се инфо објекат који садржи „targets" као вредност резервисаног кључа. Овај инфо кључ једноставно говори функцији `MPI_Comm_spawn` да погледа фајл „targets" за више информација.

**Листинг 3.13: MPI програм за паралелно копирање**

```
#include "mpi.h"
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BUFSIZE 256*1024
#define CMDSIZE 80

int main( int argc, char *argv[] )
{
  int num_hosts, mystatus, allstatus, done, numread;
  int infd, outfd;
  char utfilename[MAXPATHLEN], controlmsg[CMDSIZE];
  char buf[BUFSIZE];
  char soft_limit[20];
  MPI_Info hostinfo;
  MPI_Comm pcpslaves, all_processes;
  MPI_Init( &argc, &argv );
  makehostlist( argv[1], "targets", &num_hosts );
  MPI_Info_create( &hostinfo );
  MPI_Info_set( hostinfo, "file", "targets" );
  sprintf( soft_limit, "0:%d", num_hosts );
  MPI_Info_set( hostinfo, "soft", soft_limit );
  MPI_Comm_spawn( "pcp_slave", MPI_ARGV_NULL, num_hosts,
  hostinfo, 0, MPI_COMM_SELF, &pcpslaves,
  MPI_ERRCODES_IGNORE );
  MPI_Info_free( &hostinfo );
  MPI_Intercomm_merge( pcpslaves, 0, &all-processes );
  strcpy( outfilename, argv[3] );
  if ( (infd = open( argv[2], O_RDONLY ) ) == -1 )
```

```
    {
      fprintf( stderr, "input %s does not exist\n", argv[2] );
      sprintf( controlmsg, "exit" );
      MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, all_processes );
      MPI_Finalize();
      return -1 ;
    }
    else
    {
      sprintf( controlmsg, "ready" );
      MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, all_processes );
    }
    MPI_Bcast( outfilename, MAXPATHLEN, MPI_CHAR, 0,
    all_processes );
    if ( (outfd = open( outfilename, O_CREAT|O_WRONLY|O_TRUNC,
    S_IRWXU ) ) == -1 )
    {
      mystatus = -1;
    }
    else
    {
      mystatus = 0;
    }
    MPI_Allreduce( &mystatus, &allstatus, 1, MPI_INT, MPI_MIN,
    all_processes );
    if ( allstatus == -1 )
    {
      fprintf( stderr, "Output file %s could not be opened\n",
      outfilename );
      MPI_Finalize();
      return 1 ;
    }
    /* at this point all files have been successfully opened */
    done = 0;
    while (!done)
    {
      numread = read( infd, buf, BUFSIZE );
      MPI_Bcast( &numread, 1, MPI_INT, 0, all_processes );
      if ( numread > 0 )
      {
        MPI_Bcast( buf, numread, MPI_BYTE, 0, all_processes );
        write( outfd, buf, numread );
      }
      else
      {
        close( outfd );
        done = 1;
      }
    }
    MPI_Comm_free( &pcpslaves );
    MPI_Comm_free( &all_processes );
    MPI_Finalize();
    return 0;
  }
```

Програм конвертује интеркомуникатор `pcpslaves` који садржи покренут процес и процесе које је креирала функција `MPI_Comm_spawn` у један заједнички интракомуникатор помоћу `MPI_Intercomm_merge`. Интракомуникатор *all_processes* се користи као комуникатор између *root* машине и осталих машина. Процеси покушавају отворити улазни фајл, и уколико дође до грешке, шаље се сигнал за прекид рада.

Да би се знало да је сваки процес отворио фајл, користи се `MPI_Allreduce` функција са `MPI_MIN` операцијом. Уколико било који процес не може да отвори фајл, сви процеси ће то сазнати и позвати `MPI_Finalize` за прекид рада. Код за дете процес је сличан коду родитеља процеса, с тим што дете процеса мора да позове `MPI_Comm_get_parent` функцију да успостави контакт са родитељем. Дете процес не обрађује аргументе нити штампа поруке. Родитељ процес затим чита блок по блок фајла и шаље осталим процесима. На крају

сви процеси ослобађају интеркомуникатор креиран од стране `MPI_Comm_spawn` и обједињени интракомуникатор. Главна разлика између `MPI_Comm_spawn` функције и осталих система за слање порука је природност колективних операција. У MPI имплементацији, група процеса колективно креира другу групу процеса који су међусобно синхронизовани. Тиме се спречавају додатни услови и омогућава неопходна комуникациона инфраструктура.

# Глава 4

# Тестирање брзине извршавања У/И операција

Како би се увидела предност *Lustre* фајл система у односу на NFS фајл систем, вршено је мерење брзине извршавања У/И операција на *Medflow* кластеру високих перформанси који користи оба наведена фајл система. Тестирање је вршено помоћу три различите апликације:

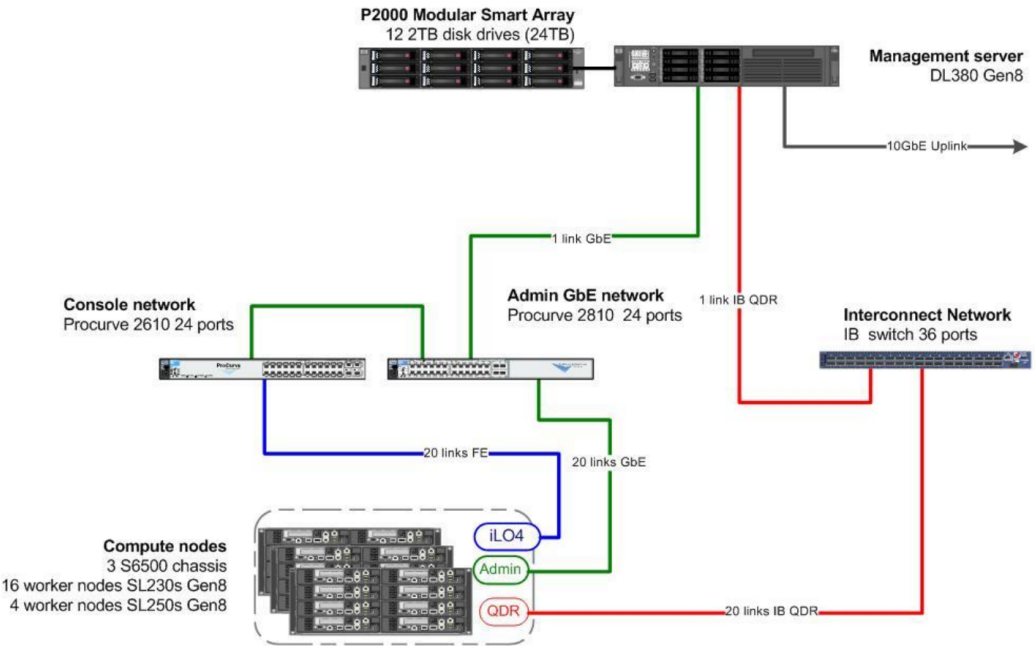
- *Iozone* - апликација намењена тестирању перформанси улазно-излазних операција,
- `dd` - једноставан UNIX системски алат за конверзију и копирање фајлова,
- *Game of Life* - "реална" апликација која поред У/И операција поседује и делове у којима се интензивно рачуна.

