###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

«Программирование многопоточных приложений. POSIX-Threads»

студента 2 курса, группы 21203

***Кондренко Кирилла Павловича***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

*Кандидат технических наук*

А. Ю. Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#__RefHeading___Toc114782465)

[ЗАДАНИЕ 3](#__RefHeading___Toc114782466)

[ОПИСАНИЕ РАБОТЫ 4](#__RefHeading___Toc114782467)

[АНАЛИЗ ПОЛУЧЕННЫХ ДАННЫХ 5](#__RefHeading___Toc5525_2357992727)

[ЗАКЛЮЧЕНИЕ 6](#__RefHeading___Toc114782468)

[Приложение 1. Результаты измерений 7](#__RefHeading___Toc1147824692)

[Приложение 3. Исходный код программы 9](#__RefHeading___Toc114782469)

[Приложение 3. Компиляция и сборка 22](#__RefHeading___Toc1147824691)

# ЦЕЛЬ

Освоить разработку многопоточных программ с использованием *POSIX-Threads* API. Познакомиться с задачей динамического распределения работы между процессорами.

# ЗАДАНИЕ

1. Есть список неделимых заданий, каждое из которых может быть выполнено независимо от другого. Задания могут иметь различный вычислительный вес, то есть требовать при одних и тех же вычислительных ресурсах различного времени для выполнения. Считается, что этот вес нельзя узнать, пока задание не выполнено. После того, как все задания из списка выполнены, появляется новый список заданий. Необходимо организовать параллельную обработку заданий на нескольких компьютерах.
2. Количество заданий существенно превосходит количество процессоров.
3. Программа не должна зависеть от числа компьютеров.
4. Для организации взаимодействия между компьютерами нужно использовать *MPI*, для организации потоков использовать *POSIX-Threads API.*
5. После каждой итерации (после каждого списка задач) каждый *MPI*-процесс должен выводить:
   * количество заданий, выполненных данным процессом за итерацию;
   * значение globalRes
   * совокупное время выполнения заданий на этой итерации
6. После каждой итерации нужно посчитать время дисбаланса

и долю дисбаланса

# ОПИСАНИЕ РАБОТЫ

Итоговая программа фактически представляет собой модель взаимодействия процессов «клиент-сервер», где в качестве сервера выступает один единственный *MPI*-процесс из запущенных («менеджер)», а в качестве «клиентов» — все остальные. То есть, если программа была запущена на (n + 1) процессе, то из них один процесс — «менеджер», остальные - «клиенты». При этом в каждом «клиентском» процессе по 3 потока («Worker», «Tx», «Rx»).

В течение работы программы необходимо балансировать нагрузку (то есть устранять появившийся дисбаланс). Для этого в программе процессы, которые уже завершили свои задания, обращаются к процессам, ещё не завершившим их, и забирают у них часть заданий для собственной работы. Если бы в каждом процессе работал лишь один поток, то с возникали бы проблемы, так как тогда единственному потоку в каждом процессе было бы необходимо постоянно прерывать выполнение заданий (например, после каждого заданий) для запроса заданий у других и обмена заданиями с другими. Очевидно, что это бы на порядки усложнило написание кода и чтение кода, к тому же, производительность программы была бы относительной низкой. Для этого в каждом процессе, выполняющем задания были запущено 2 дополнительных потока («Tx», «Rx»). Поток «Tx», в случае если у процесса осталось мало заданий, обращается к остальным процессам за заданиями. Поток «Rx» «слушает» запросы других процессов и отвечает на них частью списка заданий (которой он делится с остальными). Однако, в такой модели обмена данными непонятно, к какому процессу нужно обращаться за новыми заданиями. Можно, конечно, обращаться к каждому из по-отдельности, или использовать коллективные операции (например, MPI\_Bcast()), однако более хорошим решением показалось использования какого-то одного процесса в качестве «Менеджера». Этот процесс непосредственно не выполняет задания. Он служит некоторой базой данных, предоставляющей информации о том, у какого процесса есть наибольшее количество свободных заданий. Очевидно, что для того чтобы «Менеджер» имел актуальную информацию, либо он само должен постоянно опрашивать все процессы, либо процессы должны при выполнении-получении заданий уведомлять об этом «Менеджера». Второй вариант оказался более предпочтительным, поэтому он и был выбран.

