Sechenov HPC

Alexander Danilov

01.04.2024

Введение:

Вычислительный кластер установлен в Институте компьютерных наук и математического моделирования Научно-технологического парка биомедицины Сеченовского университета.

Важные объявления и изменения в работе кластера будут опубликованы в разделе новостей и продублированы сообщениями при входе на кластер.

Раздел с инструкциями будет дополняться. Если у вас есть предложения или пожелания, напишите об этом системным администраторам кластера.

1 Краткие инструкции

Здесь собраны основные правила работы с кластером. Более подробная информация находится в соответствующих разделах.

1.1 Получение аккаунта

Для получения аккаунта свяжитесь с системными администраторами кластера и предоставьте им необходимую информацию. *Регистрация новых пользователей*.

1.2 Смена пароля

После регистрации вы получите временный пароль для входа на кластер. Смените этот пароль после входа на кластер с помощью команды `passwd`.

1.3 Подключение к кластеру

Подключение к кластеру осуществляется по протоколу SSH. Внутренний IP-адрес сервера: `10.0.12.2`. Для доступа из внешней сети необходимо использование VPN, обратитесь к администраторам кластера за дополнительной информацией.

1.4 Узлы кластера и дисковое пространство

При подключении вы войдёте на головной узел `hn1`. Пожалуйста, не запускайте расчёты на этом узле, используйте его только для подготовки своих задач. *Обзор кластера*.

Для каждого пользователя доступна домашняя директория `/home/username`. Эта директория доступна на всех узлах кластера.

Для передачи файлов на кластер или в обратную сторону вы можете использовать любую программу, работающую по протоколу SSH. Например в системах Linux и OSX это программы `scp`, `rsync` и др., в Windows – WinSCP.

1.5 Запуск программ

Для запуска задач на кластере используется система очередей Slurm. Подготовьте скрипт для запуска вашей задачи. В нём вы можете указать желаемое количество узлов, вычислительных ядер, графических ускорителей и необходимое время для расчётов. Далее скрипт можно поставить в очередь с помощью команды `sbatch jobscript.sh`. Скрипт будет запущен на свободных узлах. Информацию о своих задачах можно получить с помощью команды `squeue -u username`.

Система очередей Slurm.

1.6 Решение вопросов

По всем вопросам, связанным с работой на кластере, обращайтесь к администраторам кластера. *Контакты*.

2 Обзор кластера

Вычислительный кластер состоит из одного головного узла `hn1`, 24 вычислительных CPU-узлов `c[01-24]` и двух вычислительных GPU-узлов `g[01-02]`. Все узлы объединены высокоскоростной сетью InfiniBand. На всех узлах установлена операционная система Rocky Linux 8.5.

2.1 Головной узел

Узел `hn1` используется для подготовки и установки задач в очередь.

2.2 Вычислительные CPU-узлы

На кластере доступно 24 узла, в каждом из которых установлено 2 процессора по 26 ядер, всего 1248 ядер. Используйте очередь задач `cpu` для запуска задач на CPU-узлах.

2.3 Вычислительные GPU-узлы

На кластере есть два узла с графическими ускорителями.

На узле `g01` установлены 4 карты Nvidia Tesla V100S. На узле `g02` установлены 8 карт Nvidia GeForce RTX 2080 Ti. Используйте очередь задач `gpu` для запуска задач на GPU-узлах. Обязательно укажите нужный тип карт и их количество с помощью опции `--gpus`, например:

```
# две карты Nvidia Tesla V100S (тип v100)
sbatch --partition=gpu --gpus=v100:2 gpu_job.sh

# четыре карты Nvidia GeForce RTX 2080 Ti (тип rtx)
sbatch --partition=gpu --gpus=rtx:4 gpu_job.sh
```

Задачи, запущенные без указания опции `--gpus`, не будут иметь доступа к графическим ускорителям.

3 Новости и объявления

3.1 Запасной IP адрес

В случае проблем с подключением к основному IP адресу кластера `10.0.12.2`, попробуйте воспользоваться запасным IP адресом `10.4.5.41`.

3.2 Отключение кластера 14 марта 2024 года

14 марта 2024 года с 11:00 до 16:00 кластер будет выключен для проведения монтажных работ. Запуск продолжительных задач будет отложен до окончания работ.

3.3 Смена IP адреса

С 14 сентября 2023 года изменяется внутренний IP адрес кластера. Новый адрес – `10.0.12.2`. Также с 14 сентября 2023 года подключение по внешнему IP адресу более недоступно. Для доступа из внешней сети необходимо использование VPN, обратитесь к администраторам кластера за дополнительной информацией.

3.4 GPU сервер `g02`

GPU сервер `g02` доступен с 19 апреля 2022 года.

4 Контакты

По вопросам, связанным с добавлением групп и пользователей, установкой дополнительных программ, с вопросами о работе с кластером, вы можете обратиться к одному из системных администраторов кластера:

- Данилов Александр Анатольевич, danilov_a_a_2@staff.sechenov.ru

5 Регистрация новых пользователей

Все пользовательские аккаунты на кластере распределены по группам. У каждого пользователя есть основная группа, а также могут быть дополнительные, например, если он участвует в нескольких проектах. *Группы пользователей*.

5.1 Анкета для регистрации

Для создания нового аккаунта предоставьте следующую информацию:

1. ФИО на русском и английском языках
2. Контактный e-mail
3. Номер мобильного телефона
4. Адрес корпоративной электронной почты Сеченовского университета (при наличии)
5. Название института, подразделения, должность
6. Название основной группы пользователей и дополнительных, если сотрудник участвует в нескольких проектах
7. Согласие руководителей указанных групп

После создания нового аккаунта временный пароль будет выслан по указанному адресу e-mail.