## 4.1 Хардверска конфигурација кластера

Medflow кластер је састављен од HP Proliant SL230s Gen8 и HP Proliant SL250s Gen8 радних чворова, заснованих на Intel® Xeon® E5-2600 (Sandy Bridge) процесорима. Кластер се састоји од следећих компоненти:

- 18 радних чворова HP Proliant SL230s Gen8 са 2 Intel® Xeon® E5-2660 процесора (2.2GHz – 8 cores - 20MB L3 cache - 95W) и 64GB RAM меморије.
- 4 радна чвора P Proliant SL250s са 2 Intel® Xeon® E5-2670 (2.6GHz – 8 cores - 20MB L3 cache - 115W) процесора и 64GB RAM меморије. Сваки од ових чворова садржи и 1 GPU NVIDIA Tesla M2090 6G графичку картицу.
- Једног *Mellanox Infiniband* QDR свич преко кога су повезани чворови.
- Једног управљачког сервера DL380pGen8 који омогућава пријаву на систем.
- Једног управљачког сервера DL380pGen8 који омогућава функционисање паралелног фајл система.
- Система за складиштење података који садржи 1 *HP P2000 G3 Modular Smart Array System* повезан на управљачки сервер, што даје укупно 12TB простора (6 HDD дискова по 2TB). Низ је конфигурисан као RAID ниво 6. Сви чворови могу приступити овим дисковима помоћу *Lustre* протокола.

Кластер је конфигурисан са Linux Kernel 2.6 на свим чворовима. Scientific Linux 6.6 64-bit је инсталиран како на управљачком чвору, тако и на радним чворовима. Ресурсима кластера се управља помоћу *TORQUE* и *Maui* сервиса. Први је намењен контроли и управљању кластер пословима, док је други распоређивач.




Слика 4.1: Компоненте кластера

Програмско окружење је засновано на OpenMPI библиотекама за Linux оперативни систем. На кластеру је инсталирана OpenMPI верзија 1.6.5. Треба нагласити и да је NFS фајл систем инсталиран постављен само на једном диску, док Lustre користи цео RAID6 низ.

## 4.2 Iozone тестирање

*Iozone* је алат за мерење брзине У/И операција фајл система. Он генерише и мери трајање великог броја операција са фајловима. *Iozone* може бити инсталиран на великом броју архитектура и такође може радити у оквиру многих оперативних система. Тестирање се врши кроз следеће операције:

- *Write* - мери брзину уписа података у нови фајл.
- *Re-write* - мери брзину уписа у фајл који већ постоји
- *Read* - мери брзину читања из фајла
- *Re-read* - мери брзину читања из фајла који је претходно прочитан
- *Random read* - мери брзину читања из фајла са насумичним приступом локацијама унутар фајла
- *Random write* - мери брзину писања у фајл са насумичним приступом локацијама унутар фајла
- *Random mix* - мери брзину писања и читања из фајла са насумичним приступом локацијама унутар фајла
- *Backwards read* - мери брзину читања из фајла уназад

- *Record rewrite* - мери брзину писања података у одређени део фајла
- *Strided read* - мери брзину читања из фајла са тачно одређеним параметрима
- *Fwrite* - мери брзину писања у фајл помоћу функције `fwrite()`
- *Fread* - мери брзину читања из фајла помоћу функције `fread()`
- *Freread* - мери поновно читања из фајла помоћу функције `fwrite()`

### 4.2.1 Инсталација и покретање програма

*Iozone* програм се инсталира помоћу следећих команди:

**Листинг 4.1: Инсталација Iozone**

```
wget http://www.iozone.org/src/current/iozone3_394.tar
tar xvf iozone3_394.tar
cd iozone3_394/src/current
make
make linux
```

Овај програм је могуће покренути и помоћу великог број параметара. Конкретни параметри који су коришћени за тестирање на кластеру имају следеће значење:

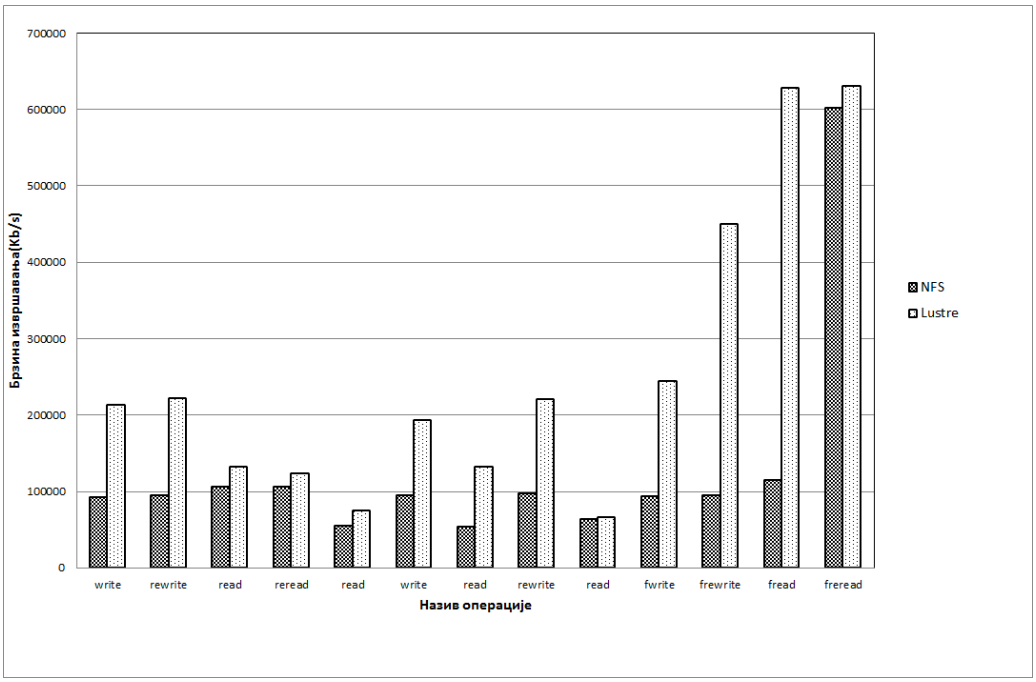
- `-b назив фајла` - генерисање Excel излазног фајла
- `-c` - мери и време које је потребно за функцију *close()*
- `-I` - користи *DIRECT I/O* заставицу за све фајл операције
- `-o` - уписује синхроно на диск. *Iozone* отвара фајл са *O_SYNC* заставицом
- `-r` - одређује величину записа у килобајтима
- `-s` - одређује величину фајла који се користи за тестирање

Програм се покреће помоћу команде:

```
./iozone -s 25000000 -r 1024 -I -c -o -b output.xls
```

Резултати извршења програма се могу видети у фајлу *output.xls*.

Посматрајући резултате представљене на Слици 4.2, закључујемо да је брзина извршавања улазно/излазних операција *Lustre* фајл система значајно већа у односу на брзину NFS система без обзира на врсту операције која се извршава. Највећа разлика у брзини се уочава код теста који користи `fread()` команду, док је најмања разлика у брзини при извршавању операције обичног читања.