В итоге, программу можно логически разбить 2 части: «Менеджер», «Клиентские процессы-рабочие». При этом каждый «Клиентский процесс» можно ещё подразбить на 3 части:

* часть, отвечающая за подкачку заданий с других клиентов
* часть, которая слушает запросы других клиент
* часть, которая исполняет задания

# АНАЛИЗ ПОЛУЧЕННЫХ ДАННЫХ

Во-первых, полученные данные показывают, что алгоритм балансировки работает, причём доля дисбаланса относительно мала. Однако, можно заметить, что при увеличении количества процессов, увеличивается и время работы (при одних и тех же остальных настройках). Это связано с тем, что при увеличении количества процессов происходило существенно больше межпроцессных коммуникаций в связи с тем, что количество заданий, которое отдаётся процессом существенно меньше, чем общее количество заданий в списке (например, для двух процессов общее количество заданий в списке было — 15000, а размер сообщения для обмена заданиями — всего 50 заданий). В том числе и этим можно объяснить, почему что доля дисбаланса относительно мала — потому что происходило много коммуникаций, и задания распределялись более точно. Однако надо заметить, что программа позволяет настраивать в том числе и такие параметры для получение наибольшей производительности. Ну действительно, если процесс в случае запроса отдаёт другому всего лишь 1 задание, то точность будет очень высокой и доля дисбаланс — очень низкой, однако будет происходить очень много межпроцессных коммуникаций. Если в случае запроса процесс отдаёт другому относительно много заданий, то межпроцессных коммуникаций будет происходить существенно меньше (чем в первом случае), однако при этом и «точность» будет существенно меньше, то есть доля дисбаланса — выше.

# ЗАКЛЮЧЕНИЕ

По результатам работы я пришёл к следующим выводам:

1. Существует 2 основных инструмента для написании многопоточных приложений на языках программирования C/C++ — *OpenMP*, *POSIX-Threads API*. Первый инструмент используется, когда поток должны выполнять практически идентичные задания, второй — когда задания потоков могут сильно варьироваться от потока к потоку.
2. В *POSIX-Threads API* есть 2 основных инструмента синхронизации потоков — *mutex*, *conditional-variables* (условные переменные).
3. Использование модели «Сервер-Клиент» для балансировки нагрузки между процессами может в значительной степени увеличить производительность всей программы.
4. Часто, при балансировке нагрузки между выполняющими задания процессами, необходимо найти «золотую середину» в размер сообщений (количестве заданий, пересылаемых от одного процесса другому) так, чтобы и доля дисбаланса, и время на межпроцессные коммуникации оставались относительно низкими.

# Приложение 1. Результаты измерений

**Измерение при 2 выполняющих задания процессах**

*Использованные параметры:*

* количество заданий в списке на процесс — 15000
* количество списков — 100
* критическое количество заданий — 100
* размер сообщения с заданиями — 50

*Полученные данные:*

* время работы — 48 секунд
* среднее время на глобальную итерацию — 451 мс
* минимальная доля дисбаланса — 6 %
* средняя доля дисбаланса — 11%
* максимальная доля дисбаланса — 14%

**Измерение при 8 выполняющих задания процессах**

*Использованные параметры:*

* количество заданий в списке на процесс — 1500
* количество списков — 100
* критическое количество заданий — 10
* размер сообщения с заданиями — 5

*Полученные данные:*

* время работы — 52 секунды
* среднее время на глобальную итерацию — 489 мс
* минимальная доля дисбаланса — 20 %
* средняя доля дисбаланса — 28%
* максимальная доля дисбаланса — 32%

**Измерение при 12 выполняющих задания процессах**

*Использованные параметры:*

* количество заданий в списке на процесс — 800
* количество списков — 100
* критическое количество заданий — 10
* размер сообщения с заданиями — 5

