Тупиковые ситуации

Process 0:
MPI_Send используя буфер send_buf

return только когда send_buf может снова быть использован

Process 1:
MPI_Recv используя буфер recv_buf

return только когда в recv_buf полностью записано всё сообщение

Что может произойти:

1. **Процесс 0** скопирует данные в системный буфер (в доступе процесса 0, процесса 1 или где угодно ещё) и **выйдет из функции** MPI_Send
2. Если системного буфера нет или данных слишком много, **процесс 0 будет ждать**, пока **процесс 1** получит сообщение
3. Fail :(

Тупиковые ситуации

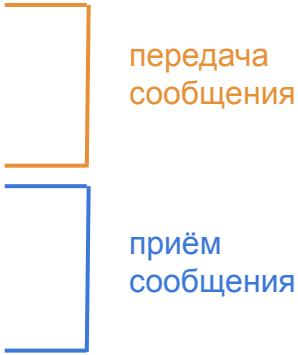
Process 0	Process 1
MPI_Send to process 1	MPI_Send to process 0
MPI_Recv from process 1	MPI_Recv from process 0

Варианты решения:

1. Правильный порядок операций MPI_Send и MPI_Recv
2. **MPI_Sendrecv**
3. Неблокирующие асинхронные операции **MPI_Isend** и **MPI_Irecv (+ MPI_Wait)**

MPI_Sendrecv

```
int MPI_Sendrecv(
        void *sendbuf,
        int sendcount,
        MPI_Datatype sendtype,
        int dest,
        int sendtag,
        void *recvbuf,
        int recvcount,
        MPI_Datatype recvtype,
        int source,
        MPI_Datatype recvtag,
        MPI_Comm comm,
        MPI_Status *status
)
```



- Гарантируется, что тупика не возникнет.
- Сообщение, отправленное операцией MPI_Sendrecv можно принять обычным способом
- Сообщение, отправленное обычным способом можно принять с помощью MPI_Sendrecv

! Буферы приёма и передачи не должны пересекаться

Передача сообщений без блокировки

- Позволяет отправлять сообщения в процессе вычислений без длительного простоя (асинхронно)

- MPI_Isend и MPI_Irecv не ожидают завершения передачи, сразу происходит возврат из функции

- Следить за тем, чтобы данные не были модифицированы до того, как передача сообщений закончится - **ответственность разработчика**

MPI_Isend & MPI_Irecv

```
int MPI_Isend(
        void *buf,
        int count,
        MPI_Datatype datatype,
        int dest,
        int msgtag,
        MPI_Comm comm,
        MPI_Request *request
)
```

```
int MPI_Irecv(
        void *buf,
        int count,
        MPI_Datatype datatype,
        int source,
        int msgtag,
        MPI_Comm comm,
        MPI_Request *request
)
```

MPI_Request - идентификатор асинхронного приёма/передачи сообщения

MPI_Wait

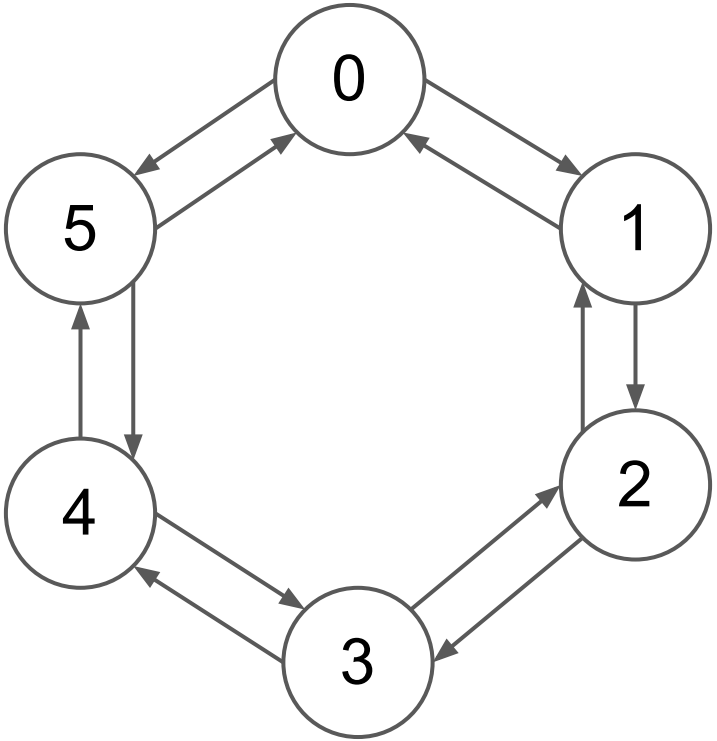
int MPI_Wait(MPI_Request *request, MPI_Status *status)

Блокирующая операция.
Возврат происходит после завершения операции, связанной с request.
В параметре status возвращается информация о законченной операции.

int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)

Процесс блокируется до тех пор, пока все count операций, связанные с requests не будут завершены.