5.2 Смена пароля

После регистрации вы получите временный пароль для входа на кластер. Смените этот пароль после входа на кластер с помощью команды `passwd`.

6 Группы пользователей

Все пользовательские аккаунты на кластере распределены по группам. У каждого пользователя есть основная группа, а также могут быть дополнительные, например, если он участвует в нескольких проектах.

Группа пользователей объединяет одного или нескольких пользователей, работающих над общим проектом. По умолчанию файлы внутри группы доступны всем участникам группы и не видны остальным пользователям.

6.1 Анкета для группы

Для создания новой группы предоставьте следующую информацию:

1. ФИО руководителя группы
2. Контакты руководителя группы: e-mail, мобильный телефон
3. Название института и подразделения, в котором выполняется проект
4. Название проекта, номер гранта (при наличии)
5. Желаемое название группы (латинские строчные буквы)
6. Оценка требуемых ресурсов: объём дискового пространства в ГБ, необходимое количество CPU ядер и/или GPU карт, ожидаемое максимальное время одного расчёта (информация собирается для планирования ресурсов и настройки приоритетов в очереди задач)

6.2 Отчётность групп

При публикации результатов расчётов, полученных с использованием вычислительного кластера, просим ссылаться в тексте на вычислительный кластер Института компьютерных наук и математического моделирования Сеченовского Университета (HPC system of the Institute for Computer Science and mathematical modeling, Sechenov University).

7 Доступ к кластеру

Подключение к кластеру осуществляется по протоколу SSH. Внутренний IP-адрес сервера: `10.0.12.2`. Для доступа из внешней сети необходимо использование VPN, обратитесь к администраторам кластера за дополнительной информацией.

7.1 Отпечатки серверных ключей

При первом входе на кластер вам будет предложено проверить отпечаток ключа сервера. Обязательно проверьте, что он совпадает с одним из перечисленных ниже:

Ключ ED25519

```
SHA256-хеш: SHA256:ja8wC67SQ4IHPnFx5J+q1b6p/bfi7gciqzmC26hPGYE
MD5-хэш: MD5:43:88:92:b5:22:61:6b:fd:65:18:f8:b5:c6:71:af:4c
```

Ключ ECDSA

```
SHA256-хеш: SHA256:AA5ypfZDkCDHeYtVu/Y9/QGBSqItG6TcjJoInES2s/4
MD5-хэш: MD5:78:ee:02:86:3e:99:1d:31:b9:84:42:71:a4:c3:f2:f1
```

Ключ RSA

```
SHA256-хеш: SHA256:kIJj6JXszxjoCJSRg1YMuVwgwLOhDozXCzkT8W4qAqc
MD5-хэш: MD5:b7:c6:d3:87:7f:16:dd:b0:f7:86:a0:f1:49:1d:a9:71
```

7.2 Смена пароля

После регистрации нового пользователя вы получите временный пароль для входа на кластер. Смените этот пароль после входа на кластер с помощью команды `passwd`.

8 Модульное управление пакетами

На кластере установлена система управления загружаемыми модулями Lmod[1].

8.1 Краткая инструкция

С помощью команды `module list` можно увидеть список активированных модулей.

С помощью команды `module avail` можно увидеть список доступных для активации модулей.

С помощью команды `module load <x>` можно загрузить модуль <x>. Можно указать конкретную версию модуля, или загрузить сразу несколько модулей, например, команда:

```
module load intel/2022.0.1 impi
```

загрузит модуль компилятора Intel версии 2022.0.1 и последнюю версию библиотеки Intel MPI.

Любой загруженный модуль можно выгрузить с помощью команды `module unload <x>`.

Можно переключиться на другую версию уже загруженного модуля с помощью команды `module load <x>/<version>`.

Одновременно может быть загружен только один модуль из категории компиляторов (`intel` или `gnu9`) и только одна библиотека MPI (`impi`, `mpich`, `openmpi4` и др.).

У модулей есть зависимости, и модули становятся доступны только после загрузки всех необходимых модулей. Например перед загрузкой MPI библиотеки нужно загрузить хотя бы один модуль с компилятором. Для

[1] https://lmod.readthedocs.io/en/latest/

загрузки параллельных библиотек (например, `petsc`) необходимо предварительно загрузить модуль с MPI библиотекой.

Используйте команду `module spider` чтобы найти все модули в системе. Также можно добавить ключевое слово, чтобы найти конкретный модуль, например, `module spider petsc`.

Команда `module show <x>` покажет список переменных окружения, которые будут добавлены при активации модуля. Эта информация может быть полезной, чтобы понять какие переменные нужно использовать для доступа к файлам модуля. Как правило они имеют вид `NAME_DIR`, `NAME_INC`, `NAME_LIB` и т.п.

С помощью команды `module purge` можно выгрузить все активированные модули.

Также вы можете использовать короткую версию команды `ml` вместо `module list`, `ml <x>` вместо `module load <x>`, `ml -<x>` вместо `ml unload <x>`, `ml av` вместо `module avail` и т.д.

Команды `module load <x>` можно добавить в свой `~/.bashrc` файл.

Дополнительную информацию можно найти в документации Lmod[2].

9 Компиляторы Intel

На всех узлах кластера доступны компиляторы и вспомогательные библиотеки Intel oneAPI.

Список установленных версий можно увидеть с помощью команды `module avail intel`.