*Полученные данные:*

* время работы — 49 секунды
* среднее время на глобальную итерацию — 445 мс
* минимальная доля дисбаланса — 35 %
* средняя доля дисбаланса — 42%
* максимальная доля дисбаланса — 60%

**Пример вывода для двух выполняющих задания процессов**

ITERATION 13Process 0Tasks executed 9203Time for iteration 457 msGlobal result 72996.86Process 1Tasks executed 20797Time for iteration 457 msGlobal result 55226.42-----------------------------------------------ITERATION 14Process 0Tasks executed 19803Time for iteration 470 msGlobal result 78391.58Process 1Tasks executed 10197Time for iteration 470 msGlobal result 63075.91-----------------------------------------------ITERATION 15Process 0Tasks executed 9629Time for iteration 429 msGlobal result 86697.21Process 1Tasks executed 20371Time for iteration 429 msGlobal result 67369.93

# Приложение 3. Исходный код программы

**main.c**

#include <mpi.h>  
#include "worker.h"  
#include "manager.h"  
  
int main(int *argc*, char\*\* *argv*) {  
 int provided;  
 MPI\_Init\_thread(&*argc*, &*argv*, **MPI\_THREAD\_MULTIPLE**, &provided);  
  
 int rank;  
 int size;  
  
 MPI\_Comm\_rank(**MPI\_COMM\_WORLD**, &rank);  
 MPI\_Comm\_size(**MPI\_COMM\_WORLD**, &size);  
  
 const int workers\_size = size - 1;  
 const int manager\_rank = workers\_size;  
  
 *MPI\_Group* global\_group;  
 MPI\_Comm\_group(**MPI\_COMM\_WORLD**, &global\_group);  
  
 *MPI\_Group* workers\_group;  
 int ranks[] = {manager\_rank};  
 MPI\_Group\_excl(global\_group, 1, ranks, &workers\_group);  
  
 *MPI\_Comm* workers\_comm;  
 MPI\_Comm\_create(**MPI\_COMM\_WORLD**, workers\_group, &workers\_comm);  
  
 if (rank == manager\_rank) {  
 manage(workers\_size);  
 } else {  
 work(workers\_comm, manager\_rank);  
 }  
  
 MPI\_Finalize();  
}

**manager.c**

#include <mpi.h>  
#include <stdbool.h>  
#include "message.h"  
#include "manager.h"  
  
int find\_process\_with\_max\_available\_tasks\_count(const *size\_t*\* *tasks\_count*, int *workers\_size*) {  
 *size\_t* max\_available\_tasks\_count = *tasks\_count*[0];  
 int ret = 0;  
  
 for (int i = 1; i < *workers\_size*; i++) {  
 if (max\_available\_tasks\_count < *tasks\_count*[i]) {  
 max\_available\_tasks\_count = *tasks\_count*[i];  
 ret = i;  
 }  
 }  
  
 return ret;  
}  
  
void manage(int *workers\_size*) {  
 *size\_t* tasks\_count[*workers\_size*];  
  
 *MPI\_Datatype* message\_data\_type = create\_message\_data\_type();  
  
 while (**true**) {  
 *message\_t* in;  
 *MPI\_Status* status;  
  
 MPI\_Recv(&in,  
 1,  
 message\_data\_type,  
 **MPI\_ANY\_SOURCE**,  
 **MPI\_ANY\_TAG**,  
 **MPI\_COMM\_WORLD**,  
 &status);  
  
 const int source = status.MPI\_SOURCE;  
  
 if (in.type == **TASKS\_UPDATE\_TYPE**) {  
 tasks\_count[source] = (*size\_t*) in.value;  
 }  
 if (in.type == **PROCESS\_RANK\_REQUEST\_TYPE**) {  
 const int response = find\_process\_with\_max\_available\_tasks\_count(tasks\_count, *workers\_size*);  
 MPI\_Send(&response, 1, **MPI\_INT**, source, **DEFAULT\_MANAGER\_RESPONSE\_TAG**, **MPI\_COMM\_WORLD**);  
 }  
 if (in.type == **TERMINATION\_TYPE**) {  
 break;  
 }  
 }  
  
