

Лабораторная работа №1:

*«Знакомство с основными синтаксическими конструкциями
технологии MPI»*

Группа: 3640301/00201
Студент: Машаев Н.К.
Преподаватель: Абрамов А.Г.

Содержание

1	О работе	2
1.1	Цель работы	2
1.2	Задачи работы	2
1.3	Ход выполнения работы	2
2	Исследование возможностей технологии MPI	2
2.1	Общие сведения о технологии MPI	2
2.2	Параллелизация программы "Hello, world!"	3
2.3	Параллелизация цикла технологиями MPI	6
2.4	Вычисление числа π через вычисление определенного интеграла . .	9
3	Выводы	11
4	Приложение	12

1 О работе

1.1 Цель работы

Цель работы заключается в изучении базовых принципов построения простейших параллельных программ средствами библиотеки MPI

1.2 Задачи работы

Задачами работы являются разработка и изучение программ, использующих простейшие базовые конструкции технологии MPI, а именно:

1. Изучить работу части кода, помещенной между вызовами `MPI_Init(&argc, &argv)` и `MPI_Finalize()`;
2. Исследовать применение `MPI_Wtime()` для измерения времени выполнения программы процессом. Для разрешения системного таймера использовать функцию `MPI_Wtick()`;
3. Реализовать программу вычисления числа π через расчет определенного интеграла.

1.3 Ход выполнения работы

- Используя средства технологии MPI распараллеливания, вывести на экран сообщение приветствия из каждого процесса. Провести запуск программы для разного числа процессов;
- Определить время, затрачиваемое на выполнение некоторого фрагмента кода, при помощи вызова `MPI_Wtime()`. В качестве исследуемого кода использовать программу подсчета числа простых чисел и чисел, являющихся степенью двойки, на некотором промежутке;
- Для предыдущего пункта определить разрешение системного таймера функцией `MPI_Wtick()` и провести исследование влияния на время выполнения числа процессов;
- Вычислить значение числа π путем вычисления определенного интеграла. Провести исследование влияния на время выполнения числа процессов.

2 Исследование возможностей технологии MPI

2.1 Общие сведения о технологии MPI

Message Passing Interface(MPI) — программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющих одну задачу. MPI является наиболее распространенным стандартом интерфейса обмена данных в параллельном программировании и представляет из

себя библиотеку функций, включающую в себя: множество функций передачи сообщений между двумя конкретными процессами; развитый набор функций для выполнения коллективных операций, работы с группами процессов и коммутаторами; функции для определения пользовательских типов данных, создания топологий связей между процессами и т.д.

MPI-программа в общем случае представляет собой совокупность автономных процессов, функционирующих в своем адресном пространстве под управлением своих собственных программ и взаимодействующих посредством стандартного набора библиотечных процедур для передачи и приема сообщений. Таким образом, MPI-программа реализует MPMD-модель программирования (Multiple Program - Multiple Data). На практике чаще всего реализуется модель SPMD (Single Program - Multiple Data), когда все процессы исполняют различные ветви одной и той же программы, что в большинстве случаев достигается путем разделения обрабатываемых данных на части пропорционально числу процессов (модель распараллеливания по данным).

В первую очередь MPI ориентирован на системы с распределенной памятью, то есть когда затраты на передачу данных велики, в то время как OpenMP ориентирован на системы с общей памятью (многоядерные с общим кэшем). Обе технологии могут использоваться совместно, чтобы оптимально использовать в кластере многоядерные системы.

2.2 Параллелизация программы "Hello, world!"

Реализуем простейшую MPI программу, в которой каждый запущенный процесс печатает на экран сообщение приветствия.

Любая прикладная MPI-программа (приложение) должна начинаться с вызова функции инициализации MPI: функции `MPI_Init`. В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммутатором `MPI_COMM_WORLD`. Эта область связи объединяет все процессы приложения. Процессы в группе упорядочены и пронумерованы от 0 до $N-1$, где N равно числу процессов в группе. Кроме этого, создается предопределенный коммутатор `MPI_COMM_SELF`, описывающий свою область связи для отдельного процесса.

Выполнение большинства MPI функций сопровождается возвратом кода ошибки. В случае успешного выполнения происходит возврат `MPI_SUCCESS`.