При подключении модуля `intel` станет доступен модуль `impi` для библиотеки Intel MPI.

Для запуска MPI-задач в скрипте используйте команду `mpirun`.

10 Система очередей Slurm

Для того чтобы запустить задачу на кластере, необходимо использовать систему управления заданиями Slurm. Обычно для этого создаётся специальный скрипт, и ставится в очередь с помощью команды `sbatch`. Этот скрипт содержит информацию о необходимых ресурсах (число узлов кластера, количество ядер процессора, количество и тип графических ускорителей, необходимое количество оперативной памяти и необходимое время). В остальном этот скрипт является обычным bash-скриптом, в нём можно настроить нужные переменные среды и запустить необходимую программу. Этот скрипт будет запущен лишь на одном ядре из выделенного списка. В обязанности скрипта входит запуск программы на всех остальных узлах, например, с помощью `mpirun` или `srun`.

10.1 Пример скрипта

Начнём с простого примера для задачи `sleep` на одно ядро с максимальным временем выполнения 5 минут. Создайте файл `sleep.sh` со следующим содержимым

```
1 #!/bin/bash
2 #SBATCH --job-name=sleep
3 #SBATCH --nodes=1
4 #SBATCH --time=05:00
5
6 echo "Start date: $(date)"
7 sleep 60
8 echo "  End date: $(date)"
```

Первая строка всегда должна указывать на программу-интерпретатор скрипта, например `/bin/bash`. В строках 2–4 указаны параметры задачи: название, количество узлов, максимальное время выполнения. В строках 6-8 указаны команды, которые будут выполнены на вычислительном узле.

[2] https://lmod.readthedocs.io/en/latest/

Отправка задачи на кластер осуществляется с помощью команды `sbatch`:

```
sbatch sleep.sh
```

Строчки, начинающиеся с `#SBATCH`, содержат параметры для команды `sbatch`. Эти параметры можно также указывать явно при вызове `sbatch`. Например:

```
sbatch  --job-name sleep  --nodes=1  --time=5:00  sleep.sh
```

10.2 Команда `sbatch`

Основные параметры команды `sbatch`:

- `-D path` или `--chdir=path`
  Определяет рабочую директорию для задания. Если не задана, то рабочей является текущая директория.
- `-e path/file` или `--error=path/file`
  `-o path/file` или `--output=path/file`
  Задаются имена файлов ошибок (`stderr`), и стандартного вывода (`stdout`). По умолчанию оба вывода объединяются в один файл `slurm-<job_id>.out` в текущей директории.
- `-J name` или `--job-name=name`
  Определяет имя задачи.
- `-p queue` или `--partition=queue`
  Задаёт очередь в которую добавляется задача. На сервере есть несколько очередей. По умолчанию задачи ставятся в очередь `cpu`.
- `-n N` или `--ntasks=N`
  Запрашивает `N` процессов для задачи.
- `-N N` или `--nodes=N`
  Запрашивает `N` вычислительных узлов для задачи.
- `--nodes=N --ntasks-per-node=M`
  Запрашивает `N` вычислительных узлов, и `M` процессов на каждом узле.
- `--cpus-per-task=N`
  Дополнительно запрашивает `N` процессорных ядер на каждый процесс (например для гибридных задач MPI+OpenMP). По умолчанию выделяется одно ядро на процесс.
- `--gpus=N` или `--gpus=type:N`
  Дополнительно запрашивает доступ к `N` графическим ускорителям типа `type` (если тип явно указан).
- `--mem=size`
  Запрашивает необходимую память на каждом узле. Размер указывается с помощью целого числа и суффикса: `K`, `M`, `G`. Например, `--mem=16G` запросит 16 Гб памяти на каждом узле.
- `-t time` или `--time=time`
  Ограничивает максимальное время выполнения задачи. По истечении этого времени программа будет завершена. Значение указывается в минутах, либо в одном из форматов `MM:CC`, `ЧЧ:MM:CC`, `ДД-ЧЧ`.

Переменные окружения, которые устанавливает Slurm:

- `SLURM_SUBMIT_DIR`
  Директория, в которой находился пользователь во время отправки задачи в очередь.
- `SLURM_JOB_ID`
  Уникальный номер задачи.
- `SLURMD_NODENAME`

Текущий узел, на котором запущен скрипт.

- `SLURM_NTASKS`
  Количество выделенных процессорных ядер.

Дополнительную информацию можно получить в документации `man sbatch`.

10.3 Команда `salloc`

В отличие от команды `sbatch`, которая ставит готовый скрипт в очередь задач, команда `salloc` позволяет выделить узлы и использовать их в интерактивном режиме. Основные параметры те же, что у команды `sbatch`.

10.4 Команда `squeue`

Просмотреть состояние задач в очереди можно с помощью команды `squeue`. Команда `squeue -u user` покажет только задачи пользователя `user`. Текущее состояние задачи отмечено в столбце `ST`. * `PD` — задача находится в очереди, ждёт освобождения ресурсов. * `R` — задача в данный момент выполняется. * Описание других состояний см. в документации `man squeue`.

В столбце `NODELIST` перечислены узлы, выделенные для задачи.

С помощью команды `squeue -l` можно также увидеть запрошенное время для каждой задачи, а с помощью `squeue --start` можно узнать ожидаемое время запуска задачи.

10.5 Команда `scancel`

Если по каким-то причинам задача так и не начала запускаться, например, запрошено слишком много ядер, или памяти, то удалить задачу из очереди можно с помощью команды `scancel <job_id>`. Точно так же задачу можно удалить, если она уже выполняется (при этом она будет сразу завершена).

