

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ
по практической работе №2
по дисциплине «Системы параллельной обработки данных»
Тема: Использование функций обмена данными «точка-точка в
библиотеке MPI.

Студент гр. 0303

Болкунов В.О.

Преподаватель

Татаринов Ю.С.

Санкт-Петербург

2024

Цель работы.

Исследование и разработка параллельной программы, использующей прямой обмен сообщениями между определёнными процессами “точка-точка”.

Постановка задачи.

Вариант 11. Разговор.

Несколько процессов обмениваются между собой сообщениями, требующими ответа источнику сообщений. Тем самым моделируется разговор между несколькими людьми.

Выполнение работы.

Разработана параллельно работающая программа с использованием библиотеки MPI, работающая по следующему алгоритму:

1. Главный процесс (с рангом 0) получает на вход суммарное количество сообщений N которое должно быть передано между процессами.
2. Главный процесс отправляет всем нечётным процессам (1, 3, 5...) (всего их p) количество сообщений которыми они должны обменяться в парах со следующим за ним чётным процессом. Итого каждая пара из чётного и нечётного процесса обмениваются $\lfloor \frac{N}{p} \rfloor$ сообщениями (последняя пара $\lfloor \frac{N}{p} \rfloor + N \text{ div } p$).
3. Далее процессы в парах обмениваются заданным количеством сообщений: процесс с нечётным рангом отправляет сообщение с флагом окончания (true / false) следующему процессу с чётным рангом и ждёт ответа. В свою очередь четный процесс (кроме 0) принимает сообщения до тех пор пока флаг окончания не истинен и отправляет ответ на каждое сообщение.
4. Нечетный процесс после обмена заданным количеством сообщений отправляет ответ главному процессу о завершении работы.
5. Главный процесс ждёт сообщений об успешном завершении работы всех пар процессов от всех процессов с нечётным рангом и завершает работу.

Сеть пети с 3 процессами для данного алгоритма представлена на рисунке 1.