Для определения числа процессоров в области связи заданного коммутатора используется функция `MPI_Comm_size`. При этом до создания явным образом групп и связанных с ними коммутаторов единственными значениями коммутатора являются `MPI_COMM_WORLD` и `MPI_COMM_SELF`, которые создаются автоматически при инициализации MPI.

За определение номера процесса отвечает `MPI_Comm_rank`, которая возвращает номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне от 0 до $m - 1$, где m — число процессов в заданном коммутаторе.

По завершению программы необходимо выполнить `MPI_Finalize`. Данная функция закрывает все MPI-процессы и ликвидирует все области связи.

Таким образом, идея реализации программы "Hello, world!" следующая:

- Инициализируем область параллельных вычислений путем вызова `MPI_Init`;
- Определим номер процесса(ранг) и общий размер коммунатора(число нитей) при помощи функций `MPI_Comm_rank` и `MPI_Comm_world` соответственно;
- Выведем на экран общее число процессов через мастер процесс(процесс с рангом 0);
- Из каждой нити выведем сообщение приветствия пользователю;
- Завершим работу параллельной области, выполнив `MPI_Finalize()`.

Код параллельной области программы приведен ниже.

Листинг 2.1: Программа вывода приветствия на экран из разных процессов(hello_mpi)

```
1  MPI_Init(&argc, &argv);
2  start_time = MPI_Wtime();
3
4  MPI_Comm_rank(MPI_COMM_WORLD, &ps_rank);
5
6  if (ps_rank == 0)
7  {
8      MPI_Comm_size(MPI_COMM_WORLD, &ps_numb);
9      printf("Number of processes: %d\n", ps_numb);
10 }
11 MPI_Barrier(MPI_COMM_WORLD);
12 MPI_Bcast(&ps_numb, 1, MPI_INT, 0, MPI_COMM_WORLD);
13 printf("Hello world from %d of %d processes\n", ps_rank, ps_numb)
14 ;
15
16 end_time = MPI_Wtime();
17 loc_time = end_time - start_time;
18 MPI_Reduce(&loc_time,
19 &time,
20 1,
21 MPI_DOUBLE,
22 MPI_MAX,
23 0,
24 MPI_COMM_WORLD);
25
26 MPI_Finalize();
27
28 if (ps_rank == 0)
29 {
30     printf("Computational time: %12.11e s\n", time);
31 }
```

Необходимо, чтобы информация об общем числе процессов в коммуникаторе отображалась первой необходимо сделать так, чтобы процесс с рангом 0 гарантированно определял число нитей и отправлял эту информацию другим процессам раньше, чем другие процессы начнут выполнять свою часть кода, связанную с выводом. Для этого используется барьер `MPI_Barrier`. Функция барьерной синхронизации `MPI_Barrier` блокирует вызывающий процесс, пока все процессы группы не вызовут ее. В каждом процессе управление возвращается только тогда, когда все процессы в группе вызовут процедуру.

Формально в данном случае вызов `MPI_Barrier` бесполезен, поскольку сразу после нее используется функция `MPI_Bcast` для широковещательной рассылки данных(в данном случае — числа процессов в `MPI_COMM_WORLD` коммуникаторе), являющаяся коллективной. Любая коллективная функция выполняет неявную синхронизацию процессов.

Также в данном случае используются функция `MPI_Wtime` для определения времени работы процесса и функция `MPI_Reduce` для определения максимального времени выполнения среди всех процессов, что по существу будет являться временем выполнения параллельной части программы.

При увеличении числа процессов время выполнения программы падает, так как вся деятельность процесса заключается в выводе сообщения в поток вывода `stdout`, а время выполнения программы в общем крайне малое: число выводов с увеличением числа нитей растет, а выходной поток один (рис.1).

