Параллельное программирование и суперкомпьютерный кодизайн

Смирнов А.В. asmirnov@srcc.msu.ru

Раздел 6. Параллельное программирование в распределенной памяти с использованием MPI

Раздел 6. План.

Параллельное программирование в распределенной памяти с использованием MPI

- Введение и простейшая программа
- Базовые приемы параллелизации с MPI
- Коммуникаторы и составные типы
- Распределенная работа с файлами
- Вариации попарного обмена в MPI
- Пример с ускорением для MPI – интегрирование

Раздел 6. План.

Параллельное программирование в распределенной памяти с использованием MPI

- **Введение и простейшая программа**
- Базовые приемы параллелизации с MPI
- Коммуникаторы и составные типы
- Распределенная работа с файлами
- Вариации попарного обмена в MPI
- Пример с ускорением для MPI – интегрирование

Message passing interface

MPI это

Библиотека MPI

Message passing interface

MPI это

- Библиотека для Си, Фортрана и иных языков

Библиотека MPI

Message passing interface

MPI это

- Библиотека для Си, Фортрана и иных языков
- Обертка над соответствующими компиляторами

Библиотека MPI

Message passing interface

MPI это

- Библиотека для Си, Фортрана и иных языков
- Обертка над соответствующими компиляторами
- Среда для запуска программ с использованием MPI

История MPI

Стандарт MPI определяет API (варианты для Си, C++, Fortran, Java)

- MPI 1.0 1995 год
- MPI 2.0 1998 год
- MPI 3.0 2012 год
- MPI 4.0 2021 год
- MPI 5.0 2025 год

История MPI

Стандарт MPI определяет API (варианты для Си, C++, Fortran, Java)

- MPI 1.0 1995 год
- MPI 2.0 1998 год появились C++ bindings
- MPI 3.0 2012 год убраны C++ bindings
- MPI 4.0 2021 год так и не вернулись C++ bindings
- MPI 5.0 2025 год так и не вернулись C++ bindings

Но все стандарты, включая разрабатываемый, увы, ориентированы на Си, а не C++!

Использование MPI

"Комплект поставки"MPI

- Библиотека
- Средства компиляции и запуска приложения

"Комплект поставки"MPI

- Библиотека
- Средства компиляции и запуска приложения

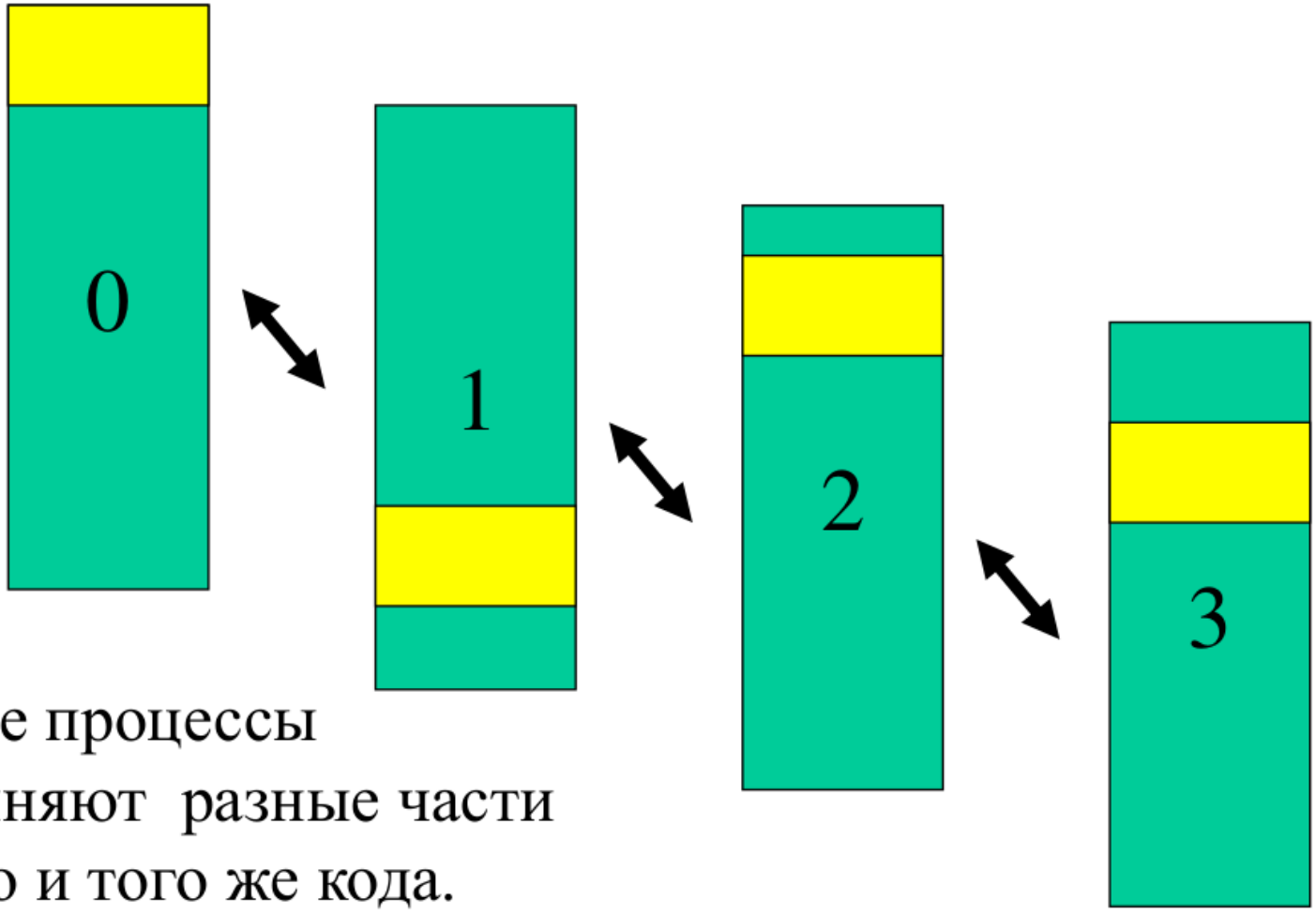
Существуют различные реализации API MPI, например, OpenMPI или MPICH. Принципиальных отличий по использованию быть не должно, но потребуется перекомпиляция программы при переходе от одного к другому (в MPI 5.0 не потребуется)

Установка MPI

- Для Ubuntu установка производится через `apt-get install openmpi-bin` или `apt-get install mpich`
- В результате вы получаете "компиляторы"mpicc и mpic++ и систему запуска MPI-приложений mpirun

SPMD-модель

- Используется концепция Single Program Multiple Data



Разные процессы выполняют разные части одного и того же кода.

MPI и потоки (OpenMP/threads)

Потоки (например, OpenMP)

- Потоки живут внутри одного процесса, делят общую память
- Обмен данными — через общие переменные, нужны блокировки и синхронизация

MPI и потоки (OpenMP/threads)

Потоки (например, OpenMP)

- Потоки живут внутри одного процесса, делят общую память
- Обмен данными — через общие переменные, нужны блокировки и синхронизация

Процессы MPI

- Отдельные процессы: у каждого свой стек, куча, глобальные переменные
- Обмен данными только через `MPI_Send`/`MPI_Recv` и коллективные операции

MPI и потоки (OpenMP/threads)

Потоки (например, OpenMP)

- Потоки живут внутри одного процесса, делят общую память
- Обмен данными — через общие переменные, нужны блокировки и синхронизация

Процессы MPI

- Отдельные процессы: у каждого свой стек, куча, глобальные переменные
- Обмен данными только через `MPI_Send`/`MPI_Recv` и коллективные операции
- Наиболее подходящее для распределенной памяти: кластер, несколько узлов, сеть

MPI и потоки (OpenMP/threads)

Потоки (например, OpenMP)

- Потоки живут внутри одного процесса, делят общую память
- Обмен данными — через общие переменные, нужны блокировки и синхронизация

Процессы MPI

- Отдельные процессы: у каждого свой стек, куча, глобальные переменные
- Обмен данными только через `MPI_Send`/`MPI_Recv` и коллективные операции
- Наиболее подходящее для распределенной памяти: кластер, несколько узлов, сеть
- В реальных приложениях часто используют гибридный подход: MPI по узлам, OpenMP/threads внутри узла

Использование MPI

Не просто стадо, а бешеное!

Простейшая программа с MPI

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello, World!\n");
    MPI_Finalize();
}
```

Простейшая программа с MPI

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello, World!\n");
    MPI_Finalize();
}
```

- Каждая программа должна вызвать `MPI_Init` и `MPI_MPI_Finalize` в каждом MPI варианте исполнения. У `MPI_Init` параметры int* и char*** (тройной указатель!)
- Откуда лишнее взятие указателей? Ответ будет после того, как покажем синтаксис запуска

Компиляция программы с MPI

- `mpic++ mpi.cpp -o mpi` – здесь mpic++ это не настоящий компилятор, а оболочка для g++

Компиляция программы с MPI

- `mpic++ mpi.cpp -o mpi` – здесь mpic++ это не настоящий компилятор, а оболочка для g++
- `which mpic++ -> /usr/bin/mpic++`

Компиляция программы с MPI

- `mpic++ mpi.cpp -o mpi` – здесь mpic++ это не настоящий компилятор, а оболочка для g++
- `which mpic++ -> /usr/bin/mpic++`
- `less /usr/bin/mpic++ ->`

```
 # mpicxx
# Simple script to compile and/or link MPI programs.
# This script knows the default flags and libraries, and can handle
# alternative C++ compilers and the associated flags and libraries.
# The important terms are:
#    includedir, libdir - Directories containing an *installed* mpich
#    prefix, execprefix - Often used to define includedir and libdir
#    CXX                - C compiler
#    WRAPPER_CXXFLAGS     - Any special flags needed to compile
#    WRAPPER_LDFLAGS      - Any special flags needed to link
#    WRAPPER_LIBS         - Any special libraries needed in order to link
#
# We assume that (a) the C++ compiler can both compile and link programs
...
```

Компиляция программы с MPI

- У `mpic++` есть параметр `-show`, если запустить с ним, то увидим реальную команду для g++

```
g++
-Wl,-Bsymbolic-functions -Wl,-z,relro mpi.cpp
-o mpi -I/usr/include/x86_64-linux-gnu/mpich
-L/usr/lib/x86_64-linux-gnu -lmpichcxx -lmpich
```

Компиляция программы с MPI

- У `mpic++` есть параметр `-show`, если запустить с ним, то увидим реальную команду для g++

```
g++
-Wl,-Bsymbolic-functions -Wl,-z,relro mpi.cpp
-o mpi -I/usr/include/x86_64-linux-gnu/mpich
-L/usr/lib/x86_64-linux-gnu -lmpichcxx -lmpich
```

- Поэтому я называю `mpic++` компилятором в кавычках, это лишь скрипт, который передает нужные параметры с реальный компилятор

Запуск программы с MPI

Нельзя просто так взять... ну вы знаете

Запуск программы с MPI

Нельзя просто так взять... ну вы знаете

- Программу скомпилированную с mpi можно запустить, но это вам не может дать никакой параллелизации.

Запуск программы с MPI

Нельзя просто так взять... ну вы знаете

- Программу скомпилированную с mpi можно запустить, но это вам не может дать никакой параллелизации.
- Правильный запуск `mpirun -np 4 ./mpi`, где 4 – число процессов, а `./mpi` – имя скомпилированной программы

Запуск программы с MPI

Нельзя просто так взять... ну вы знаете

- Программу скомпилированную с mpi можно запустить, но это вам не может дать никакой параллелизации.
- Правильный запуск `mpirun -np 4 ./mpi`, где 4 – число процессов, а `./mpi` – имя скомпилированной программы
- Такая команда запустит 4 практически идентичные копии вашей программы!

Запуск программы с MPI

Нельзя просто так взять... ну вы знаете

- Программу скомпилированную с mpi можно запустить, но это вам не может дать никакой параллелизации.
- Правильный запуск `mpirun -np 4 ./mpi`, где 4 – число процессов, а `./mpi` – имя скомпилированной программы
- Такая команда запустит 4 практически идентичные копии вашей программы!

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

Запуск программы с MPI

Нельзя просто так взять... ну вы знаете

- Программу скомпилированную с mpi можно запустить, но это вам не может дать никакой параллелизации.
- Правильный запуск `mpirun -np 4 ./mpi`, где 4 – число процессов, а `./mpi` – имя скомпилированной программы
- Такая команда запустит 4 практически идентичные копии вашей программы!

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

- Холмс, но как этим пользоваться?

И что делать с одинаковыми процессами?

И что делать с одинаковыми процессами?

- Процессы не совсем одинаковые, каждый из них может узнать свой номер, и этим можно воспользоваться при программировании
- Это программирование действительно весьма специфичное

И что делать с одинаковыми процессами?

- Процессы не совсем одинаковые, каждый из них может узнать свой номер, и этим можно воспользоваться при программировании
- Это программирование действительно весьма специфичное
- Но сначала давайте разберемся с аргументами

Простейшая программа с MPI

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    if (argc > 1) {
        printf("Hello with %s\n", argv[1]);
    }
    MPI_Finalize();
}
```

- Если эту программу запустить например как `./mpi blablabla`, то argc окажется равным 2, и программа напечатает `Hello with blablabla`

Простейшая программа с MPI

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    if (argc > 1) {
        printf("Hello with %s\n", argv[1]);
    }
    MPI_Finalize();
}
```

- Когда вы ее запускаете через `mpirun -np 4 ./mpi blablabla`, запускается не ваша программа, а программа `mpirun`, которая форкается и распадается на 4 процесса, каждый из которых в результате заменяется вашей программой. Но у вызова mpirun на 3 аргумента больше. Так вот вызов `MPI_Init` как раз отрезает лишнее и оставляет вашей программе ваши аргументы

И что делать с одинаковыми процессами?

Процессы MPI практически одинаковые, но они специальным MPI вызовом могут узнать свой номер

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

И что делать с одинаковыми процессами?

Процессы MPI практически одинаковые, но они специальным MPI вызовом могут узнать свой номер

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

И что делать с одинаковыми процессами?

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

```
I am 2 of 4
I am 3 of 4
I am 1 of 4
I am 0 of 4
```

И что делать с одинаковыми процессами?

