МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

РОЗПОДІЛЕНІ ТА ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

Лабораторний практикум

для студентів спеціальності
121 "Інженерія програмного забезпечення"
освітньої програми
"Інженерія програмного забезпечення"
першого (бакалаврського) рівня

Харків
ХНЕУ ім. С. Кузнеця
2023

УДК 004.4'2(076.034)
Р65

Укладачі: С. В. Мінухін
Ю. В. Савін

Затверджено на засіданні кафедри інформаційних систем.
Протокол № 8 від 21.11.2022 р.

Самостійне електронне текстове мережеве видання

Розподілені та паралельні обчислення [Електронний ресурс] :
Р65 лабораторний практикум для студентів спеціальності 121 "Інженерія програмного забезпечення" освітньої програми "Інженерія програмного забезпечення" першого (бакалаврського) рівня / уклад. С. В. Мінухін, Ю. В. Савін. – Харків : ХНЕУ ім. С. Кузнеця, 2023. – 171 с.

Подано методичні рекомендації до виконання лабораторних робіт, метою яких є закріплення й поглиблення знань теоретичного матеріалу, набуття навичок зі створення високопродуктивного обчислювального кластера та розроблення паралельних програм. У кожній лабораторній роботі визначено мету, завдання, засоби та порядок виконання, зміст звіту й контрольні запитання.

Рекомендовано для студентів спеціальності 121 "Інженерія програмного забезпечення" першого (бакалаврського) рівня.

УДК 004.4'2(076.034)

Вступ

Умови зростання обсягів даних і збільшення залежності бізнес-процесів підприємств від потоків даних визначають потреби створення розподілених інформаційних систем (РІС) різних рівнів, які повинні забезпечити достатній рівень оперативності оброблення даних в умовах масштабованості систем та збільшення інтенсивності даних на оброблення. Такі завдання вирішують на основі розроблення РІС, що використовують відповідні технології та програмне забезпечення відповідно до архітектур розподілених обчислювальних систем (РОС) з використанням паралельних технологій програмування.

Технології розподілених систем та паралельних обчислень є основою побудови розподілених ІС від рівня обчислювального кластера до рівня грид-систем і систем хмарних обчислень. Принципи побудови, методи та технології створення, розгортання РОС та застосування паралельних обчислень є основою для розв'язання складних з точки зору обчислень трудомістких задач у різних предметних областях науки, а також інженерних задач.

Метою викладання навчальної дисципліни "Розподілені та паралельні обчислення" є формування системи теоретичних знань і набуття практичних умінь і навичок з питань використання технологій РОС, встановлення та налаштування відповідного програмного забезпечення запуску та виконання завдань на обчислювальному кластері та використання технологій і засобів паралельного програмування.

Відповідно до ОПП підготовки бакалаврів навчальна дисципліна має формувати такі компетентності:

здатність до алгоритмічного та логічного мислення;

здатність до абстрактного мислення, аналізу та синтезу;

здатність застосовувати знання у практичних ситуаціях;

здатність до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проєктування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та складності, розв'язності та нерозв'язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем;

здатність застосовувати теоретичні та практичні основи методології та технології моделювання для дослідження характеристик і поведінки

складних об'єктів і систем, проводити обчислювальні експерименти з обробленням й аналізом результатів;

здатність реалізовувати високопродуктивні обчислення на основі кластерних обчислювальних систем шляхом установлення та налаштовування відповідного програмного забезпечення для планування та розподілу завдань в пакетному та інтерактивному режимах, використання паралельних і розподілених обчислень під час розроблення й експлуатації розподілених систем з паралельним обробленням даних.

Під час навчання за навчальною дисципліною студенти отримують такі основні результати навчання:

виконувати паралельні та розподілені обчислення з використанням програмного забезпечення кластерних обчислювальних систем, застосовувати чисельні методи та алгоритми для їх застосування в паралельних архітектурах високопродуктивних систем, мови паралельного програмування під час розроблення та експлуатації паралельного та розподіленого програмного забезпечення в цих системах;

володіти мовами системного програмування та методами розроблення програм, що взаємодіють з компонентами комп'ютерних систем, знати мережні технології, архітектури комп'ютерних мереж, мати практичні навички технології адміністрування комп'ютерних мереж та їх програмного забезпечення;

розробляти програмні моделі предметних середовищ, обирати парадигму програмування з позицій зручності та якості застосування для реалізації методів та алгоритмів розв'язання задач у галузі комп'ютерних наук;

застосовувати знання основних форм і законів абстрактно-логічного мислення, основ методології наукового пізнання, форм і методів вилучення, аналізу, оброблення та синтезу інформації.

Методичні рекомендації до виконання лабораторних робіт складені відповідно до затвердженої робочої програми з навчальної дисципліни та охоплюють практичні питання щодо застосування розподілених обчислень в різних предметних областях. Лабораторний практикум складається із п'яти лабораторних робіт. За результатами виконання кожної роботи студент має оформити електронний звіт засобами *Word*. У процесі виконання лабораторних робіт треба вирішити тестові приклади та індивідуальні завдання. У висновках з лабораторної роботи слід подати, які знання, вміння та навички були отримані під час її виконання.

Лабораторна робота 1
Створення та налаштування вузлів обчислювального кластера

Мета лабораторної роботи:

1. Набуття практичних навичок зі встановлення операційної системи (ОС) *Linux* на віртуальну машину (ВМ).
2. Набуття практичних навичок з налаштування ОС *Linux*.
3. Набуття практичних навичок з налаштування комп'ютерної мережі (КМ), та сервісів кластера в ОС *Linux* та перевірка її роботи.

Завдання:

1. Створити три ВМ та встановити на них ОС *Linux (Rocky Linux 9)*.
2. Налаштувати КМ між ВМ. Перевірити роботу КМ.
3. Налаштувати протоколи SSH та NFS.

Порядок виконання лабораторної роботи

Для виконання лабораторної роботи треба встановити *Oracle VirtualBox Manager* (рис. 1.1), з установленим пакетом VM *VirtualBox Extension Pack* (рис. 1.2).

Створити три ВМ: головний сервер (ГС) та два обчислювальних вузли (ОВ) кластера. Провести їх налаштування.

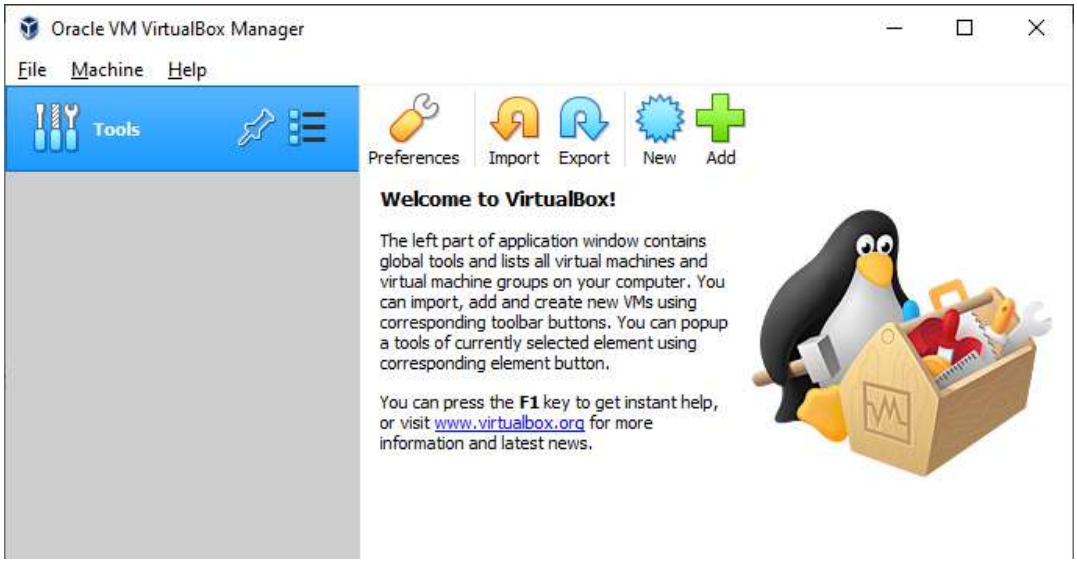

Рис. 1.1. **ВМ Oracle VM VirtualBox Manager**

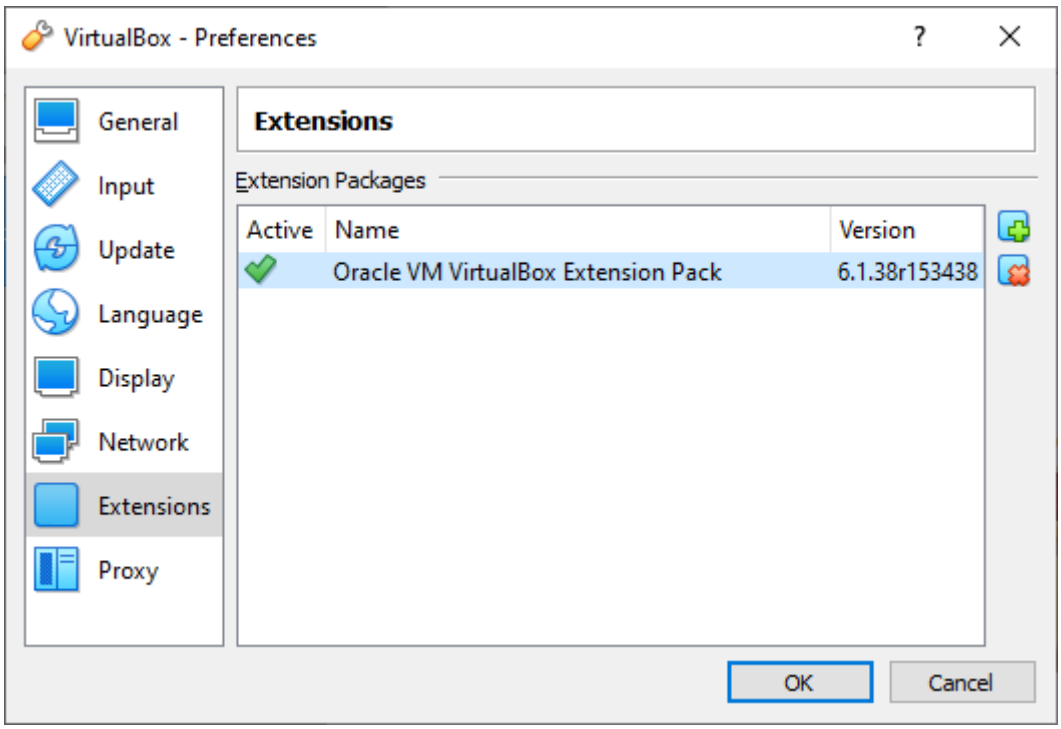

Рис. 1.2. **Пакет розширень Oracle VM VirtualBox Extension Pack**

Установлення та налаштування ОС *Linux* на ВМ

Для створення ВМ натисніть кнопку *New* (див. рис. 1.1). У вікні, що відкрилося, потрібно задати назву та тип ОС (рис. 1.3).

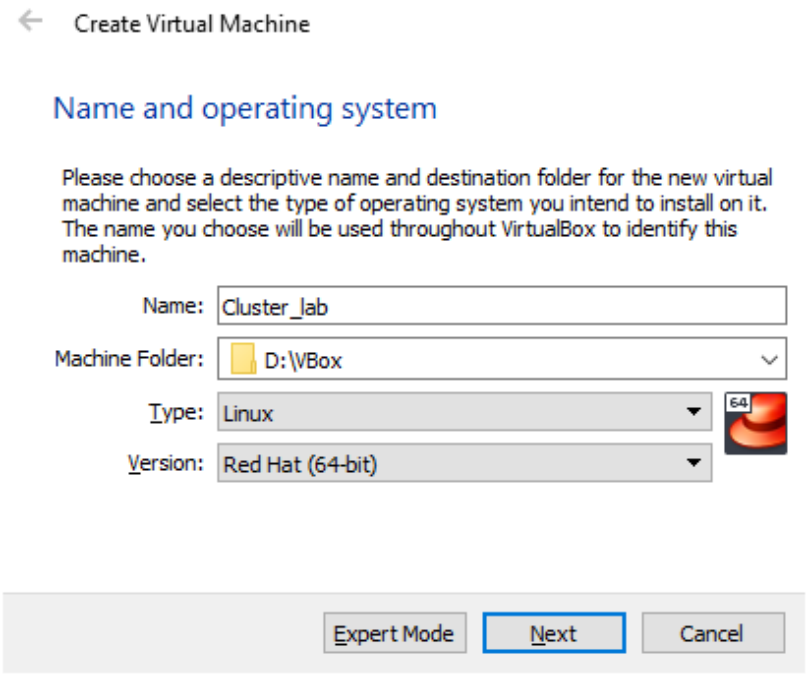

Рис. 1.3. **Вікно створення ВМ**

Указати розмір оперативної пам'яті (ОП), що виділяється для ВМ (рис. 1.4).

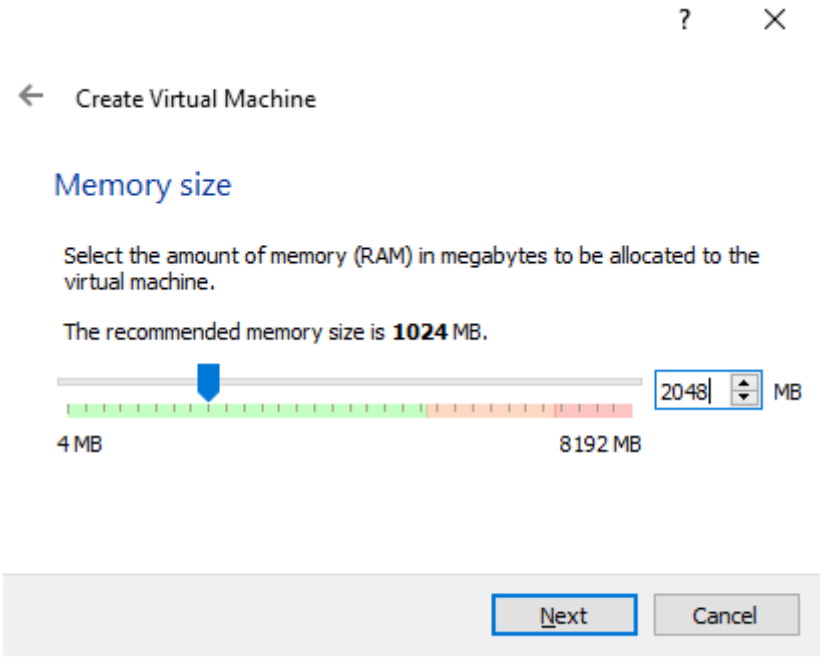

Рис. 1.4. **Вибір розміру ОП для ВМ**

Далі потрібно створити диск для ВМ, на який буде встановлено ОС (рис. 1.5).

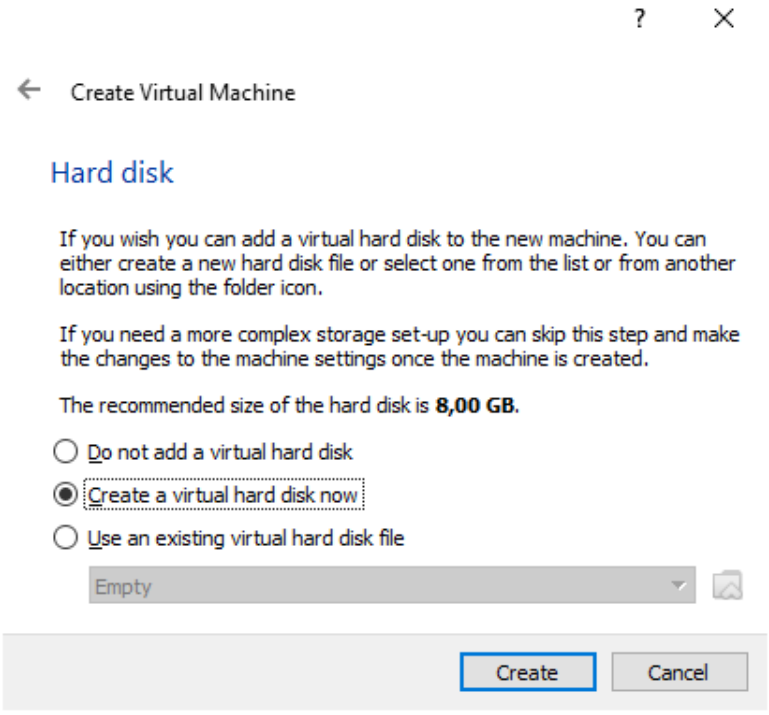

Рис. 1.5. **Створення диска для ВМ**

Обираємо тип створеного файла диска та тип розміщення (рис. 1.6 і рис. 1.7).

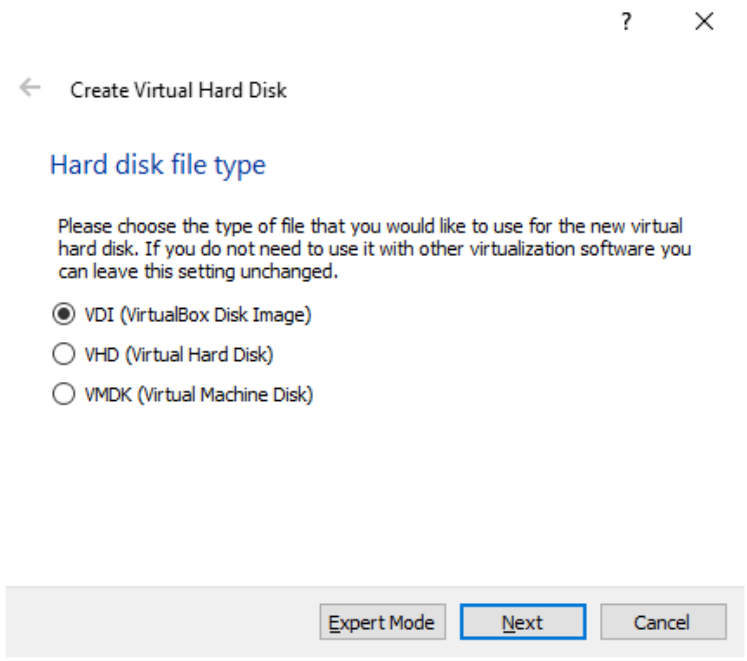

Рис. 1.6. **Вибір типу файла диска для ВМ**

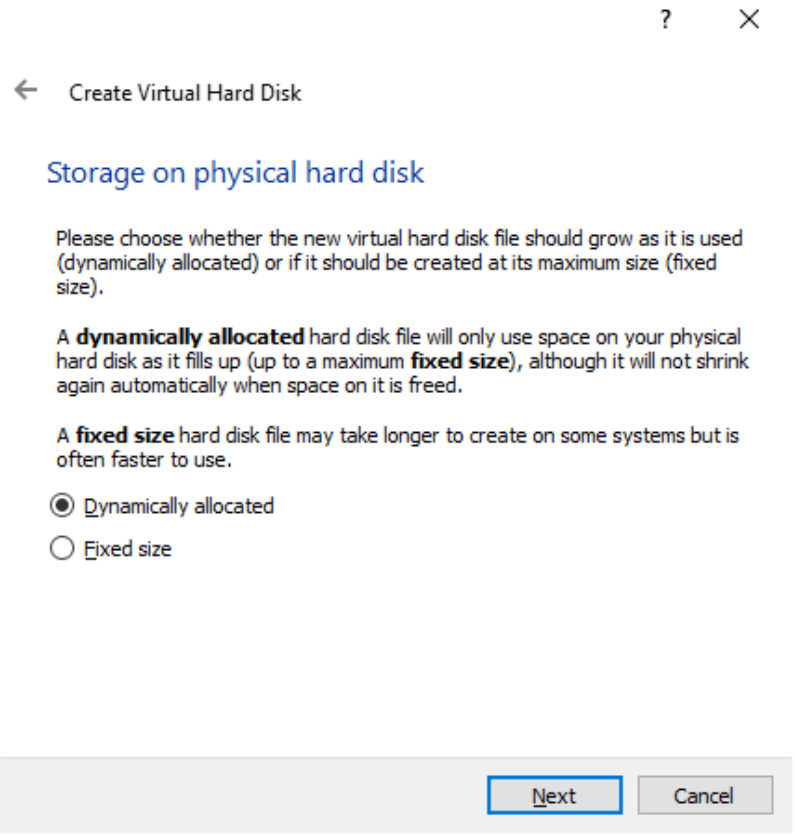

Рис. 1.7. **Вибір типу розміщення файла диска для ВМ**

Вводимо назву, шлях (місце розташування файла диска на локальному комп'ютері) та задаємо максимальний розмір файла диска, який він може зайняти на фізичному носії (рис. 1.8).

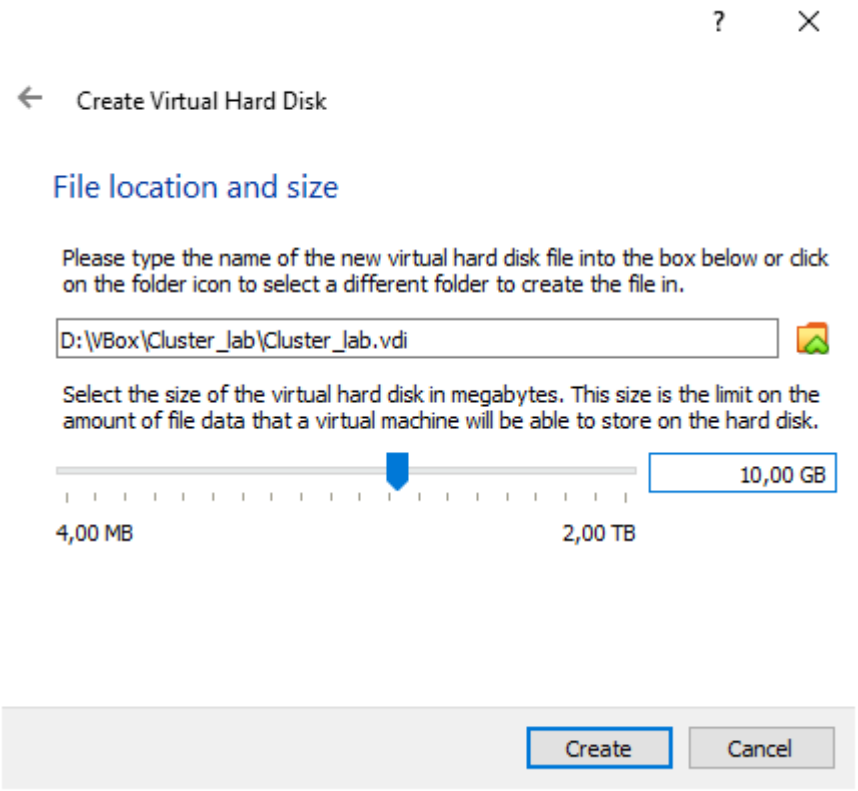

Рис. 1.8. **Вибір назви, місця розташування, розміру файла диска ВМ**

На цьому створення першої ВМ буде завершено (рис. 1.9). Далі переходимо до її налаштування (рис. 1.10).

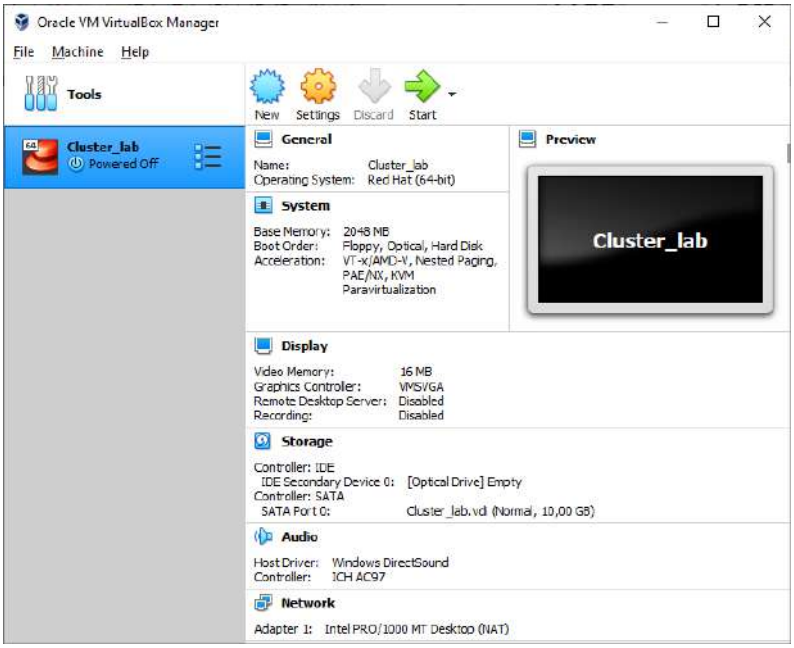

Рис. 1.9. **Створення першої ВМ**

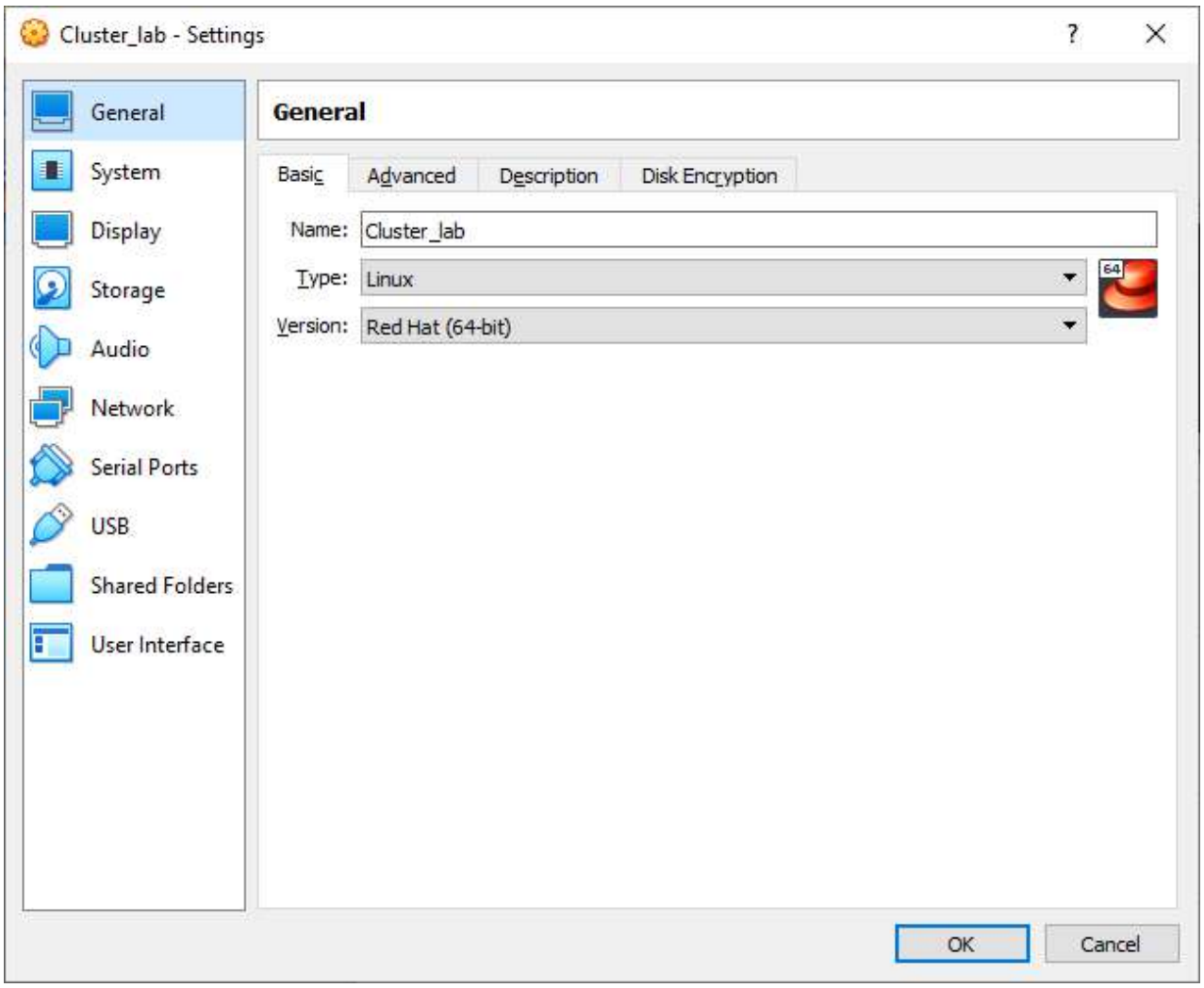

Рис. 1.10. **Панель налаштувань ВМ**

Спочатку з'ясуємо, для чого буде використовуватися перша ВМ, та з чого вона повинна складатися. Її призначення – бути ГС кластера. До його обов'язків належить: робота з користувачами, управління завданнями та моніторинг стану ОВ. Для цього на ГС повинно бути налаштовано 2 мережеві адаптери (МА). Один з яких налаштовується на зовнішню КМ (інтернет), а другий на внутрішню КМ (інтранет).

Доступу до внутрішньої КМ користувачі мати не повинні, хоча мають змогу переходити на ОВ. Але ідентифікувати, яка частка завдання виконується на яких ОВ, вони не мають можливості. Внутрішня КМ, як правило, налаштовується з політиками довіри. Це дозволяє забезпечити максимально швидке передавання даних між ГС та ОВ.

На ГС встановлюється основна частка програмного забезпечення, що висуває вимоги до його швидкодії. Тому, за можливості, бажано виділяти для роботи цієї ВМ 2 процесорних ядра та 4 ГБ ОП.

Якщо такої можливості не має, то в мінімальній конфігурації обрати одне процесорне ядро та два ГБ ОП.

Входимо в панель налаштувань ВМ, натискаючи на кнопку *Settings* (див. рис. 1.10).

Налаштовуємо кількість ядер процесора (рис. 1.11). В нашому прикладі обрано 2 ядра.

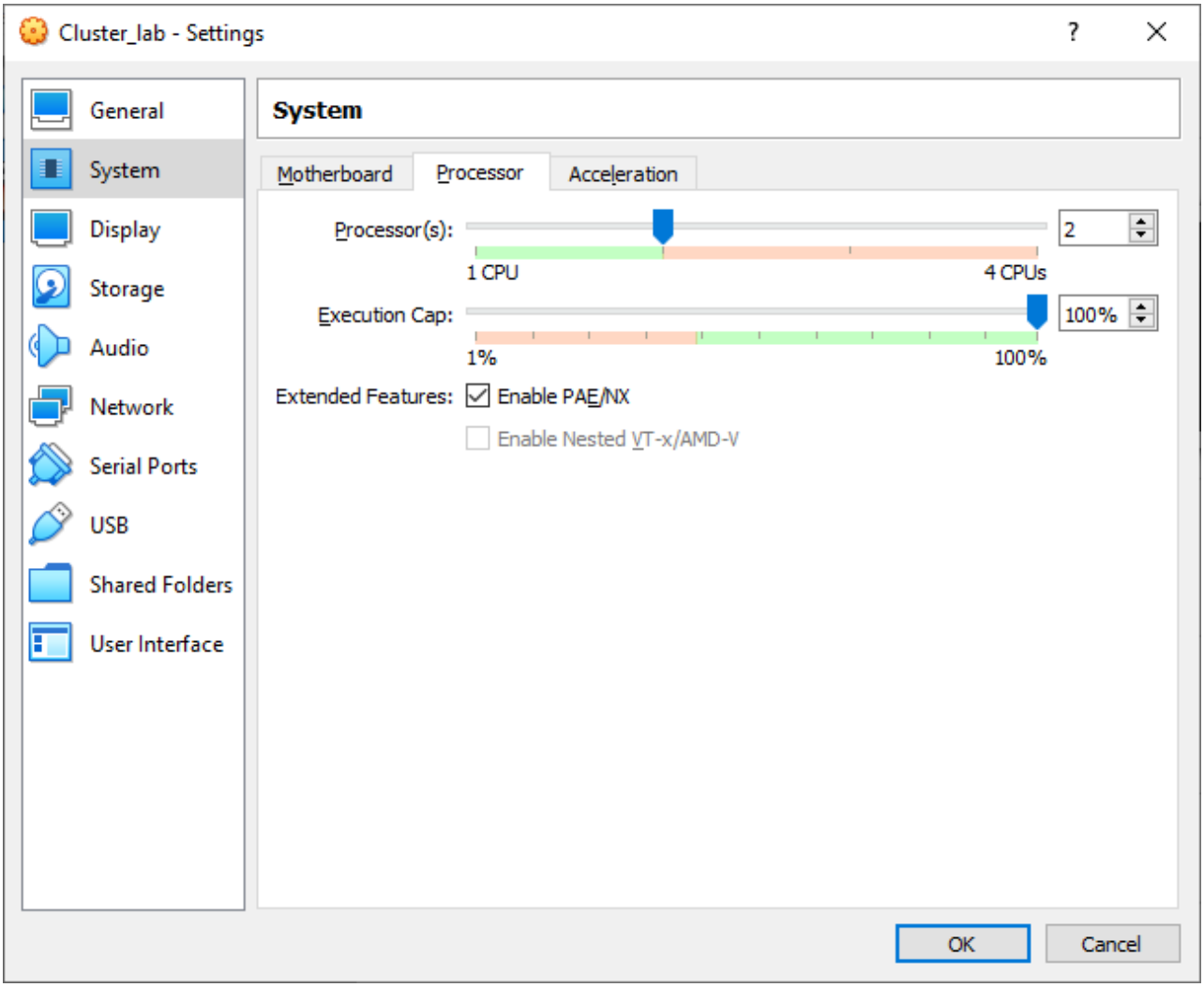

Рис. 1.11. **Налаштування кількості ядер процесора**

Уставляємо диск з образом ОС *Linux*, що буде встановлено на нашу ВМ (рис. 1.12).

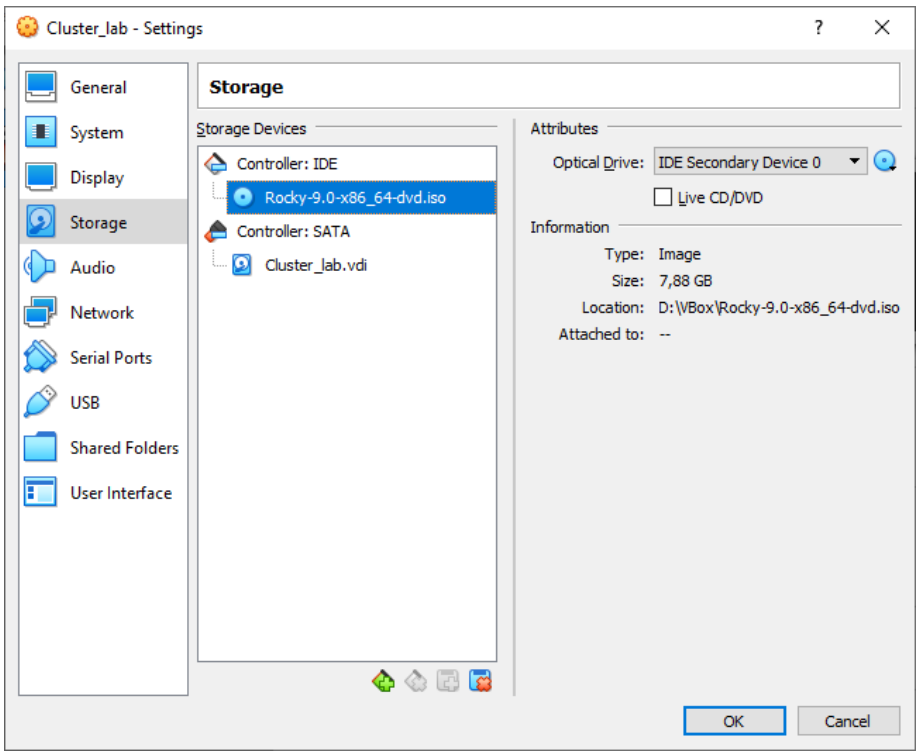

Рис. 1.12. **Установлення образу ОС *Linux***

Налаштовуємо МА. Перший МА залишаємо без змін. Другий створюємо в режимі *Host-only Adapter* (рис. 1.13).

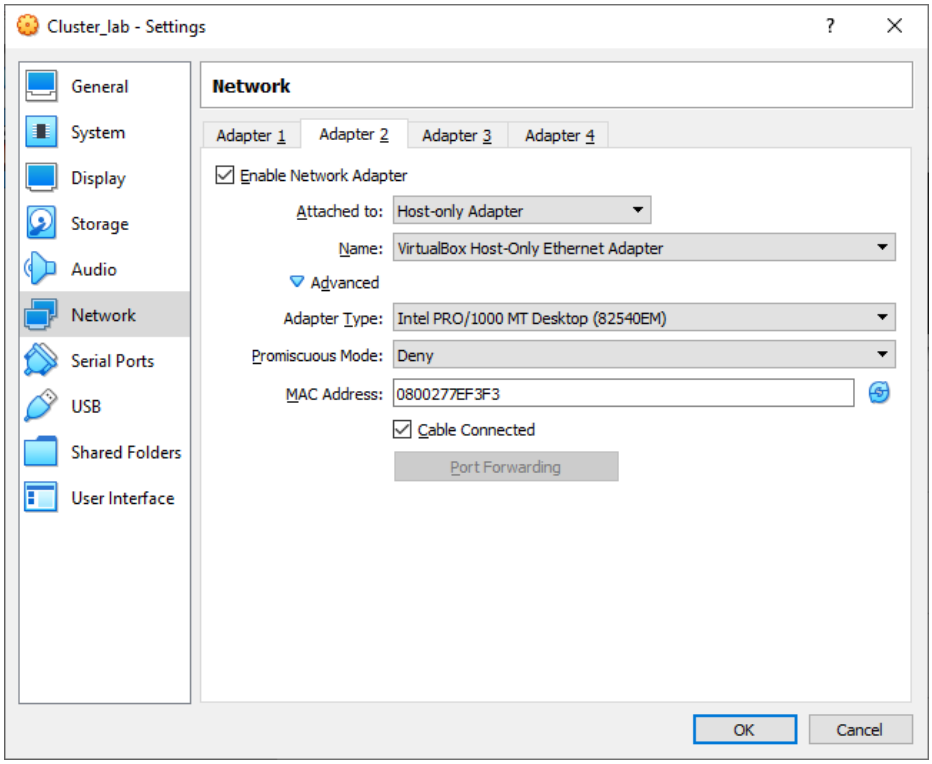

Рис. 1.13. **Налаштування МА**

На цьому налаштування ВМ завершено.

Встановлення ОС на першу ВМ

Запускаємо ВМ, натискаючи на кнопку *Start* (рис. 1.14).

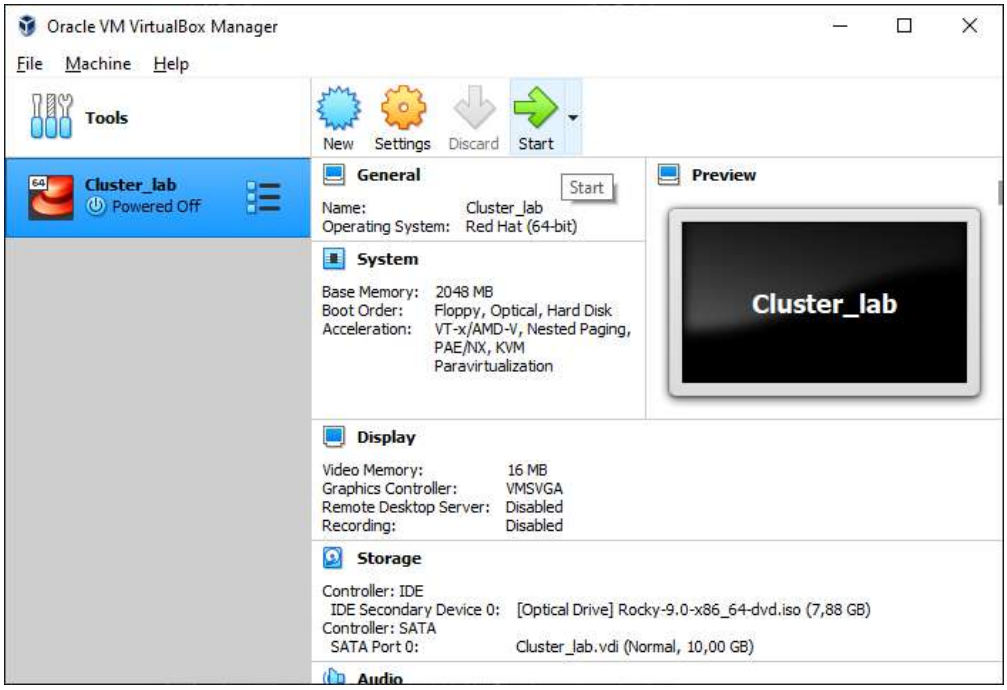

Рис. 1.14. **Запуск першої ВМ**

Обираємо встановлення ОС *Linux* (рис. 1.15).

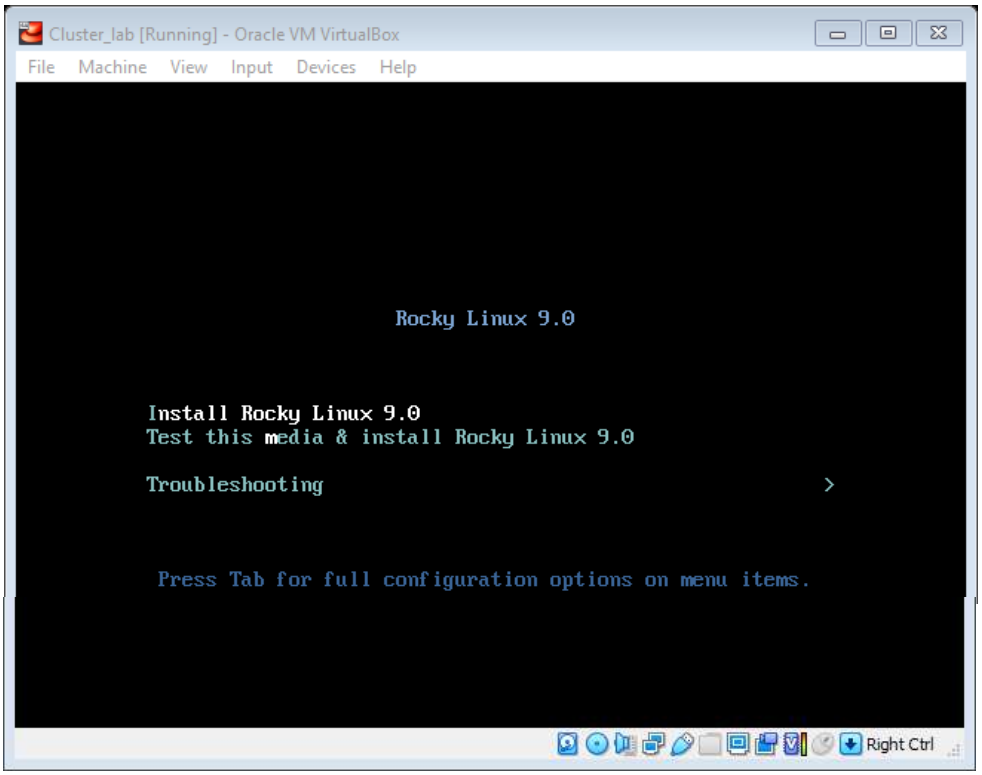

Рис. 1.15. **Вибір установки ОС**

Обираємо мову установки (рис. 1.16).

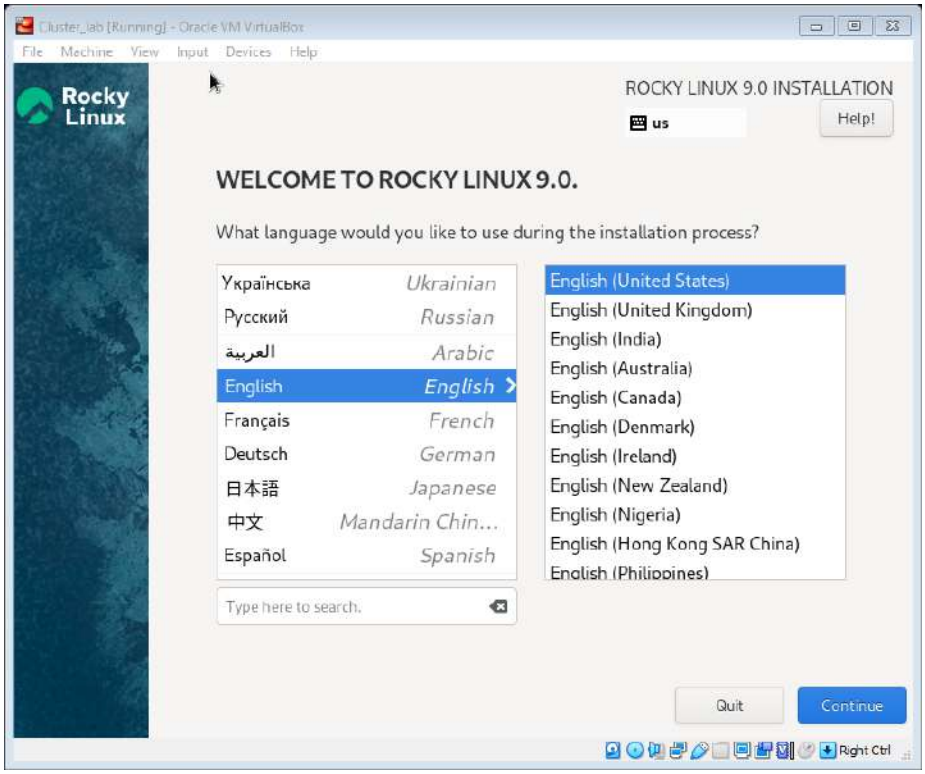

Рис. 1.16. **Вибір мови установки**

Переходимо до панелі налаштувань (рис. 1.17).

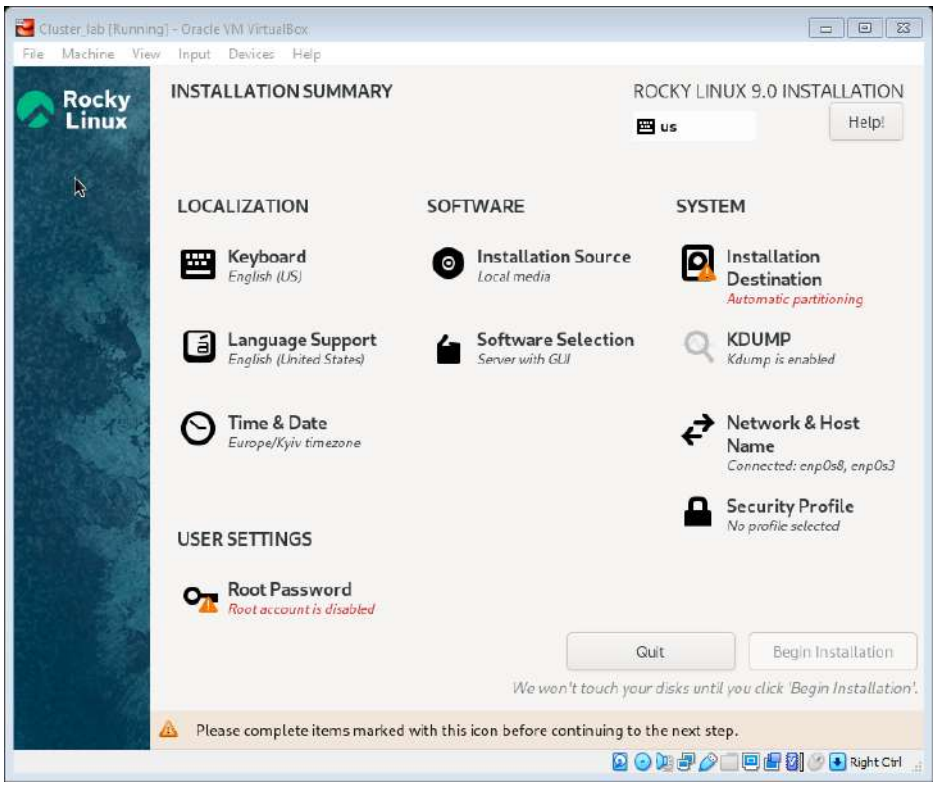

Рис. 1.17. **Панель налаштувань *Rocky Linux***

Обираємо режим установки *Server* (рис. 1.18).

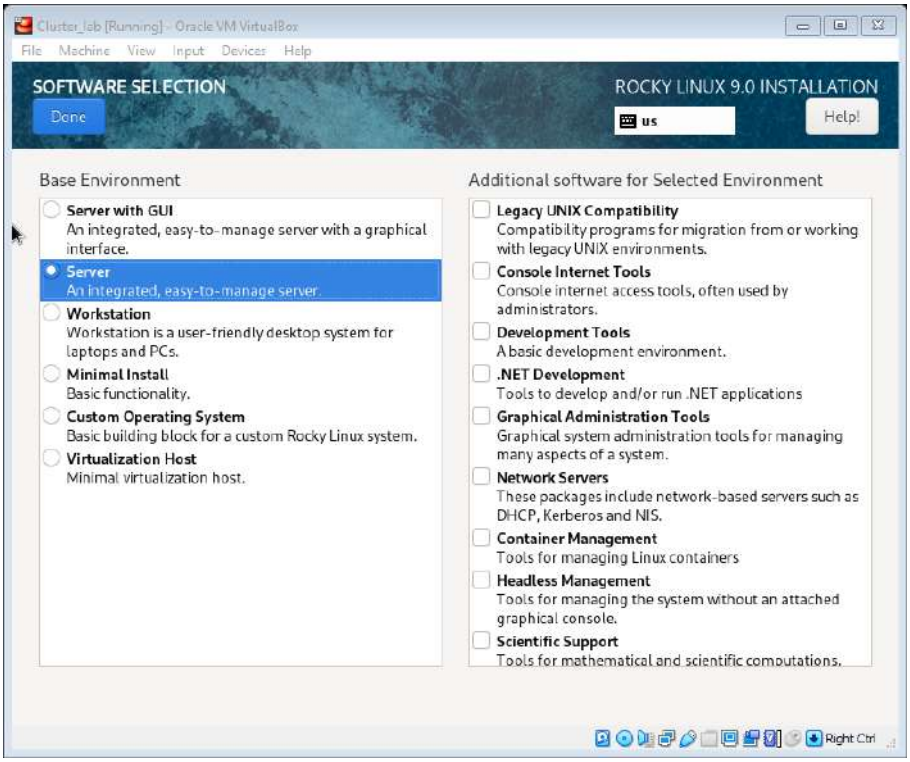

Рис. 1.18. **Вибір режиму установки Server**

Створюємо розділи диска ВМ (рис. 1.19 – 1.21).

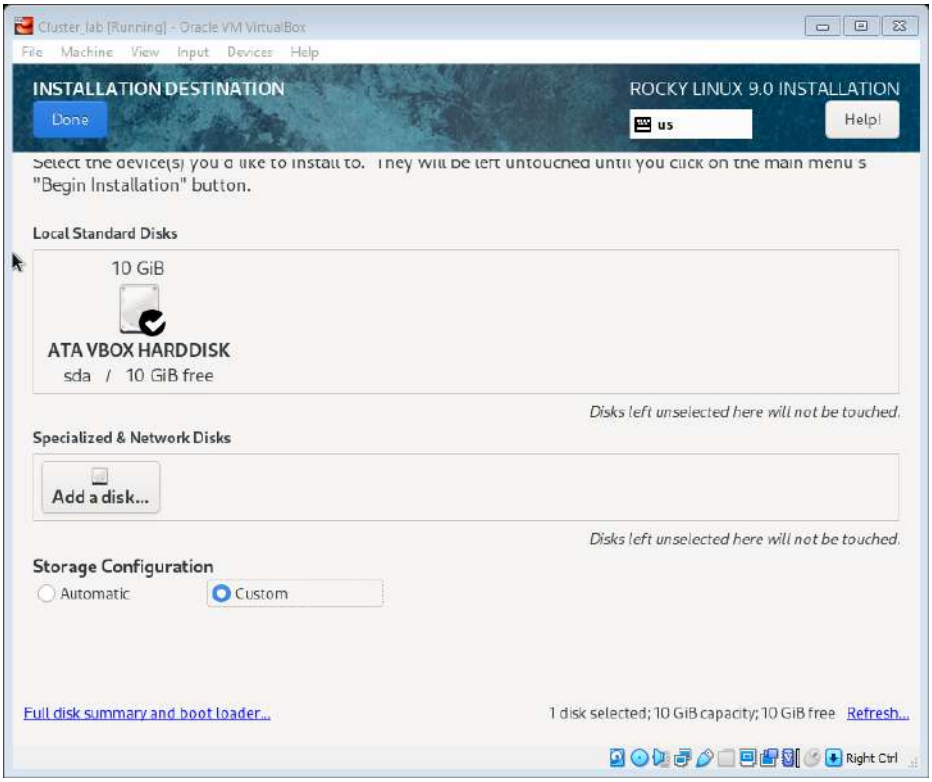

Рис. 1.19. **Вибір диска та ручного створення розділів**

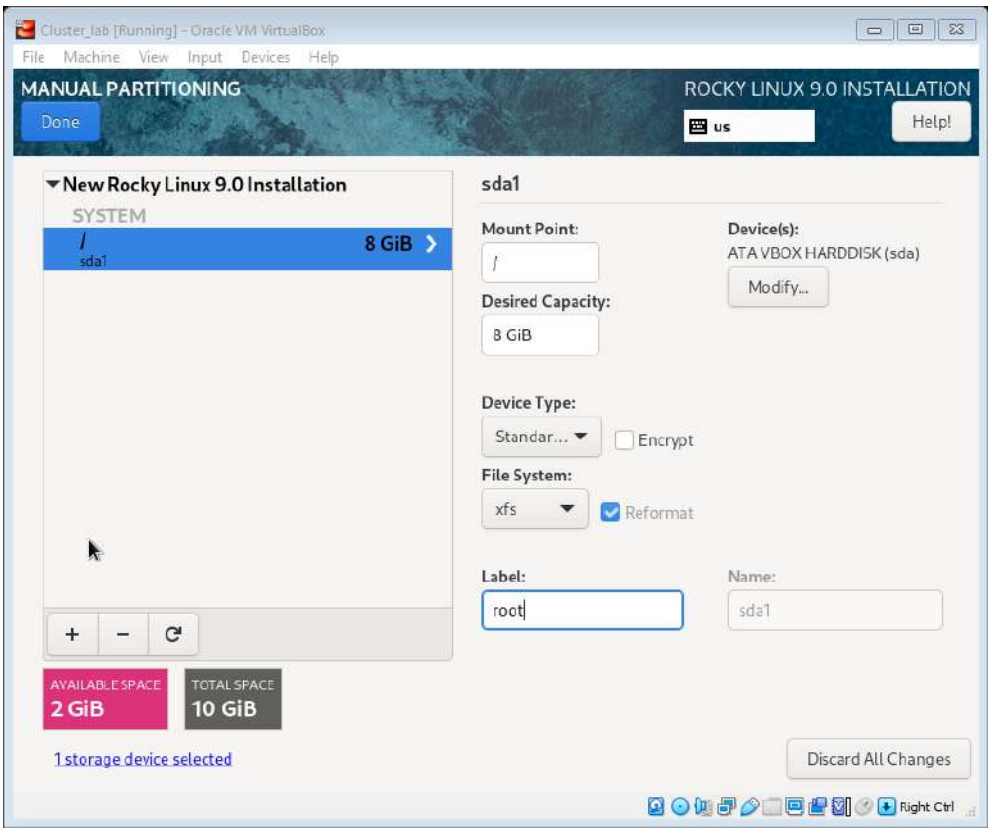

Рис. 1.20. **Налаштування першого розділу диска ВМ**

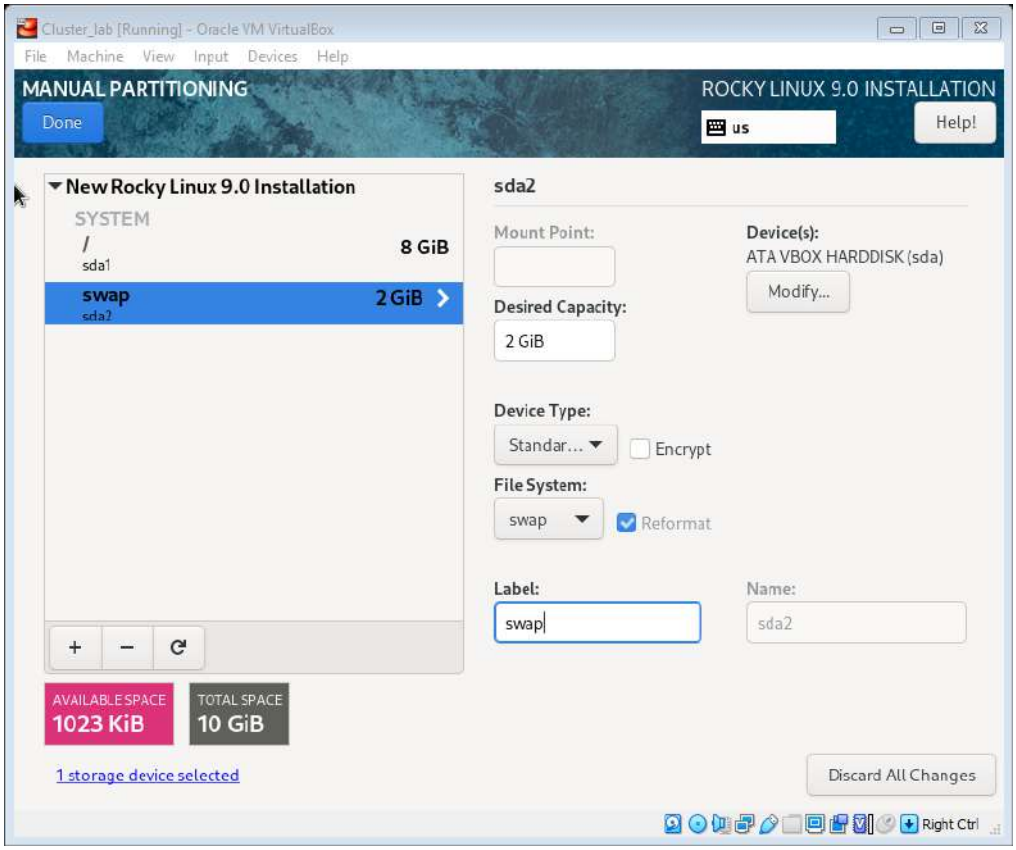

Рис. 1.21. **Налаштування другого розділу диска ВМ**

Дозволяємо зберегти зміни на диск ВМ (рис. 1.22).

