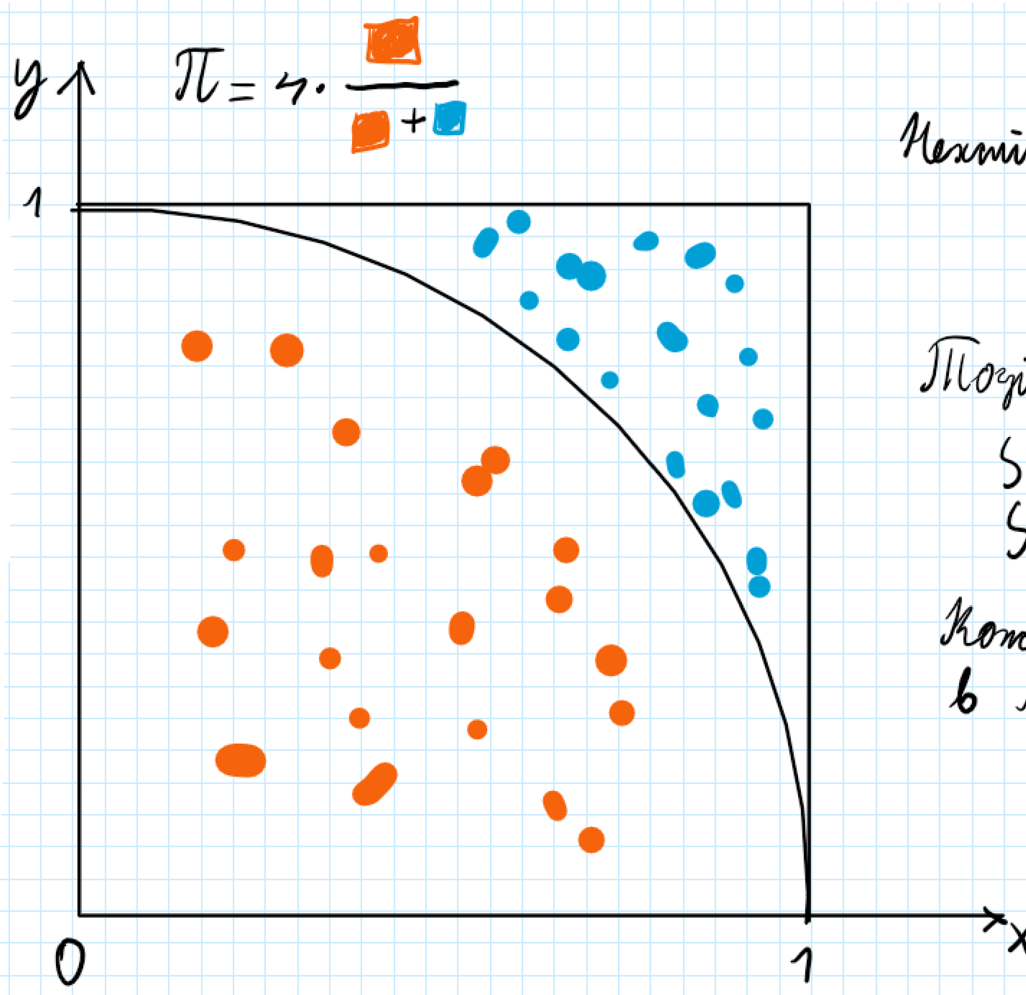
Теоретичні відомості



Нехай O – частина одиничного кола з центром у (0;0), що лежить у першій чверті координатної площини

Q – квадрат з вершинами (0;0) (0,1) (1;0) (1;1)

Тоді:

$S(O) = \frac{\pi R^2}{4}$ $(R=1)$ $\frac{S(O)}{S(Q)} = \frac{\pi R^2}{4R^2} = \frac{\pi}{4} \Rightarrow \pi = 4 \cdot \frac{S(O)}{S(Q)} \approx 4 \cdot \frac{count(O)}{count(Q)}$

$S(Q) = R^2$

Кожен вектор, координати якого генеруються випадковим чином в межах [0; 1], знаходитиметься в Q.