Функции определения ранга и числа процессов

- `int MPI_Comm_size (MPI_Comm comm, int* size )`
  comm – коммуникатор
  size – число процессов
- `int MPI_Comm_rank(MPI_Comm comm, int* rank)`
  comm – коммуникатор
  rank – ранг процесса

Запуск MPI-программ на кластере

Локальный запуск

- `mpirun -np 4 ./mpi_program`
- все процессы на одной машине

Запуск MPI-программ на кластере

Локальный запуск

- `mpirun -np 4 ./mpi_program`
- все процессы на одной машине

Запуск на кластере

- процессы распределяются по нескольким узлам
- нужно указать:
  - сколько процессов
  - какие узлы (или сколько узлов)
  - как запускать через систему очередей

Запуск MPI-программ на кластере

Локальный запуск

- `mpirun -np 4 ./mpi_program`
- все процессы на одной машине

Запуск на кластере

- процессы распределяются по нескольким узлам
- нужно указать:
  - сколько процессов
  - какие узлы (или сколько узлов)
  - как запускать через систему очередей
- Сам код MPI не меняется — меняется только команда запуска

Скрипт для очереди (упрощённо)

Пример job.sh

```
#!/bin/bash
# Here specify number of nodes, cores, time, queues..

module load mpi

mpirun -np 16 ./mpi_program
```

Скрипт для очереди (упрощённо)

Пример job.sh

```
#!/bin/bash
# Here specify number of nodes, cores, time, queues...

module load mpi

mpirun -np 16 ./mpi_program
```

- Скрипт отправляется в очередь, например:

  ```
  sbatch job.sh    # or qsub job.sh, bsub job.sh, ...
  ```

- Система очередей выбирает свободные узлы и запускает ваш `mpirun` на этих узлах

Практические моменты на кластере

- Общая файловая система
  - код и данные обычно лежат на сетевом файловом хранилище
  - все процессы видят одинаковые пути (`/home/...` `/scratch/...`)

Практические моменты на кластере

- Общая файловая система
  - код и данные обычно лежат на сетевом файловом хранилище
  - все процессы видят одинаковые пути (`/home/...` `/scratch/...`)
- Ввод/вывод
  - `printf`/`std::cout` от **всех** процессов — это много
  - чаще всего печатают только с `rank == 0`
  - большие объёмы данных — писать в файлы, а не на экран

Практические моменты на кластере

- Общая файловая система
  - код и данные обычно лежат на сетевом файловом хранилище
  - все процессы видят одинаковые пути (`/home/...` `/scratch/...`)
- Ввод/вывод
  - `printf`/`std::cout` от **всех** процессов — это много
  - чаще всего печатают только с `rank == 0`
  - большие объёмы данных — писать в файлы, а не на экран
- Отладка
  - для учебных задач проще сначала отлаживать код на одной машине (`np = 2, 3, 4`)
  - потом переносить на кластер, минимально меняя команды запуска

Вопросы?

Раздел 6. План.

Параллельное программирование в распределенной памяти с использованием MPI

- Введение и простейшая программа
- **Базовые приемы параллелизации с MPI**
- Коммуникаторы и составные типы
- Распределенная работа с файлами
- Вариации попарного обмена в MPI
- Пример с ускорением для MPI – интегрирование

Задача сложения векторов

```
#include <mpi.h>
#include <stdio.h>
#include <vector>
#include <random>
#include <algorithm>
constexpr size_t n = 32 * 1024 * 1024;
std::vector<double> a(n);
std::vector<double> b(n);
std::vector<double> c(n);
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    std::random_device rd;
    std::mt19937 gen(rd());
```

Задача сложения векторов

```
    std::uniform_real_distribution<double>
        uid(-1., 1.);
    std::generate(a.begin(),a.end(),
      [&uid,&gen]()->double {return uid(gen);}
    );
    std::generate(b.begin(),b.end(),
      [&uid,&gen]()->double {return uid(gen);}
    );
    std::generate(c.begin(),c.end(),
      [&uid,&gen]()->double {return 0.;}
    );
    printf("a[0] is %lf\n", a[0]);
    MPI_Finalize();
    return 0;
}
```

Задача сложения векторов

```
a[0] is -0.042394
a[0] is -0.348875
a[0] is -0.600241
a[0] is 0.521050
```

- У нас разные процессы, и у них вектора разные!

Задача сложения векторов

```
a[0] is -0.042394
a[0] is -0.348875
a[0] is -0.600241
a[0] is 0.521050
```

- У нас разные процессы, и у них вектора разные!
- Да, мы могли бы одинаково инициализировать счетчик случайных чисел и добиться того, чтобы вектора были одинаковыми, но это не решает общую задачу. В реальной задаче вектор может быть наполнен на одной ноде (например, считан из базы), и либо все ноды должны ее читать, либо этот вектор как-то размножить по нодам. Кроме того, после того, как все узлы сделают свою работу, надо как-то собрать результаты в одном месте – у нас нет общей памяти!

Разработка программы на MPI

Стандартным подходом к MPI-программированию является выделение одного экземпляра программы как главного

Разработка программы на MPI

Стандартным подходом к MPI-программированию является выделение одного экземпляра программы как главного

- По традиции главным узлом считается узел с номером 0

Разработка программы на MPI

Стандартным подходом к MPI-программированию является выделение одного экземпляра программы как главного

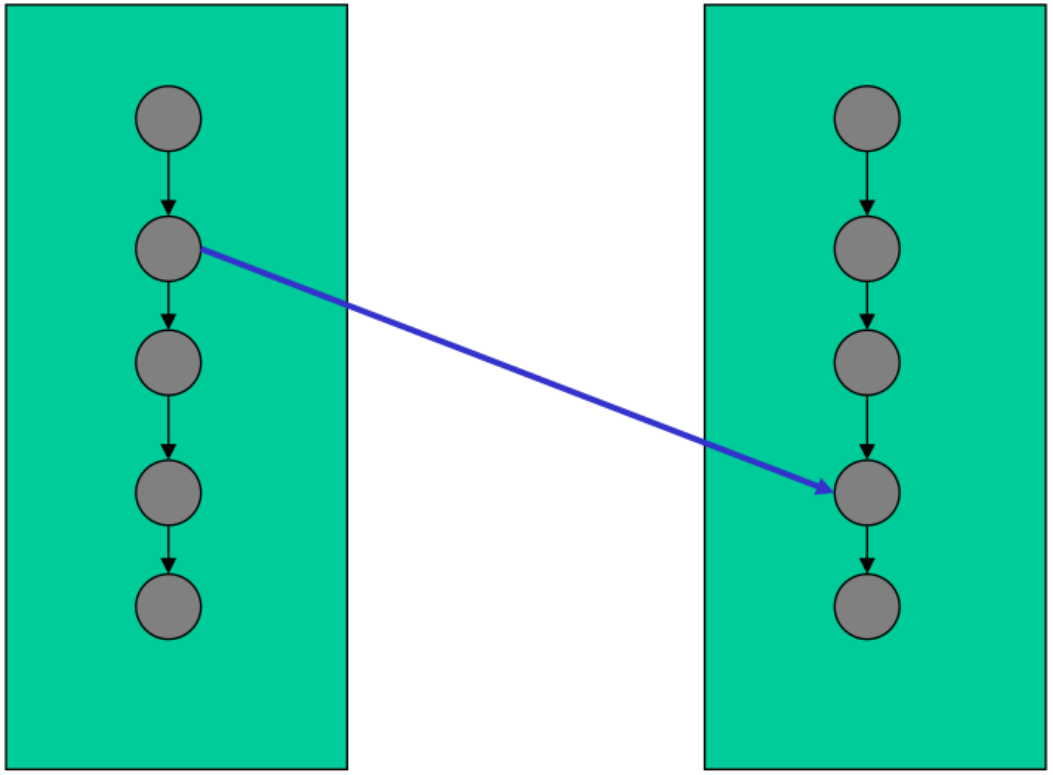
- По традиции главным узлом считается узел с номером 0
- Главный узел будет генерировать вектор, раздавать данные, а затем собирать результаты.

Разработка программы на MPI

Стандартным подходом к MPI-программированию является выделение одного экземпляра программы как главного

- По традиции главным узлом считается узел с номером 0
- Главный узел будет генерировать вектор, раздавать данные, а затем собирать результаты.
- Чтобы написать программу, нам потребуется обсудить синтаксис передачи данных

Назначение попарных взаимодействий



Простая программа с передачей данных

```
int main(int argc, char* argv[]) {
    int rank; MPI_Status st; char buf[64];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0) {
        sprintf(buf,"Hello from processor 0");
        MPI_Send(buf, 64, MPI_CHAR, 1, 0,
            MPI_COMM_WORLD);
    } else {
        MPI_Recv(buf, 64, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD, &st);
        printf("Process %d received %s \n",
            rank, buf);
    }
    MPI_Finalize();
}
```

Простая программа с передачей данных

- Если запустить эту программу с `mpirun -np 2`, то она выведет

```
Process 1 received Hello from processor 0
```

Простая программа с передачей данных

- Если запустить эту программу с `mpirun -np 2`, то она выведет

```
Process 1 received Hello from processor 0
```

- Что будет, если ее запустить с 4 процессами?

Простая программа с передачей данных

- Если запустить эту программу с `mpirun -np 2`, то она выведет

```
Process 1 received Hello from processor 0
```

- Что будет, если ее запустить с 4 процессами?
- Программа зависнет!

Функции попарных обменов

```
int MPI_Send(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm )
```

- buf – адрес начала буфера посылаемых данных
- count – число пересылаемых объектов типа, соответствующего datatype
- datatype – MPI-тип принимаемых данных
- dest – номер процесса-приемника
- tag – уникальный тэг, идентифицирующий сообщение
- comm – коммуникатор

Функции попарных обменов

```
int MPI_Recv(void* buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Status * status)
```

- buf – адрес буфера для приема сообщения
- count – максимальное число объектов типа datatype, которое может быть записано в буфер
- datatype – MPI-тип принимаемых данных
- source – номер процесса, от которого ожидается сообщение или `MPI_ANY_SOURCE`
- tag – уникальный тэг, идентифицирующий сообщение или `MPI_ANY_TAG`
- comm – коммуникатор
- status – статус завершения

Функции попарных обменов

```
typedef struct
{
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

- MPI_SOURCE – ранг процесса-передатчика данных
- MPI_TAG – тэг сообщения
- MPI_ERROR – код ошибки

Задача сложения векторов

```
#include <mpi.h>
#include <stdio.h>
#include <vector>
#include <random>
#include <algorithm>

constexpr size_t n = 32 * 1024 * 1024;
std::vector<double> a(n);
std::vector<double> b(n);
std::vector<double> c(n);

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Задача сложения векторов

```
if (size <= 1) {
    printf("Not enough MPI processes\n");
    MPI_Finalize();
    return 0;
}
if (rank == 0) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double>
      uid(-1., 1.);
    std::generate(a.begin(),a.end(),
      [&uid,&gen]()->double {return uid(gen);}
    );
    std::generate(b.begin(),b.end(),
      [&uid,&gen]()->double {return uid(gen);}
    );
```

Задача сложения векторов

```
        std::generate(c.begin(),c.end(),
          [&uid,&gen]()->double {return 0.;}
        );
        for (int i = 1; i != size; ++i) {
          MPI_Send(a.data(), n, MPI_DOUBLE, i, 0,
            MPI_COMM_WORLD);
        }
    } else {
        MPI_Status st;
        MPI_Recv(a.data(), n, MPI_DOUBLE, 0, 0,
          MPI_COMM_WORLD, &st);
    }
    printf("a[0] is %lf\n", a[0]);
    MPI_Finalize();
    return 0;
}
```

Задача сложения векторов

Вот теперь вектора a одинаковые во всех процессах

Остается

Задача сложения векторов

Вот теперь вектора a одинаковые во всех процессах

Остается

- Разослать вектора b (аналогично)

Задача сложения векторов

Вот теперь вектора a одинаковые во всех процессах

Остается

- Разослать вектора b (аналогично)
- Осуществить суммирование

Задача сложения векторов

Вот теперь вектора a одинаковые во всех процессах

Остается

- Разослать вектора b (аналогично)
- Осуществить суммирование
- Собрать вектор c из частей на основном процессе

Задача сложения векторов (main)

```
if (rank == 0) {
    // vector generation here
    for (int i = 1; i != size; ++i) {
        MPI_Send(a.data(), n, MPI_DOUBLE, i, 0,
            MPI_COMM_WORLD);
        MPI_Send(b.data(), n, MPI_DOUBLE, i, 0,
            MPI_COMM_WORLD);
    }
} else {
    MPI_Status st;
    MPI_Recv(a.data(), n, MPI_DOUBLE, 0, 0,
        MPI_COMM_WORLD, &st);
    MPI_Recv(b.data(), n, MPI_DOUBLE, 0, 0,
        MPI_COMM_WORLD, &st);
}
```

Задача сложения векторов (main)

```
for (size_t j = (rank * n) / size;
   j != ((rank + 1) * n) / size; ++j) {
     c[j] = a[j] + b[j];
}
```

Задача сложения векторов (main)

```
if (rank == 0) {
  for (int i = 1; i != size; ++i) {
    MPI_Status st;
    MPI_Recv(c.data() + ((i) * n) / size,
      ((i + 1) * n) / size - ((i) * n) / size,
      MPI_DOUBLE, i, 0,
      MPI_COMM_WORLD, &st);
  }
} else {
  MPI_Send(c.data() + ((rank) * n) / size,
    ((rank + 1) * n) / size -
    ((rank) * n) / size,
    MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
```

Задача сложения векторов

У кода выше есть пара недостатков

Задача сложения векторов

У кода выше есть пара недостатков

- Мы рассылали вектора a и b целиком на другие процессы, но складывалась там лишь часть

Задача сложения векторов

У кода выше есть пара недостатков

- Мы рассылали вектора a и b целиком на другие процессы, но складывалась там лишь часть
- Мы рассылали их в цикле, однако у MPI есть специальные функции для массовой рассылки и сбора информации

Функции массовых обменов

```
int MPI_Bcast(void* buf, int count, MPI_Datatype
datatype, int root, MPI_Comm comm )
```

- buf – адрес начала буфера посылаемых данных
- count – число пересылаемых объектов типа, соответствующего datatype
- datatype – MPI-тип передаваемых данных
- root – номер процесса, посылающего данные
- comm – коммуникатор

Функции массовых обменов

```
int MPI_Bcast(void* buf, int count, MPI_Datatype
datatype, int root, MPI_Comm comm )
```

- buf – адрес начала буфера посылаемых данных
- count – число пересылаемых объектов типа, соответствующего datatype
- datatype – MPI-тип передаваемых данных
- root – номер процесса, посылающего данные
- comm – коммуникатор

Внимание! Функцию `MPI_Bcast` должны вызывать все процессы в коммуникаторе, как отправитель, так и получатели!