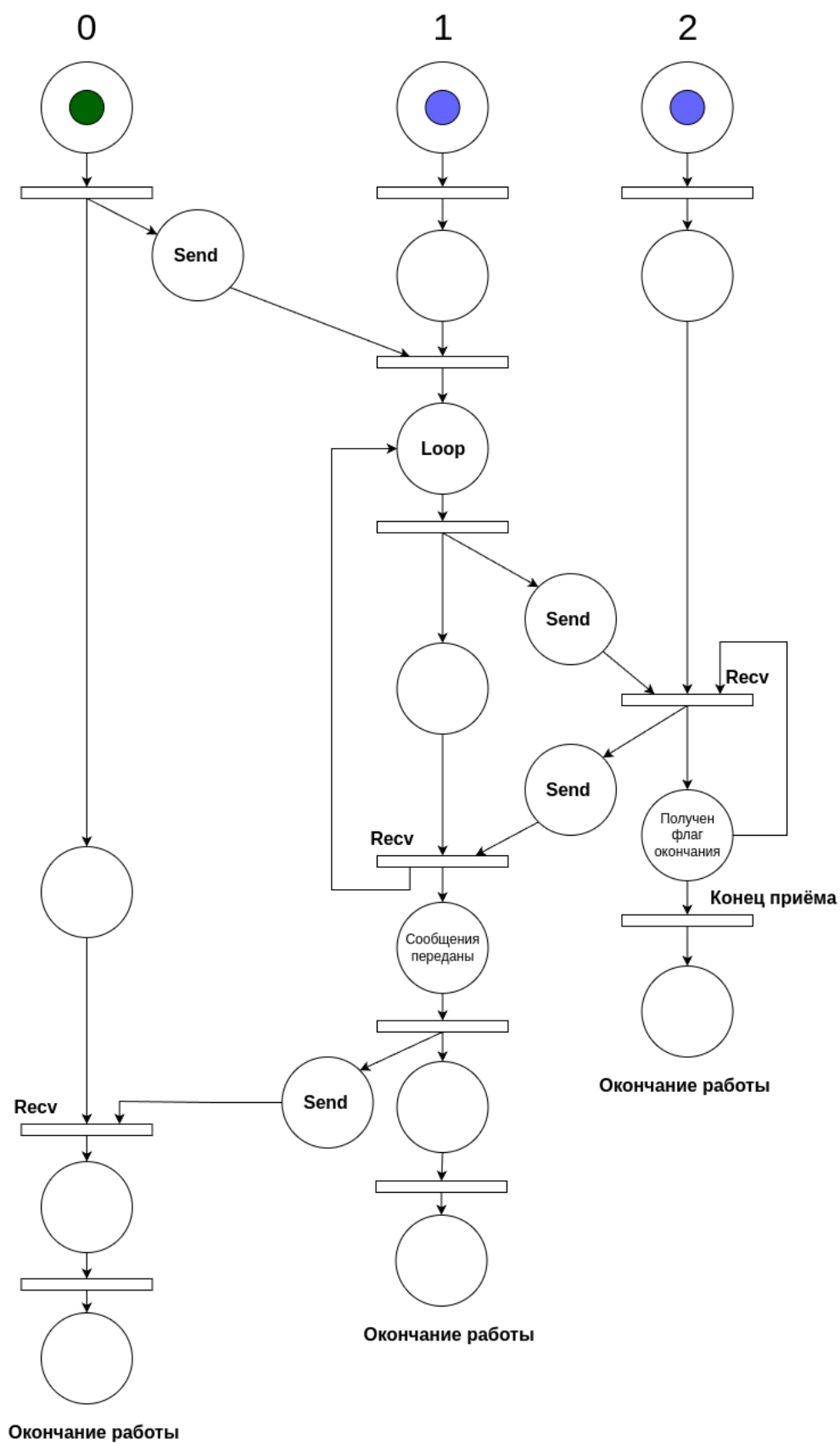
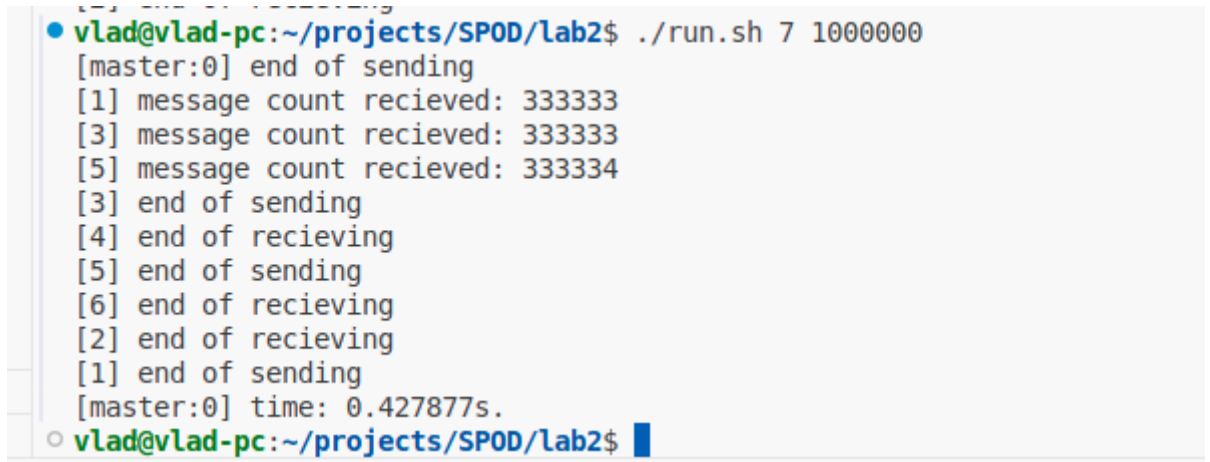