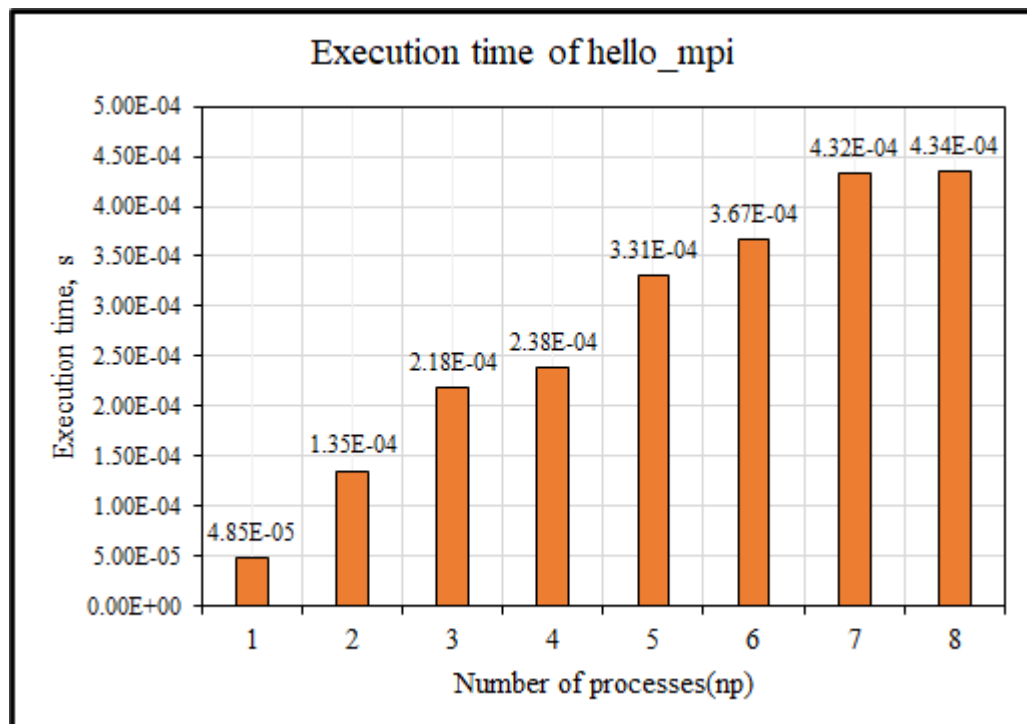


Рис. 1: Время выполнения `hello_mpi` с разным числом процессов

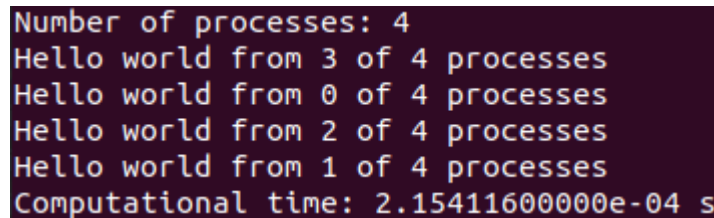
Для компиляции параллельной программы, написанной на языке C++ и использующей библиотеку MPI, следует выполнить команду:

```
mpic++ -o hello_mpi hello_mpi.cpp
```

Задание числа процессов происходит при запуске командой:

$$mpirun -np N \text{ hello_mpi},$$

где N - общее число процессов. Результат выполнения программы с четырьмя процессами приведен на рис.2.



```
Number of processes: 4
Hello world from 3 of 4 processes
Hello world from 0 of 4 processes
Hello world from 2 of 4 processes
Hello world from 1 of 4 processes
Computational time: 2.15411600000e-04 s
```

Рис. 2: Результат выполнения hello_mpi с четырьмя процессами

2.3 Параллелизация цикла технологиями MPI

Исследуем возможности параллелизации больших циклов средствами MPI. В качестве методического расчета используется задача определения количества простых чисел и степеней двойки в заданном интервале. В данном случае используется интервал от 0 до 100000000.

Листинг 2.2: Программа для определения числа простых чисел и степеней двойки(numb_mpi)

```
1  MPI_Init(&argc, &argv);
2
3  MPI_Comm_rank(MPI_COMM_WORLD, &ps_rank);
4  MPI_Comm_size(MPI_COMM_WORLD, &ps_numb);
5
6  start_time = MPI_Wtime();
7  for (unsigned int i = ps_rank; i < CHUNK_SIZE; i += ps_numb)
8  {
9      if (is_prime(i))
10     {
11         loc_prime_counter += 1;
12     }
13     if (is_pow_2(i))
14     {
15         loc_deg_2_counter += 1;
16     }
17 }
18 MPI_Reduce(&loc_prime_counter,
19 &prime_counter,
20 1,
21 MPI_UNSIGNED,
22 MPI_SUM,
23 0,
24 MPI_COMM_WORLD
```

```

25 );
26 MPI_Reduce(&loc_deg_2_counter,
27 &deg_2_counter,
28 1,
29 MPI_UNSIGNED,
30 MPI_SUM,
31 0,
32 MPI_COMM_WORLD
33 );
34 end_time = MPI_Wtime();
35 loc_time = end_time - start_time;
36
37 MPI_Reduce(&loc_time,
38 &time,
39 1,
40 MPI_DOUBLE,
41 MPI_MAX,
42 0,
43 MPI_COMM_WORLD
44 );
45
46 if (ps_rank == 0)
47 {
48     printf("Size of prime number\'s amount: %d\n", prime_counter);
49     printf("Number of degree of two: %d\n", deg_2_counter);
50     printf("Time per clock: %12.11e s\n", MPI_Wtick());
51     printf("Computational time: %12.11e s\n", time);
52 }
53 MPI_Finalize();