Задача сложения векторов (код рассылки)

Заменяем этот код на:

```
if (rank == 0) {
    // vector generation here
}
MPI_Bcast(a.data(), n, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
MPI_Bcast(b.data(), n, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
```

Задача сложения векторов (код рассылки)

Заменяем этот код на:

```
if (rank == 0) {
    // vector generation here
}
MPI_Bcast(a.data(), n, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
MPI_Bcast(b.data(), n, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
```

Синтаксис мы упростили, однако все равно каждый вектор пересылается каждому процессу

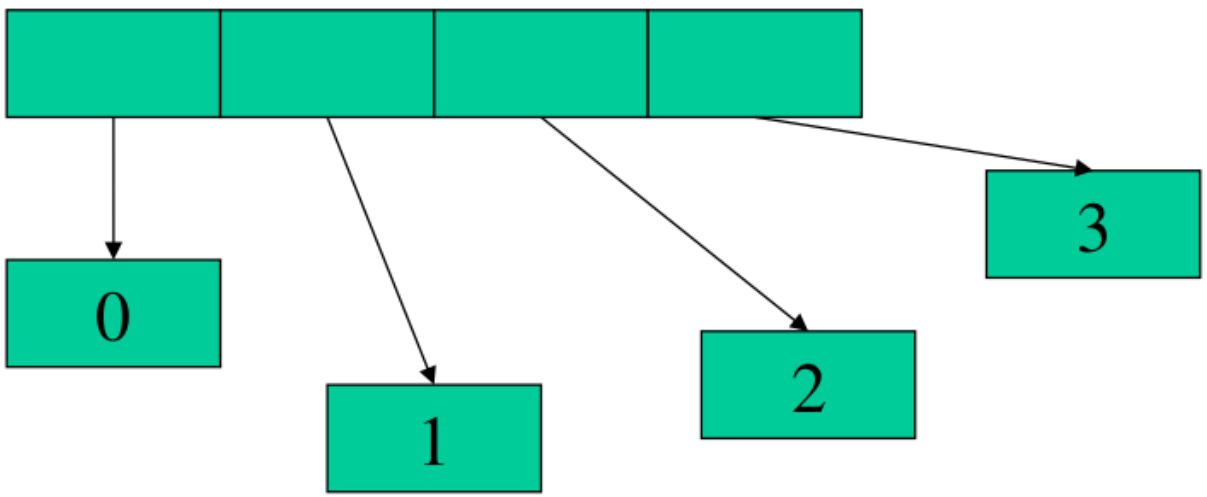
Функции массовых обменов

```
int MPI_Scatter (void* sendbuf, int sendcnt,
MPI_Datatype sendtype, void * recvbuf, int
recvcnt, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

- sendbuf – адрес начала буфера посылаемых данных
- sendcnt – число пересылаемых объектов типа, соответствующего sendtype
- sendtype – MPI-тип передаваемых данных
- recvbuf – адрес начала буфера получаемых данных
- recvcnt – число получаемых объектов типа, соответствующего recvtype
- recvtype – MPI-тип получаемых данных
- root – номер процесса, посылающего данные
- comm – коммуникатор

Функции массовых обменов

```
int MPI_Scatter (void* sendbuf, int sendcnt,
MPI_Datatype sendtype, void * recvbuf, int
recvcnt, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```



Функции массовых обменов

```
int MPI_Scatter (void* sendbuf, int sendcnt,
MPI_Datatype sendtype, void * recvbuf, int
recvcnt, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

Внимание! Функцию `MPI_Scatter` должны вызывать все процессы в коммуникаторе, как отправитель, так и получатели! При этом на головном процессе важны все параметры, поскольку он посылает данные и себе, а на остальных send... не важны. Также sendtype должно совпадать с recvtype, а sendcnt с recvcnt

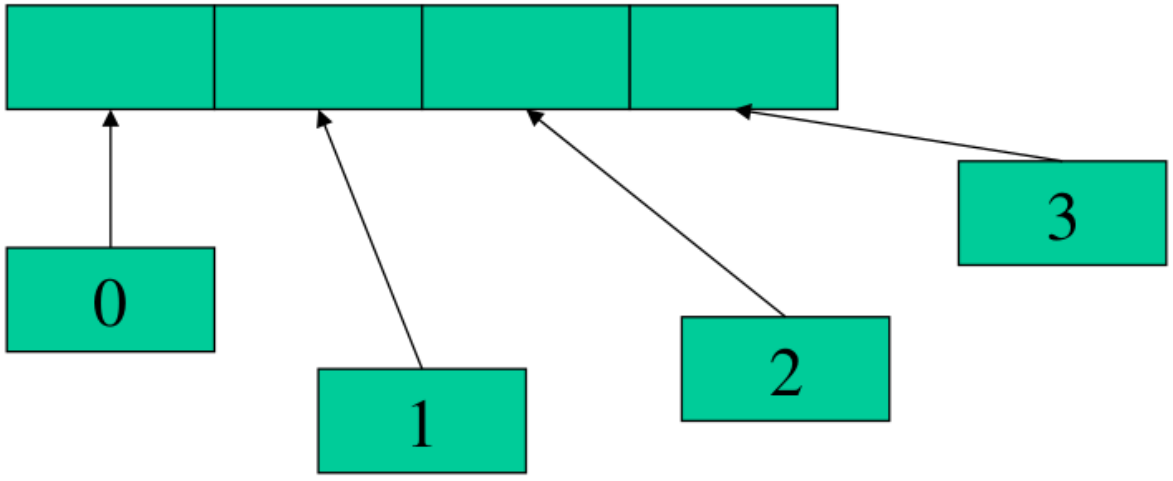
Функции массовых обменов

```
int MPI_Gather (void* sendbuf, int sendcnt,
MPI_Datatype sendtype, void * recvbuf, int
recvcnt, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

- sendbuf – адрес начала буфера посылаемых данных
- sendcnt – число пересылаемых объектов типа, соответствующего sendtype
- sendtype – MPI-тип передаваемых данных
- recvbuf – адрес начала буфера посылаемых данных
- recvcnt – число получаемых объектов типа, соответствующего recvtype
- recvtype – MPI-тип получаемых данных
- root – номер процесса, получающего данные
- comm – коммуникатор

Функции массовых обменов

```
int MPI_Gather (void* sendbuf, int sendcnt,
MPI_Datatype sendtype, void * recvbuf, int
recvcnt, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```



Функции массовых обменов

```
int MPI_Gather (void* sendbuf, int sendcnt,
MPI_Datatype sendtype, void * recvbuf, int
recvcnt, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

Внимание! Функцию `MPI_Gather` должны вызывать все процессы в коммуникаторе, как отправители, так и получатель! При этом на головном процессе важны все параметры, поскольку он посылает данные и себе, а на остальных recv... не важны. Также sendtype должно совпадать с recvtype, а sendcnt с recvcnt

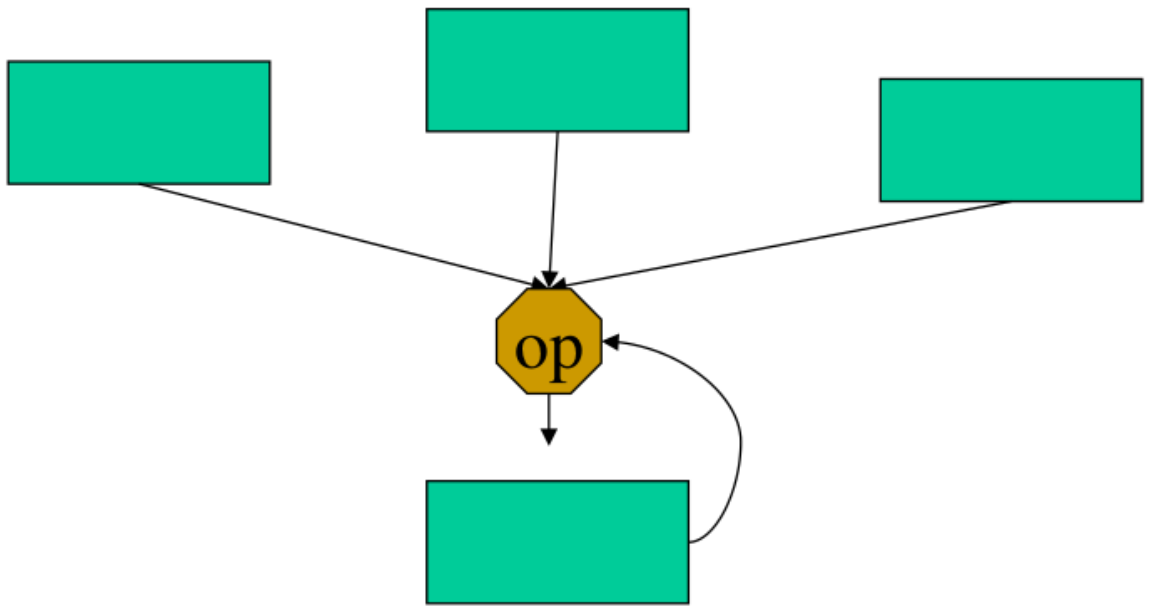
Функции массовых обменов

```
int MPI_Reduce (void* sendbuf, void * recvbuf,
int count, MPI_Datatype datatype, MPI_Op op, int
root, MPI_Comm comm)
```

- sendbuf – буфер операндов
- recvbuf – буфер приема
- count – число пересылаемых объектов типа, соответствующего sendtype
- datatype – MPI-тип данных
- op – операция, например MPI_Sum
- root – номер процесса, получающего данные
- comm – коммуникатор

Функции массовых обменов

```
int MPI_Reduce (void* sendbuf, void * recvbuf,
int count, MPI_Datatype datatype, MPI_Op op, int
root, MPI_Comm comm)
```



Функции массовых обменов

```
int MPI_Reduce (void* sendbuf, void * recvbuf,
int count, MPI_Datatype datatype, MPI_Op op, int
root, MPI_Comm comm)
```

Внимание! Функцию `MPI_Reduce` должны вызывать все процессы в коммуникаторе включая головной. Размеры sendbuf и recvbuf должны совпадать

Задача сложения векторов (main)

```
if (rank == 0) {
    // vector generation here
}
MPI_Scatter(a.data(), n / size, MPI_DOUBLE,
  rank ?
    a.data() + ((rank) * n) / size :
    MPI_IN_PLACE,
  n / size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(b.data(), n / size, MPI_DOUBLE,
  rank ?
    b.data() + ((rank) * n) / size :
    MPI_IN_PLACE,
  n / size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Задача сложения векторов (main)

```
for (size_t j = (rank * n) / size;
   j != ((rank + 1) * n) / size; ++j) {
      c[j] = a[j] + b[j];
}
MPI_Gather(
   rank ? c.data() + ((rank) * n) / size :
      MPI_IN_PLACE,
   n / size, MPI_DOUBLE, c.data(), n / size,
   MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Задача сложения векторов

Теперь лишних пересылок нет, но мы кое-что не учли

Задача сложения векторов

Теперь лишних пересылок нет, но мы кое-что не учли

- n может не делиться на size без остатка, и тогда последние элементы останутся не сложенными

Задача сложения векторов

Теперь лишних пересылок нет, но мы кое-что не учли

- n может не делиться на size без остатка, и тогда последние элементы останутся не сложенными
- Один вариант это "лишние"элементы посчитать на основном процессе

Задача сложения векторов

Теперь лишних пересылок нет, но мы кое-что не учли

- n может не делиться на size без остатка, и тогда последние элементы останутся не сложенными
- Один вариант это "лишние"элементы посчитать на основном процессе
- Другой – использовать функции `MPI_Scaterv` и `MPI_Gatherv`, позволяющие задавать разное количество элементов пересылаемое на разные процессы. Для этого предварительно нужно создать массив из этих длин. Для тривиальной задачи сложения векторов это точно было бы бессмыслицей, поскольку больше времени ушло бы на заполнение такого вектора.

Произведем замер времени

```
mpirun -np 1 ./mpi
48 milliseconds
mpirun -np 2 ./mpi
102 milliseconds
mpirun -np 3 ./mpi
168 milliseconds
mpirun -np 4 ./mpi
253 milliseconds
```

Накладные расходы на обмен сообщениями весьма высоки! Если при обычной параллелизации мы могли максимум не получить ускорение для сложения векторов, то здесь мы получаем существенное замедление! Нужно выбирать правильно задачи и как можно меньше обмениваться сообщениями!

Схемы распределения данных: блочное разбиение

Идея блочного разбиения

- Есть вектор из n элементов
- Есть `size` процессов MPI
- Каждый процесс получает непрерывный блок индексов

Схемы распределения данных: блочное разбиение

Идея блочного разбиения

- Есть вектор из n элементов
- Есть `size` процессов MPI
- Каждый процесс получает непрерывный блок индексов
- Процесс с рангом `rank` обрабатывает диапазон

$$j = \frac{\texttt{rank} \cdot n}{\texttt{size}}, \ \ldots, \ \frac{(\texttt{rank} + 1) \cdot n}{\texttt{size}} - 1$$

- Хорошо работает при равномерной нагрузке: каждый элемент вектора «стоит» примерно одинаково по времени

Идея циклического разбиения

- Вместо одного непрерывного блока процесс получает элементы через шаг
- Процесс с рангом `rank` обрабатывает индексы:

$$j = \texttt{rank},\ \texttt{rank} + \texttt{size},\ \texttt{rank} + 2 \cdot \texttt{size},\ \ldots$$

- Полезно при неравномерной нагрузке: разные элементы вектора требуют разного времени
- Нагрузка выравнивается автоматически, без сложного планировщика

Попробуем сложение элементов в векторе

```
constexpr size_t n = 128 * 1024 * 1024;
std::vector<double> a(n);
MPI_Scatter(a.data(), n / size, MPI_DOUBLE,
      rank ? a.data() + (rank * (n / size)) :
      MPI_IN_PLACE,
      n / size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (size_t j = rank * (n / size);
      j != (rank + 1) * (n / size); ++j) {
      res += a[j];
}
double fres = 0.;
MPI_Reduce(&res, &fres, 1, MPI_DOUBLE,
   MPI_SUM, 0, MPI_COMM_WORLD);
```

Произведем замер времени

```
mpirun -np 1 ./mpi
125 milliseconds
mpirun -np 2 ./mpi
158 milliseconds
mpirun -np 3 ./mpi
226 milliseconds
mpirun -np 4 ./mpi
324 milliseconds
```

Накладные расходы на обмен сообщениями весьма высоки! Если при обычной параллелизации мы для этой задачи получали существенное ускорение, здесь по-прежнему существенное замедление! Нужно выбирать правильно задачи и как можно меньше обмениваться сообщениями!

Раздел 6. План.