Рисунок 1. Сеть петри

Пример работы программы с 7 процессами и 1000000 сообщениями представлен на рисунке 2. Исходный код программы представлен в приложении А.



```
• vlad@vlad-pc:~/projects/SP0D/lab2$ ./run.sh 7 1000000
[master:0] end of sending
[1] message count recieved: 333333
[3] message count recieved: 333333
[5] message count recieved: 333334
[3] end of sending
[4] end of recieving
[5] end of sending
[6] end of recieving
[2] end of recieving
[1] end of sending
[master:0] time: 0.427877s.
○ vlad@vlad-pc:~/projects/SP0D/lab2$
```

Рисунок 2. Пример работы программы

На данном примере видно, что нечетные процессы (1, 3, 5) получили сообщение с числами 333333 и 333334 - требуемым количеством обменов; обмениваются сообщениями с процессами (2, 4, 6) и завершают свою работу.

Программа была запущена на разном количестве процессоров: от 3 до 49 с шагом 2 и на разном входном количестве требуемых сообщений: 1000, 100000 и 10000000. Время работы программы соответствует времени работы главного процесса, так как он ждёт все группы процессов и последним завершает свою работу. Для лучшего наблюдения зависимости времени работы программы от количества входных сообщений в данных тестах было использовано серверное оборудование с 32-мя процессорными ядрами.

На рисунках 3-5 представлены графики зависимости времени работы программы от количества задействованных процессоров для 1000, 100000 и 10000000 входных сообщений соответственно. На рисунках 6-8 представлены графики ускорения в каждом из описанных случаев.