```

Ниже представлена реализация функций `is_prime` и `is_pow_2`

Листинг 2.3: Функция для определения является ли число простым

```

1 bool is_prime(unsigned int val)
2 {
3     if (val < 2)
4         return false;
5     unsigned int n = static_cast<unsigned int>(std::sqrt(val)) + 1;
6     for (unsigned int i = 2; i < n; ++i)
7     {
8         if (val % i == 0)
9         {
10             return false;
11         }
12     }
13     return true;
14 }

```

Листинг 2.4: Функция для определения является ли число степенью двойки


```

1  bool is_pow_2(unsigned int val)
2  {
3      if (val < 1)
4          return false;
5          return !(val & (val - 1));
6  }

```

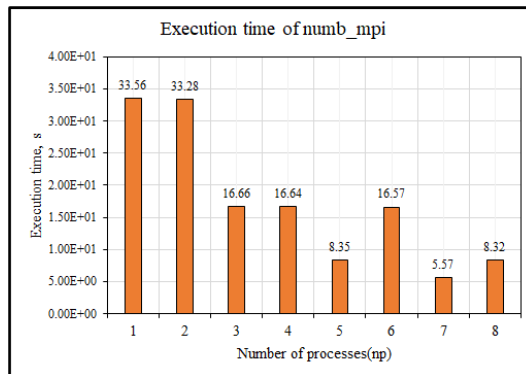
При назначении блока итераций цикла для конкретного процесса используется его ранг и общее количество формирующих коммунитор MPI_COMM_WORLD процессов.

Промежуточные результаты подсчета количества простых чисел и степеней двойки для каждого из процесса записываются в переменные loc_prime_counter и loc_deg_2_counter соответственно

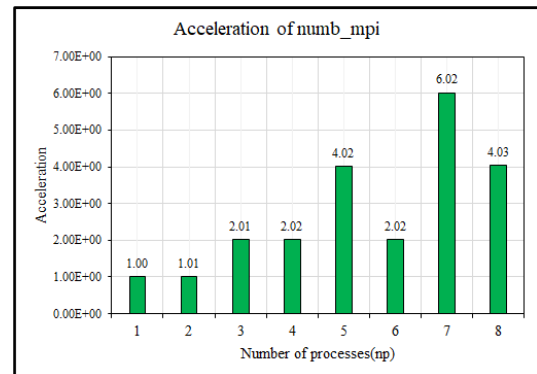
Затем для суммирования результатов работы каждого из процессов используется функция MPI_Reduce, которая объединяет элементы входного буфера каждого процесса в группе, используя операцию MPI_SUM, и возвращает объединенное значение в выходной буфер процесса с номером 0.

Для вычисления времени работы с циклом используется функция MPI_Wtime() в связке с функцией MPI_Reduce, где редуцирование происходит посредством вызова операции MPI_MAX. С помощью MPI_Wtick можно получить разрешение системного таймера, то есть сколько времени приходится на один такт процессора. На локальной машине данное время составляет 10^{-9} секунд.

Исследуем эффективность параллелизации данной программы в зависимости от числа процессов.