Параллельное программирование в распределенной памяти с использованием MPI

- Введение и простейшая программа
- Базовые приемы параллелизации с MPI
- **Коммуникаторы и составные типы**
- Распределенная работа с файлами
- Вариации попарного обмена в MPI
- Пример с ускорением для MPI – интегрирование

Коммуникаторы в MPI

Коммуникатор

- Логическая группа процессов + правила коммуникации между ними
- Почти все функции MPI принимают аргумент `MPI_Comm comm`
- До этого мы использовали только `MPI_COMM_WORLD`

Коммуникаторы в MPI

Коммуникатор

- Логическая группа процессов + правила коммуникации между ними
- Почти все функции MPI принимают аргумент `MPI_Comm comm`
- До этого мы использовали только `MPI_COMM_WORLD`
- `MPI_COMM_WORLD` — «весь мир» MPI-процессов

Коммуникаторы в MPI

Коммуникатор

- Логическая группа процессов + правила коммуникации между ними
- Почти все функции MPI принимают аргумент `MPI_Comm comm`
- До этого мы использовали только `MPI_COMM_WORLD`
- `MPI_COMM_WORLD` — «весь мир» MPI-процессов
- Можно создавать свои коммуникаторы:
  - подзадача только на части процессов
  - разные уровни параллелизма

Коммуникаторы в MPI

Коммуникатор

- Логическая группа процессов + правила коммуникации между ними
- Почти все функции MPI принимают аргумент `MPI_Comm comm`
- До этого мы использовали только `MPI_COMM_WORLD`
- `MPI_COMM_WORLD` — «весь мир» MPI-процессов
- Можно создавать свои коммуникаторы:
  - подзадача только на части процессов
  - разные уровни параллелизма
- Ранг процесса всегда относится к конкретному коммуникатору (rank в `MPI_COMM_WORLD` и rank в новом коммуникаторе могут различаться)

Пример: разбиение мира на две группы

Разделим процессы на чётные и нечётные

- В исходном коммуникаторе `MPI_COMM_WORLD` есть процессы с рангами `0..size-1`
- Создадим два новых коммуникатора:
  - один для чётных рангов
  - другой для нечётных рангов

Пример: разбиение мира на две группы

```
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank % 2;
MPI_Comm subcomm;
MPI_Comm_split(MPI_COMM_WORLD, color,
               world_rank, &subcomm);

int sub_rank, sub_size;
MPI_Comm_rank(subcomm, &sub_rank);
MPI_Comm_size(subcomm, &sub_size);
```

Зачем нужны подкоммуникаторы

MPI_Comm_split в примере

- `color` определяет, в какую группу попадёт процесс
- `key` (у нас `world_rank`) определяет порядок рангов внутри новой группы
- В результате у нас два коммуникатора, условно:
  - `subcomm_even` с рангами `0,2,4,...`
  - `subcomm_odd` с рангами `1,3,5,...`

Зачем нужны подкоммуникаторы

MPI_Comm_split в примере

- `color` определяет, в какую группу попадёт процесс
- `key` (у нас `world_rank`) определяет порядок рангов внутри новой группы
- В результате у нас два коммуникатора, условно:
  - `subcomm_even` с рангами `0,2,4,...`
  - `subcomm_odd` с рангами `1,3,5,...`
- Внутри каждого подкоммуникатора можно вызывать:
  - `MPI_Bcast`, `MPI_Reduce` и т.д.
  - point-to-point операции

Зачем нужны подкоммуникаторы

MPI_Comm_split в примере

- `color` определяет, в какую группу попадёт процесс
- `key` (у нас `world_rank`) определяет порядок рангов внутри новой группы
- В результате у нас два коммуникатора, условно:
  - `subcomm_even` с рангами `0,2,4,...`
  - `subcomm_odd` с рангами `1,3,5,...`
- Внутри каждого подкоммуникатора можно вызывать:
  - `MPI_Bcast`, `MPI_Reduce` и т.д.
  - point-to-point операции
- Примеры применения:
  - разные этапы алгоритма выполняют разные группы процессов
  - многоуровневые разбиения домена (по осям, по блокам и т.п.)

MPI-типы: зачем они нужны

Базовые типы

- `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`, `MPI_FLOAT`, ...
- Используются во всех `MPI_Send`/`MPI_Recv`, `MPI_Bcast`, `MPI_Reduce`, ...

MPI-типы: зачем они нужны

Базовые типы

- `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`, `MPI_FLOAT`, ...
- Используются во всех `MPI_Send`/`MPI_Recv`, `MPI_Bcast`, `MPI_Reduce`, ...

Зачем нужны «сложные» типы

- Хочется пересылать структуры, а не голые массивы
- Хочется описывать произвольные паттерны памяти (например, каждый k-й элемент, столбец матрицы и т.п.)
- Не хочется вручную городить массивы `MPI_Send` или `memcpy` в промежуточные буферы

MPI-типы: зачем они нужны

Базовые типы

- `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`, `MPI_FLOAT`, ...
- Используются во всех `MPI_Send`/`MPI_Recv`, `MPI_Bcast`, `MPI_Reduce`, ...

Зачем нужны «сложные» типы

- Хочется пересылать структуры, а не голые массивы
- Хочется описывать произвольные паттерны памяти (например, каждый k-й элемент, столбец матрицы и т.п.)
- Не хочется вручную городить массивы `MPI_Send` или `memcpy` в промежуточные буферы

- Для этого в MPI есть семейство функций `MPI_Type_*`
- И есть возможность упаковывать данные в линейный буфер с помощью `MPI_Pack`/`MPI_Unpack`

MPI_Type_contiguous

Идея

- Создать новый тип как «N подряд идущих элементов базового типа»
- Пример: блок из 3 `double` как один элемент MPI-типа

MPI_Type_contiguous

Идея

- Создать новый тип как «N подряд идущих элементов базового типа»
- Пример: блок из 3 `double` как один элемент MPI-типа

```
MPI_Datatype triple_t;
MPI_Type_contiguous(3, MPI_DOUBLE, &triple_t);
MPI_Type_commit(&triple_t);

// sending triples
MPI_Send(buf, n_triples, triple_t, dest, tag,
         MPI_COMM_WORLD);
```

MPI_Type_contiguous

Идея

- Создать новый тип как «N подряд идущих элементов базового типа»
- Пример: блок из 3 `double` как один элемент MPI-типа

```
MPI_Datatype triple_t;
MPI_Type_contiguous(3, MPI_DOUBLE, &triple_t);
MPI_Type_commit(&triple_t);

// sending triples
MPI_Send(buf, n_triples, triple_t, dest, tag,
         MPI_COMM_WORLD);
```

На стороне приёмника:

```
MPI_Recv(buf, n_triples, triple_t, source, tag,
         MPI_COMM_WORLD, &status);
```

MPI_Pack и MPI_Unpack

Когда это нужно

- Когда данные в памяти лежат в «сломанном» формате
- Когда нужно в одном сообщении передать смесь типов: `int + double + массив char` и т.п.

MPI_Pack и MPI_Unpack

Когда это нужно

- Когда данные в памяти лежат в «сломанном» формате
- Когда нужно в одном сообщении передать смесь типов: `int + double + массив char` и т.п.

```
int i = 42;
double x = 3.14;
char buf[128]; int pos = 0;
MPI_Pack(&i, 1, MPI_INT, buf,
  sizeof(buf), &pos, MPI_COMM_WORLD);
MPI_Pack(&x, 1, MPI_DOUBLE, buf,
 sizeof(buf), &pos, MPI_COMM_WORLD);
// one send to rule them all
MPI_Send(buf, pos, MPI_PACKED, dest, tag,
  MPI_COMM_WORLD);
```

MPI_Unpack на стороне приёмника

```
MPI_Status st;
MPI_Recv(buf, 128, MPI_PACKED, source, tag,
         MPI_COMM_WORLD, &st);
MPI_Unpack(buf, 128, &pos, &i, 1, MPI_INT,
           MPI_COMM_WORLD);
MPI_Unpack(buf, 128, &pos, &x, 1, MPI_DOUBLE,
           MPI_COMM_WORLD);
```

MPI_Unpack на стороне приёмника

```
MPI_Status st;
MPI_Recv(buf, 128, MPI_PACKED, source, tag,
         MPI_COMM_WORLD, &st);
MPI_Unpack(buf, 128, &pos, &i, 1, MPI_INT,
           MPI_COMM_WORLD);
MPI_Unpack(buf, 128, &pos, &x, 1, MPI_DOUBLE,
           MPI_COMM_WORLD);
```

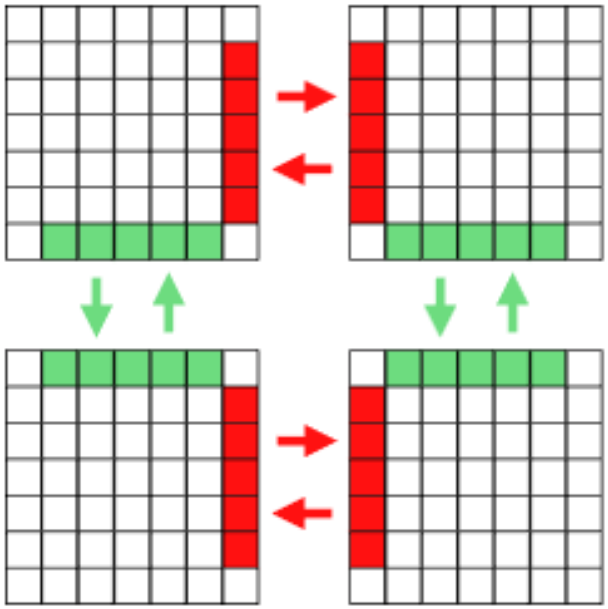
- Порядок `Unpack` должен совпадать с порядком Pack
- Типы тоже должны совпадать (`INT` с `INT`, `DOUBLE` с `DOUBLE`, ...)
- Тип сообщения в `MPI_Recv` — `MPI_PACKED`

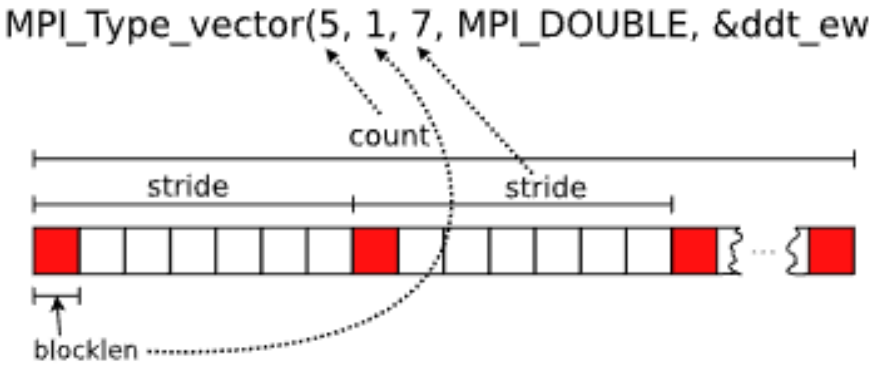
Альтернативы MPI_Type_contiguous

MPI_Type_vector

- Создаёт тип с регулярными «шагами» в памяти

```
MPI_Datatype col_t;
MPI_Type_vector(count, blocklength, stride,
                MPI_DOUBLE, &col_t);
MPI_Type_commit(&col_t);
```





Сводная таблица MPI пользовательских типов

Тип MPI	**Назначение**
MPI_Type_contiguous	Последовательные элементы
MPI_Type_vector	Одинаковый шаг между блоками
MPI_Type_indexed	Нерегулярные смещения
MPI_Type_create_struct	Структуры с разными типами/размерами
MPI_Pack / MPI_Unpack	Ручная упаковка данных, позволяет пересылать произвольные данные без создания типа

Раздел 6. План.

Параллельное программирование в распределенной памяти с использованием MPI

- Введение и простейшая программа
- Базовые приемы параллелизации с MPI
- Коммуникаторы и составные типы
- **Распределенная работа с файлами**
- Вариации попарного обмена в MPI
- Пример с ускорением для MPI – интегрирование

Распределённые файловые системы (DFS)

- Обычная ФС: данные на одном диске, доступ ограничен одним узлом.
- DFS: множество узлов используют общие данные одновременно.
- Основные цели:
  - единое пространство имён для всех узлов
  - параллельный доступ к файлам
  - масштабируемость по объёму и пропускной способности

Распределённые файловые системы (DFS)

- Обычная ФС: данные на одном диске, доступ ограничен одним узлом.
- DFS: множество узлов используют общие данные одновременно.
- Основные цели:
  - единое пространство имён для всех узлов
  - параллельный доступ к файлам
  - масштабируемость по объёму и пропускной способности

Принципы работы DFS

- Файлы разбиваются на крупные блоки (**страйпы**) и распределяются по нескольким серверам данных
- Отдельные серверы хранят метаданные: структура каталогов, права, расположение страйпов
- Клиенты видят DFS как обычную файловую систему, но доступ к данным идёт параллельно

Файловая система Lustre

- **MDS (Metadata Server)** хранит метаданные файлов и каталогов
- **OSS (Object Storage Server)** хранит данные в виде объектов
- **OST (Object Storage Target)** физические устройства хранения (RAID, диск)
- Клиент Lustre:
  - обращается к MDS за информацией о расположении данных
  - читает/пишет данные напрямую на OST

Файловая система Lustre

- **MDS (Metadata Server)** хранит метаданные файлов и каталогов
- **OSS (Object Storage Server)** хранит данные в виде объектов
- **OST (Object Storage Target)** физические устройства хранения (RAID, диск)
- Клиент Lustre:
  - обращается к MDS за информацией о расположении данных
  - читает/пишет данные напрямую на OST

Страйпинг

- Файл разбивается на полосы (страйпы) и разносится по OST
- Процессы MPI могут параллельно читать/писать разные части файла
- Размер страйпа и число OST определяют эффективность параллельного доступа (lfs setstripe)

Общая канва замера времени

Шаблон замера времени

```
MPI_Barrier(MPI_COMM_WORLD);
double t0 = MPI_Wtime();

read_file(...);  // read file (in separate function)

MPI_Barrier(MPI_COMM_WORLD);
double t1 = MPI_Wtime();

// compute maximum time among all processes
double tmax;
MPI_Reduce(&t1 - t0, &tmax, 1, MPI_DOUBLE,
  MPI_MAX, 0, MPI_COMM_WORLD);

if(rank == 0) printf("Max time = %f\n", tmax);
```

Чтение файла с POSIX

Код чтения файла

```
FILE *f = fopen("data.bin", "rb");
// open file
fread(buffer, 1, size, f);
// read entire file into buffer
fclose(f);
// close file
```

- Каждый процесс читает файл независимо.

MPI-IO

- `MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)`
  - Коллективное открытие файла всеми процессами в коммуникаторе.
  - `amode`: `MPI_MODE_RDONLY`, `MPI_MODE_WRONLY`, `MPI_MODE_RDWR` и др.
- `MPI_File_get_size(MPI_File fh, MPI_Offset *size)`
  - Возвращает общий размер файла (в байтах).
- `MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype dtype, MPI_Status *status)`
  - Коллективное чтение: все процессы участвуют в вызове.
  - Каждый процесс задаёт `offset` и количество элементов `count`.