10.6 Команда `scontrol`

Если нужно изменить параметры уже поставленной в очередь задачи, то можно использовать команду `scontrol`. Например, чтобы изменить максимальное время расчёта выполните

```
scontrol update jobid=<id> timelimit=<limit>
```

где `<id>` – идентификатор задачи, `<limit>` – новый лимит времени (увеличить максимальное время можно только до начала расчёта).

Описание других параметров команды см. в документации `man scontrol`.

10.7 Вспомогательные команды

Команда `slurmtop` отображает основную информацию о состоянии кластера (общая информация, занятость узлов и очередь задач). Для выхода из программы нажмите клавишу `q`.

Команда `pestat` показывает краткую сводку о занятости узлов кластера. При запуске с параметром `pestat -G` будет также отображена информация о количестве и типе GPU устройств.

11 Очереди задач

На кластере доступно три очереди задач:

- `cpu` – очередь для запуска задач с использованием CPU-узлов (эта очередь используется по умолчанию).
- `gpu` – очередь для запуска задач с использованием GPU-узлов.
- `mix` – смешанная очередь для запуска гибридных задач с использованием CPU-узлов совместно с GPU-узлами.

Чтобы отправить задачу, например, в очередь `gpu` нужно добавить параметр `-p gpu` или `--partition=gpu` для команды `sbatch`:

```
sbatch --partition=gpu --gpus=v100:2 --ntasks-per-gpu=4 --time=0-6 run_gpu.sh
```

Также можно добавить строчку:

```
#SBATCH --partition=gpu
```

в скрипт для `sbatch`

11.1 Очередь `gpu`

При использовании очереди с графическими ускорителями обязательно укажите нужный тип карт (`v100` или `rtx`) и их количество с помощью опции `--gpus`, например:

```
sbatch --partition=gpu --gpus=v100:2 gpu_job.sh
```

Задачи, запущенные без указания опции `--gpus`, не будут иметь доступа к графическим ускорителям.

11.2 Очередь `mix`

Используйте очередь `mix` только если вам действительно нужны одновременно CPU-узлы и графические ускорители. Гетерогенная структура кластера потребует дополнительной подготовки скрипта для запуска задачи и специальной организации межпроцессорных обменов для эффективной работы.

12 Пример запуска HPLinpack

В этом примере мы запустим популярный тест Linpack[3] для оценки производительности параллельных систем.

12.1 Установка HPLinpack

Для установки нам потребуется архив с исходным кодом пакета, доступный на сайте Linpack[4]. Скачиваем и распаковываем архив:

```
wget https://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz
tar xf hpl-2.3.tar.gz
cd hpl-2.3
```

[3] https://www.netlib.org/benchmark/hpl/
[4] https://www.netlib.org/benchmark/hpl/

Для компиляции кода HPL нам потребуется шаблон Makefile с параметрами компиляции. За основу можно взять шаблон `setup/Make.Linux_Intel64` для компиляторов Intel. В этом шаблоне мы обновим значение переменной `TOPdir` и вместо устаревшего ключа `-openmp` укажем новый ключ `-qopenmp`:

```
# команда sed изменяет значение TOPdir на $(pwd) - текущую директорию
# и также изменяет параметр -openmp на -qopenmp
sed "/TOPdir\s\+=/s:=.*:= $(pwd):; s:-openmp:-qopenmp:" setup/Make.Linux_Intel64 >␣
↪Make.Linux_Intel64
```

Для сборки загрузим модули для компиляторов Intel и библиотеки Intel MPI и запустим `make` с указанием профиля `Linux_Intel64`:

```
module load intel impi
make arch=Linux_Intel64
```

Скомпилированная программа будет расположена в `bin/Linux_Intel64/xhpl`.

12.2 Запуск HPLinpack

После сборки мы можем запустить простейший тест с использованием стандартного файла `HPL.dat`. Для запуска нам потребуется как минимум 4 ядра.

Тестовый запуск проведём в интерактивном режиме:

```
cd bin/Linux_Intel64
# переходим в интерактивный режим с использованием 4 ядер
salloc --ntasks=4
# внутри интерактивного режима запускаем программу
mpirun ./xhpl
# вывод программы довольно большой, в конце содержит следующие строки:
# Finished    864 tests with the following results:
#              864 tests completed and passed residual checks,
#                0 tests completed and failed residual checks,
#                0 tests skipped because of illegal input values.

# для выхода из интерактивного режима используем команду exit
exit
```

Теперь попробуем запустить расчёт с использованием 4 CPU-узлов (208 ядер). Подготовим отдельную директорию для расчёта с файлом конфигурации *HPL.dat* и скриптом для очереди задач *xhpl-4nodes.sh*.

Список 1: `HPL.dat`

```
1  HPLinpack benchmark input file
2  Innovative Computing Laboratory, University of Tennessee
3  HPL.out      output file name (if any)
4  6            device out (6=stdout,7=stderr,file)
5  2            # of problems sizes (N)
6  79872 319488 Ns
7  1            # of NBs
8  384          NBs
9  0            PMAP process mapping (0=Row-,1=Column-major)
10 1            # of process grids (P x Q)
11 13           Ps
12 16           Qs
13 16.0         threshold
14 1            # of panel fact
15 2            PFACTs (0=left, 1=Crout, 2=Right)
16 1            # of recursive stopping criterium
17 4            NBMINs (>= 1)
18 1            # of panels in recursion
```

(continues on next page)