Серед цих векторів знайдуться такі, що лежатимуть в O.

Якщо загальна кількість векторів прямує до $+\infty$, то відношення $\frac{count(O)}{count(Q)} \longrightarrow \frac{S(O)}{S(Q)}$, де count(O) – кількість векторів у O

count(Q) – кількість векторів у Q

Про генерацію випадкових чисел

Генерація вектору з випадковими координатами в межах [0; 1] реалізована наступним чином (файл v2.c):

```
void v2_init(v2 * vector)
{
        vector->xcdr = (long double)(rand()) / (long double)RAND_MAX;
        vector->ycdr = (long double)(rand()) / (long double)RAND_MAX;
}
```

Наявне використання функції rand() стандартної бібліотеки C (stdlib.h). Згідно документації, стандарт C не гарантує потокобезпечність функції rand: "`rand()` is not guaranteed to be thread-safe. ".
Реалізація використовує компілятор MSVC та CRT реалізацію стандартної бібліотеки C.

Версія CRT, використана в роботі, реалізує функції rand() та srand(seed) наступним чином:

```
//
// rand.cpp
//
//      Copyright (c) Microsoft Corporation. All rights reserved.
//
// Defines rand(), which generates psuedorandom numbers.
//
#include <corecrt_internal.h>
#include <stdlib.h>



// Seeds the random number generator with the provided integer.
extern "C" void __cdecl srand(unsigned int const seed)
{
    __acrt_getptd()->_rand_state = seed;
}



// Returns a pseudorandom number in the range [0,32767].
extern "C" int __cdecl rand()
{
    __acrt_ptd* const ptd = __acrt_getptd();

    ptd->_rand_state = ptd->_rand_state * 214013 + 2531011;
    return (ptd->_rand_state >> 16) & RAND_MAX;
}
```

Бачимо як rand() генерує число:
1. виклик __acrt_getptd() повертає вказівник на деяку структуру типу __acrt_ptd.
2. із структури, на яку вказує ptd, береться змінна _rand_state
3. виконання арифметичних та бітових операцій над _rand_state, збереження зміненого значення у структурі, повернення результату.

Реалізація srand(seed) встановлює значення _rand_state структури __acrt_ptd рівним seed.

Таким чином потокобезпечність функцій rand() та srand(seed) залежить від того, чи має кожен потік свою структуру типу __acrt_ptd.

thread.cpp

```
template <typename ThreadProcedure>
static unsigned long WINAPI thread_start(void* const parameter) throw()
{
    if (!parameter)
    {
        ExitThread(GetLastError());
    }

    __acrt_thread_parameter* const context = static_cast<__acrt_thread_parameter*>(parameter);

    __acrt_getptd()->_beginthread_context = context;

    if (__acrt_get_begin_thread_init_policy() == begin_thread_init_policy_ro_initialize)
    {
        context->_initialized_apartment = __acrt_RoInitialize(RO_INIT_MULTITHREADED) == S_OK;
    }

    __try
    {
        ThreadProcedure const procedure = reinterpret_cast<ThreadProcedure>(context->_procedure);

        _endthreadex(invoke_thread_procedure(procedure, context->_context));
    }
    __except (_seh_filter_exe(GetExceptionCode(), GetExceptionInformation()))
    {
        // Execution should never reach here:
        _exit(GetExceptionCode());
    }

    // This return statement will never be reached.  All execution paths result
    // in the thread or process exiting.
    return 0;
}
```