Чтение файла с MPI-IO

Код чтения файла

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, "data.bin",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

MPI_File_read_at_all(fh, offset, buffer, count,
                     MPI_BYTE, &status);
      // collective read

MPI_File_close(&fh); // close file
```

- Коллективное чтение: процессы синхронизированы.
- Совместный доступ к файлу может оптимизировать I/O на распределённой файловой системе.

Сравнение времени чтения файлов

- Замеры на одном узле или локальной машине мало что показывают.
- Эффект распределённого чтения проявляется только на суперкомпьютере.

Тестовая среда

- Суперкомпьютер Ломоносов2
- Тестовая очередь, 210 MPI процессов: 15 узлов по 14 процессов на каждом

Сравнение времени чтения файлов

Сценарий замера

- Файл размером 1 ГБ (дальше размер умножается на 14 процессов, сильно больше не получится)
- Чтение разными способами:
  - POSIX fread
  - MPI-IO collective read (MPI_File_read_at_all)
- Синхронизация через MPI_Barrier, максимальное время через MPI_MAX

Сравнение времени чтения файлов

Сценарий замера

- Файл размером 1 ГБ (дальше размер умножается на 14 процессов, сильно больше не получится)
- Чтение разными способами:
  - POSIX fread
  - MPI-IO collective read (MPI_File_read_at_all)
- Синхронизация через MPI_Barrier, максимальное время через MPI_MAX

Результаты

- Ожидание: MPI-IO должно быть быстрее

Сравнение времени чтения файлов

Сценарий замера

- Файл размером 1 ГБ (дальше размер умножается на 14 процессов, сильно больше не получится)
- Чтение разными способами:
  - POSIX fread
  - MPI-IO collective read (MPI_File_read_at_all)
- Синхронизация через MPI_Barrier, максимальное время через MPI_MAX

Результаты

- Ожидание: MPI-IO должно быть быстрее
- Фактический результат: что так, что так читает порядка 2.7 секунды

Распределённое чтение большого файла

Постановка задачи

- Есть большой бинарный файл (например, десятки гигабайт).
- Нужно, чтобы каждый MPI-процесс прочитал свою непрерывную часть данных:
  - дано общее число процессов P,
  - дан размер файла N байт,
  - процесс с рангом `rank` получает отрезок

$$[\,\texttt{offset} = \frac{N}{P} \cdot \texttt{rank},\ \texttt{offset} + \frac{N}{P}\,)$$

  (для простоты — без учёта остатка).
- Сравниваем три подхода:
  1. POSIX: каждый процесс читает свой участок.
  2. Читает только `rank 0`, затем раздаёт данные через `MPI_Scatter`.
  3. MPI-IO: коллективное чтение `MPI_File_read_at_all`.

1: POSIX, каждый читает свой участок

- Все процессы открывают один и тот же файл `bigfile.bin`.
- Каждый вычисляет свой `offset` и `local_size`.
- Читает свою часть с помощью `fseek + fread` (или `pread`).

```
// open file on each process
FILE *f = fopen(filename, "rb");
if (!f) { /* handle error */ }
// determine file size (on rank 0 and broadcast)
long long global_size = ...;
long long local_size  = global_size / size;
long long offset      = local_size * rank;
// allocate local buffer
std::vector<char> buf(local_size);

// move file position and read
fseek(f, offset, SEEK_SET);       // set position
fread(buf.data(), 1, local_size, f); // read local

fclose(f);
```

2: чтение на rank 0 и MPI_Scatter

- Только процесс `rank 0` читает **весь** файл.
- Затем делит данные на `size` частей.
- Раздаёт по куску каждому процессу с `MPI_Scatter`.

```
std::vector<char> global_buf;
if (rank == 0) {
    FILE *f = fopen(filename, "rb");
    global_size = ...;
    global_buf.resize(global_size);
    fread(global_buf.data(), 1, global_size, f);
    fclose(f); // read entire file
}
MPI_Bcast(&global_size,1,MPI_LONG_LONG,0,MPI_COMM_WORI
long long local_size = global_size / size;
std::vector<char> local_buf(local_size);
// scatter chunks from root to all processes
MPI_Scatter(global_buf.data(), local_size, MPI_BYTE,
            local_buf.data(), local_size, MPI_BYTE,
            0, MPI_COMM_WORLD);
```

3: MPI-IO коллективное чтение

- Используем `MPI_File_open` для коллективного открытия.
- Каждый процесс задаёт свой `offset` и `local_size`.
- `MPI_File_read_at_all` для совместного чтения.

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, filename,
    MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
// determine global file size collectively
MPI_Offset global_size;
MPI_File_get_size(fh, &global_size);
MPI_Offset local_size = global_size / size;
MPI_Offset offset     = local_size * rank;

std::vector<char> buf(local_size);
// collective read at given offsets
MPI_Status st;
MPI_File_read_at_all(fh, offset, buf.data(),
    local_size, MPI_BYTE, &st);
MPI_File_close(&fh);
```

Результаты: файл 1 ГБ

Замеренное максимальное время (по MPI_MAX)

- `[POSIX partition]` ≈ 0.2 s
- `[MPI-IO collective]` ≈ 0.2 s
- `[Root+Scatterv]` ≈ 6 s

Выводы

- **Распределённое чтение**: POSIX и MPI-IO дают сопоставимое время ~ 0.2 секунды.
- **Чтение целиком на root**: почти на порядок медленнее, чем распределённое чтение.
- *Отдельный результат*: чтение теми же процессами, но «все читают целиком» — около 2.7 s (существенно медленнее, чем 0.2 s при разделении файла).

Результаты: файлы большего размера

10 ГБ

- root+Scatter больше не тестируем
- Время плавает, но в обоих случаях ≈ 2 s

Результаты: файлы большего размера

10 ГБ

- root+Scatter больше не тестируем
- Время плавает, но в обоих случаях ≈ 2 s

100 ГБ

- Время выросло сильно непропорционально размеру файла: порядка 160 секунд.
- Основное узкое место - сама файловая система, а не выбор API (POSIX vs MPI-IO).

Результаты: файлы большего размера

10 ГБ

- root+Scatter больше не тестируем
- Время плавает, но в обоих случаях ≈ 2 s

100 ГБ

- Время выросло сильно непропорционально размеру файла: порядка 160 секунд.
- Основное узкое место - сама файловая система, а не выбор API (POSIX vs MPI-IO).
- Может, что-то может показать преимущество MPI функций?

Разреженное чтение (Sparse Read)

Постановка задачи

- Есть большой файл представляющий собой матрицу
- Цель: эффективно прочитать свои **столбцы** на всех процессах MPI.
- Два подхода:
  1. POSIX: чтение с пропуском данных (lseek/fread)
  2. MPI-IO: использование MPI_Type_vector для описания "шагов"в файле

POSIX sparse read

```
// Each process reads every N-th element
for (size_t i = rank; i < total_elements;
     i += size) {
     fseek(fp, i * sizeof(double), SEEK_SET);
     // move to the correct offset
     fread(&local_buf[i / size], sizeof(double),
          1, fp);
     // read one element
}
```

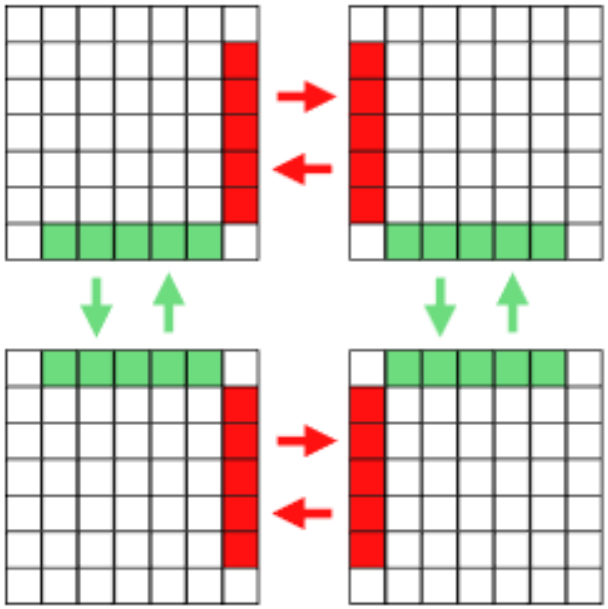
- Простая реализация
- Потенциально много seek — плохая производительность при больших файлах

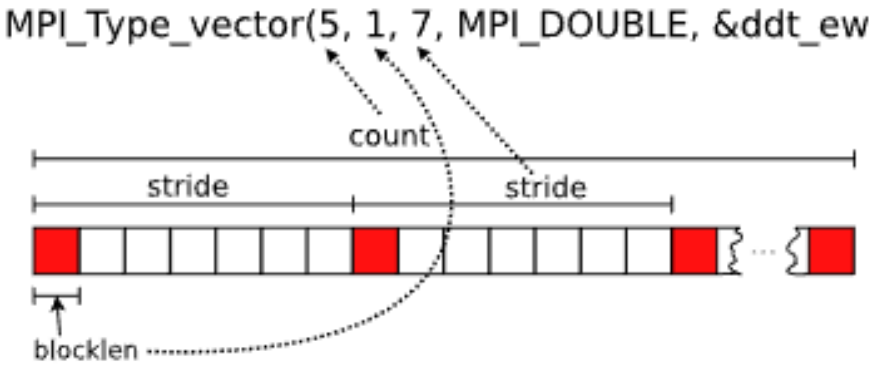
Альтернативы MPI_Type_contiguous

MPI_Type_vector

- Создаёт тип с регулярными «шагами» в памяти

```
MPI_Datatype col_t;
MPI_Type_vector(count, blocklength, stride,
                MPI_DOUBLE, &col_t);
MPI_Type_commit(&col_t);
```





MPI-IO vector read

```
// Define MPI vector type: stride of size elements,
// 1 element per block
MPI_Type_vector(count, 1, size, MPI_DOUBLE,
    &vector_type);
MPI_Type_commit(&vector_type);

MPI_File_open(MPI_COMM_WORLD, "data.bin",
    MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, rank * sizeof(double),
    MPI_DOUBLE, vector_type);
MPI_File_read_all(fh, local_buf, count,
    MPI_DOUBLE, MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

- MPI_Type_vector для определения разреженного типа
- MPI_File_set_view чтобы каждым процессом на свой отступ
- MPI_File_read_all каждый читает по отступу (альтернатива MPI_File_read_at_all)

Результаты замера

Sparse Read: ожидание vs реальность

Ожидание

- MPI_IO с `MPI_Type_vector` должен оптимизировать разреженное чтение
- Потенциальное ускорение за счёт collective I/O и минимизации seek
- POSIX sparse read предполагается медленным

Реальность (тест на 10 GB, 210 процессов)

- **MPI-IO sparse read:** max time = 344.5 s
- **POSIX sparse read:** max time = 2.25 s
- Причина: MPI_Type_vector создаёт большое количество маленьких обращений, которые сильно замедляют работу на Lustre
- POSIX-реализация просто последовательно обращается к нужным байтам, эффективно используя файловую систему

Запись данных из MPI-процессов в файл

- Есть P MPI-процессов, у каждого — свой буфер данных одинакового размера.
- Требуется записать данные **последовательно по рангу** в один общий файл:

$$\text{rank } 0 \to [0..N-1],\ \text{rank } 1 \to [N..2N-1],\ \ldots$$

- не собирать всё на одном процессе (т.е. `MPI_Gather` + `write` не рассматриваем);
- это было бы медленно и потребовало бы слишком много памяти на одном узле.

Два рассматриваемых варианта

1. MPI-IO: `MPI_File_write_at_all` с явным смещением для каждого процесса.
2. POSIX: `open` + `pwrite` / `lseek+write` на всех процессах.

Запись через MPI-IO

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, filename,
              MPI_MODE_WRONLY | MPI_MODE_CREATE,
              MPI_INFO_NULL, &fh);

MPI_Offset offset =
    (MPI_Offset)rank * count * sizeof(double);

// Collective write: all processes participate
MPI_File_write_at_all(fh, offset, buf,
                      count, MPI_DOUBLE,
                      MPI_STATUS_IGNORE);

MPI_File_close(&fh);
```

- `MPI_File_write_at_all` — коллективная операция записи с явным смещением.

Запись через POSIX

```
FILE *f = fopen(filename, "r+b");
long long offset = rank * count * sizeof(double);
if (fseeko(f, (off_t)offset, SEEK_SET) != 0) {
    perror("fseeko");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
size_t written = fwrite(local_buf, 1,
  (size_t)count * sizeof(double), f);
if (written != (size_t)count * sizeof(double)) {
    perror("fwrite");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
fflush(f);
fclose(f);
```

- Каждый процесс сам вычисляет смещение в файле
- Да, все процессы пишут в свой блок в файле, а если надо, удлинняют.

Результаты

- Запись на различных файлах с записываемым суммарным размером 1-100ГБ показывает, что POSIX примерно в два раза быстрее

Результаты

- Запись на различных файлах с записываемым суммарным размером 1-100ГБ показывает, что POSIX примерно в два раза быстрее
- Мне не удалось найти ни одного примера, где MPI вызовы были бы лучше

Результаты

- Запись на различных файлах с записываемым суммарным размером 1-100ГБ показывает, что POSIX примерно в два раза быстрее
- Мне не удалось найти ни одного примера, где MPI вызовы были бы лучше
- Значит, магия не в MPI, а в файловой системе

Результаты

- Запись на различных файлах с записываемым суммарным размером 1-100ГБ показывает, что POSIX примерно в два раза быстрее
- Мне не удалось найти ни одного примера, где MPI вызовы были бы лучше
- Значит, магия не в MPI, а в файловой системе
- Не верьте документации и мануалам! Точнее, доверяй, но проверяй!

Результаты замера

Суперкомпьютерный кодизайн

Смысл суперкомпьютерного кодизайна заключается в согласованном решении всех этапов сложной вычислительной задачи

Суперкомпьютерный кодизайн

Смысл суперкомпьютерного кодизайна заключается в согласованном решении всех этапов сложной вычислительной задачи

- Разработка подходящего алгоритма

Суперкомпьютерный кодизайн

Смысл суперкомпьютерного кодизайна заключается в согласованном решении всех этапов сложной вычислительной задачи

- Разработка подходящего алгоритма
- Эффективная реализация алгоритма

Суперкомпьютерный кодизайн

Смысл суперкомпьютерного кодизайна заключается в согласованном решении всех этапов сложной вычислительной задачи

- Разработка подходящего алгоритма
- Эффективная реализация алгоритма
- Выбор подходящих методов параллелизации, чтобы обеспечить максимально полную загрузку как одного узла, так и многих с использование MPI

Суперкомпьютерный кодизайн

Смысл суперкомпьютерного кодизайна заключается в согласованном решении всех этапов сложной вычислительной задачи

- Разработка подходящего алгоритма
- Эффективная реализация алгоритма
- Выбор подходящих методов параллелизации, чтобы обеспечить максимально полную загрузку как одного узла, так и многих с использование MPI
- Поиск узких мест при запусках на суперкомпьютере (обычно очень сложно предвидеть заранее) и из устранение, обращения в поддержку за диагностикой

Суперкомпьютерный кодизайн

Смысл суперкомпьютерного кодизайна заключается в согласованном решении всех этапов сложной вычислительной задачи

- Разработка подходящего алгоритма
- Эффективная реализация алгоритма
- Выбор подходящих методов параллелизации, чтобы обеспечить максимально полную загрузку как одного узла, так и многих с использование MPI
- Поиск узких мест при запусках на суперкомпьютере (обычно очень сложно предвидеть заранее) и из устранение, обращения в поддержку за диагностикой
- Коммуникация с поддержкой суперкомпьютера с целью нахождения причин падений программы, подстройки опций суперкомпьютера под большую задачу, и, возможно, выбор более подходящего суперкомпьютера или создание нового :)

Суперкомпьютерный кодизайн

Пример в области физики элементарных частиц

- Многие вычислительные задачи могут сводиться к решению огромных разреженных систем линейных уравнений, и в моей области эти системы имеют еще полиномиальные коэффициенты от нескольких переменных.
- Прямолинейное решение таких систем плохо поддается параллелизации и требует наличия серверов со значительными объемами оперативной памяти и большим количеством ядер. Выполнение программ может занимать месяцы и потому очень чувствительно к сбоям питания на серверах и прочим ошибках.
- Если хочется использовать суперкомпьютеры, нужны новые подходы, и эти подходы есть.