(продолжение с предыдущей страницы)

```
2            NDIVs
1            # of recursive panel fact.
2            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
384          swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

В этом примере файла `HPL.dat` мы указываем размеры систем в строке 6. Размер `N=79872` является довольно маленьким, и обычно требует чуть больше 1 минуты для решения, размер `N=319488` в 4 раза больше, требует в 16 раз больше оперативной памяти и в 64 раза больше арифметических операций. Как правило, чем больше система, тем более эффективно будет работать Linpack, поэтому время для большой матрицы будет меньше чем время для маленькой, умноженное на 64. В строке 8 мы указываем большое значение `NB=384`, так как процессоры, установленные в CPU-узлах содержат по два модуля AVX-512. Разбиение системы на блоки указывается в строках 11-12. Произведение `P×Q` должно совпадать с количеством доступных процессоров `13×16 = 4×52`.

Список 2: `xhpl-4nodes.sh`

```
#!/bin/bash
#SBATCH --job-name=xhpl-4      # название задачи
#SBATCH --nodes=4              # 4 вычислительных узла
#SBATCH --ntasks-per-node=52   # по 52 MPI-процесса на узел
#SBATCH --time=1:20:00         # оценка времени - 1 час 20 минут

# загружаем модули intel и impi
module load intel impi
# исправьте путь к xhpl, если он отличается
mpirun ~/benchmark/hpl-2.3/bin/Linux_Intel64/xhpl
```

В скрипте указываются параметры задачи: в строке 2 – название задачи, в строках 3-4 – желаемое количество узлов и MPI-процессов на узле, в строке 5 – максимальное время выполнения задачи (оценочное время – до 60 минут, мы добавим ещё 20 на всякий случай). В строке 8 мы загружаем модули `intel` и `impi` на случай, если они не были загружен ранее. В строке 10 запускается MPI-программа `xhpl` (не забудьте указать корректный путь).

Для запуска задачи в системе очередей используется команда `sbatch`:

```
sbatch xhpl-4nodes.sh
```

Результаты расчёта будут сохранены в файле `slurm-XXX.out`, где `XXX` – номер задачи. Время расчёта с указанными параметрами обычно составляет от 50 до 60 минут. Сводную таблицу с временем работы и оценкой скорости вычислений можно получить с помощью следующей команды:

```
# вместо XXX нужно указать номер задачи, или использовать *
cat slurm-XXX.out | (grep -B2 -m1 WR; grep WR)
# пример вывода:
#  T/V                N    NB     P     Q               Time                 Gflops
# ----------------------------------------------------------------------------------
# WR01R2R4       79872   384    13    16             59.26             5.7322e+03
# WR01R2R4      319488   384    13    16           2837.69             7.6614e+03
```

В результате получена производительность `R_max = 7661 Gflops`. Оценим максимальную производительность вычислений на 4 вычислительных узлах. Всего используется `4×52=208` ядер. В каждом ядре установлены два AVX-512 модуля, что позволяет выполнять до 32 арифметических операций с плавающей

точкой на одном ядре за один такт. При частоте AVX-512 модуля 2.1ГГц получается оценка `R_peak = 208×32×2.1 = 13977 Gflops`. Эффективность `R_max / R_peak = 55%`.

13 Пример запуска HPL-NVIDIA

В этом примере мы запустим модифицированный тест Linpack от NVIDIA[5] для оценки производительности графических ускорителей.

13.1 Установка HPL-NVIDIA

Один из самых простых способов запустить HPL-NVIDIA – воспользоваться готовым контейнером `hpc-benchmarks` с сайта NVIDIA[6]. На кластере для работы с контейнерами используется пакет Singularity, он доступен в системе загрузки модулей. Для загрузки контейнера `hpc-benchmarks` нужно зарегистрироваться на сайте NVIDIA NGC[7] и получить API key[8].

```
# загружаем модуль singularity
module load singularity
# скачиваем образ Docker-контейнера и конвертируем в образ Singularity
singularity pull --docker-login hpc-benchmarks:21.4-hpl.sif \
    docker://nvcr.io/nvidia/hpc-benchmarks:21.4-hpl
# для скачивания потребуется API key
```

Образ `hpc-benchmarks` будет сохранён в текущей директории в файле `hpc-benchmarks:21.4-hpl.sif`.

13.2 Запуск HPL-NVIDIA

Запустим расчёт с использованием 4 графических ускорителей Nvidia Tesla V100S. Подготовим файл конфигурации *HPL-4xV100.dat* и скрипт для очереди задач *hpl-nvidia-4-v100.sh*.

Список 3: `HPL-4xV100.dat`

```
1  HPLinpack benchmark input file
2  Innovative Computing Laboratory, University of Tennessee
3  HPL.out      output file name (if any)
4  6            device out (6=stdout,7=stderr,file)
5  1            # of problems sizes (N)
6  130560       Ns
7  1            # of NBs
8  384          NBs
9  0            PMAP process mapping (0=Row-,1=Column-major)
10 1            # of process grids (P x Q)
11 2            Ps
12 2            Qs
13 16.0         threshold
14 1            # of panel fact
15 1            PFACTs (0=left, 1=Crout, 2=Right)
16 1            # of recursive stopping criterium
17 4            NBMINs (>= 1)
18 1            # of panels in recursion
19 2            NDIVs
20 1            # of recursive panel fact.
```

(continues on next page)

[5] https://catalog.ngc.nvidia.com/orgs/nvidia/containers/hpc-benchmarks
[6] https://catalog.ngc.nvidia.com/orgs/nvidia/containers/hpc-benchmarks
[7] https://ngc.nvidia.com/signin/
[8] https://ngc.nvidia.com/setup/api-key

(продолжение с предыдущей страницы)

```
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
3            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
1            SWAP (0=bin-exch,1=long,2=mix)
192          swapping threshold
1            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
0            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

В этом примере файла `HPL.dat` мы указываем размер системы в строке 6. Размер `N=130560` подобран из соображений максимального использования GPU памяти. 4 карты по 32 Гб памяти в сумме дают 128 Гб памяти, что позволяет хранить плотную матрицу 131072×131072 чисел с плавающей точкой с двойной точностью. Мы немного уменьшим значение `N`, чтобы оно стало кратным параметру `NB=384`. Разбиение системы на блоки указывается в строках 11-12. Произведение `P×Q` должно совпадать с количеством используемых графических ускорителей.

