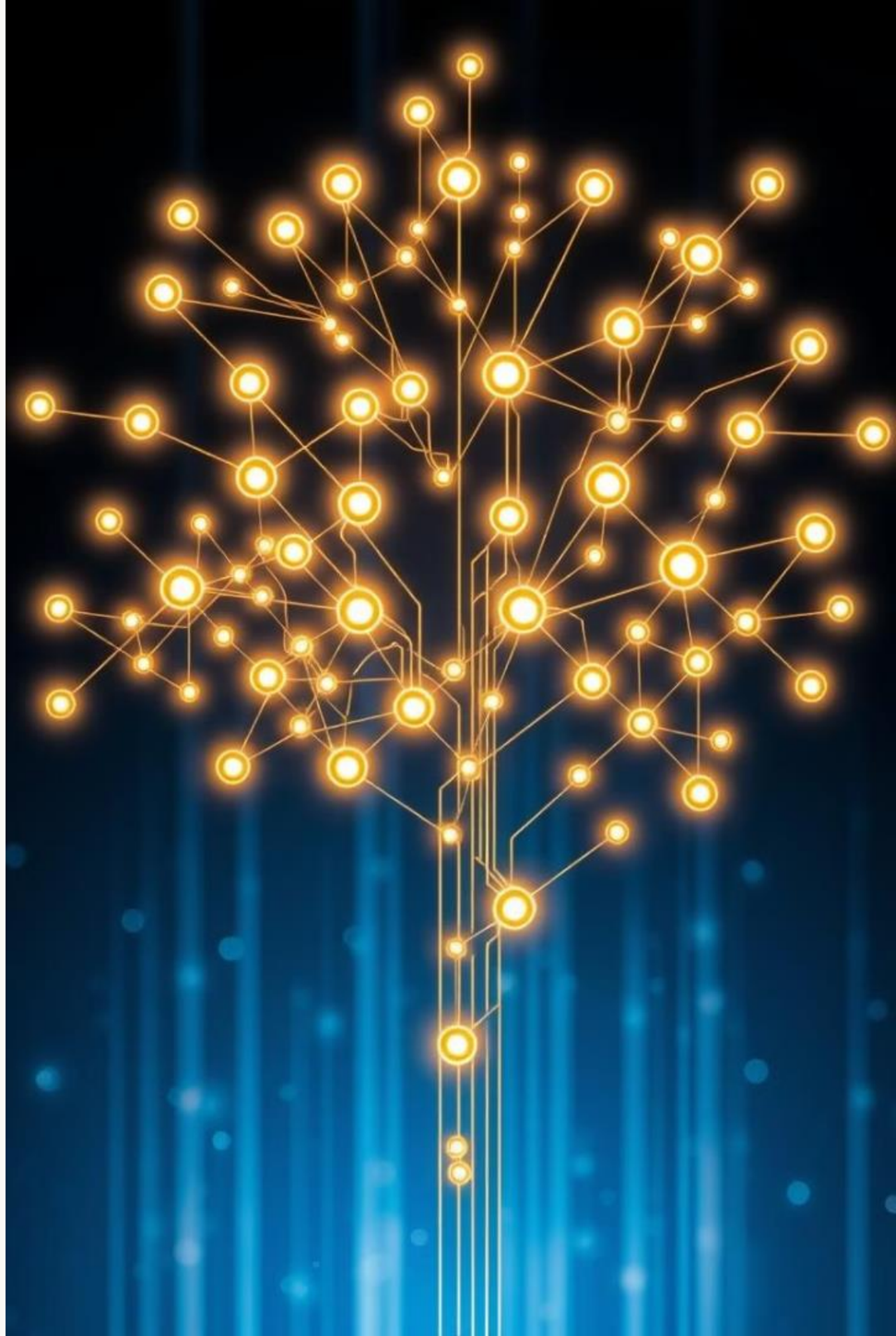
Древовидная структура коммуникации в параллельных вычислениях



Представим, что у нас есть восемь процессов, каждый из которых имеет число для суммирования. Пусть процессы с четными рангами отправляют свои данные процессу с нечетным рангом, большим на один. Нечетные процессы получают данные непосредственно от процессов, находящихся ниже их по рангу. Таким образом, мы за один временной шаг выполняем половину работы.