(а) Зависимость времени выполнения от числа процессов



(б) Зависимость ускорения от числа процессов

Рис. 3: Исследование параллелизации программы numb_mpi

Таблица 1: Исследование параллелизации цикла при помощи MPI

Число процессов	1	2	3	4	5	6	7	8
Время,с	33.56	33.28	16.66	16.64	8.35	16.57	5.57	8.32
Ускорение	1.00	1.01	2.01	2.02	4.02	2.02	6.02	4.03

Ускорение в данном случае вычисляется как отношение времени выполнения

программы одним процессом к времени выполнению программы с рассматриваемым числом процессов. Видно (рис.3), что скорость выполнения программы на нескольких процессах выше чем при одном, однако ускорение возрастает немонотонно с их увеличением. Это может быть связано с неравномерной нагрузкой процессов: несмотря на видимую симметричность размеров блоков стоит учитывать что время выполнения функций `is_row_2` и особенно `is_prime` существенным образом зависят от размеров числа. Соответственно, при неудачном выборе числа процессов в некоторых из блоков могут преобладать числа с большей разрядностью по отношению к другим, что приведет к большему времени выполнения данного блока процессом, а значит и к большему времени выполнения параллельной области.

2.4 Вычисление числа π через вычисление определенного интеграла

Вычисление числа π происходит путем вычисления определенного интеграла

$$\int_0^1 \frac{4}{1+x^2} \quad (1)$$

на промежутке от нуля до единицы.

Код программы приведен ниже. Интеграл вычисляется по методу прямоугольников. Число участков интегрирования `CHUNK_NUMB = 100000000`.

Листинг 2.5: Вычисление числа π (`pi_mpi`)

```

1  MPI_Init(&argc, &argv);
2
3  MPI_Comm_rank(MPI_COMM_WORLD, &ps_rank);
4  MPI_Comm_size(MPI_COMM_WORLD, &ps_num);
5
6  start_time = MPI_Wtime();
7  for (unsigned int i = ps_rank; i < CHUNK_NUMB; i += ps_num)
8  {
9      curr_x = x_start + (i + 0.5) * x_step;
10     loc_res += func(curr_x);
11 }
12 loc_res *= x_step;
13 MPI_Reduce(&loc_res,
14 &pi,
15 1,
16 MPI_DOUBLE,
17 MPI_SUM,
18 0,
19 MPI_COMM_WORLD);
20 end_time = MPI_Wtime();
21 loc_time = end_time - start_time;
22
23 MPI_Reduce(&loc_time,
24 &time,
```

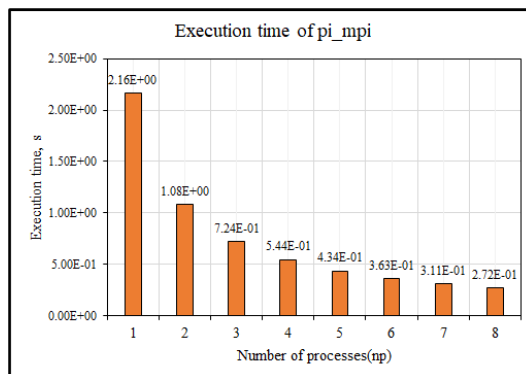
```

25 1,
26 MPI_DOUBLE,
27 MPI_MAX,
28 0,
29 MPI_COMM_WORLD);
30
31 if (ps_rank == 0)
32 {
33     printf("Integration result: %12.11e\n", pi);
34     printf("Numerical error: %12.11e\n", std::abs(pi - PRECISE_PI))
35     ;
36     printf("Computational time: %12.11e s\n", time);
37 }
38 MPI_Finalize();

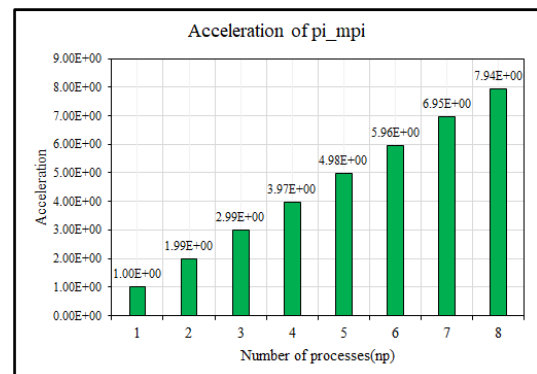
```

При назначении блока итераций цикла для конкретного процесса используется его ранг и общее количество формирующих коммунитор MPI_COMM_WORLD процессов.