corecrt_internal.h

```
//-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//
// AppCRT Per-Thread Data
//
//-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
typedef struct __acrt_ptd
{
    // These three data members support signal handling and runtime errors
    struct __crt_signal_action_t* _pxcptacttab;     // Pointer to the exception-action table
    EXCEPTION_POINTERS*           _tpxcptinfoptrs;  // Pointer to the exception info pointers
    int                           _tfpecode;        // Last floating point exception code

    terminate_handler  _terminate;  // terminate() routine

    int                _terrno;          // errno value
    unsigned long      _tdoserrno;       // _doserrno value

    unsigned int       _rand_state;      // Previous value of rand()

    // Per-thread strtok(), wcstok(), and mbstok() data:
    char*              _strtok_token;
    unsigned char*     _mbstok_token;
    wchar_t*           _wcstok_token;

    // Per-thread tmpnam() data:
    char*              _tmpnam_narrow_buffer;
    wchar_t*           _tmpnam_wide_buffer;

    // Per-thread time library data:
    char*              _asctime_buffer;  // Pointer to asctime() buffer
    wchar_t*           _wasctime_buffer; // Pointer to _wasctime() buffer
    struct tm*         _gmtime_buffer;   // Pointer to gmtime() structure

    char*              _cvtbuf;          // Pointer to the buffer used by ecvt() and fcvt().

    // Per-thread error message data:
    char*      _strerror_buffer;         // Pointer to strerror()  / _strerror()  buffer
    wchar_t*   _wcserror_buffer;         // Pointer to _wcserror() / __wcserror() buffer

    // Locale data:
    __crt_multibyte_data*                  _multibyte_info;
    __crt_locale_data*                     _locale_info;
    __crt_qualified_locale_data            _setloc_data;
    __crt_qualified_locale_data_downlevel* _setloc_downlevel_data;
    int                                    _own_locale;   // If 1, this thread owns its locale

    // The buffer used by _putch(), and the flag indicating whether the buffer
    // is currently in use or not.
    unsigned char  _putch_buffer[MB_LEN_MAX];
    unsigned short _putch_buffer_used;

    // The thread-local invalid parameter handler
    _invalid_parameter_handler _thread_local_iph;

    // If this thread was started by the CRT (_beginthread or _beginthreadex),
    // then this points to the context with which the thread was created.  If
    // this thread was not started by the CRT, this pointer is null.
    __acrt_thread_parameter* _beginthread_context;

} __acrt_ptd;
```

per_thread_data.cpp

```
extern "C" __acrt_ptd* __cdecl __acrt_getptd()
{
    __acrt_ptd* const ptd = internal_getptd_noexit();
    if (!ptd)
    {
        abort();
    }

    return ptd;
}
```

per_thread_data.cpp

```
// This functionality has been split out of __acrt_getptd_noexit so that we can
// force it to be inlined into both __acrt_getptd_noexit and __acrt_getptd.  These
// functions are performance critical and this change has substantially improved
// __acrt_getptd performance.
static __forceinline __acrt_ptd* __cdecl internal_getptd_noexit() throw()
{
    __crt_scoped_get_last_error_reset const last_error_reset;

    __acrt_ptd* const ptd_head = internal_get_ptd_head();
    if (!ptd_head)
    {
        return nullptr;
    }

    return ptd_head + __crt_state_management::get_current_state_index();
}
```

per_thread_data.cpp

```
_Success_(return != nullptr)
static __forceinline __acrt_ptd* internal_get_ptd_head() throw()
{
    // We use the CRT heap to allocate the PTD.  If the CRT heap fails to
    // allocate the requested memory, it will attempt to set errno to ENOMEM,
    // which will in turn attempt to acquire the PTD, resulting in infinite
    // recursion that causes a stack overflow.
    //
    // We set the PTD to this sentinel value for the duration of the allocation
    // in order to detect this case.
    static void* const reentrancy_sentinel = reinterpret_cast<void*>(SIZE_MAX);

    __acrt_ptd* const existing_ptd_head = try_get_ptd_head();
    if (existing_ptd_head == reentrancy_sentinel)
    {
        return nullptr;
    }
    else if (existing_ptd_head != nullptr)
    {
        return existing_ptd_head;
    }

    if (!__acrt_FlsSetValue(__acrt_flsindex, reentrancy_sentinel))
    {
        return nullptr;
    }

    __crt_unique_heap_ptr<__acrt_ptd> new_ptd_head(_calloc_crt_t(__acrt_ptd,
__crt_state_management::state_index_count));
    if (!new_ptd_head)
    {
        __acrt_FlsSetValue(__acrt_flsindex, nullptr);
        return nullptr;
    }

    if (!__acrt_FlsSetValue(__acrt_flsindex, new_ptd_head.get()))
    {
        __acrt_FlsSetValue(__acrt_flsindex, nullptr);
        return nullptr;
    }

    construct_ptd_array(new_ptd_head.get());
    return new_ptd_head.detach();
}
```