 MPI\_Type\_free(&message\_data\_type);  
}

**message.c**

#include "message.h"  
  
*MPI\_Datatype* create\_message\_data\_type(void) {  
 const int block\_lengths[] = {1, 1};  
 const *MPI\_Aint* offsets[] = {0, sizeof(int)};  
 const *MPI\_Datatype* fields\_types[] = {**MPI\_INT**, **MPI\_INT**};  
  
 *MPI\_Datatype* ret;  
  
 MPI\_Type\_create\_struct(2,  
 block\_lengths,  
 offsets,  
 fields\_types,  
 &ret);  
  
 MPI\_Type\_commit(&ret);  
  
 return ret;  
}

**task.c**

#include <math.h>#include <stdlib.h>#include "task.h"#include "config.h"*task\_t* create\_task(int *iter\_counter*, int *task\_index*, int *rank*, int *size*) { const *task\_t* ret = { .repeat\_number = abs(**TASKS\_IN\_LIST\_PER\_PROCESS** / 2 - *task\_index* % **TASKS\_IN\_LIST\_PER\_PROCESS**) \* abs(*rank* - (*iter\_counter* % *size*)) \* **L** }; return ret;}double execute\_task(const *task\_t*\* *task*) { double ret = 0; for(long i = 1; i <= *task*->repeat\_number; i++) { ret += sin((double)i); } return ret;}*MPI\_Datatype* create\_task\_data\_type() { const int block\_lengths[] = {1}; const *MPI\_Aint* offsets[] = {0}; const *MPI\_Datatype* fields\_types[] = {**MPI\_UNSIGNED\_LONG**}; *MPI\_Datatype* ret; MPI\_Type\_create\_struct(1, block\_lengths, offsets, fields\_types, &ret); MPI\_Type\_commit(&ret); return ret;}

**tasks\_list.c**

#include <malloc.h>#include "tasks\_list.h"#include "config.h"void init\_tasks\_list(*tasks\_list\_t*\* *tasks\_list*) { *tasks\_list*->current\_task\_index = 0; *tasks\_list*->new\_task\_index = 0; *tasks\_list*->capacity = **TASKS\_IN\_LIST\_PER\_PROCESS**; *tasks\_list*->tasks = calloc(*tasks\_list*->capacity, sizeof(*task\_t*));}void add\_task(*tasks\_list\_t*\* *tasks\_list*, *task\_t task*) { if (*tasks\_list*->new\_task\_index == *tasks\_list*->capacity) { *tasks\_list*->capacity \*= 2; *tasks\_list*->tasks = reallocarray(*tasks\_list*->tasks, *tasks\_list*->capacity, sizeof(*task*)); } *tasks\_list*->tasks[*tasks\_list*->new\_task\_index++] = *task*;} **bool** reached\_end(const *tasks\_list\_t*\* *tasks\_list*) { return get\_remaining\_tasks\_count(*tasks\_list*) == 0;}*size\_t* get\_remaining\_tasks\_count(const *tasks\_list\_t*\* *tasks\_list*) { return *tasks\_list*->new\_task\_index - *tasks\_list*->current\_task\_index;}*task\_t* get\_task(*tasks\_list\_t*\* *tasks\_list*) { return *tasks\_list*->tasks[*tasks\_list*->current\_task\_index++];}void free\_tasks\_list(*tasks\_list\_t*\* *tasks\_list*) { free(*tasks\_list*->tasks);}

**tasks\_response.c**

#include "tasks\_response.h"*MPI\_Datatype* create\_tasks\_response\_data\_type(*MPI\_Datatype task\_data\_type*) { const int block\_lengths[] = {1, 100}; const *MPI\_Aint* offsets[] = {0, sizeof(long)}; const *MPI\_Datatype* fields\_types[] = {**MPI\_INT**, *task\_data\_type*}; *MPI\_Datatype* ret; MPI\_Type\_create\_struct(2, block\_lengths, offsets, fields\_types, &ret); MPI\_Type\_commit(&ret); return ret;}