Список 4: `hpl-nvidia-4-v100.sh`

```
#!/bin/bash
#SBATCH --job-name=hpl-nvidia-4-v100    # название задачи
#SBATCH --partition=gpu                 # очередь gpu
#SBATCH --gpus=v100:4                   # 4 ускорителя v100
#SBATCH --nodes=1                       # 1 вычислительный узел
#SBATCH --ntasks-per-node=4             # 4 процесса на один узел
#SBATCH --cpus-per-task=6               # 6 ядер на один процесс
#SBATCH --time=0:10:00                  # оценка времени 10 минут

# загружаем модуль singularity для контейнера
module load singularity
# предполагается, что контейнер hpc-benchmarks:21.4-hpl.sif
# находится в текущей директории
srun --mpi=pmi2  singularity run --nv \
    ./hpc-benchmarks:21.4-hpl.sif hpl.sh \
    --cpu-affinity all:all:all:all \
    --cpu-cores-per-rank $SLURM_CPUS_PER_TASK \
    --gpu-affinity 0:1:2:3 \
    --dat HPL-4xV100.dat
```

В скрипте указываются параметры задачи: в строке 2 – название задачи, в строке 3 – название очереди `gpu` для серверов с графическими ускорителями, в строке 4 – необходимое количество и тип графических ускорителей, в строке 5 – количество GPU-узлов (в данном случае этот параметр можно опустить), в строке 6 – количество процессов на одном сервере (можно также просто указать `--ntasks=4`), в строке 7 – количество CPU ядер на один процесс, в строке 8 – максимальное время выполнения задачи. В строке 11 загружается модуль `singularity` для работы с контейнерами Singularity. В строках 14-19 запускается параллельный расчёт задачи: в строке 14 программа `srun` запускает 4 процесса `singularity`, параметр `--nv` у `singularity` предоставляет контейнеру доступ к графическим ускорителям, в строке 15 указано название контейнера `./hpc-benchmarks:21.4-hpl.sif` и запускаемый скрипт внутри контейнера `hpl.sh`, в строке 16 указывается привязка процессов к ядрам CPU, мы указываем ключевое слово `all`, чтобы использовать привязку Slurm, в строке 17 указывается количество ядер на один процесс, мы указываем значение `$SLURM_CPUS_PER_TASK`, которое совпадает с параметром `--cpus-per-task`, в строке 18 указывается привязка GPU-карт к процессам, в строке 19 указывается имя файла с параметрами `HPL.dat`.

Для запуска задачи в системе очередей используется команда `sbatch`:

```
sbatch hpl-nvidia-4-v100.sh
```

Результаты расчёта будут сохранены в файле `slurm-XXX.out`, где `XXX` – номер задачи. Время расчёта с указанными параметрами обычно составляет несколько минут. Сводную таблицу с временем работы и оцен-

кой скорости вычислений можно получить с помощью следующей команды:

```
# вместо XXX нужно указать номер задачи, или использовать *
grep -B2 WR slurm-XXX.out
# пример вывода:
# T/V                N    NB     P     Q               Time                 Gflops
# --------------------------------------------------------------------------------
# WR03C2C4      130560   384     2     2             61.55              2.411e+04
```

В результате получена производительность `R_max = 24110 Gflops`.

В примере выше мы использовали не все доступные ядра на узле `g01`. Наш выбор обусловлен особенностями подключения GPU-карт к материнской плате. На узле `g01` установлены 4 карты Nvidia Tesla V100S, и все они подключены к PCI-шине первого процессора. Такое подключение позволяет ускорить обмены между GPU-картами через общую PCI-шину. С другой стороны, обмены между вторым процессором и GPU-картами происходят медленнее. При использовании 6 ядер на одну GPU-карту нам достаточно 24 ядер, и они все могут быть размещены на первом процессоре.

Попытка добавить больше ядер приведёт лишь к снижению производительности из-за дополнительных накладных расходов при обменах между вторым процессором и GPU-картами:

```
# увеличиваем количество ядер на процесс в два раза
sbatch --cpus-per-task=12 hpl-nvidia-4-v100.sh
grep -B2 WR slurm-XXX.out
# пример вывода:
# T/V                N    NB     P     Q               Time                 Gflops
# --------------------------------------------------------------------------------
# WR03C2C4      130560   384     2     2             87.42              1.697e+04
```

В результате подключения к работе второго процессора производительность уменьшилась до `R = 16970 Gflops`.

Система очередей Slurm автоматически выбирает свободные ядра первого процессора при запросе графических ускорителей V100 на узле `g01`.

14 Запуск гибридных задач MPI+OMP

Система очередей Slurm поддерживает запуск гибридных задач, использующих технологии MPI и OMP. В этом примере мы продемонстрируем работу гибридных задач на простейшей программе xthi.c[9].