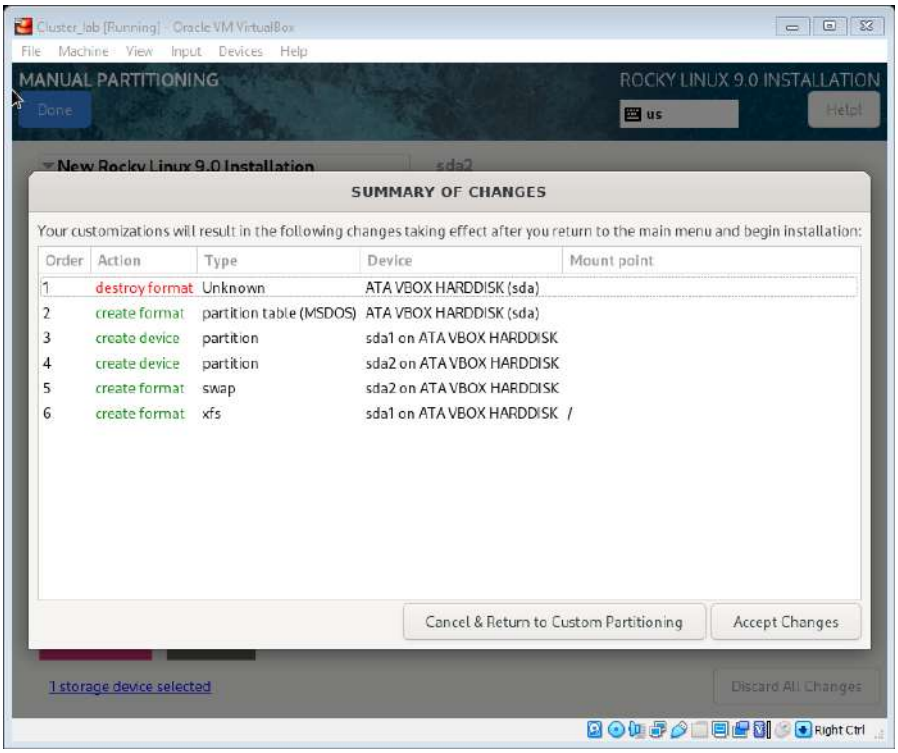

Рис. 1.22. **Зберегти зміни на диск ВМ**

Налаштовуємо МА. Налаштування першого МА залишаємо без змін. Задаємо ім'я ГС, що створюємо на першій ВМ (рис. 1.23).

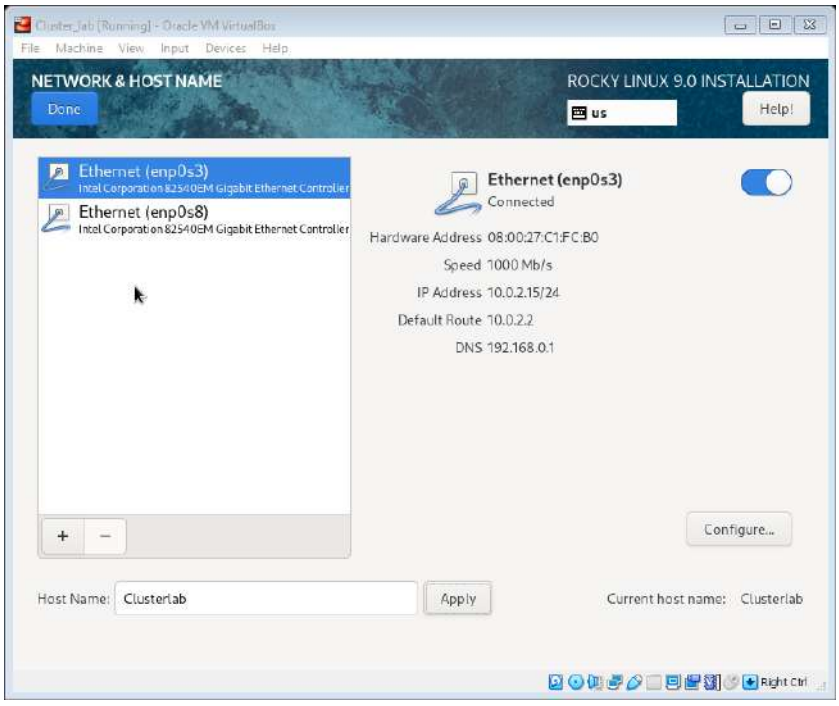

Рис. 1.23. **Налаштування першого МА першої ВМ**

Другий МА потребує налаштувань (рис. 1.24 і 1.25).

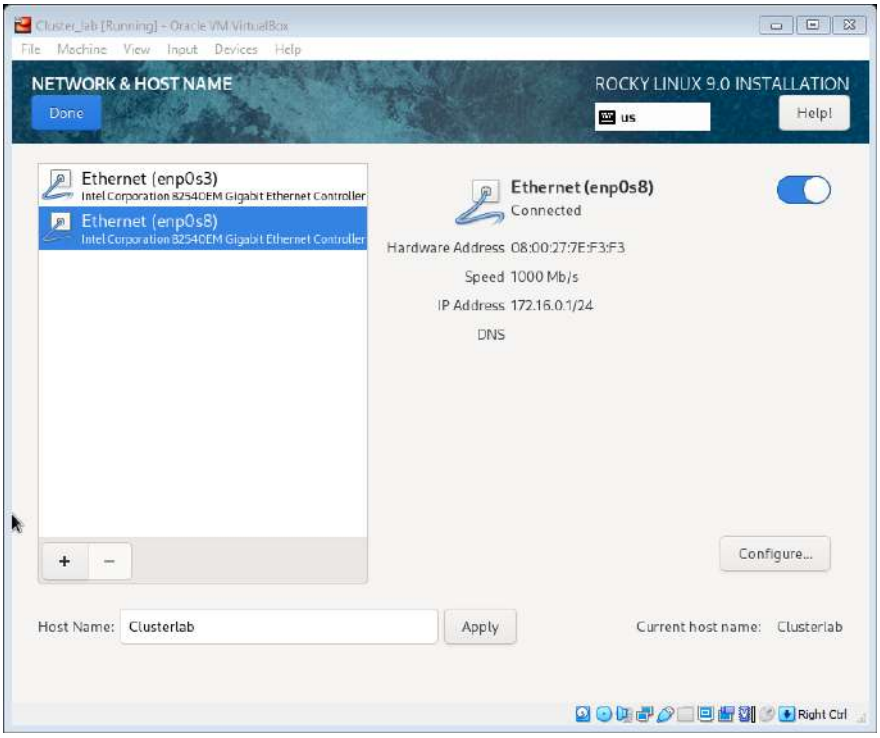

Рис. 1.24. **Налаштування другого МА першої ВМ**

Натискаємо кнопку *Configure … .* Вписуємо IP-адресу і мережеву маску в пункт *IPv4 Settings*.

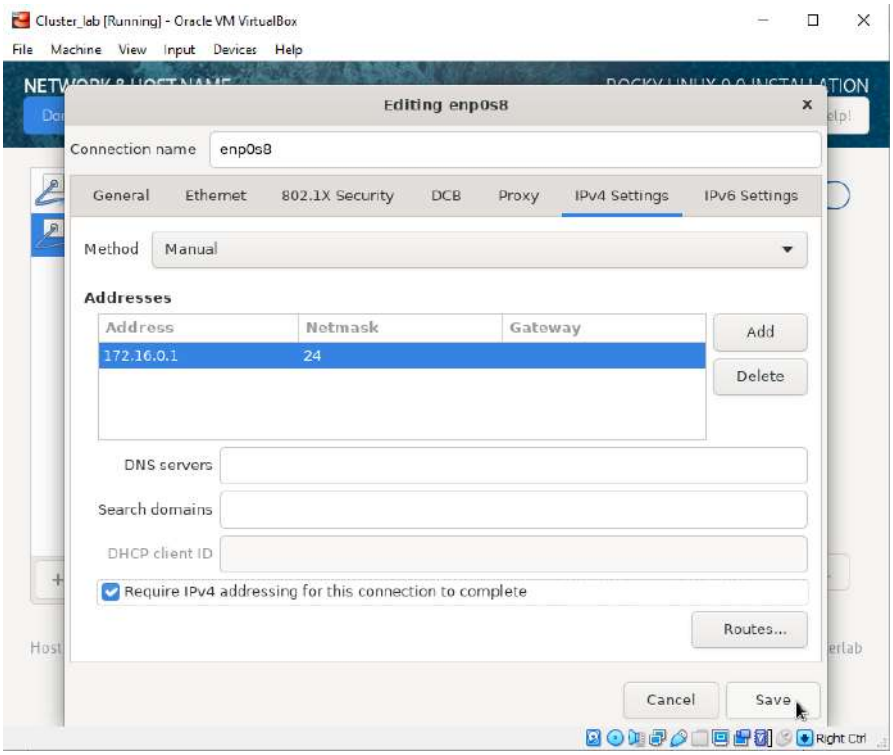

Рис. 1.25. **Задавання IP-адреси та мережевої маски**

Задаємо пароль *root* (рис. 1.26).

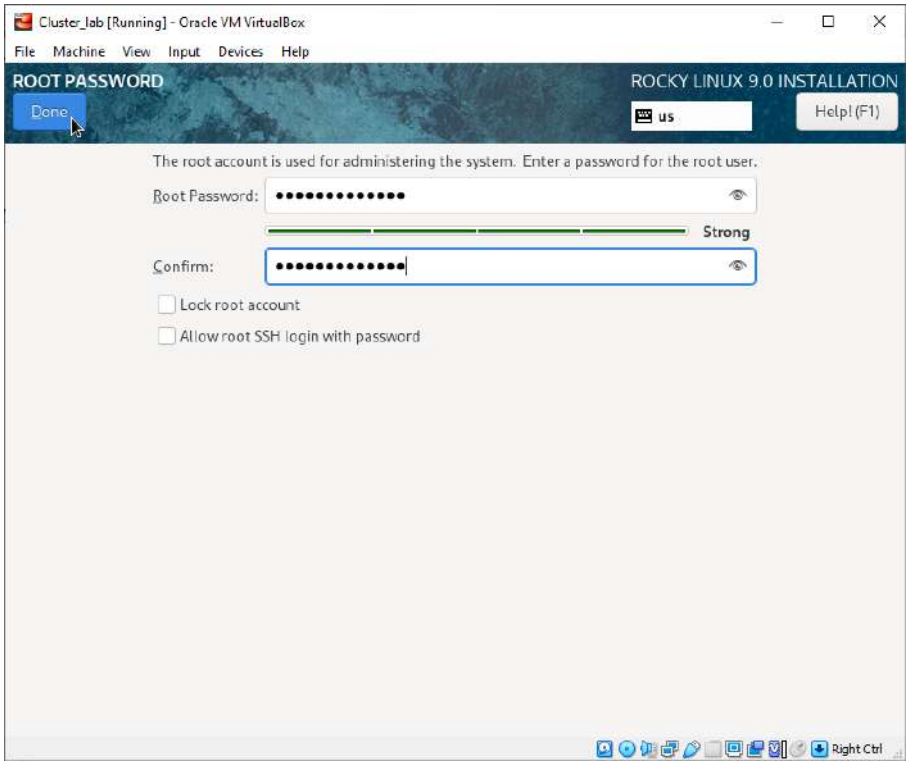

Рис. 1.26. **Налаштування пароля для root**

Створюємо користувача *User* (рис. 1.27).

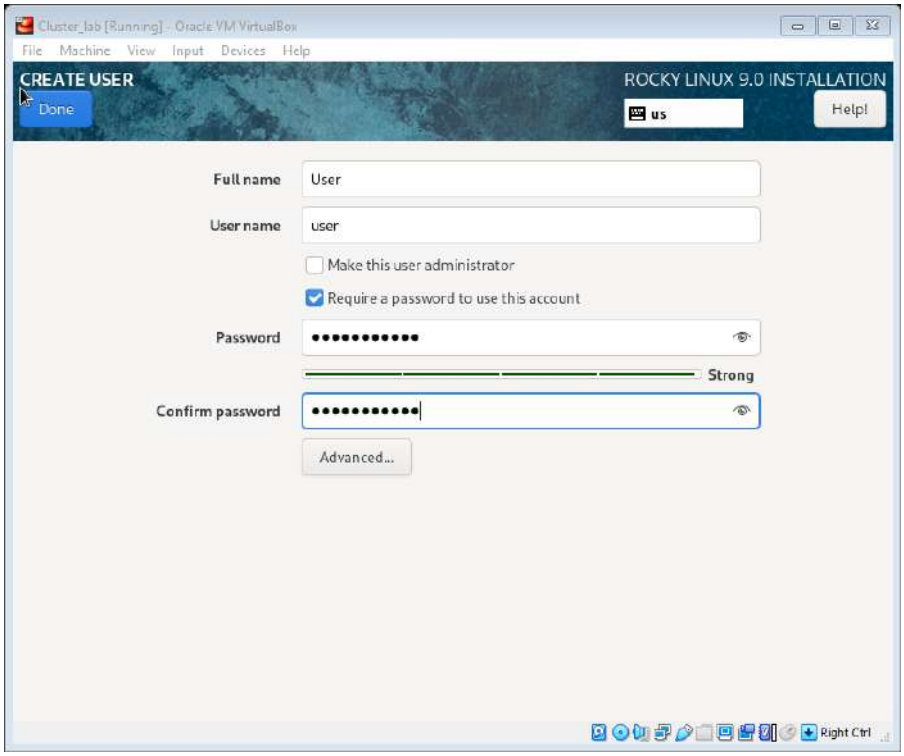

Рис. 1.27. **Створення користувача User**

Натискаємо кнопку *Begin Installation* для початку встановлення ОС (рис. 1.28).

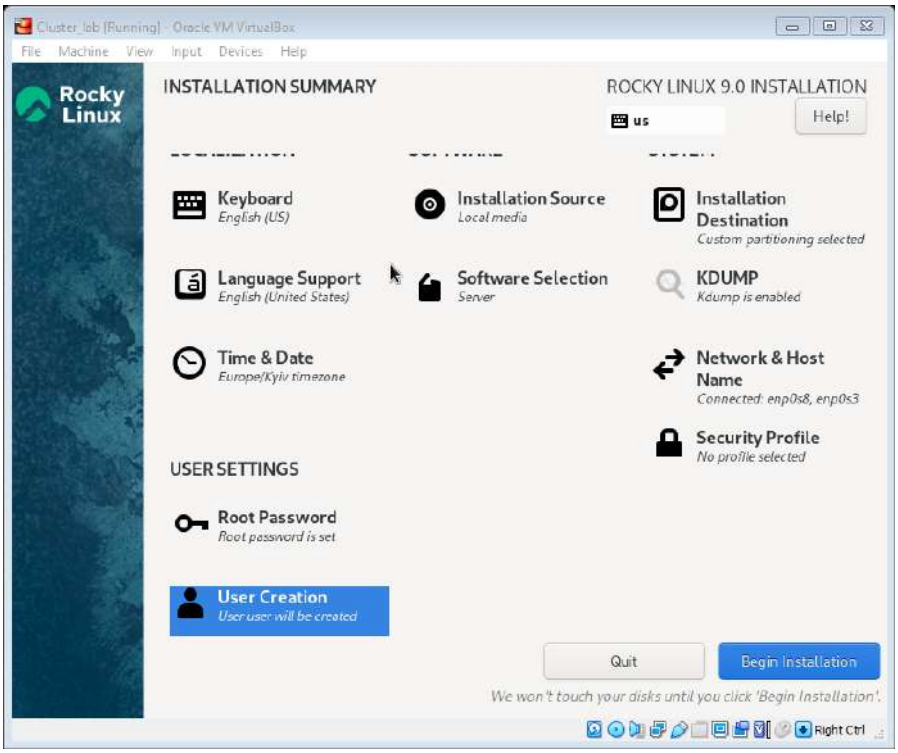

Рис. 1.28. **Вікно налаштувань перед установкою пакетів ОС**

Починаємо встановлення пакетів (рис. 1.29).

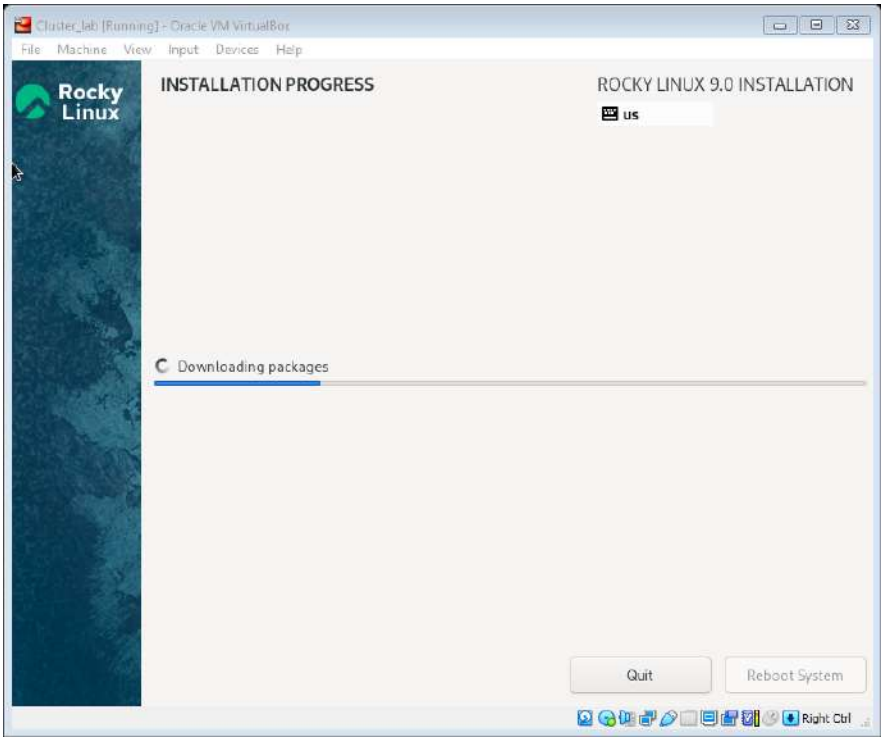

Рис. 1.29. **Установка пакетів ОС *Linux***

Після завершення установки ОС перезавантажуємо систему кнопкою *Reboot System* (рис. 1.30).

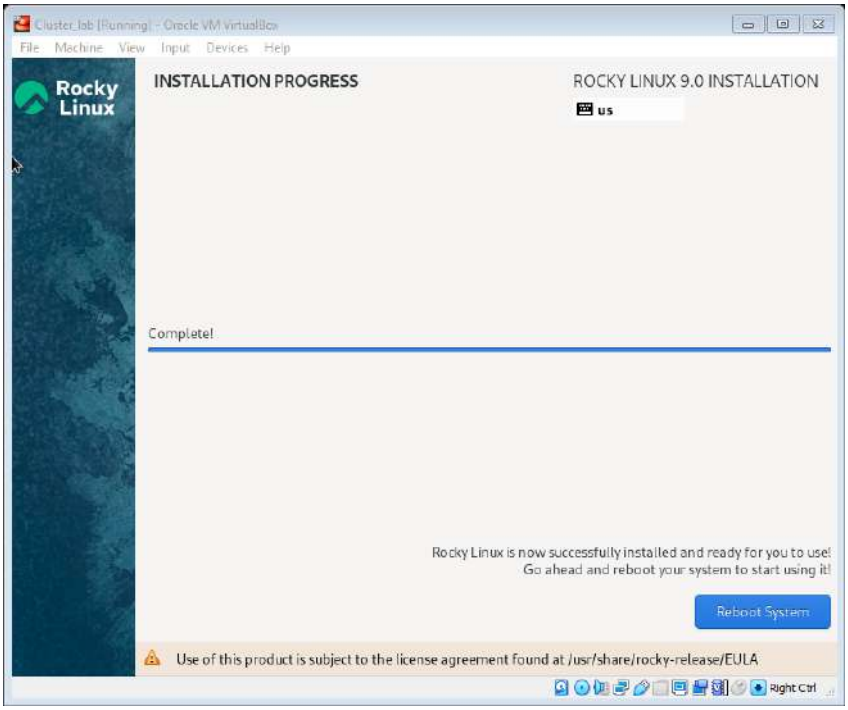

Рис. 1.30. **Завершення установки пакетів ОС *Linux***

Після завантаження ОС з'явиться запрошення на вхід користувача (рис. 1.31).

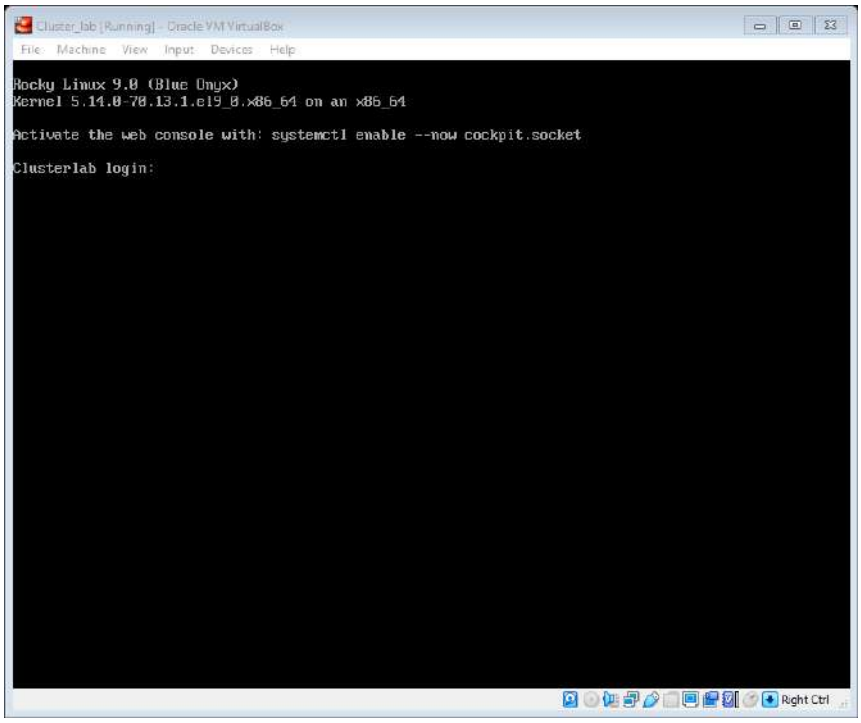

Рис. 1.31. **Запрошення на вхід користувача в ОС *Linux***

Переходимо до створення другої ВМ: натискаємо кнопку *New* (рис. 1.32) та в вікні, що відкрилося, задаємо назву й тип ОС (рис. 1.33).

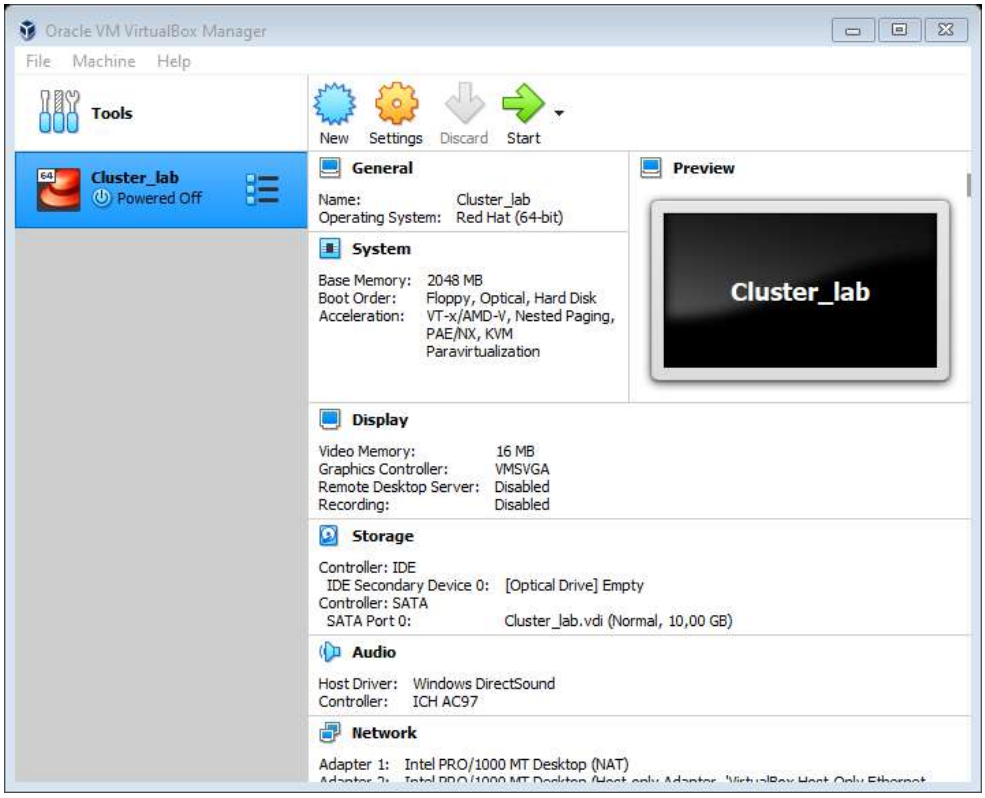

Рис. 1.32. **ВМ Oracle VM VirtualBox Manager з першою ВМ**

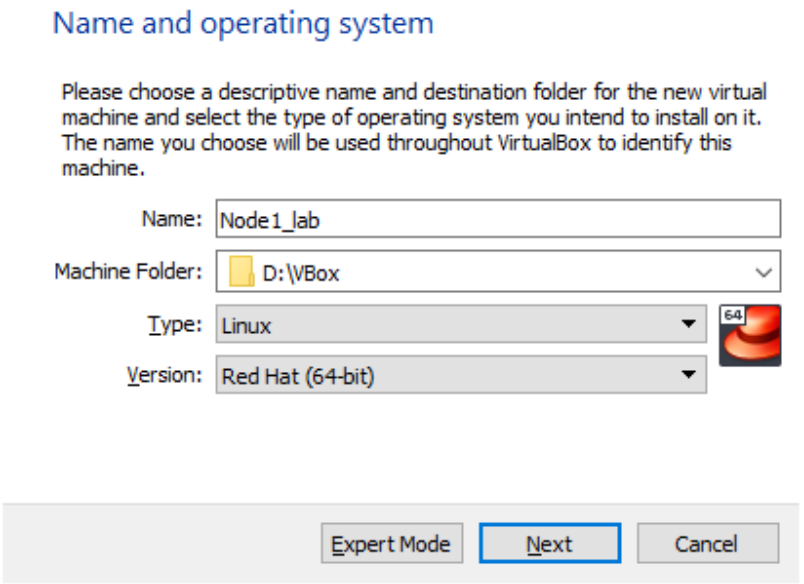

Рис. 1.33. **Створення другої ВМ**

Задаємо розмір ОП, аналогічно як наведено на (див. рис. 1.4). Створюємо диск для ВМ, на який буде встановлено ОС (див. рис. 1.5). Обираємо тип створеного файла диска та тип розміщення (див. рис. 1.6 і 1.7).

Вводимо назву, шлях (місце розташування диска на локальному комп'ютері) та задаємо максимальний розмір файла диска, який він може зайняти на фізичному носії (рис. 1.34).

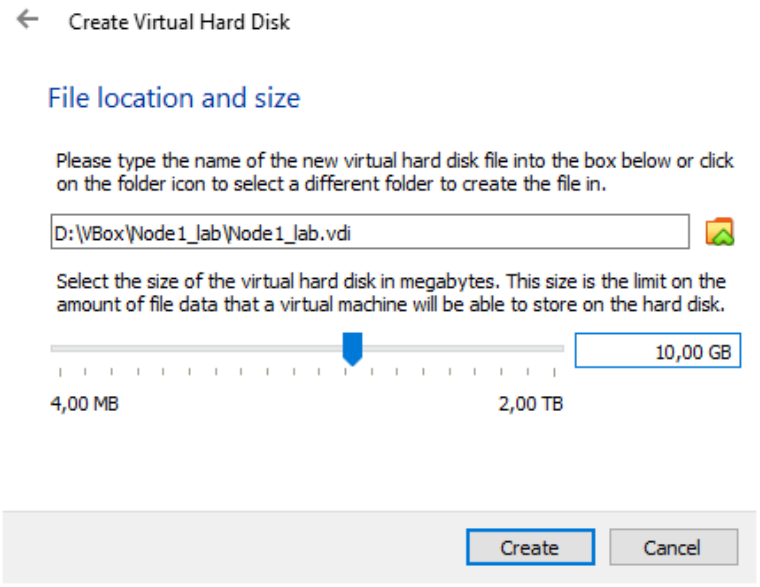

Рис. 1.34. **Вибір назви, місця зберігання та розмір файла диска ВМ**

На цьому створення другої ВМ буде завершено (рис. 1.35).

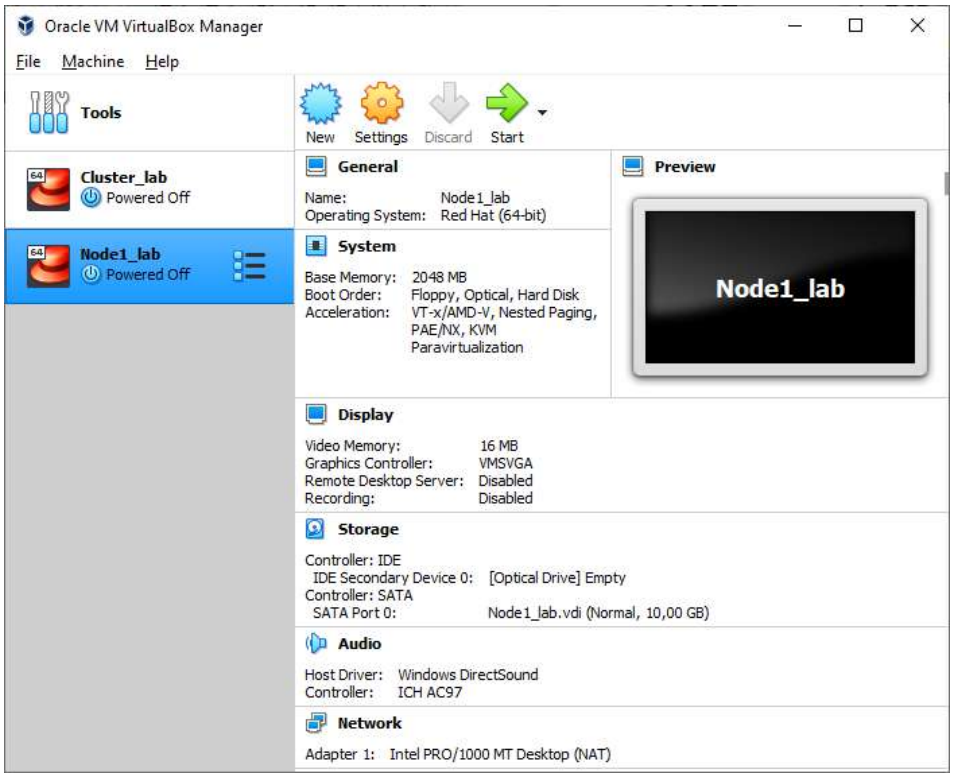

Рис. 1.35. **ВМ Oracle VM VirtualBox Manager з другою ВМ**

Для налаштування ВМ розглянемо її призначення. Друга і третя ВМ створюються як ОВ. Їм потрібен один МА для зв'язку з внутрішньою мережею (інтранет), який налаштовується з політиками довіри. Це дозволяє забезпечити максимально швидке передавання даних між ОВ. Вимоги до ОП та ядер процесора залежать від можливостей основної системи.

Наприклад, якщо основна система складається з 16 ядер процесора та 32 Гб ОП, то можна розподілити ресурси таким чином: 2 ядра та 4 Гб ОП виділити для ГС, 6 ядер та 12 Гб ОП виділити для першого ОВ, 6 ядер та 12 Гб ОП виділити для другого ОВ, а 2 ядра та 4 Гб ОП – для роботи основної системи. Мінімальні вимоги до ОВ: 2 ядра та 2 Гб ОП. Ураховуючи всі рекомендації, визначено, що мінімальна конфігурація комп'ютера для виконання лабораторної роботи складає 5 процесорних ядер та 6 ГБ ОП.

Налаштування кількості ядер процесора (див. рис. 1.11), підключення образу ОС *Linux* (див. рис. 1.12) виконується аналогічно тому, як це робилось для першої ВМ. Налаштовуємо МА (рис. 1.36). МА створюємо в режими *Host-only Adapter*. На цьому налаштування другої ВМ завершено.

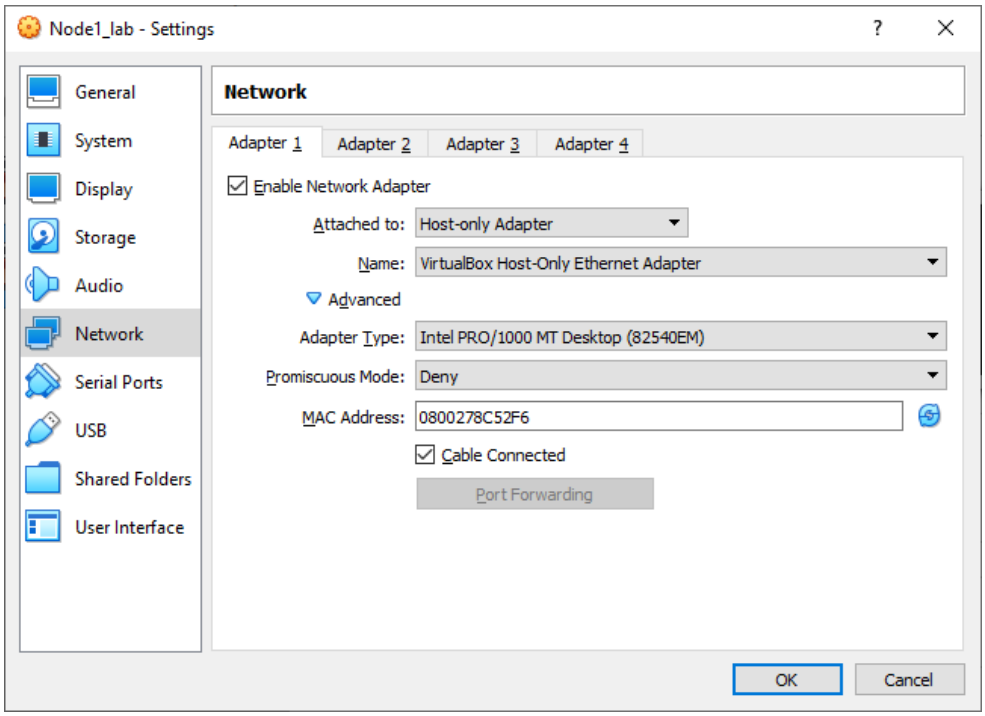

Рис. 1.36. **Налаштування МА**

Ввійдіть в панель налаштувань створеної ВМ, натискаючи кнопку *Settings* (рис. 1.37), а потім необхідно перейти до потрібних вкладок з відповідними параметрами.

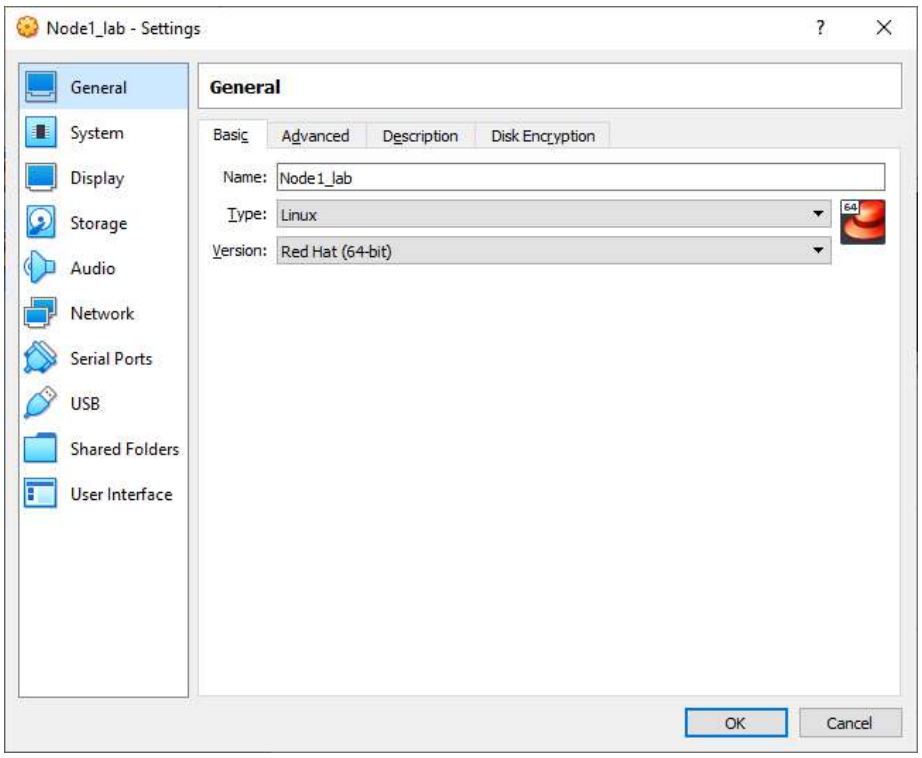

Рис. 1.37. **Панель налаштувань ВМ**

Установлення ОС на другу ВМ

Запускаємо ВМ, натискаючи на кнопку *Start* (рис. 1.38).

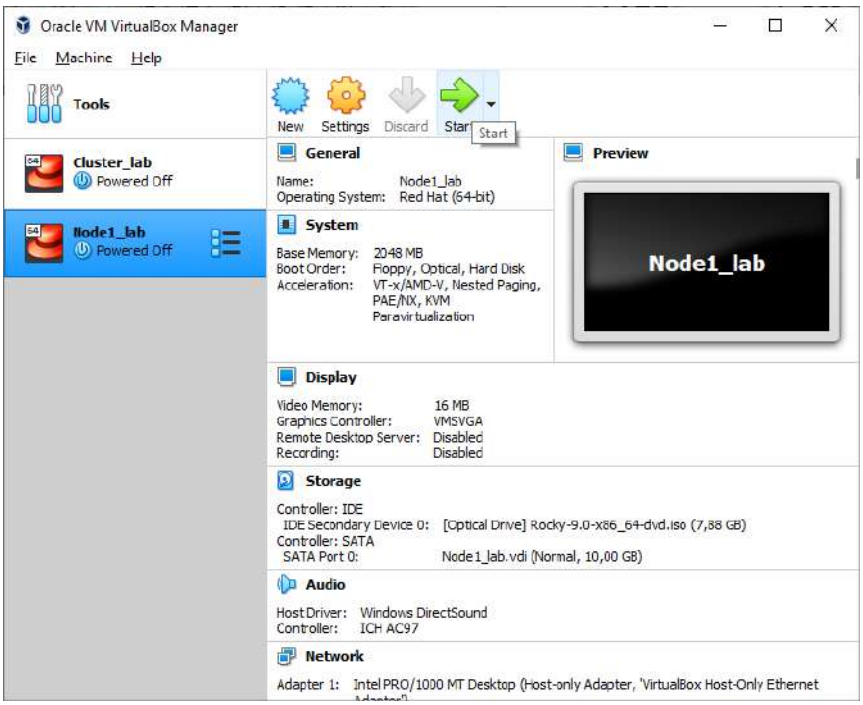

Рис. 1.38. **Запуск другої ВМ**

Обираємо встановлення ОС *Linux* (рис. 1.39).

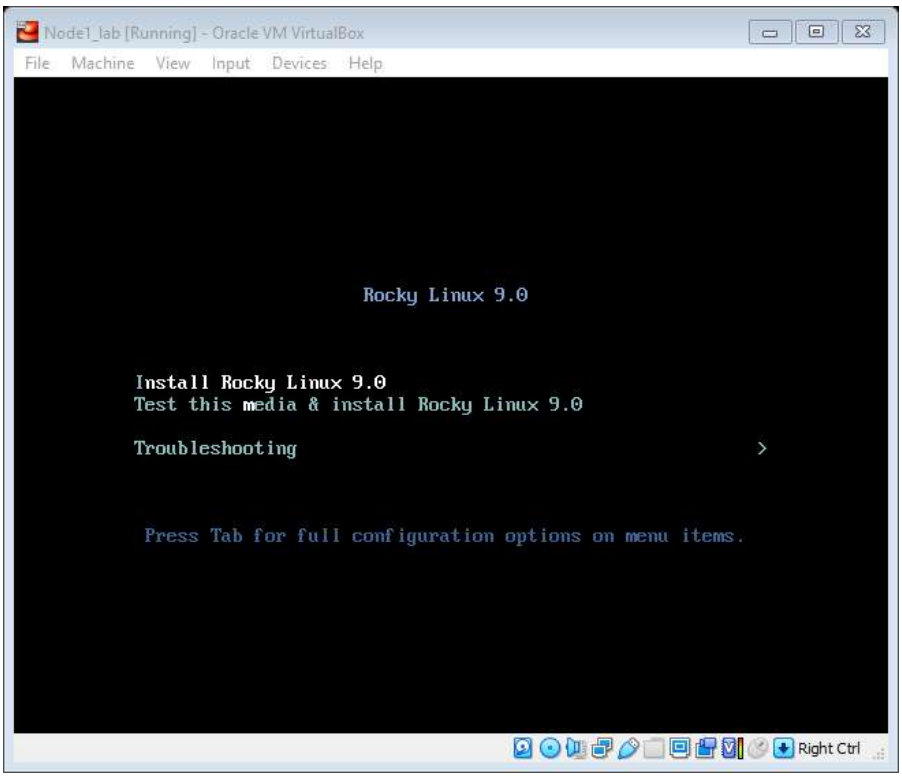

Рис. 1.39. **Вибір установки ОС**

Обираємо мову установки (рис. 1.40).

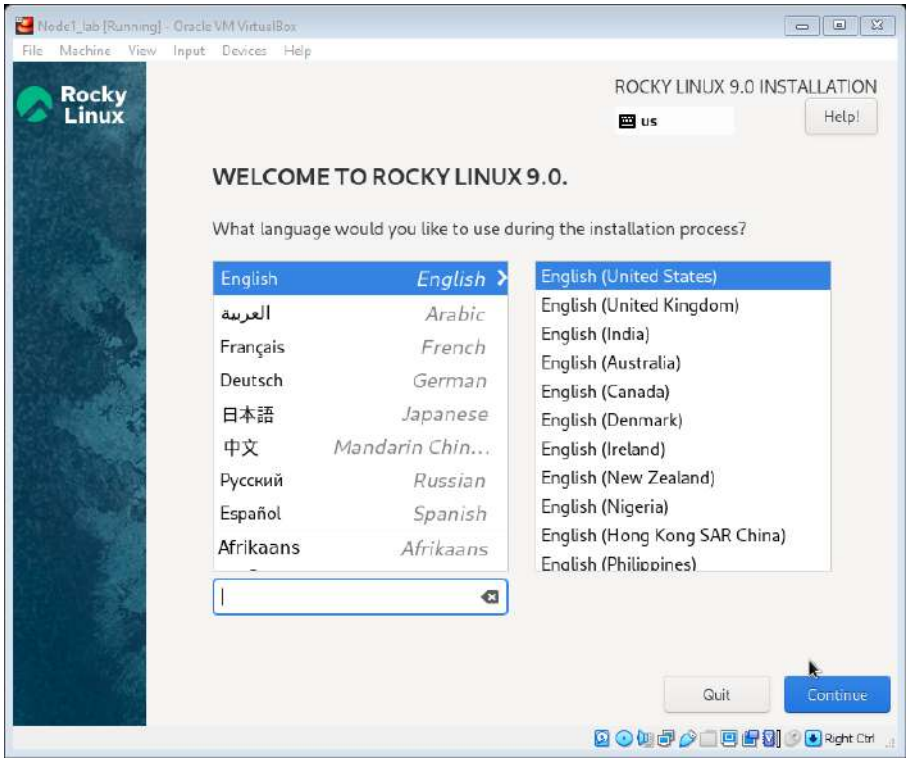

Рис. 1.40. **Вибір мови установки**

Переходимо до панелі налаштувань (рис. 1.41).

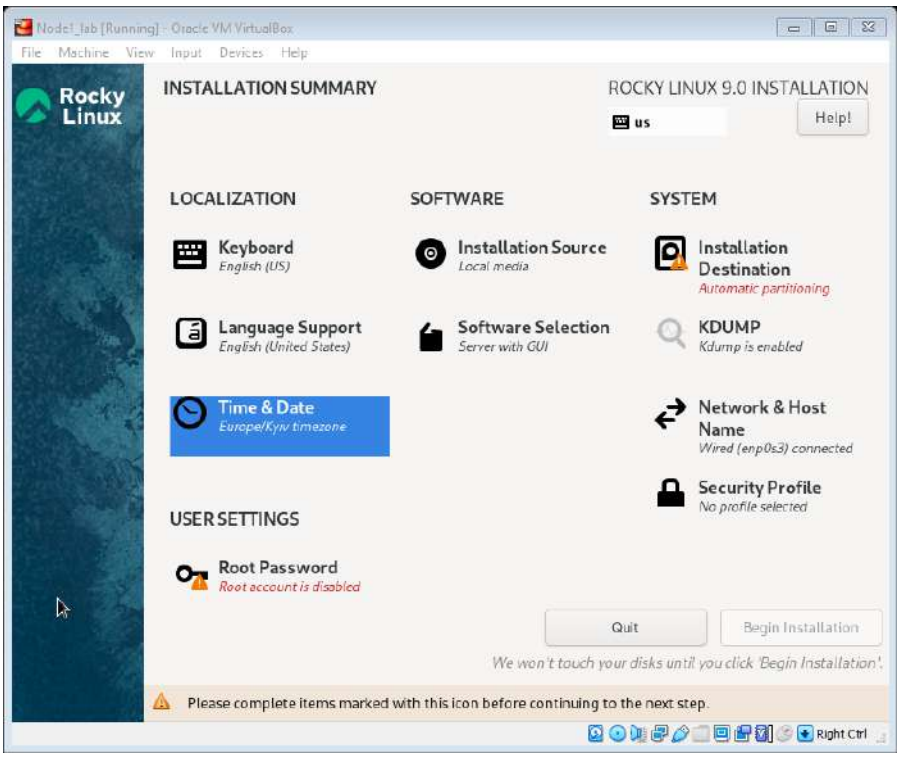

Рис. 1.41. **Панель налаштувань**

Обираємо режим установки *Server* (рис. 1.42).

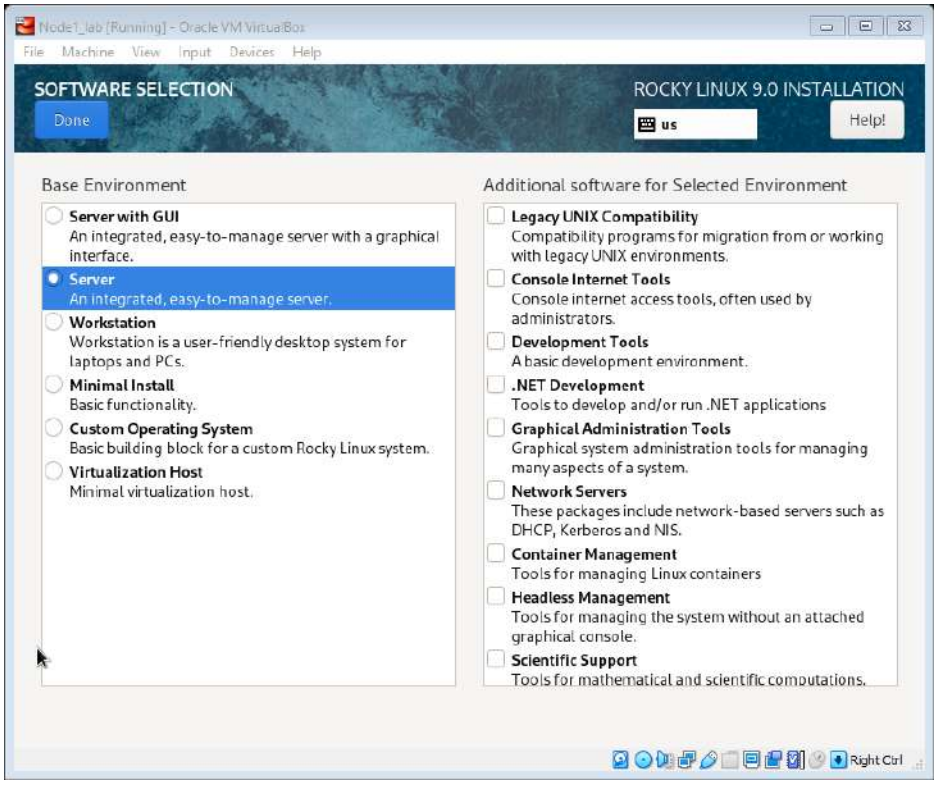

Рис. 1.42. **Вибір режиму установки Server**

Створюємо розділи диска ВМ (рис. 1.43 – 1.45).

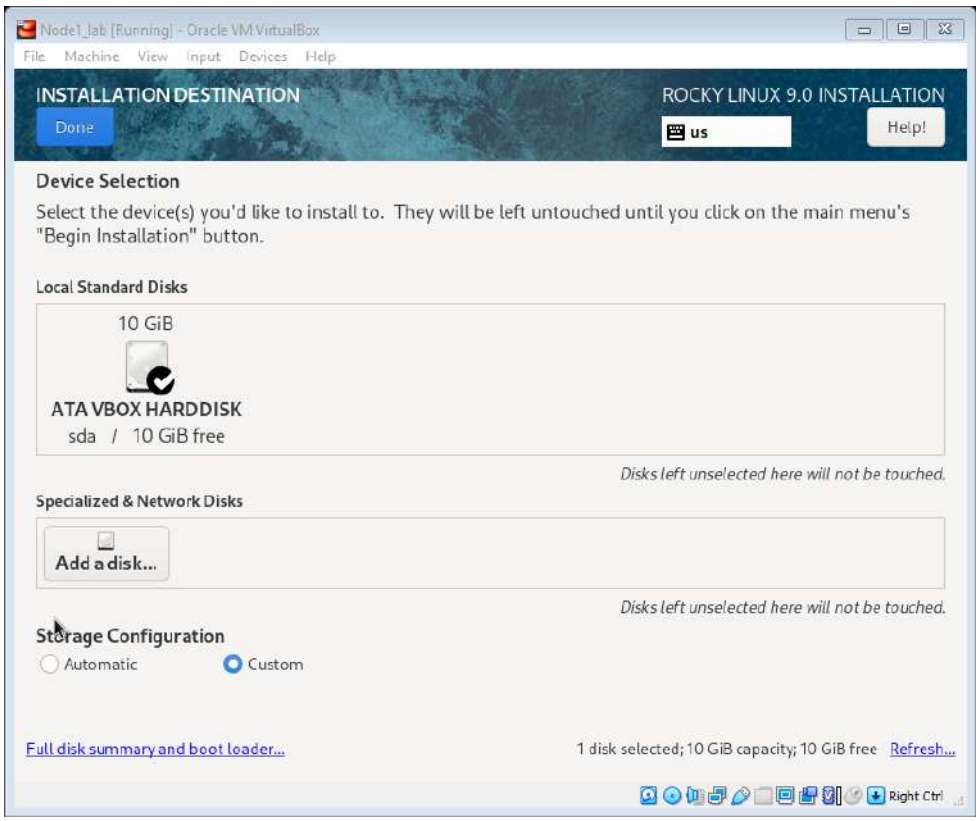

Рис. 1.43. **Вибір диска та ручного створення розділів**

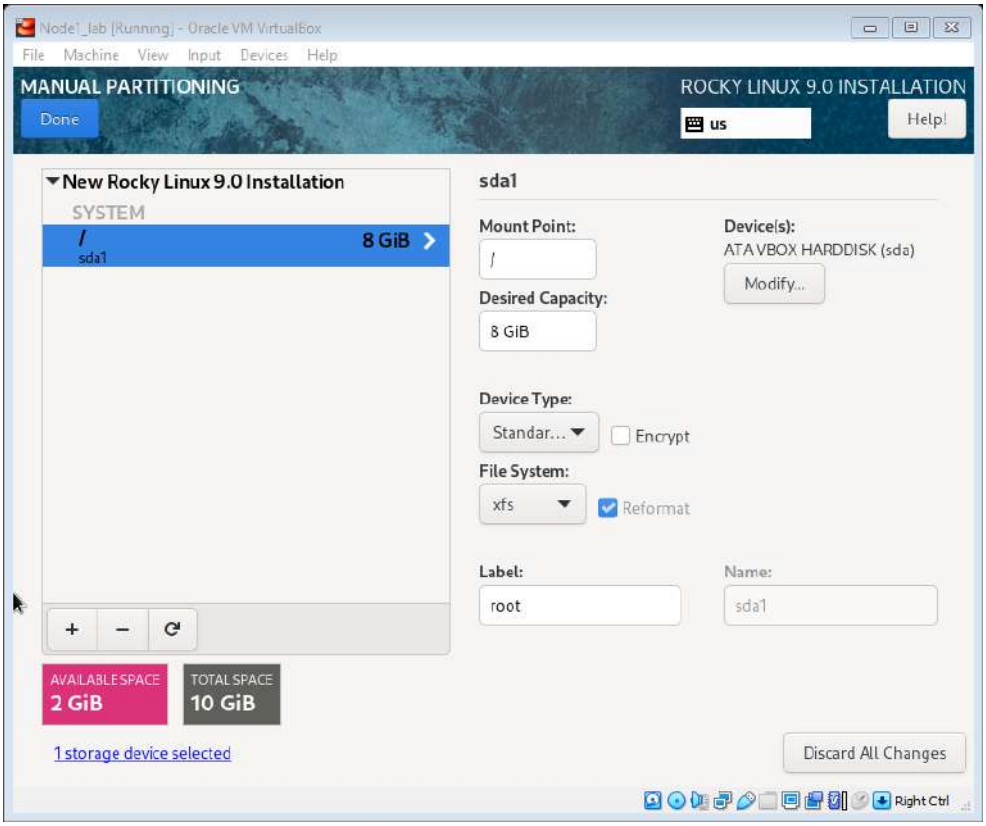

Рис. 1.44. **Налаштування першого розділу диска ВМ**

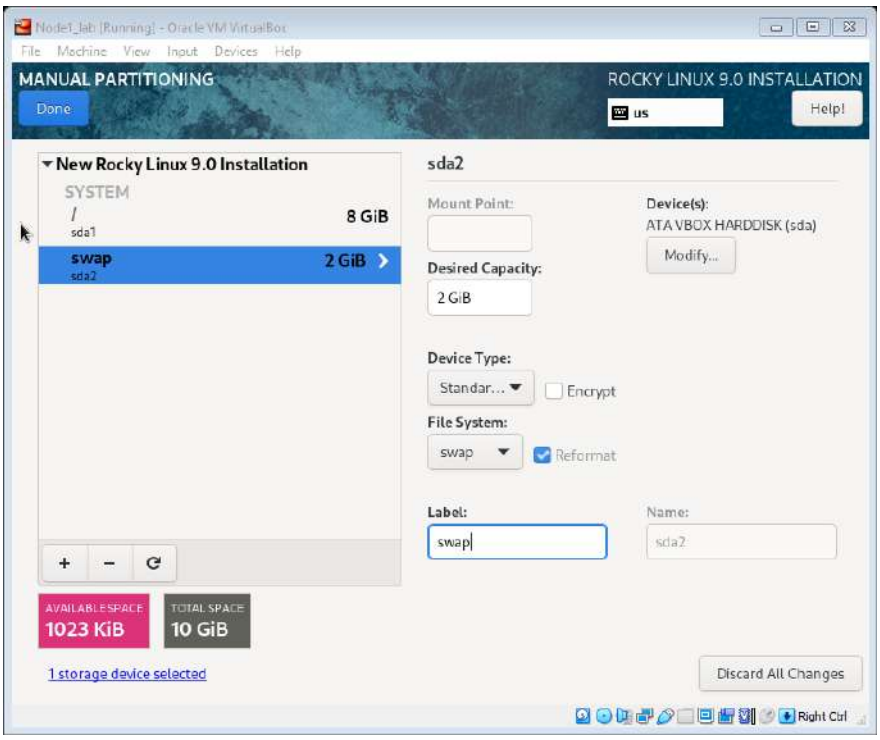

Рис. 1.45. **Налаштування другого розділу диска ВМ**

Дозволяємо зберегти зміни на диск ВМ (рис. 1.46).