**worker.c**

#include <mpi.h>#include <pthread.h>#include <stdio.h>#include <malloc.h>#include <math.h>#include <unistd.h>#include "manager.h"#include "worker.h"#include "task.h"#include "tasks\_list.h"#include "config.h"#include "message.h"#include "tasks\_response.h"#define **OUTPUT\_BUFFER\_SIZE** 1024#define **DEFAULT\_LISTENER\_RESPONSE\_TAG** 1#define **DEFAULT\_NOTIFY\_TAG** 2#define **DEFAULT\_RANK\_REQUEST\_TAG** 3#define **DEFAULT\_TASK\_REQUEST\_TAG** 4#define **SUCCESS\_CODE** 0#define **MAX**(A, B) ((A) > (B) ? (A) : (B))#define **MIN**(A, B) ((A) > (B) ? (B) : (A))typedef struct { double min\_imbalance\_part; double max\_imbalance\_part; double average\_imbalance\_part;} *final\_stats\_t*;double seconds\_to\_millis(double *seconds*) { return *seconds* \* 1000;}void print\_iteration\_stats(*MPI\_File out*, char\* *buffer*, *size\_t tasks\_executed*, double *global\_result*, int *iter\_counter*, double *current\_iteration\_time\_ms*, int *size*, int *rank*) { const int length1 = *rank* == 0 ? sprintf(*buffer*, "ITERATION %d\n\n", *iter\_counter*) : 0; const int length2 = sprintf(*buffer* + length1, "Process %d\n%-40s%zu\n%-40s%.0lf ms\n%-40s%.2lf\n\n", *rank*, "Tasks executed", *tasks\_executed*, "Time for iteration", *current\_iteration\_time\_ms*, "Global result", *global\_result*); const int length3 = *rank* == *size* - 1 ? sprintf(*buffer* + length1 + length2, "-----------------------------------------------\n\n") : 0; MPI\_File\_write\_shared(*out*, *buffer*, length1 + length2 + length3, **MPI\_CHAR**, **MPI\_STATUS\_IGNORE**);}*final\_stats\_t* calculate\_final\_stats(const double\* *total\_iterations\_time\_ms*, int *size*) { double min\_imbalance\_part = 1; double average\_imbalance\_part = 0; double max\_imbalance\_part = 0; for (int i = 0; i < **LISTS\_NUMBER**; i++) { double max\_time\_ms = *total\_iterations\_time\_ms*[i \* *size* + *size* - 1]; double imbalance\_time\_ms = 0; for (int j = 0; j < *size* - 1; j++) { const double t1 = *total\_iterations\_time\_ms*[i \* *size* + j]; max\_time\_ms = **MAX**(max\_time\_ms, t1); for (int k = j + 1; k < *size*; k++) { const double t2 = *total\_iterations\_time\_ms*[i \* *size* + k]; const double current\_delta = fabs(t2 - t1); imbalance\_time\_ms = **MAX**(imbalance\_time\_ms, current\_delta); } } const double current\_imbalance\_part = imbalance\_time\_ms / max\_time\_ms; min\_imbalance\_part = **MIN**(min\_imbalance\_part, current\_imbalance\_part); average\_imbalance\_part += current\_imbalance\_part; max\_imbalance\_part = **MAX**(max\_imbalance\_part, current\_imbalance\_part); } average\_imbalance\_part /= **LISTS\_NUMBER**; *final\_stats\_t* ret = { .min\_imbalance\_part = min\_imbalance\_part, .max\_imbalance\_part = max\_imbalance\_part, .average\_imbalance\_part = average\_imbalance\_part }; return