per_thread_data.cpp

```
// Constructs each of the 'state_index_count' PTD objects in the array of PTD
// objects pointed to by 'ptd'.
static void __cdecl construct_ptd_array(__acrt_ptd* const ptd) throw()
{
    for (size_t i = 0; i != __crt_state_management::state_index_count; ++i)
    {
        construct_ptd(&ptd[i], &__acrt_current_locale_data.dangerous_get_state_array()[i]);
    }
}
```

per_thread_data.cpp

```
// Constructs a single PTD object, copying the given 'locale_data' if provided.
static void __cdecl construct_ptd(
    __acrt_ptd*         const ptd,
    __crt_locale_data** const locale_data
    ) throw()
{
    ptd->_rand_state  = 1;
    ptd->_pxcptacttab = const_cast<__crt_signal_action_t*>(__acrt_exception_action_table);

    // It is necessary to always have GLOBAL_LOCALE_BIT set in perthread data
    // because when doing bitwise or, we won't get __UPDATE_LOCALE to work when
    // global per thread locale is set.
    ptd->_own_locale = _GLOBAL_LOCALE_BIT;

    ptd->_multibyte_info = &__acrt_initial_multibyte_data;

    // Initialize _setloc_data. These are the only valuse that need to be
    // initialized.
    ptd->_setloc_data._cachein[0]  = L'C';
    ptd->_setloc_data._cacheout[0] = L'C';

    // Downlevel data is not initially used
    ptd->_setloc_downlevel_data = nullptr;

    __acrt_lock_and_call(__acrt_multibyte_cp_lock, [&]
    {
        _InterlockedIncrement(&ptd->_multibyte_info->refcount);
    });

    // We need to make sure that ptd->ptlocinfo in never nullptr, this saves us
    // perf counts when UPDATING locale.
    __acrt_lock_and_call(__acrt_locale_lock, [&]
    {
        replace_current_thread_locale_nolock(ptd, *locale_data);
    });
}
```

МАЄМО

corecrt_internal.h
struct __acrt_ptd

thread.cpp
thread_start

→ **per_thread_data.cpp**
__acrt_getptd()

↓ **per_thread_data.cpp**
internal_getptd_noexit()

↓ **per_thread_data.cpp**
internal_get_ptd_head()

↓ **per_thread_data.cpp**
construct_ptd_array()

↓ **per_thread_data.cpp**
construct_ptd()

Висновки (генерація випадкових чисел)