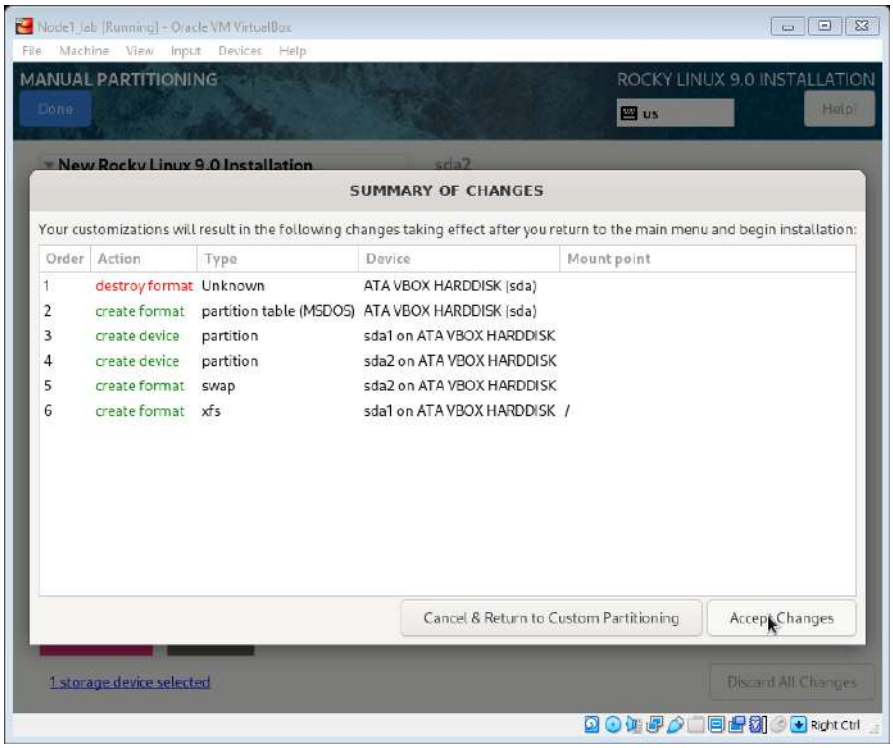

Рис. 1.46. **Зберегти зміни на диск ВМ**

Налаштовуємо МА. Задаємо ім'я першого ОВ, що створюємо на другій ВМ (рис. 1.47).

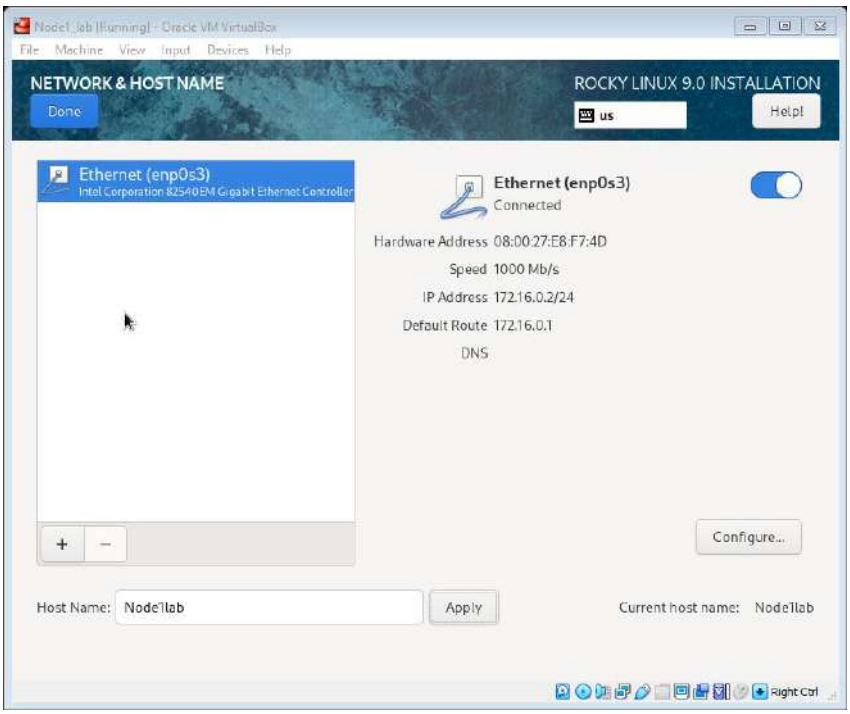

Рис. 1.47. **Налаштування МА першого ОВ другої ВМ**

Натискаємо кнопку *Configure … .* Вписуємо IP-адресу, мережеву маску та адресу шлюзу в пункт *IPv4 Settings* для МА (рис. 1.48).

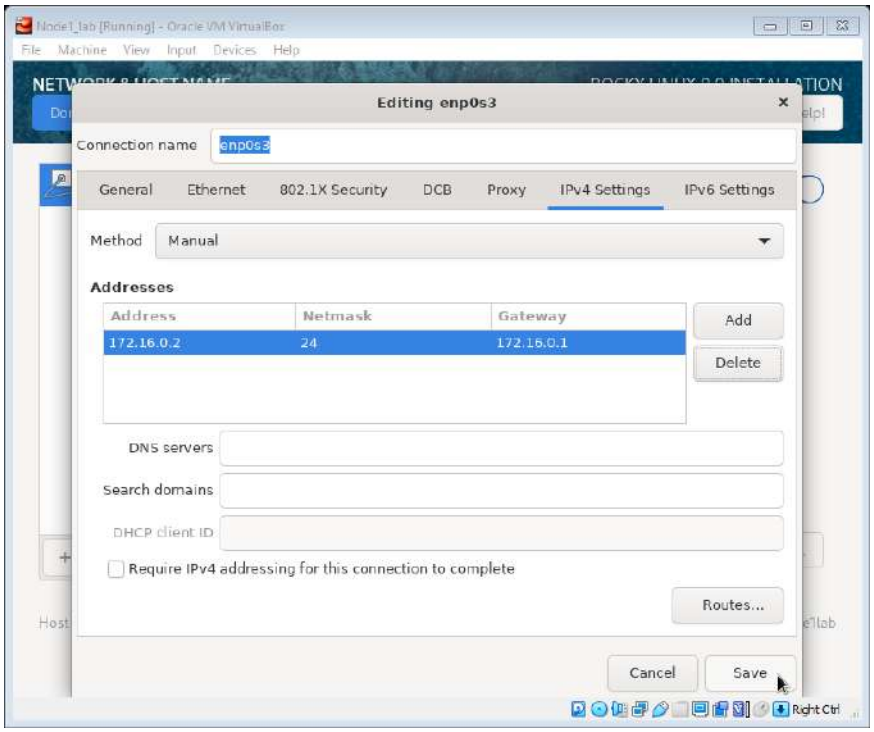

Рис. 1.48. **Задаємо IP-адресу, мережеву маску та адресу шлюзу**

Задаємо пароль *root* (рис. 1.49). Пароль повинен співпадати з паролем на ГС, що створений на першій ВМ.

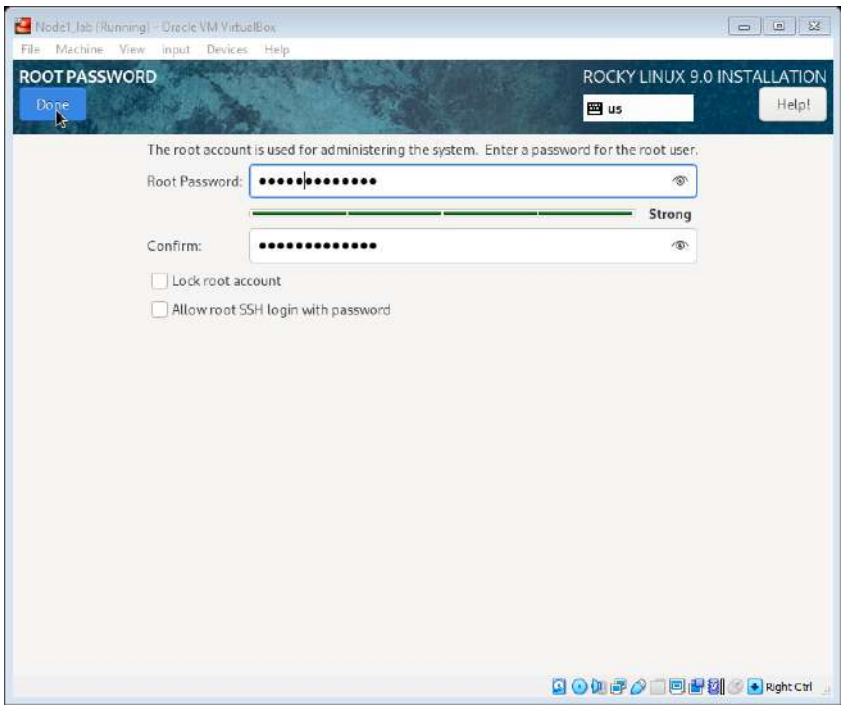

Рис. 1.49. **Налаштування пароля для root**

Створюємо користувача *User* (рис. 1.50). Пароль повинен співпадати з паролем на ГС, створений на першій ВМ.

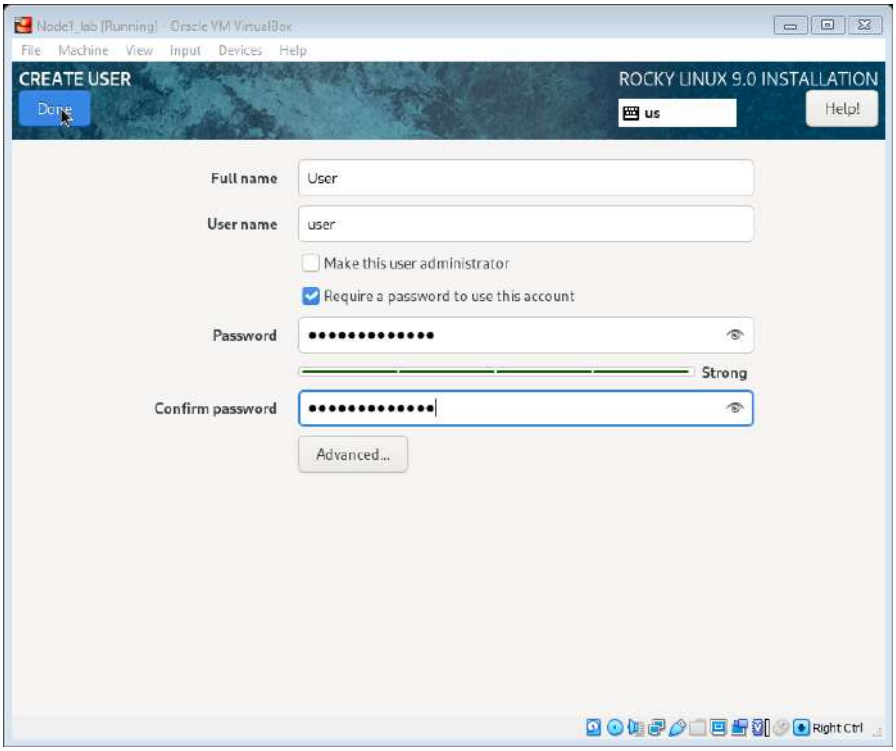

Рис. 1.50. **Створення користувача User**

Натискаємо кнопку *Begin Installation* для початку установки ОС (рис. 1.51).

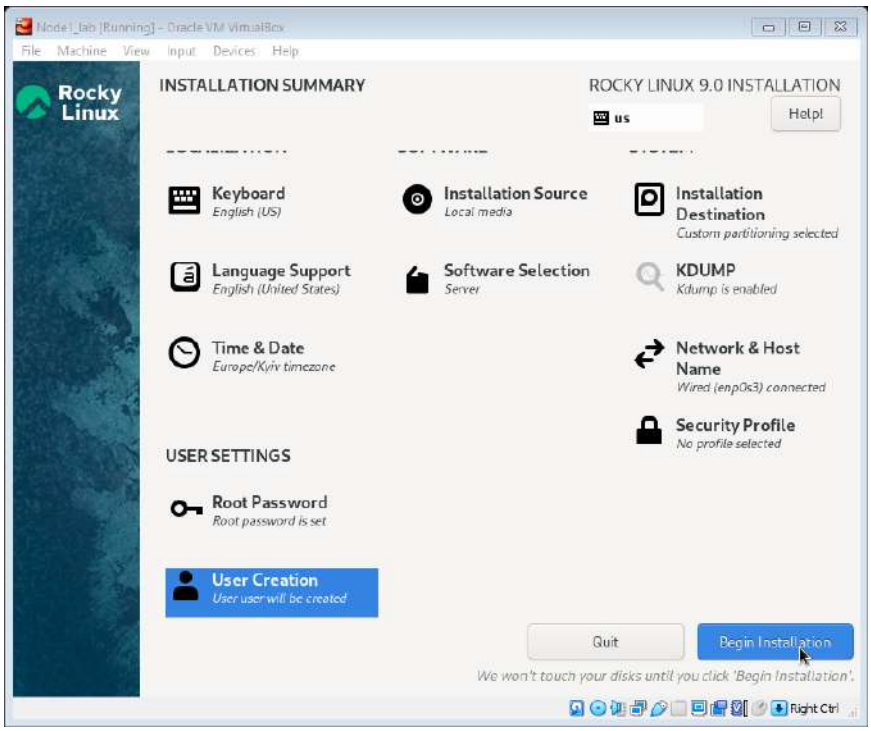

Рис. 1.51. **Вікно налаштувань перед установкою ОС**

Установлення пакетів (рис. 1.52).

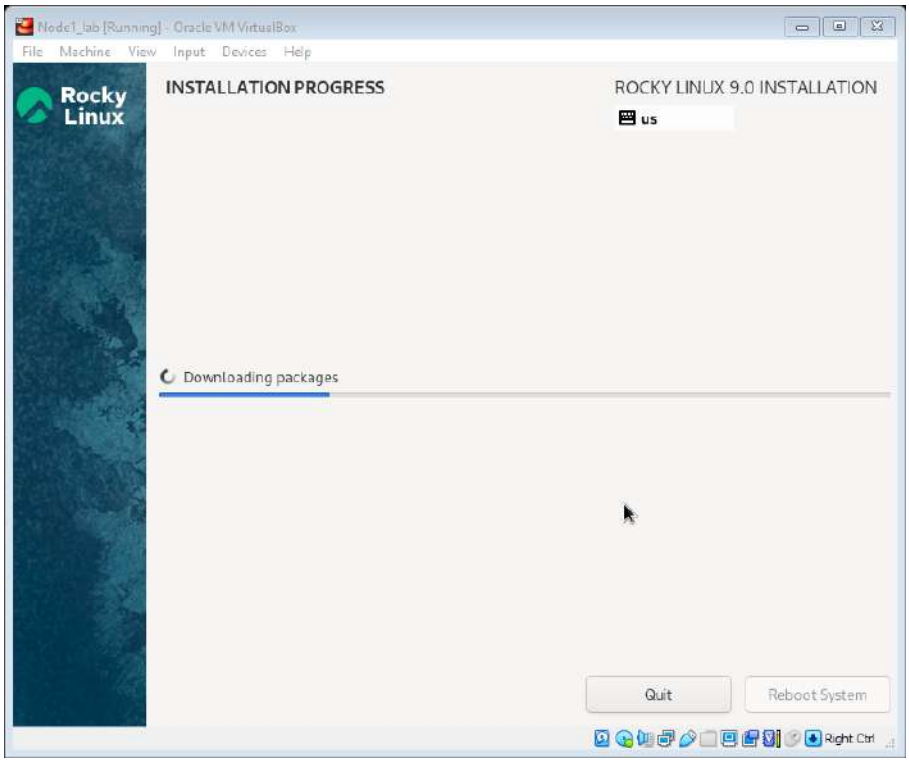

Рис. 1.52. **Установка пакетів ОС *Linux***

Після завершення установки ОС перезавантажуємо систему кнопкою *Reboot System* (рис. 1.53).

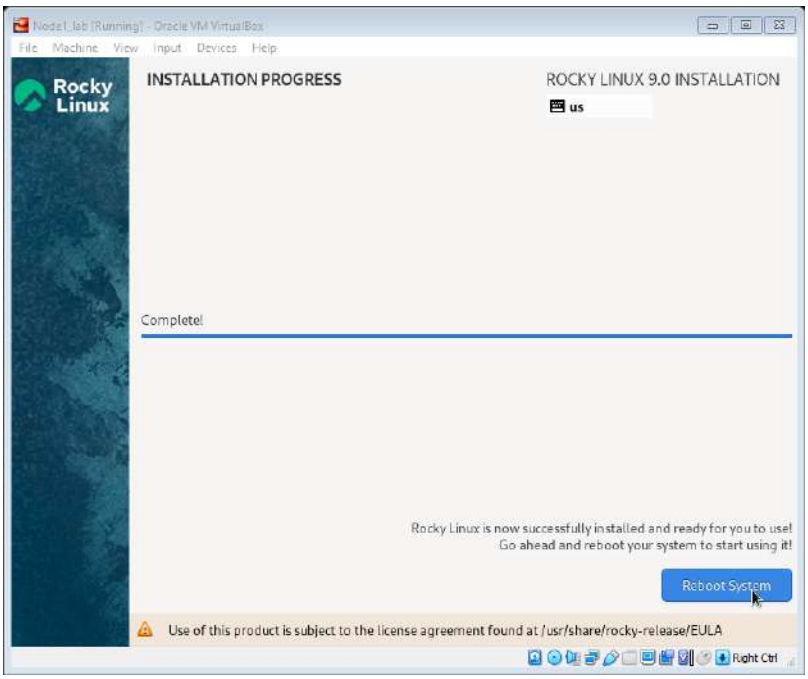

Рис. 1.53. **Завершення установки пакетів ОС *Linux***

Після завантаження ОС з'явиться запрошення на вхід користувача (рис. 1.54).

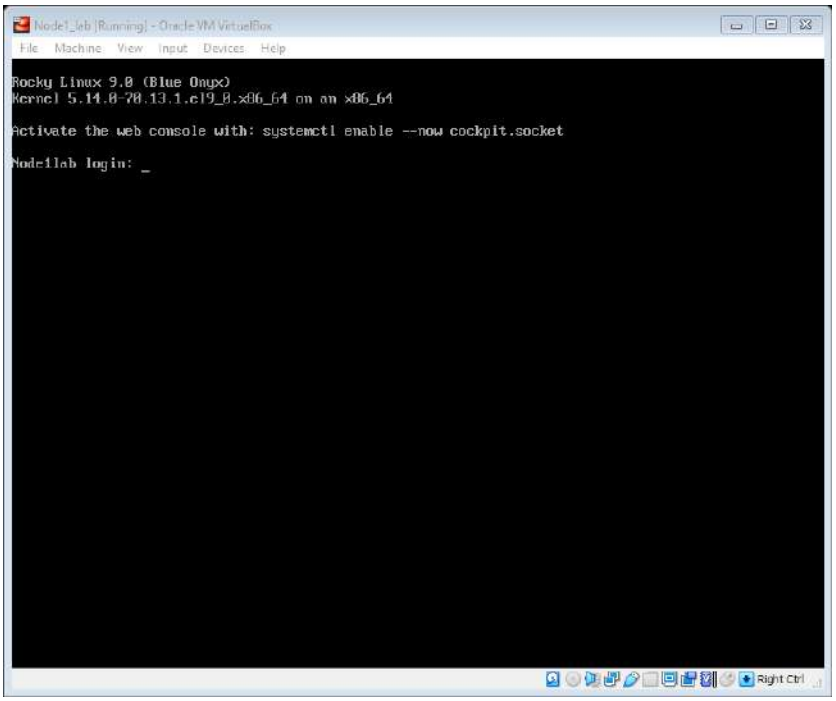

Рис. 1.54. **Запрошення на вхід користувача в ОС *Linux***

Переходимо до створення третьої ВМ. Повторимо всі етапи створення другої ВМ з тими ж самими рекомендаціями, окрім: назви файла

третьої ВМ (Node2_lab), назви другого ОВ (Node2lab) та IP-адреси МА (рис. 1.55).

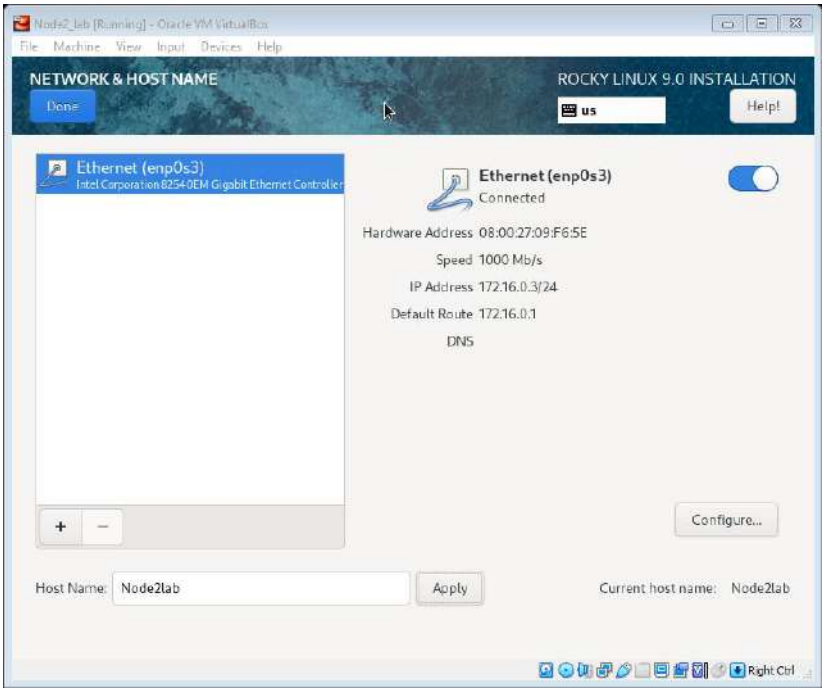

Рис. 1.55. **Налаштування МА другого ОВ третьої ВМ**

Налаштування КМ та служб кластера

Завантажуємо всі три створені ВМ, якщо вони вимкнені. Для завантаження ВМ використовуємо команду *Start* (рис. 1.56).

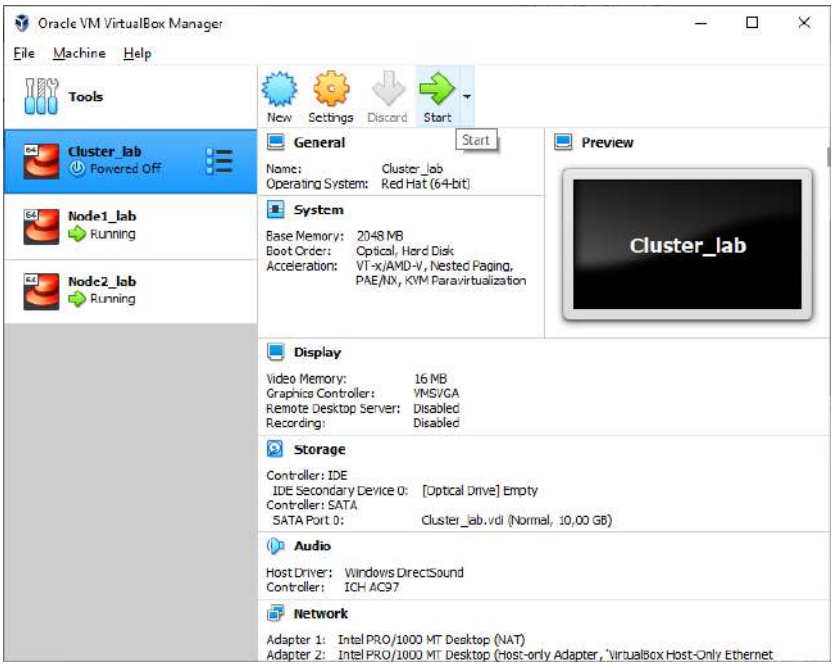

Рис. 1.56. **Завантаження ВМ**

Входимо на ГС *Clusterlab* під користувачем *user*. Зверніть увагу, що пароль під час введення на екрані не відображається (рис. 1.57).

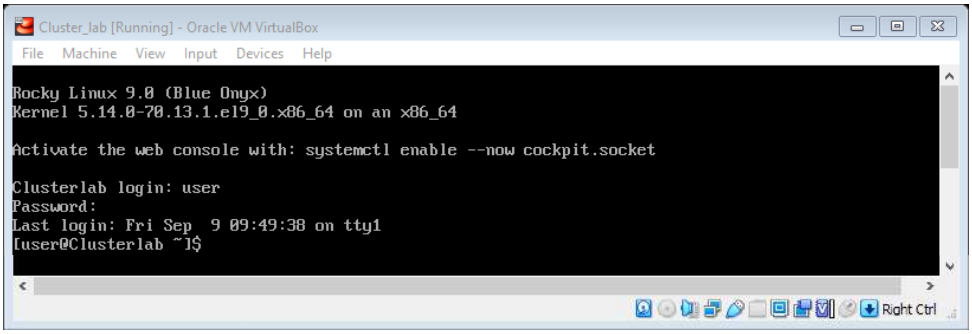

Рис. 1.57. **Вхід на кластер**

Перевіряємо мережеві налаштування ГС (рис. 1.58).

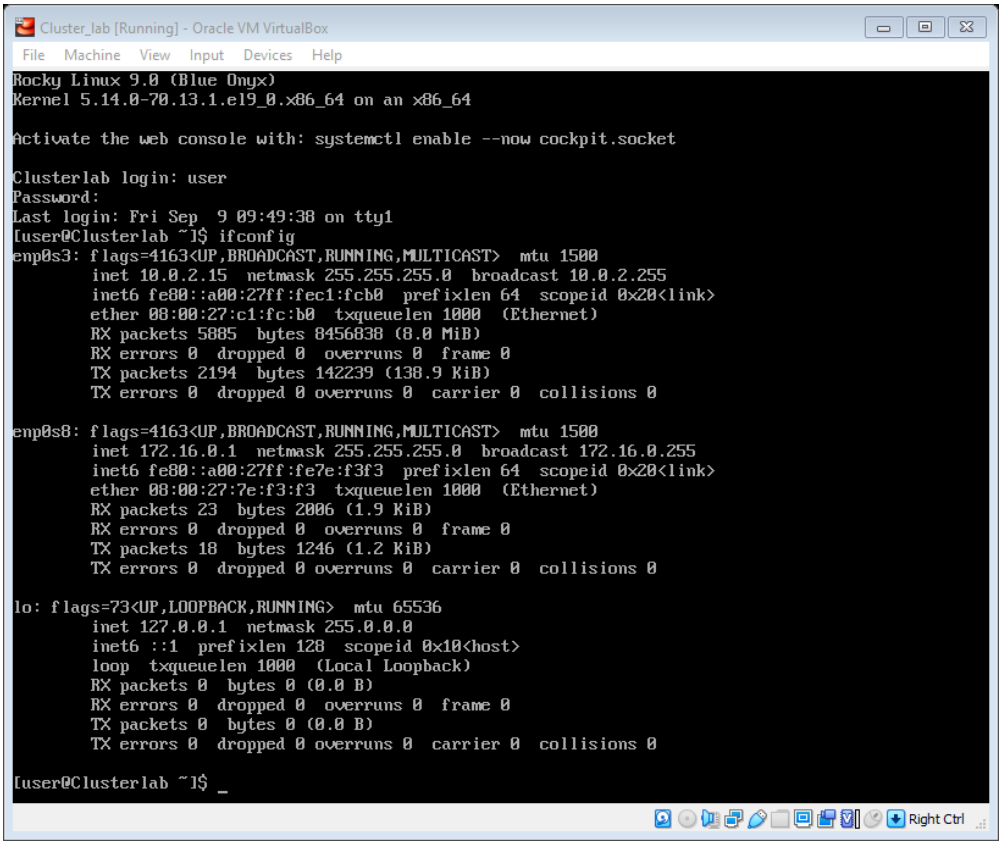

Рис. 1.58. **Налаштування МА на ГС**

Використовуючи команду *ping*, перевіряємо доступність ОВ кластера (рис. 1.59): ping 172.16.0.2; ping 172.16.0.3. Якщо ОВ недоступні, то перевірте, що обидві ВМ *Node1_lab* та *Node2_lab* завантажені

правильно та знаходяться в стані запрошення введення імені користувача.

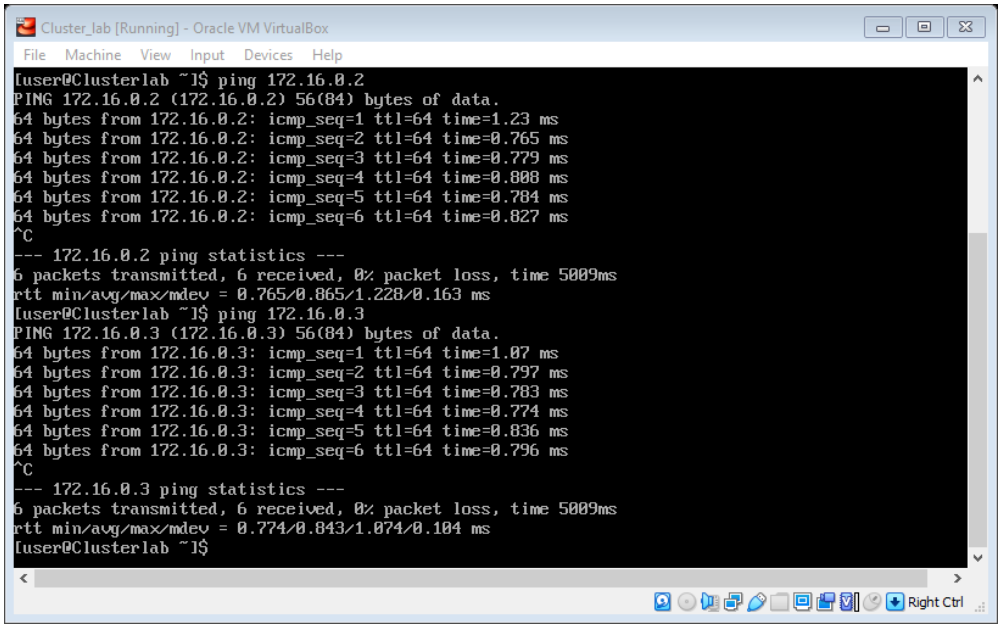

Рис. 1.59. **Перевірка доступності обох ОВ з ГС кластера**

Усі налаштування системи відбуваються з правами адміністратора *root* (рис. 1.60), а програми виконуються від імені користувача *user*. Для того, щоб отримати права адміністратора системи, виконайте команду *su* та введіть пароль, що задавали під час установки ОС *Linux* для користувача *root*.

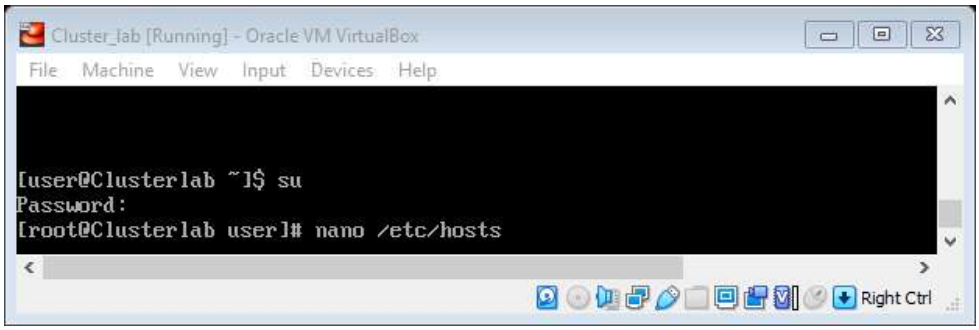

Рис. 1.60. **Отримання прав root та редагування файла hosts**

Змінювати конфігураційні файли системи будемо за допомогою редактора *nano*. Підказки щодо роботи редактора подано на нижній панелі вікна (рис. 1.61).

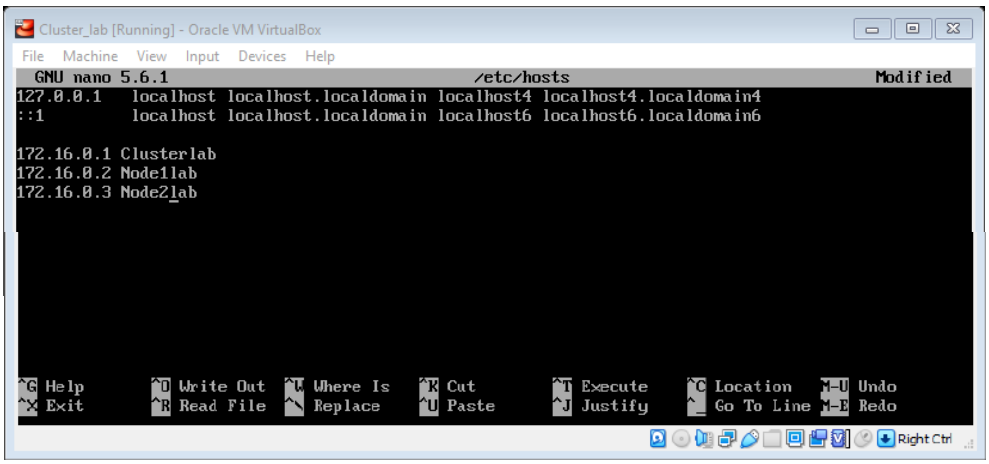

Рис. 1.61. **Налаштований файл hosts**

Щоб мати можливість звертатися до вузлів не лише за IP-адресою, а також за ім'ям, потрібно зробити відповідні налаштування в файлі /etc/hosts (див. рис. 1.60 і 1.61).

Дану операцію потрібно виконати також на ОВ *Node1lab* та ОВ *Node2lab* кластера.

Налаштування директорії віддаленого доступу за протоколом NFS

Підключаємо образ дистрибутиву ОС *Linux* до ВМ *Cluster_lab* (рис. 1.62).

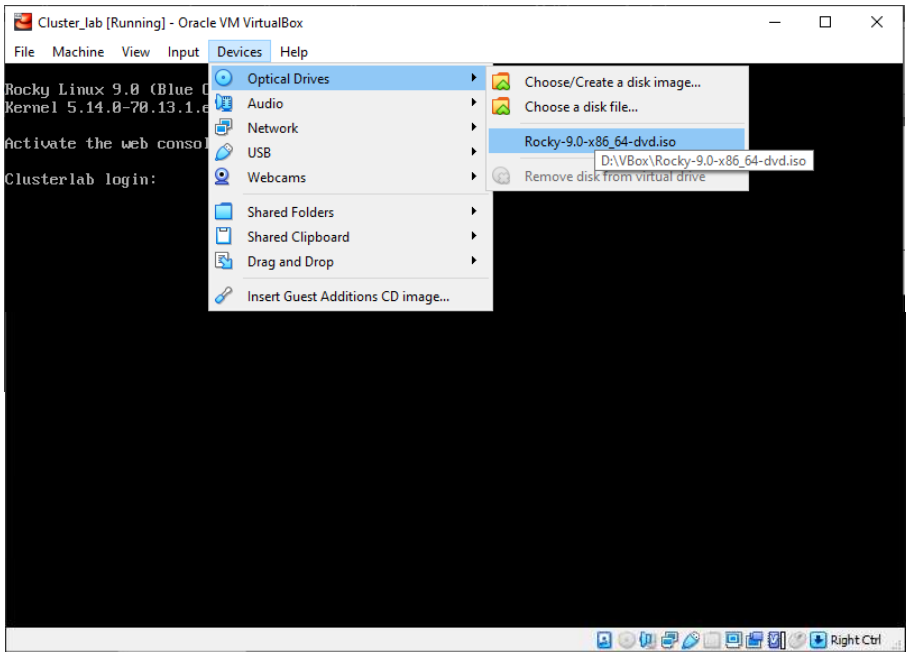

Рис. 1.62. **Підключення образу ОС *Linux* до ВМ**

Якщо працюєте під користувачем *user*, то варто отримати права адміністратора *root* на ГС *Clusterlab* (див. рис. 1.60). Створюємо файл сховища AppStream.repo (рис. 1.63).

Рис. 1.63. **Створення файла сховища *AppStream.repo***

Заносимо дані в файл сховища *AppStream.repo* (рис. 1.64).

Рис. 1.64. **Вміст файла сховища AppStream.repo**

Створюємо файл сховища *BaseOS.repo* (рис. 1.65).

Рис. 1.65. **Створення файла сховища BaseOS.repo**

Заносимо дані в файл сховища *BaseOS.repo* (рис. 1.66).

Рис. 1.66. **Вміст файла сховища BaseOS.repo**

Видаляємо файли сховищ, установлені за замовчуванням (рис. 1.67).

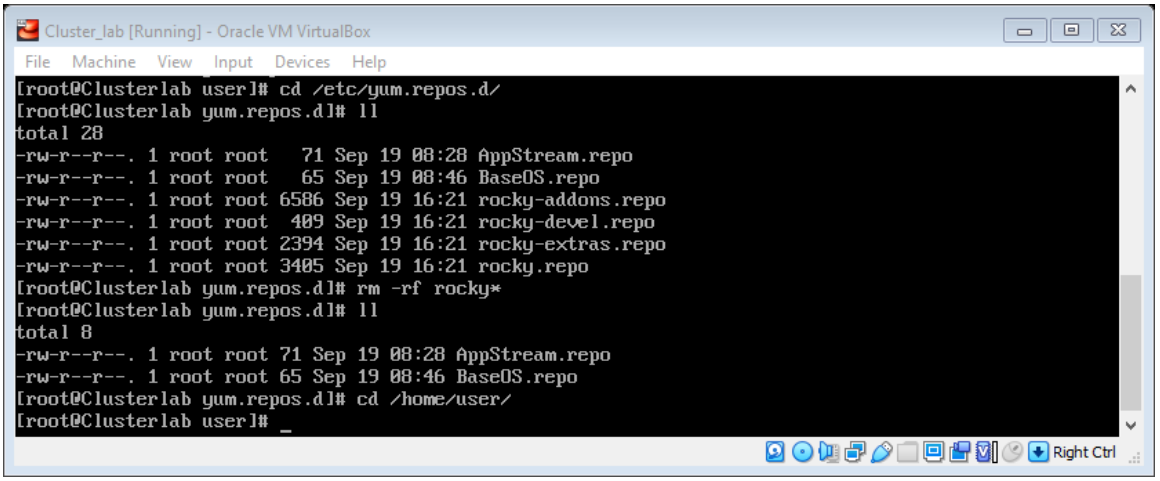

Рис. 1.67. **Виділення файлів сховищ за замовчуванням**

Приєднуємо підключений образ ОС *Linux* до директорії *media* (рис. 1.68).

Рис. 1.68. **Приєднуємо образ ОС *Linux***

Установлюємо необхідні пакети для роботи за протоколом NFS (рис. 1.69).

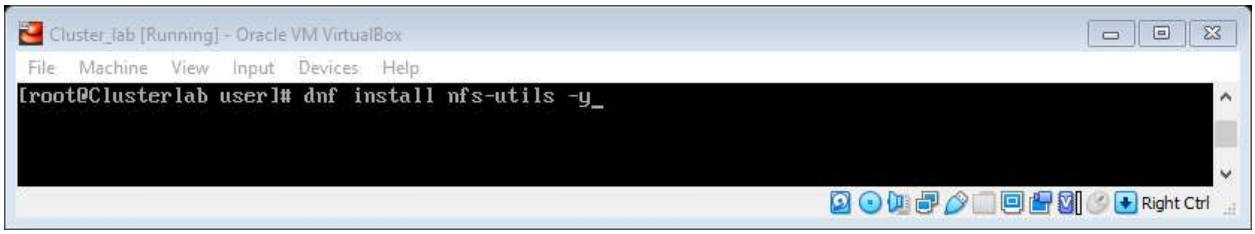

Рис. 1.69. **Установлення пакетів для протоколу NFS**

Зупиняємо службу брандмауера та перевіряємо його статус (рис. 1.70).

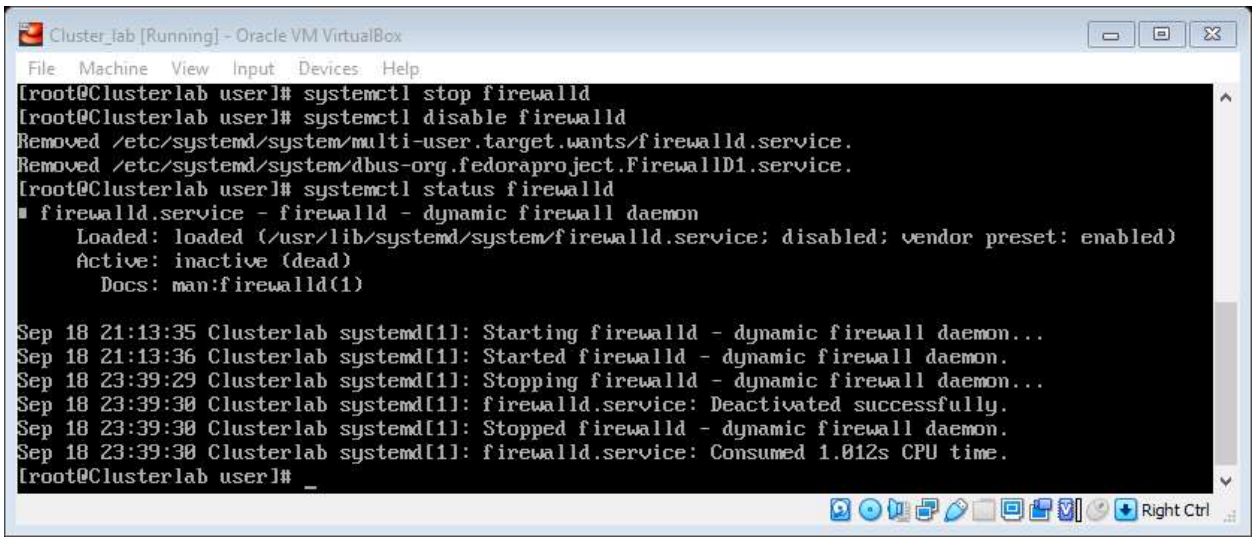

Рис. 1.70. **Відключення брандмауера**

Редагуємо конфігураційний файл *exports* служби протоколу NFS командою: *nano/etc/exports*. Вписуємо рядок: */home* 172.16.0.0/24 (*rw,no_root_squash*) (рис. 1.71), де *rw* – дозволяє отримати права на читання та запис в загальну директорію, *no_root_squash* – дозволяє користувачу *root* ОВ підключатися до вказаної директорії.

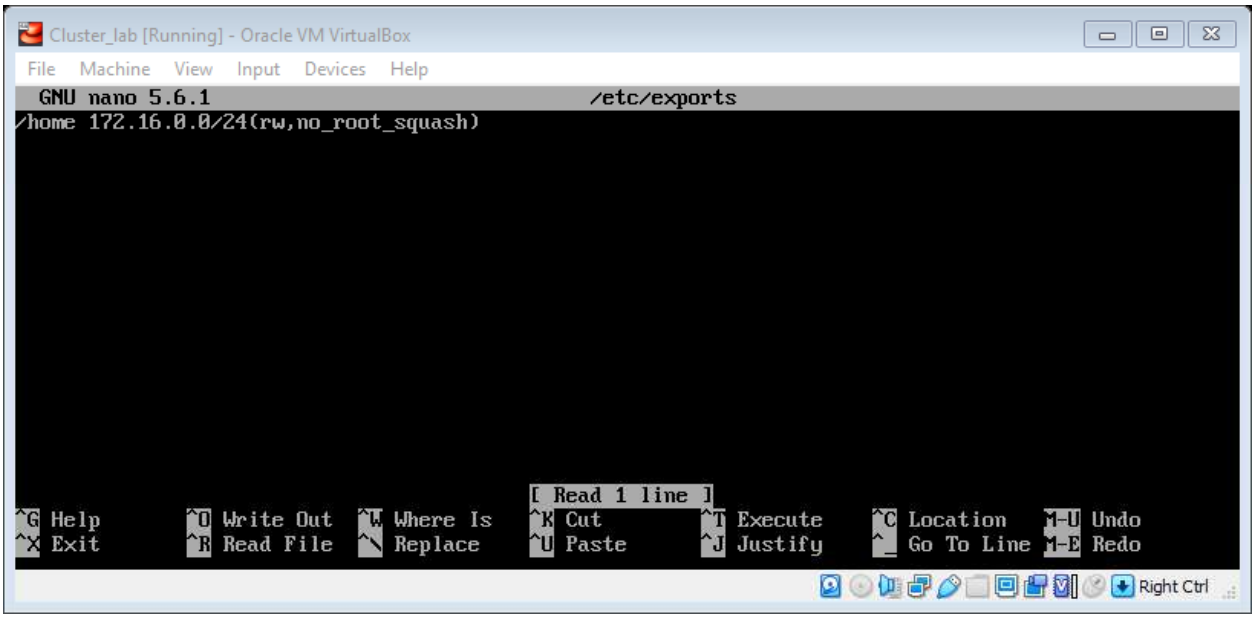

Рис. 1.71. **Файл налаштувань exports**

Перечитуємо налаштування, запускаємо, перевіряємо статус, перевіряємо налаштування та дозволяємо завантаження під час старту системи служби протоколу NFS (рис. 1.72).

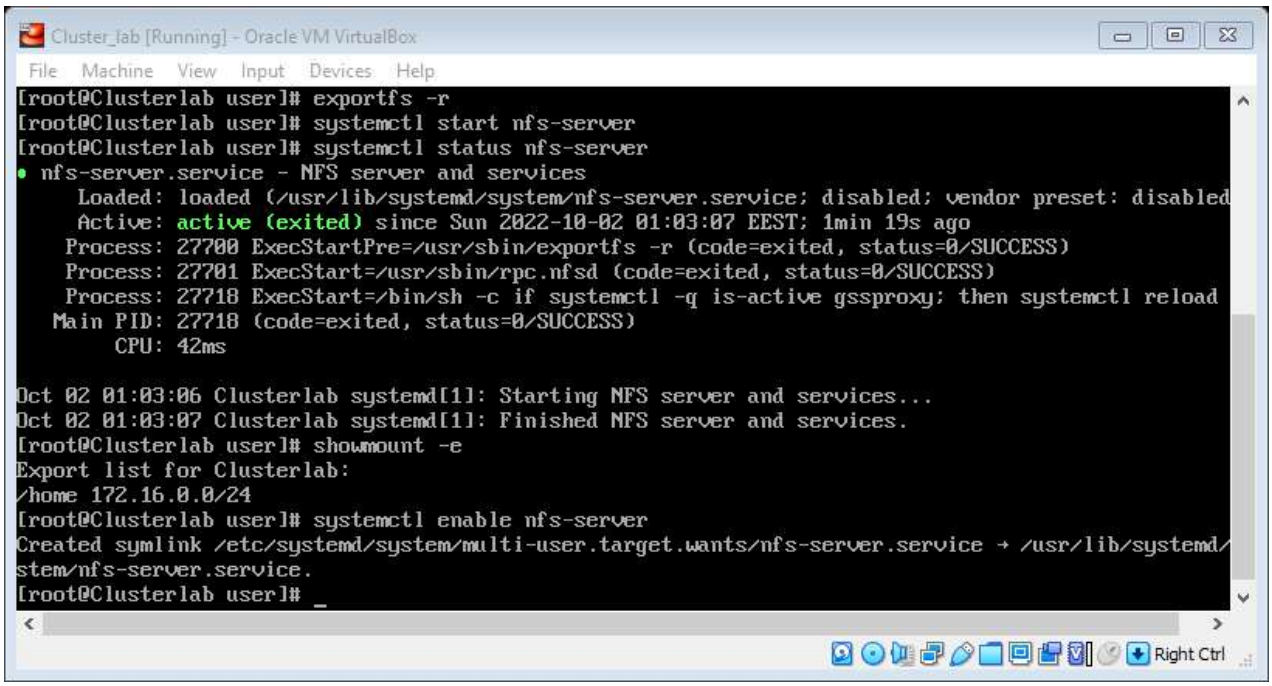

Рис. 1.72. **Налаштування NFS**

Переходимо до налаштувань ОВ *Node1lab*. Підключаємо образ дистрибутиву ОС *Linux*. Входимо під користувачем *user*. Отримуємо права адміністратора *root*. Створюємо файли сховищ *AppStream.repo* та *BaseOS.repo*, такі самі, як для ГС *Clusterlab*.

Видаляємо файли сховищ, установлені за замовчуванням. Приєднуємо підключений образ ОС *Linux* до директорії *media*. Установлюємо необхідні пакети для роботи з протоколом NFS (лише встановлюємо, ніяких налаштувань робити не треба). Зупиняємо службу брандмауера та перевіряємо її статус. Ці дії, повністю повторюють дії для ГС *Clusterlab*, тому окремо розглядати їх не має потреби.

Перевіряємо доступні віддалені ресурси на ГС *Clusterlab* (рис. 1.73).

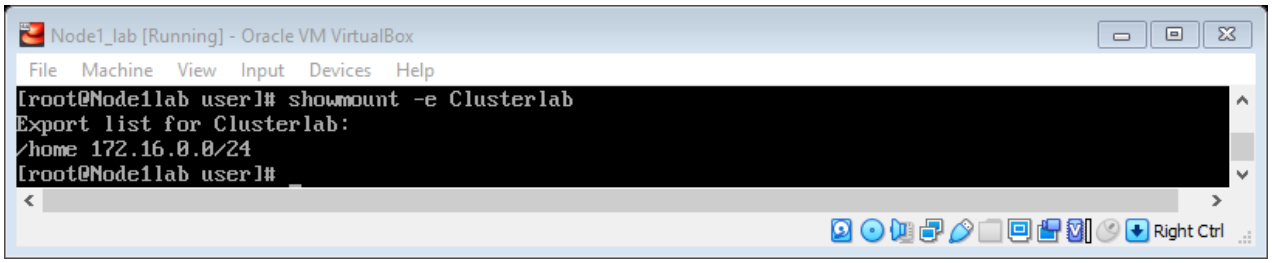

Рис. 1.73. **Доступні віддалені ресурси на ГС *Clusterlab***

Налаштовуємо підключення віддаленої директорії *home* з ГС *Clusterlab* на ОВ *Node1ab* під час завантаження ОС. Для цього треба відредагувати файл *fstab* (рис. 1.74).

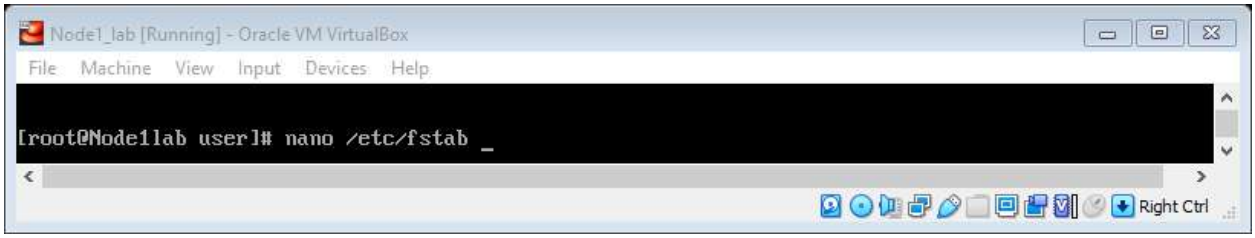

Рис. 1.74. **Відкриття файла *fstab* для редагування**

У кінець файла треба додати наступний рядок для монтування віддаленої директорії *home*: Clusterlab:/home /home nfs defaults 0 0 (рис. 1.75).

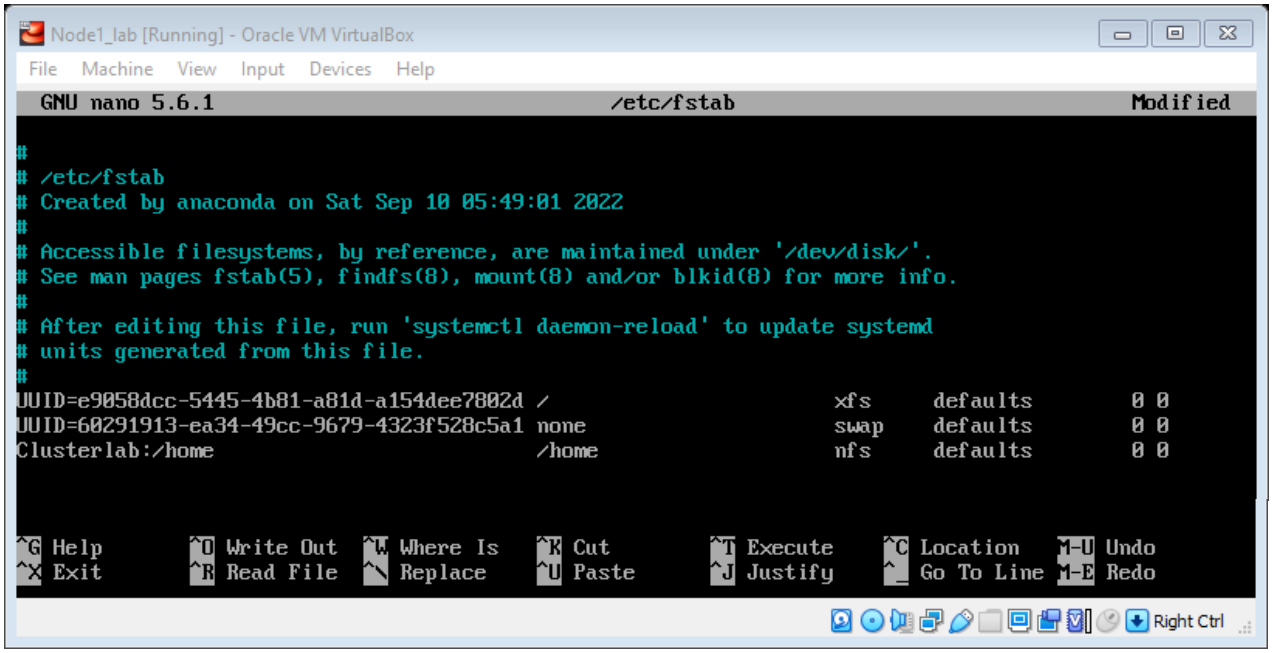

Рис. 1.75. **Редагування файла *fstab***

Відключаємо *службу SELinux* (рис. 1.76).

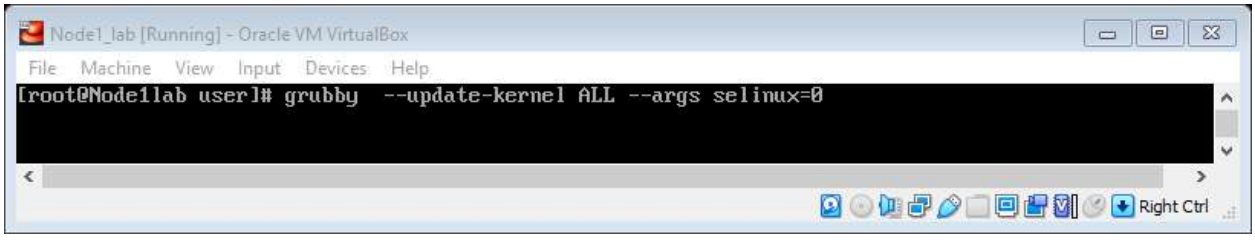

Рис. 1.76. **Відключення служби *SELinux***

Далі потрібно від'єднати диск з образом ОС *Linux* (рис. 1.77) та вилучити його з образу ВМ *Node1_lab* (рис. 1.78). Якщо цього не зробити, то система під час перезавантаження буде завантажуватися з образу ОС *Linux*, почне встановлення ОС знову. Як ці операції буде виконано, то перезавантажуємо ОВ *Node1lab* для перевірки працездатності та застосування конфігураційних змін (рис. 1.79).