14.1 Компиляция гибридных программ

Для экспериментов будем использовать пример простейшей гибридной программы `xthi`, использующей технологии MPI и OMP. Код программы можно найти, например, на сайте NERSC[10].

```
wget https://docs.nersc.gov/jobs/affinity/xthi.c
```

Для компиляции программы с поддержкой MPI нам потребуется MPI-компилятор, для поддержки OMP как правило нужно указать дополнительный параметр. Например, при использовании компиляторов Intel и библиотеки Intel MPI используется параметр `-qopenmp`:

```
# загружаем модули intel и impi
module load intel impi
# компилятор mpiicc, добавляем параметр -qopenmp
mpiicc -qopenmp xthi.c -o xthi.impi
```

При использовании компиляторов GCC и библиотеки OpenMPI используется параметр `-fopenmp`:

[9] https://docs.nersc.gov/jobs/affinity/xthi.c
[10] https://docs.nersc.gov/jobs/affinity/xthi.c

```
# загружаем модули gnu9 и openmpi4
module load gnu9 openmpi4
# компилятор mpicc, добавляем параметр -fopenmp
mpicc -fopenmp xthi.c -o xthi.openmpi
```

Обратите внимание, что при использовании компиляторов Intel MPI-компилятор называется `mpiicc`, а при использовании компиляторов GCC компилятор называется `mpicc`. Также в первом случае для поддержки OMP параметр называется `-qopenmp`, а во втором – `-fopenmp`.

В примерах выше мы скомпилировали две версии программы `xthi`, одну с использованием Intel MPI – `xthi.impi` и одну с использованием OpenMPI – `xthi.openmpi`.

14.2 Запуск гибридных программ

Подготовим скрипт для запуска программы в системе очередей Slurm:

```
1  #!/bin/bash
2  #SBATCH --job-name=xthi          # название задачи
3  #SBATCH --nodes=2                # 2 вычислительных узла
4  #SBATCH --ntasks-per-node=4      # по 4 MPI-процесса на узел
5  #SBATCH --cpus-per-task=8        # по 8 ядер на MPI-процесс
6  #SBATCH --time=05:00             # оценка времени - 5 минут
7
8  # выгружаем все загруженные модули
9  module purge
10 # загружаем модули intel и impi
11 module load intel impi
12 # запускаем с помощью mpirun
13 # сохраняем результат в result.impi.mpirun
14 echo "Intel MPI, run with mpirun"
15 mpirun ./xthi.impi | tee result.impi.mpirun
16 # запускаем с помощью srun
17 # сохраняем результат в result.impi.srun
18 echo "Intel MPI, run with srun --mpi=pmi2"
19 srun --mpi=pmi2 ./xthi.impi | tee result.impi.srun
20
21 # выгружаем все загруженные модули
22 module purge
23 # загружаем модули gnu9 и openmpi4
24 module load gnu9 openmpi4
25 # запускаем с помощью mpirun
26 # сохраняем результат в result.openmpi.mpirun
27 echo "OpenMPI, run with mpirun"
28 mpirun ./xthi.openmpi | tee result.openmpi.mpirun
29 # добавляем опцию --map-by
30 # сохраняем результат в result.openmpi.map
31 echo "OpenMPI, run with mpirun"
32 mpirun --map-by slot:PE=$SLURM_CPUS_PER_TASK ./xthi.openmpi | tee result.openmpi.
   ↪map
```

В этом скрипте мы просим 2 вычислительных узла (строка 3), на каждом по 4 MPI-процесса (строка 4), и по 8 ядер на один MPI-процесс (строка 5). В строках 9 и 22 вы удаляем все загруженные ранее модули, чтобы в строках 11 и 24 загрузить модули `intel impi` или `gnu9 openmpi4`. При использовании Intel MPI у нас есть два способа запустить гибридную программу: с помощью `mpirun` (строка 15), и с помощью `srun` (строка 19). При использовании OpenMPI запуск гибридных программ возможен с помощью `mpirun` (строка 28), также необходимо добавить дополнительный параметр `--map-by` (строка 32).