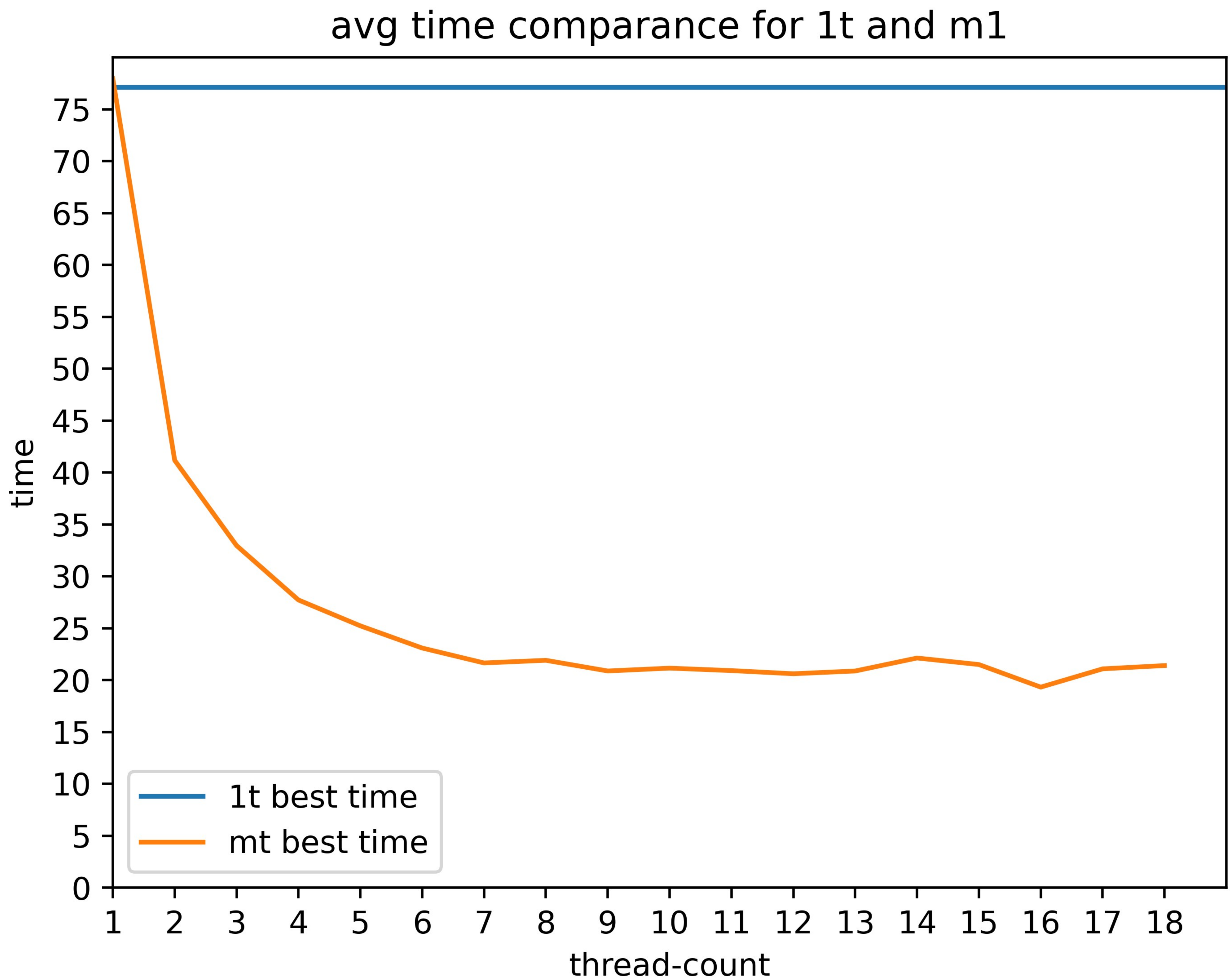
Таким чином, кожен потік має власну структуру типу __acrt_ptd, що створюється та ініціалізується під час створення потоку.
Отже функції rand() та srand(seed) є **потокобезпечними** у реалізації стандартної бібліотеки C, що була використана у роботі.

Функція srand(seed) встановлює seed для структури поточного потоку (див. rand.cpp). Отже srand(seed) має бути викликана для кожного потоку, щоб проініціалізувати зерно для кожного потоку. Інакше всі потоки матимуть однаковий seed, рівний 1 (див. construct_ptd).