Этап 1 - алгоритм

- Уже классическим (где-то с 2010 года) подходом является вариант, когда задача решается много раз для фиксированных значений переменных (и по модулю большого простого числа, чтобы задействовать вещественную арифметику), а потом происходит процедура *восстановления*.
- Самый простой пример восстановления - интерполяционный многочлен Ньютона, когда по набору проб восстанавливается многочлен, а для рациональных функций многих переменных все существенно сложнее.
$Pn(x) = b0 + b1(x - x0) + b2(x - x0)(x - x1) + b3(x - x0)(x - x1)(x - x2) + \ldots + bn(x - x0)\ldots(x - xn)$

Этап 1 - алгоритм

- Уже классическим (где-то с 2010 года) подходом является вариант, когда задача решается много раз для фиксированных значений переменных (и по модулю большого простого числа, чтобы задействовать вещественную арифметику), а потом происходит процедура *восстановления*.
- Правильный выбор алгоритма восстановления и разработка нового являются существенным моментом в решении задачи, поскольку оно определяет количество “точек” — наборов значений переменных, в которых нужно произвести решение системы, а именно оно (теоретически) является определяющим для времени решения задачи.

Этап 1 - линейная реализация

- Если в том, что отдельные запуски решения задачи для конкретных точек будут параллелизованы при помощи MPI нет сомнений, то процедура восстановления вызывает вопросы. Потребуется работа с полиномиальной алгеброй, а понятно, что c++ сам с полиномиальной алгеброй не работает, то есть нужны сторонние библиотеки (или вообще другой язык). Первые пробные решения были сделаны моим студентов на Питоне и, ожидаемо, показали отвратительную производительность.
- Выбор подходящей библиотеки, оптимально работающей с многочленами, а также выбор наиболее подходящих ее функций становится предметом отдельного исследования, в том числе, с использованием профайлеров для поиска “бутылочных горлошек”.

Этап 2 - параллельная реализация на одном узле

- Когда мы начинаем подступаться к сложным задачам, замеры показывают, что восстановление начинает занимать все более существенное время даже после внутренней оптимизации. При этом некоторые этапы восстановления блокируют другие вычисления, и их нужно ускорять. Естественным шагом является параллелизация восстановления, сначала на один узел.
- Приходит довольно прямолинейное решение — поскольку необходимо восстанавливать сразу много коэффициентов, можно разделить это по потокам и восстанавливать параллельно. Простое оказывается эффективным, и алгоритм ускоряется. На данном этапе этого хватает.

Этап 3 - оптимизация под суперкомпьютер

- Отладка производительности на суперкомпьютере — сложный процесс, поскольку вы не можете просто взять и подключить свой профилировщик. Для поиска узких мест желательно использовать профилировщик от команды, поддерживающий суперкомпьютер, чтобы они могли, например, определить, что ваша программа проводит слишком много времени на обмене MPI. Такая диагностика проводится, производится оптимизация.
- При этом вычисления в различных точках параллелизованы на супекомпьютере с использованием MPI

Этап 3 - оптимизация под суперкомпьютер

- Восстановление коэффициентов занимает по-прежнему слишком много времени, по этому нужно использовать MPI для восстановления. Однако восстановление предполагает сначала считывание большого количества файлов. Если читать их все на одном узле и рассылать через MPI, то это перегружает систему MPI. Если считывать полностью на каждом узле, перегружает файловую систему. Приходится изменять структуру промежуточных файлов, чтобы позволить использовать специальные функции MPI для одновременного чтения разных частей файлов на узлах.
- По результатам переработки удается существенно увеличить скорость восстановления, мы выходим на уровень, когда можем делать восстановление с использованием более ста тысяч значений.

Этап 4 - подстройка под суперкомпьютер

- Начиная с какого-то количества точек (порядка 300 тысяч) программа начинает неожиданно обваливаться. Стектрейс указывает на какие-то проблемы в MPI. После безуспешного поиска приходится обращаться в поддержку, предоставлять много информации, и оказывается, что проблемы имеются в распределенной файловой системе lustre, в которой есть проблемы, когда накапливается значительное количество файлов в одной папке. Это можно было бы решить, обновив драйвер файловой системы, но для этого нужно много что обновить и останавливать все вычисления на суперкомпьютере.

Этап 4 - подстройка под суперкомпьютер

- Было принято решение доработать код с тем, чтобы файлы хранились не в одной директории, а в нескольких вложенных в зависимости от номера. Это позволило продвинуться дальше до миллиона значений.

Этап 4 - подстройка под суперкомпьютер

- Было принято решение доработать код с тем, чтобы файлы хранились не в одной директории, а в нескольких вложенных в зависимости от номера. Это позволило продвинуться дальше до миллиона значений.
- Но проблемы могут не закончится, и в идеале для больших задач с финансированием нужно изначально подходить к правильному проектированию суперкомпьютера или выбору подходящего суперкомпьютера. А в нашем случае была история с багом в старых библиотеках на суперкомпьютере, невозможностью обновить и решением через контейнеризацию...

Раздел 6. План.

Параллельное программирование в распределенной памяти с использованием MPI

- Введение и простейшая программа
- Базовые приемы параллелизации с MPI
- Коммуникаторы и составные типы
- Распределенная работа с файлами
- **Вариации попарного обмена в MPI**
- Пример с ускорением для MPI – интегрирование

Варианты попарного обмена MPI

`MPI_Send`

- Базовый режим посылки. Может быть как блокирующим, так и не блокирующим
- Вызов гарантирует то, что если код перешел на следующую строчку, то массив данных, из которого отправляли, можно использовать для иных задач

Варианты попарного обмена MPI

MPI_Send

- Базовый режим посылки. Может быть как блокирующим, так и не блокирующим
- Вызов гарантирует то, что если код перешел на следующую строчку, то массив данных, из которого отправляли, можно использовать для иных задач
- При этом система не дает гарантии, что сообщение уже было получено приемником! В случае, если это важно для логики программы, нужно использовать иной метод! Решение остается за системой mpi в зависимости от размера сообщения и загруженности системного буфера

Варианты попарного обмена MPI

MPI_Ssend

- Блокирующий режим посылки
- Вызов гарантирует то, что если код перешел на следующую строчку, сообщение уже было получено приемником, что в частности означает, что массив данных, из которого отправляли, можно использовать для иных задач

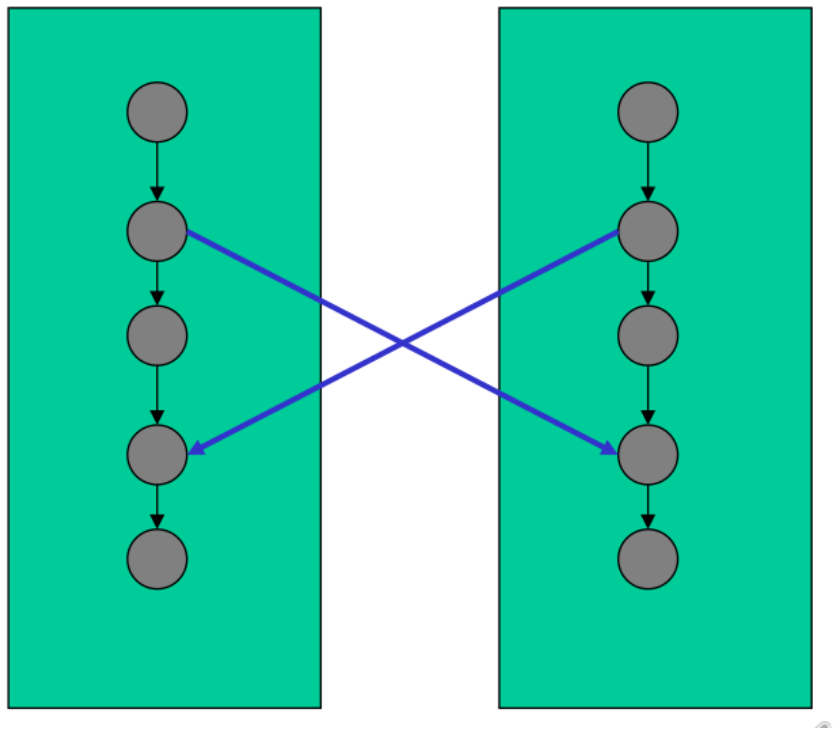
Варианты попарного обмена MPI

`MPI_Ssend`

- Блокирующий режим посылки
- Вызов гарантирует то, что если код перешел на следующую строчку, сообщение уже было получено приемником, что в частности означает, что массив данных, из которого отправляли, можно использовать для иных задач
- Данный метод считается более медленным, чем `MPI_Send`

Deadlock

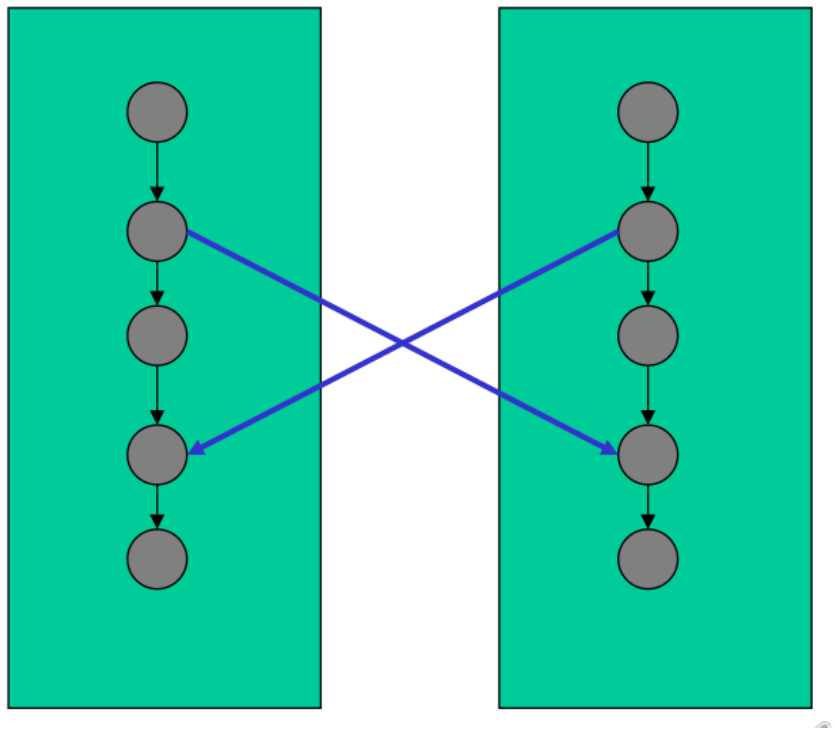
```
if(rank == 0) {
    MPI_Ssend(...,1 ,)
    MPI_Recv(...,1 ,)
} else {
    MPI_Ssend(...,0 ,)
    MPI_Recv(...,0 ,)
}
```



Deadlock

Deadlock недетерминированный

```
if(rank == 0) {
      MPI_Send(...,1   ,)
      MPI_Recv(...,1   ,)
} else {
      MPI_Send(...,0   ,)
      MPI_Recv(...,0   ,)
}
```



Ошибки в MPI-программах

- Deadlock — процессы ждут друг друга в `MPI_Recv` / `MPI_Ssend`
- Несогласованные вызовы: один процесс делает `MPI_Bcast`, другой — нет
- Несоответствие типов/размеров: `MPI_Send` посылает 100 `double`, `MPI_Recv` ждёт 50 `int`
- Неправильный `tag` или `source` — сообщение «теряется»

Ошибки в MPI-программах

- Deadlock — процессы ждут друг друга в `MPI_Recv` / `MPI_Ssend`
- Несогласованные вызовы: один процесс делает `MPI_Bcast`, другой — нет
- Несоответствие типов/размеров: `MPI_Send` посылает 100 `double`, `MPI_Recv` ждёт 50 `int`
- Неправильный `tag` или `source` — сообщение «теряется»
- Ошибки в индексах при разбиении данных часто дают мусор в данных, а не падение
- Многие ошибки проявляются только при определённом числе процессов или размере задачи
- Почти все функции MPI возвращают `int` - проверять на `MPI_SUCCESS`

Отладка MPI-программ: `mpirun + gdb`

- Вариант с отдельным `xterm` для каждого процесса:

```
mpirun -np 4 xterm -e gdb --args ./mpi_program
```

- На каждом окне `gdb` нужно:
  - набрать `run` (или `r`)
  - при падении или зависании использовать `bt` для получения backtrace

Отладка MPI-программ: `mpirun + gdb`

- Вариант с отдельным `xterm` для каждого процесса:

```
mpirun -np 4 xterm -e gdb --args ./mpi_program
```

- На каждом окне `gdb` нужно:
  - набрать `run` (или `r`)
  - при падении или зависании использовать `bt` для получения backtrace

Замечания

- Нужен установленный `xterm`
- Удобнее начинать с `-np` 2–3, иначе будет слишком много окон
- Работает только на локальной машине или с X-forwarding по SSH

Мини-шпаргалка по gdb

Старт

- Запуск: `gdb ./mpi_program`
- Запуск с аргументами: `gdb -args ./mpi_program arg1 arg2`
- Старт программы: `run` или `r`

Мини-шпаргалка по gdb

Старт

- Запуск: `gdb ./mpi_program`
- Запуск с аргументами: `gdb -args ./mpi_program arg1 arg2`
- Старт программы: `run` или `r`

Точки останова и просмотр состояния

`break main`	остановиться в начале `main`
`break foo`	остановиться в функции `foo`
`break file.cpp:42`	остановиться на строке 42 в `file.cpp`
`continue`	продолжить выполнение
`next`	следующая строка (не заходя в функции)
`step`	шаг с заходом в функцию
`print x`	вывести значение переменной `x`
`backtrace` (`bt`)	показать стек вызовов

Варианты попарного обмена MPI

`MPI_Rsend`

- Интересный вариант в определенных случаях
- Вызов можно применять только в том случае, если программист каким-то образом может гарантировать, что к этому моменту уже был сделан вызов `MPI_Recv`, в противных случаях поведение не определено

Варианты попарного обмена MPI

`MPI_Rsend`

- Интересный вариант в определенных случаях
- Вызов можно применять только в том случае, если программист каким-то образом может гарантировать, что к этому моменту уже был сделан вызов `MPI_Recv`, в противных случаях поведение не определено
- Данный метод считается более быстрым, чем `MPI_Send`, поскольку не предполагает задержек или буферизации, но использовать его требуется с осторожностью

Варианты попарного обмена MPI

`MPI_Bsend`

- Неблокирующий вызов, который требует от программиста завести свой собственный буфер, в который будут помещены данные
- Требует особого синтаксиса и подготовки буфера. Поскольку отправляемые данные сразу помещаются в буфер, их также можно переиспользовать после вызова.