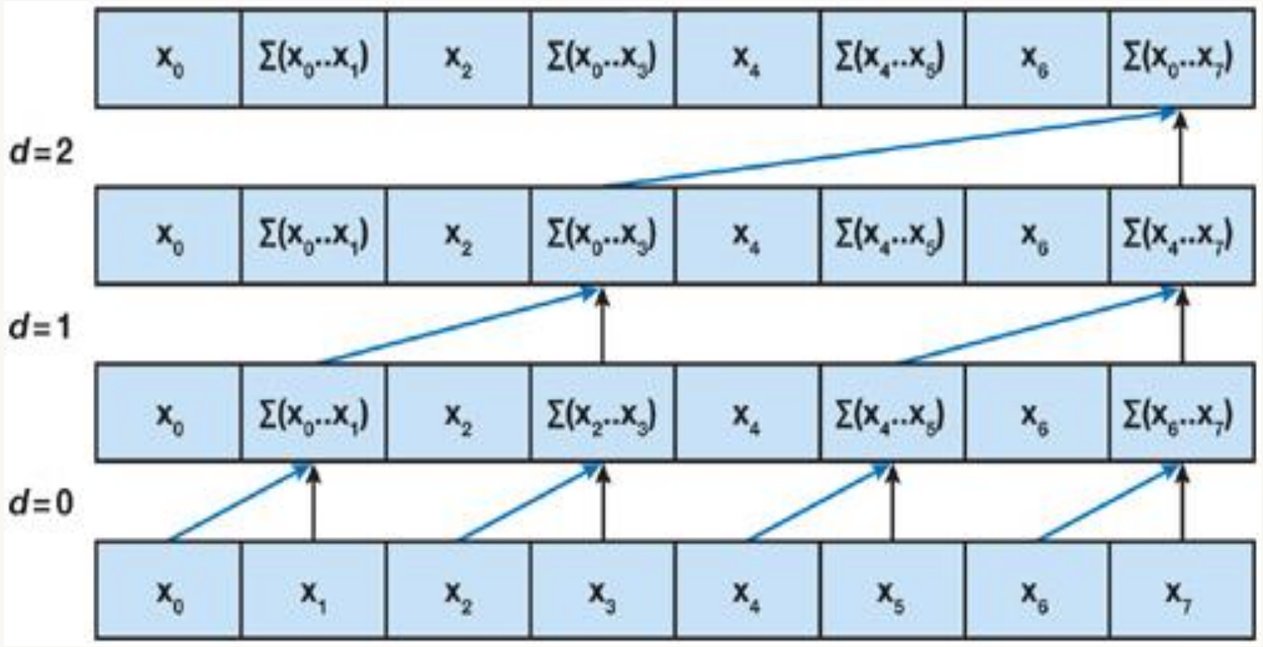


Рисунок 1 (Быстрое суммирование)

От этого момента мы можем повторить процесс с нечетными процессами, дополнительно разделяя их. Суммирование, выполненное таким образом, создает древовидную структуру (см. Рисунок 1). MPI реализовала быстрые суммирования и многое другое в методах, называемых "коллективной коммуникацией". Метод, показанный на рисунке, называется Reduce.

Кроме того, MPI имеет методы, которые также могут эффективно распространять информацию. Представьте себе, что стрелки на рисунке Быстрого суммирования направлены в обратную сторону. Это функция Broadcast MPI.
Следует отметить, что Рисунок 1 является упрощением того, как MPI выполняет коллективную коммуникацию. На практике это отличается в каждой реализации и обычно более эффективно, чем упрощенный пример. MPI предназначена для оптимизации этих методов под капотом, поэтому нам нужно только правильно применять эти функции.

Простейший случай — это глобальная сумма

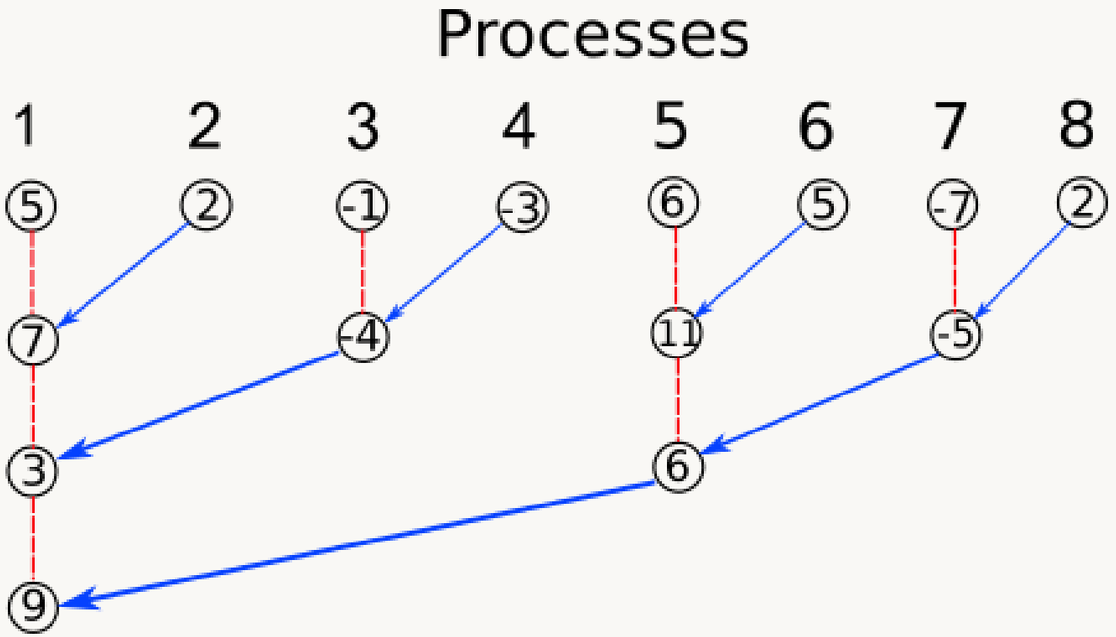


Рисунок 2 (глобальная сумма)

В этом подходе работа по вычислению операции (в данном случае суммы) распределяется между некоторыми участвующими процессами.