После запуска этого скрипта с помощью команды `sbatch` мы получим 4 файла, описывающих привязку OMP-thread'ов к ядрам вычислительных узлов. Пример файла `result.impi.mpirun` (отсортированный для удобства чтения):

```
Hello from rank 0, thread 0, on c01. (core affinity = 0-7)
Hello from rank 0, thread 1, on c01. (core affinity = 0-7)
Hello from rank 0, thread 2, on c01. (core affinity = 0-7)
Hello from rank 0, thread 3, on c01. (core affinity = 0-7)
Hello from rank 0, thread 4, on c01. (core affinity = 0-7)
Hello from rank 0, thread 5, on c01. (core affinity = 0-7)
Hello from rank 0, thread 6, on c01. (core affinity = 0-7)
Hello from rank 0, thread 7, on c01. (core affinity = 0-7)
Hello from rank 1, thread 0, on c01. (core affinity = 8-15)
Hello from rank 1, thread 1, on c01. (core affinity = 8-15)
Hello from rank 1, thread 2, on c01. (core affinity = 8-15)
Hello from rank 1, thread 3, on c01. (core affinity = 8-15)
Hello from rank 1, thread 4, on c01. (core affinity = 8-15)
Hello from rank 1, thread 5, on c01. (core affinity = 8-15)
Hello from rank 1, thread 6, on c01. (core affinity = 8-15)
Hello from rank 1, thread 7, on c01. (core affinity = 8-15)
Hello from rank 2, thread 0, on c01. (core affinity = 16-23)
Hello from rank 2, thread 1, on c01. (core affinity = 16-23)
Hello from rank 2, thread 2, on c01. (core affinity = 16-23)
Hello from rank 2, thread 3, on c01. (core affinity = 16-23)
Hello from rank 2, thread 4, on c01. (core affinity = 16-23)
Hello from rank 2, thread 5, on c01. (core affinity = 16-23)
Hello from rank 2, thread 6, on c01. (core affinity = 16-23)
Hello from rank 2, thread 7, on c01. (core affinity = 16-23)
Hello from rank 3, thread 0, on c01. (core affinity = 26-33)
Hello from rank 3, thread 1, on c01. (core affinity = 26-33)
Hello from rank 3, thread 2, on c01. (core affinity = 26-33)
Hello from rank 3, thread 3, on c01. (core affinity = 26-33)
Hello from rank 3, thread 4, on c01. (core affinity = 26-33)
Hello from rank 3, thread 5, on c01. (core affinity = 26-33)
Hello from rank 3, thread 6, on c01. (core affinity = 26-33)
Hello from rank 3, thread 7, on c01. (core affinity = 26-33)
Hello from rank 4, thread 0, on c02. (core affinity = 0-7)
Hello from rank 4, thread 1, on c02. (core affinity = 0-7)
Hello from rank 4, thread 2, on c02. (core affinity = 0-7)
Hello from rank 4, thread 3, on c02. (core affinity = 0-7)
Hello from rank 4, thread 4, on c02. (core affinity = 0-7)
Hello from rank 4, thread 5, on c02. (core affinity = 0-7)
Hello from rank 4, thread 6, on c02. (core affinity = 0-7)
Hello from rank 4, thread 7, on c02. (core affinity = 0-7)
Hello from rank 5, thread 0, on c02. (core affinity = 8-15)
Hello from rank 5, thread 1, on c02. (core affinity = 8-15)
Hello from rank 5, thread 2, on c02. (core affinity = 8-15)
Hello from rank 5, thread 3, on c02. (core affinity = 8-15)
Hello from rank 5, thread 4, on c02. (core affinity = 8-15)
Hello from rank 5, thread 5, on c02. (core affinity = 8-15)
Hello from rank 5, thread 6, on c02. (core affinity = 8-15)
Hello from rank 5, thread 7, on c02. (core affinity = 8-15)
Hello from rank 6, thread 0, on c02. (core affinity = 16-23)
Hello from rank 6, thread 1, on c02. (core affinity = 16-23)
Hello from rank 6, thread 2, on c02. (core affinity = 16-23)
Hello from rank 6, thread 3, on c02. (core affinity = 16-23)
Hello from rank 6, thread 4, on c02. (core affinity = 16-23)
Hello from rank 6, thread 5, on c02. (core affinity = 16-23)
Hello from rank 6, thread 6, on c02. (core affinity = 16-23)
Hello from rank 6, thread 7, on c02. (core affinity = 16-23)
Hello from rank 7, thread 0, on c02. (core affinity = 26-33)
Hello from rank 7, thread 1, on c02. (core affinity = 26-33)
Hello from rank 7, thread 2, on c02. (core affinity = 26-33)
Hello from rank 7, thread 3, on c02. (core affinity = 26-33)
Hello from rank 7, thread 4, on c02. (core affinity = 26-33)
Hello from rank 7, thread 5, on c02. (core affinity = 26-33)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Hello from rank 7, thread 6, on c02. (core affinity = 26-33)
Hello from rank 7, thread 7, on c02. (core affinity = 26-33)
```

Из вывода видно, что на каждом из узлов `c01` и `c02` запущено по 4 MPI-процесса, в каждом создано 8 OMP-thread'ов, и они привязаны к ядрам, выделенным на узлах блоками по 8 штук. Причём виден пропуск в блоках: `0-7, 8-15, 16-23, <пропуск>, 26-33`. Этот пропуск обусловлен выравниванием блоков по физическим процессорам. На каждом вычислительном узле установлены два процессора, первому соответствуют ядра `0-25`, второму – `26-51`. Так как мы запрашиваем на каждом узле по 4 блока по 8 ядер, то первые три блока попадают целиком на первый процессор, а четвёртый – на второй процессор.

Также стоит обратить внимание на вывод в файле `result.openmpi.mpirun`. Он сильно отличается от остальных, так как команда `mpirun` из библиотеки OpenMPI по умолчанию ограничивает каждый MPI-процесс одним ядром, поэтому в выводе мы не видим OMP-thread'ов:

```
Hello from rank 0, thread 0, on c01. (core affinity = 0)
Hello from rank 1, thread 0, on c01. (core affinity = 1)
Hello from rank 2, thread 0, on c01. (core affinity = 2)
Hello from rank 3, thread 0, on c01. (core affinity = 3)
Hello from rank 4, thread 0, on c02. (core affinity = 0)
Hello from rank 5, thread 0, on c02. (core affinity = 1)
Hello from rank 6, thread 0, on c02. (core affinity = 2)
Hello from rank 7, thread 0, on c02. (core affinity = 3)
```

Использование дополнительной опции `--map-by slot:PE=$SLURM_CPUS_PER_TASK` увеличивает количество ядер на каждый MPI-процесс, и результат `result.openmpi.map` получается аналогичным результату `result.impi.mpirun`.

Дополнительные опции команд `sbatch`, `srun`, `mpirun` могут влиять на конфигурацию выделенных ядер и на привязку процессов к ядрам. Обратитесь к документации Slurm, OpenMPI, Intel MPI за дополнительной информацией.