Слика 4.2: *Iozone* резултати

## 4.3 Тестирање помоћу системског алата `dd`

`dd` је једноставан алат који служи за писање и читање блокова података диска. Он такође мери и брзину којом је операција извршена.

Параметри командне линије су:

- `if` - улазни фајл
- `of` - излазни фајл
- `bs` - величина блока
- `count` - број блокова

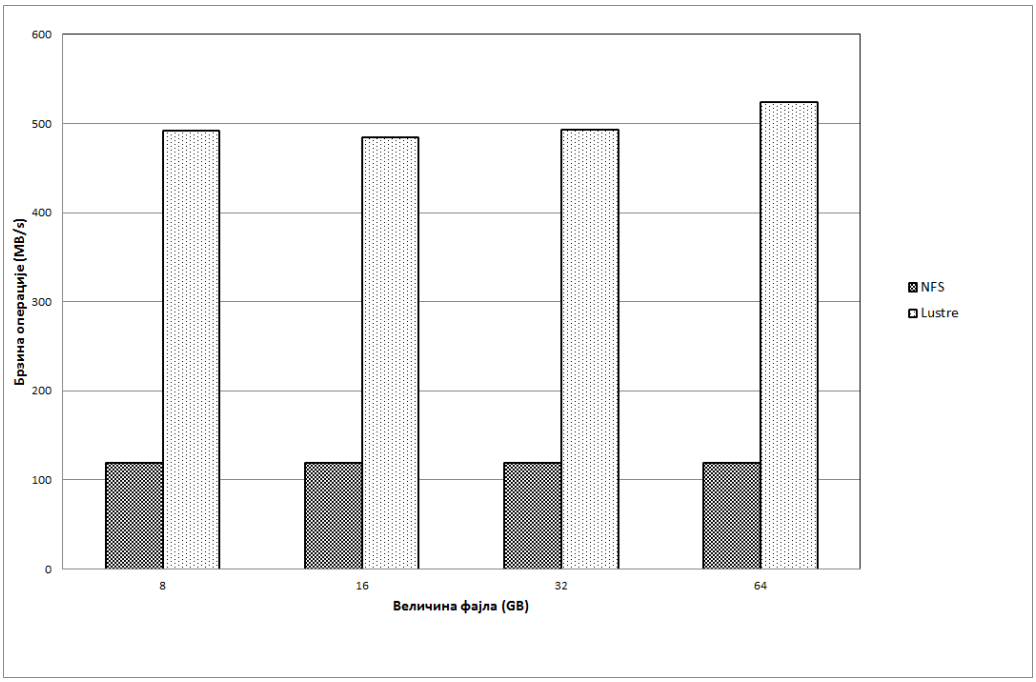
Програм за упис у фајл се покреће командом:

```
dd if=/dev/zero bs=1M count=16384 of=file_16GB
```

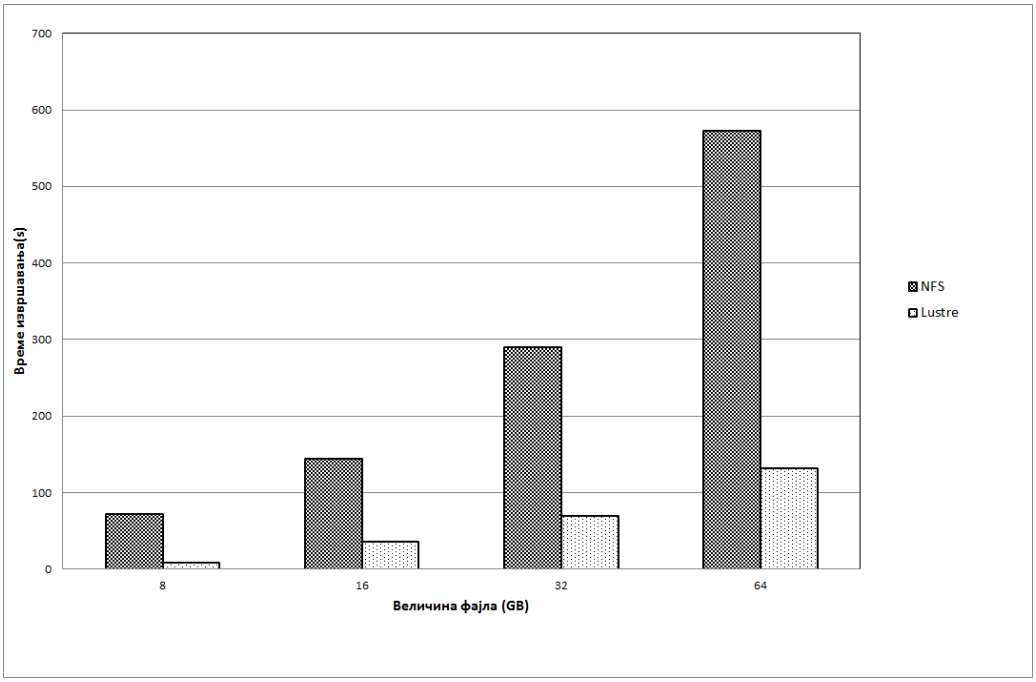
а за читање из фајла:

```
dd if=file_16GB bs=1M of=/dev/null
```

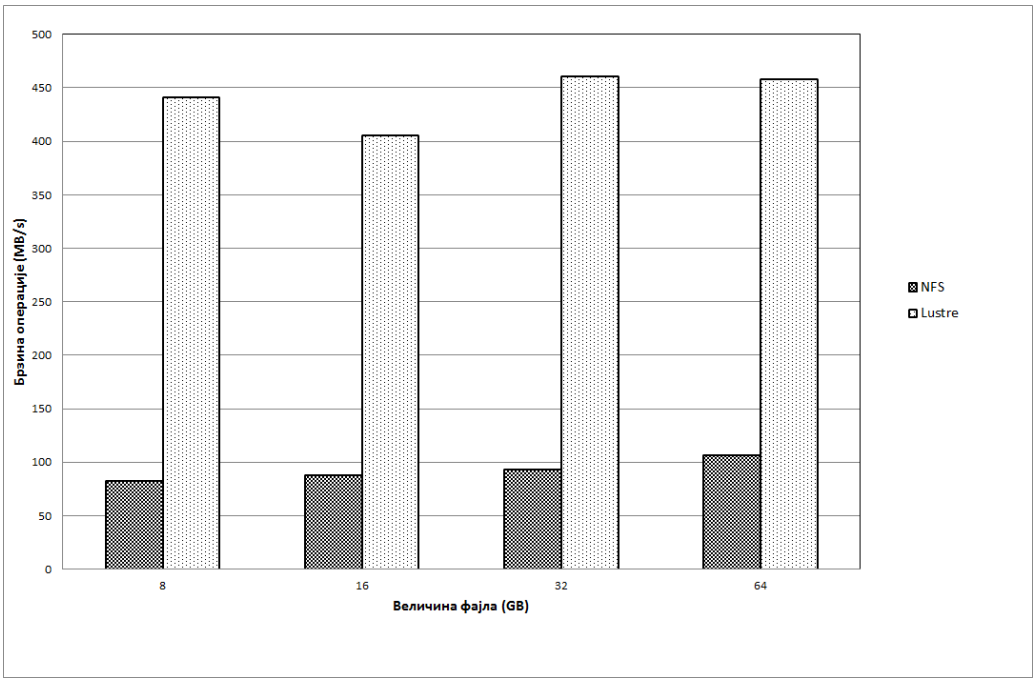
Добијени су следећи резултати:



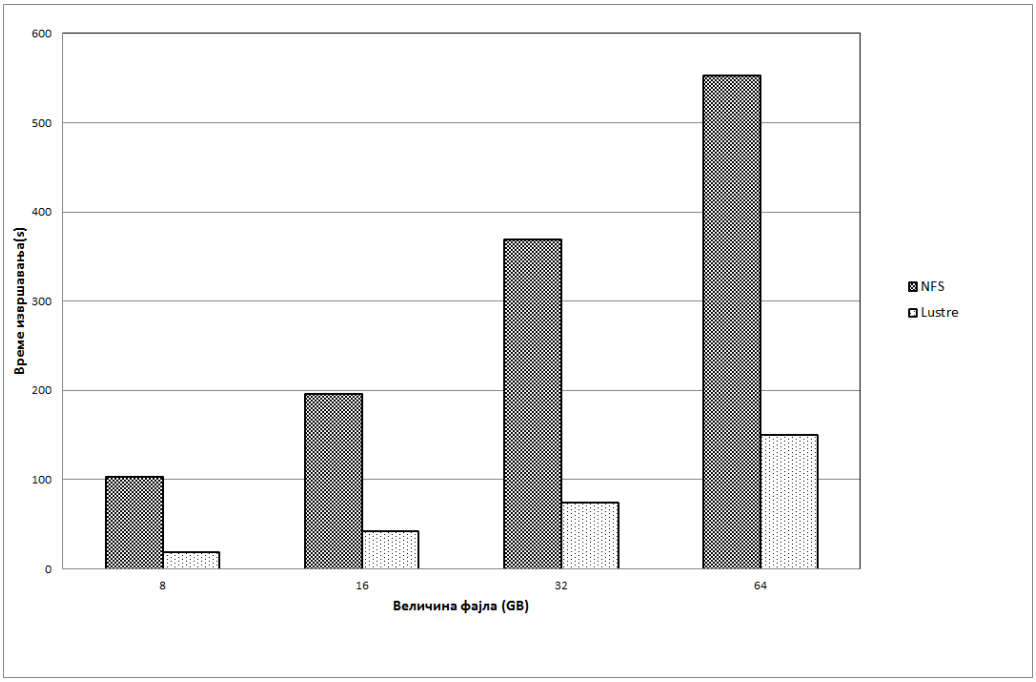

Слика 4.3: Резултати брзине читања




Слика 4.4: Резултати времена извршавања програма читања




Слика 4.5: Резултати брзине писања




Слика 4.6: Резултати времена извршавања програма писања

Анализирајући слике 4.3 и 4.5 на којима је приказана брзина читања, односно писања, закључује се да је *Lustre* фајл систем вишеструко бржи, независно од величине фајла који се чита, односно који се уписује. На сликама 4.4 и 4.6, које приказују време извршавања програма читања, односно писања, уочава се да је разлика у овим фајл системима значај-

нија уколико је величина фајла већа. Што је фајл већи, *Lustre* показује већу супериорност у односу на NFS.

## 4.4 Game of Life

*Game of Life* је најпознатији пример ћелијског аутомата који је осмислио британски математичар Џон Конвеј 1970. године. Ова игра је игра без играча, што значи да је њена еволуција одређена првобитним стањем. За тестирање је коришћен *Game of life* MPI-2 програм, коме су дадате функције читања и уписа стања ћелија, које се позивају након сваког корака симулације. Програм је покретан на *Lustre* фајл систему, а затим са истим улазним параметрима на NFS фајл систему. *Game of Life* је погодан као алатка за тестирање, јер не захтева додатне корисничке уносе након покретања програма. Матрицу помоћу које се прате стања ћелија могуће је декомпоновати независно од броја процеса. Подешавајући улазне параметре, мења се и величина података коју је потребно уписати у фајл.

### 4.4.1 Правила

*Game of Life* је представљена бесконачном дводимензионалном мрежом квадратних ћелија од којих је свака у једном од два могућа стања: активном или пасивном. Свака ћелија је у односу са осам суседних ћелија, које су са њом повезане хоризонтално, вертикално или дијагонално. Са сваким кораком у времену јављају се следеће транзиције:

- Било која активна ћелија која има мање од две активне ћелије које су јој суседне постаје пасивна.
- Било која жива ћелија са више од три активне ћелије које су јој суседне постаје пасивна да не би дошло до пренатрпаности.
- Било која ћелија са две или три активне суседне ћелије живи, не мења се и преноси на следећу генерацију.
- Било која пасивна ћелија која има три активне суседне ћелије постаће и сама активна.

Почетни модел представља почетну популацију система. Свака популација је чиста функција претходне. Правило се наставља примењивати узастопно ради креирања наредних генерација.

### 4.4.2 Порекло

Конвеј је био заинтересован за проблем представљен четрдесетих година двадесетог века од стране реномираног математичара Џона ван Нојмана, који је покушавао да пронађе машину која би могла да изради копије себе и успео је када је пронашао математички модел за такву машину са веома компликованим правилима на правоугаоној мрежи. Игра је доживела своје прво јавно појављивање у октобру 1970-е у колумни „Математичке игре“ под називом „Фантастичне комбинације Џона Конвеја“. Са теоретске тачке гледишта је занимљива, јер има моћ универзалне Тјурингове машине. Све што се може алгоритамски обрачунавати, може се израчунати и са Конвејевом *Game of Life*.

Још од њеног објављивања, *Game of Life* је привукла велико интересовање, због изненађујућих начина на који се обрасци могу развијати. Занимљиво је за физичаре, биологе, економисте, математичаре, филозофе, научнике и остале да посматрају начин на који се сложени обрасци могу појавити из примене веома једноставних правила. На пример, филозоф и научник Даниел Ц. Денет је користио Конвејеве *Game of Life* интензивно да илуструје могућу еволуцију сложених филозофских конструкција, као што су свест и слободна воља,

од релативно једноставног скупа детерминистичких физичких закона који регулишу наш сопствени универзум. Популарност Конвејеве игре је потпомогнута њеном појавом баш у време појаве нове генерације јефтиних мини рачунара који су пуштени у промет. Игра може да се активира ноћу, током сати када су машине иначе неискоришћене. За многе, *Game of Life* је једноставно програмирање, изазов, забаван начин да губимо циклусе процесора. За неке, међутим, *Game of Life* поседује и филозофске конотације.

## 4.5 Опис програма

Програм је написан у програмском језику C користећи MPI функције. На почетку програма се учитавају параметри програма.