В этом примере процессы 1, 3, 5 и 7 вычисляют промежуточные результаты и передают их следующему процессу. Это приводит к тому, что большая часть вычислений выполняется одновременно. (например, корневой узел не отвечает за вычисление всего результата).

Чтобы продемонстрировать эту идею, мы можем реализовать операцию глобального суммирования, используя древовидную топологию с примитивами Send и Recv.

```
from mpi4py
import MP
limport numpy as np
import random

MAX_CONTRIB = 10
def global_sum(my_contrib, my_rank, p, comm):
    sum_val = my_contrib
    bitmask = 1
    done = False
    while not done and bitmask < p:
        partner = my_rank ^ bitmask
        if my_rank < partner:
            temp = np.array(0, dtype='i’)
            comm.Recv(temp, source=partner, tag=0)
            sum_val += temp
            bitmask <<= 1
        else:
            comm.Send(np.array(sum_val, dtype='i'),
dest=partner, tag=0)
            done = True
    # Valid only on rank 0
    return sum_val
```

```
def main():

    comm = MPI.COMM_WORLD
    my_rank = comm.Get_rank()
    p = comm.Get_size()

    # Инициализируем генератор случайных чисел
    # для каждого процесса

    random.seed(my_rank + 1)
    my_contrib = random.randint(0, MAX_CONTRIB - 1)
    print(f"Proc {my_rank} > my_contrib = {my_contrib}")

    # Считаем глобальную сумму
    sum_val = global_sum(my_contrib, my_rank, p, comm)

    if my_rank == 0:
        print(f"Proc {my_rank} > global sum = {sum_val}")

if __name__ == "__main__":
    main()
```

Результат выполнения кода:

```
C:\Users\Admin\mpi>mpiexec -n 8 "C:\Program Files\Python312\python.exe"  treempi.py
Proc 1 > my_contrib = 0
Proc 5 > my_contrib = 9
Proc 7 > my_contrib = 3
Proc 3 > my_contrib = 3
Proc 6 > my_contrib = 5
Proc 2 > my_contrib = 3
Proc 4 > my_contrib = 9
Proc 0 > my_contrib = 2
Proc 0 > global sum = 34
```

Сопряжение процессов осуществляется с помощью побитовой операции «исключающее ИЛИ» - «^»

X	Y	X^Y
0	0	0
0	1	1
1	0	1
1	1	0

Ниже представлена таблица, показывающая процесс, сопряженный с 8 процессами (r = my_rank, остальные заголовки столбцов — битовые маски)

r	t	001	010	100
0	000	001	010	100
1	001	000	x	x
2	010	011	000	x
3	011	010	x	x
4	100	101	110	000
5	101	100	x	x
6	110	111	100	x
7	111	110	x	x

Эта схема требует, чтобы p, количество процессов, было степенью двойки.

Хотя создание собственной реализации древовидной структуры коммуникации является хорошим упражнением для понимания принципов, MPI реализует это за нас в функции Reduce.

Следующий код, для вычисления интеграла методом трапеций, использует функцию Reduce, чтобы воспользоваться этим преимуществом.

```
import numpy
import sys
from mpi4py import MPI
from mpi4py.MPI import ANY_SOURCE

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# принимает аргументы командной строки [a, b, n]
a = (sys.argv[1])
b = (sys.argv[2])
n = (sys.argv[3])
# произвольно определяем функцию для интегрирования
def f(x):
    return x*x


# это последовательная версия правила трапеций
# параллелизация происходит путем деления диапазона между процессами
```

```
def integrateRange(a, b, n):
    integral = -(f(a) + f(b))/2.0
    # n+1 endpoints, but n trapazoids
    for x in numpy.linspace(a,b,n+1):
        integral = integral + f(x)
    integral = integral* (b-a)/n
    return integral
# h - шаг. n - общее количество трапеций
h = (b-a)/n
# local_n - количество трапеций, которые каждый процесс будет вычислять
local_n = n/size
# вычисляем интервал, с которым работает каждый процесс
# local_a - начальная точка, а local_b - конечная точка
local_a = a + rank*local_n*hlocal_b = local_a + local_n*h
```

```
# инициализация переменных. mpi4py требует, чтобы мы передавали объекты numpy.
integral = numpy.zeros(1)
total = numpy.zeros(1)

# выполняем локальные вычисления. Каждый процесс интегрирует свой собственный интервал
integral[0] = integrateRange(local_a, local_b, local_n)
# коммуникация
# корневой процесс получает результаты с помощью коллективного «reduce»
comm.Reduce(integral, total, op=MPI.SUM, root=0)
# корневой процесс выводит результаты
if comm.rank == 0:
        print "With n =", n, "trapezoids, our estimate of the integral from"\
              , a, "to", b, "is", total
```

Могут возникнуть ситуации, когда вам потребуется изменить эту схему для реализации пользовательских операций.

Древовидная структура коммуникации хорошо масштабируется для очень больших проблем.

Спасибо за внимание!