ret;}*pthread\_mutex\_t* tasks\_list\_mutex;*pthread\_mutex\_t* reached\_critical\_tasks\_count\_mutex;*pthread\_mutex\_t* started\_new\_list\_mutex;*pthread\_cond\_t* reached\_critical\_tasks\_count\_cond;*pthread\_cond\_t* started\_new\_list\_cond;*tasks\_list\_t* tasks\_list;*MPI\_Datatype* task\_data\_type;*MPI\_Datatype* message\_date\_type;*MPI\_Datatype* tasks\_response\_data\_type;*MPI\_Comm* workers\_comm;int manager\_rank; **bool** finished\_all\_tasks\_flag = **false**; **bool** reached\_critical\_tasks\_count\_flag = **false**; **bool** started\_new\_list\_flag = **false**;void\* requester\_routine(\_\_attribute\_\_((unused)) void\* *arg*) { int rank; MPI\_Comm\_rank(workers\_comm, &rank); while (!finished\_all\_tasks\_flag) { pthread\_mutex\_lock(&reached\_critical\_tasks\_count\_mutex); while (!reached\_critical\_tasks\_count\_flag) { pthread\_cond\_wait(&reached\_critical\_tasks\_count\_cond, &reached\_critical\_tasks\_count\_mutex); } pthread\_mutex\_unlock(&reached\_critical\_tasks\_count\_mutex); const *message\_t* request = { .type = **PROCESS\_RANK\_REQUEST\_TYPE**, .value = 0 }; MPI\_Send(&request, 1, message\_date\_type, manager\_rank, **DEFAULT\_RANK\_REQUEST\_TAG**, **MPI\_COMM\_WORLD**); int rank\_of\_process\_with\_available\_tasks; MPI\_Recv(&rank\_of\_process\_with\_available\_tasks, 1, **MPI\_INT**, manager\_rank, **DEFAULT\_MANAGER\_RESPONSE\_TAG**, **MPI\_COMM\_WORLD**, **MPI\_STATUS\_IGNORE**); if (rank\_of\_process\_with\_available\_tasks == rank) { pthread\_mutex\_lock(&started\_new\_list\_mutex); while (!started\_new\_list\_flag) { pthread\_cond\_wait(&started\_new\_list\_cond, &started\_new\_list\_mutex); } pthread\_mutex\_unlock(&started\_new\_list\_mutex); continue; } int fictive = 0; MPI\_Send(&fictive, 1, **MPI\_INT**, rank\_of\_process\_with\_available\_tasks, **DEFAULT\_TASK\_REQUEST\_TAG**, workers\_comm); *tasks\_response\_t* tasks\_response; MPI\_Recv(&tasks\_response, 1, tasks\_response\_data\_type, rank\_of\_process\_with\_available\_tasks, **DEFAULT\_LISTENER\_RESPONSE\_TAG**, workers\_comm, **MPI\_STATUS\_IGNORE**); if (tasks\_response.tasks\_count) { pthread\_mutex\_lock(&tasks\_list\_mutex); for (long i = 0; i < tasks\_response.tasks\_count; i++) { add\_task(&tasks\_list, tasks\_response.tasks[i]); } pthread\_mutex\_unlock(&tasks\_list\_mutex); } else { pthread\_mutex\_lock(&started\_new\_list\_mutex); while (!started\_new\_list\_flag) { pthread\_cond\_wait(&started\_new\_list\_cond, &started\_new\_list\_mutex); } pthread\_mutex\_unlock(&started\_new\_list\_mutex); } } return **SUCCESS\_CODE**;}void\* listener\_routine(\_\_attribute\_\_((unused)) void\* *arg*) { int rank; MPI\_Comm\_rank(workers\_comm, &rank); int received = 1; while (!finished\_all\_tasks\_flag) { *MPI\_Request* request; if (received) { int fictive; MPI\_Irecv(&fictive, 1, **MPI\_INT**, **MPI\_ANY\_SOURCE**, **DEFAULT\_TASK\_REQUEST\_TAG**, workers\_comm, &request); received = 0; } *MPI\_Status* status; MPI\_Test(&request, &received, &status); if (received) { pthread\_mutex\_lock(&tasks\_list\_mutex); *tasks\_response\_t* tasks\_response = {.tasks\_count = 0}; for (int i = 0; i < **TASKS\_RESPONSE\_LENGTH**; i++) { if (get\_remaining\_tasks\_count(&tasks\_list) == 0) { break; } tasks\_response.tasks[tasks\_response.tasks\_count++] = get\_task(&tasks\_list); } pthread\_mutex\_unlock(&tasks\_list\_mutex); MPI\_Send(&tasks\_response, 1, tasks\_response\_data\_type, status.MPI\_SOURCE, **DEFAULT\_LISTENER\_RESPONSE\_TAG**, workers\_comm); } } return **SUCCESS\_CODE**;}void work(*MPI\_Comm \_workers\_comm*, int *\_manager\_rank*) { manager\_rank = *\_manager\_rank*; workers\_comm = *\_workers\_comm*; int rank, size; MPI\_Comm\_rank(workers\_comm, &rank); MPI\_Comm\_size(workers\_comm, &size); const int ROOT\_RANK = 0; task\_data\_type = create\_task\_data\_type(); message\_date\_type = create\_message\_data\_type(); tasks\_response\_data\_type = create\_tasks\_response\_data\_type(task\_data\_type); pthread\_mutex\_init(&tasks\_list\_mutex, **NULL**); pthread\_mutex\_init(&reached\_critical\_tasks\_count\_mutex, **NULL**); pthread\_mutex\_init(&started\_new\_list\_mutex, **NULL**); pthread\_cond\_init(&reached\_critical\_tasks\_count\_cond, **NULL**); pthread\_cond\_init(&started\_new\_list\_cond, **NULL**); *pthread\_attr\_t* attrs; pthread\_attr\_init(&attrs); *pthread\_t* requester, listener; pthread\_create(&requester, &attrs, requester\_routine, **NULL**); pthread\_create(&listener, &attrs, listener\_routine, **NULL**); remove(**OUTPUT\_FILE\_NAME**); *MPI\_File* out; MPI\_File\_open(workers\_comm, **OUTPUT\_FILE\_NAME**, **MPI\_MODE\_WRONLY** | **MPI\_MODE\_CREATE**, **MPI\_INFO\_NULL**, &out); char\* buffer = calloc(**OUTPUT\_BUFFER\_SIZE**, sizeof(char)); double\* iterations\_time\_ms = calloc(**LISTS\_NUMBER**, sizeof(double)); double\* total\_iterations\_time\_ms = rank == ROOT\_RANK ? calloc((unsigned long) (**LISTS\_NUMBER** \* size), sizeof(double)) : **NULL**; double global\_result = 0; for (int iter\_counter = 0; iter\_counter < **LISTS\_NUMBER**; iter\_counter++) { pthread\_mutex\_lock(&tasks\_list\_mutex); init\_tasks\_list(&tasks\_list); for (int task\_index = 0; task\_index < **TASKS\_IN\_LIST\_PER\_PROCESS**; task\_index++) { const *task\_t* new\_task = create\_task(iter\_counter, task\_index, rank, size); add\_task(&tasks\_list, new\_task); } const double start\_time\_s = MPI\_Wtime(); *size\_t* tasks\_executed = 0; started\_new\_list\_flag = **true**; pthread\_cond\_signal(&started\_new\_list\_cond); while (!reached\_end(&tasks\_list)) { const *task\_t* current\_task = get\_task(&tasks\_list); pthread\_mutex\_unlock(&tasks\_list\_mutex); global\_result += execute\_task(&current\_task); pthread\_mutex\_lock(&tasks\_list\_mutex); tasks\_executed++; const *message\_t* message = { .type = **TASKS\_UPDATE\_TYPE**, .value = (int)get\_remaining\_tasks\_count(&tasks\_list) }; MPI\_Send(&message, 1, message\_date\_type, manager\_rank, **DEFAULT\_NOTIFY\_TAG**, **MPI\_COMM\_WORLD**); reached\_critical\_tasks\_count\_flag = get\_remaining\_tasks\_count(&tasks\_list) < **CRITICAL\_TASKS\_COUNT**; if (reached\_critical\_tasks\_count\_flag) { pthread\_cond\_signal(&reached\_critical\_tasks\_count\_cond); } } pthread\_mutex\_unlock(&tasks\_list\_mutex); const