Промежуточные результаты суммирования каждого из процессов записываются в соответствующие переменные loc_res, а затем суммируются в переменную pi процесса с рангом 0 применением функции MPI_Reduce с операцией MPI_SUM. В результате вычисления на четырех процессах ошибка составляет $4.61 \cdot 10^{-13}$ от точного решения, что менее 1%.



(а) Зависимость времени выполнения от числа процессов



(б) Зависимость ускорения от числа процессов

Рис. 4: Исследование параллелизации программы вычисления числа π

Таблица 2: Исследование параллелизации вычисление числа π

Число процессов	1	2	3	4	5	6	7	8
Время,с	2.16	1.08	0.72	0.54	0.43	0.36	0.31	0.27
Ускорение	1.00	1.99	2.99	3.97	4.98	5.96	6.95	7.94

Из рис.4 видно, что увеличение числа процессов ведет к увеличению скорости выполнения программы, в отличие от выполнения программы numb_mpi. Это связано с тем, что здесь число вычислений не зависит от номера итерации, а значит

аналогичный алгоритм в данной ситуации дает равномерное распределение нагрузки, что влечет к монотонному росту производительности с ростом процессов.

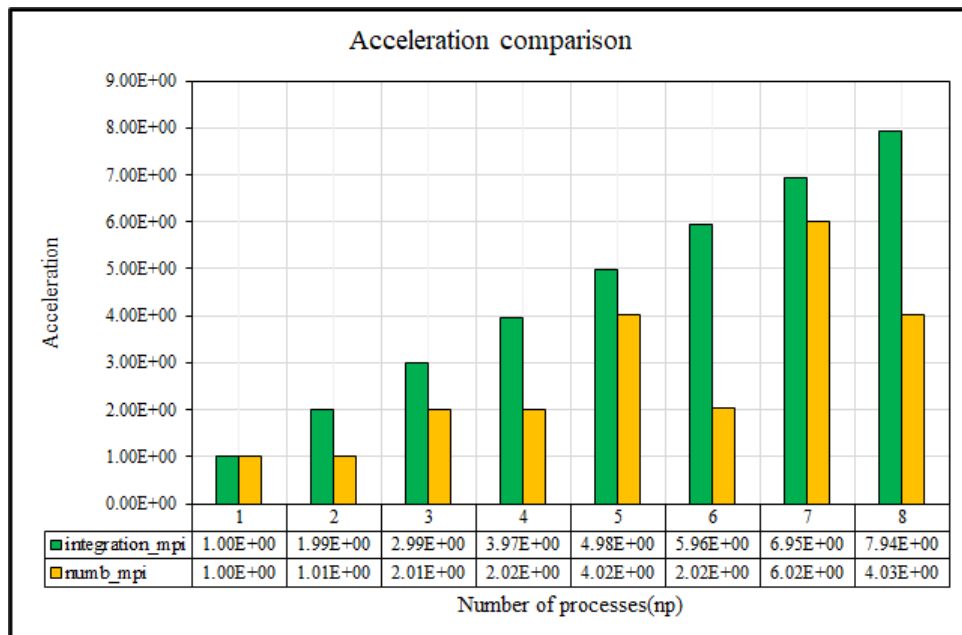


Рис. 5: Сравнение ускорения алгоритма распределения нагрузки между процессами для разных задач

3 Выводы

Проведено знакомство с базовыми конструкциями параллелизма технологии MPI:

1. Реализована программа "Hello,world!", выполняющая приветствие из каждого процесса, а также вывода сведений о числе процессов из процесса с рангом 0. Определение номера(ранга) процесса происходило посредством вызова `MPI_Comm_rank`, числа процессоров - `MPI_Comm_size`;
2. Изучена возможность применения функций `MPI_Bcast` и `MPI_Reduce` для передачи информации между нитями;
3. Исследовано применение параллелизации для работы с большим циклом время выполнения каждой итерации которого зависит от ее номера. Получено, что способ распределения нагрузки SMPD методом не обеспечивает равномерной занятости процессов, в следствие чего эффективность выполнения программы при увеличении числа процессов растет немонотонно;
4. Исследовано применение параллелизации для вычисления числа π методом расчета определенного интеграла. В виду того, что здесь время выполнения итерации не зависит от номера итерации SMPD метод дает равномерное распределение нагрузки среди нитей, в следствие чего скорость вычисления монотонно возрастает.

4 Приложение

https://github.com/nkmashaev/MPI_Labs — ссылка на код программы