Варианты попарного обмена MPI

MPI_Bsend

- Неблокирующий вызов, который требует от программиста завести свой собственный буфер, в который будут помещены данные
- Требует особого синтаксиса и подготовки буфера. Поскольку отправляемые данные сразу помещаются в буфер, их также можно переиспользовать после вызова.
- Используется в случаях организовать перекрестный обмен или для быстрой посылки данных

Варианты попарного обмена MPI – MPI_Bsend

- Посылка каждому процессу числа с его номером

```
int msize;
MPI_Pack_size(1, MPI_INT,
    MPI_COMM_WORLD, &msize);
int blen = M * (msize + MPI_BSEND_OVERHEAD);
buf = malloc(blen);
MPI_Buffer_attach(buf, blen);
for(i = 0; i < M; i ++) {
    n = i;
    MPI_Bsend(&n, 1, MPI_INT, i, 0,
        MPI_COMM_WORLD);
}
MPI_Buffer_detach(&abuf, &ablen);
free(abuf);
```

Варианты попарного обмена MPI

MPI_Isend и MPI_Irecv

- Неблокирующий небуферизующий вызов.
- Дает возможность проверять ниже, дошло сообщение или нет

Варианты попарного обмена MPI

`MPI_Isend` и `MPI_Irecv`

- Неблокирующий небуферизующий вызов.
- Дает возможность проверять ниже, дошло сообщение или нет
- Должна быть проявлена особая осторожность, чтобы не перезаписать область, из которых отправлялись данные, необходимо ниже проверять, дошли ли данные

Варианты попарного обмена MPI

`MPI_Isend` и `MPI_Irecv`

- Неблокирующий небуферизующий вызов.
- Дает возможность проверять ниже, дошло сообщение или нет
- Должна быть проявлена особая осторожность, чтобы не перезаписать область, из которых отправлялись данные, необходимо ниже проверять, дошли ли данные
- А еще есть комбинации `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend`, но они используются реже. Разница между `MPI_Isend`, `MPI_Ibsend` и `MPI_Issend` проявляется в момент проверки того, дошло ли сообщение

Зачем использовать `MPI_Irecv`

- Одна из причин, это если процесс должен регулярно проверять, не пришли ли новые данные, но при этом выполняет какую-то работу. Даже если он сделал отдельную ветку для ожидания в `MPI_Recv`, эта ветка пытается полностью загрузить процессор

Варианты попарного обмена MPI

Зачем использовать `MPI_Irecv`

- Одна из причин, это если процесс должен регулярно проверять, не пришли ли новые данные, но при этом выполняет какую-то работу. Даже если он сделал отдельную ветку для ожидания в `MPI_Recv`, эта ветка пытается полностью загрузить процессор
- Запуск с `mpirun -np N -mca mpi_yield_when_idle 1 program` снижает эту нагрузку, но не существенно

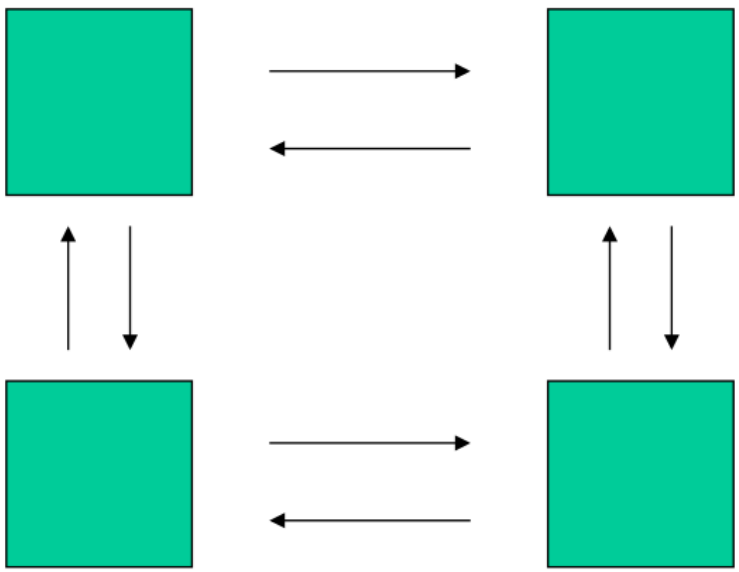
Варианты попарного обмена MPI

Зачем использовать `MPI_Irecv`

- Одна из причин, это если процесс должен регулярно проверять, не пришли ли новые данные, но при этом выполняет какую-то работу. Даже если он сделал отдельную ветку для ожидания в `MPI_Recv`, эта ветка пытается полностью загрузить процессор
- Запуск с `mpirun -np N -mca mpi_yield_when_idle 1 program` снижает эту нагрузку, но не существенно
- Решение – цикл, состоящий из `MPI_Irecv` и `std::this_thread::sleep_for`

Варианты попарного обмена MPI – MPI_Isend

Кольцевой сдвиг



Варианты попарного обмена MPI – MPI_Isend

```
#include <mpi.h>
#include <stdio.h>
void main (int argc, char* argv[])
{
  int numtasks, rank, next, prev, buf[2],
  tag1 = 1, tag2 = 2;
  MPI_Request reqs[4];
  MPI_Status stats[4];
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  prev = (rank == 0) ?
    (numtasks - 1) : (rank - 1);
  next = (rank == (numtasks - 1)) ?
    0 : (rank + 1);
```

Варианты попарного обмена MPI – MPI_Isend

```
    MPI_Irecv (&buf[0], 1, MPI_INT, prev, tag1,
      MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv (&buf[1], 1, MPI_INT, next, tag2,
      MPI_COMM_WORLD, &reqs[1]);
    MPI_Isend (&rank, 1, MPI_INT, prev, tag2,
      MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend (&rank, 1, MPI_INT, next, tag1,
      MPI_COMM_WORLD, &reqs[3]);
    MPI_Waitall (4, reqs, stats);
    printf("rank: %d, buf[0]: %d, buf[1]: %d\n",
      rank, buf[0], buf[1]);
    MPI_Finalize ();
}
```

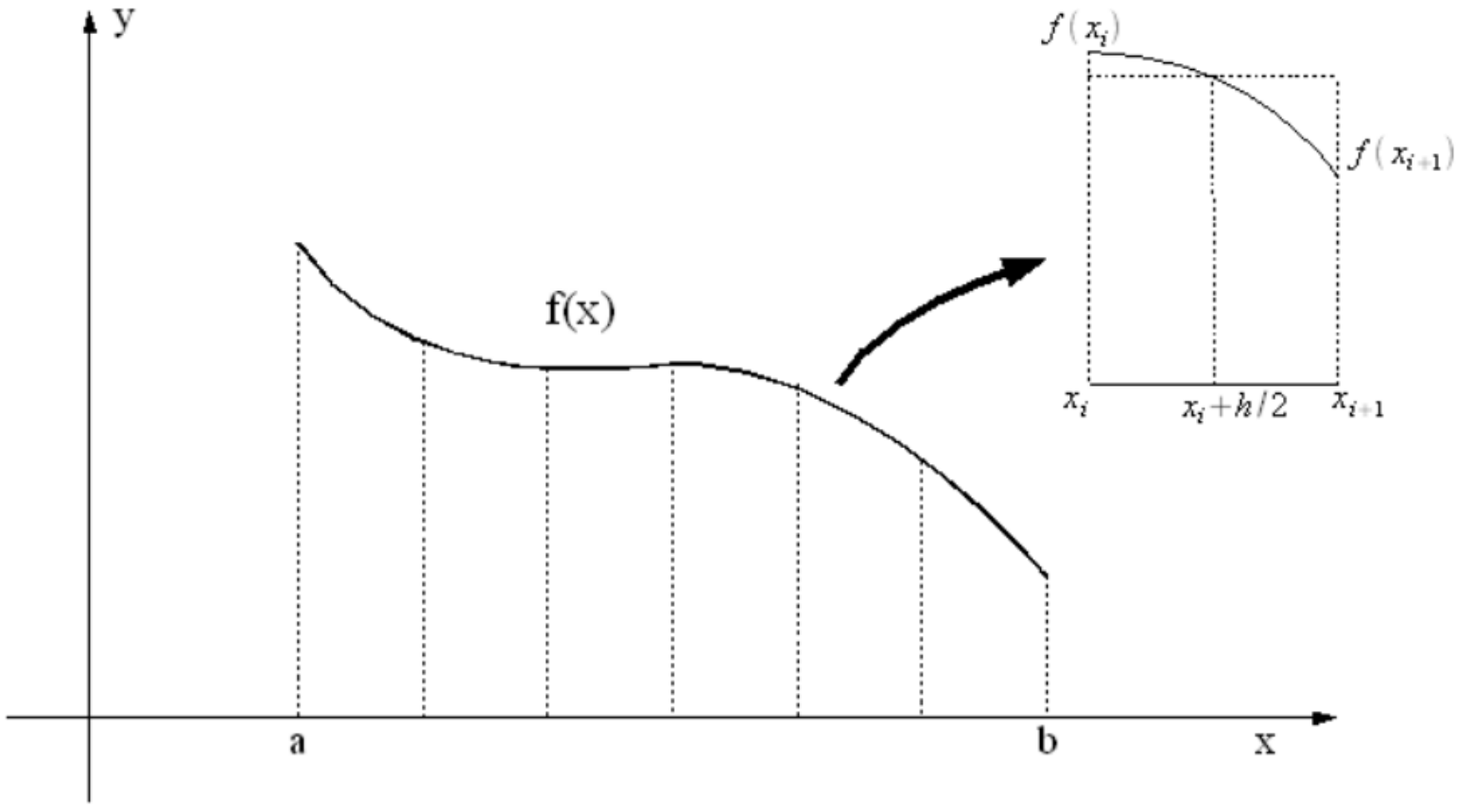
Вопросы?

Раздел 6. План.

Параллельное программирование в распределенной памяти с использованием MPI

- Введение и простейшая программа
- Базовые приемы параллелизации с MPI
- Коммуникаторы и составные типы
- Распределенная работа с файлами
- Вариации попарного обмена в MPI
- **Пример с ускорением для MPI – интегрирование**

Численное интегрирование



Численное интегрирование

```
#include <mpi.h>
#include <chrono>
#include <iostream>

double f(double x) {
    return 4./(1 + x * x);
}
```

Численное интегрирование

```
int main(int argc, char* argv[]) {
    int rank, size;
    int i;
    constexpr size_t n = 256 * 1024 * 1024;
    double a = 0.0, b = 1.0;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    auto start =
        std::chrono::steady_clock::now();
    double h = (b - a) / n;
    double sum = 0.;
    for (size_t i = rank; i < n; i += size) {
        sum += f(a + (i + 0.5) * h);
    }
    sum *= h;
```

Численное интегрирование

```
if (rank != 0) {
    MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0,
        MPI_COMM_WORLD);
} else {
    double s;
    for (int i = 1; i < size; i ++) {
        MPI_Status st;
        MPI_Recv(&s, 1, MPI_DOUBLE, i, 0,
            MPI_COMM_WORLD, &st);
        sum += s;
    }
    std::cout << "Integral value =
        " << sum << "\n";
}
```

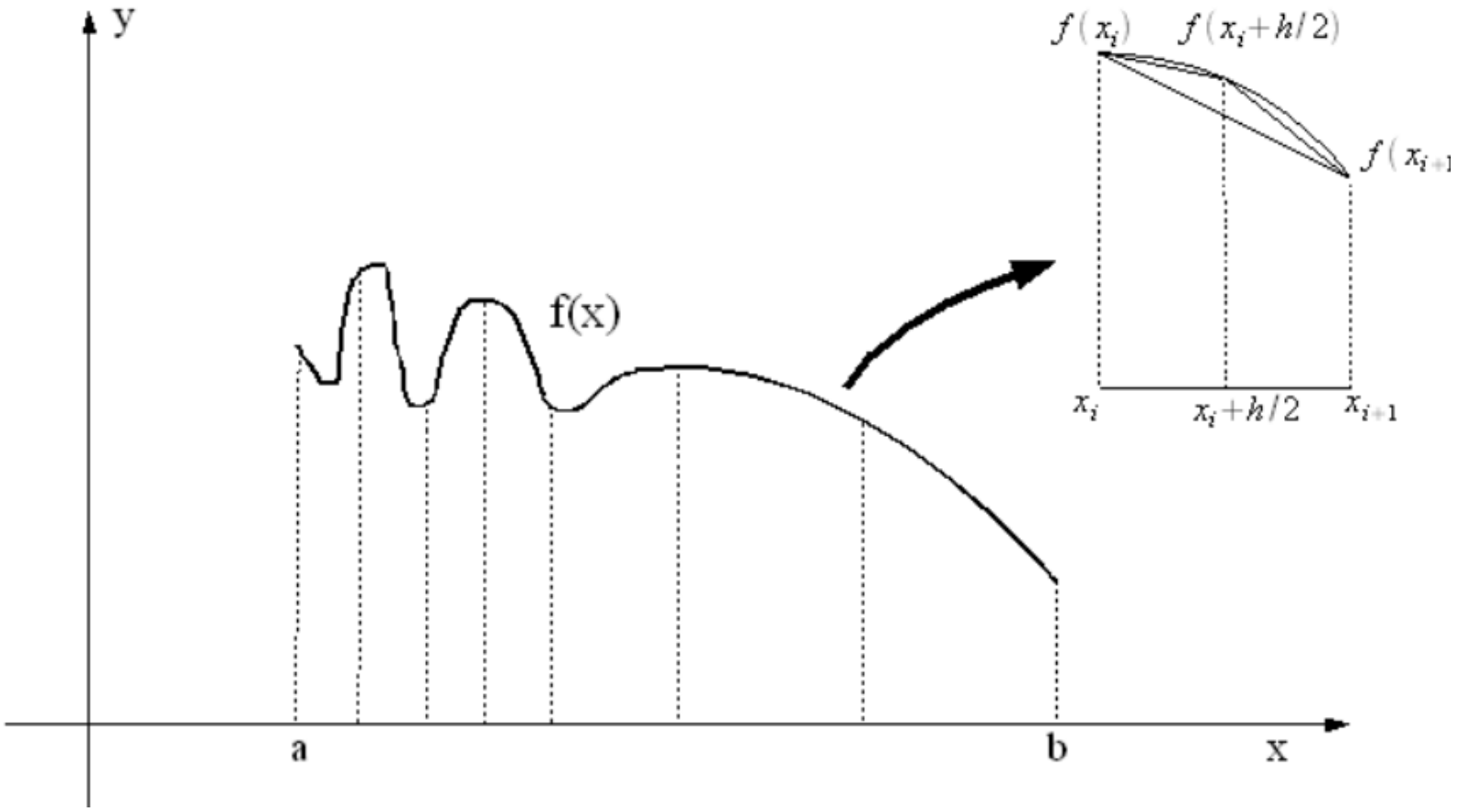
```
    auto end =
        std::chrono::steady_clock::now();
    auto diff = end-start;
    int time = std::chrono::duration_cast
    <std::chrono::milliseconds>(diff).count();
    if (rank == 0) {
        std::cout << time << " milliseconds "
            << std::endl;
    }
    MPI_Finalize();
}
```