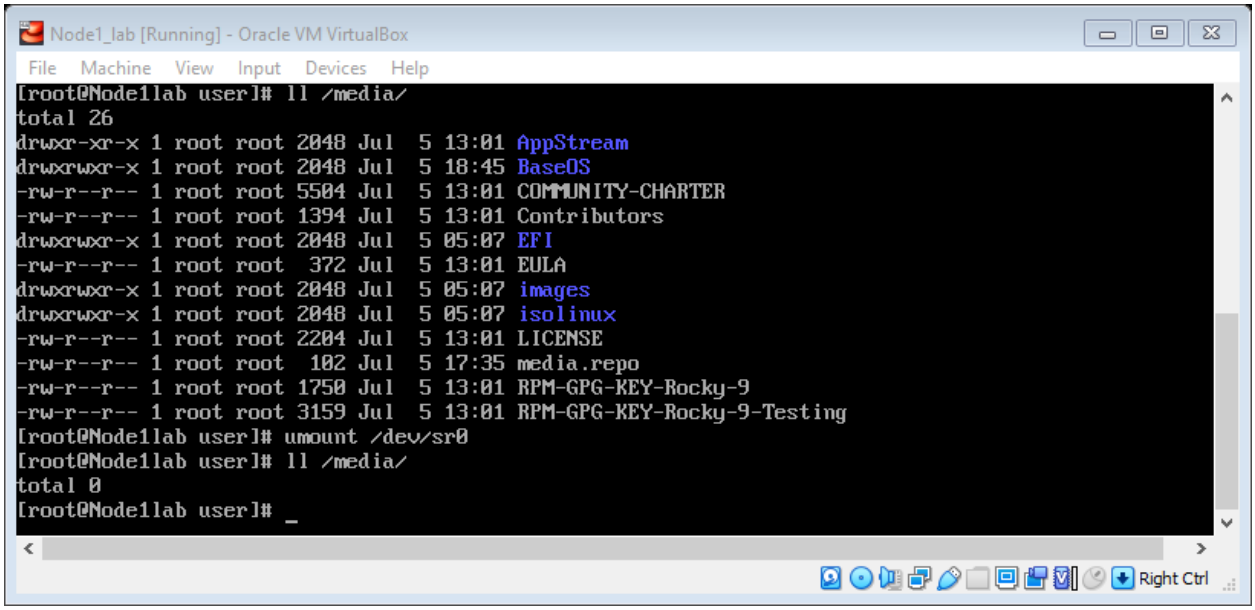

Рис. 1.77. **Від'єднання диска з образом ОС *Linux***

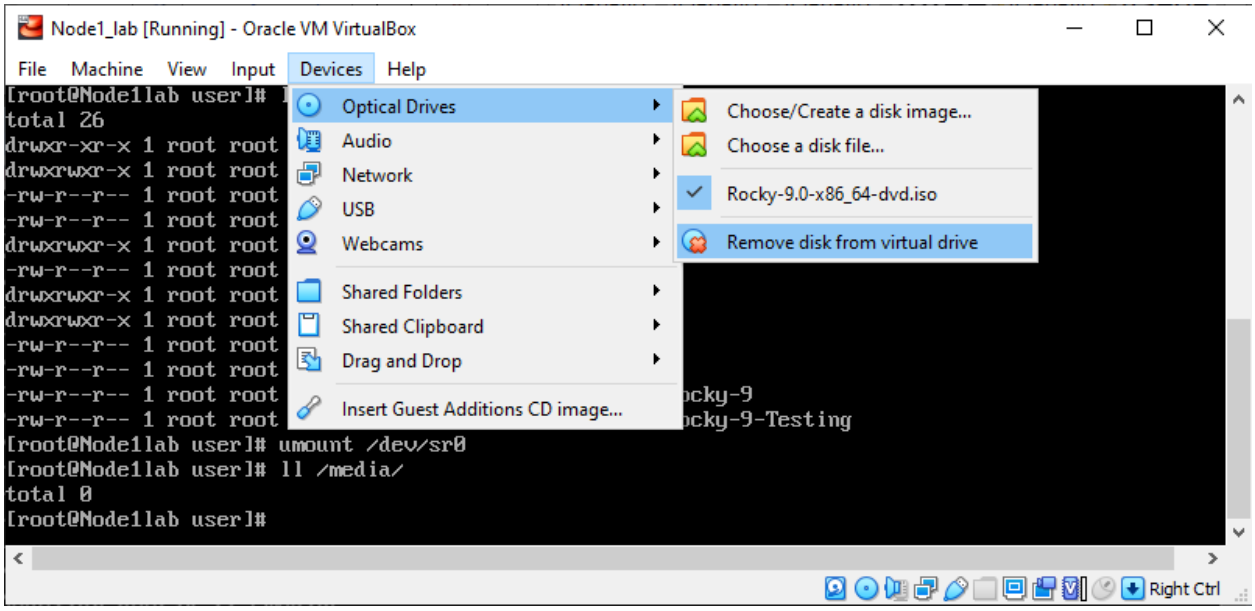

Рис. 1.78. **Вилучення диска з образом ОС від образу ВМ Node1_lab**

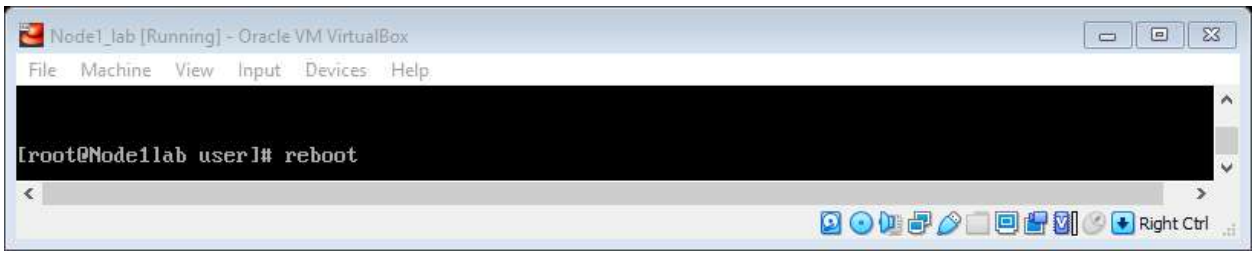

Рис. 1.79. **Перезавантаження ОВ Node1lab**

Переходимо до налаштувань ОВ *Node2lab*. Для цього треба виконати всі ті самі дії та в тій самій послідовності, що й під час налаштування ОВ *Node1lab*.

Після налаштування обох ОВ *Node1lab* та *Node2lab*, повертаємося до ГС *Clusterlab* та відключаємо для нього службу *SELinux* (див. рис. 1.76).

Під'єднуємо диск з образом ОС *Linux*. Вилучаємо цей диск з образу ВМ *Cluster_lab*. Усі дії аналогічні тим, що виконувалися для ОВ *Node1lab* та ОВ *Node2lab*.

Для остаточної перевірки правильності налаштувань служби протоколу NFS на кластері потрібно спочатку в правильній послідовності вимкнути всі його вузли, а потім ввімкнути.

Першими повинні вимикатися вузли *Node2lab* та *Node1lab*, а останнім – ГС *Clusterlab*. Для вимкнення вузлів на кожному з них з-під адміністратора *root* повинна виконатися команда *poweroff* (рис. 1.80).

Рис. 1.80. **Вимкнення вузла**

Вмикати вузли потрібно в такій послідовності: першим вмикається ГС *Clusterlab*. Після його остаточного завантаження вмикаються ОВ *Node1lab* та *Node2lab*. Їх можна вмикати одночасно. Якщо під час завантаження, ОВ перейшли на запрошення до введення імені користувача, налаштування служби NFS можна вважати вдалим. Підключення віддаленої директорії можна перевірити командою df-h, виконаною на ОВ кластера під обліковим записом користувача *user* (рис. 1.81).

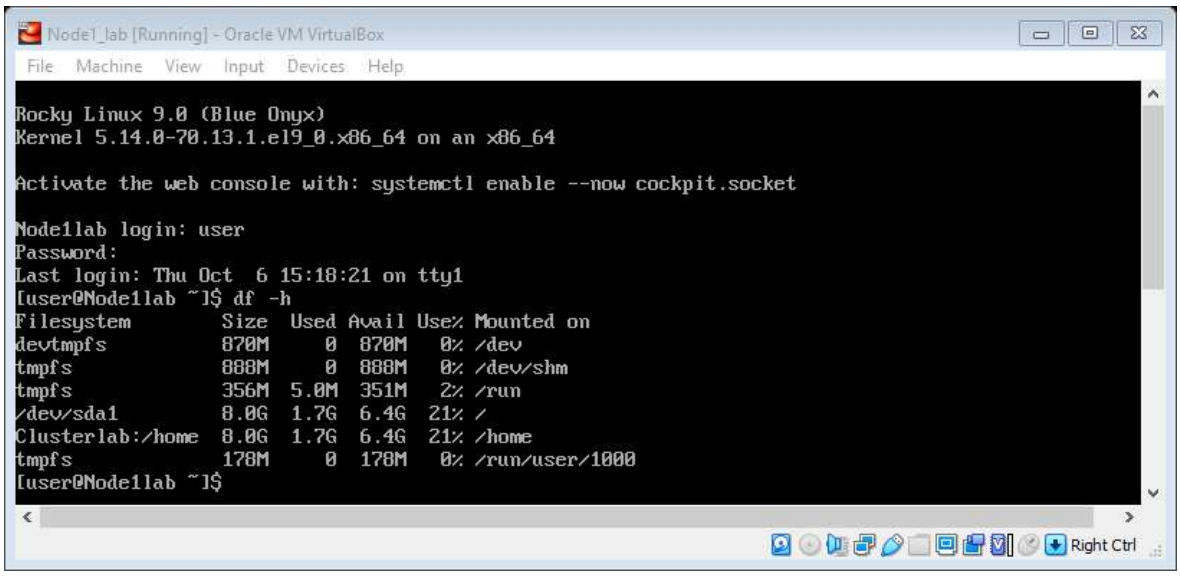

Рис. 1.81. **Перевірка підключення віддаленої директорії**

Налаштування служби протоколу *SSH* для користувача *user*

Після перезавантаження трьох ВМ заходимо на ГС *Clusterlab* під користувачем *user* та перевіряємо роботу протоколу *SSH* (рис. 1.82). Для цього, використовуючи команду: *ssh Node1lab*, заходимо на ОВ *Node1lab*. Погоджуємося на продовження з'єднання з ОВ.

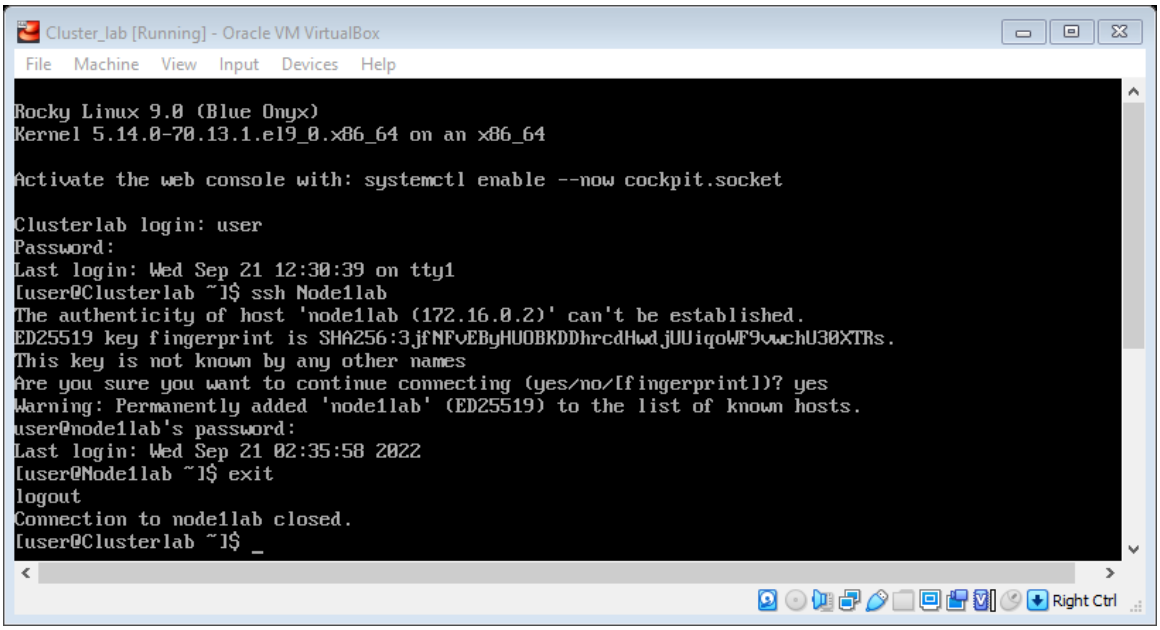

Рис. 1.82. **Перевірка роботи протоколу SSH**

На запрошення пароля вводимо пароль користувача *user*. Якщо всі дії виконано правильно, то ви побачите, що здійснено віддалений вхід на ОВ *Node1lab*. Це означає, що протокол *SSH* працює у вашій системі. Після перевірки повертаємося на ГС *Clusterlab* командою *exit*.

Переходимо до налаштувань безпарольного доступу по протоколу *SSH* для користувача *user* (рис. 1.83).

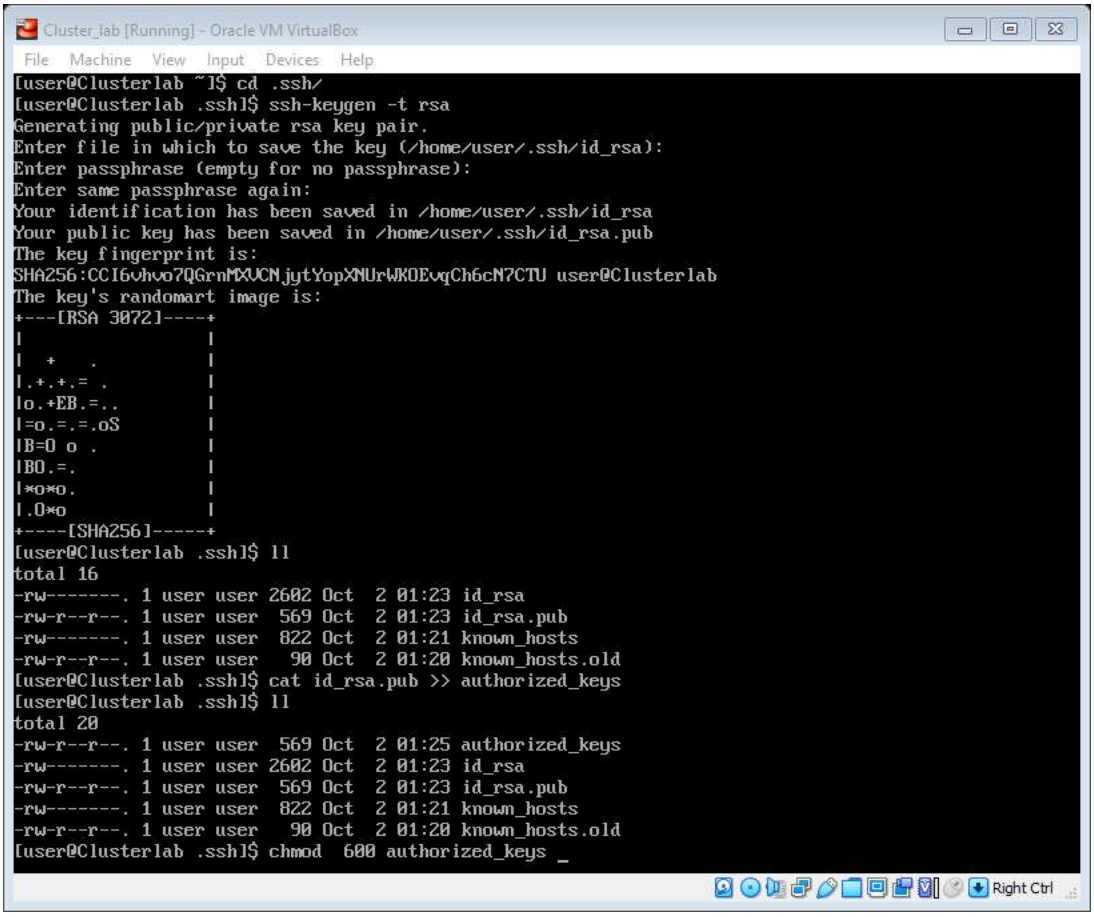

Рис. 1.83. **Створення ключів авторизації**

Заходимо в директорію *.ssh*. Генеруємо відкритий та закритий ключі *ssh rsa* для клієнтського з'єднання. На чотири запрошення введення паролів нічого не пишемо, лише натискаємо клавішу *Enter*. Командою ll перевіряємо, що генерація ключів пройшла вдало, вони були створені в нашій директорії. Далі треба створити ключ авторизації *authorized_keys* для безпарольного доступу командою *cat.* Та перевірити вдале завершення операції командою ll.

Як можна бачити на рис. 1.83, ключ було створено з атрибутами, що дозволяють його читання не тільки володарю, але й групі, до якої належить володар та ін. З такими значеннями атрибута неможливо забезпечити достатню міру безпеки з'єднання.

Тому атрибути цього файла потрібно змінити на надання можливості запису та читання лише для володаря ключа авторизації. Для цього використовується команда *chmod* з параметром 600. Перевіряємо нове значення атрибутів командою ll (рис. 1.84).

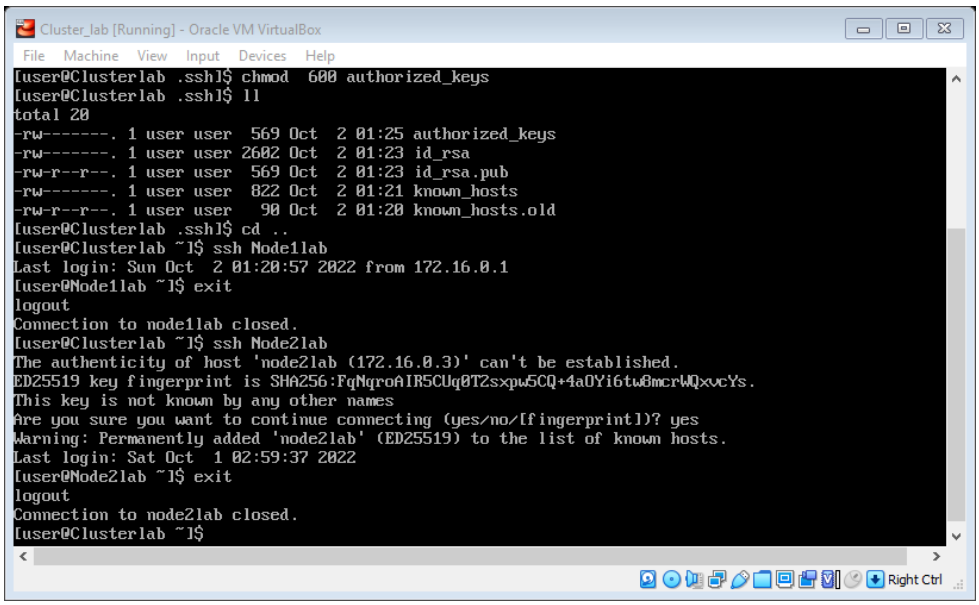

Рис. 1.84. **Перевірка атрибутів файла *authorized_keys* та з'єднання *SSH***

Виходимо з директорії *.ssh* командою *cd* (див. рис. 1.84). Перевіряємо спочатку *ssh* з'єднання ГС *Clusterlab* з ОВ *Node1lab*. Повертаємося назад командою *exit*, а потім й ГС *Clusterlab* з ОВ *Node2lab* (див. рис. 1.84), і також, повертаємося назад командою *exit*.

Для остаточної перевірки роботи протоколу SSH треба виконати послідовний вхід на всі вузли в обох напрямах, використовуючи команду *ssh* (рис. 1.85).

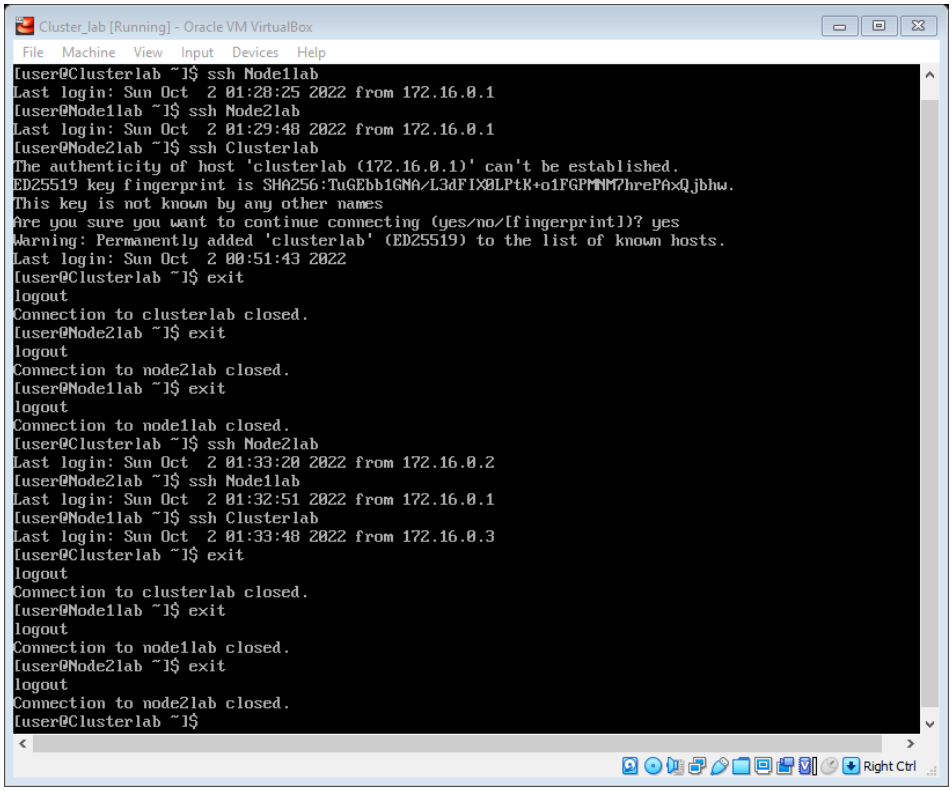

Рис. 1.85. **Перевірка *SSH* з'єднання всіх вузлів в обох напрямах**

В одному напрямі: ГС *Clusterlab*–>ОВ *Node1lab*–>ОВ *Node2lab*–>ГС *Clusterlab*. Після чого повернутися на ГС *Clusterlab* повтором введення команди *exit*, а потім у зворотньому напрямі: ГС *Clusterlab*–>ОВ *Node2lab*–>ОВ *Node1lab*–>ГС *Clusterlab*. Після чого треба повернутися на ГС *Clusterlab* повтором команди *exit*.

У процесі виконання входів може з'являтися питання на дозвіл до під'єднання, даєте згоду. Якщо всі операції виконуються успішно, налаштування безпарольного доступу для користувача *user* можна вважати закінченим.

Контрольні запитання до лабораторної роботи 1

1. Яку Ви знаєте операційну систему, програмне забезпечення до якої поширюється на безкоштовній основі?
2. Як виконати вхід у систему з обліковим записом *root*?
3. Як перевірити правильність налаштування КМ та поясніть параметри введення команди?
4. Дайте характеристику параметрам, які вводяться під час налаштування мережевих інтерфейсів?
5. Що означають поняття основна операційна система та гостьова ОС?
6. Для чого потрібен файл */etc/hosts*?
7. Як перезавантажити налаштування без перезавантаження машини?
8. Як подивитися налаштування мережевих інтерфейсів?
9. Для чого потрібна команда *mount*?
10. Для чого потрібна команда *ping*?
11. Для чого потрібен та як працює протокол NFS?
12. Яка основна відмінність протоколу NFS від FTP?
13. Для чого потрібен та як працює протокол SSH?

Лабораторна робота 2
Налаштування системи *SLURM* на обчислювальному кластері

Мета лабораторної роботи:

Вивчення встановлення та налаштування ПЗ у ОС *Linux (Rocky Linux 9)*.

Набуття практичних навичок з налаштування системи управління ресурсами (СУР) *SLURM*.

Набуття практичних навичок з адміністрування обчислювального кластера.

Завдання:

На основі виконаної лабораторної роботи 1 встановити та налаштувати СУР *SLURM* на головному сервері (ГС) та обчислювальних вузлах (ОВ) кластера.

За допомогою відповідних команд перевірити стан кластера та його конфігурацію.

Запустити на виконання завдання на кластері та перевірити його роботу.

Порядок виконання лабораторної роботи

Перед виконанням лабораторної роботи отримайте у викладача архів з необхідним програмним забезпеченням.

Відкриваємо менеджер віртуальних машин (ВМ) (рис. 2.1).

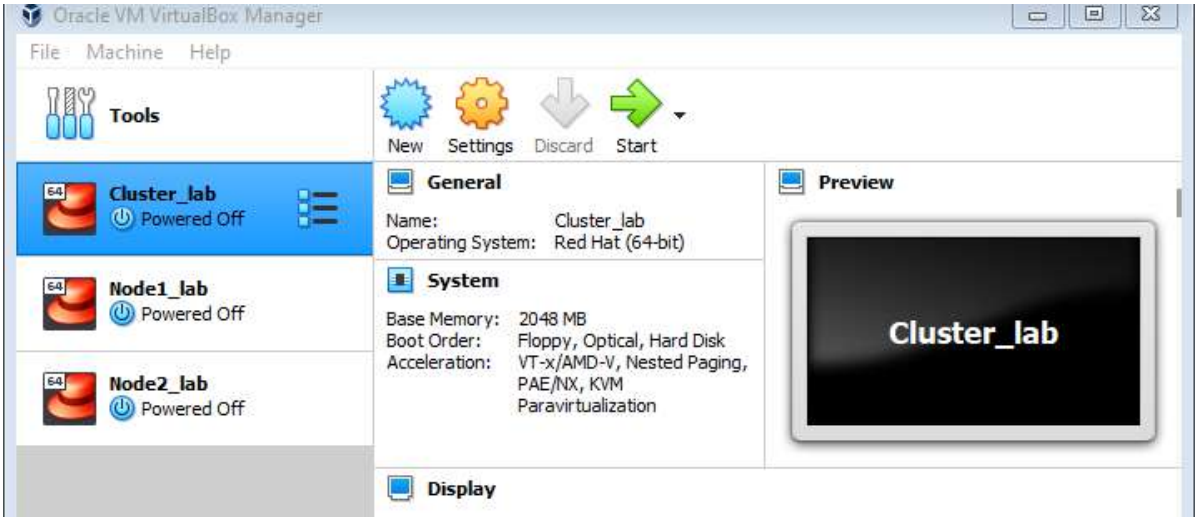

Рис. 2.1. **Менеджер ВМ**

Переходимо до налаштувань ВМ *Cluster_lab*, натискаючи кнопку *Settings*. Заходимо в розділ створення загальних директорій (рис. 2.2).

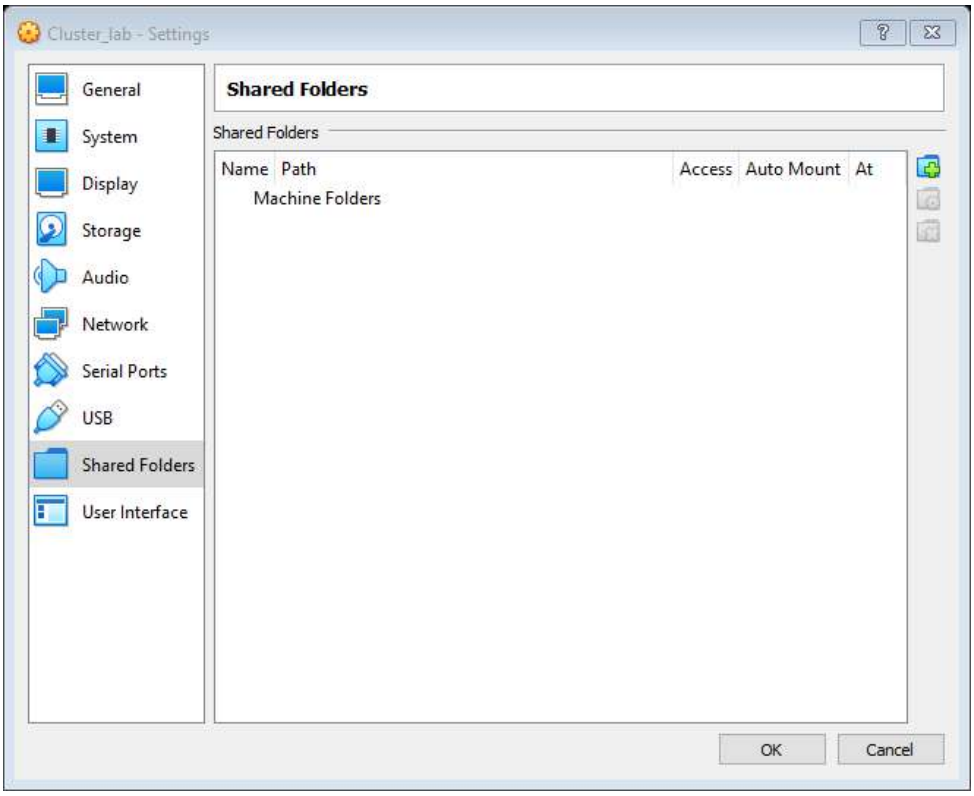

Рис. 2.2. **Розділ створення загальних директорій ВМ**

Створюємо загальну директорію, вказуючи існуючу директорію з файлами, що треба перенести з основної операційної системи (ОС) у гостьову ОС. У якості точки монтування оберіть шлях до директорії у гостьовій ОС, до якої буде приєднано директорію основної ОС (рис. 2.3).

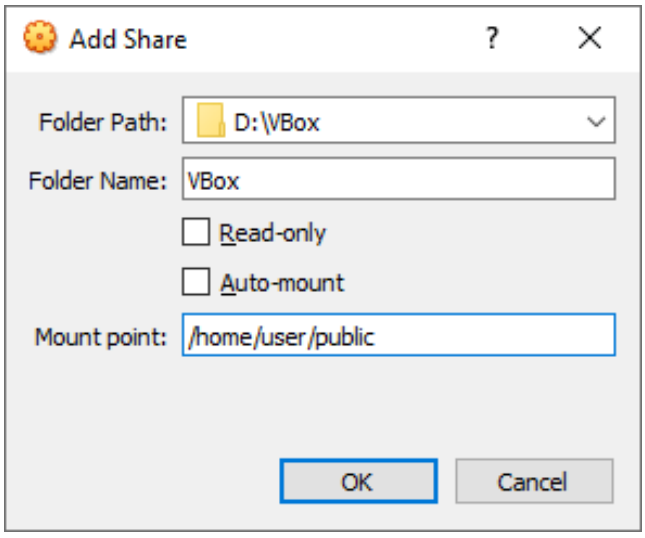

Рис. 2.3. **Додавання загальної директорії**

Якщо такої директорії не існує, то її можна створити після входу на ГС *Clusterlab*. На цьому налаштування ВМ *Cluster_lab* закінчено (рис. 2.4).

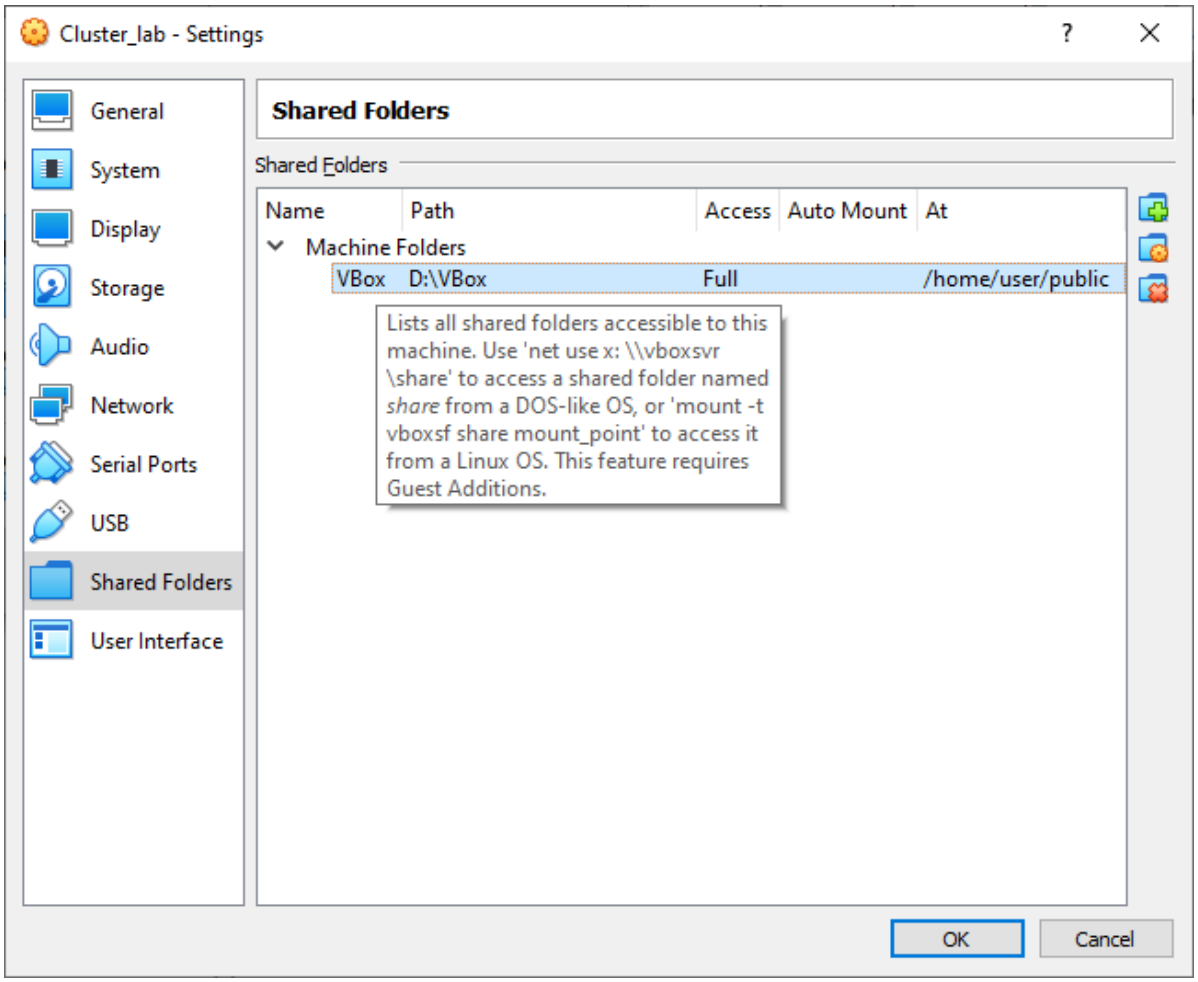

Рис. 2.4. **Створена загальна директорія**

Завантажуємо ВМ *Cluster_lab*. Підключаємо дистрибутив ОС *Linux* (рис. 2.5).

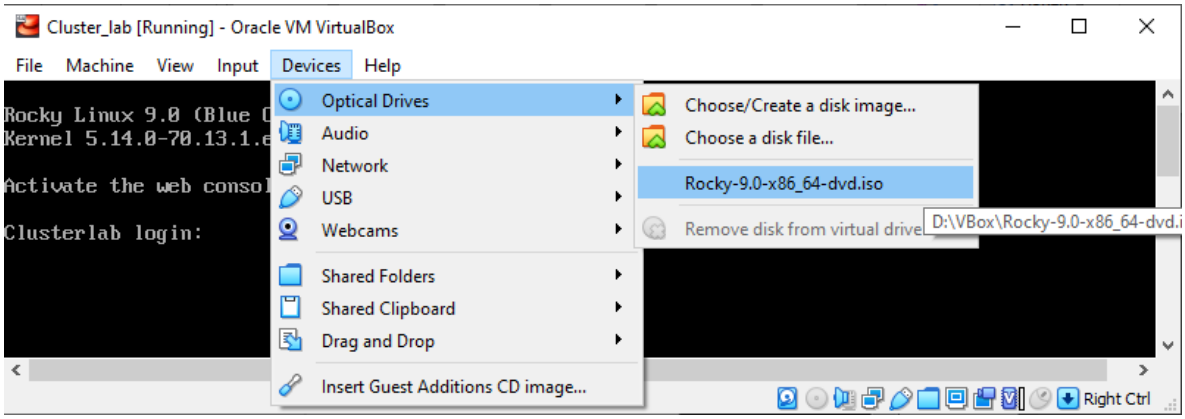

Рис. 2.5. **Підключення дистрибутиву ОС *Linux***

Заходимо в систему під користувачем *user* (рис. 2.6) та отримуємо права адміністратора системи.

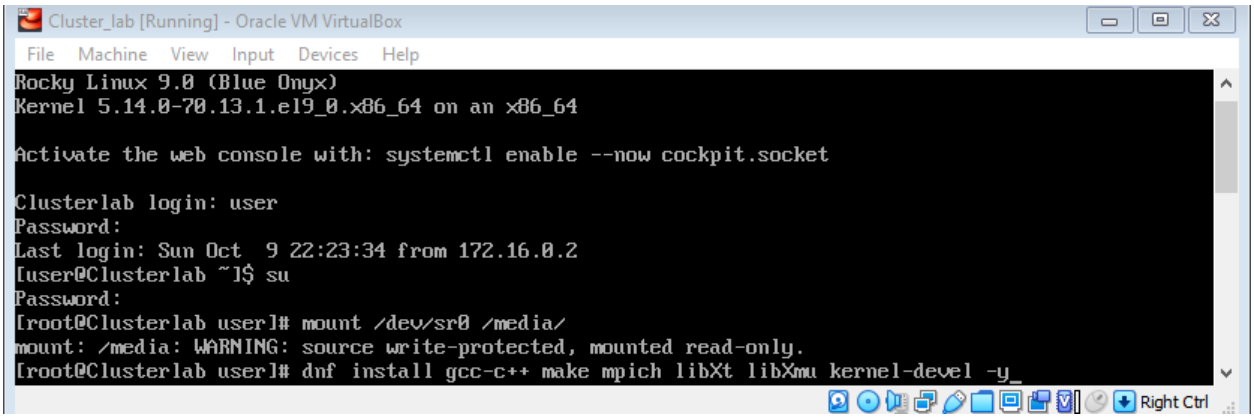

Рис. 2.6. **Установлення необхідних пакетів**

Приєднуємо підключений раніше дистрибутив ОС *Linux* до директорії *media*. Далі встановлюємо наступні пакети командою: *dnf install gcc-c++ make mpich libXt libXmu kernel-devel*, які необхідні для налаштування доповнень для гостьової ОС та його подальшого функціонування. Після закінчення установки треба від'єднати дистрибутив ОС *Linux* від директорії *media* (рис. 2.7).

Рис. 2.7. **Від'єднання дистрибутиву ОС *Linux***

Вилучаємо образ диска з ВМ *Cluster_lab* (рис. 2.8).

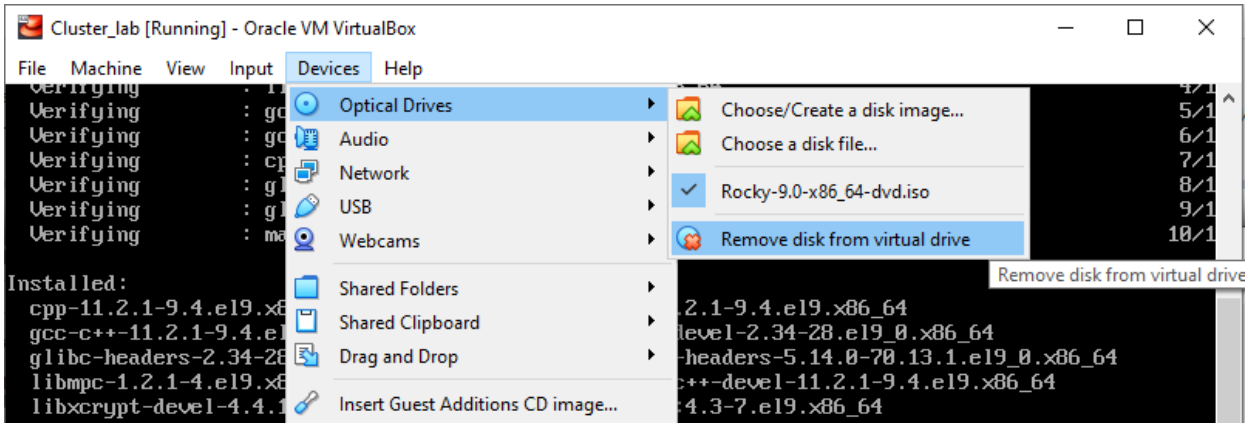

Рис. 2.8. **Вилучення образу диска з ВМ *Cluster_lab***

Далі вставляємо образ диска з доповненнями для гостьової ОС у ВМ *Cluster_lab* (рис. 2.9).

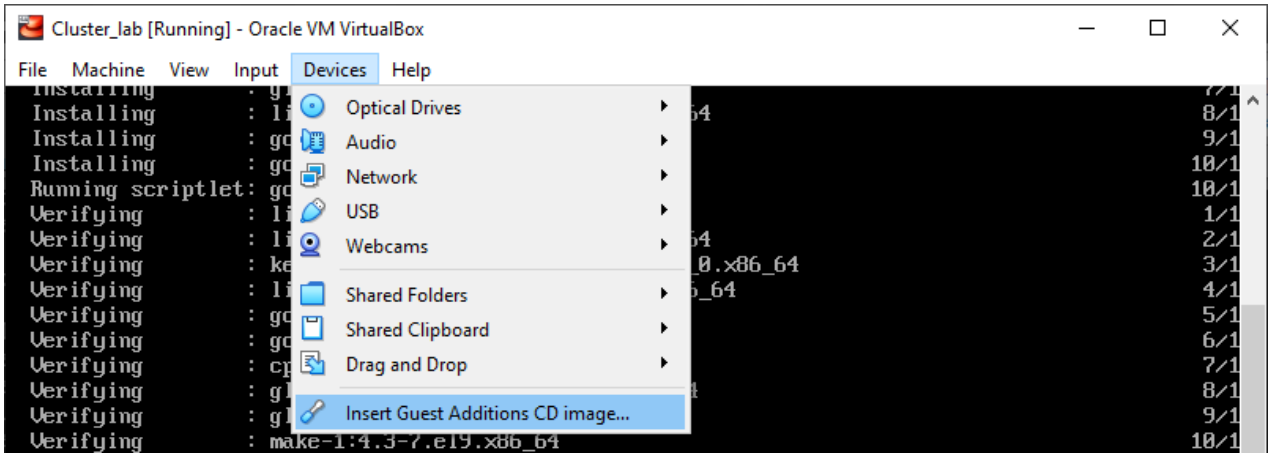

Рис. 2.9. **Підключення образу з доповненнями для гостьової ОС**

Приєднуємо образ з доповненнями до гостьової ОС до директорії *media* (рис. 2.10).

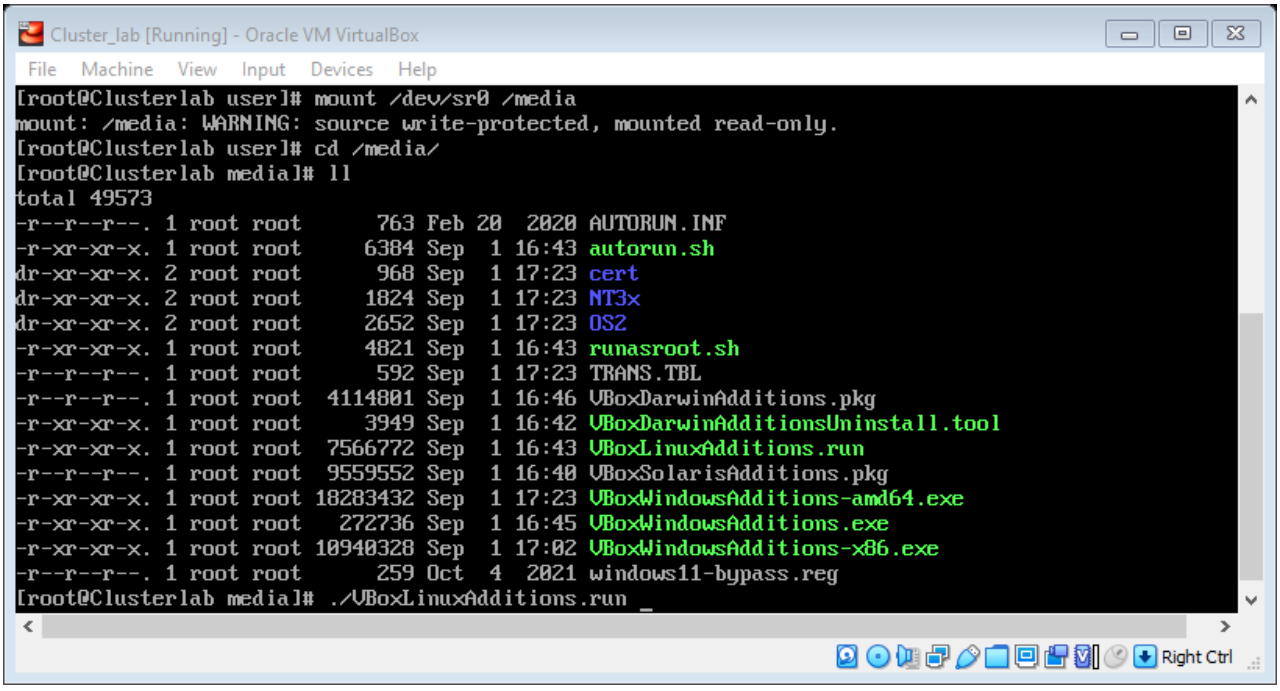

Рис. 2.10. **Установлення доповнень до гостьової ОС**

Далі переходимо в неї та виконуємо *script VBoxLinuxAdditions.run*, встановлення доповнень в гостьову ОС. І повертаємося в користувача

user (рис. 2.11). Для отримання файлів з основної (хостової) ОС треба створити директорію *public* (рис. 2.12). Потім повертаємося до облікового запису адміністратора та приєднуємо директорію з основної ОС до створеної директорії *public*. Перевіряємо вміст підключеної директорії.

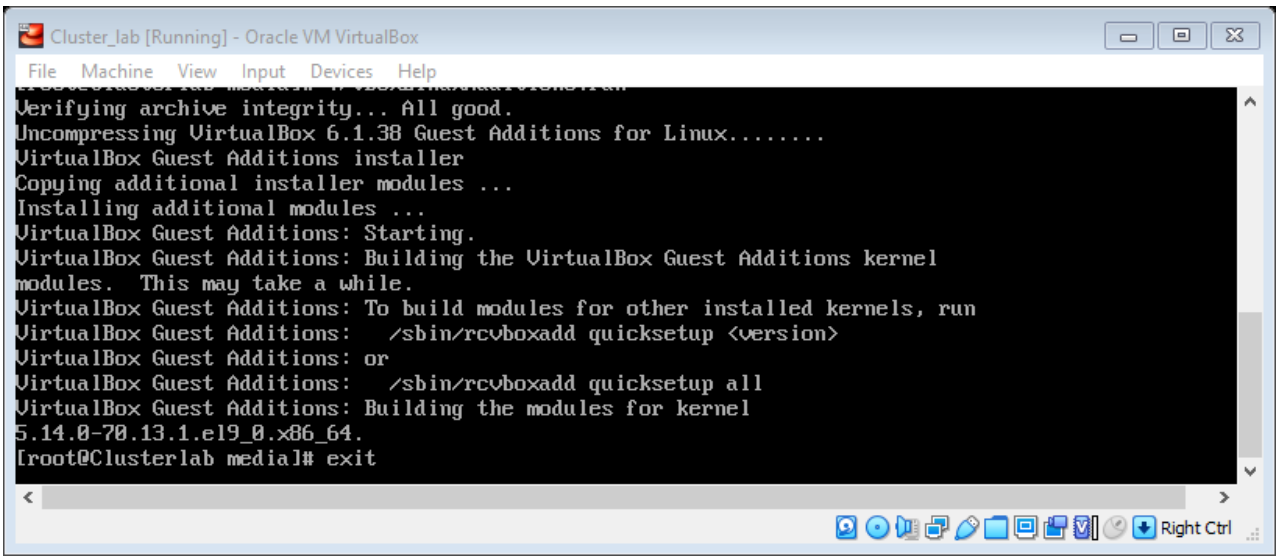

Рис. 2.11. **Вихід з облікового запису адміністратора**

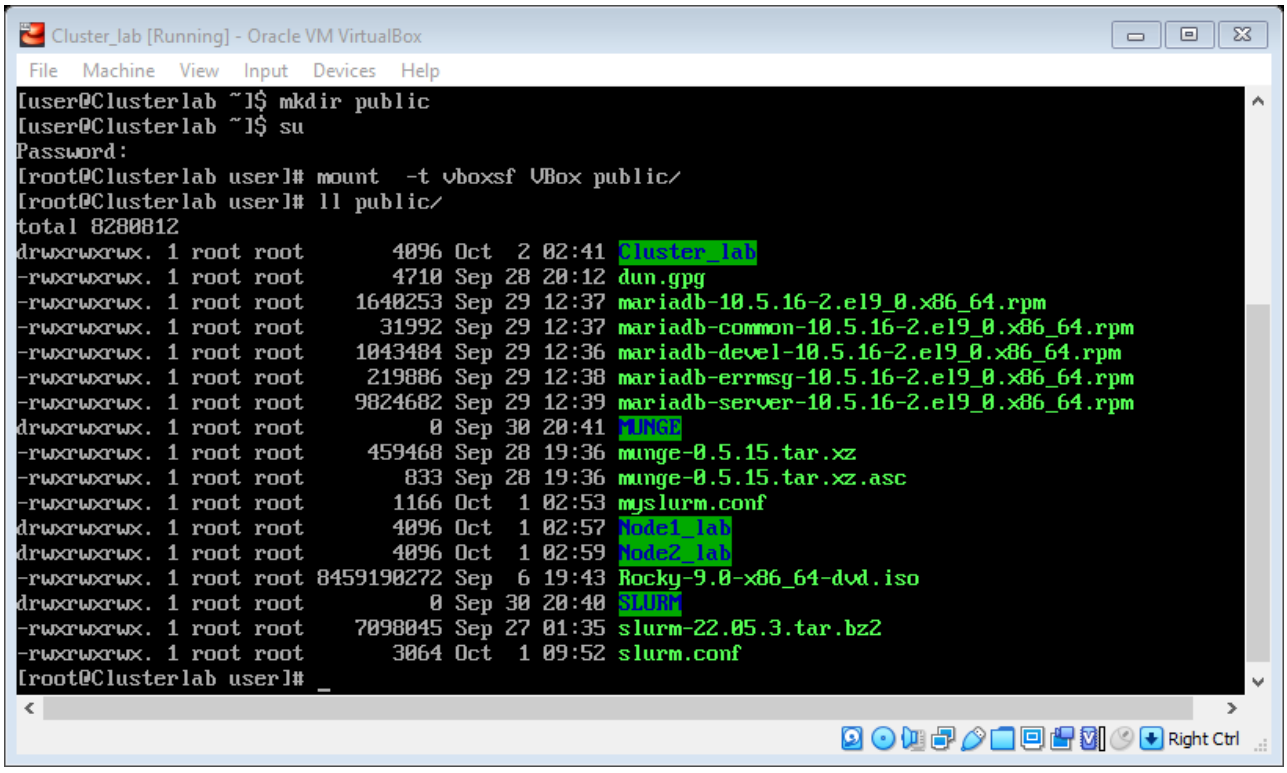

Рис. 2.12. **Приєднання директорії з основної ОС до гостьової ОС**

Якщо налаштування виконано правильно, то можна побачити однаковий вміст файлів у директоріях – як на основній, так і на гостьовій ОС. Для більш зручного подальшого налаштування менеджера ресурсів треба створити безпарольний доступ для облікового запису адміністратора.

Спочатку треба відредагувати файл конфігурації *sshd_config* (рис. 2.13). Для цього потрібно додати такий рядок: *PermitRootLogin yes* (рис. 2.14). Після збереження змін та виходу з редактора перезавантажуємо службу *sshd* (рис. 2.15).

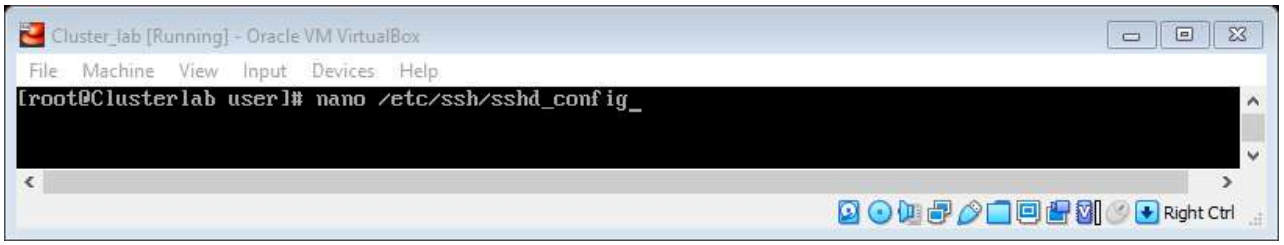

Рис. 2.13. **Відкриття файла конфігурації sshd_config**

Рис. 2.14. **Редагування файла конфігурації sshd_config**

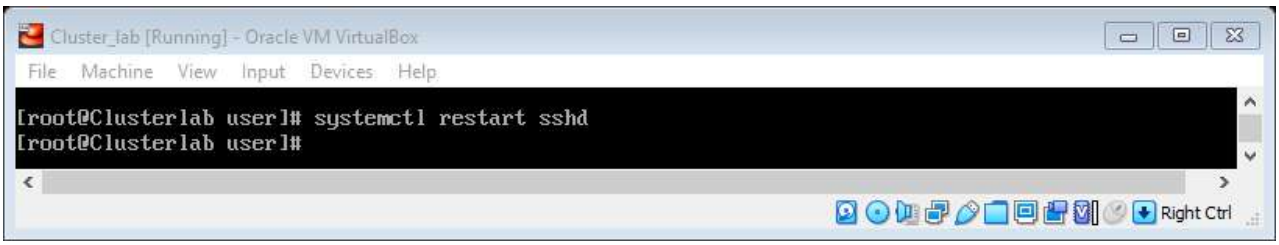

Рис. 2.15. **Перезавантаження служби sshd**

Відредагувати файл *sshd_config* та перезавантажити протокол SSH треба також на ОВ *Node1lab* та *ОВ Node2lab*.

З ГС *Clusterlab*, знаходячись під обліковим записом адміністратора *root*, заходимо на ОВ *Node1lab* для перевірки роботи протоколу *SSH* (рис. 2.16). На запит пароля треба ввести пароль адміністратора *root*.