Улазни параметри програма су:

- начин генерисања почетне популације,
- број процеса,
- величина квадратне матрице,
- број итерација,
- начин уписа - уколико је "0", онда се матрица уписује у један једини фајл. Уколико је "1", онда сваки процес свој део матрице уписује у сопствени фајл.

Свака ћелија је представљена као поље у матрици. На основу унетих параметара, креира се и генерише почетна популација. У свакој итерацији се на основу објашњених правила рачуна вредност поља у матрици, а затим се, на основу начина уписа, резултат уписује у фајл или фајлове. Као резултат извршења програма, добија се време које је потребно да се програм изврши. Дакле, стандардна апликација је модификована тако да у свакој итерацији уписује своје стање у фајл или фајлове. Оваква апликација ће послужити као погодан општи тест перформанси, јер, за разлику од претходна два алата поседује и делове у којима се интензивно рачуна, као и делове са интензивним улазно/излазним операцијама.

На кластеру програм се покреће помоћу следеће скрипте:

**Листинг 4.2: Скрипта за покретање програма на кластеру**

```
#!/bin/sh
#PBS -N lustre_mpi
#PBS -q batch
#PBS -l nodes=8:ppn=8

module load openmpi-pmf-x86_64
chmod 755 mpi_lustre
mpirun ./mpi_lustre random 128 128 1
```

Упис и читање из фајла се врши у свакој итерацији. У зависности од начина уписа, потребно је отворити фајл или фајлове за читање и писање. Уколико је `write_type` једнак нули онда сви процеси врше операције са једном фајлом, у супротном сваки процес има посебан фајл из ког чита и у који уписује.

**Листинг 4.3: Део кода за отварање фајлова за читање и писање**

```
MPI_File thefile;
int nnp, *myold;
MPI_Status status;

  if (write_type == 0)
  {
      MPI_File_open(MPI_COMM_WORLD, filename,
```

```
                        MPI_MODE_CREATE | MPI_MODE_RDWR,
                        MPI_INFO_NULL, &thefile);

            MPI_File_set_view(thefile, nnp * id * sizeof(int),
                              MPI_INT, MPI_INT, "native", MPI_INFO_NULL)
                                  ;
        }
        else
        {
            MPI_File_open(MPI_COMM_SELF, filename,
                        MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL
                            , &thefile);
        }
```

**Листинг 4.4: Функције за читање и писање у фајл**

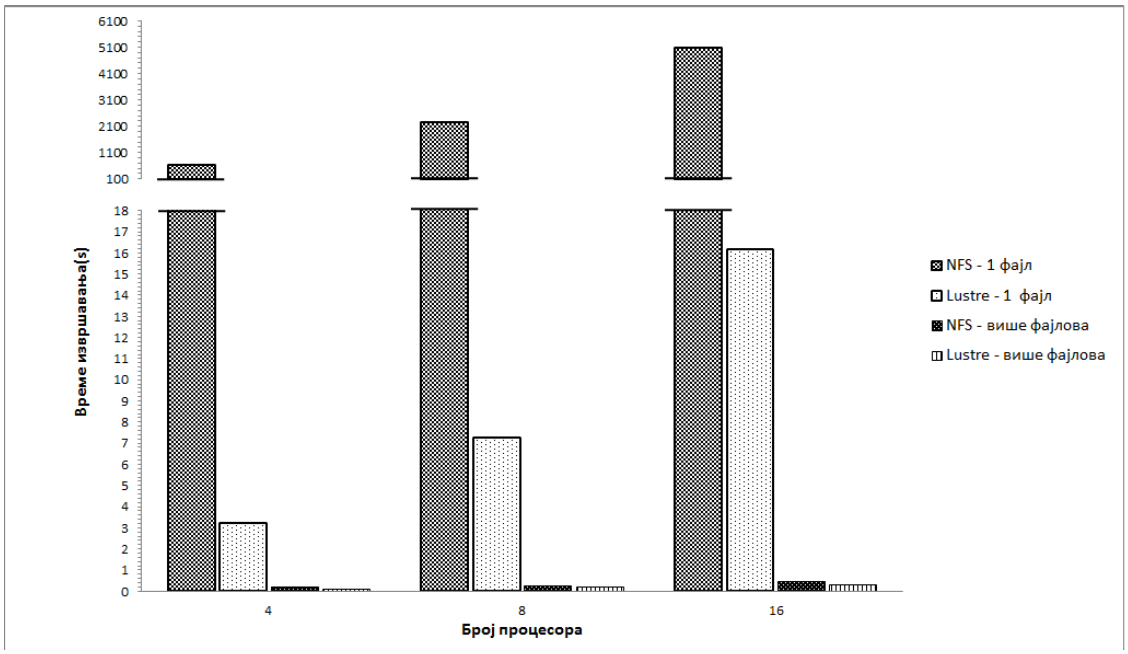
```
    MPI_File_read(thefile, myold, nnp, MPI_INT, &status);

    MPI_File_write(thefile, myold, nnp, MPI_INT, MPI_STATUS_IGNORE);
```

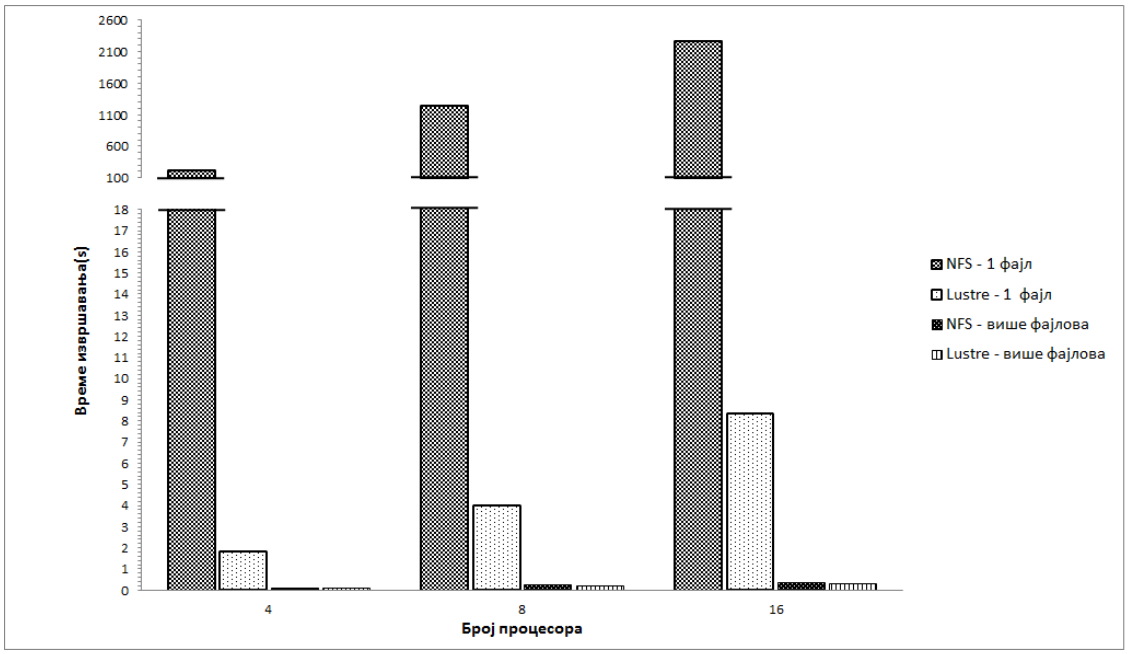
Садржај низа `myold` чита или уписује у фајл `thefile`.