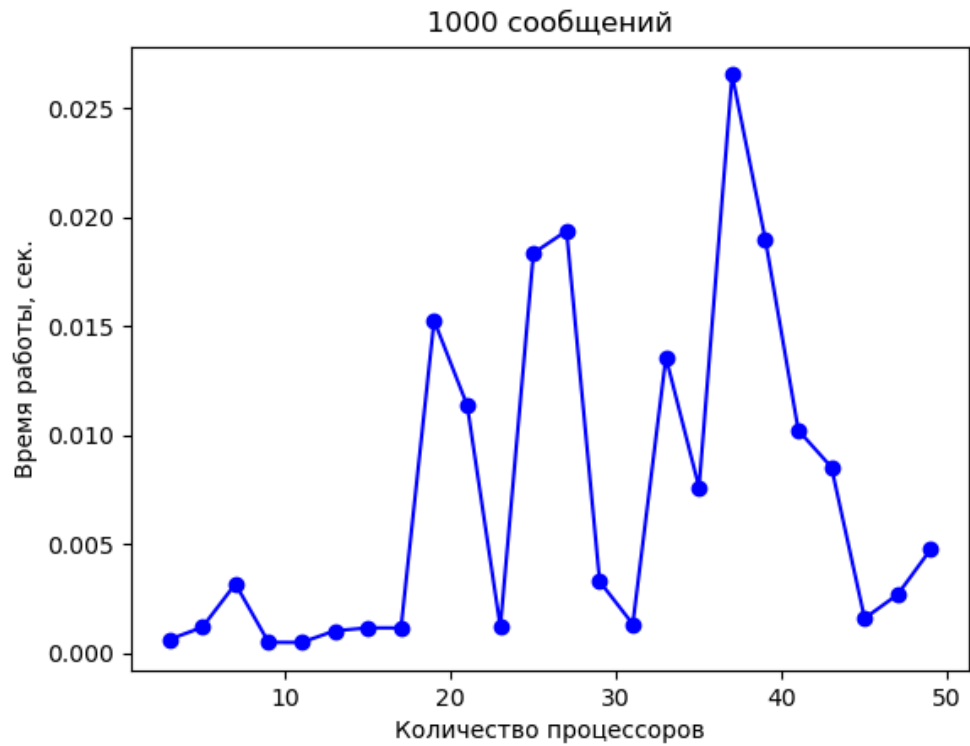


Рисунок 3. График времени работы программы для 1000 входных сообщений

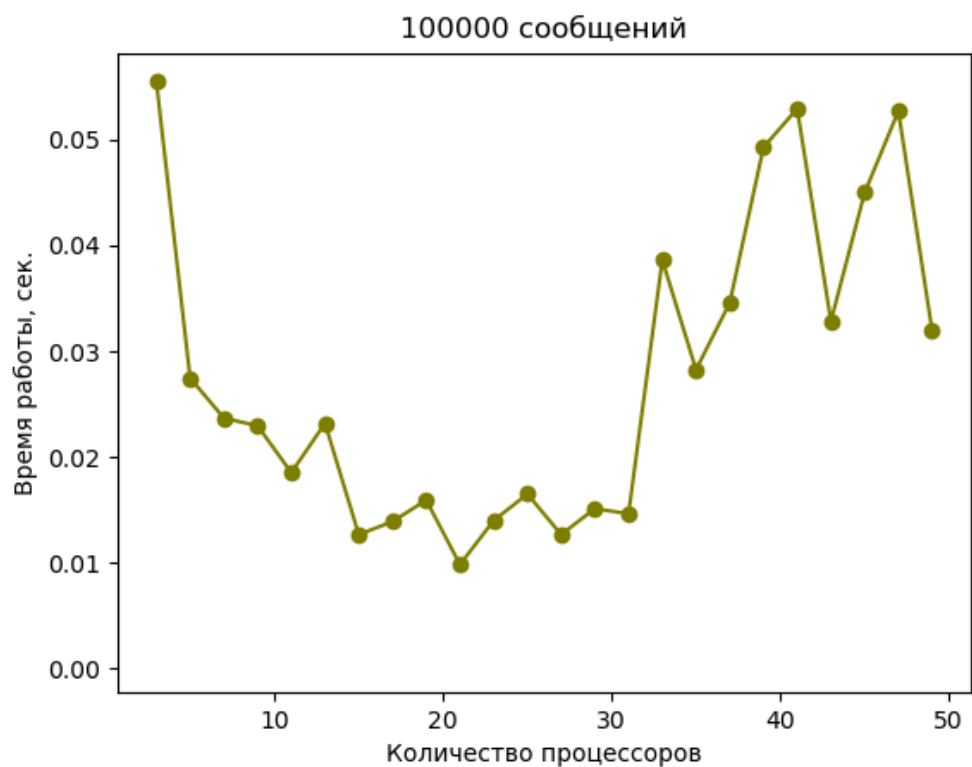


Рисунок 4. График времени работы программы для 100000 входных сообщений

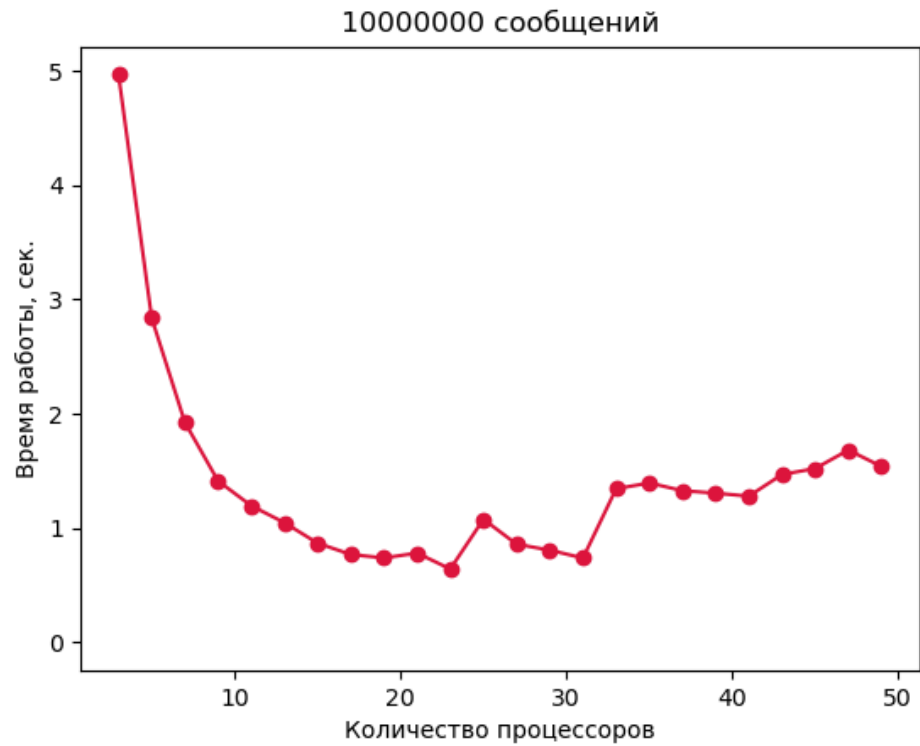


Рисунок 5. График времени работы программы для 10000000 входных сообщений

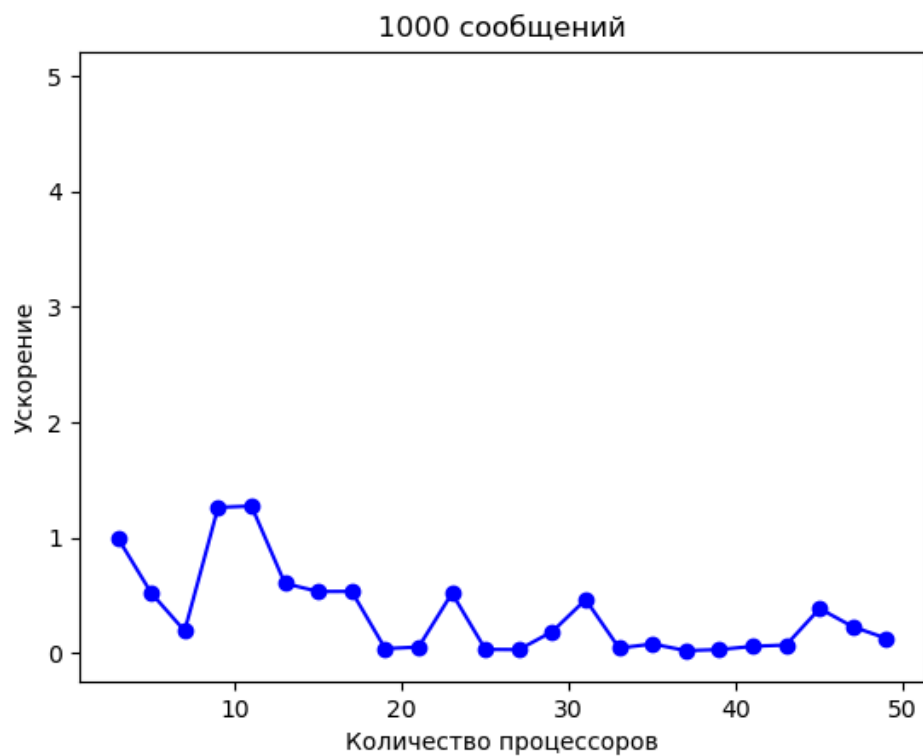