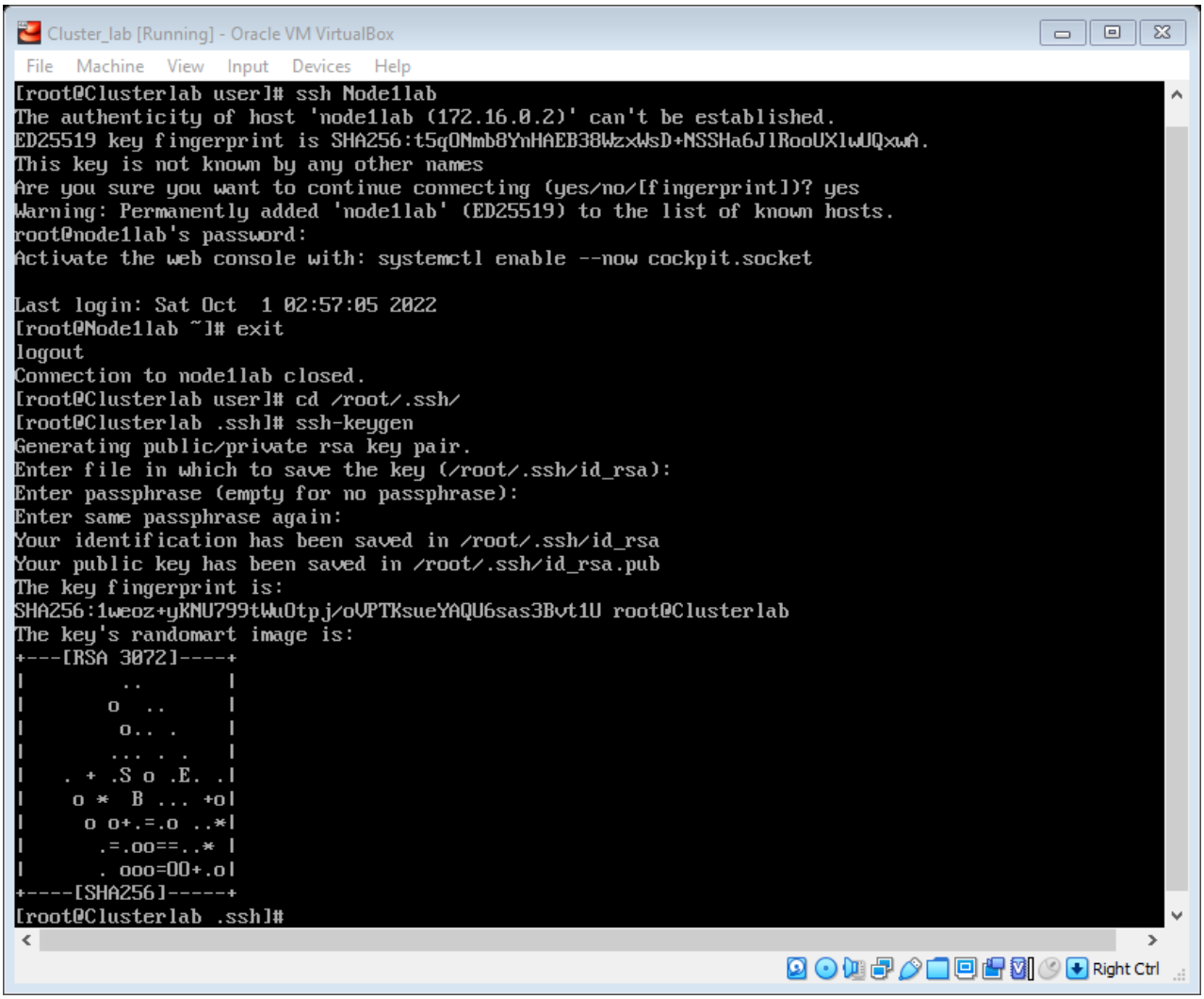

Рис. 2.16. **Генерація відкритого та закритого ключів ssh rsa**

Переходимо в директорію *.ssh*, що належить обліковому запису адміністратора *root*, та генеруємо в ній відкритий та закритий ключі *ssh rsa* для безпарольного з'єднання (див. рис. 2.16). Створюємо ключ авторизації *authorized_keys* (рис. 2.17). Атрибути цього файла треба змінити на значення, що дозволяють запис та читання лише для його власника.

Для цього використовуємо команду: *chmod 600* (див. рис. 2.17). Перевіряємо значення атрибутів командою ll (див. рис. 2.17).

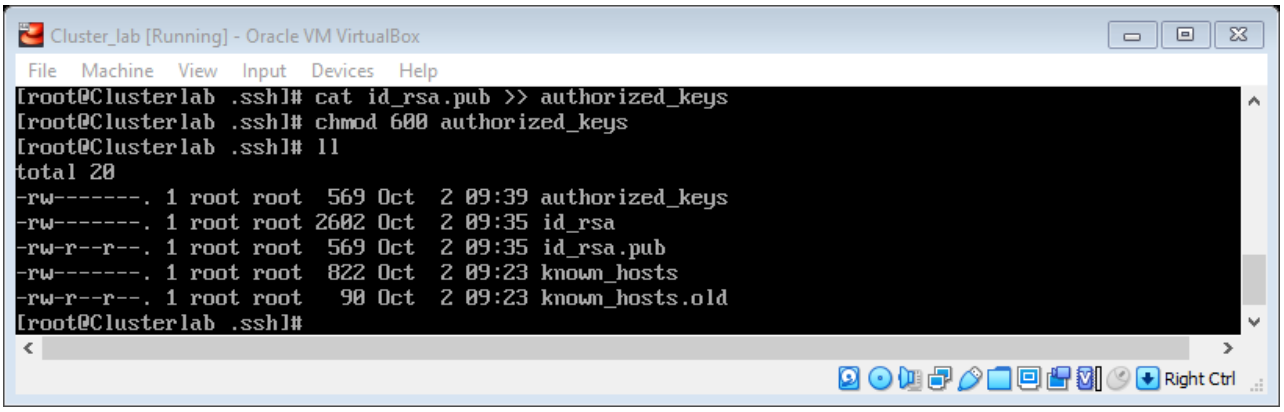

Рис. 2.17. **Створення ключа авторизації authorized_keys**

Копіюємо створені ключі в директорії *.ssh* облікових записів *root* на ОВ *Node1lab* та ОВ *Node2lab* (рис. 2.18), та якщо з'явиться питання на продовження з'єднання, то погодитися. На запит пароля ввести пароль адміністратора *root*.

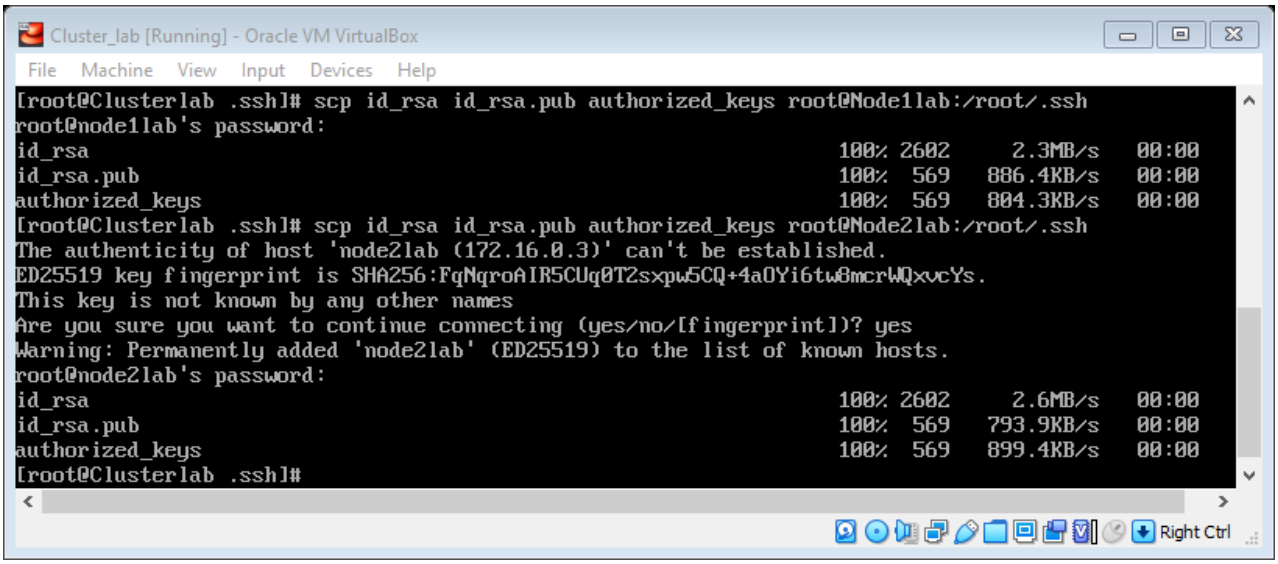

Рис. 2.18. **Копіювання ключів на ОВ Node1lab та ОВ Node2lab**

Виходимо з директорії *.ssh* та перевіряємо безпарольний доступ шляхом: ГС *Clusterlab*–>ОВ *Node1lab*–>ОВ *Node2lab*–>ГС *Clusterlab*, повертаємося назад командами *exit* (рис. 2.19). Якщо в процесі підключення виникає питання дозволу на з'єднання, то погоджуємося.

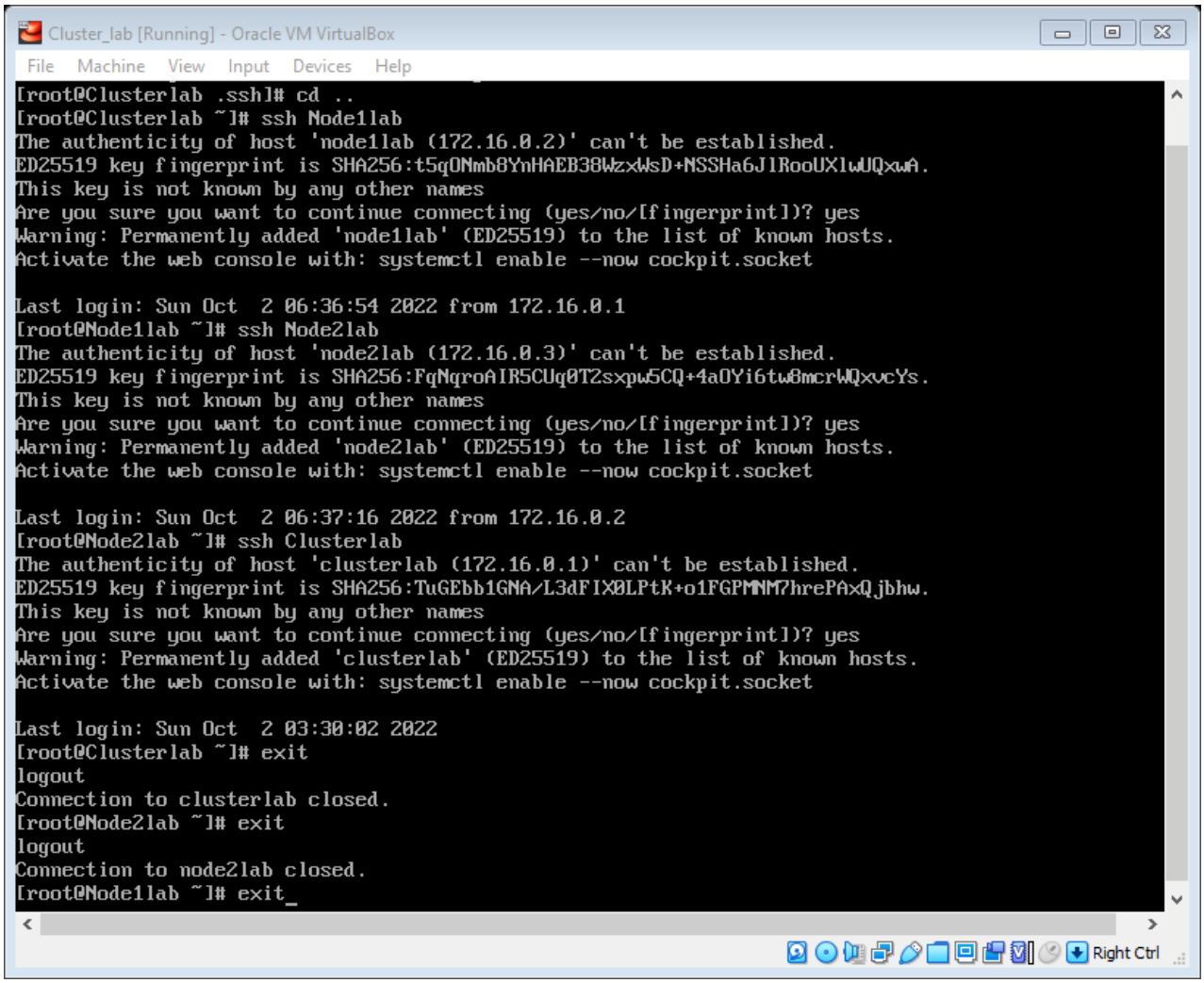

Рис. 2.19. **Перевірка безпарольного доступу**

Потім перевіряємо з'єднання в зворотному напрямі (рис. 2.20). У разі виникнення питання на з'єднання, також погоджуємося. Повертаємося назад, використовуючи команду *exit*.

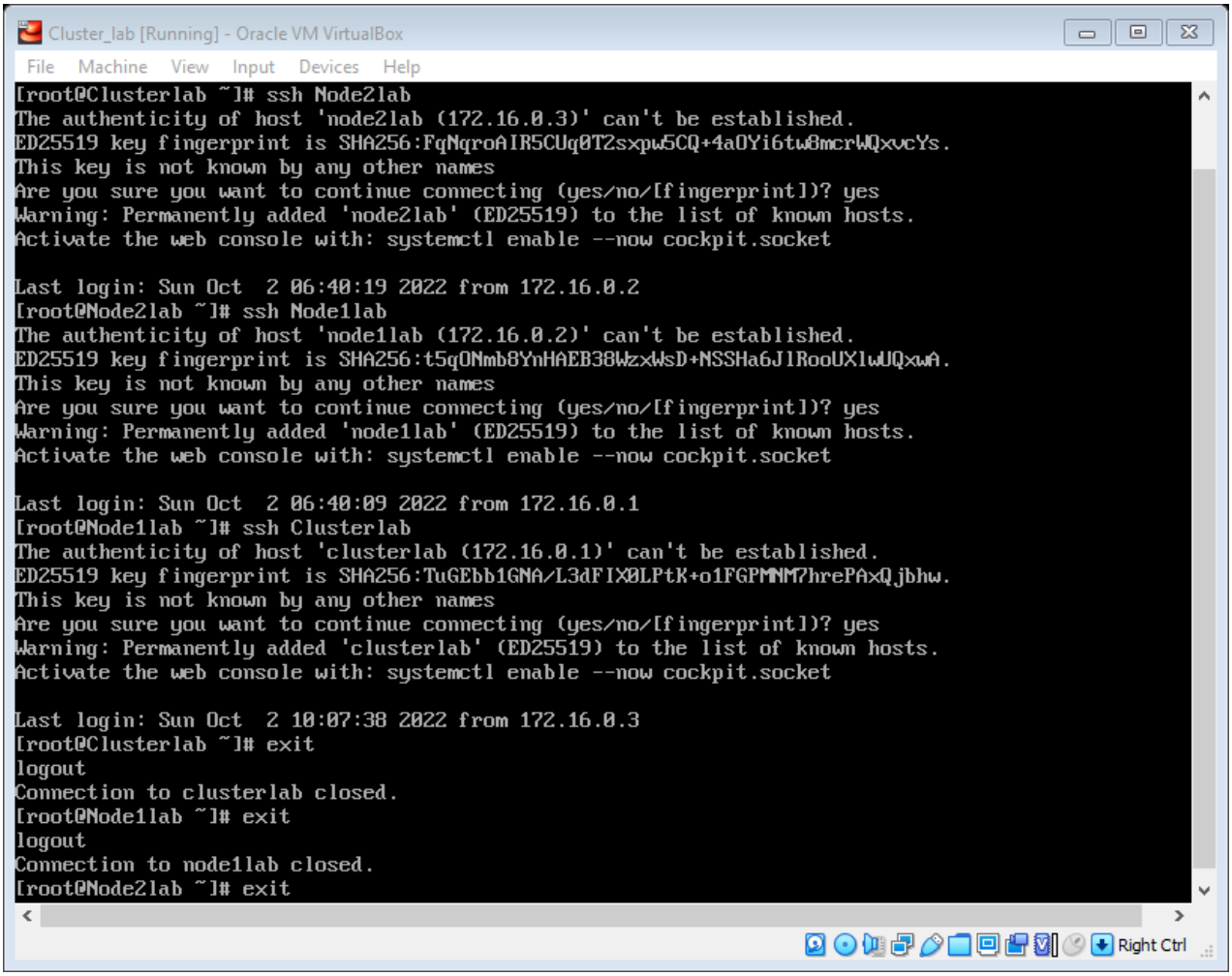

Рис. 2.20. **Перевірка безпарольного доступу в зворотному напрямі**

Якщо всі команди виконані успішно, то налаштування безпарольного доступу між ГС та ОВ під адміністративним обліковим записом *root* можна вважати закінченим.

З такими налаштуваннями буде зручніше встановлювати менеджер ресурсів. Від'єднуємо образ диска з доповненнями гостьової ОС від ГС (рис. 2.21).

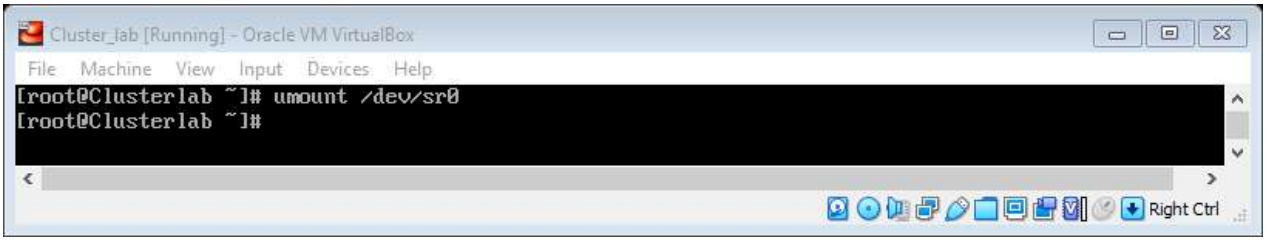

Рис. 2.21. **Від'єднання образу диска від ГС *Clusterlab***

Видаляємо диск образу доповнень гостьової ОС з ВМ *Cluster_lab* (рис. 2.22).

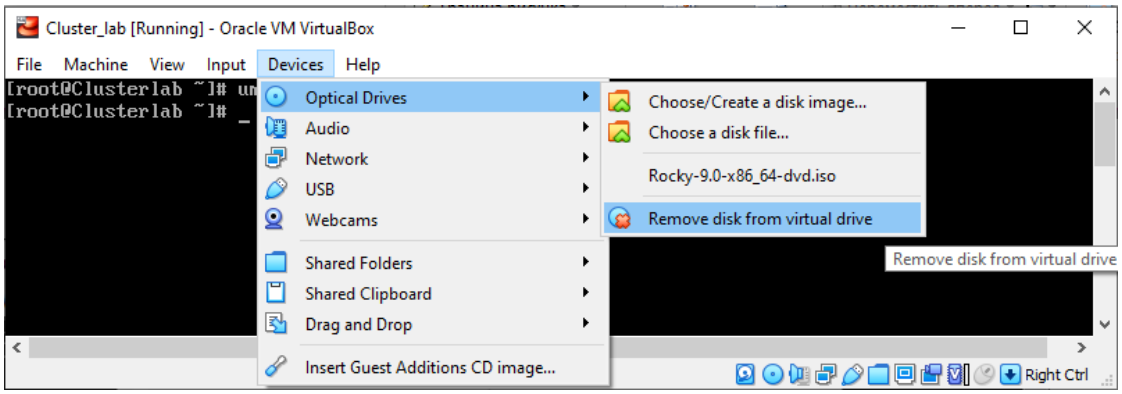

Рис. 2.22. **Вилучення диска образу доповнень гостьової ОС з ВМ**

Установлення та налаштування СУР *SLURM*

Перевірте, чи зареєстровані в системі з правами адміністратора *root*. Заходимо до загальної директорії /home/user/public та знаходимо файли з вихідними кодами пакетів менеджера ресурсів та автентифікації: *slurm-22.05.3.tar.bz2*, *munge-0.5.15.tar.xz*, *munge-0.5.15.tar.xz.asc*, *dun.gpg* (рис. 2.23).

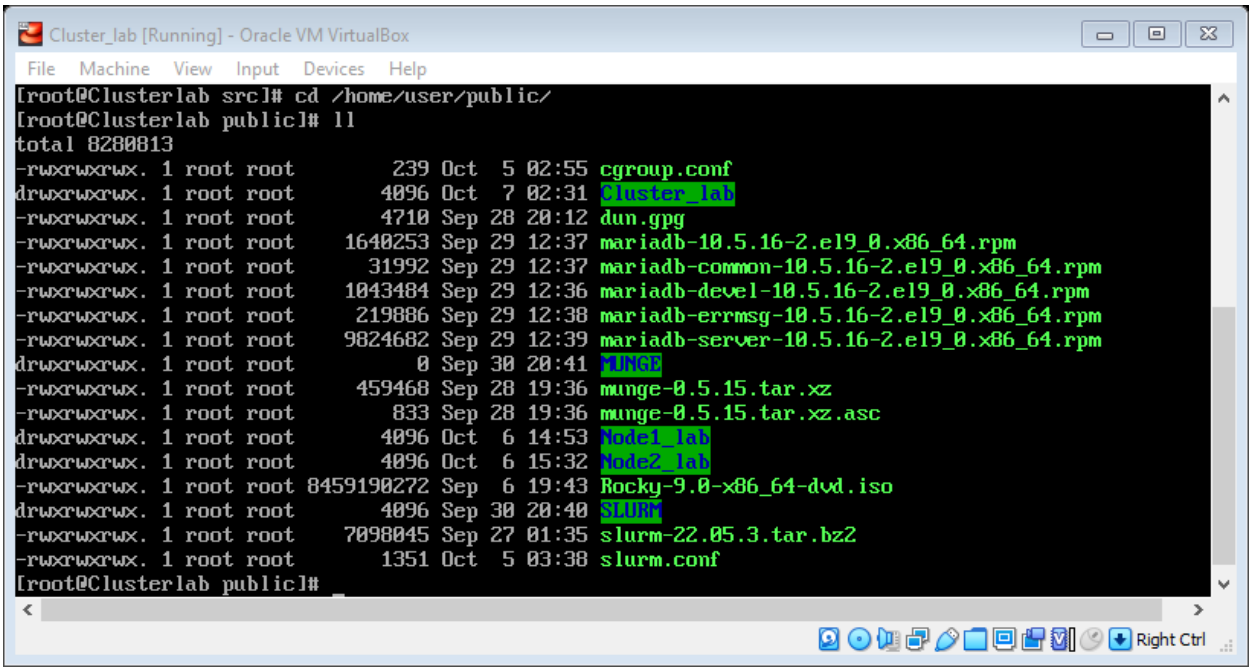

Рис. 2.23. **Перевірка наявності файлів з вихідним кодом та ключів**

Приєднуємо до ВМ *Cluster_lab* диск з образом ОС *Linux* (рис. 2.24).

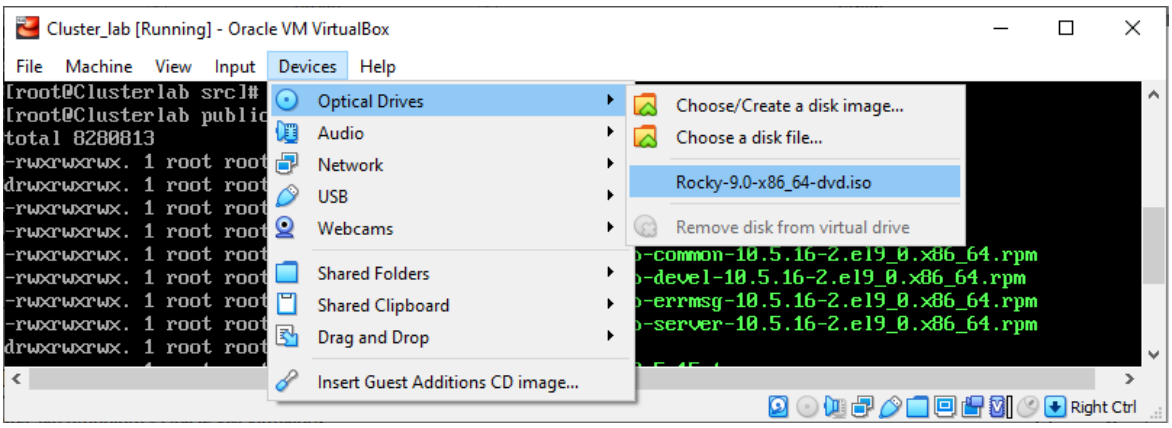

Рис. 2.24. **Приєднання диска з образом ОС *Linux* до ВМ *Cluster_lab***

Приєднуємо диск з образом ОС *Linux* до директорії *media* (рис. 2.25).

Рис. 2.25. **Приєднання диска з образом ОС *Linux* до вузла *Clusterlab***

Установлюємо пакети для побудови *RPM* з вихідного коду (рис. 2.26).

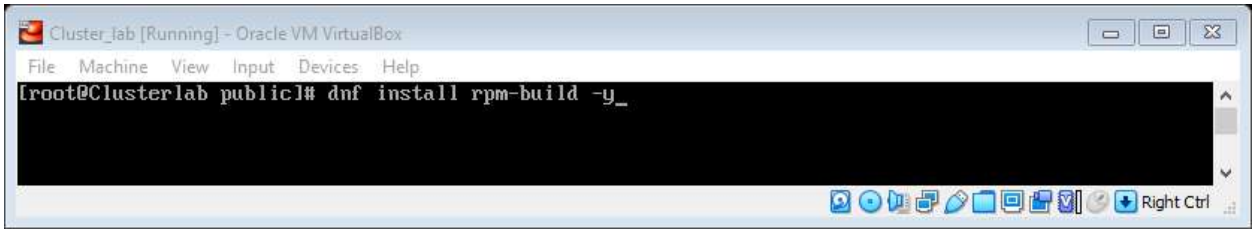

Рис. 2.26. **Установлення пакета *rpm-build***

Створюємо пакет *RPM* з вихідним кодом (рис. 2.27).

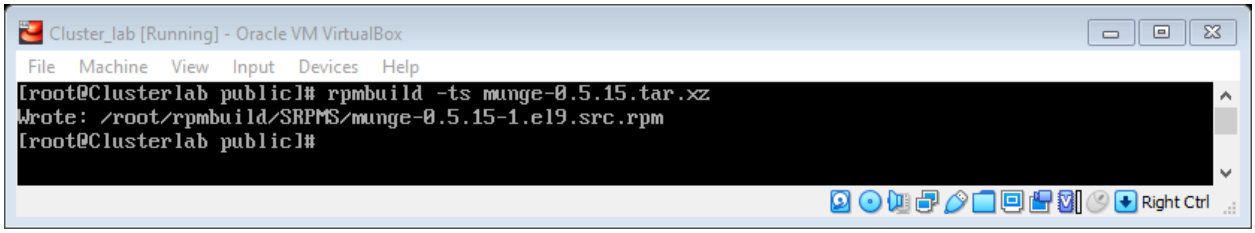

Рис. 2.27. **Створення пакета *src.rpm***

Установлюємо залежності для створення пакета *Munge* (рис. 2.28).

Рис. 2.28. **Установлення відсутніх залежностей**

Створюємо *RPM* пакети *Munge* з пакета з вихідним кодом (рис. 2.29).

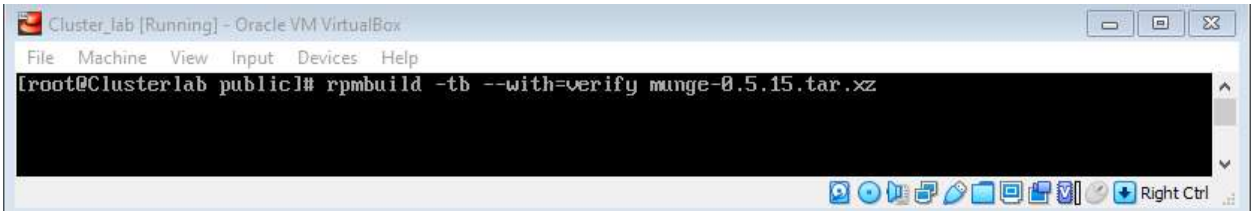

Рис. 2.29. **Створення *RPM* пакетів *Munge***

Перевіряємо, що пакети створено успішно (рис. 2.30).

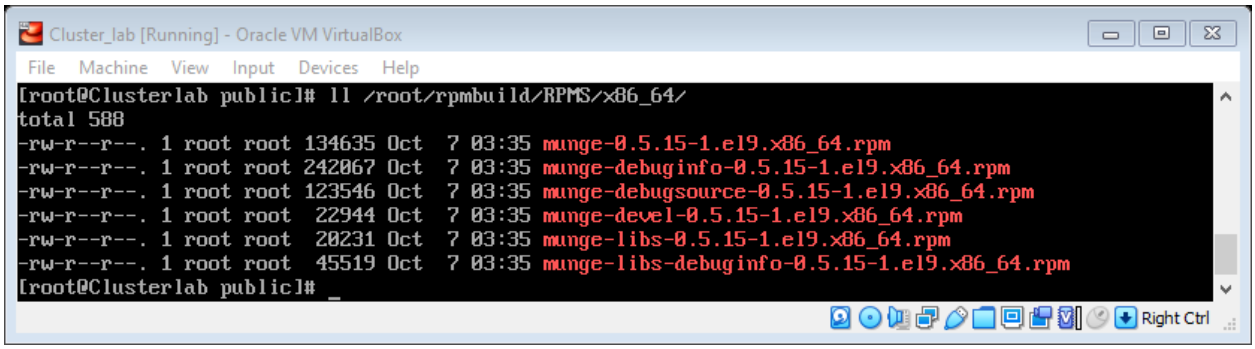

Рис. 2.30. ***RPM* пакети *Munge***

Пакет *Munge* містить демон *munged*, файл *mungekey* та файли клієнта (*munge, unmunge, remunge*). Пакет *munge-devel* містить файл *munge.h* для розроблення програм з *Munge*. Пакет *munge-libs* містить бібліотеку для запуску програм, що потребують *Munge*. Переходимо до директорії з *RPM* пакетами *Munge*, та встановлюємо потрібні для роботи СУР *SLURM* (рис. 2.31).

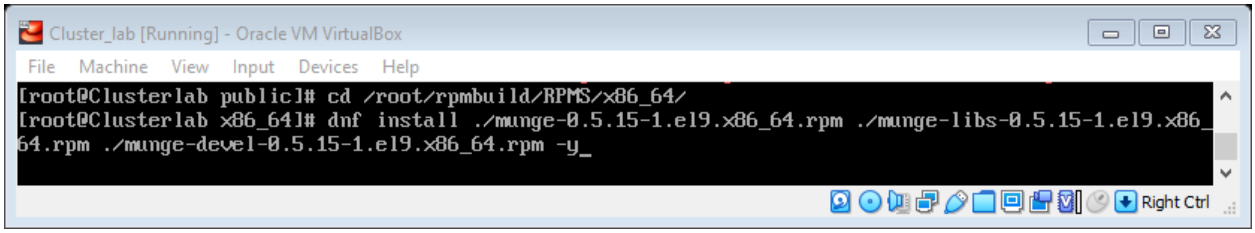

Рис. 2.31. **Установлення пакетів *Munge***

Додаємо пакети необхідні для складання *RPM* пакетів (рис. 2.32).

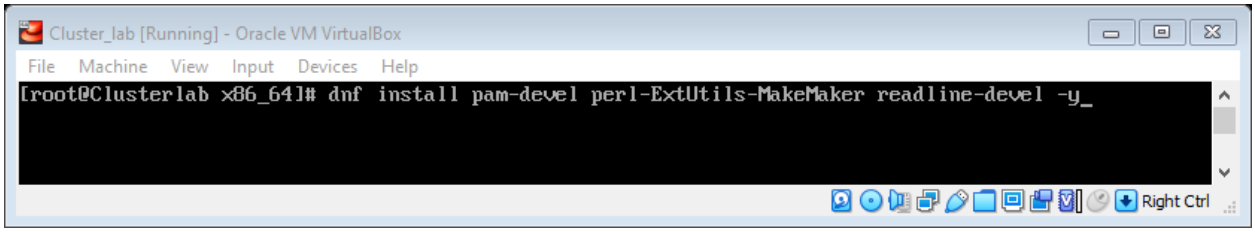

Рис. 2.32. **Установлення залежностей пакета СУР *SLURM***

Створюємо користувача *slurm* та задаємо пароль (рис. 2.33). Далі будемо встановлювати пакет бази даних (БД) *MariaDB*. Файли БД треба отримати у викладача та розмістити в раніш налаштованій загальній директорії *public*. Переходимо до директорії /home/user/public/ та встановлюємо БД *MariaDB* (рис. 2.34).

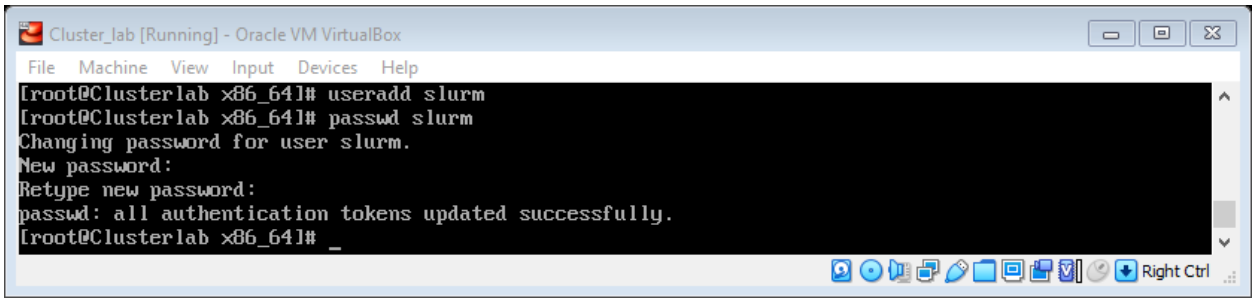

Рис. 2.33. **Створення користувача *slurm***

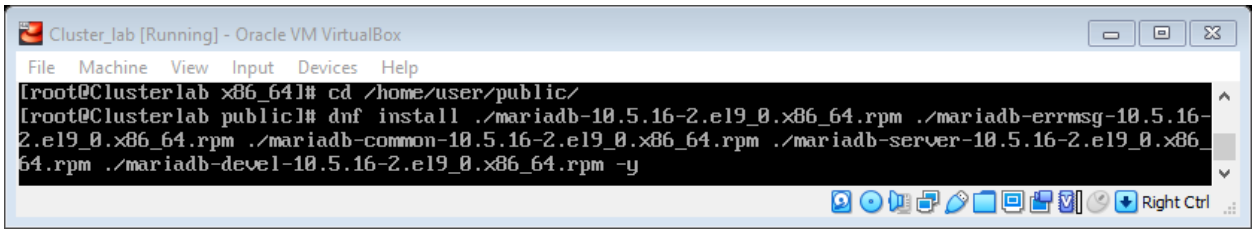

Рис. 2.34. **Установлення БД *MariaDB***

Створюємо *RPM* пакети *SLURM* з пакета з вихідним кодом (рис. 2.35).

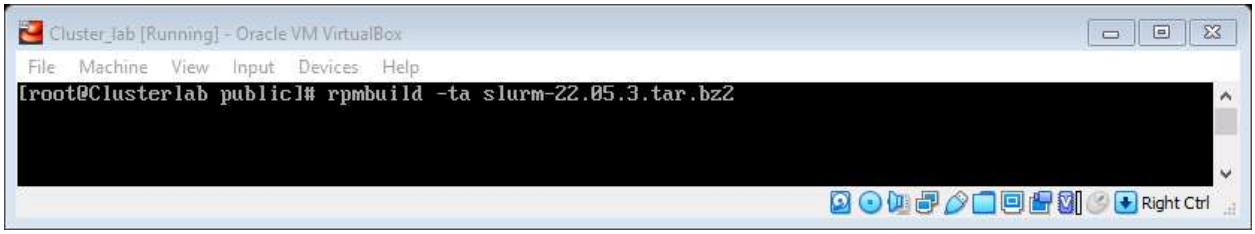

Рис. 2.35. **Створення *RPM* пакетів СУР *SLURM***

Якщо все виконано правильно, то отримуємо такі пакети (рис. 2.36).

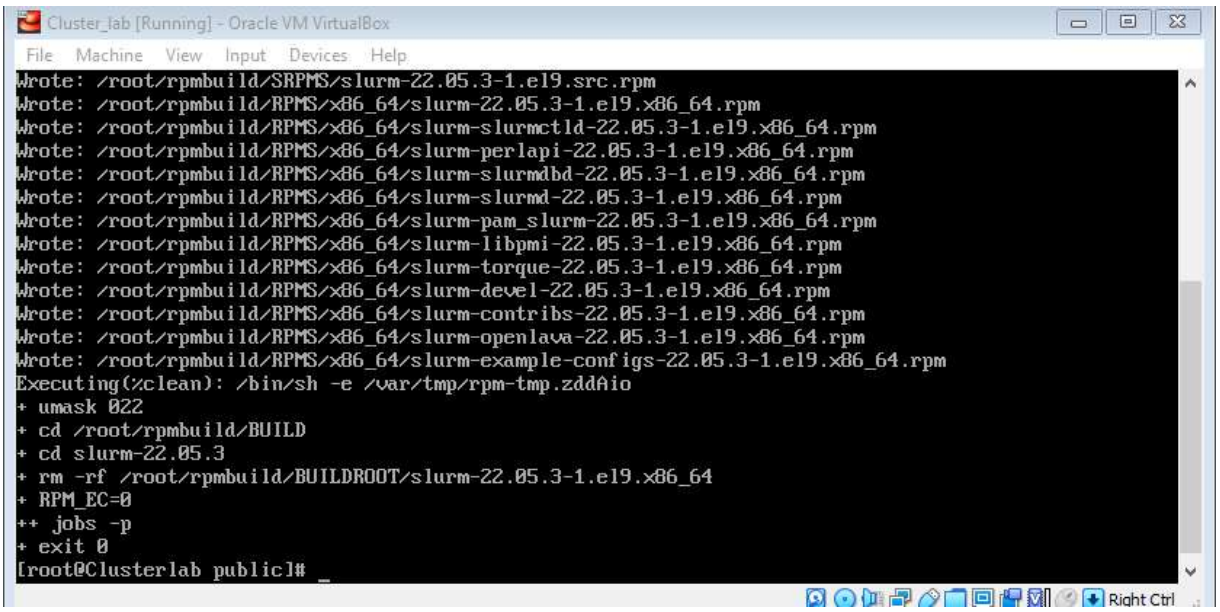

Рис. 2.36. **Зібрані *RPM* пакети СУР *SLURM***

На цьому етапі всі попередні налаштування кластера виконано і пакети для установки отримано. Далі треба перейти до створення кластера з СУР *SLURM*. Налаштовуємо службу синхронізації часу та відкриваємо файл конфігурації *Chrony* (рис. 2.37).

Рис. 2.37. **Відкриття файла конфігурації *Chrony***

Дозволяємо підключення до служби *chronyd* на ГС з локальної мережі (ЛМ) (рис. 2.38).

Рис. 2.38. **Дозвіл на підключення з ЛМ до служби *chronyd***

Перезавантажуємо та перевіряємо стан служби *chronyd* (рис. 2.39). Перевіряємо синхронізацію *Chrony* (рис. 2.40). Налаштовуємо службу автентифікації *Munge*. Перевіряємо та змінюємо атрибути директорій служби *Munge* (рис. 2.41).

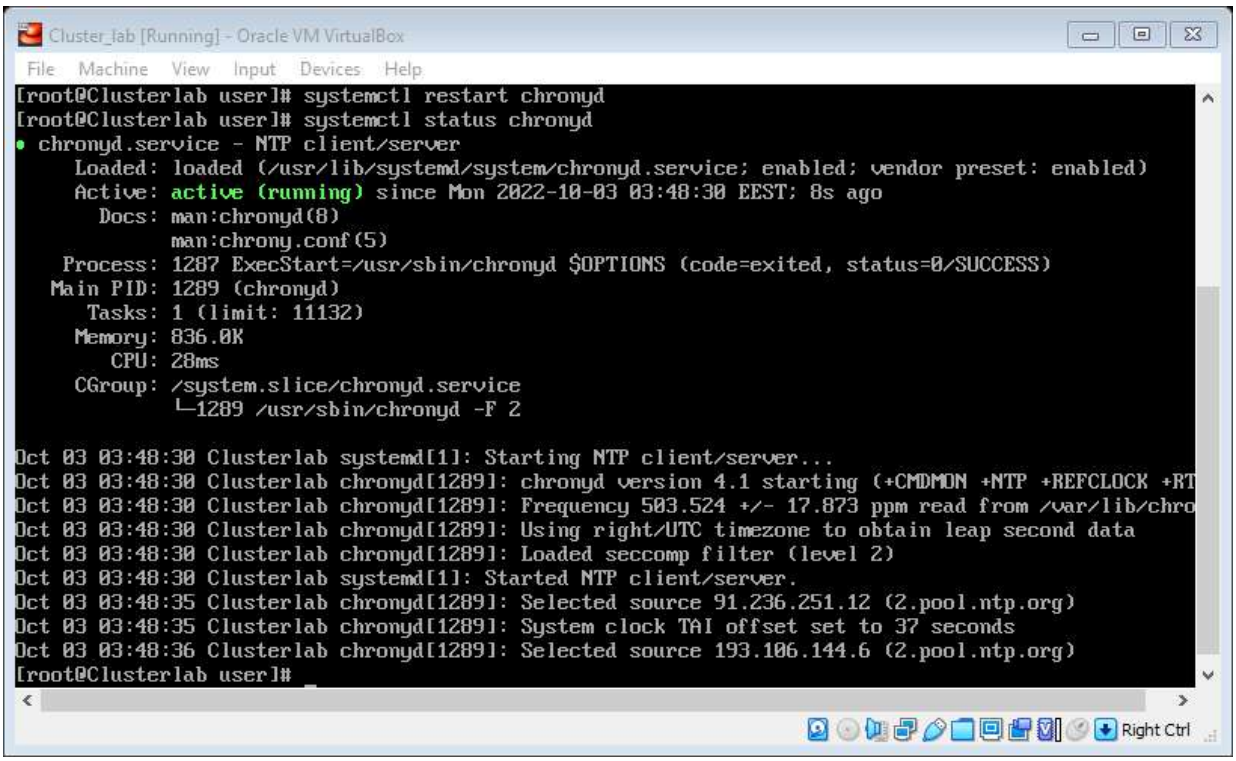

Рис. 2.39. **Стан служби *chronyd***

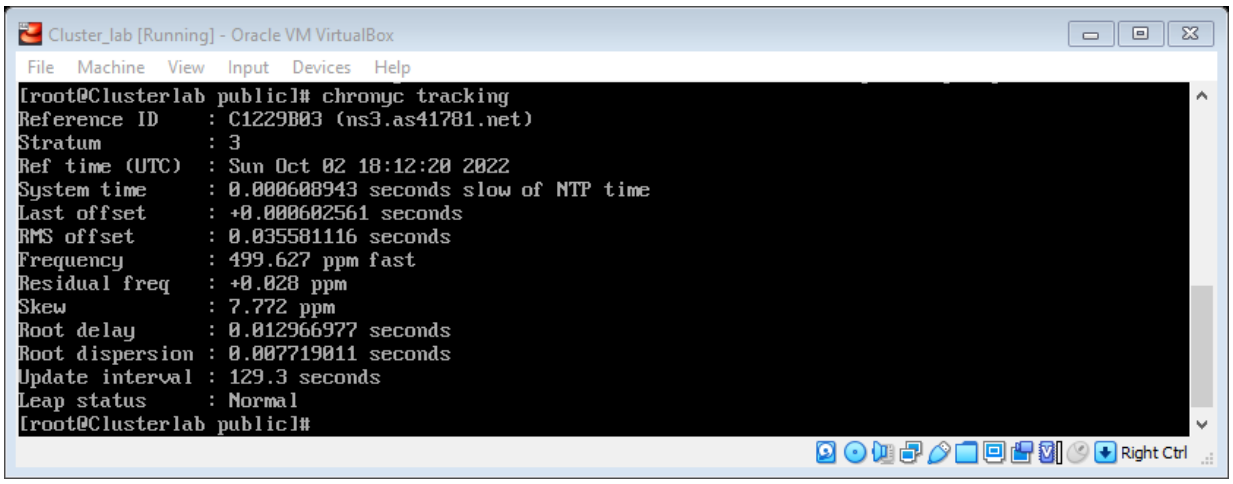

Рис. 2.40. **Перевірка синхронізації *chrony***

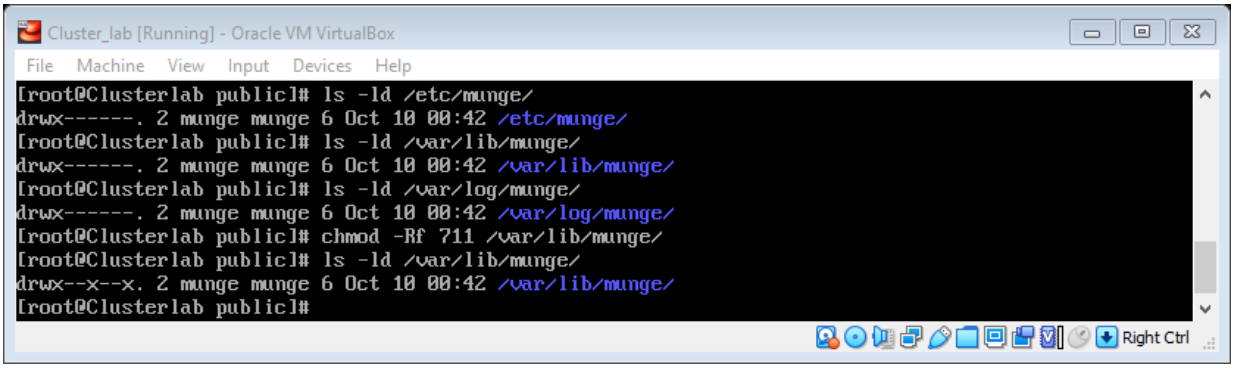

Рис. 2.41. **Налаштування атрибутів директорій служби *Munge***

Створюємо ключ доступу *munge.key*, перевіряємо та налаштовуємо його володаря і групу (рис. 2.42). Запускаємо службу *munge.*

Рис. 2.42. **Створення та налаштування ключа *munge.key***

Перевіряємо, щоб запуск пройшов без збоїв, та дозволяємо завантажувати службу під час старту ОС *Linux* (рис. 2.43).

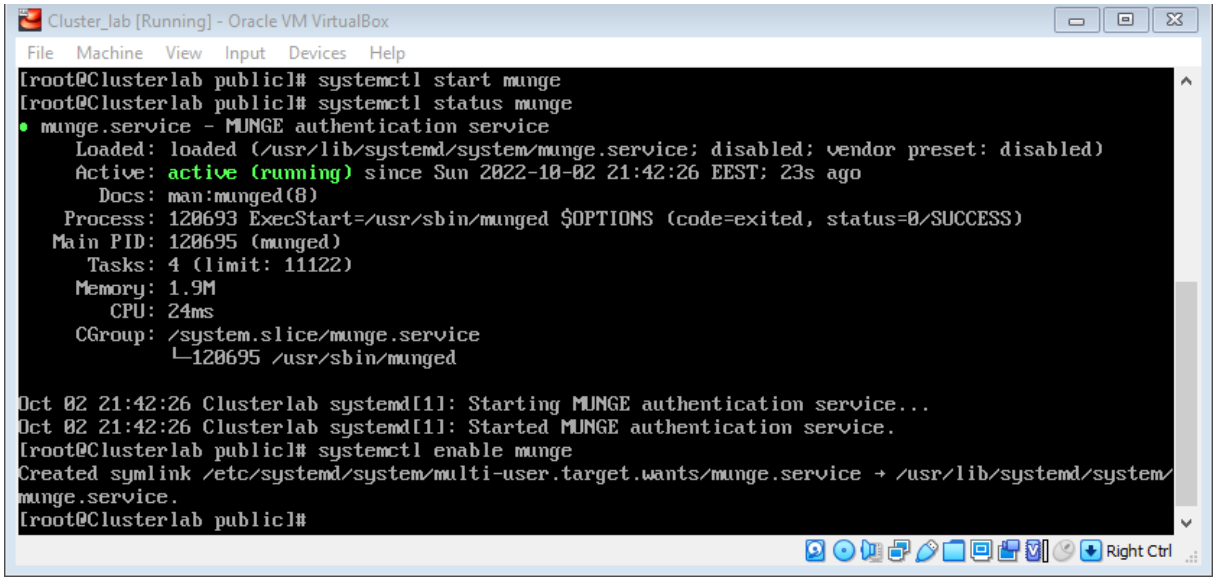

Рис. 2.43. **Запуск служби *munge* та дозвіл на завантаження під час старту ОС**

Переходимо до директорії з *RPM* пакетами СУР *SLURM* та встановлюємо їх для роботи на ГС (рис. 2.44).

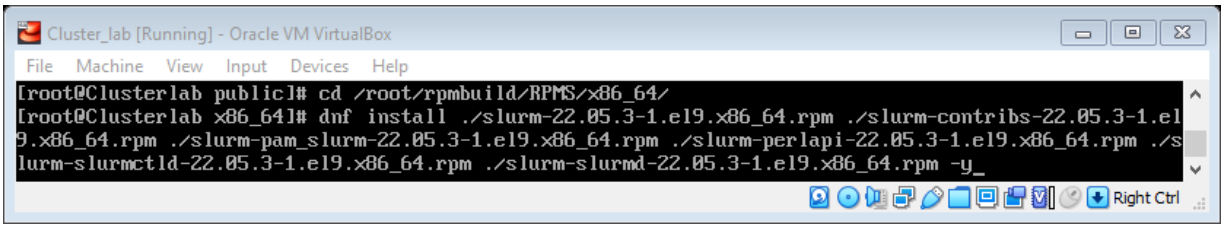

Рис. 2.44. **Установлення пакетів СУР *SLURM* на ГС**

Копіюємо файл конфігурації та налаштовуємо атрибути для запуску (рис. 2.45).

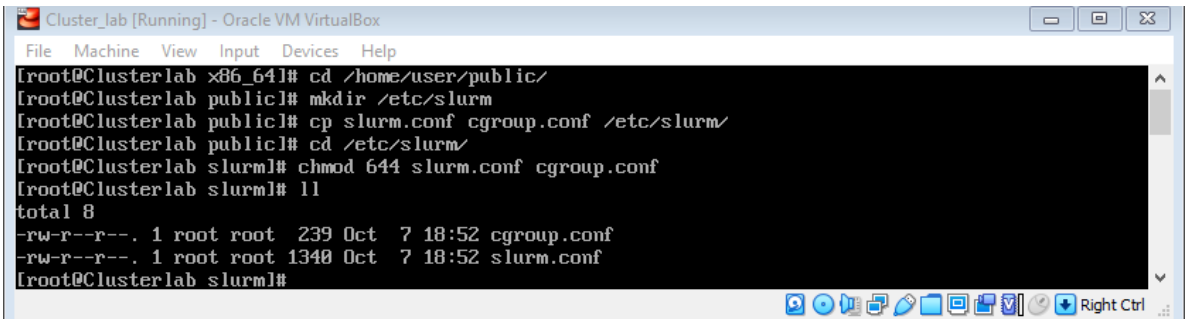

Рис. 2.45. **Налаштування конфігураційних файлів СУР *SLURM***

Налаштовуємо ОВ *Node1lab*. Копіюємо *RPM* пакети та файли конфігурації *Munge* та *SLURM* з ГС *Clusterlab* на ОВ *Node1lab* (рис. 2.46).

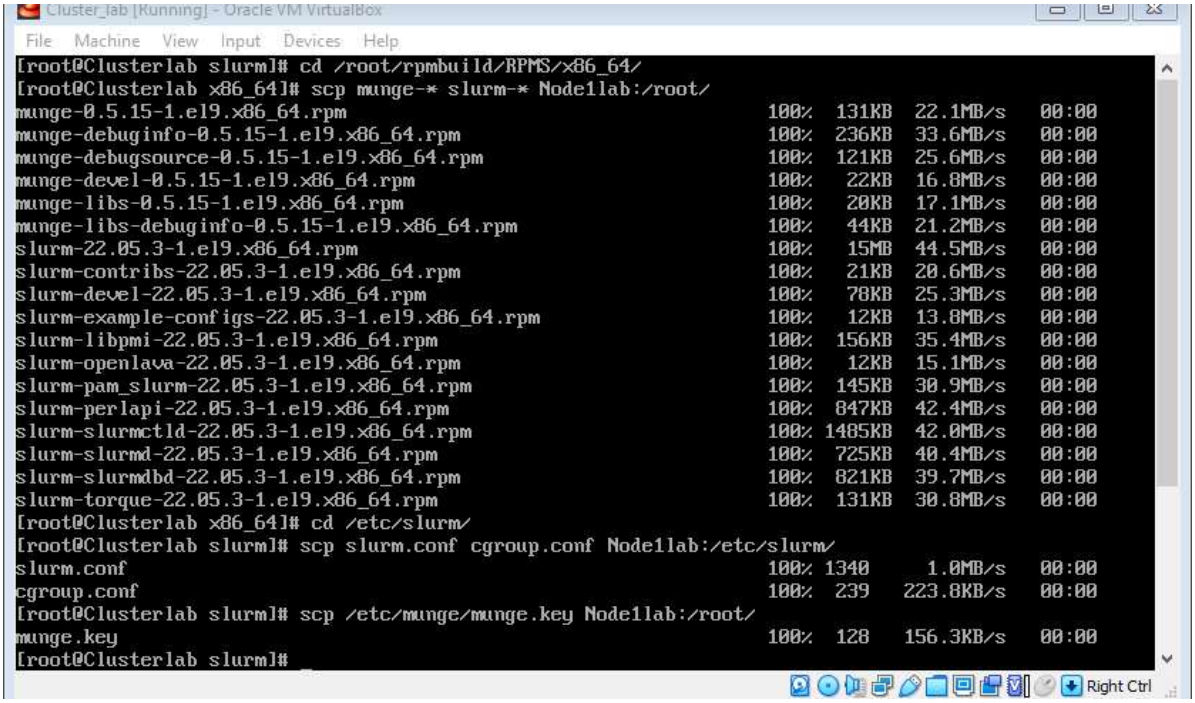

Рис. 2.46. **Копіюємо пакети і файли конфігурації *Munge* та СУР *SLURM***

Потім заходимо на ОВ *Node1lab* під користувачем *user* та отримуємо права адміністратора (рис. 2.47).

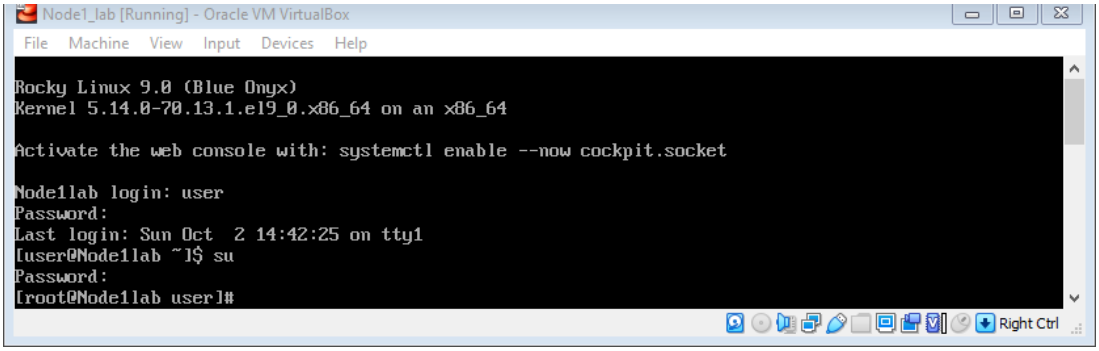

Рис. 2.47. **Вхід на ОВ *Node1lab***

Приєднуємо до ВМ *Node1_lab* диск з образом ОС *Linux* (рис. 2.48) та приєднуємо диск з образом ОС *Linux* до директорії *media* (рис. 2.49).

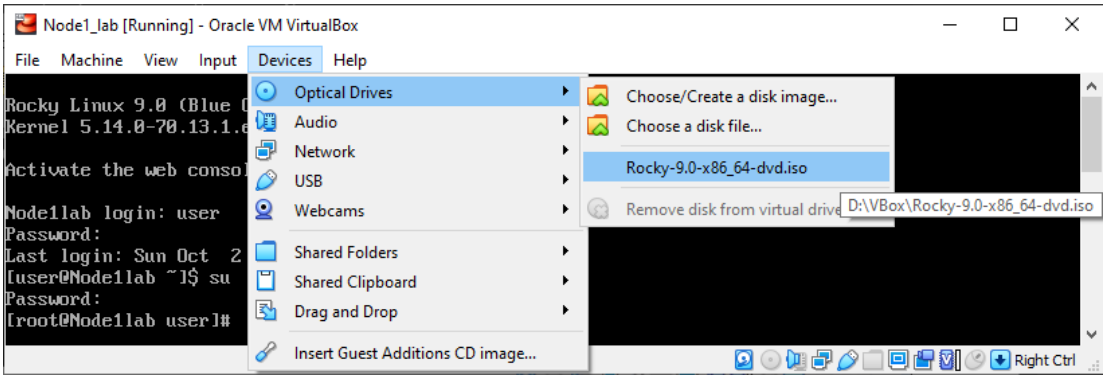

Рис. 2.48. **Приєднання диску з образом ОС *Linux* до ВМ *Node1_lab***

Рис. 2.49. **Приєднання диску з образом ОС *Linux* до ОВ *Node1_lab***

Налаштовуємо службу синхронізації часу: відкриваємо файл конфігурації *Chrony* (рис. 2.50) та прописуємо адресу ГС *Clusterlab* (рис. 2.51).