## 4.6 Резултати тестирања

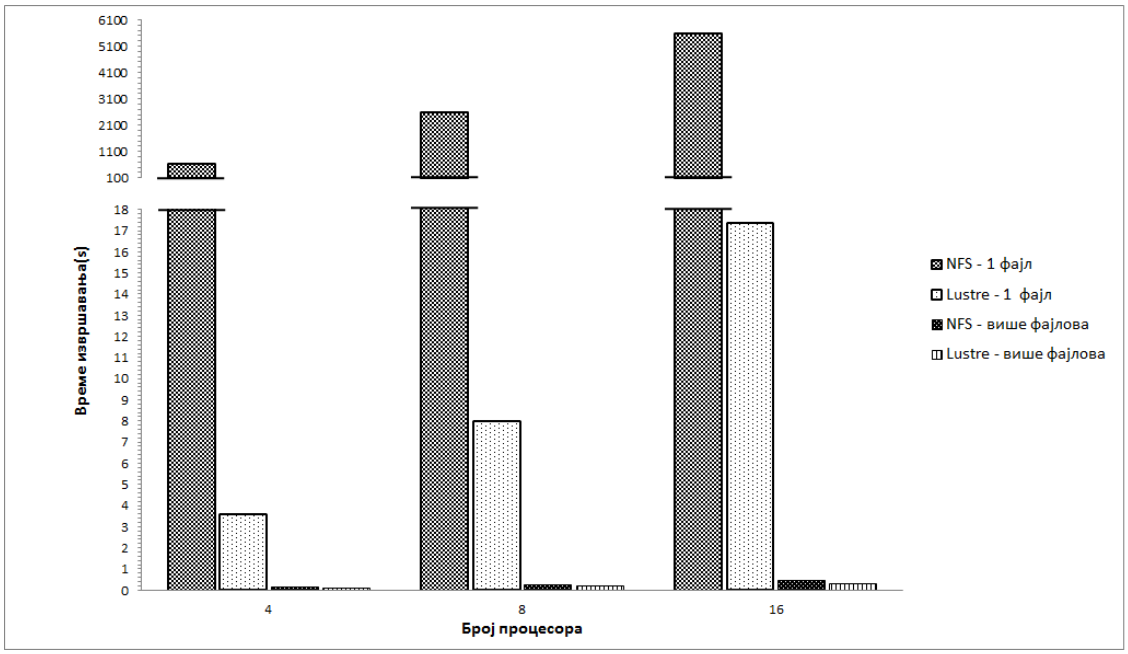
На графиконима је приказано време извршавања у зависности од улазних параметара.