Дослідження швидкодії

Загальна кількість векторів дорівнює 2147483640 для всіх багатопоточних та однопоточного тестувань.
Кількість потоків для багатопоточної реалізації змінювалась у межах [1;18].

```
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo s 2147483640
Run 0 | pi = 3.141522421097466; time = 77.097151
Run 1 | pi = 3.141522424822757; time = 77.380007
Run 2 | pi = 3.141522421097466; time = 77.515874
Run 3 | pi = 3.141522421097466; time = 95.446748
Run 4 | pi = 3.141522419234821; time = 77.416460
Min time = 77.097151
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 1
Run 0 | pi = 3.141522421097466; time = 78.033975
Run 1 | pi = 3.141522419234821; time = 78.248226
Run 2 | pi = 3.141522758236240; time = 77.940108
Run 3 | pi = 3.141522760098885; time = 78.511687
Run 4 | pi = 3.141522760098885; time = 77.975751
Min time = 77.940108
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 2
Run 0 | pi = 3.141522421097466; time = 41.165316
Run 1 | pi = 3.141522754510949; time = 44.884671
Run 2 | pi = 3.141522758236240; time = 44.539542
Run 3 | pi = 3.141522424822757; time = 44.434374
Run 4 | pi = 3.141522424822757; time = 44.763309
Min time = 41.165316
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 3
Run 0 | pi = 3.141509948825500; time = 32.952201
Run 1 | pi = 3.141517146086384; time = 35.233360
Run 2 | pi = 3.141515639206453; time = 35.314248
Run 3 | pi = 3.141512675738009; time = 35.320071
Run 4 | pi = 3.141518589636380; time = 35.099387
Min time = 32.952201
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 4
Run 0 | pi = 3.141516885316062; time = 27.712195
Run 1 | pi = 3.141536677783492; time = 30.654514
Run 2 | pi = 3.141519636442958; time = 31.007162
Run 3 | pi = 3.141536631217363; time = 30.793014
Run 4 | pi = 3.141517079031158; time = 30.839769
Min time = 27.712195
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 5
Run 0 | pi = 3.141485522096923; time = 25.220196
Run 1 | pi = 3.141535331091044; time = 28.231851
Run 2 | pi = 3.141525334274491; time = 28.452459
Run 3 | pi = 3.141557086786468; time = 28.224950
Run 4 | pi = 3.141517963787608; time = 28.341522
Min time = 25.220196
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 6
Run 0 | pi = 3.141552543794932; time = 23.081363
Run 1 | pi = 3.141575968420416; time = 26.397793
Run 2 | pi = 3.141532846322406; time = 25.933661
Run 3 | pi = 3.141494036247932; time = 26.261213
Run 4 | pi = 3.141525108894427; time = 26.283448
Min time = 23.081363
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 7
Run 0 | pi = 3.141486283918792; time = 21.643927
Run 1 | pi = 3.141481373986160; time = 24.501711
Run 2 | pi = 3.141522512367079; time = 24.875150
Run 3 | pi = 3.141525442307910; time = 24.588225
Run 4 | pi = 3.141496131723732; time = 24.436871
Min time = 21.643927
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 8
Run 0 | pi = 3.141489597564525; time = 21.897090
Run 1 | pi = 3.141527409261195; time = 23.585267
Run 2 | pi = 3.141520323759021; time = 23.874334
Run 3 | pi = 3.141550470670873; time = 23.722709
Run 4 | pi = 3.141560299849362; time = 23.763223
Min time = 21.897090
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 9
Run 0 | pi = 3.141533822348467; time = 20.870063
Run 1 | pi = 3.141503232127068; time = 24.058846
Run 2 | pi = 3.141565410947671; time = 23.970969
Run 3 | pi = 3.141479030778553; time = 24.008459
Run 4 | pi = 3.141494697486962; time = 24.265237
Min time = 20.870063
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 10
Run 0 | pi = 3.141545752590693; time = 21.142825
Run 1 | pi = 3.141528120791644; time = 24.422836
Run 2 | pi = 3.141498106127598; time = 23.564162
Run 3 | pi = 3.141516674837160; time = 23.979508
Run 4 | pi = 3.141516149571226; time = 24.129900
Min time = 21.142825
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 11
Run 0 | pi = 3.141529862364866; time = 20.906328
Run 1 | pi = 3.141480500405581; time = 24.017303
Run 2 | pi = 3.141527759438484; time = 23.701433
Run 3 | pi = 3.141491350313616; time = 23.966143
Run 4 | pi = 3.141489057397429; time = 23.656969
Min time = 20.906328
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 12
Run 0 | pi = 3.141502203946942; time = 20.602543
Run 1 | pi = 3.141482094829835; time = 24.534069
Run 2 | pi = 3.141504895469192; time = 24.531052
Run 3 | pi = 3.141552040880740; time = 24.161348
Run 4 | pi = 3.141566366484636; time = 23.591431
Min time = 20.602543
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 13
Run 0 | pi = 3.141531030243378; time = 20.866737
Run 1 | pi = 3.141518885796960; time = 23.800841
Run 2 | pi = 3.141553378259962; time = 23.772852
Run 3 | pi = 3.141516445731805; time = 23.890665
Run 4 | pi = 3.141537273829942; time = 23.558448
Min time = 20.866737
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 14
Run 0 | pi = 3.141493332168062; time = 22.118333
Run 1 | pi = 3.141505614450222; time = 23.907900
Run 2 | pi = 3.141537612831360; time = 23.564107
Run 3 | pi = 3.141526680966939; time = 23.498432
Run 4 | pi = 3.141523203408432; time = 24.053162
Min time = 22.118333
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 15
Run 0 | pi = 3.141499508699400; time = 21.489275
Run 1 | pi = 3.141533801859371; time = 23.915583
Run 2 | pi = 3.141493233447869; time = 23.885569
Run 3 | pi = 3.141514976104777; time = 23.457160
Run 4 | pi = 3.141551813638031; time = 23.782775
Min time = 21.489275
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 16
Run 0 | pi = 3.141569871982820; time = 19.312971
Run 1 | pi = 3.141548997318555; time = 23.988785
Run 2 | pi = 3.141511960482269; time = 24.186892
Run 3 | pi = 3.141528288429708; time = 23.850696
Run 4 | pi = 3.141533863326661; time = 23.917776
Min time = 19.312971
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 17
Run 0 | pi = 3.141558882376398; time = 21.074850
Run 1 | pi = 3.141525656512103; time = 23.978431
Run 2 | pi = 3.141541595166704; time = 24.159926
Run 3 | pi = 3.141525570830426; time = 23.734220
Run 4 | pi = 3.141533855876080; time = 24.024386
Min time = 21.074850
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release> .\mcpi_demo m 2147483640 18
Run 0 | pi = 3.141534563681240; time = 21.391211
Run 1 | pi = 3.141473398139601; time = 23.652560
Run 2 | pi = 3.141572362339394; time = 23.921295
Run 3 | pi = 3.141562276115873; time = 23.473305
Run 4 | pi = 3.141508939271826; time = 23.888659
Min time = 21.391211
PS J:\repos\Parallel computing\MonteCarloPI\x64\Release>
```