Рис. 2.50. **Відкриття файла конфігурації *Chrony***

Рис. 2.51. **Прописування вузла *Clusterlab* для синхронізації часу**

З неї ОВ *Node1lab* буде отримувати значення часу. Дозволяємо *Chrony* доступ до ЛМ (рис. 2.52). Перезавантажуємо та перевіряємо стан служби *chronyd* (рис. 2.53) та перевіряємо синхронізацію *Chrony* (рис. 2.54).

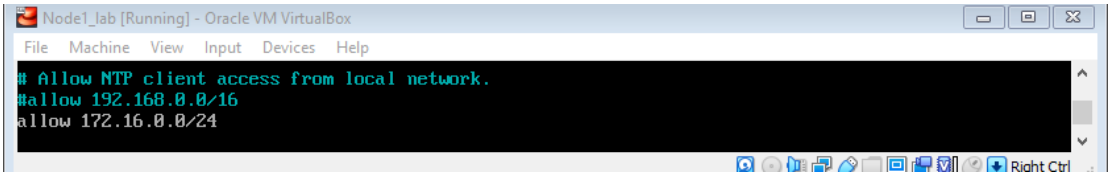

Рис. 2.52. **Дозвіл на підключення з ЛМ до служби *chronyd***

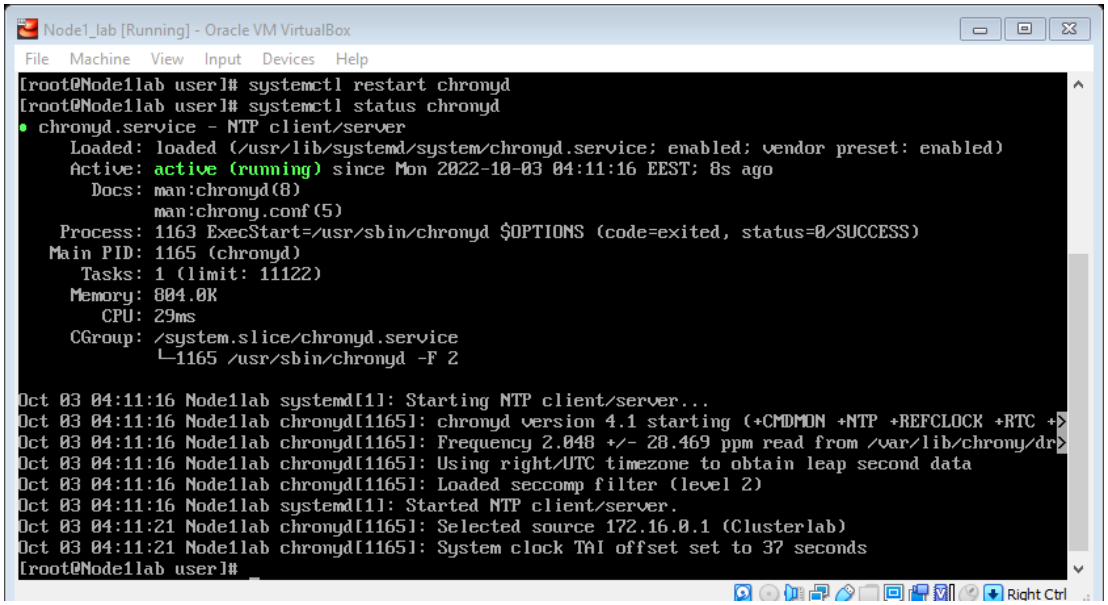

Рис. 2.53. **Перезавантаження служби *chronyd***

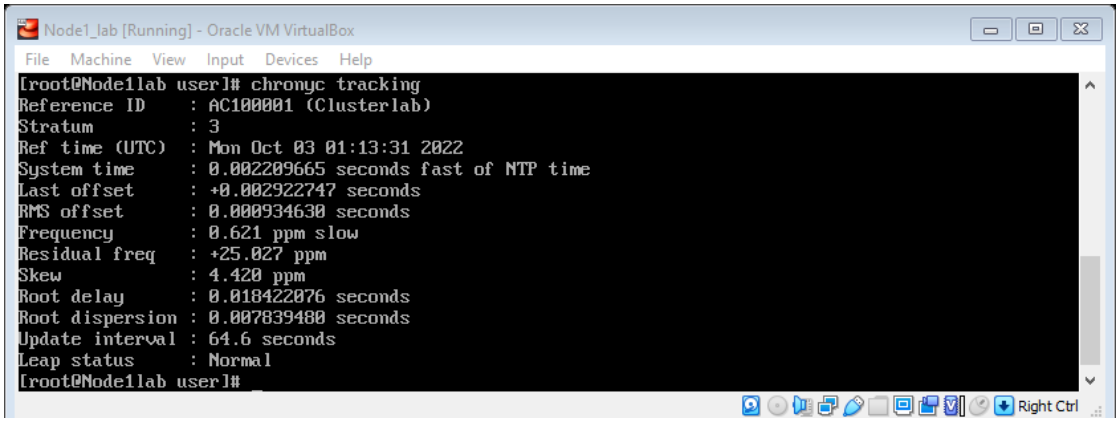

Рис. 2.54. **Перевірка синхронізації *Chrony***

Налаштовуємо службу автентифікації *Munge*. Переходимо до директорії / *root* та встановлюємо необхідні *RPM* пакети (рис. 2.55).

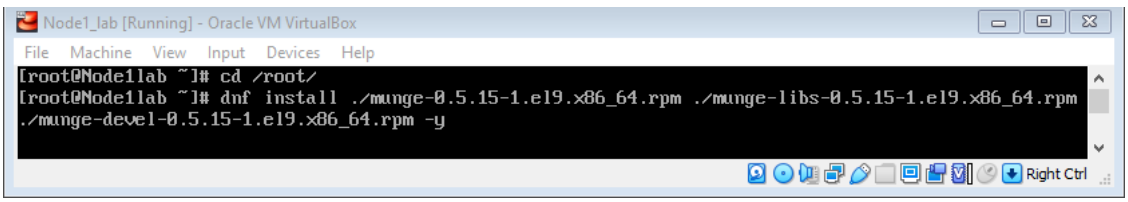

Рис. 2.55. **Установлення *RPM* пакетів *Munge***

Копіюємо ключ автентифікації у конфігураційну директорію. Налаштовуємо потрібні атрибути (рис. 2.56), запускаємо службу *Munge* та перевіряємо, що запуск пройшов без збоїв, та дозволяємо завантажувати службу під час старту ОС *Linux* (рис. 2.57).

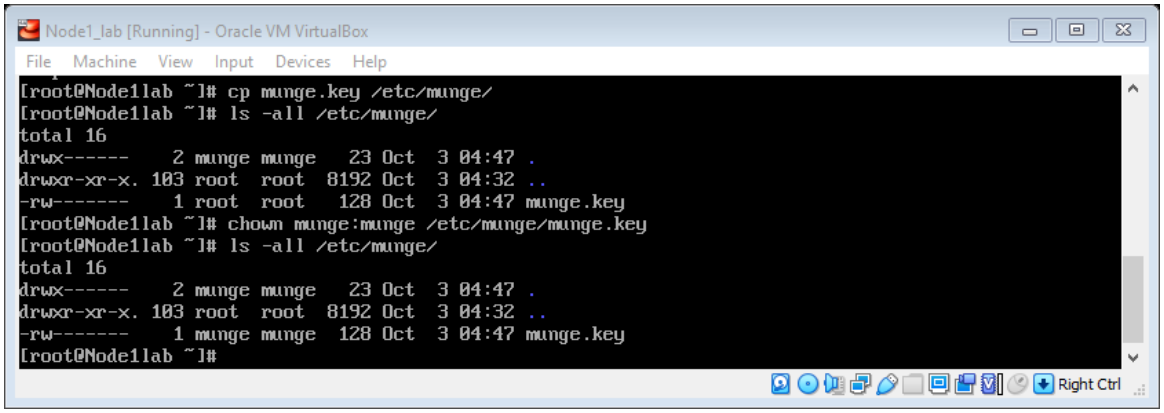

Рис. 2.56. **Установлення ключа автентифікації пакета *Munge***

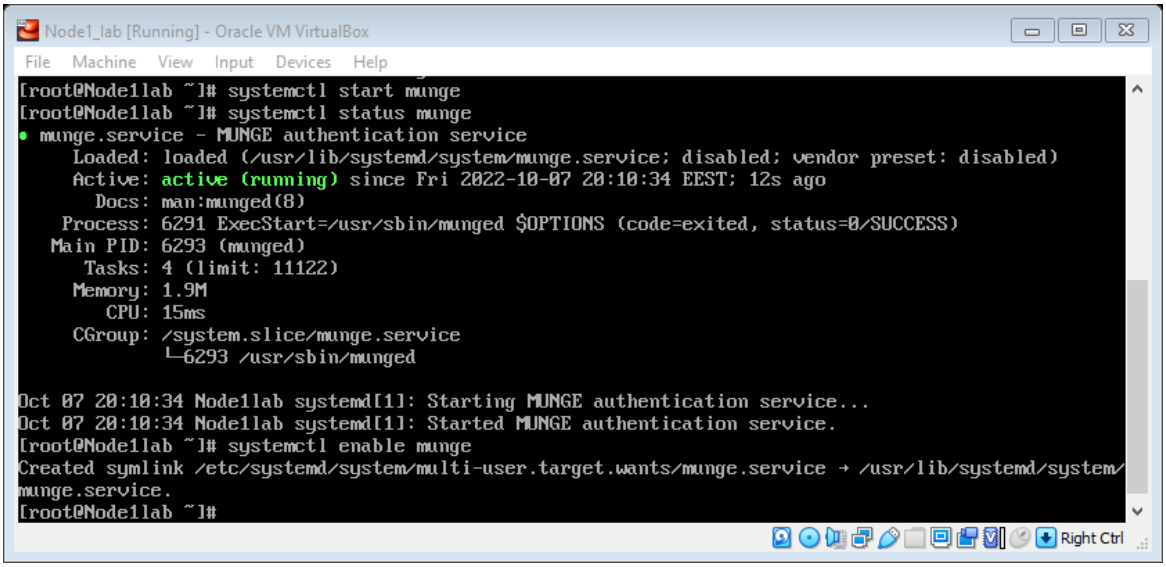

Рис. 2.57. **Завантаження служби *Munge***

Перевіряємо та змінюємо атрибути директорій служби *Munge* (рис. 2.58).

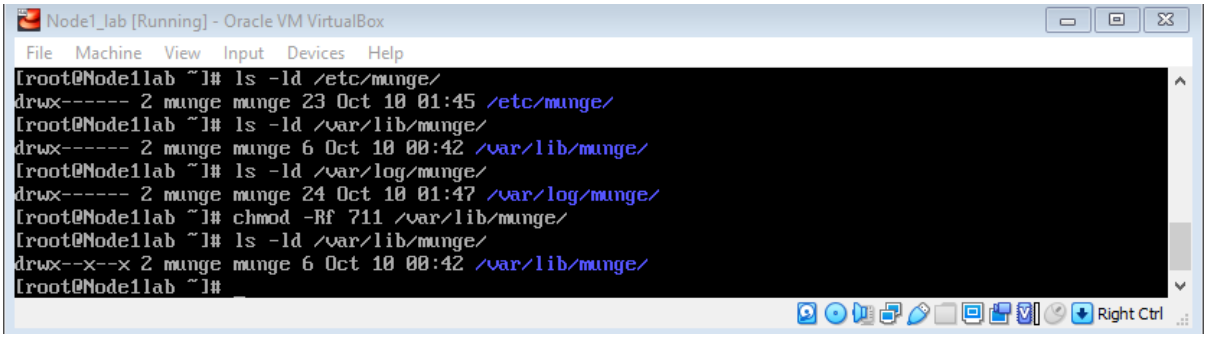

Рис. 2.58. **Налаштування атрибутів директорій служби *Munge***

Установлюємо пакети СУР *SLURM* (рис. 2.59): створюємо користувача *slurm*, задаємо пароль для користувача *slurm* (як на ГС) та створюємо директорію */var/spool/slurmd* (рис. 2.60).

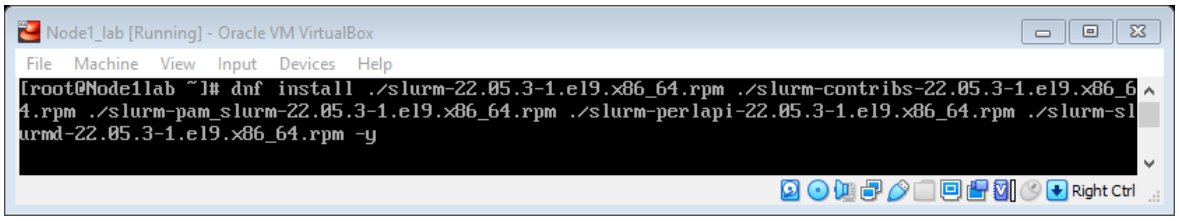

Рис. 2.59. **Установлюємо пакети СУР *SLURM***

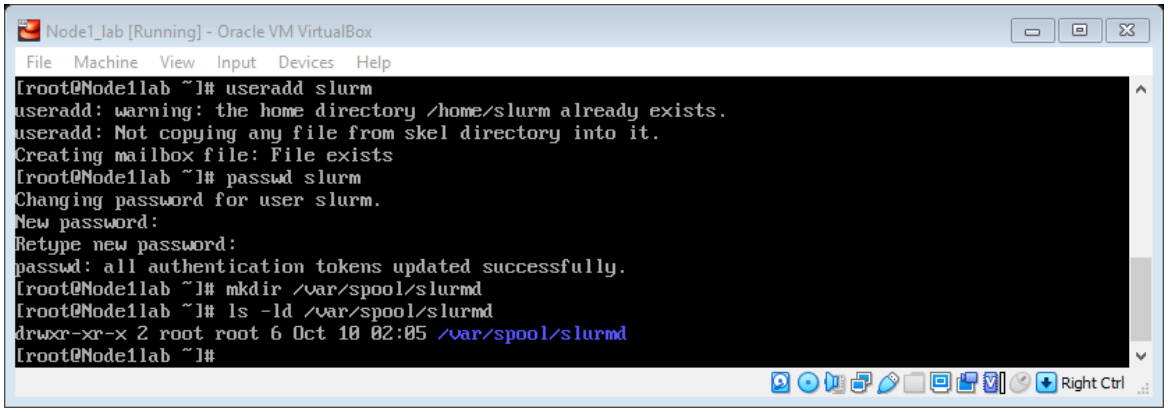

Рис. 2.60. **Створення користувача *slurm* та робочої директорії *slurmd***

Запускаємо службу *slurmd*, перевіряємо її статус та дозволяємо завантаження служби під час старту ОС (рис. 2.61).

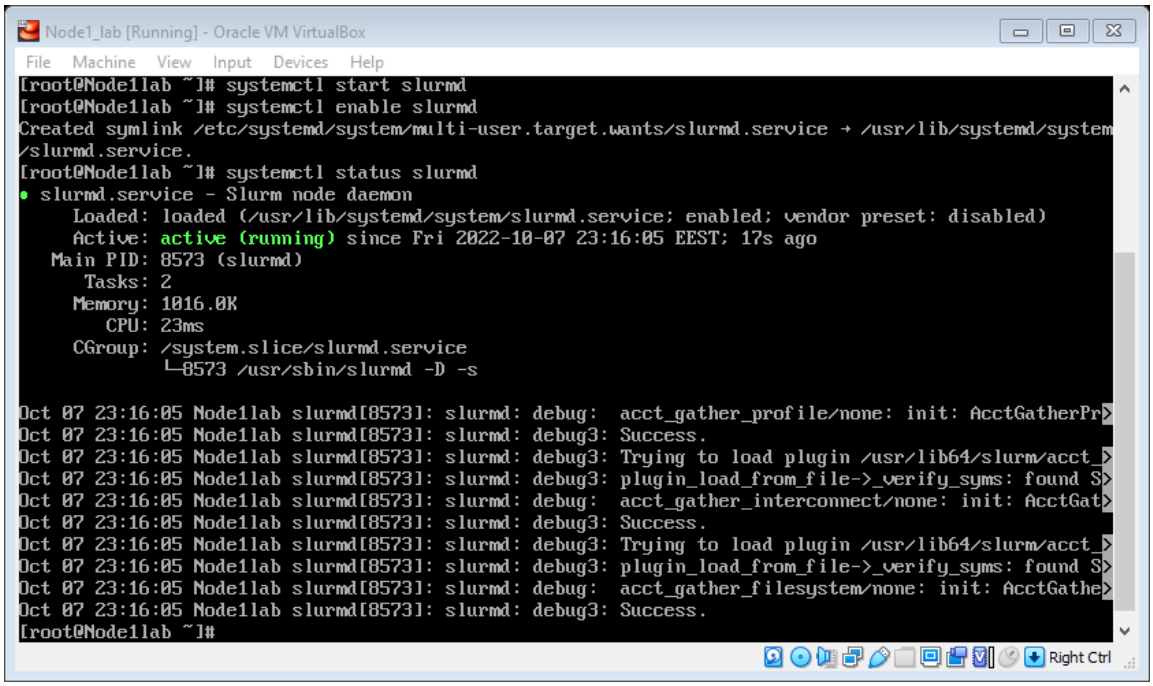

Рис. 2.61. **Завантаження служби *slurmd***

Установлюємо компілятор *C++* та бібліотеку *MPICH* на ОВ (рис. 2.62).

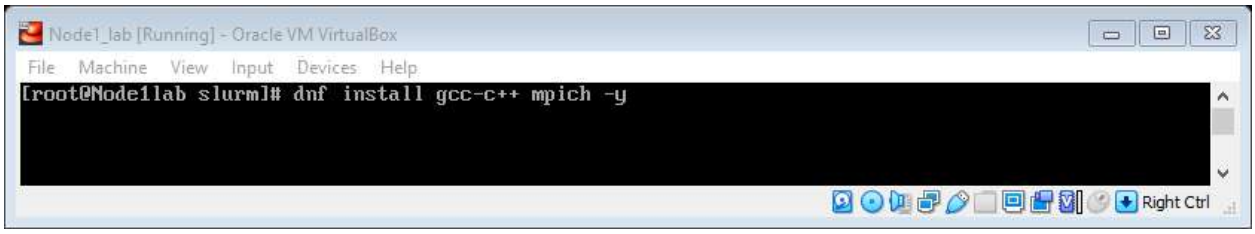

Рис. 2.62. **Установлення компілятора C++ та бібліотеки MPI**

Від'єднуємо дистрибутив ОС *Linux* від директорії *media* (рис. 2.63).

Рис. 2.63. **Від'єднання дистрибутиву ОС *Linux***

Вилучаємо образ диска з ВМ *Node1_lab* (рис. 2.64). На цьому налаштування ОВ *Node1lab* закінчено.

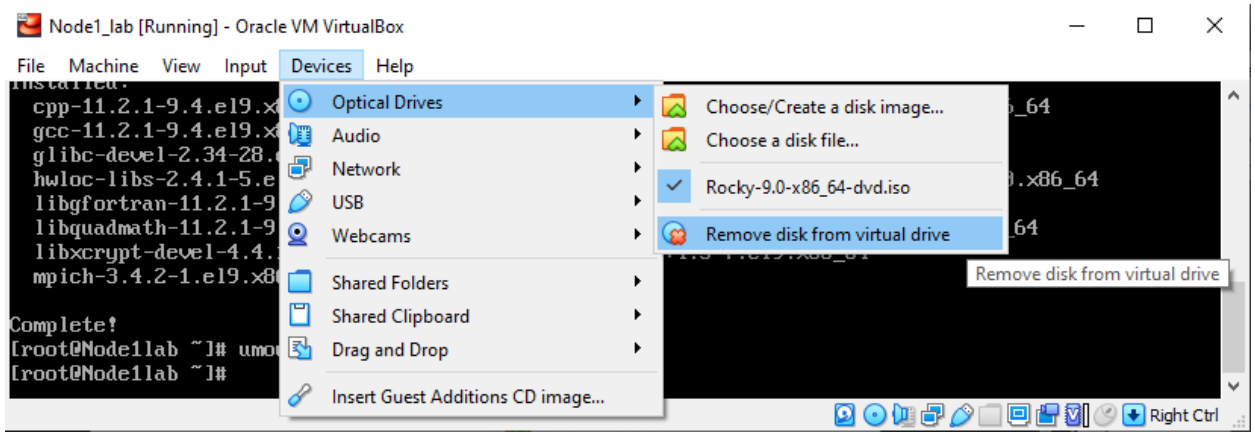

Рис. 2.64. **Вилучення образу диска з *ВМ Node1_lab***

Налаштовуємо ОВ *Node2lab*. Копіюємо пакети *RPM* та файли конфігурації *Munge* та СУР *SLURM* з ГС *Clusterlab* на ОВ *Node2lab* (рис. 2.65).

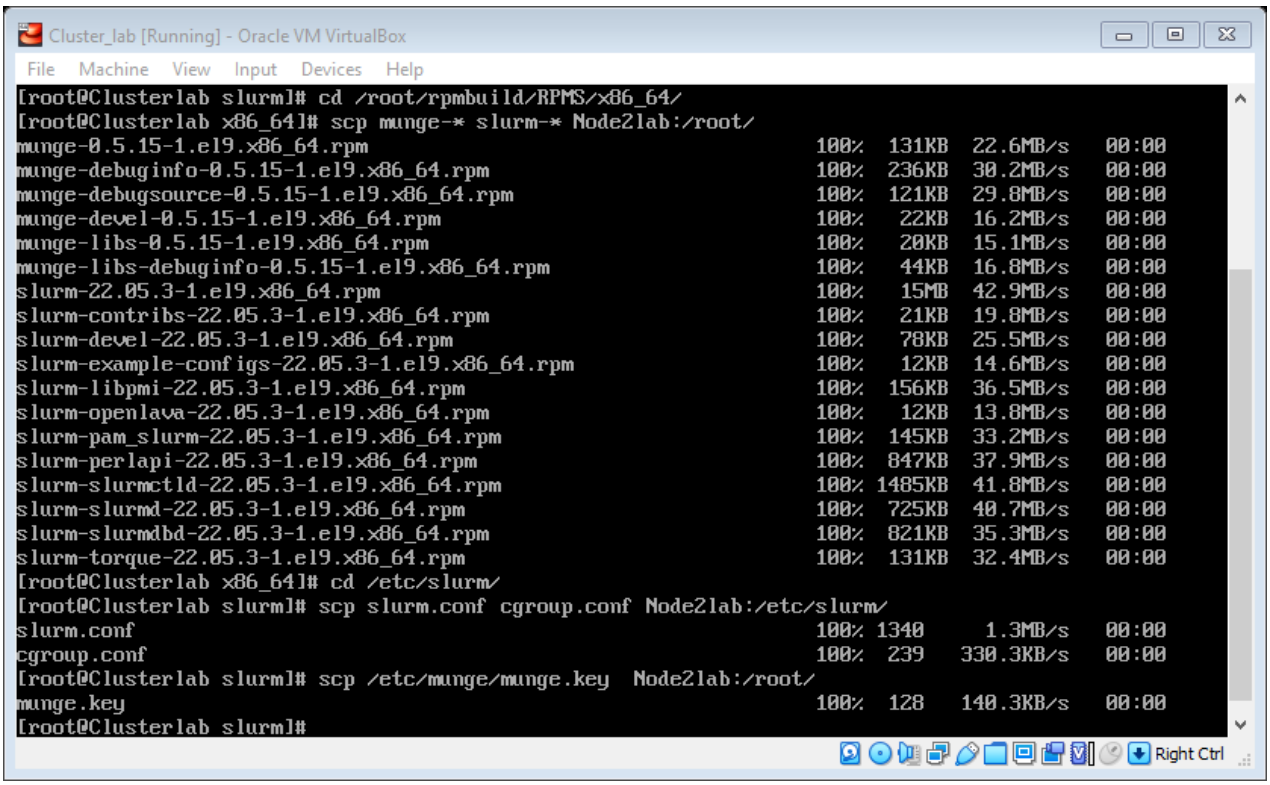

Рис. 2.65. **Копіювання пакетів та файлів конфігурації *Munge* та *SLURM***

Переходимо на ОВ *Node2lab* та повторюємо ті самі дії для налаштування для ОВ *Node2lab*, що виконувалися для ОВ *Node1lab*. Коли всі дії для ОВ *Node2lab* буде виконано, то повертаємося до ГС *Clusterlab*.

На ГС *Clusterlab* створюємо директорію */var/spool/slurmctld* та */var/spool/slurmd* (рис. 2.66).

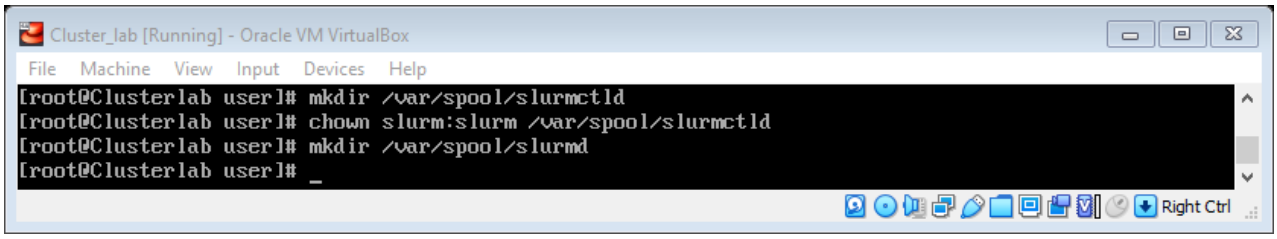

Рис. 2.66. **Створення робочих директорій *slurmd* та *slurmctld***

Директорію */var/spool/slurmctld* налаштовуємо на володіння користувачем *slurm*. На ГС запускаємо службу *slurmld*, перевіряємо її статус та дозволяємо службі завантажуватися під час старту ОС (рис. 2.67).

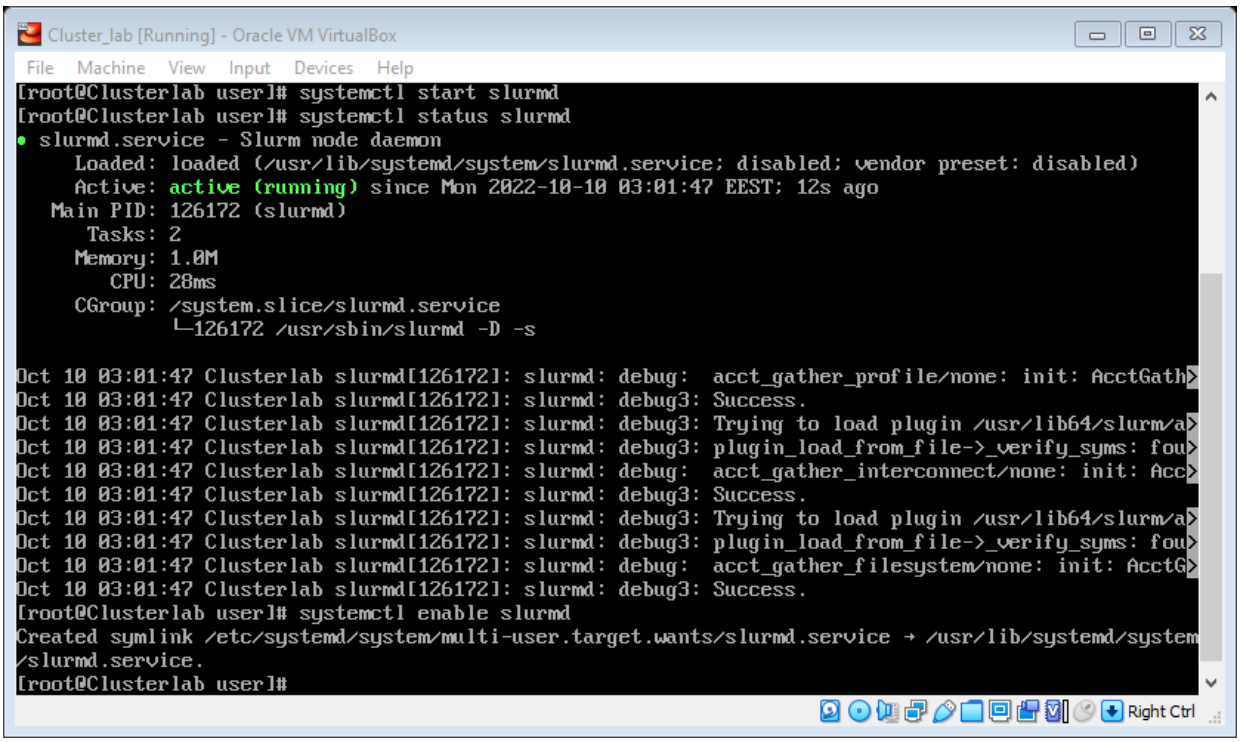

Рис. 2.67. **Завантаження служби *slurmd***

На ГС запускаємо службу *slurmctld*, перевіряємо її статус та дозволяємо службі завантажуватися під час старту ОС (рис. 2.68). На цьому налаштування кластера з СУР *SLURM* закінчується.

Від'єднуємо дистрибутив ОС *Linux* від директорії *media* (див. рис. 2.7). Вилучаємо образ диска з ВМ *Cluster_lab* (див. рис. 2.8). Переходимо до перевірки працездатності створеного кластера.

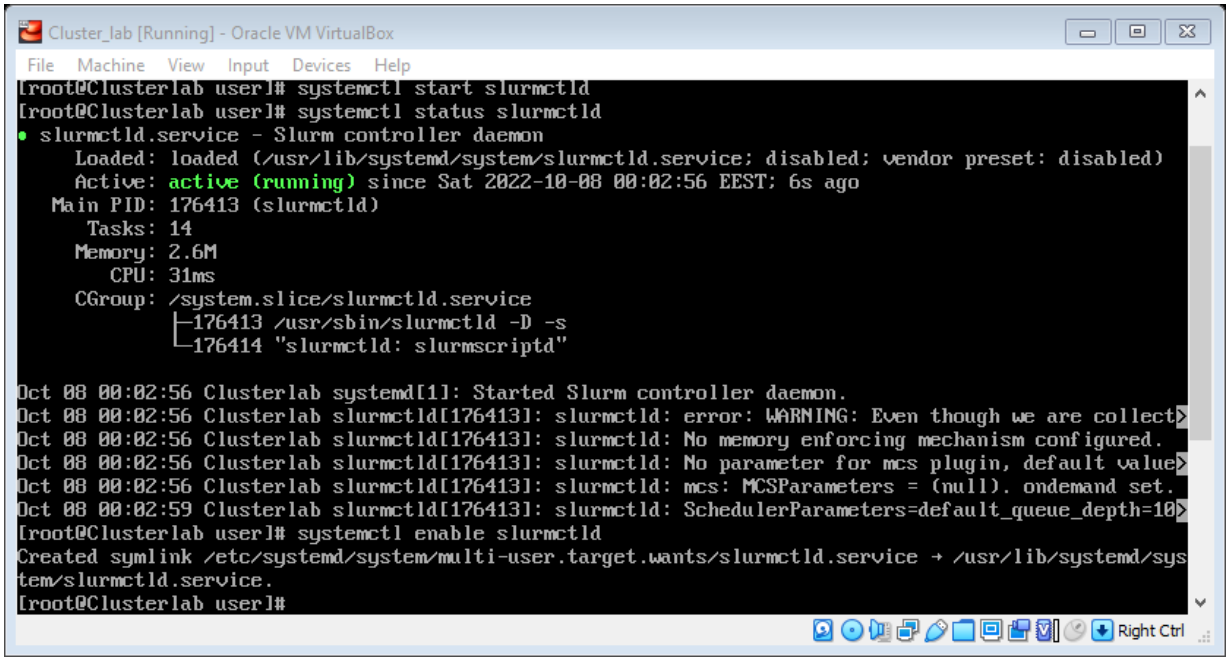

Рис. 2.68. **Завантаження служби *slurmctld***

Перевіряємо стан кластера командою *sinfo* (рис. 2.69).

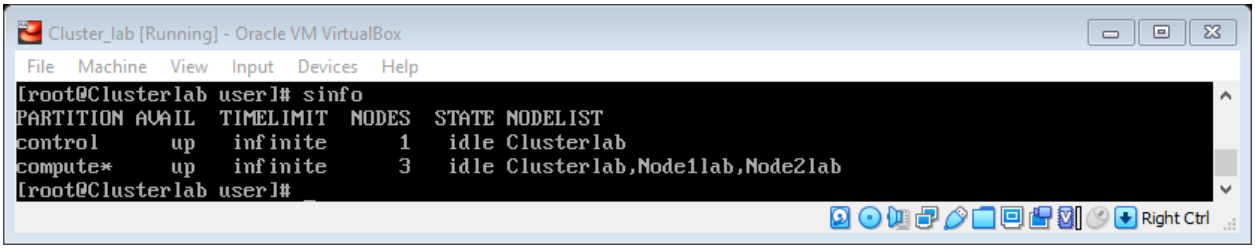

Рис. 2.69. **Перевірка стану ГС та ОВ кластера**

Отримуємо інформацію про розділи кластера, командою *scontrol show partition* (рис. 2.70).

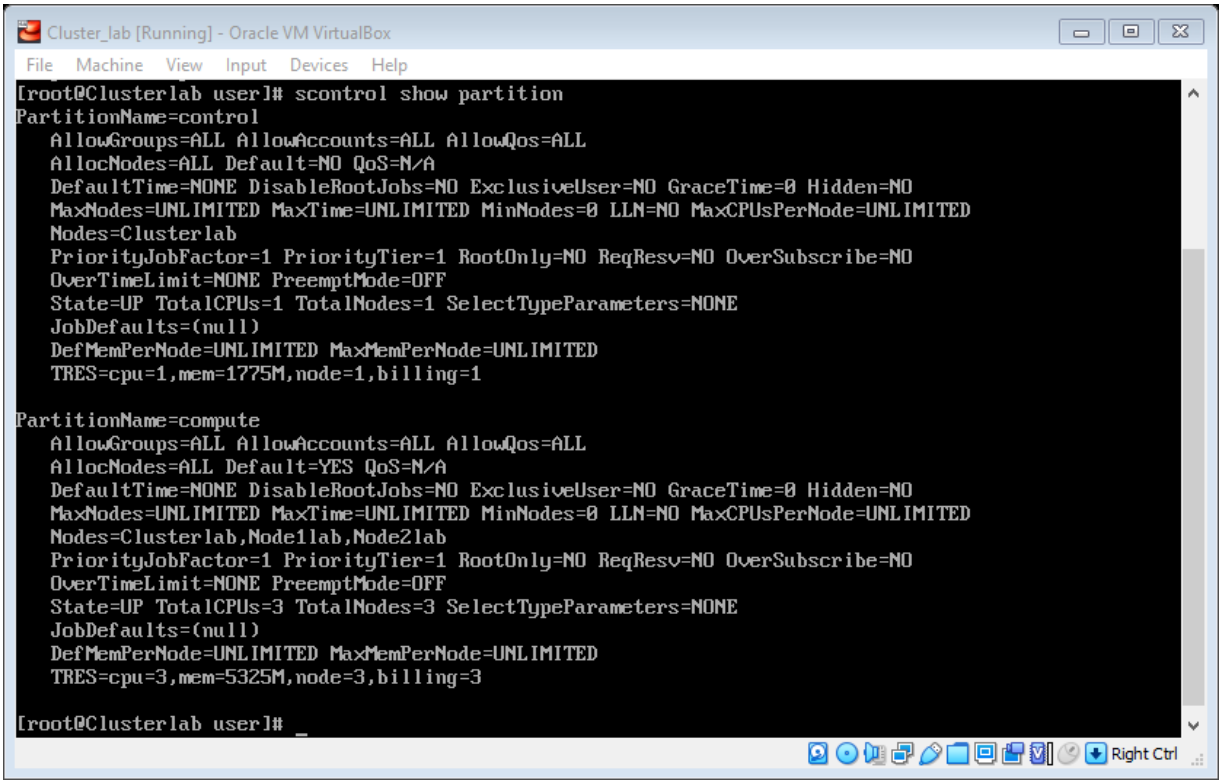

Рис. 2.70. **Інформація про розділи кластера**

Отже головний сервер та обчислювальні вузли кластера налаштовано. Запускаємо тестове завдання (рис. 2.71).

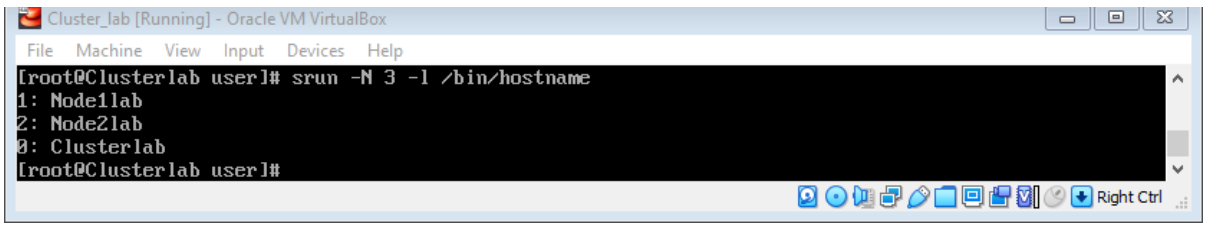

Рис. 2.71. **Запуск тестового завдання**

Таким чином, усі обчислювальні вузли прийняли та обробили отримане завдання.

На цьому перевірку працездатності кластера з системою управління ресурсами *SLURM* завершено.

Контрольні запитання до лабораторної роботи 2

1. Наведіть основні функції служби *Munge* та принципи реалізації.
2. Наведіть основні функції системи управління ресурсами та особливості налаштування *SLURM*.
3. Порівняйте за призначенням головний сервер та обчислювальні вузли кластера.
4. Наведіть основні функції служби *slurmctld*.
5. Наведіть основні функції служби *slurmd*.
6. Наведіть основні функції сервісу *Chrony*.
7. Поясніть результати виведення команди sinfo.
8. Поясніть значення параметрів під час виведення інформації про розділи кластера.
9. Для чого на всіх обчислювальних вузлах кластера встановлюється пакет *mpich*?
10. Для чого створюють окремого користувача *slurm*?

Лабораторна робота 3
Дослідження побудови обчислювального кластера та принципів роботи з ним

Мета лабораторної роботи:

Дослідження апаратної та програмної складової обчислювального кластера.

Набуття практичних навичок із використання основних команд щодо управління обчислювальним кластером.

Удосконалення практичних навичок з розроблення програмного забезпечення для платформи *Linux*.

Набуття практичних навичок запуску власних додатків на обчислювальному кластері.

Загальні відомості про принципи побудови та роботи обчислювального кластера

Програмна складова обчислювального кластера *Torque*

Об'єднання ресурсів у систему управління розподіленими обчисленнями (*Portable Batch System, PBS*) зазвичай зменшує технічне управління ресурсами, пропонуючи одноманітний підхід до користувачів. Після правильного налаштування система абстрагується від багатьох деталей, пов'язаних з роботою та управлінням завданнями, що дозволяє підвищити рівень використання ресурсів. Наприклад, користувачам потрібно вказати тільки мінімальні обмеження на завдання і не потрібно знати окремі імена машин кожного вузла, на якому запущене це завдання.

Torque – це менеджер ресурсів, є однією з версій PBS, відповідає за відстеження доступної кількості ресурсів на вузлах кластера та запуск завдань. Він керує завантаженням обчислювальних комплексів, що складаються з певної кількості обчислювальних вузлів, що працюють під управлінням операційної системи сімейства *Linux*.

Менеджер ресурсів *Torque* має вбудований планувальник завдань, реалізований у вигляді демона *pbs_sched*. Він забезпечує режим пакетного запуску завдань. Менеджер ресурсів забезпечує низькорівневі функціональні можливості, а саме запуск, утримування (тимчасове припинення), скасування і контроль виконання завдання. Без цих можливостей менеджера ресурсів, планувальник не зможе самостійно контролювати завдання.

Кластер складається з головного сервера та багатьох обчислювальних вузлів. На головному сервері працює демон *pbs_server*. А на обчислювальних вузлах запускається демон *pbs_mom* (може працювати й на сервері), який взаємодіє з демоном PBS сервера *pbs_server*. Клієнтські команди для подання і управління завданнями можуть бути встановлені на будь-якому комп'ютері (у тому числі на вузлах, де відсутні демони *pbs_server* або *pbs_ sched*).

На головному вузлі також працює демон планувальника *pbs_ sched*. Демон планувальника взаємодіє з демоном *pbs_server* для прийняття рішень, пов'язаних з локальними політиками використання ресурсів, відповідності вузлів кластера вимогам завдань в черзі і виокремленню доступних для черги вузлів для виконання завдання. Простий FIFO планувальник завдань, а також код для побудови більш складних планувальників наданий у дистрибутиві вихідного коду.

У більшості випадків користувачі *Torque* використовують планувальники з більш широким функціоналом, а саме *Maui* або *Moab*. Користувачі відправляють завдання на головний сервер, де розгорнуто демон *pbs_server* за допомогою команди *qsub*. Коли демон *pbs_server* отримує нове завдання, то він інформує про це планувальника. Коли планувальник знаходить вільні вузли, що відповідають вимогам для виконання завдання, то він посилає інструкції для виконання завдання на вузли зі списку доступних черзі вузлів кластера. Далі *pbs_server* посилає нове завдання на перший вузол у списку вузлів і дає вузлу команди відповідно до запуску завдання. Структурна схема управління розподіленими обчисленнями наведена на (рис. 3.1).

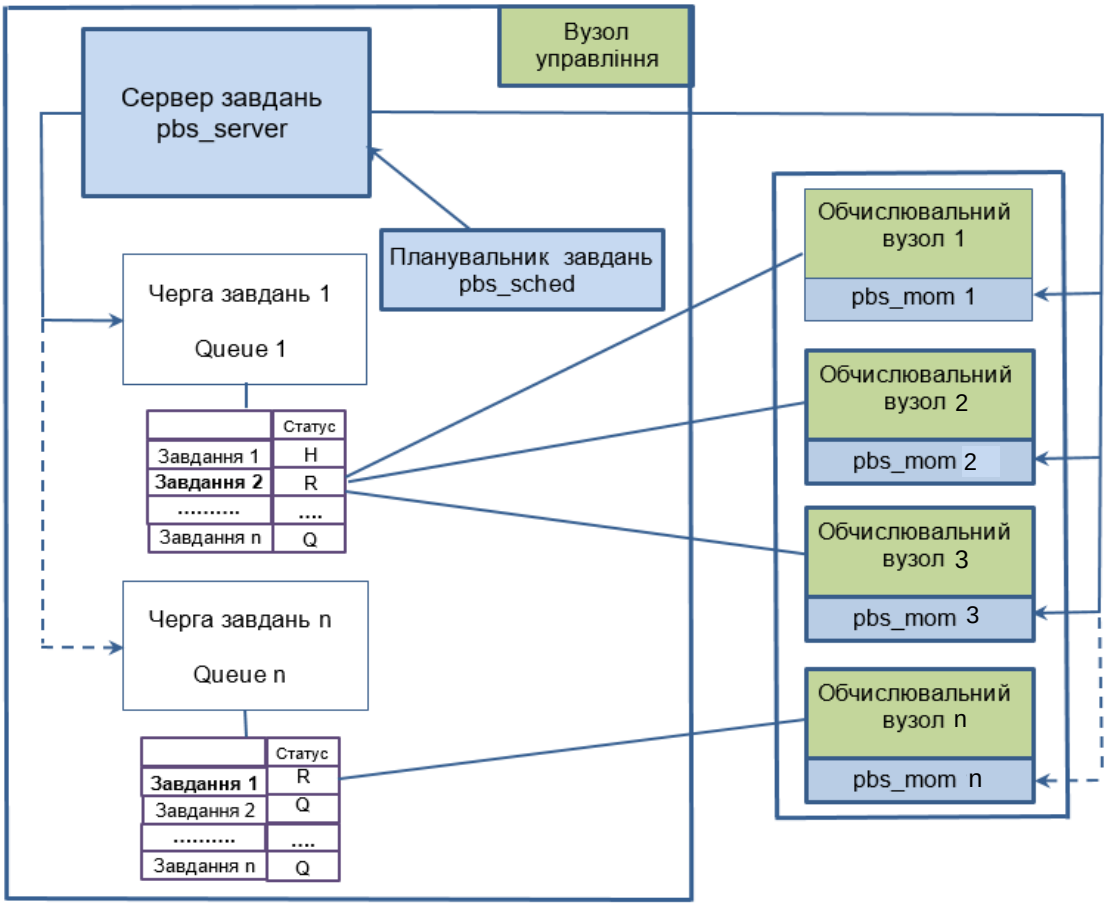

Рис. 3.1. **Структурна схема управління розподіленими обчисленнями**

Взаємодія компонентів Torque

Щоб запустити програму на кластері, необхідно використовувати систему управління завданнями PBS *Torque*. Зазвичай для цього створюється спеціальний *pbs_script* та ставиться в чергу за допомогою команди *qsub*. *Pbs_script* містить інформацію про необхідні ресурси (число вузлів кластера, необхідна кількість оперативної пам'яті, необхідна кількість часу та ін.). В усьому іншому цей файл є звичайним *bash*-файлом. У ньому можливо налаштувати потрібні змінні оточення, що дозволяє запустити необхідні програми.

Pbs_script буде виконано лише на одному вузлі з виділеного списку. В його обов'язки входить запуск програми на решті вузлів. Наприклад, за допомогою команди *mpiexec*.

У прикладі (рис. 3.2) у чергу додається завдання *sleep* на один вузол, з восьмипроцесорними ядрами, максимальним часом виконання 5 хвилин, використанням 100 МБ віртуальної пам'яті – *pbs_script sleep.pbs*.

Рис. 3.2. **Приклад файла *sleep.pbs***

Варто звернути увагу на ключ "–l". Після тире стоїть англійська літера "л", а не цифра "1".

Відправка завдання на кластер виконується за допомогою команди *qsub: qsub sleep.pbs*.

Рядки, що починаються з #PBS, є параметрами для команди *qsub*. Ці параметри можна також вказувати явно, під час виконання *qsub*.

Наприклад: qsub –N sleep –l nodes=1:ppn=8 –l pvmem = 100mb –d /home/student/ –l walltime = 5:00 sleep.pbs.

Параметри, відповідальні за надання ресурсів, можна вказувати через кому: qsub –N sleep –d /home/student/ –l nodes=1:ppn=8, pvmem = 100mb, walltime = 5:00 sleep.pbs.

Основні параметри команди *qsub*:

- – d path визначає робочу директорію для завдання. Якщо не задана, то робочою є домашня директорія користувача;
- – e path, – o path задаються імена файлів помилок (*stderr*), та стандартного виведення (*stdout*). За замовчуванням цей файл <ім'я_завдання>.e<job_id> та <ім'я_завдання>.o<job_id> у поточній директорії;
- – j oe, – j eo об'єднання файлів виведення/помилок. oe – файли об'єднуються в стандартний файл виведення, eo – у файл помилок;
- – m aben.

Події, під час яких треба відправляти повідомлення електронною поштою, "a" у разі аварійного завершення завдання, "b" у момент виконання завдання, "e" у момент завершення завдання, "n" – не відправляти

повідомлення. Можна вказати декілька букв "abe", або одну букву "n". За замовчуванням використовується "a".

- – M e-mail.

Адреса отримувача чи список адресатів отримувачів, кому будуть відправлені повідомлення. За замовчуванням – власник завдання;

- – N name.

Задає ім'я для файла виведення результатів та файла помилок;

- – q queue.

Задає чергу, в яку додається завдання. На сервері може бути декілька черг: основна *batch*, створюється за замовчуванням у процесі установки. Завдання ставляться в чергу *batch*;

- – l resource_list.

Визначає список ресурсів, необхідних для завдань.

Основні ресурси, що використовуються з опцією –l:

- nodes = N, nodes=N:ppn = C.

Визначає необхідну кількість вузлів, або вузлів та процесорних ядер. У першому випадку запитується N вузлів. У другому випадку запитується N вузлів, на кожному з яких використовується C процесорних ядер, де C може бути від 1 до 128. Наприклад: *nodes* = 24 запросить 24 вузли, *nodes* = 3: *ppn* = 8 запросить 3 вузли і 8 процесорних ядер на кожному, *nodes* = 6: *ppn* = 4 запросить 6 вузлів і 4 процесорних ядра на кожному. За замовчуванням виділяється одне процесорне ядро на вузол;

- pmem = size, pvmem = size.

Визначає необхідну одному процесу кількість фізичної та віртуальної пам'яті відповідно. Розмір вказується за допомогою цілого числа із суфіксом: b, kb, mb, gb. Наприклад, *pvmem* = 1gb запросить 1 Гб віртуальної пам'яті для процесу. За замовчуванням обмеження відсутнє. Однак у випадку, якщо завданню виділити дуже мало пам'яті, то воно може "зависнути" й "підвісити" обчислювальний вузол;

- walltime = time.

Визначає максимальний час виконання завдання. Після закінчення цього часу програму буде завершено. За замовчуванням встановлюється значення у 24 години. Обмеження максимального значення складає 72 години. Наприклад: walltime = 1:45:00 – завершити виконання завдання через 1 годину 45 хвилин.

Змінні оточення, які встановлює *Torque*:

- PBS_O_WORKDIR – директорія, у якій знаходиться користувач під час відправлення завдання в чергу;
- PBS_JOBID – унікальний номер завдання;
- PBS_O_HOST – поточний вузол, на якому запускають *pbs_script*;
- PBS_NODEFILE – файл, у якому перераховані всі обчислювальні вузли.

Черга завдань

Команда **qstat**

Переглянути стан завдань в черзі можна за допомогою команди *qstat*. Команда *qstat –a* надає більшої інформації. Поточний стан завдання відмічено в стовпці **S**:

- **Q** – завдання знаходиться в черзі, чекає звільнення ресурсів;
- **R** – завдання в даний момент виконується;
- **C** – завдання завершилося. Інформація про виконані завдання зберігається 5 хвилин;
- **E** – завдання завершилося помилкою.

За допомогою команди *qstat –n* можна отримати інформацію, які саме вузли виділені для запуску завдання.

Команда **qdel**

Якщо з якихось причин завдання так і не почало виконуватися, наприклад, потрібна дуже велика кількість процесорних ядер або пам'яті, то можна вилучити завдання із черги:

qdel <job_id>

Команда **pbstop**

Команда *pbstop* дозволяє займатися моніторингом основної інформації про стан кластера (загальна інформація, зайнятість вузлів та черг). Наприклад: *pbstop*.

Usage Totals: 100/135 Procs, 13/17 Nodes, 4/5 Jobs Running
14:00:49
Node States: 5 free 12 job-exclusive

```
visible CPUs: 0,1,2,3,4,5,6,7
                1          2          3          4
             ------------------------------------------------------------------------
  cl1n001 ........  ........  ........  ........
  cl1n005 vvvvvvvv gggggggg gggggggg gggggggg
  cl1n009 AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
  cl1n013 vvvvvvvv vvvvvvvv vvvvvvvv gggggggg
  cluster2 dddd....
             ------------------------------------------------------------------------
     Job#     Username Queue    Jobname   CPUs/Nodes  S
Elapsed/Requested
  v = 2098    volodin    batch      a32.exe          32/4       R  23:28:12/72:00:00
        2111    glas       batch      sfdyn            16/16      C  16:24:12/24:00:00
  g = 3235    galin      batch      a32h7i.out       32/32      R  52:23:53/72:00:00
  A = 3236    galin      batch      a32h7.out        32/32      R  52:10:34/72:00:00


  d = 2116    danilov    batch      test4            4/4        R  00:00:41/00:10:00
        [?] unknown  [@] busy  [*] down  [.] idle  [%] offline  [!] other
```

Черги завдань на кластері

На кластері діють декілька черг для завдань. Усі вузли кластера розподілені на декілька груп. Кожна черга може виконуватися тільки вузлом з однієї або декількох визначених груп (табл. 3.1).

Таблиця 3.1

Розподіл вузлів кластера між чергами

Назва черги	cluster2	cl1n001 – cl1n004	cl1n005 – cl1n016	cl1n017 – cl1n022	cl1n031 – cl1n042	Кількість ядер	Обмеження у часі
1	2	3	4	5	6	7	8
short**		+				32	2 години
long			+			96	96 годин
regular			+	+		144	24 години

Закінчення табл. 3.1

1	2	3	4	5	6	7	8
sixcore					+	144	24 години
mix**			+	+	+	288	24 години

Черги, відмічені (**), на даний момент відключені з технічних причин.

Для кожної черги встановлюються свої обмеження на час виконання завдань, кількість процесорів, об'єму пам'яті, дискового простору, кількості вузлів, що виділені для одного завдання та ін. Також є можливість ввести обмежений доступ різних користувачів, груп користувачів, системних груп до різних черг.

Щоб відправити завдання, наприклад, у чергу *long*, потрібно додати параметр –q long для команди *qsub: qsub –q long –N big-problem –l nodes = 4: ppn = 8 qbig.pbs*, або додати рядок *#PBS –q long* у *pbs_script* для *qsub*.

За замовчуванням тепер використовується черга *regular*.

Планувальник Maui

Локальний планувальник *Maui* займається опитуванням менеджера ресурсів *Torque* на предмет стану вільних ресурсів і завдань в черзі, які необхідно виконати. На основі отриманих даних і своїх налаштувань він приймає рішення про обмеження, що накладаються на завдання під час запуску і посилає сервісу *Torque* команду виконати їх.

Maui дозволяє гнучко налаштувати різні стратегії заповнення кластера. Установлювати обмеження для завдань за різними критеріями: кількістю запитуваних ресурсів, приладдя користувача до якоїсь групи і т. д. *Maui* дозволяє обирати різні політики планування, підтримує динамічну зміну пріоритетів та виключення. Усе це поліпшує керованість та ефективність обчислювальних вузлів, починаючи з невеликих кластерів, закінчуючи великими ЦОД.

Завдання

Далі розглянемо базові команди щодо керування завданнями на сервері *PBS*. Життєвий цикл завдання можна розподілити на чотири етапи:

1. Створення.
2. Подання до черги.

3. Виконання.
4. Завершення.

Створення. Створюється завдання. Воно компілюється та налагоджується. Створюється *pbs_script* з описанням вимог, що висуває завдання для виконання і зберігання всіх його параметрів. Вони можуть містити значення: тривалості виконання завдання (фактичний час виконання завдання), кількості ресурсів, необхідних для запуску завдання, робочу директорію та ін. Приклад файла *pbs_script job1.pbs* (рис. 3.3).

Рис. 3.3. **Приклад файла job1.pbs**

Цей *pbs_script* вказує ім'я файлів виведення та помилок: *job1*.

Вимоги, до апаратного забезпечення: завданню потрібні 2 ядра на одному вузлі (nodes = 1: ppn = 2). Максимальний час виконання протягом 10 хвилин (00:10:00). Коли час буде вичерпано, то завдання буде вилучено з черги.

Після завершення завдання треба вислати сповіщення про його завершення (виконання завдання або про його зрив) на електронну пошту (username @ localhost). Ще, користувач вказує, в якій директорії знаходиться завдання (/home/student/) та що потрібно для його виконання. У даному випадку у ролі завдання буде команда **(sleep 13)**. Файл опису завдання має розширення *pbs*.

Подання до черги. Завдання подається до черги за допомогою команди *qsub*. Після розгляду політик, встановлених адміністратором, визначається його пріоритет, а також формулювання завдання до обраної черги. Якщо в черзі не було завдань, а на кластері є вільний ресурс, достатній для вимог, що висунуто в *job1.pbs*, то завдання встановлюється на виконання. Приклад запуску завдання:

qsub job1.pbs.

Виконання. Завдання виконується на обчислювальних вузлах, обраних менеджером ресурсів, що налаштовані для використання черги.

Обрати самостійно конкретний вузол для виконання свого завдання користувач не в змозі. Отже, це робиться системою автоматично, в міру звільнення ресурсів черги. Під час виконання завдання його статус можна отримати за допомогою команди *qstat*. Приклад перевірки статусу завдання наведено на рис. 3.4.