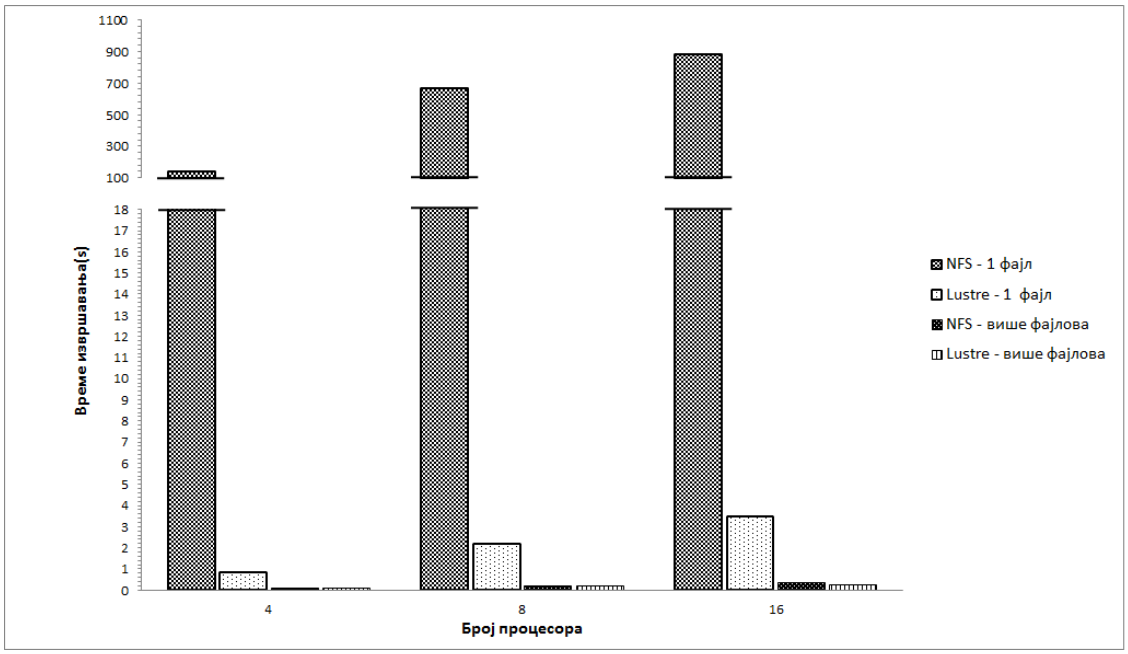

Слика 4.7: Матрица 32x32, 32 итерација



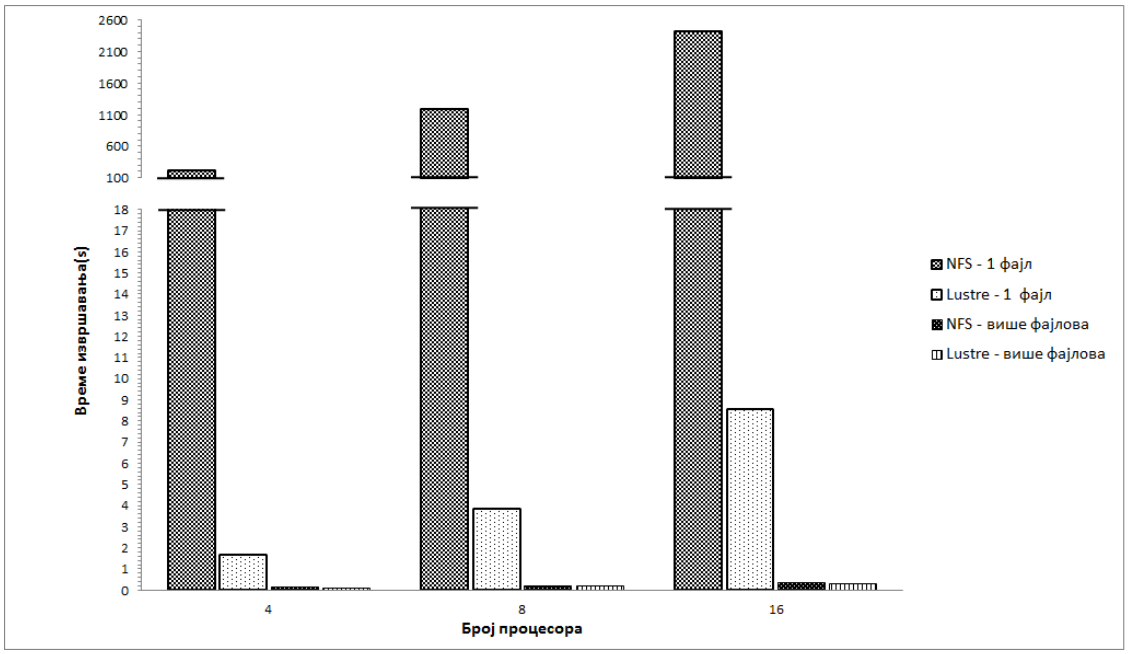

Слика 4.8: Матрица 32x32, 64 итерација



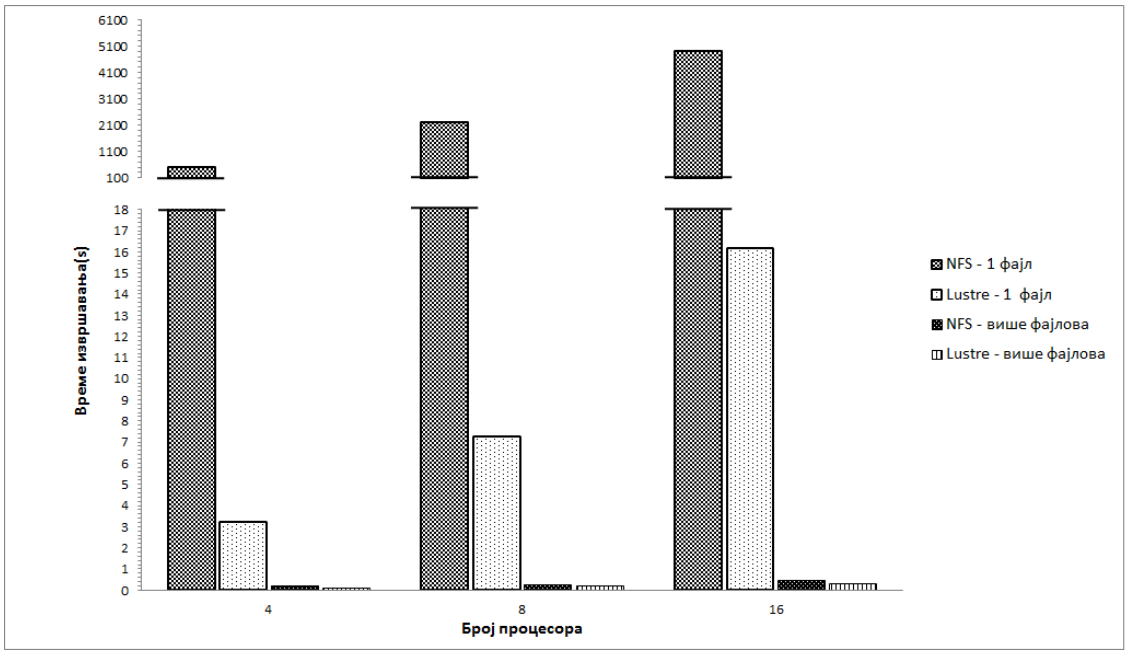

Слика 4.9: Матрица 32x32, 128 итерација



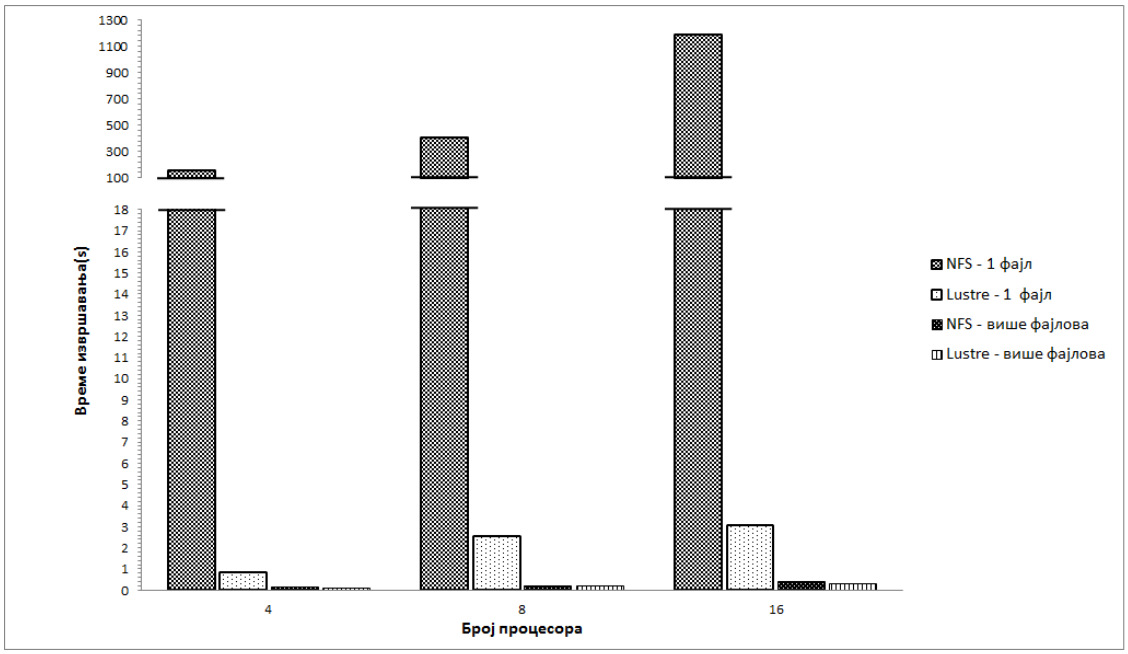

Слика 4.10: Матрица 64x64, 32 итерација



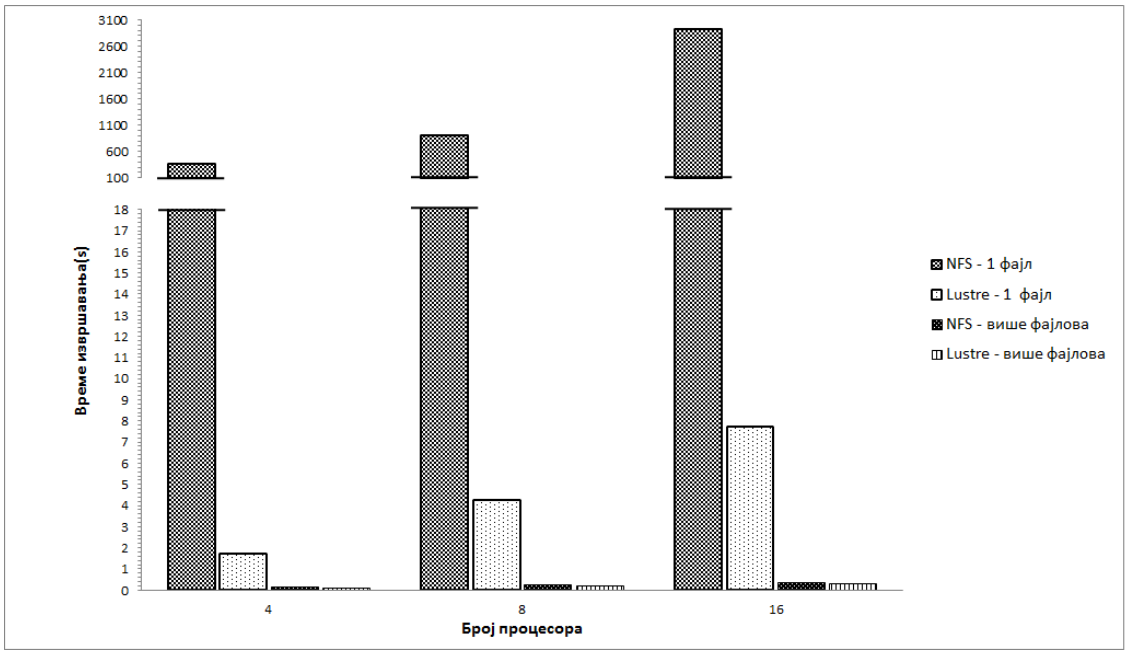

Слика 4.11: Матрица 64x64, 64 итерација



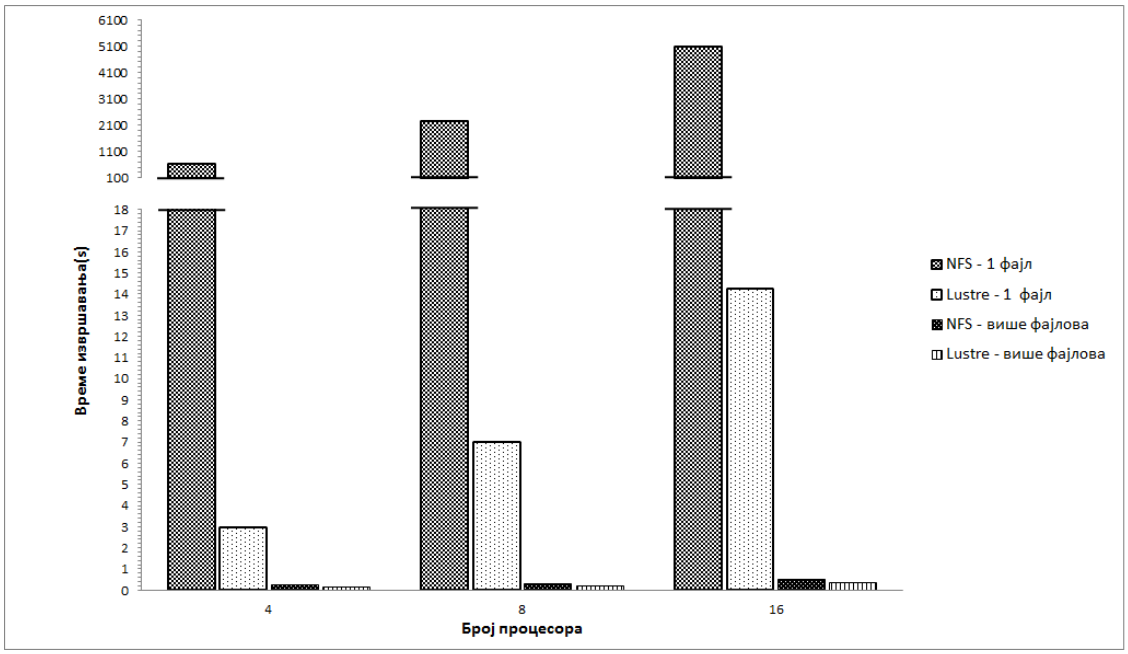

Слика 4.12: Матрица 64x64, 128 итерација




Слика 4.13: Матрица 128x128, 32 итерација




Слика 4.14: Матрица 128x128, 64 итерација




Слика 4.15: Матрица 128x128, 128 итерација

Од слике 4.7 до 4.15 приказано је време извршавања програма *Game of Life* у зависности од комбинације параметара као што су: број процеса, величине матрице, број итерација и начин коришћења улазно/излазних операција. Уочава се да је време извршавања значајно мање код *Lustre* фајл система, посебно када сваки процес има засебан фајл. Када сви процеси користе један фајл, *Lustre* фајл систему је опет потребно мање времена него NFS фајл систему. Повећањем броја процеса ова разлика постаје још уочљивија. Што је већа матрица и већи број итерација, то је количина података већа. Читањем и писањем већег броја података, *Lustre* показује већу ефикасност.

# Глава 5

# Закључак

Тестирањем брзина улазно/излазних операција код *Lustre* фајл система и NFS-а, добијени су резултати који показују да је *Lustre* фајл систем далеко бржи у свим типовима улазно-излазних операција. Главни недостатак *Lustre* фајл система је нешто компликована инсталација, конфигурација и одржавање у односу на његову алтернативу NFS. Међутим, уштеда времена у току вршења тестирања је огромна