Висновки

На відрізку n = [1; 7] бачимо приріст, приблизно кратний кількості потоків. Починаючи з n = 7, швидкодія практично не змінюється.
При цьому найкращий час було досягнуто при n = 16 (19.312971 секунд), що є неочікуваним.
Загалом результати схожі на дослідження, проведені у попередніх лабораторних роботах.

Реалізація (код)

Посилання на github репозиторій: https://github.com/Bohdan628318ylypchenko/parallel-programming-lab5.git

v2.h

```c
#pragma once

/// <summary>
/// Two dimensional vector definition
/// </summary>
typedef struct
{
    double xcdr;
    double ycdr;
} v2;

/// <summary>
/// Initializes coords of vector with random values in [0, 1].
/// </summary>
/// <param name="vector"> Pointer to vector struct to initialize coords of. </param>
void v2_init(v2 * vector);

/// <summary>
/// Calculates square of vector module.
/// </summary>
/// <param name="vector"> Pointer to vector struct. </param>
/// <returns> square of given vector module. </returns>
double v2_module2(v2 * vector);
```

v2.c

```c
#include "pch.h"

#include "v2.h"

#include <stdlib.h>

/// <summary>
/// Initializes coords of vector with random values in [0, 1].
/// </summary>
/// <param name="vector"> Pointer to vector struct to initialize coords of. </param>
void v2_init(v2 * vector)
{
    vector->xcdr = (long double)(rand()) / (long double)RAND_MAX;
    vector->ycdr = (long double)(rand()) / (long double)RAND_MAX;
}

/// <summary>
/// Calculates square of vector module.
/// </summary>
/// <param name="vector"> Pointer to vector struct. </param>
/// <returns> square of given vector module. </returns>
double v2_module2(v2 * vector)
{
    double xcdr = vector->xcdr;
    double ycdr = vector->ycdr;

    return xcdr * xcdr + ycdr * ycdr;
}
```