Рисунок 6. График ускорения работы программы для 1000 входных сообщений

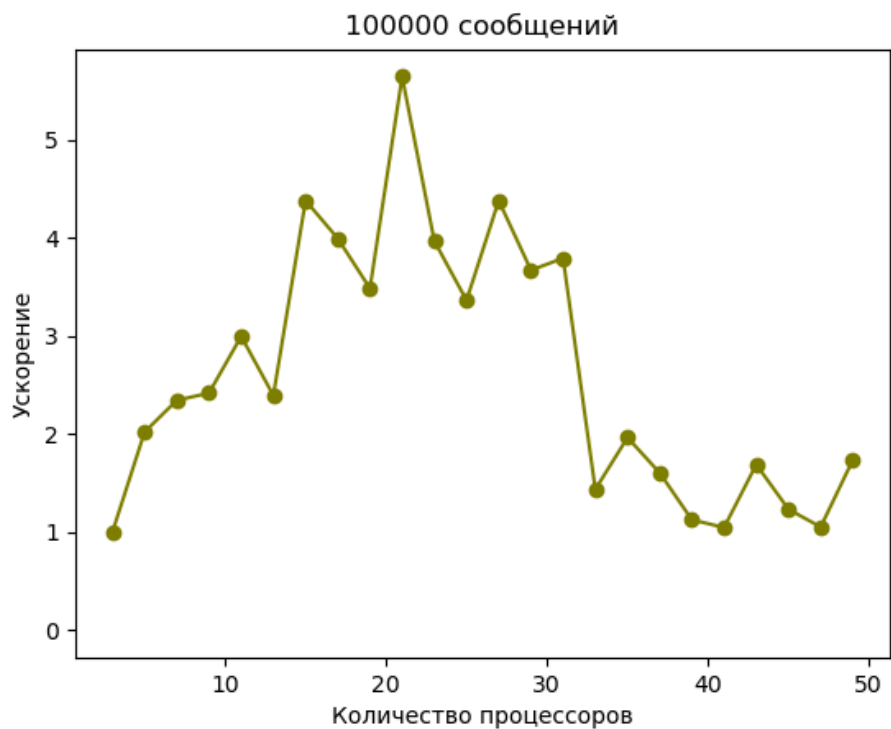


Рисунок 7. График ускорения работы программы для 100000 входных сообщений

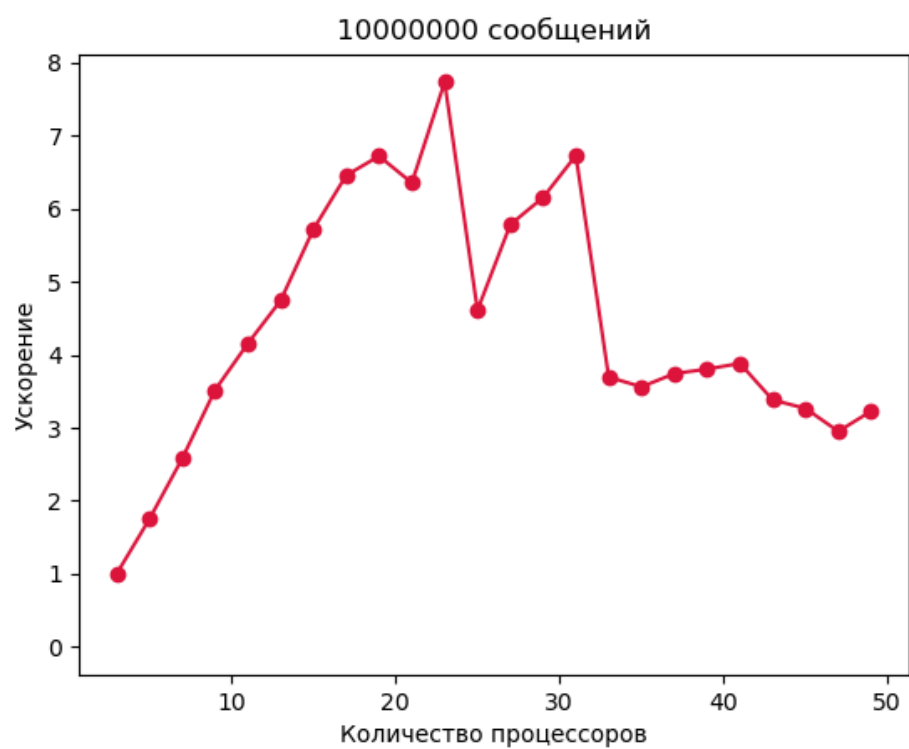


Рисунок 8. График ускорения работы программы для 10000000 входных сообщений

По данным графикам видно, что на небольшом количестве сообщений ускорение за счёт параллелизации незначительное, в силу относительно больших дополнительных расходов на обмен сообщениями. Для большего же количества входных сообщений работа программы значительно ускоряется пропорционально количеству задействованных процессоров. Также при меньшем количестве сообщений график времени работы содержит больше колебаний, так как на небольшое время работы сильнее влияют сторонние процессы, запущенные операционной системой.

Выводы.