Каждый процесс отправляет сообщение своим соседям и получает от них ответ



https://github.com/Vruchtel/ParallelDistributed/tree/master/sem02

MPI_ANY_SOURCE & MPI_ANY_TAG

Предопределённые константы, которые можно использовать при **приёме** сообщения вместо **идентификатора отправляющего процесса** или **идентификатора сообщения**.

MPI_ANY_SOURCE - подходит сообщение от любого процесса

MPI_ANY_TAG - подходит сообщение с любым идентификатором

! При отправке необходимо обязательно указывать id-адресата и номер сообщения.

MPI_Status

MPI_Status - это структура содержащая 3 поля:

- **MPI_SOURCE** - номер процесса отправителя
- **MPI_TAG** - идентификатор сообщения
- **MPI_ERROR** - код ошибки

Не получая сообщение...

Можно не принимая сообщение получить информацию о его атрибутах

int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status)
Записывает в структуру status параметры принимаемого сообщения

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
по значению поля status определяет число принятых (вызов после MPI_Recv) или принимаемых (после MPI_Probe) элементов соответствующего сообщения

Сообщение не принимается!
Если после MPI_Probe вызвать MPI_Recv, то примется то же самое сообщение

Время

$$S = \frac{T_1}{T_p}$$

Ускорение

T_1 Время работы алгоритма на одном процессоре (ядре)

T_p Время работы алгоритма на **p** процессорах (ядрах)

Закон Амдала

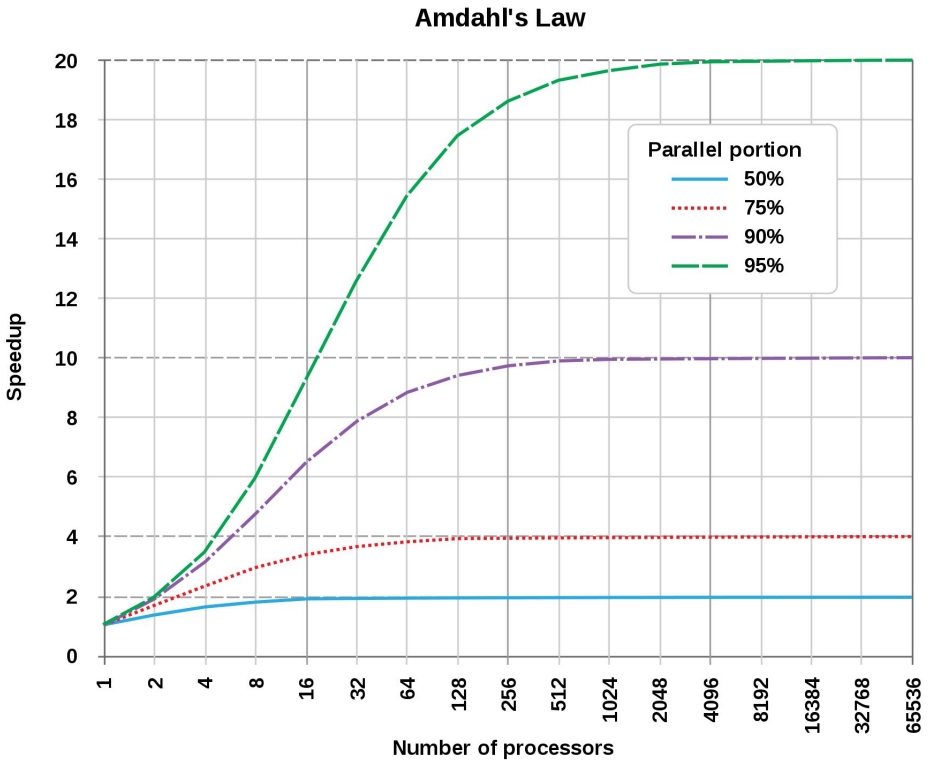
Джин Амдал предложил формулу, отражающую зависимость ускорения вычислений от числа процессов и от соотношения последовательной и распараллеливаемой части программы