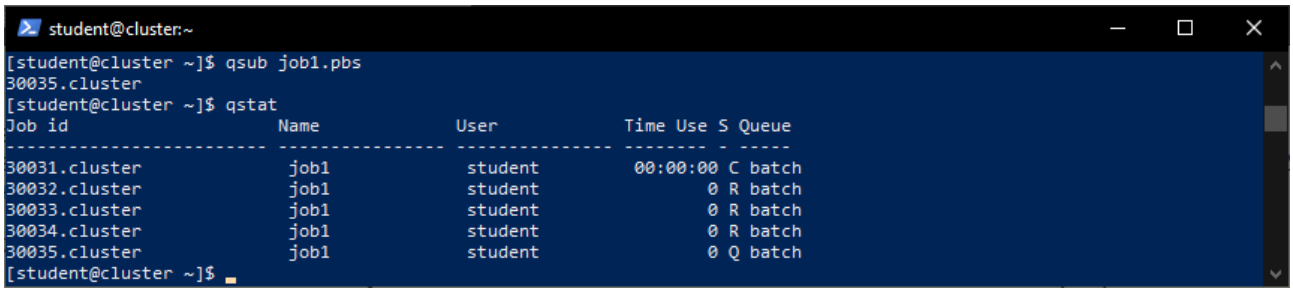

Рис. 3.4. **Перевірка статусу завдань**

Завершення. Коли виконання завдання закінчено, то за замовчуванням, файли *job1.o<job_id>* стандартного виведення і *job1.e<job_id>* стандартного потоку помилок копіюються в директорію, з якої було надіслано завдання (рис. 3.5).

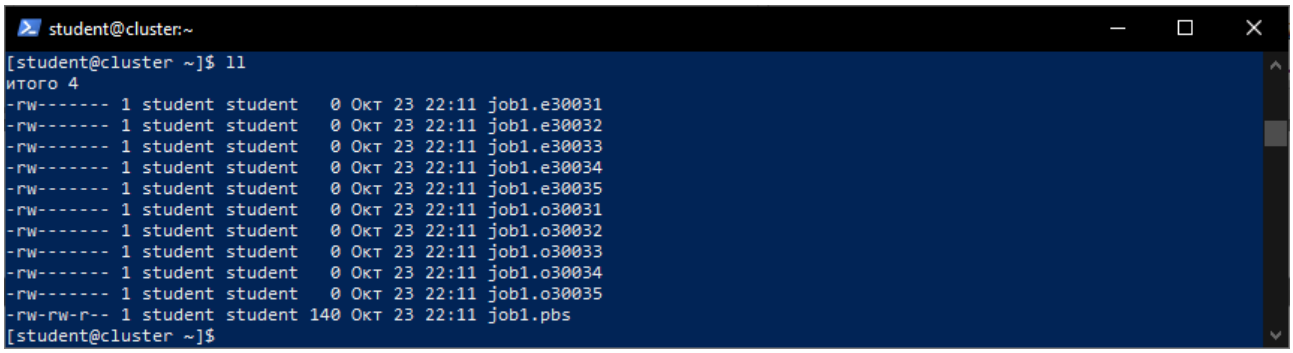

Рис. 3.5. **Файли стандартного виведення та потоку помилок**

Команди взаємодії зі кластером

У табл. 3.2 наведені основні команди визначення стану кластера та статистики роботи черг.

Таблиця 3.2

Команди для перегляду статистики та стану планувальника

Назва команди	Опис команди
showq	Відображає різні подання черги з точки зору активних завдань у черзі завдань, які простоюють та завдань, які не відповідають політикам планування
qstat	Відображає статус завдань
checkjob	Відображає стан завдання, потреби в ресурсах, обмеження повноважень, історію, надані ресурси та використання ресурсів
showbf	Показує наявність ресурсів у системі для виконання певного завдання
qnodes	Показує стан вузлів
canceljob	Відміна завдання, що виконується

Формулювання завдання

Виконати вправи та оформити результатами їх роботи у вигляді звіту з послідовності скриншотів.

1. Розробити програму, яка виводить назву факультету, групу та прізвище, ім'я та по батькові студента, який виконує лабораторну роботу.

2. Відповідно до варіанта завдання (табл. 3.3) розробити програму, яка перевіряє результат виконання однієї з математичних функцій.

3. Відповідно до варіанта завдання, що наведено далі, розробити програму, яка виконує оброблення одновимірних масивів.

Порядок виконання лабораторної роботи

1. Для входу на обчислювальний кластер потрібно виконати такі дії залежно від ОС, яку використовують:

- **для *Windows* 10** – увійти в *Windows PowerShell* та виконати команду: ssh.exe ім'я_користувача@IP_адреса_кластера –p порт_ssh_ з'єднання;
- **для *Linux*** – увійти в термінал та виконати команду: ssh ім'я_ користувача @IP_адреса_кластера: порт_ssh_з'єднання.

З'явиться запрошення для введення паролю *Password*. У цьому рядку треба ввести default_пароль.

Ім'я_користувача, default_пароль, IP_адресу_кластера, порт_ssh_ з'єднання отримати у викладача. Default_пароль можна змінити на індивідуальний. Якщо пароль втрачено, то відновити його можна за допомогою адміністратора кластера.

2. Створити директорію для виконання завдань лабораторної роботи та увійти в неї:

mkdir lab3

cd lab3.

У директорії створити файл програми zadanie1.cpp:

nano zadanie1.cpp.

Увести в нього код програми, наведений далі:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>
using namespace std;
int main(int argc, char **argv) {
    int i, j;
    const int num = 2;
    const int n = 500000;
    const int m = 32;
  double **a = new double *[m];
  double **b = new double *[m];
  double **c = new double *[m];
  for(i = 0; i < m; i++) {
     a[i] = new double[n];
     b[i] = new double[n];
     c[i] = new double[n];
  }
 omp_set_num_threads(num);
 double t1 = omp_get_wtime();
#pragma omp parallel for private (i,j) shared (a,b,c)
```

```
  for(i = 0; i < m; i++)
     for(j = 0; j < n; j++) {
        a[i][j] = double(rand()) / RAND_MAX;
        b[i][j] = double(rand()) / RAND_MAX;
                c[i][j] = double(rand()) / RAND_MAX;
     }
  double t2 = omp_get_wtime();

  cout << "****************************************" << endl;
  cout << "\t\tThreads: " << num << endl;
  cout << "\ta[" << m << "][" << n << "] = " << a[m - 1][n - 1] << endl;
  cout << "\tb[" << m << "][" << n << "] = " << b[m - 1][n - 1] << endl;
  cout << "\tc[" << m << "][" << n << "] = " << c[m - 1][n - 1] << endl;

  double t3 = omp_get_wtime();
#pragma omp parallel for private (i,j) shared (a,b,c)
  for(i = 0; i < m; i++)
     for(j = 0; j < n; j++)
           a[i][j] += b[i][j] + c[i][j];
  double t4 = omp_get_wtime();

  cout << "\tA[" << m << "][" << n << "] = " << a[m - 1][n - 1] << endl;
  cout << "****************************************" << endl;
  cout << "\tTime = " << (t2 - t1) + (t4 - t3) << endl;
  cout << "****************************************" << endl;
    for(i = 0; i < m; i++) {
      delete[] a[i];
      delete[] b[i];
      delete[] c[i];
  }
  delete[] a;
  delete[] b;
  delete[] c;
  return 0;
..}
```

Щоб зберегти файл, натисніть клавіші Ctrl + O. Задати нове ім'я чи погодитися з запропонованим. Для завершення роботи редактора натисніть клавіші Ctrl + X, а потім відкомпілювати програму:

g++ –O3 zadanie1.cpp –o zadanie1.bin –fopenmp.

3. Створити *pbs_script*, який описує вимоги для запуску програми *zadanie*1 на кластері:

nano zadanie1_script.pbs.

Увести в нього текст, що наведено далі:

```
#!/bin/bash
#PBS –d /home/student<number>//lab3/
#PBS –N zadanie1
#PBS –l nodes=1:ppn=2,walltime=00:10:00
#PBS –m e
#PBS –V
./zadanie1.bin
```

Зберегти зміни і вийти з редактора. Запустити завдання на виконання:

qsub zadanie1_script.pbs

4. Щоб подивитися результати роботи програми, потрібно переглянути файл виведення, який буде згенерований автоматично, після завершення її роботи:

cat zadanie1.o<job_id>.

Щоб подивитися помилки, що виникли під час роботи програми, потрібно переглянути файл помилок, який також буде згенеровано автоматично після завершення її роботи:

cat zadanie1.e<job_id>.

5. Навести статистичну інформацію, щодо стану завдань та кластера за командами з таблиці (див. табл. 3.2).

Виконання завдань провести за варіантами з табл. 3.3.

Таблиця 3.3

Варіанти завдань

№ з/п	Кількість вузлів та ядер	Функція	Опис	Приклад
1	nodes = 2 ppn = 2	**abs(a)**	модуль або абсолютне значення від **a**	abs(-3.0) = 3.0 abs(5.0) = 5.0
2	nodes = 3 ppn = 1	**sqrt(a)**	корінь квадратний з **a**, причому **a** не негативне	sqrt(9.0) = 3.0
3	nodes = 1 ppn = 2	**pow(a,b)**	зведення **a** в ступінь **b**	pow(2,3) = 8
4	nodes = 2 ppn = 1	**ceil(a)**	округлення **a** до найменшого цілого, але не менше ніж a	ceil(2.3) = 3.0 ceil(-2.3) = -2.0
5	nodes = 2 ppn = 2	**floor(a)**	округлення **a** до найбільшого цілого, але не більше ніж **a**	floor(12.4) = 12 floor(-2.9) = -3
6	nodes = 3 ppn = 2	**fmod(a, b)**	обчислення залишку від **a / b**	fmod(4.4, 7.5) = 4.4 fmod(7.5, 4.4) = 3.1
7	nodes = 1 ppn = 1	**exp(a)**	обчислення експоненти **e^a**	exp(0) = 1
8	nodes = 1 ppn = 2	**sin(a)**	**a** задається в радіанах	sin(0) = 0
9	nodes = 2 ppn = 1	**cos(a)**	**a** задається в радіанах	cos(0) = 1
10	nodes = 1 ppn = 2	**ln(a)**	натуральний логарифм **a** (основою є експонента)	ln(1.0) = 0.0
11	nodes = 3 ppn = 2	**lg(a)**	десятковий логарифм **a**	lg(10) = 1

Варіанти завдань розроблення програми

Створити одновимірний масив на n елементів, де 50 ≤ n ≤ 100. Кожен елемент масиву треба ініціалізувати з використанням функції генератора випадкових чисел зі значеннями від –9 до 9.

Розробити програму, згідно зі своїм варіантом, що виконує такі обчислення над масивом:

Варіант 1. У одновимірному масиві, що складається з n елементів, обчислити:

1) суму негативних елементів масиву;

2) добуток елементів масиву, розташованих між максимальним і мінімальним елементами;

3) упорядкувати елементи масиву за зростанням.

Варіант 2. У одновимірному масиві, що складається з n елементів, обчислити:

1) суму позитивних елементів масиву;

2) добуток елементів масиву, розташованих між максимальним за модулем і мінімальним за модулем елементами;

3) упорядкувати елементи масиву за спаданням.

Варіант 3. У одновимірному масиві, що складається з n елементів, обчислити:

1) добуток елементів масиву з парними номерами;

2) суму елементів масиву, розташованих між першим і останнім нульовими елементами;

3) перетворити масив таким чином, щоб спочатку в ньому розташовувалися всі позитивні, а потім – все негативні елементи (елементи зі значеннями 0 вважати позитивними). Відносний порядок елементів у масиві повинен зберігатися.

Варіант 4. У одновимірному масиві, що складається з n елементів, обчислити:

1) суму елементів масиву з непарними номерами;

2) суму елементів масиву, розташованих між першим і останнім негативними елементами;

3) стиснути масив, виокремивши з нього елементи, модуль яких < 3, але ≥ 1. А вивільненим елементам привласнити значення 10, розташувати на початку масиву. Відносний порядок елементів у масиві повинен зберігатися.

Варіант 5. У одновимірному масиві, що складається з n елементів, обчислити:

1) максимальний елемент масиву;

2) суму елементів масиву, розташованих між першим елементом масиву та останнім позитивним елементом;

3) стиснути масив, виокремивши з нього елементи, модуль яких ≤ 3, але > 1. А вивільненим елементам масиву привласнити значення 13, розташувати на початку масиву. Відносний порядок елементів у масиві повинен зберігатися.

Варіант 6. У одновимірному масиві, що складається з n елементів, обчислити:

1) мінімальний елемент масиву;

2) суму елементів масиву, розташованих між першим і останнім позитивними елементами;

3) перетворити масив таким чином, щоб в його кінці розташовувалися всі елементи, які будуть > –3, але ≤ 5, а спочатку – всі інші. Відносний порядок елементів у масиві повинен зберігатися.

Варіант 7. У одновимірному масиві, що складається з n елементів, обчислити:

1) номер максимального елемента масиву;

2) добуток елементів масиву, розташованих між першим і другим нульовими елементами;

3) перетворити масив таким чином, щоб у першій половині розташовувалися його елементи, що знаходяться в непарних позиціях, а в другій половині, що знаходяться в парних позиціях. Відносний порядок елементів у масиві повинен зберігатися.

Варіант 8. У одновимірному масиві, що складається з n елементів, обчислити:

1) номер мінімального елемента масиву;

2) суму елементів масиву, розташованих між першим і другим негативними елементами;

3) перетворити масив таким чином, щоб спочатку розташовувалися всі елементи, модуль яких ≥ 1, а потім – усі інші. Відносний порядок елементів у масиві повинен зберігатися.

Варіант 9. У одновимірному масиві, що складається з n елементів, обчислити:

1) максимальний за модулем елемент масиву;

2) суму елементів масиву, розташованих між першим і другим позитивними елементами;

3) перетворити масив таким чином, щоб елементи рівні 0, розташовувалися попереду всіх інших. Відносний порядок елементів у масиві повинен зберігатися.

Варіант 10. У одновимірному масиві, що складається з n елементів, обчислити:

1) мінімальний за модулем елемент масиву;

2) суму модулів елементів масиву, розташованих після першого елемента, рівного нулю;

3) перетворити масив таким чином, щоб у першій половині розташувалися його елементи, що стояли в парних позиціях, а в другій половині – елементи, що стояли в непарних позиціях. Відносний порядок елементів у масиві повинен зберігатися.

Варіант 11. У одновимірному масиві, що складається з n елементів, обчислити:

1) номер мінімального за модулем елемента масиву;

2) суму модулів елементів масиву, розташованих після першого негативного елемента;

3) стиснути масив, виокремивши з нього елементи, які ≥ 0 та < 2. А вивільненим елементам масиву привласнити значення 17, розташувати на початку масиву. Відносний порядок елементів у масиві повинен зберігатися.

Варіант 12. У одновимірному масиві, що складається з n елементів, обчислити:

1) мінімальний елемент масиву;

2) суму елементів масиву, розташованих між першим і останнім позитивними елементами;

3) перетворити масив таким чином, щоб в його кінці розташовувалися всі елементи, які будуть > –5, але ≤ 3, а спочатку – всі інші. Відносний порядок елементів у масиві повинен зберігатися.

У звіті виконання лабораторної роботи навести за кожним завданням:

вихідні файли програм;

файли *pbs_script*;

screenshot з результатами роботи програм;

screenshot з результатами роботи команд з табл. 3.2.

Контрольні запитання до лабораторної роботи 3

1. Яка загальна ідея роботи обчислювального кластера?
2. Що становлять завдання на кластері?
3. Для чого призначений, за що відповідає та чим керує менеджер ресурсів *torque*?
4. Що таке *PBS*?
5. Призначення *pbs_script*?
6. У якому вигляді додатки подаються на кластер?
7. Де можна знайти результати виконання завдання?
8. Яку інформацію можна подивитися за допомогою команд, що наведені в табл. 3.1?
9. Які файли утворюються після виконання команд?
10. У чому полягає принцип дії планувальників?
11. Наведіть приклади основних команд визначення стану кластера та статистики роботи черг.
12. У чому перевага планувальника *Maui* над вбудованим планувальником?
13. Як підготувати первинні тексти програми до виконання?

Лабораторна робота 4
Паралельне програмування у відкритому стандарті *OpenMP* та його застосування під час розв'язання системи лінійних алгебраїчних рівнянь (СЛАР)

Мета лабораторної роботі:

1. Дослідження принципів роботи стандарту *OpenMP*.

2. Набуття практичних навичок під час створення паралельних програм методом ітеративного та кінцевого паралелізму в стандарті *OpenMP*.

3. Розроблення паралельних програм розв'язання СЛАР методами Гауса, простої ітерації та сполучених градієнтів з використанням стандарту *OpenMP*.

Завдання:

1. Дослідити програми, що використовують під час роботи стандарту *OpenMP*, виконати їх на кластері в пакетному режимі (ПР) та проаналізувати результати роботи.

2. Дослідити роботу програм розв'язання системи лінійних алгебраїчних рівнянь методом Гауса без, а потім з використанням стандарту *OpenMP*. Виконати їх на кластері в ПР та проаналізувати результати роботи.

3. Розробити програми розв'язання СЛАР методами простої ітерації та сполучених градієнтів з використанням стандарту *OpenM*P, взявши за основу приклади розглянутих однопотокових реалізацій.

Загальні відомості про принципи роботи відкритого стандарту OpenMP

Стандарт *OpenMP* – це набір директив компілятора, бібліотек операцій та змінних оточення для використання паралелізму в *SMP*-системах зі спільною пам'яттю мовами C та C++. Стандарт *OpenMP* дозволяє мати користувачу один варіант програми для паралельного та послідовного виконання.

Методи розпаралелювання і моделі програм у відкритому стандарті OpenMP

Стандарт *OpenMP* передбачає SPMD-модель паралельного програмування, в рамках якої для усіх паралельних потоків використовується однаковий код програми.

Програма починається з послідовної області. Спочатку виконується один потік. Під час входу в паралельну область породжуються ще декілька потоків, між якими розподіляться частини коду програми.

Після завершення паралельної області всі потоки, крім одного (потік "майстер"), завершуються і починається послідовна область. У програмі може бути будь-яка кількість паралельних та послідовних областей.

Паралельні області можуть бути також вкладені одна в одну (рис. 4.1).

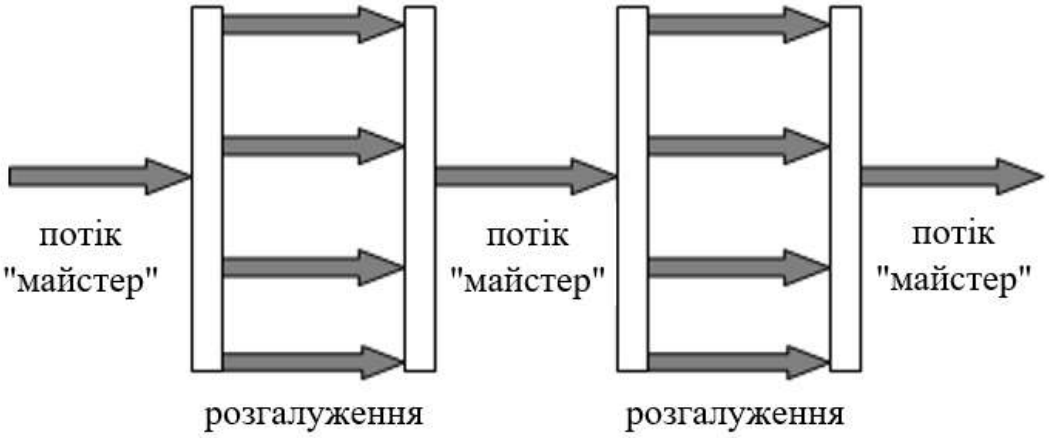

Рис. 4.1. **Модель розгалуження – об'єднання**

Для підвищення ефективності програми необхідно, щоб всі потоки, які в ній працюють одночасно, були рівномірно завантажені.

Важливо синхронізувати доступ до загальних даних, що використовуються під час роботи декількох потоків. Порушення цього правила може призвести до конфліктів під час одночасного неузгодженого доступу. Тому необхідно уважно відслідковувати синхронізацію працюючих потоків, явно використовуючи директиви синхронізації чи відповідні бібліотечні функції.

*OpenM*P має явну (не автоматичну) модель програмування, пропонуючи програмісту повне управління процесом розпаралелювання.

Порядок виконання лабораторної роботи

1. Для входу на обчислювальний кластер потрібно виконати такі дії залежно від ОС, яку використовують:

• **для *Windows* 10** – увійти в *Windows PowerShell*, виконати команду: ssh.exe ім'я_користувача@IP_адреса_кластера –p порт_ssh_ з'єднання;

• **для *Linux*** – увійти в термінал, виконати команду: ssh ім'я_ користувача @IP_адреса_кластера: порт_ssh_з'єднання.

З'явиться запрошення для введення паролю *Password*. У цьому рядку треба ввести default_пароль.

Ім'я_користувача, default_пароль, IP_адресу_кластера, порт_ssh_ з'єднання отримати у викладача. Default_пароль можна змінити на індивідуальний. Якщо пароль втрачено, то відновити його можна за допомогою адміністратора кластера.

2. Створити директорію для виконання завдань лабораторної роботи та увійти в неї:

mkdir lab4
cd lab4.

3. **Приклад 1.** Створити в редакторі *nano* текстовий файл example1.cpp:

nano example1.cpp.

Ввести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <omp.h>
using namespace std;

int main(int argc, char **argv) {
  int count, num;

   count=omp_get_num_threads();
   cout <<  endl;
   cout << " All threads: " << count << endl;
   #pragma omp parallel num_threads(5)
    {
      num = omp_get_thread_num();
      count=omp_get_num_threads();
```

```
      cout << " All threads: " << count << " Thread number: " << num << endl;
    }
   count=omp_get_num_threads();
   cout << " All threads: " << count << endl << endl;
 return 0;
}
```

Щоб зберегти файл, натисніть клавіші Ctrl + O та задайте нове ім'я чи погодьтесь із запропонованим. Для завершення роботи редактора натисніть клавіші Ctrl + X. Відкомпілювати програму командою:

g++ example1.cpp –o example1.bin –fopenmp,

де ключ – *fopenmp* використовується для вказівки компілятору обробляти всі директиви препроцесора, пов'язані зі стандартом *OpenMP*. Виклики функцій стандарту *OpenMP* будуть виконуватися незалежно від того, вказаний ключ чи ні. Ключ – дозволяє вказати ім'я файла для зберігання скомпільованої програми. Його можна не вказувати. При цьому, скомпільована програма буде зберігатися в файлі, створеному за замовчуванням з ім'ям a.out.

4. Для виконання програми в пакетному режимі на кластері треба створити в редакторі *nano pbs_script example1.pbs*:

nano example1.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab4/
#PBS –l nodes=1:ppn=2,walltime=00:30:00
#PBS –N example1
echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE
echo "Start date: `date`"
nohup ./example1.bin
echo " End date: `date`"
```

Зберегти зміни і вийти з редактора. Параметр *nodes* може набувати значення 1, 2, *node*1 або *node*2. Параметр *ppn* у *pbs_script* повинен збігатися з кількістю процесорних ядер вузла. Відправити завдання в чергу, запустивши на виконання *pbs_script*:

qsub example1.pbs.

5. Перевірити результати роботи програми можна, переглянувши вміст файла example1.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example1.e<job_id>:

cat example1.o<job_id>
cat example1.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//29896.cluster
Nodes for computing:
node2
node2
Start date: Чтв Жовт 13 11:13:29 EEST 2023
 All threads: 1
 All threads: 5 Thread number: 4
 All threads: 5 Thread number: 1
 All threads: 5 Thread number: 2
 All threads: 5 Thread number: 3
 All threads: 5 Thread number: 0
 All threads: 1
  End date: Чтв Жовт 13 11:13:29 EEST 2023
```

6. **Приклад 2.** Обчислити вираз: A×B + C×D, де A, B, C, D – матриці розмірністю *dimension* × *dimension*. Створити в редакторі *nano* текстовий файл example2.cpp:

nano example2.cpp.

Увести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <cstdlib>
#include <omp.h>
using namespace std;

int main(int argc, char**argv) {
  if (argc != 3) {
     cout << "\n !!!ERROR!!! Run of program: ./example2.bin <matrix
dimension> <number of threads>!!!ERROR!!!\n" << endl, exit(1);
     cout << "**********************************************************************"
<< endl;
  }
  int dimension = atoi(*++argv);
  int threads = atoi(*++argv);
  int i, j, k, initialisation = 5;
  double **a = new double *[dimension];
  double **b = new double *[dimension];
  double **c = new double *[dimension];
  double **d = new double *[dimension];
  double **e = new double *[dimension];
  double **f = new double *[dimension];
  for(i = 0; i < dimension; i++) {
     a[i] = new double[dimension];
     b[i] = new double[dimension];
     c[i] = new double[dimension];
     d[i] = new double[dimension];
     e[i] = new double[dimension];
     f[i] = new double[dimension];
  }

  omp_set_num_threads(threads);
  double t1 = omp_get_wtime();
  #pragma omp parallel for private (i,j) shared (a, b, c, d, e, f)
  for(i = 0; i < dimension; i++)
     for(j = 0; j < dimension; j++) {
        a[i][j] = initialisation % 10;
```

```
            b[i][j] = initialisation % 10;
            c[i][j] = initialisation % 10;
            d[i][j] = initialisation % 10;
            e[i][j] = 0;
            f[i][j] = 0;
            initialisation += 2;
        }

    double t2 = omp_get_wtime();
    cout << "****************************************" << endl;
    cout << "\tRAND Time = " << t2 - t1 << endl;
    cout << "****************************************" << endl;
    cout << "\tThreads: " << threads << endl;
    double t3 = omp_get_wtime();
    #pragma omp parallel for private(i, j, k) shared(a, b, e)
    for (i = 0; i < dimension; i++)
        for (j = 0; j < dimension; j++)
            for (k = 0; k < dimension; k++)
                e[i][j] += a[i][k] * b[k][j];

    #pragma omp parallel for private(i, j, k) shared(c, d, f)
    for (i = 0; i < dimension; i++)
        for (j = 0; j < dimension; j++)
            for (k = 0; k < dimension; k++)
                f[i][j] += c[i][k] * d[k][j];

    #pragma omp parallel for private (i,j) shared (e, f)
    for(i = 0; i < dimension; i++)
       for(j = 0; j < dimension; j++)
          e[i][j] += f[i][j];
    double t4 = omp_get_wtime();

    cout << "\te[" << dimension << "][" << dimension << "] = " << e[dimension -
1][dimension - 1] << endl;
    cout << "****************************************" << endl;
    cout << "\tRuntime = " << t4 - t3<< endl;
    cout << "****************************************" << endl;
```

```
  for(i = 0; i < dimension; i++) {
     delete[] a[i];
     delete[] b[i];
     delete[] c[i];
     delete[] d[i];
     delete[] e[i];
     delete[] f[i];
  }
  delete[] a;
  delete[] b;
  delete[] c;
  delete[] d;
  delete[] e;
  delete[] f;
 return 0;
}
```

Зберегти зміни, вийти з редактора та відкомпілювати програму:

g++ – O3 example2.cpp – o example2.bin – fopenmp,

де ключ – O3 компілятора, який відповідає за ступінь оптимізації програми.

7. Щоб виключити з порівняння результатів залежність від апаратних та програмних особливостей, програму буде виконано на одному й тому ж вузлі. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano pbs_script example2.pbs*:

nano example2.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab4/
#PBS –l nodes=node1:ppn=2,walltime=00:30:00
#PBS –N example2

echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
```

```
cat $PBS_NODEFILE

echo "Start date: `date`"
nohup ./example2.bin 1000 1
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Як видно з прикладу *pbs_script*, програма налаштована на виконання завдань на вузлі *node1*.

Відправити завдання в чергу треба, запустивши на виконання *pbs_script*:

qsub example2.pbs.

8. Перевірити результати роботи програми можна, переглянувши вміст файла example2.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example2.e<job_id>:

cat example2.o<job_id>
cat example2.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//29952.cluster
Nodes for computing:
node1
node1
Start date: Нед Жовт 16 20:22:38 EEST 2023.
****************************************
        RAND Time = 0.0469936
****************************************
        Threads: 1
        e[1000][1000] = 30000
****************************************
        Runtime = 66.8646
****************************************
  End date: Нед Жовт 16 20:23:45 EEST 2023.
```

9. Виконати програми декілька разів у пакетному режимі. Зробити скриншоти результатів роботи кожної програми.

Завдання 1

1. Опрацювати роботу програм-прикладів.

2. Скомпілювати, створити *pbs_scripts* й запустити програми-приклади в пакетному режимі.

3. Перевірити роботу програми *example2.cpp* під час різної кількості потоків, задаючи їх значення вручну іншим параметром у процесі виклику програми example2.bin у *pbs_script example2.pbs*.

Перший параметр (розмір матриць) змінювати не треба. Результати виконання програми example2.bin у пакетному режимі оформити у вигляді табл. 4.1.

Таблиця 4.1

Результати виконання програми

Threads	RAND Time, c	Runtime, c
1		
2		
4		
8		
16		
32		

У прикладі 2 розглянуто метод ітеративного паралелізму. Його особливістю є одночасне виконання одних і тих самих обчислювальних операцій над різними наборами даних.

Зараз розглянемо метод кінцевого паралелізму, де число паралельних областей залежить від інформаційної структури завдання, а не від вхідних даних програми.

Для підтримки такої організації паралельних обчислень в стандарті *OpenMP* передбачено директиви, що визначають набір незалежних програмних секцій *(sections)*, кожна з яких виконує свій потік.

Приклад 3. Створити в редакторі *nano* текстовий файл example3.cpp:

nano example3.cpp.

1. Увести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <omp.h>
using namespace std;

int main(int argc, char **argv ) {
 int n;
 #pragma omp parallel private(n) num_threads(5)
 {
  cout << "All threads: " << omp_get_num_threads() << endl;
  n=omp_get_thread_num();
  #pragma omp sections
  {
   #pragma omp section
   {
    cout << "1 section, thread: " <<  n << endl;
   }
   #pragma omp section
   {
    cout << "2 section, thread: " <<  n << endl;
   }
   #pragma omp section
   {
    cout << "3 section, thread: " <<  n << endl;
   }
  }
 cout << "Parallel region, thread: " << n << endl;
 }
 return 0;
}
```

Зберегти зміни і вийти з редактора. Відкомпілювати програму:

g++ example3.cpp –o example3.bin –fopenmp.

2. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі nano pbs_script example3.pbs:

nano example3.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab4/
#PBS –l nodes=1:ppn=2,walltime=00:30:00
#PBS –N example3

echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
nohup ./example1.bin
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання *pbs_script*:

qsub example3.pbs.

3. Перевірити результати роботи програми можна, переглянувши вміст файла example3.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example3.e<job_id>:

cat example3.e<job_id>.

Результат виконання програми наведений далі.

```
.File containing nodes:
/var/spool/torque/aux//29972.cluster
Nodes for computing:
node2
node2
Start date: Пнд Жовт 17 01:30:24 EEST 2023
All threads: 5
All threads: 5
All threads: 5
1 section, thread: 1
2 section, thread: 0
3 section, thread: 1

All threads: 5
All threads: 5

Parallel region, thread: 4
Parallel region, thread: 1
Parallel region, thread: 3
Parallel region, thread: 0
Parallel region, thread: 2
End date: Пнд Жовт 17 01:30:24 EEST 2023
```

Приклад 4. Обчислити вираз: A×B + C×D, де A, B, C, D – матриці розмірністю *dimension* × *dimension*, за допомогою секцій.

Створити в редакторі *nano* текстовий файл example4.cpp:

nano example4.cpp.

Увести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <cstdlib>
#include <omp.h>
using namespace std;

int main(int argc, char**argv) {
  if (argc != 3) {
```

```
    cout << "\n !!!ERROR!!! Run of program: ./example2.bin <matrix
dimension> <number of threads>!!!ERROR!!!\n" << endl, exit(1);
    cout << "**************************************************************************"
<< endl;
  }
  int dimension = atoi(*++argv);
  int threads = atoi(*++argv);
  int i, j, k, initialisation = 5;
  double **a = new double *[dimension];
  double **b = new double *[dimension];
  double **c = new double *[dimension];
  double **d = new double *[dimension];
  double **e = new double *[dimension];
  double **f = new double *[dimension];
  for(i = 0; i < dimension; i++) {
    a[i] = new double[dimension];
    b[i] = new double[dimension];
    c[i] = new double[dimension];
    d[i] = new double[dimension];
    e[i] = new double[dimension];
    f[i] = new double[dimension];
  }
  omp_set_num_threads(threads);
  double t1 = omp_get_wtime();

  #pragma omp parallel for private (i,j) shared (a, b, c, d, e, f)
  for(i = 0; i < dimension; i++)
    for(j = 0; j < dimension; j++) {
      a[i][j] = initialisation % 10;
      b[i][j] = initialisation % 10;
      c[i][j] = initialisation % 10;
      d[i][j] = initialisation % 10;
      e[i][j] = 0;
      f[i][j] = 0;
      initialisation += 2;
    }

  double t2 = omp_get_wtime();
  cout << "****************************************" << endl;
```

```
  cout << "\tRAND Time = " << t2 - t1 << endl;
  cout << "****************************************" << endl;
  cout << "\tThreads: " << threads << endl;
  double t3 = omp_get_wtime();
  #pragma omp parallel private(i, j, k) shared(a, b, c, d, e, f)
  {
   #pragma omp sections
   {
    #pragma omp section
    {
     for (i = 0; i < dimension; i++)
         for (j = 0; j < dimension; j++)
             for (k = 0; k < dimension; k++)
                 e[i][j] += a[i][k] * b[k][j];
    }
    #pragma omp section
    {
     for (i = 0; i < dimension; i++)
         for (j = 0; j < dimension; j++)
             for (k = 0; k < dimension; k++)
                 f[i][j] += c[i][k] * d[k][j];
    }
   }
  }
  #pragma omp parallel for private (i,j) shared (e, f)
   for(i = 0; i < dimension; i++)
      for(j = 0; j < dimension; j++)
          e[i][j] += f[i][j];
  double t4 = omp_get_wtime();

  cout << "\te[" << dimension << "][" << dimension << "] = " << e[dimension -
1][dimension - 1] << endl;
  cout << "****************************************" << endl;
  cout << "\tRuntime = " << t4 - t3<< endl;
  cout << "****************************************" << endl;
  for(i = 0; i < dimension; i++) {
     delete[] a[i];
     delete[] b[i];
```

```
        delete[] c[i];
        delete[] d[i];
        delete[] e[i];
        delete[] f[i];
    }
    delete[] a;
    delete[] b;
    delete[] c;
    delete[] d;
    delete[] e;
    delete[] f;
 return 0;
}
```

Зберегти зміни і вийти з редактора. Відкомпілювати програму:

g++ –O3 example4.cpp –o example4.bin –fopenmp.

4. Щоб виключити з порівняння результатів залежність від апаратних та програмних особливостей, програму буде виконано на одному й тому ж вузлі. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі nano pbs_script example4.pbs:

nano example4.pbs.

Ввести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab4/
#PBS –l nodes=node2:ppn=2,walltime=00:30:00
#PBS –N example4

echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE
```

```
echo "Start date: `date`"
nohup ./example4.bin 1000 1
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Як видно з прикладу *pbs_script*, програма налаштована на виконання на вузлі *node*2. Відправити завдання в чергу, запустивши на виконання *pbs_script*:

qsub example4.pbs.

5. Перевірити результати роботи програми можна, переглянувши вміст файла example4.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example4.e<job_id>:

cat example4.o<job_id>

cat example4.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//29973.cluster
Nodes for computing:
node2
node2
Start date: Пнд Жовт 17 03:39:39 EEST 2023
****************************************
      RAND Time = 0.0391331
****************************************
      Threads: 1
      e[1000][1000] = 30000
****************************************
      Runtime = 72.421
****************************************
  End date: Пнд Жовт 17 03:40:52 EEST 2023
```

6. Виконати програми декілька разів у пакетному режимі. Зробити *screenshot* результатів роботи кожної програми.

Завдання 2

1. Опрацювати роботу програм-прикладів.

2. Скомпілювати, створити *pbs_scripts* й запустити програми-приклади в пакетному режимі.

3. Перевірити роботу програми *example4.cpp* за умови різної кількості потоків, задаючи їх значення вручну другим параметром під час виклику програми *example4.bin* у *pbs_script example4.pbs*. Перший параметр (розмір матриць) змінювати не треба. Результати виконання програми *example4.bin* у пакетному режимі оформити у вигляді табл. 4.2.

Таблиця 4.2

Результати виконання програми

Threads	RAND Time, c	Runtime, c
1		
2		
4		
8		
16		
32		

Виконати індивідуальні завдання, видані викладачем згідно з варіантами у табл. 4.3.

Таблиця 4.3

Індивідуальні завдання

№ варіанта	Завдання
1	2
1	Обчислити вираз A + B × C – D, використовуючи метод ітеративного паралелізму, де A, B, C, D – матриці розмірністю 1 000 × 1 000
2	Обчислити вираз A × B + C – D, використовуючи метод ітеративного паралелізму, де A, B, C, D – матриці розмірністю 1 000 × 1 000

Закінчення табл. 4.3

1	2
3	Обчислити вираз A × B – C + D, використовуючи метод кінцевого паралелізму, де A, B, C, D – матриці розмірністю 1 000 × 1 000
4	Обчислити вираз: A – B × C + D, використовуючи метод кінцевого паралелізму, де A, B, C, D – матриці розмірністю 1 000 × 1 000

Загальне формулювання задачі розв'язку СЛАР методом Гауса

СЛАР часто виникають під час вирішення прикладних інженерних та наукових задач: математичного програмування, статистичного оброблення даних, методів скінченних різниць, методів скінченних елементів, тощо.

Лінійне рівняння з n невідомими $x_0, x_1, \ldots x_{n-1}$ може бути записано як:

$$a_0x_0 + a_1x_1 + \ldots + a_{n-1}x_{n-1} = b$$

у якому $a_0, a_1, \ldots, a_{n-1}$ та b є величини, що мають постійне значення. Множина n лінійних рівнянь:

$$\begin{cases} a_{0,0}x_0+a_{0,1}x_1+\ldots+a_{0,n-1}x_{n-1} = b_0 \\ a_{1,0}x_0+a_{1,1}x_1+\ldots+a_{1,n-1}x_{n-1} = b_1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n-1,0}x_0+a_{n-1,1}x_1+\ldots+a_{n-1,n-1}x_{n-1} = b_{n-1} \end{cases}$$

має назву система лінійних рівнянь.

У матричному вигляді така система записується як:

$$Ax = b,$$

де $A=(a_{i,j})$ – дійсна матриця розміром nn елементів;

x та b – вектори, що мають розмір n елементів.

Рішенням системи лінійних рівнянь для заданої матриці A та вектора b є знаходження значень вектора x, для якого виконуються всі лінійні рівняння системи.

У лабораторній роботі для рішення СЛАР використано метод Гауса.

Цей метод є прямим алгоритмом рішення СЛАР, для котрих матриці коефіцієнтів є щільними. Якщо СЛАР є невиродженою, то метод Гауса гарантує знаходження рішення з помилкою, визначеною точністю комп'ютерних розрахунків. Головна ідея методу полягає у приведенні матриці A

шляхом еквівалентних перетворень (не змінюючих рішення системи) до трикутного вигляду. Після цього значення невідомих $x_0, x_1, \ldots x_{n-1}$ можуть бути отримані безпосередньо в явному вигляді.

Метод Гауса базується на можливості перебудови лінійних рівнянь, котрі не змінюють при цьому рішення системи. Такі перебудови мають назву еквівалентних. Приклади таких перебудов: перестановка рівнянь, помноження будь-якого з рівнянь на ненульову константу, додавання до рівняння будь-якого іншого рівняння системи тощо.

Метод Гауса містить виконання двох етапів:

1. Прямий хід метода Гауса. Первинна СЛАР за допомоги послідовного виключення невідомих наводиться до трикутного вигляду відносно головної діагоналі:

$$Ax = b,$$

де матриця коефіцієнтів має вигляд:

$$A = \begin{bmatrix} a_{0,0} a_{0,1} \ldots a_{0,n-1} \\ 0 a_{1,1} \ldots a_{1,n-1} \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ 0 0 \ldots a_{n-1,n-1} \end{bmatrix}.$$

2. Зворотній хід метода Гауса. Знаходимо значення невідомих $x_0, x_1 \ldots x_{n-1}$ останнього рівняння перебудованої системи, обчислюємо значення змінної x_{n-1}, з передостаннього рівняння вичислити значення змінної x_{n-2} й далі, до змінної x_0.

Алгоритм реалізації рішення СЛАР методом Гауса

Прямий хід реалізації алгоритму містить послідовне виключення невідомих у рівняннях СЛАР, що розв'язується. На ітерації i, де $0 \le i < n - 1$, буде виконано виключення невідомих i для усіх рівнянь з номерами k, більших за i, де $i < k \le n - 1$. Для реалізації алгоритму потрібно віднімання від рядків рядка i, помноженого на константу $a_{ki} \div a_{ii}$ таким чином, щоб коефіцієнт у рядках при змінної $x_i = 0$. Такі ж самі обчислення виконуються й над елементами вектора b. Усі необхідні обчислення можуть бути записані співвідношеннями:

$$\left.\begin{aligned} a'_{kj} &= a_{kj} - (a_{ki} \div a_{ii}) \times a_{ij} \\ b'_k &= b_k - (a_{ki} \div a_{ii}) \times b_i \end{aligned}\right\} i \le j \le n - 1,\ i < k \le n - 1, 0 \le i < n - 1.$$

На рис. 4.2 зображена схема стану даних на *i*-й ітерації прямого ходу алгоритму Гауса розв'язання СЛАР. Усі коефіцієнти при невідомих, що розташовані нижче головної діагоналі та лівіше стовпця *i,* вже є нульовими. На *i*-й ітерації прямого ходу метода Гауса здійснюється обнуління коефіцієнтів стовпця *i,* розташованих нижче головної діагоналі, шляхом віднімання рядка *i,* помноженого на ненульову константу. Після проведення *(n – 1)*-ї ітерації початкова матриця набуде трикутного вигляду відносно головної діагоналі.

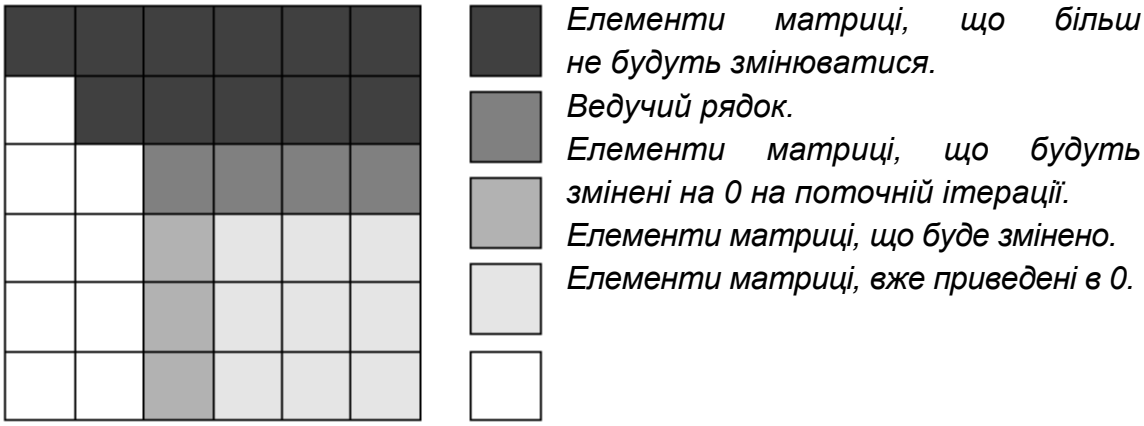

Рис. 4.2. **Стан даних на *i*-й ітерації прямого ходу алгоритму**

Під час виконання прямого ходу реалізації алгоритму методу Гауса рядок, який використовується для виключення невідомих, називають ведучим, а діагональний елемент ведучого рядка – ведучим елементом.

Виконання обчислень є можливим за умови, що ведучий елемент не дорівнює 0. У випадку, коли ведучий елемент має мале значення, то ділення та множення рядків на цей елемент призводить до накопичування обчислювальної помилки та нестійкості роботи алгоритму.

Для запобігання подібних проблем потрібно під час виконання кожної наступної ітерації прямого ходу методу Гауса знаходити коефіцієнт з максимальним значенням за абсолютним значенням у стовпці відповідно до виключного невідомого:

$$y = max_{i \le k \le n-1} V a_{ki} V,$$

та обрати ведучий рядок, у якому розташований цей коефіцієнт (метод головних елементів).

Зворотній хід алгоритму Гауса. Після зведення матриці коефіцієнтів до трикутного вигляду відносно головної діагоналі, з'являється можливість знаходження значення невідомих. З останнього рівняння перебудованої системи може бути знайдено значення змінної $x_{n-1,}$ з передостаннього рівняння вичислити значення змінної $x_{n-2,}$ й далі, до змінної x_0. Виконані обчислення можуть бути подані за допомогою таких співвідношень:

$$\left.\begin{array}{l} x_{n-1} = b_{n-1} \div a_{n-1,\,n-1} \\ x_i = \left(b_i - x_j\right) \div a_{ii} \end{array}\right\} \; i = n-2,\, n-3,\ldots,0.$$

Схема алгоритму метода Гауса наведена на рис. 4.3.

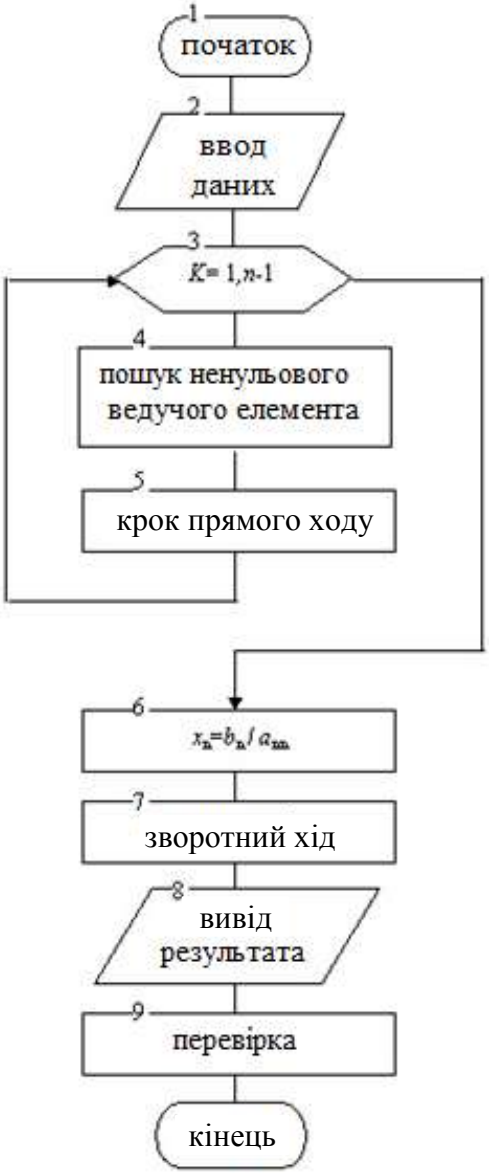

Рис. 4.3. **Схема алгоритму реалізації розв'язання СЛАР методом Гауса**

Програмна реалізація алгоритму рішення СЛАР методом Гауса з застосуванням відкритого стандарту OpenMP

1. Для перевірки роботи програми рішення СЛАР методом Гауса потрібно згенерувати матрицю коефіцієнтів. Для цього треба створити в редакторі *nano* текстовий файл gaussmatrixfactor.cpp:

nano gaussmatrixfactor.cpp.

Увести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <fstream>
using namespace std;

int main(int argc, char**argv) {
        int i, j;
        if (argc != 2)
                cout << "\n !!!ERROR!!! Example of command: ./gaussmatrixfactor
<SoLE dimension> !!!ERROR!!!\n" << endl, exit(1);
        cout << ************************************************************" << endl;
        int num = atoi(*++argv); // Enter the size SoLE
        cout << "\t\tSoLE dimension = " << num << endl;
        double **massiv = new double *[num];
        for(i = 0; i < num; i++)
       massiv[i] = new double[num + 1];
        double *x = new double[num];
    srand(time(NULL));
        for(i = 0; i < num; i++)
            for(j = 0; j < num; j++)
                while(!(massiv[i][j] = rand() % 10));
    for(j = 0; j < num; j++)
        while(!(x[j] = rand() % 10));
```

```
    for(i = 0; i < num; i++) {
       massiv[i][num] = 0;
           for(j = 0; j < num; j++)
              massiv[i][num] += massiv[i][j] * x[j];
           }

        ofstream outFile;
        outFile.open("MatrixFactor.txt");
        for(i = 0; i < num; i++) {
                for(j = 0; j <= num; j++)
                   outFile << massiv[i][j] <<" ";
       outFile << endl;
     }
        outFile <<
"*****************************************************************************************
*************************************************************************" << endl;
        outFile << "\tEquation roots: ";
        for(i = 0; i < num; i++)
                outFile << x[i] <<" ";
    outFile << endl;
    outFile <<
"*****************************************************************************************
*************************************************************************" << endl;
    outFile << "\tMatrix dimension: " << num << " x " << num + 1 << endl;

        cout << "\t\tFile MatrixFactor.txt created" << endl;
        cout << ************************************************************" << endl;
        for (i = 0; i < num; i++)
                delete[] massiv[i];
        delete[] massiv;
        delete[] x;
        outFile.close();
        return 0;
}
```

Зберегти зміни і вийти з редактора. Відкомпілювати програму:

g++ –O3 gaussmatrixfactor.cpp –o gaussmatrixfactor.bin.

2. Програма генератор виконується в одному потоці. Тому, для неї достатньо значення ppn = 1. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano pbs_script gaussmatrixfactor.pbs:*

nano gaussmatrixfactor.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab4/
#PBS –l nodes=1:ppn=1,walltime=01:00:00
#PBS –N gaussmatrixfactor

echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
nohup ./gaussmatrixfactor.bin 1000
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання *pbs_script*:

qsub gaussmatrixfactor.pbs.

3. Перевірити результати роботи програми можна переглянувши вміст файла gaussmatrixfactor.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі gaussmatrixfactor.e<job_id>:

cat gaussmatrixfactor.o<job_id>

cat gaussmatrixfactor.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//29989.cluster
Nodes for computing:
node2
Start date: Срд Жовт 19 15:52:14 EEST 2023
**************************************************************************
        SoLE dimension = 5000
        File MatrixFactor.txt created
**************************************************************************
 End date: Срд Жовт 19 15:52:37 EEST 2023
```

Ще буде створено файл *MatrixFactor.txt* з матрицею коефіцієнтів, змінними та вказано розмір створеної матриці. Дані в файлі *MatrixFactor.t*xt будуть перезаписані під час кожного нового виклику *gaussmatrixfactor.bin.*

У якості параметра програма приймає розмір СЛАР. Варто враховувати ці особливості виклику програми *gaussmatrixfactor.bin.*

4. Переходимо безпосередньо до перевірки роботи програми рішення СЛАР методом Гауса. Задля цього треба створити в редакторі *nano* текстовий файл *gaussomp.cpp*:

nano gaussomp.cpp.

Ввести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <omp.h>
#include <fstream>
#include <cmath>

using namespace std;

int main(int argc, char**argv) {
    if (argc != 3)
```

```
            cout << "\n !!!ERROR!!! Example of command: ./gaussomp <SoLE
dimension> <number of processor cores> !!!ERROR!!!\n" << endl,
exit(EXIT_FAILURE);

        cout <<
"*****************************************************************************************
*************************" << endl;
        int num = atoi(*++argv); // Enter the size SoLE
        int numproc = atoi(*++argv); // Enter the number of processors
        cout << "\tSoLE dimension = " << num << "\tNumber of processor cores
= " << numproc << endl;
        double **massiv = new double *[num];
        for(int i = 0; i < num; i++)

        massiv[i] = new double[num + 1];
   double *x = new double[num];

   ifstream inFile;
   inFile.open("MatrixFactor.txt");
   if(!inFile.is_open()) {
      cout << "\tCould not open the file MatrixFactor.txt" << endl;
      cout <<
"*****************************************************************************************
**********************" << endl;
      exit(EXIT_FAILURE);
   }
        for(int i = 0; i < num; i++)
                for(int j = 0; j <= num; j++)
                        inFile >> massiv[i][j];
        for(int i = 0; i < num; i++)
                if(!massiv[i][i]) {
                        cout << "The diagonal element massiv[" << i + 1<< "][" << i +
1<< "] = 0" << endl;
                        exit(EXIT_FAILURE);
                }
        omp_set_num_threads(numproc);
        int i, j;
```

```
        double t1 = omp_get_wtime();
        for(int k = 0; k < num; k++)  {
                        double tmp = abs(massiv[k][k]);
                        i = k;

#pragma omp parallel for private (j) firstprivate(k) shared (massiv)
                        for(j = k + 1; j < num; j++)
                                if(abs(massiv[j][k]) > tmp)
                                        i = j, tmp = abs(massiv[j][k]);

                        if (i != k) {
#pragma omp parallel for private (j, tmp) firstprivate(k) shared (massiv)
                                for(j = k; j <= num; j++) {
                                        tmp = massiv[k][j];
                                        massiv[k][j] = massiv[i][j];
                                        massiv[i][j] = tmp;
                                }
                        }

#pragma omp parallel for private (j) firstprivate(k) shared (massiv)
                        for (j = num; j > k; --j)
                    massiv[k][j] /= massiv[k][k];
                        massiv[k][k] = 1;

#pragma omp parallel for private (i, j) firstprivate(k) shared (massiv)
                        for (i = k + 1; i < num; i++) {
                                if (massiv[i][k])
                                        for (j = num; j >= k; --j)
                                            massiv[i][j] -= massiv[i][k] * massiv[k][j];
                        }
                }

              for (j = num - 1; j >= 0; j--) {
                        x[j] = massiv[j][num];
#pragma omp parallel for private (i) firstprivate(j) shared (massiv)
                        for (i = j; i >= 0; i--)
                            massiv[i][num] -= massiv[i][j] * x[j];
                 }
        double t2 = omp_get_wtime();
```

```
    cout << "\n\t\t\tRuntime = " << t2 - t1 << endl;
    cout << endl << "Solution:";
    for (i = 0; i < 5; ++i)
         cout << "  x" << i + 1 << " =" << setw(2) << x[i];
    cout << " ...";
    for (i = num - 5; i < num; ++i)
         cout << "  x" << i + 1 << " =" << setw(2) << x[i];
    cout << endl;
    cout << "**********************************************************
**********************************************************" << endl;
    for (i = 0; i < num; i++)
          delete[] massiv[i];
   delete[] massiv;
   delete[] x;
 return 0;
}
```

Зберегти зміни і вийти з редактора та відкомпілювати програму:

g++ –O3 gaussomp.cpp –o gaussomp.bin –fopenmp.

5. Щоб виключити з порівняння результатів залежність від апаратних та програмних особливостей, програму буде виконано на одному й тому ж вузлі.

Є можливість обирати між *node*1 та *node*2.

Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano pbs_script gaussomp.pbs*:

nano gaussomp.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab4/
#PBS –l nodes=node1:ppn=2,walltime=01:00:00
#PBS –N gaussomp

echo "File containing nodes:"
echo $PBS_NODEFILE
```

```
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
nohup ./gaussomp.bin 5000 2
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання *pbs_script*:

qsub gaussomp.pbs.