В результате выполнения работы была разработана параллельная программа, моделирующая разговор, в основе которой лежит распределение входного количества сообщений по парам процессов, обменивающимся сообщениями. Разработанная программа запущена на разном количестве процессоров: от 2 до 49, и с разными входными данными. В результате тестирования программы построены графики времени работы программы и ускорения.

На малых значениях входных данных увеличение количества процессоров негативно повлияло на производительность из-за относительно больших расходов ресурсов на дополнительные обмены сообщениями. При достаточно больших значениях входных данных увеличение количества задействованных процессоров дало значительное ускорение, за счёт уменьшения количества обменов в каждой паре процессов. При этом, при достижении определённого количества процессов (более 32) время выполнения программы начинает увеличиваться, так как число процессов превышает число доступных процессоров.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

Файл *main.cpp*

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

int main(int argc, char **argv) {
    MPI::Init(argc, argv);
    MPI::Comm &comm = MPI::COMM_WORLD;

    // Общее количество сообщений, которое должно быть отправлено
    const int msgCount = std::atoi(argv[1]);

    int procCount = comm.Get_size();
    int procRank = comm.Get_rank();

    if (procCount % 2 == 0 || procCount < 3) {
        throw MPI::Exception(0);
    }

    int group = procRank % 2;
    int groupsCount = (procCount - 1) / 2;

    double time = 0;

    // master: Отправляем количество сообщений каждой паре и ждём их обменов
    if (procRank == 0) {
        int msgPerPair = msgCount / groupsCount;
        int msgLastPair = msgPerPair + msgCount % groupsCount;

        double start = MPI::Wtime();

        for (size_t i = 0; i < groupsCount - 1; i++) {
            comm.Send(&msgPerPair, 1, MPI::INT, i * 2 + 1, 0);
        }
        comm.Send(&msgLastPair, 1, MPI::INT, (groupsCount - 1) * 2 + 1, 0);

        double end = MPI::Wtime();
        time += end - start;

        std::cout << "[master:0] end of sending " << std::endl;

        start = MPI::Wtime();
        for (size_t i = 0; i < groupsCount; i++) {
            comm.Recv(nullptr, 0, MPI::BOOL, i * 2 + 1, 0);
        }
        end = MPI::Wtime();
        time += end - start;

        std::cout << "[master:0]" << " time: " << time << "s." << std::endl;
    }
    // odd slave: получаем требуемое количество обменов и обмениваемся со следующим
    // чётным процессом
    else if (group == 1) {
        int currentMsgCount;
        bool end = false;
        comm.Recv(&currentMsgCount, 1, MPI::INT, 0, MPI::ANY_TAG);

        std::cout << "[" << procRank << "]" message count recieved: "
        << currentMsgCount << std::endl;
    }
}
```

```

    for (size_t i = 0; i < currentMsgCount - 1; i++) {
        comm.Send(&end, 1, MPI::BOOL, procRank + 1, 0);
        comm.Recv(nullptr, 0, MPI::BOOL, procRank + 1, 0);
    }
    end = true;
    comm.Send(&end, 1, MPI::BOOL, procRank + 1, 0);
    comm.Recv(nullptr, 0, MPI::BOOL, procRank + 1, 0);

    std::cout << "[" << procRank << "]" end of sending" << std::endl;
    comm.Send(nullptr, 0, MPI::BOOL, 0, 0);
}
// even slave: обмениваемся с предыдущим нечётным процессом сообщениями, пока
не придёт флаг окончания
else {
    bool end = false;
    while (!end) {
        comm.Recv(&end, 1, MPI::BOOL, procRank - 1, MPI::ANY_TAG);
        comm.Send(nullptr, 0, MPI::BOOL, procRank - 1, 0);
    }
    std::cout << "[" << procRank << "]" end of recieving" << std::endl;
}

MPI::Finalize();
}

```

Файл build.sh

```
mpicxx.openmpi main.cpp -o lab2
```

Файл run.sh

```
mpiexec.openmpi -n $1 --oversubscribe ./lab2 $2
```