mcpi.h

```c
#pragma once

/// <summary>
/// Initializes seed for parent thread (e.g. thread - mcpi function caller).
/// </summary>
/// <param name="seed"> Sequence seed. </param>
void mcpi_init(int seed);

/// <summary>
/// Single thread implementation of
/// Monte Carlo PI calculation method.
/// </summary>
/// <param name="v_count"> Total vector count to generate. </param>
/// <returns> PI value. </returns>
double mcpi_1t(int v_count);

/// <summary>
/// Multi thread implementation of
/// Monte Carlo PI calculation method.
/// </summary>
/// <param name="v_count"> Total vector count to generate. </param>
/// <returns> PI value. </returns>
double mcpi_mt(int v_count);
```

mcpi.c

```c
#include "pch.h"

#include "v2.h"

#include <stdlib.h>

/// <summary>
/// Initializes coords of vector with random values in [0, 1].
/// </summary>
/// <param name="vector"> Pointer to vector struct to initialize coords of. </param>
void v2_init(v2 * vector)
{
    vector->xcdr = (long double)(rand()) / (long double)RAND_MAX;
    vector->ycdr = (long double)(rand()) / (long double)RAND_MAX;
}

/// <summary>
/// Calculates square of vector module.
/// </summary>
/// <param name="vector"> Pointer to vector struct. </param>
/// <returns> square of given vector module. </returns>
double v2_module2(v2 * vector)
{
    double xcdr = vector->xcdr;
    double ycdr = vector->ycdr;

    return xcdr * xcdr + ycdr * ycdr;
}
```

main.c

```c
#include "mcpi.h"

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <time.h>

#define USAGE "Usage: [s]ingle-thread v_count:int | [m]ulti-thread v_count:int thread_count:int"
#define RUN_COUNT 5

int main(int argc, char ** argv)
{
    if (argc < 3)
    {
        puts(USAGE);
        return EXIT_FAILURE;
    }

    mcpi_init(time(NULL));

    unsigned long v_count;
    double s_time, e_time, c_time, min_time = DBL_MAX;
    double pi;
    switch (argv[1][0])
    {
        case 's':

            v_count = atol(argv[2]);
            if (v_count <= 0)
            {
                printf("Invalid v_count: %s\n", argv[2]);
                return EXIT_FAILURE;
            }

            for (int i = 0; i < RUN_COUNT; i++)
            {
                s_time = omp_get_wtime();
                pi = mcpi_1t(v_count);
                e_time = omp_get_wtime();

                c_time = e_time - s_time;
                printf("Run %d | pi = %.15lf; time = %lf\n", i, pi, c_time);

                if (min_time > c_time)
                    min_time = c_time;
            }
            printf("Min time = %lf\n", min_time);

            break;

        case 'm':

            v_count = atol(argv[2]);
            if (v_count <= 0)
            {
                printf("Invalid v_count: %s\n", argv[2]);
                return EXIT_FAILURE;
            }

            int thread_count = atoi(argv[3]);
            if (thread_count <= 0)
            {
                printf("Invalid thread_count: %s\n", argv[3]);
                return EXIT_FAILURE;
            }

            omp_set_num_threads(thread_count);
            for (int i = 0; i < RUN_COUNT; i++)
            {
                s_time = omp_get_wtime();
                pi = mcpi_mt(v_count);
                e_time = omp_get_wtime();

                c_time = e_time - s_time;
                printf("Run %d | pi = %.15lf; time = %lf\n", i, pi, c_time);

                if (min_time > c_time)
                    min_time = c_time;
            }
            printf("Min time = %lf\n", min_time);

            break;

        default:

            puts(USAGE);
            return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```