Произведем замер времени

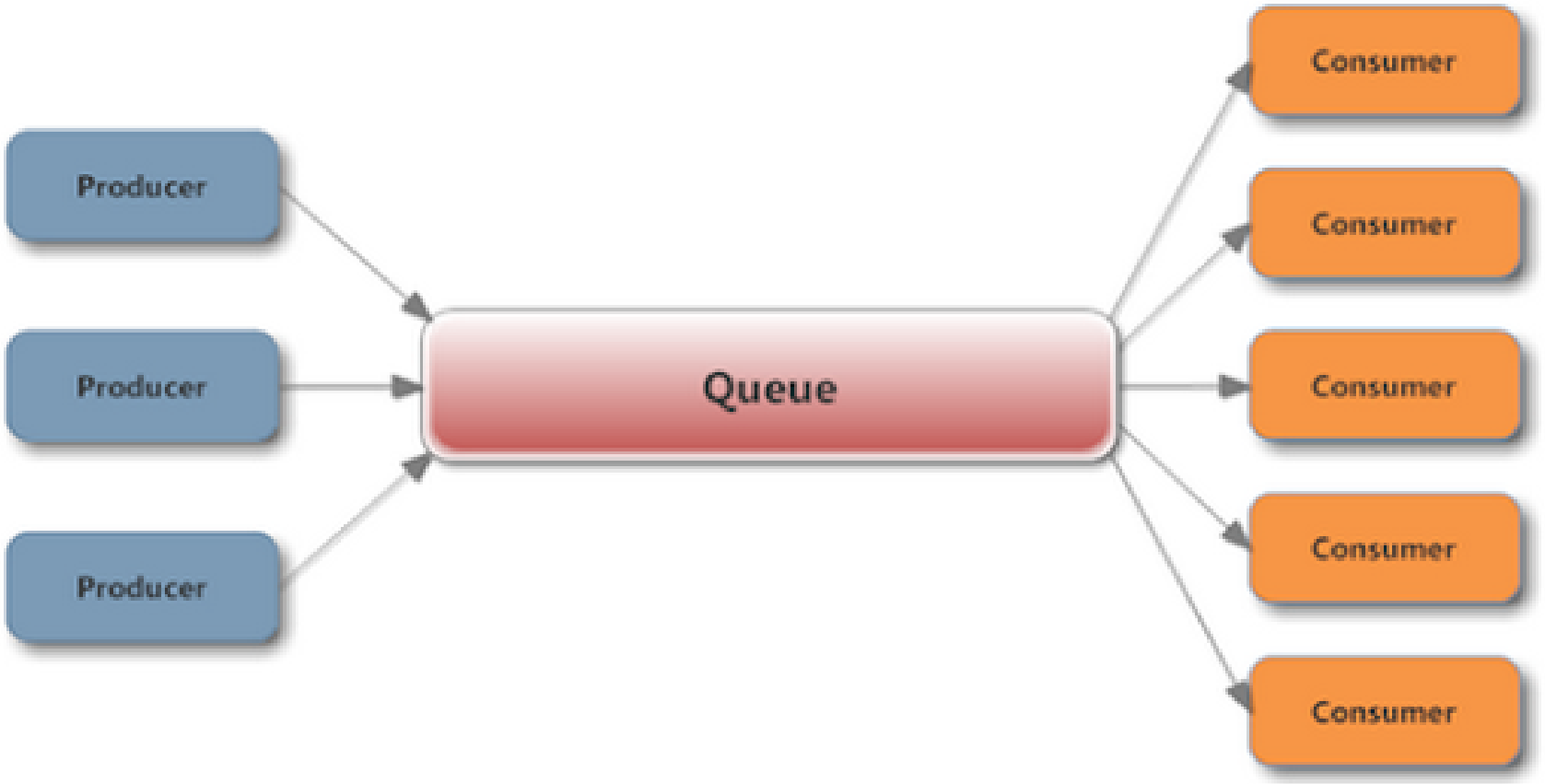
```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 2 ./mpi
210 milliseconds
mpirun -np 3 ./mpi
147 milliseconds
mpirun -np 4 ./mpi
113 milliseconds
```

Мы получили ускорение, близкое к линейному. Причина в том, что отдельные процессы никак не борются за данные в памяти. При этом ускорение выходит очень близкое к тому, что получается на threads (проверял)

Адаптивная квадратура – пример на consumer-producer



Паттерн producer-consumer



Численное интегрирование

```
#include <chrono>
#include <iostream>
#include <math.h>
#include <mpi.h>
#include <atomic>

std::atomic<size_t> total = {};
double f(double x) {
    return sin(1. / x);
}
```

Численное интегрирование

```
bool add_integration_part(double(*f)(double),
double left, double right, double eps,
double& I) {double h = 0.5 * (right - left);
    double mid = left + h;
    double Fleft = f(left);
    double Fright = f(right);
    double Fmid = f(mid);
    double Iold = h * (Fleft + Fright);
    double Ileft = 0.5 * h * (Fleft + Fmid);
    double Iright = 0.5 * h * (Fmid + Fright);
    I = Ileft + Iright;
    if (abs(Iold - I) < eps) {
        total += 2; //points counter
        return true;
    } else return false;
}
```

Численное интегрирование

```
double recursive_add_integration_part(
    double (*f)(double), double left,
    double right, double eps) {
    double I;
    if(add_integration_part(f, left, right,
            eps, I)) return I; else {
        double mid = 0.5 * (right + left);
        double Ileft =
            recursive_add_integration_part
            (f, left, mid, 0.5 * eps);
        double Iright =
            recursive_add_integration_part
            (f, mid, right, 0.5 * eps);
        return Ileft + Iright;
    }
}
```

Численное интегрирование

```
int main(int argc, char* argv[]) {
    int rank, size;
    int n = 128 * 1024;
    int stopped = 0;
    double a = 0.0001,b = 1.0,eps = 0.000001;
    double I = 0.;
    MPI_Status st;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    double h = (b - a) / n;
    --n;
    auto start =
        std::chrono::steady_clock::now();
```

Численное интегрирование

```
if (rank == 0) {
    while (stopped != (size-1)) {
        double Islave;
        MPI_Recv(&Islave, 1, MPI_DOUBLE,
            MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &st);
        I += Islave;
        MPI_Send(&n, 1, MPI_INT,
            st.MPI_SOURCE,
            0, MPI_COMM_WORLD);
        if (n >= 0)
            --n;
        else
            ++stopped;
    }
} else {
```

Численное интегрирование

```
//    } else {
        int m;
        while(1) {
            MPI_Send(&I, 1, MPI_DOUBLE, 0,
                0, MPI_COMM_WORLD);
            MPI_Recv(&m, 1, MPI_INT, 0,
              MPI_ANY_TAG,MPI_COMM_WORLD,&st);
            if (m >= 0) I =
                recursive_add_integration_part
                (f, a + h * m, a + h * (m + 1),
                    eps * h / (b - a));
            else {
                break;
            }
        }
    }
```

```
    if (rank == 0) {
      std::cout<<"Integral value: "<< I <<"\n";
      auto end=std::chrono::steady_clock::now();
      auto diff = end-start;
      int time = std::chrono::duration_cast
      <std::chrono::milliseconds>(diff).count();
      std::cout << time << " ms "<< std::endl;
    }
    long long tot = 0.;
    MPI_Reduce(&total, &tot, 1, MPI_LONG_LONG,
      MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        std::cout<<"Total points " <<tot<<"\n";
    }
    MPI_Finalize();
}
```

Произведем замер времени

```
mpirun –np 2 ./mpi3
Integral value: 0.504067
355 milliseconds
Total points 5765816
mpirun –np 3 ./mpi3
Integral value: 0.504067
189 milliseconds
Total points 5765816
mpirun –np 4 ./mpi3
Integral value: 0.504067
165 milliseconds
Total points 5765816
mpirun –np 5 ./mpi3
Integral value: 0.504067
133 milliseconds
Total points 5765816
```

Коммуникации vs. вычисления

- MPI полезен, когда вычислений много, а обмена данными мало

Коммуникации vs. вычисления

- MPI полезен, когда вычислений много, а обмена данными мало
- Сложение векторов:
  - много пересылок (Scatter/Gather),
  - мало работы на элемент (одно сложение)
  - результат - замедление

Коммуникации vs. вычисления

- MPI полезен, когда вычислений много, а обмена данными мало
- Сложение векторов:
  - много пересылок (Scatter/Gather),
  - мало работы на элемент (одно сложение)
  - результат - замедление
- Численное интегрирование:
  - данные почти не пересылаются
  - каждый процесс считает свою часть
  - результат - почти линейное ускорение

Практические советы по производительности

- Минимизируйте число сообщений
  - объединяйте мелкие сообщения в более крупные
  - избегайте лишних `MPI_Barrier`, «на всякий случай»

Практические советы по производительности

- Минимизируйте число сообщений
  - объединяйте мелкие сообщения в более крупные
  - избегайте лишних `MPI_Barrier`, «на всякий случай»
- Используйте коллективные операции
  - `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Allreduce` часто реализованы очень эффективно
  - ручные циклы из `Send/Recv` обычно хуже

Практические советы по производительности

- Минимизируйте число сообщений
  - объединяйте мелкие сообщения в более крупные
  - избегайте лишних `MPI_Barrier`, «на всякий случай»
- Используйте коллективные операции
  - `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Allreduce` часто реализованы очень эффективно
  - ручные циклы из `Send/Recv` обычно хуже
- Следите за балансом нагрузки
  - если одна нода считает 90% работы — весь кластер ждёт её
  - адаптивные алгоритмы (как в интегрировании) требуют аккуратного распределения задач

Вопросы?

Домашнее задание: MPI-паттерны

Внимание! Задание!

- Каждому студенту назначается *MPI-паттерн* (способ организации параллельных вычислений)
- Необходимо:
  - выбрать или придумать задачу, для которой этот паттерн уместен,
  - реализовать простейшую программу на MPI, демонстрирующую пользу этого паттерна,
  - минимизировать использование других возможностей MPI (чтобы акцент был именно на выбранном паттерне).

Домашнее задание: выбор паттерна

Как определить ваш паттерн

- Пусть `number` — ваш номер в списке группы
- Вычислите:

$$pattern = ((number/4)\%10) + 1$$

- Число `pattern` от 1 до 10 и определяет ваш MPI-паттерн

Список MPI-паттернов (1–5)

1. **SPMD + статическое разбиение данных**
   Каждый процесс обрабатывает свой “кусок” данных, разбиение определяется по рангу. Возможны простые коллективные операции (`Bcast`, `Reduce` и т.п.).
2. **Master–worker (динамическая раздача работы)**
   Один процесс (обычно ранг 0) раздаёт задания другим и собирает результаты через `MPI_Send` / `MPI_Recv`.
3. **Группы процессов и подкоммуникаторы**
   Разделение `MPI_COMM_WORLD` на подгруппы с помощью `MPI_Comm_split` / `MPI_Group_*`.
4. **Producer–consumer на сообщениях**
   Один или несколько процессов производят задания, другие потребляют; очередью задач.
5. **Кольцевой обмен (ring)**
   Процессы образуют цикл, данные (токен, частичный результат и т.п.) передаются по кругу: `rank` $\rightarrow$ `(rank+1) % size`.

Список MPI-паттернов (6–10)

6. **Линейная цепочка / конвейер (pipeline)**
   Процессы образуют цепочку, данные проходят от процесса 0 к последнему, по пути последовательно обрабатываются.
7. **Коллективные операции как основной механизм**
   Активное использование `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Allreduce` и др. вместо ручных циклов с `Send/Recv`.
8. **Пользовательские типы данных** (`MPI_Type_*`)
   Определение и использование MPI-типа для пересылки нетривиальной структуры данных: подматрицы, структур, “редких” элементов массива.
9. **Упаковка и распаковка** (`MPI_Pack/Unpack`)
   Передача нескольких разнотипных фрагментов данных одним сообщением с явной упаковкой в буфер.
10. **Особые режимы посылки** (`MPI_Bsend/Rsend`)
    Использование буферизованной или “готовой” посылки вместо базового `MPI_Send`, демонстрация их особенностей.

Домашнее задание: выбор задачи

Задачу можно выбрать самостоятельно, но она должна быть разумной

Возможные варианты:

- обработка одномерного массива (сумма, максимум, скалярное произведение и т.п.);
- численное интегрирование на отрезке;
- операции с матрицами (умножение на вектор, суммирование по строкам/столбцам);
- работа со структурированными данными (массив структур, точки в 2D/3D, сетка);
- своя прикладная задача (имитация очередей, модель, вычислительный эксперимент и т.п.).

В отчёте/комментариях объясните, почему ваш паттерн подходит именно для выбранной задачи.

Домашнее задание: тип данных массива

Внимание! Задание!

Варианты чисел в массиве:

1. int (32-bit)
2. float
3. long int / long long int (64-bit)
4. double

Чтобы определить ваш вариант, возьмите остаток от деления на 4 и прибавьте 1

$$(number\%4) + 1$$

Домашнее задание: требования к программе

Внимание! Задание!

- Программа должна быть консольной и работать на любой системе, без специфичных IDE
- Должны быть инструкции по сборке программы (например, Makefile)
- Минимум — команда сборки; желательно — полноценная организация проекта
- Программа должна выводить ваш номер и какую задачу вы решали

Домашнее задание: рекомендации по сборке

Внимание! Задание!

- Makefile или CMake
- Подача программы единым архивом или через репозиторий
- Комментарии к коду, поясняющие реализацию

Вопросы?