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{fT(1) + \frac{(1-f)T(1)}{n}} = \frac{1}{f + \frac{1-f}{n}}$$

n - количество процессов
f - доля вычислений, которую невозможно распараллелить

Закон Амдала

$$S_p = \frac{1}{f + \frac{1-f}{p}}$$



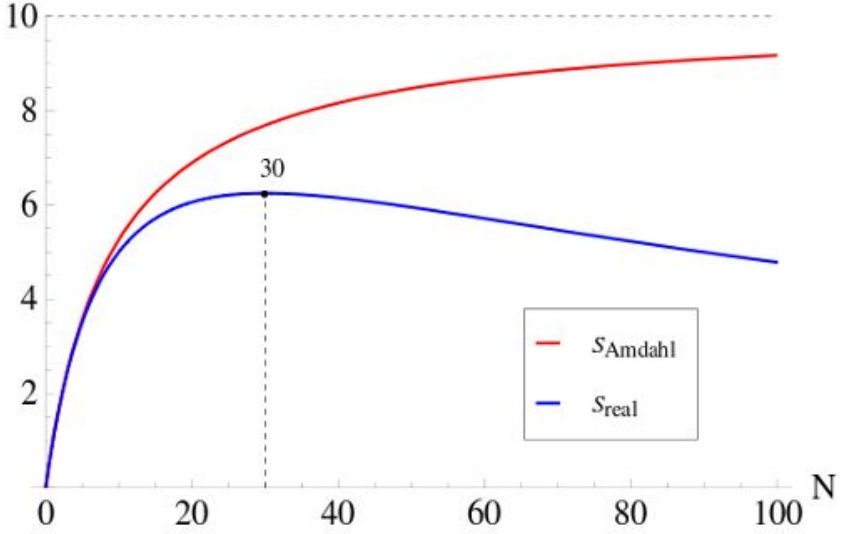
$$\lim_{p \to \infty} S_p = \frac{1}{f}$$

Если f = 0.25, то ускорение больше чем в 4 раза невозможно получить при любом p

Влияние коммуникационной сети

Потери времени на межпроцессорный обмен сообщениями.

Они могут не только снизить ускорение, но и замедлить вычисления по сравнению с последовательными вычислениями



$$S_p = \frac{T_1}{T_p} = \frac{T_1}{\left(f + \frac{1-f}{p}\right)T_1 + N_{comm} \cdot t_{comm}} =$$

$$= \frac{1}{f + \frac{1-f}{p} + \frac{N_{comm} \cdot t_{comm}}{T_1}} = \frac{1}{f + \frac{1-f}{p} + c}$$

N_{comm} - число коммуникационных операций

t_{comm} - время выполнения одной операции

Домашнее задание

Задача № 1

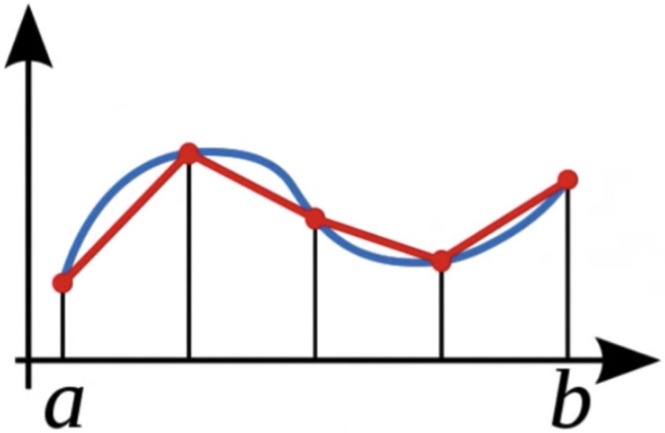
(Нахождение интеграла с использованием MPI)

Постановка задачи.

$$\left(\int_0^1 \frac{4}{(1+x^2)} dx \right) \qquad (1)$$

Решить определенный интеграл (1) методом трапеций.

Домашнее задание



Trapezoid rule for integrating $\int_a^b = f(x)dx$

with $h = (b - a)/n$ is

$f(x) \approx \frac{h}{2}(f(x_0) + f(x_n)) + h\sum_{i=1}^{n-1} f(x_i)$

where $x_i = a + ih, i = 0, 1, ..., n$

Given p processes, each process can work on n/p intervals

Note: for simplicity will assume n/p is an integer

process	interval
0	$[a, a + \frac{n}{p}h]$
1	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
...	...
p-1	$[a + (p - 1)\frac{n}{p}h, b]$

Полезная видеолекция: https://www.youtube.com/watch?v=tqTX8nBM774