double end\_time\_s = MPI\_Wtime(); const double current\_iteration\_time\_ms = seconds\_to\_millis(end\_time\_s - start\_time\_s); iterations\_time\_ms[iter\_counter] = current\_iteration\_time\_ms; for (int i = 0; i < size; i++) { if (i == rank) { print\_iteration\_stats(out, buffer, tasks\_executed, global\_result, iter\_counter, current\_iteration\_time\_ms, size, rank); } MPI\_Barrier(workers\_comm); } pthread\_mutex\_lock(&tasks\_list\_mutex); free\_tasks\_list(&tasks\_list); pthread\_mutex\_unlock(&tasks\_list\_mutex); } finished\_all\_tasks\_flag = **true**; *MPI\_Datatype* recv\_type; *MPI\_Datatype* resized\_recv\_type; MPI\_Type\_vector(**LISTS\_NUMBER**, 1, size, **MPI\_DOUBLE**, &recv\_type); MPI\_Type\_create\_resized(recv\_type, 0, sizeof(double), &resized\_recv\_type); MPI\_Type\_commit(&resized\_recv\_type); MPI\_Gather(iterations\_time\_ms, **LISTS\_NUMBER**, **MPI\_DOUBLE**, total\_iterations\_time\_ms, 1, resized\_recv\_type, ROOT\_RANK, workers\_comm); if (rank == ROOT\_RANK) { const *message\_t* termination\_message = { .type = **TERMINATION\_TYPE**, .value = 0 }; MPI\_Send(&termination\_message, 1, message\_date\_type, manager\_rank, 0, **MPI\_COMM\_WORLD**); const *final\_stats\_t* final\_stats = calculate\_final\_stats(total\_iterations\_time\_ms, size); char\* tmp = buffer; tmp += sprintf(tmp, "%-40s%.0lf %%\n%-40s%.0lf %%\n%-40s%.0lf %%\n\n", "Min imbalance percentage", final\_stats.min\_imbalance\_part \* 100, "Max imbalance percentage", final\_stats.max\_imbalance\_part \* 100, "Average imbalance percentage", final\_stats.average\_imbalance\_part \* 100); MPI\_File\_write\_shared(out, buffer, (int)(tmp - buffer), **MPI\_CHAR**, **MPI\_STATUS\_IGNORE**); } pthread\_join(requester, **NULL**); pthread\_join(listener, **NULL**); MPI\_File\_close(&out); pthread\_attr\_destroy(&attrs); pthread\_mutex\_destroy(&tasks\_list\_mutex); pthread\_mutex\_destroy(&reached\_critical\_tasks\_count\_mutex); pthread\_mutex\_destroy(&started\_new\_list\_mutex); pthread\_cond\_destroy(&reached\_critical\_tasks\_count\_cond); pthread\_cond\_destroy(&started\_new\_list\_cond); free(iterations\_time\_ms); free(total\_iterations\_time\_ms); free(buffer); MPI\_Type\_free(&task\_data\_type); MPI\_Type\_free(&message\_date\_type); MPI\_Type\_free(&tasks\_response\_data\_type); MPI\_Type\_free(&recv\_type); MPI\_Type\_free(&resized\_recv\_type);}

# **Приложение 3.** Компиляция и сборка

**CmakeLists.txt**

cmake\_minimum\_required(VERSION 3.22)project(POSIX C)set(CMAKE\_C\_STANDARD 11)find\_package(MPI)include\_directories(${MPI\_INCLUDE\_PATH})SET(CMAKE\_C\_COMPILER mpicc)add\_compile\_options( -Werror -Wall -Wextra -pedantic -Wduplicated-branches -Wduplicated-cond -Wconversion -Wsign-conversion -Wlogical-op -pthread -O0 )add\_executable(POSIX src/main.c src/task.c src/tasks\_list.c src/manager.c src/message.c src/worker.c src/tasks\_response.c)target\_link\_libraries(POSIX PRIVATE m)