6. Перевірити результати роботи програми можна переглянувши вміст файла gaussomp.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі gaussomp.e<job_id>:

cat gaussomp.o<job_id>

cat gaussomp.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//30000.cluster
Nodes for computing:
node1
node1
Start date: Срд Жовт 19 23:44:56 EEST 2023
********************************************************************************
*****
      SoLE dimension = 5000   Number of processor cores = 2

                  Runtime = 132.581

Solution:  x1 = 8  x2 = 9  x3 = 9  x4 = 7  x5 = 6 ...  x4996 = 4  x4997 = 9
x4998 = 3  x4999 = 9  x5000 = 9
**************************************************
**************************************************
  End date: Срд Жовт 19 23:47:16 EEST 2023
```

7. Перевірити працездатність програми у разі застосування різних розмірностей СЛАР.

Результати виконання завдання оформити у вигляді табл. 4.4.

Таблиця 4.4

Результати виконання завдання

Розмірність СЛАР	Кількість ядер, шт.	Час виконання, с
1 000	1	
1 000	2	
2 000	1	
2 000	2	
3 000	1	
3 000	2	
4 000	1	
4 000	2	
5 000	1	
5 000	2	

Проаналізувати результати табл. 4.4. Пояснити їх, та зробити висновки до ефективності розпаралелювання методу Гауса з використанням відкритого стандарту OpenMP.

Розв'язання СЛАР методом простої ітерації

Дана система лінійних алгебраїчних рівнянь:

$$Ax = f$$

$$a_{0,0}x_0 + a_{0,1}x_1 + \ldots + a_{0,n}x_n = f_0$$

$$a_{1,0}x_0 + a_{1,1}x_1 + \ldots + a_{1,n}x_n = f_1$$

... ...

$$a_{n,0}x_0 + a_{n,1}x_1 + \ldots + a_{n,n}x_n = f_n$$

Коріння цієї системи методом простої ітерації обчислюють за формулою:

$$x_i^{k+1} = x_i^k - \tau \times \left(\sum_{j=0}^{N} a_{ij} x_j^k - f_i \right), i = 0, ..., N,$$

де верхній індекс позначає номер ітераційного кроку;
i – номер кореня у правій частині;
τ – ітераційний крок.

Завершення обчислень виконується за умови:

$$\frac{\left\| \sum_{j=0}^{N} a_{ij} x_{ij}^k - f_i \right\|}{\| f \|} < \varepsilon, \qquad (4.1)$$

де $\| f \| = \sqrt{\sum_{i=0}^{N} (f_i \times f_i)}$.

8. Програмна реалізація послідовного алгоритму методу простої ітерації розв'язання СЛАР. Задля цього треба створити в редакторі *nano* текстовий файл *siterations.cpp*:

nano siterations.cpp.

Увести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <sys/time.h>
using namespace std;
int main(int argc, char**argv) {
  int i, j;
  const int N = 190;
  long int dt;
  double t, e;
  double A[N][N], F[N], mf, X[N], X1[N], S[N], msf;
  struct timeval tv1, tv2;
```

/* Генерація даних. Тут задається матриця з елементами, які дорівнюють значенню 1, а елементи діагоналі дорівнюють значенню 2. Матриця береться добре обумовленою. У правій частині, що дорівнює N + 1, усі корені будуть дорівнювати 1. */

```
     mf = 0;
   for(i = 0; i < N; i++)
    { for(j = 0; j < N; j++)
       (i==j)?(A[i][j]=2):(A[i][j]=1);
      F[i] = N+1;
 /* Розраховується сума квадратів елементів вектора F, тобто підкорений вираз. */
      mf += F[i]*F[i];
    }
/* Задаються кроки t і e. */
   t = 0.01;
   e = 0.00001;
/* Задається начальне приближення коренів. У X1 зберігаються значення коренів (к + 1)-ї ітерації. */
   for(i = 0; i < N; i++)
    X1[i] = 0.6;

/* Визначається стартовий час виконання обчислень. */
    gettimeofday(&tv1,NULL);

    do
     { for(i = 0; i < N; i++)
/* У X зберігаються значення коренів k-ї ітерації. */
        X[i] = X1[i];
       for(msf=0,i = 0; i < N; i++) {

        for(S[i] = 0,j = 0; j < N; j++)
              S[i] += A[i][j] * X[j];
               X1[i] = X[i] - t*(S[i] - F[i]);
 /* Розраховується сума квадратів елементів нев'язки. */
```

```
            msf += (S[i] - F[i]) * (S[i] - F[i]);
          }
        }
      while(msf/mf > e*e);   /* Перевіряється вимога (4.1). */

/* Розраховується час закінчення обчислень. */
    gettimeofday(&tv2,NULL);
    dt = (tv2.tv_sec - tv1.tv_sec) × 1000000 + tv2.tv_usec - tv1.tv_usec;
    cout << "************************************************************" << endl;
/* Виведення часу обчислень. */
    cout << "\t\tTime = " << dt << "\tSoLE dimension = " << N << endl;
/* Виведення перших 4-х коренів. */
    for(i = 0; i < 8; i++)
         cout << " X" << i << " = " << X[i];
    cout << endl;
    cout << "************************************************************" << endl;
return 0;
}
```

Зберегти зміни і вийти з редактора та відкомпілювати програму:

g++ –O3 siterations.cpp –o siterations.bin.

9. Програма генератор виконується в одному потоці. Тому для неї достатньо значення *ppn=1*. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano pbs_script siterations.pbs*:

nano siterations.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab4/
#PBS –l nodes=1:ppn=1,walltime=00:30:00
#PBS –N siterations
```

```
echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
nohup ./siterations.bin
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора, відправити завдання в чергу, запустивши на виконання pbs_script:

qsub siterations.pbs.

10. Перевірити результати роботи програми можна, переглянувши вміст файла siterations.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі siterations.e<job_id>:

cat siterations.o<job_id>

cat siterations.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//30025.cluster
Nodes for computing:
node2
Start date: Птн Жовт 21 09:08:05 EEST 2023

****************************************************************
              Time = 10999     SoLE dimension = 190
 X0 = 1.00001 X1 = 1.00001 X2 = 1.00001 X3 = 1.00001 X4 = 1.00001 X5 =
1.00001 X6 = 1.00001 X7 = 1.00001
****************************************************************
  End date: Птн Жовт 21 09:08:05 EEST 2023
```

11. Перевірити працездатність послідовної програми за умови різних розмірностей СЛАР, та навести результати у табл. 4.5.

Таблиця 4.5

Результати виконання завдання

Розмірність СЛАР	Час виконання, с
100	
130	
150	
170	
190	

Розв'язання СЛАР методом сполучених градієнтів

Обирається початкове наближення: $x_0, r_0 = f - Ax_0;$ $z_0 = r_0$.
Розраховується коефіцієнт α_k для на k-ого ітераційного циклу:

$$\alpha_k = (r_{k-1}, r_{k-1}) / (Az_{k-1}, z_{k-1}).$$

Розраховується вектор рішень на k-й ітерації: $x_k = x_{k-1} + \alpha_k z_{k-1}$.

Розраховується вектор нев'язки на k-й ітерації: $r_k = r_{k-1} - \alpha_k A z_{k-1}$.

Розраховується коефіцієнт: $\beta_k = (r_k, r_k) / (r_{k-1}, r_{k-1})$.

Розраховується вектор спуску (пов'язаний напрям) на k-й ітерації:

$$z_k = r_k + \beta_k z_{k-1}.$$

Закінчення ітераційного процесу виконується за умови:

$$\frac{\|r_k\|}{\|f\|} < \varepsilon. \qquad (4.2)$$

12. Програмна реалізація послідовного алгоритму методу сполучених градієнтів розв'язання СЛАР. Задля цього треба створити в редакторі *nano* текстовий файл *cgradients.cpp*:

nano cgradients.cpp.

Увести в нього текст програми, що наведений далі.

```
#include<iostream>
#include<ctime>
#include<sys/time.h>
using namespace std;

int main(int argc, char**argv) {
   int  i, j;
   const int M = 1000;
   const double E = 0.00001;
   struct timeval tv1, tv2;
   long int dt1;
   double A[M][M], F[M], Xk[M], Zk[M];
   double Rk[M], Sz[M], alf, bet, mf;
   double Spr, Spr1, Spz;
/* Генерація даних. Задається матриця з елементами, що дорівнюють 1, а по діагоналі дорівнюють 2. Матриця повинна бути добре обумовленою та симетричною. При правій частині, рівній M + 1, всі коріння будуть дорівнювати 1. */
  for(mf = 0, i = 0; i < M; i++) {
       for(j = 0; j < M; j++)
             if(i == j)
           A[i][j] = 2.0;
         else
           A[i][j] = 1.0;
       F[i] = M + 1;
/*Обчислюється сума квадратів елементів вектора F, тобто підкорений вираз. */
       mf += F[i] * F[i];
    }
/*Задається начальне приближення значень коренів. У Xk зберігаються значення коренів к-ї ітерації. */
   for(i = 0; i < M; ++i)
      Xk[i] = 0.2;
/* Задається начальне приближення значень r0 та z0. */
   for(i = 0; i < M; i++) {
```

```
        for(Sz[i] = 0, j = 0; j < M;  ++j)
           Sz[i] += A[i][j] * Xk[j];
        Rk[i] = F[i] - Sz[i];
        Zk[i] = Rk[i];
    }
/* Початковий час проведення обчислень. */
    gettimeofday(&tv1, NULL);
    do {
/* Обчислюються чисельник і знаменник для коефіцієнта αk = (rk-1, rk-1)/
/(Azk-1, zk-1). */
        Spz = 0;
        Spr = 0;
        for(i = 0; i < M; i++) {

              for(Sz[i]=0, j = 0; j < M; j++)
              Sz[i] += A[i][j] * Zk[j];
              Spz += Sz[i] * Zk[i];
              Spr += Rk[i] * Rk[i];
        }
        alf = Spr/Spz;  /* αk */
/* Обчислюється вектор розв'язку: xk = xk-1+ αkzk-1, вектор нев'язки: rk = rk-1 –
– αkAzk-1, та чисельник для коефіцієнта βk = (rk,rk). */
        Spr1 = 0;
        for(i = 0; i < M; i++) {
              Xk[i] += alf * Zk[i];
              Rk[i] -= alf * Sz[i];
              Spr1 += Rk[i] * Rk[i];
        }
       bet = Spr1 / Spr; // Обчислюється коефіцієнт βk
/* Обчислюється вектор спуску: zk = rk+ βkzk-1. */
        for(i = 0; i < M; i++)
             Zk[i] = Rk[i] + bet * Zk[i];
/* Перевіряється умова виходу з ітераційного циклу. */
    } while(Spr1 / mf > E * E);
/* Час завершення виконання завдання виводиться на екран. */
   gettimeofday(&tv2, (struct timezone*)0);
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
```

```
    cout << "*****************************************************************" << endl;
    cout << "\t\tTime = " << dt1 << "\tSoLE dimension = " << M << endl;
/* Для контролю результатів виводяться 8 перших коренів. */

    for(i = 0; i < 8; i++)
         cout << " Xk" << i << " = " << Xk[i];
    cout << endl;
    cout << "*****************************************************************" << endl;
    return 0;
  }
```

Зберегти зміни і вийти з редактора та відкомпілювати програму:

g++ –O3 cgradients.cpp –o cgradients.bin.

13. Програма генератор виконується в одному потоці. Тому для неї достатньо значення *ppn = 1*. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano pbs_script cgradients.pbs*:

nano cgradients.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab4/
#PBS –l nodes=1:ppn=1,walltime=00:30:00
#PBS –N cgradients

echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
nohup ./cgradients.bin
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання *pbs_script*:

qsub cgradients.pbs.

14. Перевірити результати роботи програми можна, переглянувши вміст файла *cgradients.o<job_id>*, перевірити помилки, які виникли під час роботи програми, можна в файлі *cgradients.e<job_id>*:

cat cgradients.o<job_id>

cat cgradients.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//30023.cluster
Nodes for computing:
node2
Start date: Птн Жовт 21 08:53:02 EEST 2023
********************************************************************************
            SoLE dimension = 1000   Time = 4365
 Xk0 = 1 Xk1 = 1 Xk2 = 1 Xk3 = 1 Xk4 = 1 Xk5 = 1 Xk6 = 1 Xk7 = 1
********************************************************************************
 End date: Птн Жовт 21 08:53:02 EEST 2023
```

15. Перевірити працездатність послідовної програми за умови різних розмірностей СЛАР, та навести результати у табл. 4.6.

Таблиця 4.6

Результати виконання завдання

Розмірність СЛАР	Час виконання, с
500	
700	
800	
900	
1 000	

16. Виконати індивідуальні завдання згідно з варіантами табл. 4.7, результати занести у таблицю, аналогічну до табл. 4.4.

Таблиця 4.7

Варіанти індивідуальних завдань

№ варіанта	Завдання
1	Розробити програму реалізації алгоритму розв'язання СЛАР методом простої ітерації з використанням відкритого стандарту *OpenMP*
2	Розробити програму реалізації алгоритму розв'язання СЛАР методом сполучених градієнтів з використанням відкритого стандарту *OpenMP*

Контрольні запитання до лабораторної роботи 4

1. Що таке відкритий стандарт *OpenMP*?
2. Поясніть роботу моделі розгалуження – об'єднання, та поняття "потік".
3. Поясніть призначення директив відкритого стандарту *OpenMP*, що зустрічаються в програмах.
4. Поясніть призначення функцій відкритого стандарту *OpenMP*, що зустрічаються в програмах.
5. Що зветься методом ітеративного паралелізму?
6. Що зветься методом кінцевого паралелізму?
7. Поясніть алгоритм рішення СЛАР методом Гауса та роботу програми реалізації.
8. Поясніть алгоритм рішення СЛАР методом простих ітерацій та роботу програми реалізації.
9. Поясніть алгоритм рішення СЛАР методом сполучених градієнтів та роботу програми реалізації.
10. Поясніть призначення ключів, що використовуються в директивах відкритого стандарту *OpenMP*.

Лабораторна робота 5
Паралельне програмування в стандарті MPI

Мета лабораторної роботи:

Дослідження принципів роботи стандарту MPI.

Набуття практичних навичок під час створення паралельних програм в стандарті *MPI*.

Завдання:

1. Дослідити програми, що використовують у процесі роботи стандарт MPI, виконати їх на кластері в пакетному режимі (ПР) та проаналізувати результати роботи.
2. Дослідити роботу програми розв'язання СЛАР методом Гауса з використанням стандарту MPI. Виконати на кластері в ПР та проаналізувати результат роботи.

Загальні відомості про принцип роботи стандарту MPI

Методи розпаралелювання і моделі програм, які підтримуються MPI

Message Passing Interface (*MPI*, інтерфейс передавання повідомлень) – це програмний интерфейс передавання інформації, що дозволяє обмін повідомленнями між процесами однієї задачі. *MPI* – найбільш поширений кросплатформенний стандарт для написання паралельних програм. У стандарті *MPI* описано інтерфейс передавання повідомлень, який повинен підтримуватися як платформою, так і в додатку користувача.

Існує велика кількість вільних та комерційних реалізацій *MPI*. Є реалізації для C, C++, Фортран*, Java*. *MPI* орієнтований на системи з розподіленою пам'яттю, коли затрати на передавання даних великі, у той час як *OpenMP* орієнтований на системи з загальною пам'яттю. Обидва стандарти можуть використовуватися разом для оптимального використання обчислювальних ресурсів.

Основою механізму зв'язку між процесами *MPI* є передавання та приймання повідомлень. Вони несуть дані та інформацію, що дозволяють приймаючій стороні здійснювати їхнє вибіркове приймання:

- відправник – ранг (номер в групі) відправника повідомлення;
- отримувач – ранг отримувача;
- ознака – може використовуватися для розділу різних видів повідомлень;
- комунікатор – код групи процесів. Іншим способом зв'язку є віддалений доступ до пам'яті (*RMA*), що дозволяє читати та змінювати область пам'яті віддаленого процесу.

Локальний процес може переносити область пам'яті віддаленого процесу в свою пам'ять та повертати її, комбінувати дані, передаючи віддаленому процесу в його пам'ять дані (шляхом сумування). Важливою особливістю інтерфейсу є можливість написання паралельних програм без урахування архітектурних особливостей конкретних мультикомп'ютерів.

MPI надає користувачеві віртуальний мультикомп'ютер з розподіленою пам'яттю і з віртуальною мережею зв'язку між віртуальними комп'ютерами. Користувач визначає кількість комп'ютерів, необхідних для вирішення його завдання, а також топологію зв'язків між ними. *MPI* реалізує цю вимогу на конкретній обчислювальній системі. Обмеженням при цьому є обсяг оперативної пам'яті фізичного мультикомп'ютера. Таким чином, користувач працює у віртуальному середовищі, що забезпечує переносимість його паралельних програм.

Однією з цілей, що постають у процесі вирішення задач на обчислювальних системах, у тому числі і на паралельних, є збільшення ефективності. Ефективність паралельної програми істотно залежить від співвідношення часу обчислень та часу на обмін даними.

Чим менше частка часу, витраченого на передавання повідомлень в загальному часі обчислень, тим більше ефективність програми. Для паралельних систем з передаванням повідомлень оптимальне співвідношення між обчисленнями і комунікаціями забезпечує метод крупнозернистого розпаралелювання, коли паралельні алгоритми будуються з великих блоків, які рідко взаємодіють.

Розробник повинен мати уяву про паралельні програми, що налаштовані на розмір обчислювальної системи як на параметр. Варто засвоїти функції парних і колективних взаємодій між процесами паралельної

програми, набути навичок у розробленні програм для проведення обчислень на кластері під керуванням ОС *Linux*.

Структура програми MPI

Існує декілька базових функцій, які використовуються в програмі з *MPI*:

1. Ініціалізація бібліотек. Першою функцією у програмі повинна бути:

MPI_Init (& argc, & argv).

Вона створює групу процесів додатка та область зв'язку, що описана зумовленим комунікатором *MPI_COMM_WORLD*. У цій області об'єднані всі процеси, які впорядковані та пронумеровані від 0 до SIZE – 1, де SIZE це число процесів у групі. Крім того, створюється зумовлений комунікатор *MPI_COMM_SELF*, що описує свою область зв'язку для кожного окремого процесу. *MPI_Init* отримує адреси аргументів, що одержує програма під час виклику в якості ключів від користувача та операційної системи. У кінець командного рядка програми *MPI*-завантажувач (команда *mpirun*) додає ряд інформаційних параметрів, які потрібні *MPI_Init*.

2. Нормальне закриття бібліотек. Останньою функцією у програмі повинна бути *MPI_Finalize*(). Вона закриває всі процеси *MPI* і ліквідує всі області зв'язку. Ця функція використовується наприкінці програми: перед кожним викликом стандартної функції *exit* у мові C++; перед кожним оператором *return* (якщо це неповернення значення функції) в мові C++; у мові Сі, якщо функції *main* призначений тип *void* і вона не закінчується оператором *return* (тоді *MPI_ Finalize*() слід поставити в кінець функції *main*).

3. Інформація області зв'язку. Це дві функції, що визначають загальне число процесів в області зв'язку і порядковий номер процесу, що викликає цю функцію:

```
int size, rank;
MPI_Comm_size ( MPI_COMM_WORLD, &size );
MPI_Comm_rank ( MPI_COMM_WORLD, &rank ).
```

4. Аварійне закриття бібліотеки. Викликається, якщо для користувача програма завершується через помилки часу виконання, пов'язані з *MPI: MPI_Abort* (описувач галузі зв'язку, код помилки *MPI*). Виклик *MPI_Abort* з будь-якого завдання примусово завершує роботу всіх завдань,

приєднаних до заданої області зв'язку. Якщо вказано комунікатор *MPI_COMM_WORLD*, то буде завершено всі процеси, що є в області зв'язку. Використовуйте код помилки *MPI_ERR_OTHER*, якщо невідомо, як охарактеризувати помилку в класифікації *MPI*.

Порядок виконання лабораторної роботи

1. Для входу на обчислювальний кластер потрібно виконати такі дії залежно від ОС, яку використовують:

• **для *Windows* 10** – увійти в *Windows PowerShell* та виконати команду: ssh.exe ім'я_користувача@IP_адреса_кластера –p порт_ssh_ з'єднання;

• **для *Linux*** – увійти в термінал та виконати команду: ssh ім'я_ користувача @IP_адреса_кластера: порт_ssh_з'єднання.

З'явиться запрошення для введення паролю *Password*. У цьому рядку треба ввести default_пароль.

Ім'я_користувача, default_пароль, IP_адресу_кластера та порт_ssh _з'єднання отримати у викладача. Default_пароль можна змінити на індивідуальний. Якщо пароль втрачено, то треба відновити його за допомогою адміністратора кластера.

2. Створити директорію та увійти в неї:

mkdir lab5
cd lab5.

Приклад 1.

1. Створити в редакторі *nano* текстовий файл example1.cpp:

nano example1.cpp.

Увести в нього текст програми, що наведений далі.

```
#include<iostream>
#include<mpi.h>         // підключення MPI-бібліотеки
using namespace std;

int main(int argc, char **argv) { // заголовок головної функції. Параметри вказувати обов'язково)
   int size, rank, namelen;
   char processor[MPI_MAX_PROCESSOR_NAME];
```

```
   MPI_Init(&argc, &argv); //функція ініціалізації MPI процесів
   MPI_Comm_size(MPI_COMM_WORLD, &size); // записує в змінну size загальну кількість процесів
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // записує в змінну rank номер поточного процесу
   MPI_Get_processor_name(processor, &namelen); // унікальний описник фактичного вузла
   cout << " Number of processes = " << size << " Process number = " << rank << " Processor: " << processor << endl;
   MPI_Finalize();  // функція завершення процесів MPI
   return 0;
 }
```

Щоб зберегти файл, натисніть клавіші Ctrl + O. Задати нове ім'я чи погодитись з запропонованим. Для завершення роботи редактора натисніть клавіші Ctrl + X. Відкомпілювати програму:

mpic++ example1.cpp –o example1.bin.

2. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі nano pbs_script example1.pbs:

nano example1.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/student<number>/lab5/
#PBS –l nodes=3:ppn=2,walltime=00:30:00
#PBS –N example1

echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
OMPI_MCA_shmem_mmap_enable_nfs_warning=0
export OMPI_MCA_shmem_mmap_enable_nfs_warning
mpirun –hostfile mpi.hosts ./example1.bin
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Для виключення повідомлення про створення тимчасового файла пам'яті в директорії підключеній NFS, встановити змінну середи OMPI_MCA_shmem_mmap_enable_nfs _warning в 0.

Зверніть увагу, що запуск на виконання програм у стандарті *MPI* проводиться командою *mpirun*. Якщо виконати програму без *mpirun*, то вона виконається в одному процесі з номером 0. Ключ –hostfile визначає кількість доступних вузлів для виконання програми. Перелік вузлів вказується в файлі mpi.hosts, по одній назві вузла в рядку:

```
[student@cluster lab5]$ cat mpi.hosts
cluster
node1
node2.
```

Відправити завдання в чергу, запустивши на виконання pbs_script:

qsub example1.pbs.

3. Перевірити результати роботи програми можна, переглянувши вміст файла example1.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example1.e<job_id>:

cat example1.o<job_id>

cat example1.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//30053.cluster
Nodes for computing:
node2
node2
node1
node1
cluster
cluster
Start date: Пнд Жовт 24 04:05:58 EEST 2023
 Number of processes = 6 Process number = 4 Processor: node2
 Number of processes = 6 Process number = 5 Processor: node2
```

```
Number of processes = 6 Process number = 0 Processor: cluster
Number of processes = 6 Process number = 1 Processor: cluster
Number of processes = 6 Process number = 3 Processor: node1
Number of processes = 6 Process number = 2 Processor: node1
 End date: Пнд Жовт 24 04:06:00 EEST 2023
```

Кожен процес програми виводить на екран загальне число процесів в області зв'язку, номер безпосередньо процесу, який виконується, та унікальний описник фактичного вузла, в який завантажується процес.

Приклад 2.

1. Створити програму, де процес з номером 0 створює масив з десяти цифр. Виводить на екран свій номер та значення масиву.

Пересилає масив процесу з номером 3. Процес з номером 3 виводить на екран свій номер і прийнятий від процесу з номером 0 масив. Створити в редакторі *nano* текстовий файл example2.cpp:

nano example2.cpp.

Увести в нього текст програми, що наведений далі.

```
#include<iostream>
#include<cstdlib>
#include<mpi.h>
using namespace std;

int main(int argc, char **argv) {
   int size, rank, i = 0, a[10], b[10];
   MPI_Status st;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if(rank == 0) {
      while(i < 10) {a[i] = rand() % 100, ++i;}
      cout <<" Process number  = " << rank << " a:";
```

```
        for(i = 0; i < 10; ++i) cout << " " << a[i];
        cout << endl;
/* Передавання повідомлень з блокуванням:
          &a – адреса комірки пам'яті початку буфера передавання повідомлення;
          10 – кількість елементів вказаного типу, що передаються;
          MPI_INT – вказує на тип переданих елементів;
           3  – номер процесу отримувача;
          15 – ідентифікатор повідомлення;
MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси. */
   MPI_Send(&a,10,MPI_INT,3,15,MPI_COMM_WORLD);
     }
     else {
         if(rank == 3) {
/* Приймання повідомлення з блокуванням:
          &b – адреса комірки пам'яті початку буфера приймання повідомлення;
           10 – кількість прийнятих елементів указаного типу;
           MPI_INT – вказує на тип прийнятих елементів;
           0  – номер процесу, що передається;
          15 – ідентифікатор повідомлення;
  MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси;
          &st – статус завершення приймання. */
          MPI_Recv(&b,10,MPI_INT,0,15,MPI_COMM_WORLD,&st);
          cout <<" Process number  = " << rank << " b:";
          for(i = 0; i < 10; ++i) cout << " " << b[i];
          cout << endl;
        }
  MPI_Finalize();
 return 0;
}
```

Зберегти зміни і вийти з редактора. Відкомпілювати програму командою:

```
mpic++ example2.cpp –o example2.bin.
```

2. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі nano pbs_script example2.pbs:

nano example2.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/ student<number>/lab5/
#PBS –l nodes=3:ppn=2,walltime=00:30:00
#PBS –N example2
echo "File containing nodes:"
echo $PBS_NODEFILE

echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
OMPI_MCA_shmem_mmap_enable_nfs_warning=0
export OMPI_MCA_shmem_mmap_enable_nfs_warning
mpirun –hostfile mpi.hosts ./example2.bin
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання pbs_script:

qsub example2.pbs.

3. Перевірити результати роботи програми можна переглянувши вміст файла example2.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example2.e<job_id>:

cat example2.o<job_id>

cat example2.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//30057.cluster
```

```
Nodes for computing:
node2
node2
node1
node1
cluster
cluster
Start date: Пнд Жовт 24 11:00:20 EEST 2023
 Process number  = 0 a: 90 59 63 26 40 26 72 36 11 68
 Process number  = 3 b: 90 59 63 26 40 26 72 36 11 68
  End date: Пнд Жовт 24 11:00:22 EEST 2023
```

Приклад 3.

1. Створити програму, де процес з номером 0 створює масив з десяти цифр. Виводить на екран свій номер та значення масиву. Пересилає масив процесу з останнім номером у множині запущених процесів. Процес з останнім номером виводить на екран свій номер і прийнятий від процесу з номером 0 масив.

Для виконання треба створити в редакторі *nano* текстовий файл example3.cpp:

nano example3.cpp.

Увести в нього текст програми, що наведено далі.

```
#include<iostream>
#include<cstdlib>
#include<mpi.h>
using namespace std;

int main(int argc, char **argv) {
   int size, rank, i = 0, a[10], b[10];
   MPI_Status st;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if(rank == 0) {
```

```
        while(i < 10) {a[i] = rand() % 100, ++i;}
        cout <<" Process number  = " << rank << " a:";
        for(i = 0; i < 10; ++i) cout << " " << a[i];
        cout << endl;
/* Передавання повідомлень з блокуванням:
          &a – адреса комірки пам'яті початку буфера передавання повідомлення;
          10  – кількість елементів вказаного типу, які передають;
          MPI_INT – вказує на тип переданих елементів;
          size-1 – номер процесу отримувача;
          15 – ідентифікатор повідомлення;
MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси. */
          MPI_Send(&a,10,MPI_INT,size-1,15,MPI_COMM_WORLD);
      }
      else {
          if(rank == size-1) {
/* Приймання повідомлення з блокуванням:
          &b – адреса комірки пам'яті початку буфера приймання повідомлення;
           10 – кількість прийнятих елементів вказаного типу;
           MPI_INT – вказує на тип прийнятих елементів;
           0  – номер процесу передавання;
          15 – ідентифікатор повідомлення;
  MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси;
          &st – статус завершення приймання. */
          MPI_Recv(&b,10,MPI_INT,0,15,MPI_COMM_WORLD,&st);
          cout <<" Process number  = " << rank << " b:";
          for(i = 0; i < 10; ++i) cout << " " << b[i];
          cout << endl;
         }
  MPI_Finalize();
 return 0;
}
```

Зберегти зміни і вийти з редактора. Відкомпілювати програму:

mpic++ example3.cpp –o example3.bin.

Звернути увагу на умову задачі, що останній номер гілки в безлічі запущених гілок дорівнює *size* – 1. Це простий приклад паралельної програми, яка налаштовується на розмір обчислювальної системи, як на параметр. Таку програму можна без переробок і без перекомпіляції запускати на будь-якій кількості комп'ютерів і вона буде видавати правильний результат.

Основними параметрами в паралельній програмі є:

- розмір обчислювальної системи – *size*;
- номер гілки – *rank*.

У цьому випадку цих параметрів досить, щоб програма автоматично визначала розмір системи.

2. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі nano pbs_script example3.pbs:

nano example3.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/ student<number>/lab5/
#PBS –l nodes=3:ppn=2,walltime=00:30:00
#PBS –N example3

echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
OMPI_MCA_shmem_mmap_enable_nfs_warning=0
export OMPI_MCA_shmem_mmap_enable_nfs_warning
mpirun –hostfile mpi.hosts ./example3.bin
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання pbs_script:

qsub example3.pbs.

3. Перевірити результати роботи програми можна, переглянувши вміст файла example3.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example3.e<job_id>:

cat example3.o<job_id>

cat example3.e<job_id>.

Результат виконання програми наведено далі.

```
File containing nodes:
/var/spool/torque/aux//30058.cluster
Nodes for computing:
node2
node2
node1
node1
cluster
cluster
Start date: Пнд Жовт 24 12:12:55 EEST 2023
 Process number  = 0 a: 90 59 63 26 40 26 72 36 11 68
 Process number  = 5 b: 90 59 63 26 40 26 72 36 11 68
 End date: Пнд Жовт 24 12:12:57 EEST 2023
```

Приклад 4.

1. Створити програму, де процес з номером 0 створює масив з десяти цифр. Пересилає масив усім процесам з множини запущених. Процеси приймають масив. Усі процеси виводять на екран свій номер, унікальний описник фактичного вузла і 0 масив.

Створити в редакторі *nano* текстовий файл example4.cpp:

nano example4.cpp.

Увести в нього текст програми, що наведений далі.

```
#include<iostream>
#include<cstdlib>
#include<mpi.h>
using namespace std;
```

```
int main(int argc, char **argv) {
   int size, rank, namelen, i = 0, a[10];
   char processor[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(processor, &namelen);
   if(rank == 0)
     while(i < 10) {a[i] = rand() % 100, ++i;}
```

/* Розсилка повідомлення від процесу з рангом *root* усім процесам комунікатора:

&a – адреса комірки пам'яті буфера та початку передавання повідомлення;

10 – кількість переданих елементів указаного типу;

MPI_INT – вказує на тип переданих елементів;

0 – номер процесу з рангом *root;*

MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси. */

```
   MPI_Bcast(&a,10,MPI_INT,0,MPI_COMM_WORLD);
   cout <<" Process number  = " << rank << " Processor: " << processor << " a:";
   for(i = 0; i < 10; ++i) cout << " " << a[i];
   cout << endl;

   MPI_Finalize();
  return 0;
 }
```

Зберегти зміни і вийти з редактора. Відкомпілювати програму:

mpic++ example4.cpp –o example4.bin.

2. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano* pbs_script example4.pbs:

nano example4.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/ student<number>/lab5/
#PBS –l nodes=3:ppn=2,walltime=00:30:00
#PBS –N example4
```

```
echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
OMPI_MCA_shmem_mmap_enable_nfs_warning=0
export OMPI_MCA_shmem_mmap_enable_nfs_warning
mpirun –hostfile mpi.hosts ./example4.bin
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання pbs_script:

qsub example4.pbs.

3. Перевірити результати роботи програми можна, переглянувши вміст файла example4.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example4.e<job_id>:

cat example4.o<job_id>

cat example4.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//30063.cluster
Nodes for computing:
node2
node2
node1
node1
cluster
cluster
Start date: Пнд Жовт 24 15:16:19 EEST 2023
```

```
Process number  = 4 Processor: node2 a: 90 59 63 26 40 26 72 36 11 68
 Process number  = 5 Processor: node2 a: 90 59 63 26 40 26 72 36 11 68
 Process number  = 0 Processor: cluster a: 90 59 63 26 40 26 72 36 11 68
 Process number  = 1 Processor: cluster a: 90 59 63 26 40 26 72 36 11 68
 Process number  = 3 Processor: node1 a: 90 59 63 26 40 26 72 36 11 68
 Process number  = 2 Processor: node1 a: 90 59 63 26 40 26 72 36 11 68
 End date: Пнд Жовт 24 15:16:21 EEST 2023
```

Приклад 5.

1. Створити програму, що знаходить скалярний добуток двох векторів. Скалярний добуток двох векторів є сумою добутків пар значень кожного вектора. Обчислення кожної пари не залежить від інших пар. Отже, це можна робити паралельно в різних процесах і прискорити роботу програми. Кожен процес може підсумовувати свою частину добутків, отримати суму результатів кожного процесу.

Створити в редакторі *nano* текстовий файл example5.cpp:

nano example5.cpp.

Увести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <cstdlib>
#include <mpi.h>
using namespace std;

int main(int argc, char *argv[]) {
   int total, div, size, rank, i, j;
   double sum, rem, startwtime, result;

   if(argc != 2) {
      cout << " Usage: " << argv[0] << " <length of vector>" << endl,
exit(EXIT_FAILURE);
   }
   total = atoi(argv[1]);
   double *a = new double [total];
```

```
   double *b = new double [total];
   if((a == NULL) || (b == NULL)) {
      cout << "Error allocating vectors (not enough memory?)" << endl, exit(1);
   }
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if(rank == 0) {
     for(i = 0; i < total; i++) {
        a[i] = drand48();
        b[i] = drand48();
      }
      div = (total - total % size) / size;
      startwtime = MPI_Wtime();
   }
   MPI_Bcast(a, total, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b, total, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(&div, 1, MPI_INT, 0, MPI_COMM_WORLD);
   sum = 0.0;
   for(i = rank * div; i < rank * div + div; ++i)
        sum += a[i] * b[i];
   cout << "Rank: " << rank << " Sum: " << sum << endl;
/* Об'єднує елементи вхідного буфера кожного процесу в групі, згідно
зі вказаною операцією, повертає підсумкове значення у вказаний процес
root:
&sum – адреса комірки пам'яті початку буфера передавання
повідомлення;
&result – адреса комірки пам'яті початку буфера приймання
повідомлення;
10 – кількість переданих елементів указаного типу;
MPI_INT – вказує на тип переданих елементів;
MPI_SUM – операція редукції;
0 – номер процесу з рангом root, якому повертають результат;
MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси. */
   MPI_Reduce(&sum, &result, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

```
    if(rank == 0) {
       rem = 0.0;
       for(i = total - total % size; i < total; ++i)
          rem += a[i] * b[i];
       cout << " Answer: " << result + rem << " wall clock time = " <<
MPI_Wtime() - startwtime << endl;
    }
    delete []a;
    delete []b;
   MPI_Finalize();
 return 0;
}
```

Зберегти зміни і вийти з редактора. Відкомпілювати програму:

mpic++ example5.cpp –o example5.bin.

2. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano* pbs_script example5.pbs:

nano example5.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/ student<number>/lab5/
#PBS -l nodes=3:ppn=2,walltime=00:30:00
#PBS -N example5
echo "File containing nodes:"
echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
OMPI_MCA_shmem_mmap_enable_nfs_warning=0
export OMPI_MCA_shmem_mmap_enable_nfs_warning
```

mpirun -hostfile mpi.hosts ./example5.bin 5000
echo " End date: `date`"
Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання pbs_script:

qsub example5.pbs.

3. Перевірити результати роботи програми можна, переглянувши вміст файла example5.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example5.e<job_id>:

cat example5.o<job_id>

cat example5.e<job_id>.

Результат виконання програми наведений далі.

File containing nodes:
/var/spool/torque/aux//30067.cluster
Nodes for computing:
node2
node2
node1
node1
cluster
cluster
Start date: Втр Жовт 25 20:25:47 EEST 2023
Rank: 0 Sum: 204.795
Rank: 5 Sum: 216.784
Rank: 4 Sum: 208.661
Rank: 1 Sum: 204.962
 Answer: 1247.4 wall clock time = 0.0277641
Rank: 3 Sum: 203.64
Rank: 2 Sum: 207.974
  End date: Втр Жовт 25 20:25:49 EEST 2023

Приклад 6.

1. Створити програму, що знаходить добуток двох матриць розмірністю не менше числа процесів в області зв'язку. Добуток двох матриць є сумою добутків відповідних рядків та стовпців. Обчислення кожного окремого значення не залежить від інших. Отже, це можна робити паралельно в різних процесах і прискорити роботу програми. Кожен процес може підсумовувати свою частину добутків, і отримати суму результатів кожного процесу. Створити в редакторі *nano* текстовий файл example6.cpp:

nano example6.cpp.

Увести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <mpi.h>
using namespace std;

int main(int argc, char **argv) {
   int size, rank, i, j, k;
   double startwtime;

   if(argc != 2) {
     cout << " Usage: " << *argv << " <matrix dimenssion>" << endl,
exit(EXIT_FAILURE);
   }
   const int dimension = atoi(*++argv);
   int *a = new int[dimension * dimension];
   int *b = new int[dimension * dimension];

   int *c = new int[dimension * dimension];
   int *la = new int[dimension * dimension];
   int *lc = new int[dimension * dimension];

   MPI_Status status;
   MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0) {
      if(dimension < size) {
        cout << " !!! ERROR: Matrix dimension " << dimension << " < the size
of the group " << size << ". !!! ERROR" << endl, exit(EXIT_FAILURE);
      }
      for(i = 0; i < dimension; ++i)
         for(j = 0; j < dimension; ++j) {
            a[i * dimension + j] = rand() % 10;
            b[i * dimension + j] = rand() % 10;
         }
       cout << "*******!0!*******" << endl;
       for(i = 0; i < dimension; ++i) {
          cout << "rank: " << rank << " a:";
          for(j = 0; j < dimension; j++)
             cout << " " << a[i * dimension + j];
          cout << endl;

       }
       cout << "*******!0!*******" << endl;
       for(i = 0; i < dimension; i++) {
          cout << "rank: " << rank << "  b:";
          for(j = 0; j < dimension; j++)
             cout << " " << b[i * dimension + j];
          cout << endl;
       }
       cout << "*******!0!*******" << endl;
       startwtime = MPI_Wtime();
    }
/*Відправка даних з одного процесу всім іншим в області зв'язку:
a – адреса комірки пам'яті початку буфера передавання повідомлення
dimension / size * dimension – кількість елементів, відправлених кожному
процесу;
MPI_INT – вказує на тип переданих елементів;
la – кількість елементів у прийомному буфері;
dimension / size * dimension – кількість елементів прийомного буфера;
```

MPI_INT – вказує на тип прийнятих елементів;
0 – номер процесу з рангом *root*;
MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси. */
MPI_Scatter(a, dimension / size * dimension, MPI_INT, la, dimension / size * dimension, MPI_INT, 0, MPI_COMM_WORLD);
/* Розсилка повідомлення від процесу з рангом *root* усім процесам комунікатора:
b – адреса комірки пам'яті початку буфера передавання повідомлення
dimension * dimension – кількість переданих елементів указаного типу;
MPI_INT – вказує на тип переданих елементів;
0 – номер процесу з рангом *root*;
MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси. */
MPI_Bcast(b, dimension * dimension, MPI_INT, 0, MPI_COMM_WORLD);

```
for(i = 0; i < dimension / size; i++) {
    cout << "rank: " << rank << " la:";
    for(j = 0; j < dimension; j++)
        cout << " " << la[i * dimension + j];
    cout << endl;
  }
for(i = 0; i < dimension; i++) {
    cout << "rank: " << rank << "  b:";
    for(j = 0; j < dimension; j++)
        cout << " " << b[i * dimension + j];
    cout << endl;
}
for(i = 0; i < dimension / size; i++)
    for(j = 0; j < dimension; j++){
        lc[i * dimension + j] = 0;
        for(k = 0; k < dimension; k++)
            lc[i * dimension + j] += la[i * dimension + k] * b[k * dimension + j];
    }
for(i = 0; i < dimension / size; i++) {
    cout << "rank: " << rank << " lc:";
```

```
        for(j = 0; j < dimension; j++)
            cout << " " << lc[i * dimension + j];
        cout << endl;
      }
/*Збирає дані в один процес з усіх інших в області зв'язку:
lc – адреса комірки пам'яті початку буфера передавання повідомлення
dimension / size * dimension – кількість елементів, відправлених кожним процесом;
MPI_INT – вказує на тип відправлених елементів;
c – кількість елементів у прийомному буфері;
dimension / size * dimension – кількість елементів прийомного буфера;
MPI_INT – вказує на тип прийнятих елементів;
0 – номер процесу з рангом root;
MPI_COMM_WORLD – глобальний комунікатор, що містить всі процеси. */
       MPI_Gather(lc, dimension / size * dimension, MPI_INT, c, dimension /
size * dimension, MPI_INT, 0, MPI_COMM_WORLD);

      if(dimension % size)
        for(i = dimension - dimension % size - 1; i < dimension; i++)
          for(j = 0; j < dimension; j++){
            c[i * dimension + j] = 0;
            for(k = 0; k < dimension; k++)
              c[i * dimension + j] += a[i * dimension + k] * b[k * dimension + j];
          }

   if(rank == 0) {
     cout << "******!Answer!******" << endl;
     for(i = 0; i < dimension; i++) {
        cout << "rank: " << rank << "  c:";
        for(j = 0; j < dimension; j++)
          cout << " " << setw(4) << c[i * dimension + j];
        cout << endl;
        }
      cout << " RunTime: " << MPI_Wtime() - startwtime << endl;
      cout << "******!Answer!******" << endl;
    }
```

```
    delete[] a;
    delete[] b;
    delete[] c;
    delete[] la;
    delete[] lc;
  MPI_Finalize();
 return 0;
}
```

Зберегти зміни і вийти з редактора та відкомпілювати програму:

mpic++ example6.cpp –o example5.bin.

2. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano* pbs_script example6.pbs:

nano example6.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/ student<number>/lab5/
#PBS -l nodes=3:ppn=2,walltime=00:30:00
#PBS -N example6
echo "File containing nodes:"

echo $PBS_NODEFILE
echo "Nodes for computing:"
cat $PBS_NODEFILE

echo "Start date: `date`"
OMPI_MCA_shmem_mmap_enable_nfs_warning=0
export OMPI_MCA_shmem_mmap_enable_nfs_warning
mpirun -hostfile mpi.hosts ./example6.bin 7
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання pbs_script:

qsub example6.pbs.

3. Перевірити результати роботи програми можна, переглянувши вміст файла example6.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example6.e<job_id>:

cat example6.o<job_id>

cat example6.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:
/var/spool/torque/aux//30077.cluster
Nodes for computing:
node2
node2
node1
node1
cluster
cluster
Start date: Чтв Жовт 27 08:46:13 EEST 2023
*******!0!*******
rank: 0 a: 0 3 0 2 1 7 2
rank: 0 a: 2 7 9 2 9 3 1
rank: 0 a: 9 1 4 8 5 3 1
rank: 0 a: 6 2 6 5 4 6 6
rank: 0 a: 3 4 2 4 4 3 7
rank: 0 a: 6 8 3 4 2 6 9
rank: 0 a: 6 4 5 4 7 7 7
*******!0!*******
rank: 0  b: 9 6 6 6 8 9 0
rank: 0  b: 3 5 2 8 7 6 2
```

rank: 0 b: 3 9 7 4 0 6 0
rank: 0 b: 3 0 1 5 7 5 9
rank: 0 b: 7 5 5 7 4 0 8
rank: 0 b: 8 4 1 9 0 8 2
rank: 0 b: 6 9 0 8 1 2 2
*******!0!*******
rank: 4 la: 3 4 2 4 4 3 7
rank: 4 b: 9 6 6 6 8 9 0
rank: 4 b: 3 5 2 8 7 6 2
rank: 4 b: 3 9 7 4 0 6 0
rank: 4 b: 3 0 1 5 7 5 9
rank: 4 b: 7 5 5 7 4 0 8
rank: 4 b: 8 4 1 9 0 8 2
rank: 4 b: 6 9 0 8 1 2 2
rank: 4 lc: 151 151 67 189 103 121 96
rank: 5 la: 6 8 3 4 2 6 9
rank: 5 b: 9 6 6 6 8 9 0
rank: 5 b: 3 5 2 8 7 6 2
rank: 5 b: 3 9 7 4 0 6 0
rank: 5 b: 3 0 1 5 7 5 9
rank: 5 b: 7 5 5 7 4 0 8
rank: 5 b: 8 4 1 9 0 8 2
rank: 5 b: 6 9 0 8 1 2 2
rank: 5 lc: 215 218 93 272 149 206 98
rank: 3 la: 6 2 6 5 4 6 6
rank: 3 b: 9 6 6 6 8 9 0
rank: 3 b: 3 5 2 8 7 6 2
rank: 3 b: 3 9 7 4 0 6 0
rank: 3 b: 3 0 1 5 7 5 9
rank: 3 b: 7 5 5 7 4 0 8
rank: 3 b: 8 4 1 9 0 8 2
rank: 3 b: 6 9 0 8 1 2 2
rank: 3 lc: 205 198 113 231 119 187 105
rank: 1 la: 2 7 9 2 9 3 1
rank: 1 b: 9 6 6 6 8 9 0
rank: 1 b: 3 5 2 8 7 6 2

rank: 1 b: 3 9 7 4 0 6 0
rank: 1 b: 3 0 1 5 7 5 9
rank: 1 b: 7 5 5 7 4 0 8
rank: 1 b: 8 4 1 9 0 8 2
rank: 1 b: 6 9 0 8 1 2 2
rank: 1 lc: 165 194 139 212 116 150 112
rank: 2 la: 9 1 4 8 5 3 1
rank: 2 b: 9 6 6 6 8 9 0
rank: 2 b: 3 5 2 8 7 6 2
rank: 2 b: 3 9 7 4 0 6 0
rank: 2 b: 3 0 1 5 7 5 9
rank: 2 b: 7 5 5 7 4 0 8
rank: 2 b: 8 4 1 9 0 8 2
rank: 2 b: 6 9 0 8 1 2 2
rank: 2 lc: 185 141 120 188 156 177 122
rank: 0 la: 0 3 0 2 1 7 2
rank: 0 b: 9 6 6 6 8 9 0
rank: 0 b: 3 5 2 8 7 6 2
rank: 0 b: 3 9 7 4 0 6 0
rank: 0 b: 3 0 1 5 7 5 9
rank: 0 b: 7 5 5 7 4 0 8
rank: 0 b: 8 4 1 9 0 8 2
rank: 0 b: 6 9 0 8 1 2 2
rank: 0 lc: 90 66 20 120 41 88 50
******!Answer!******
rank: 0 c: 90 66 20 120 41 88 50
rank: 0 c: 165 194 139 212 116 150 112
rank: 0 c: 185 141 120 188 156 177 122
rank: 0 c: 205 198 113 231 119 187 105
rank: 0 c: 151 151 67 189 103 121 96
rank: 0 c: 215 218 93 272 149 206 98
rank: 0 c: 240 227 125 276 139 198 128
 Run Time: 0.0111542
******!Answer!******
  End date: Чтв Жовт 27 08:46:15 EEST 2023

Приклад 7.

1. Створити програму, що реалізує передавання даних для двох завдань в разі, коли розмір повідомлень невідомий. Створити в редакторі *nano* текстовий файл example7.cpp:

nano example7.cpp.

Увести в нього текст програми, що наведений далі.

```
#include <iostream>
#include <mpi.h>
using namespace std;

int main( int argc, char **argv ) {
   int size, rank, count;
/* Ідентифікатори повідомлень. */
   const int tagFloatData = 1;
   const int tagDoubleData = 2;
   float floatData[10];
   double doubleData[20];
   MPI_Status status;
   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   /* Користувач повинен запустити програму на двох процесах, тобто емалювати роботу двох завдань, інакше помилка. */
   if( size != 2 ) {
      /* Завдання 0 повідомляє користувачеві про помилку. */
      if( rank == 0 )
      cout << "Error: 2 processes required instead of " << size << ", abort" << endl;
/* Усі завдання-абоненти області зв'язку MPI_COMM_WORLD будуть чекати, поки завдання 0 не виведе повідомлення. */
      MPI_Barrier( MPI_COMM_WORLD );
/* Без точки синхронізації може статися, що одне з завдань викличе MPI_Abort раніше, ніж встигне закінчити роботу printf() в завданні 0, тоді
```

MPI_Abort негайно примусово завершить всі завдання і повідомлення виведено не буде. Усі завдання аварійно завершать роботу;
MPI_COMM_WORLD – описувач області зв'язку, на яку поширюється дія помилки;
MPI_ERR_OTHER – код помилки. */
```
    MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER );
    return –  1;
    }
    if( rank == 0 ) {
```
/* Завдання 0 передає дані завданню 1:
    floatData – адреса масива, який пересилають;
    5 – кількість комірок: floatData[0]..floatData[4];
    MPI_FLOAT – тип даних, які пересилають;
    1 – номер завдання-отримувача;
    tagFloatData – ідентифікатор повідомлення;
    MPI_COMM_WORLD – описувач області зв'язку, через який відбувається одна або інше передавання (дані іншого типу). */
```
 MPI_Send(floatData, 5, MPI_FLOAT, 1, tagFloatData,
MPI_COMM_WORLD);
 MPI_Send( doubleData, 6, MPI_DOUBLE, 1, tagDoubleData;
MPI_COMM_WORLD);
    }
     else {
```
 /* Завдання 1 приймає дані від завдання 0, очікуючи повідомлення і розміщує прийняті дані в буфері:
doubleData – адреса масива, куди складаються прийняті дані;
(sizeof(doubleData) / sizeof(doubleData[0])) – фактична довжина приймального масива в кількості комірок;
MPI_DOUBLE – повідомлення *MPI*, про тип отриманого повідомлення;
  0 – номер процесу відправника;
  tagDoubleData – очікування повідомлення з таким ідентифікатором;
 MPI_COMM_WORLD – описувач області зв'язку, через яку очікується повідомлення;
&status – сюди записується статус завершення операції приймання даних. */
```
 MPI_Recv(doubleData, (sizeof(doubleData) / sizeof(doubleData[0])),
MPI_DOUBLE, 0, tagDoubleData, MPI_COMM_WORLD, &status );
```

```
/* Обчислюємо фактично прийняту кількість даних;
&status – статус завершення операції;
MPI_DOUBLE – повідомляємо MPI, тип повідомлення, що надійшло;
&count – сюди буде записано результат операції. */
MPI_Get_count(&status, MPI_DOUBLE, &count );
/* Виводимо фактичну довжину прийнятої кількості даних на екран. */
        cout << "Received" << count << "elems" << endl;
/* Аналогічно приймаємо повідомлення з даними типу float. Зверніть увагу:
завдання-приймач має можливість приймати повідомлення не в тому
порядку, в якому вони відправлялися, якщо ці повідомлення мають різні
ідентифікатори. */
   MPI_Recv( floatData, (sizeof(floatData) / sizeof(floatData[0])), MPI_FLOAT,
0, tagFloatData, MPI_COMM_WORLD, &status );
   MPI_Get_count( &status, MPI_FLOAT, &count );
   }
      /* Обидва завдання завершують своє виконання. */
   MPI_Finalize();
  return 0;
}
```

Зберегти зміни і вийти з редактора. Відкомпілювати програму:

mpic++ example7.cpp –o example5.bin.

2. Для виконання програми в пакетному режимі на кластері необхідно створити в редакторі *nano* pbs_script example7.pbs:

nano example7.pbs.

Увести в нього текст, що наведений далі.

```
#!/bin/sh
#PBS –d /home/ student<number>/lab5/
#PBS -l nodes=node1:ppn=2,walltime=00:30:00
#PBS -N example7

echo "File containing nodes:"
echo $PBS_NODEFILE
```

```
echo "Nodes for computing:"
cat $PBS_NODEFILE
echo "Start date: `date`"
OMPI_MCA_shmem_mmap_enable_nfs_warning=0
export OMPI_MCA_shmem_mmap_enable_nfs_warning
mpirun -n 2 -hostfile mpi.hosts ./example7.bin
echo "  End date: `date`"
```

Зберегти зміни і вийти з редактора. Відправити завдання в чергу, запустивши на виконання pbs_script:

qsub example7.pbs.

3. Перевірити результати роботи програми можна, переглянувши вміст файла example7.o<job_id>, перевірити помилки, які виникли під час роботи програми, можна в файлі example7.e<job_id>:

cat example7.o<job_id>

cat example7.e<job_id>.

Результат виконання програми наведений далі.

```
File containing nodes:

/var/spool/torque/aux//30079.cluster
Nodes for computing:
node1
node1
Start date: Чтв Жовт 27 09:20:32 EEST 2023
Received 6 elems
  End date: Чтв Жовт 27 09:20:33 EEST 2023.
```

Завдання. Опрацювати всі наведені приклади. Пояснити роботу кожної програми з прикладів. Розібрати роботу функцій стандарту *MPI*,

що використані в прикладах. У звіті навести *screenshot* з вихідними кодами програм, файли *pbs_script* та screenshot з результатами роботи програм за кожним прикладом.

Контрольні запитання до лабораторної роботи 5

1. Що таке відкритий стандарт *MPI*?
2. Пояснити поняття *rank* і *size*?
3. Які ключі використовуються під час запуску програм у стандарті *MPI*?
4. Пояснити призначення функцій стандарту *MPI,* що використовуються в програмах-прикладах.
5. Поясніть структуру програми *MPI*.
6. Що означає оператор *MPI_COMM_WORLD*?
7. Поясніть поняття "процес в програмі", що реалізовано в стандарті *MPI*.
8. Дайте порівняльну характеристику між стандартами *MPI* та *OpenMP*.
9. Поясніть, чим відрізняються операції типу "кома – кома" та колективні операції?
10. Поясніть взаємозв'язок ключів файла *pbs_script* та ключів команди запуску програми, що написана з використанням стандарту *MPI*.

Рекомендована література

Основна

1. Жуков І. А. Паралельні та розподілені обчислення : навч. посіб. / І. А. Жуков, О. В. Корочкін. – 2-ге вид. – Київ : "Корнійчук", 2014. – 284 с.

2. Корочкін О. В. Паралельні та розподілені обчислення. Вибрані розділи : навч. посіб. / О. В. Корочкін, О. В. Русанова. – Київ : КПІ ім. Ігоря Сікорського, 2020. – 123 с.

3. Семеренко В. П. Технології паралельних обчислень : навч. посіб. / В. П. Семеренко. – Вінниця : ВНТУ, 2018. – 104 с.

4. Ясько М. М. Навчальний посібник до вивчення курсів "Паралельна обробка даних" та "Мови обчислень та кластерні системи" / М. М. Ясько. – Дніпро : РВВ ДНУ, 2010. – 76 с.

5. Grama A. Introduction to Parallel Computing / A. Grama, A. Gupta, V. Kumar. – Harlow : Addison-Wesley, 2003. – 452 p.

Додаткова

6. Коцовський В. М. Теорія паралельних обчислень : навч. посіб. / В. М. Коцовський. – Ужгород : ПП "АУТДОР-Шарк", 2021. – 188 с.

7. Малашонок Г. І. Паралельні обчислення на розподіленій пам'яті: OpenMPI, Java, Math Partner : підручник / Г. І. Малашонок, А. А. Сідько. – Київ : НаУКМА, 2020. – 266 с.

Інформаційні ресурси

8. Introduction to Parallel Programming with MPI and OpenMP [Electronic resource]. – Access mode : https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf.

9. Message Passing Interface. CHameleon [Electronic resource]. – Access mode : https://www.mpich.org.

10. OpenMP (Open Multi-Processing) [Electronic resource]. – Access mode : https://www.openmp.org.

11. Slurm Workload Manager – Documentation [Electronic resource]. – Access mode : https://slurm.schedmd.com.

12. The OpenMP API specification for parallel programming [Electronic resource]. – Access mode : http://openmp.org/wp/openmp-specifications/.

13. Victor Eijkhout. Parallel Programming in MPI and OpenMP 2016 [Electronic resource]. – Access mode : https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/Eijkhout ParallelProgramming.pdf.

Зміст

Вступ 3
Лабораторна робота 1. Створення та налаштування вузлів обчислювального кластера 5
Лабораторна робота 2. Налаштування системи *SLURM* на обчислювальному кластері 49
Лабораторна робота 3. Дослідження побудови обчислювального кластера та принципів роботи з ним 77
Лабораторна робота 4. Паралельне програмування у відкритому стандарті *OpenMP* та його застосування під час розв'язання системи лінійних алгебраїчних рівнянь (СЛАР) 96
Лабораторна робота 5. Паралельне програмування в стандарті MPI 137
Рекомендована література 169

НАВЧАЛЬНЕ ВИДАННЯ

РОЗПОДІЛЕНІ ТА ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

Лабораторний практикум

для студентів спеціальності
121 "Інженерія програмного забезпечення"
освітньої програми
"Інженерія програмного забезпечення"
першого (бакалаврського) рівня

Самостійне електронне текстове мережеве видання

Укладачі: **Мінухін** Сергій Володимирович
Савін Юрій Вікторович

Відповідальний за видання *Д. О. Бондаренко*

Редактор *В. О. Дмитрієва*

Коректор *Н. В. Завгородня*

План 2023 р. Поз. № 95 ЕВ. Обсяг 171 с.

Видавець і виготовлювач – ХНЕУ ім. С. Кузнеця, 61166, м. Харків, просп. Науки, 9-А

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру
ДК № 4853 від 20.02.2015 р.