што се никако не сме занемарити. Та предност је нарочито значајна при извршавању обимних математичких операција које захтевају рад са фајловима. MPI-2 стандард са новим функцијама које омогућавају паралелне улазно/излазне операције заједно са *Lustre* фајл системом чине најбољу комбинацију за развој савремених апликација високих перформанси. Обзиром да је *Lustre* фајл систем доступан бесплатно, да је његов опоравак лакши у случају отказивања чврстог диска, као и да подржава паралелне операције са фајловима, треба му се дати апсолутна предност у односу на NFS. Једина алтернатива коју *Lustre* тренутно има на тржишту је *PanFS* компаније *Panasas*, који се нешто лакше конфигурише и администрира, али по цену затворености кода и више цене.

# Библиографија


[1] http://stormerider.com/wp-content/uploads/2011/05/820-7390.pdf, март 2014

[2] http://docs.oracle.com/cd/E19527-01/821-2076-10/821-2076-10.pdf, мај 2014

[3] http://linux.cloudibee.com/2009/11/lustre-cluster-filesystem-quick-setup-guide/, октобар 2014

[4] https://www.kernel.org/doc/ols/2003/ols2003-pages-380-386.pdf, октобар 2014

[5] http://imi.pmf.kg.ac.rs/moodle/file.php/127/Materijali_za_vezbe/2013_14_Primeri/13_GOL.c, мај 2015

[6] http://www.conwaylife.com/wiki/Conway's_Game_of_Life, мај 2015

[7] Using MPI-2 Advanced Features of the Message-Passing Interface, William Gropp, Ewing Lusk, Rajeev Thakur, 1999

[8] MPI – the Complete Reference. Vol. 2, The MPI-2 Extensions Scientific and Engineering Computation Series Gropp, William. MIT Press, 1998

[9] MPI-2: Extensions to the Message-Passing Interface - Message Passing Interface Forum, 2009

[10] Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting, Paris France, 2007