Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

**КУРСОВАЯ РАБОТА**

**Аспекты параллельного программирования**

**Выполнил**:студент группы 0836-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Поляков И.О.

Подпись

**Научный руководитель**:

Старший преподаватель каф. МОСТ, к.т.н.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сысоев А.В.

Подпись

Нижний Новгород

2016

**Содержание**

[Введение 3](#_Toc450403288)

[1. Постановка задачи 4](#_Toc450403289)

[2. Обзор технологий 5](#_Toc450403290)

[3. Вычисление числа Пи 13](#_Toc450403291)

[4. Умножение матрицы на вектор 17](#_Toc450403292)

[5. Сортировка массива 20](#_Toc450403293)

[6. Сравнение технологий 27](#_Toc450403294)

[7. Заключение 31](#_Toc450403295)

[8. Литература 32](#_Toc450403296)

# Введение

Современные научные и прикладные исследования часто требуют проведения ресурсоемких вычислений. Соответственно встает проблема ускорения проводимых вычислений и грамотного использования имеющихся ресурсов (память, ядра процессоров и т.д.). Одним из путей достижения ускорения является использование параллельного программирования. Распараллеливании вычислений позволяет более эффективно использовать вычислительные ресурсы. Параллельное программирование применяется для решения задач следующего рода: прогноз погоды, структурная биологии и генетика человека, астрономия, экономика и др.. Чтобы добиться максимальной эффективности параллельного решения задачи, программист должен оптимально распределить данные и вычисления. В то же время при написании параллельной программы сложность программирования резко возрастает (соответственно возрастает вероятность совершения ошибки), по сравнению с написанием последовательной версии программы. Возрастает соответственно и сложность отладки. Также появляются новые виды ошибок, специфичные для параллельных программ: взаимная блокировка (deadlock), гонки данных (race conditions) и т.д. Эти ошибки сложны для обнаружения и вызывают неопределенное поведение программы.

К настоящему моменту создано большое число технологий разработки параллельных программ, в особенности для систем с общей памятью. Вопрос состоит в том, какую из этих технологий лучше использовать.

# 1. Постановка задачи

В рамках данной работы требуется освоить технологии разработки параллельных программ для система с общей памятью: WinAPI, OpenMP и библиотека <thread> (присутствует в языке программирования C++, начиная со стандарта С++11), а также провести их сравнение друг с другом.

Сравнение технологий должно быть выполнено по следующим критериям:

* простота освоения технологии,
* простота использования,
* время выполнения поставленных задач.

Для выполнения сравнения требуется реализовать последовательную и параллельные версии программ (с использованием перечисленных технологий), решающих следующие задачи:

* вычисление числа Пи,
* умножение матрицы на вектор,
* сортировка массива.

# 2. Обзор технологий

Большинство современных операционных систем являются многозадачными и позволяют одновременно выполнять несколько приложений. Разумеется, при наличии одного процессора многозадачность может быть реализована лишь переключением между различными приложениями, которым выделяется определенный квант времени. При достаточной скорости обработки это создает иллюзию одновременного выполнения нескольких приложений [4].

Каждый выполняемый процесс (программа) имеет хотя бы один поток выполнения. Этот поток создается автоматически и условно называется главным. В последовательных программах программный код выполняется только в этом потоке. Однако в процессе можно создать несколько потоков, при этом они будут выполняться параллельно, а процесс может рассматриваться как контейнер для потоков. Все потоки, выполняющиеся в контексте некоторого процесса, разделяют одно адресное пространство, поэтому потоки могут оперировать одними и теми же данными [4].

2.1. WinAPI

2.1.1. Краткое описание технологии WinAPI

Эта технология применима только для ОС семейства Microsoft Windows. В начале выполнения программы создается главный поток, который (если понадобится) может породить множество других потоков. При создании каждый из дополнительных потоков выполняет функцию, которая передается ему как параметр. Для синхронизации работы нескольких потоков используются следующие механизмы:

* Семафоры, мьютексы, критические секции.
* Глобальные переменные.
* Функции, которые ожидают перехода всех выполняющихся потоков в сигнальное состояние (WaitForMultipleObjects).

Для работы с WinAPI потоками требуется подключить библиотеку <windows.h>.

2.1.2. Функция CreateThread

Использование WinAPI потоков начинается с функции CreateThread [[1]](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682453(v=vs.85).aspx):

HANDLE WINAPI CreateThread(

LPSECURITY\_ATTRIBUTES  lpThreadAttributes,

SIZE\_T                 dwStackSize,

LPTHREAD\_START\_ROUTINE lpStartAddress,

LPVOID                 lpParameter,

DWORD                  dwCreationFlags,

LPDWORD                lpThreadId);

LpThreadAttributes – атрибуты защиты. Для присвоения атрибутов защиты по умолчанию можно сюда передать NULL .

DwStackSize –  параметр определяет размер стека, выделяемый для потока из общего адресного пространства процесса. При передаче "0" размер устанавливается в значение по умолчанию.

LpStartAddress – указатель на функцию потока, которая будет им исполняться.

LpParameter – адрес параметра, который будет использоваться внутри функции потока.

DwCreationFlags – принимает одно из двух значений: 0 - исполнение начинается немедленно, или CREATE\_SUSPENDED - исполнение приостанавливается до последующих указаний.

LpThreadId – Адрес переменной типа DWORD, в который функция возвращает идентификатор, приписанный системой новому потоку.

2.1.3. Функция WaitForMultipleObjects

Функция WaitForMultipleObjects используется для ожидания завершения работы потоков [[2]](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms687025(v=vs.85).aspx):

DWORD WINAPI WaitForMultipleObjects(

DWORD nCount,

HANDLE \*lpHandles,

BOOL bWaitAll,

DWORD dwMilliseconds

);

nCount – число потоков (размер массива lpHandles).

lpHandles – массив объектов типа HANDLE (дескрипторы потоков).

bWaitAll – если true, то выход из этой функции произойдет в случае, когда все потоки перейдут в сигнальное состояние. Если false, то достаточно ожидать перехода одного из всех потоков в сигнальное состояние.

dwMilliseconds – сколько времени ждать наступления сигнального состояния потоков. Чтобы ожидать бесконечно, нужно передать значение INFINITE.

2.1.4. Функция CloseHandle

Для удаления описателей потоков из таблицы дескрипторов используется функция CloseHandle [[3]](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms724211(v=vs.85).aspx):

BOOL WINAPI CloseHandle(

HANDLE hObject

);

hObject – описатель объекта типа HANDLE.

2.1.5. Пример использования WinAPI

Простой пример использования технологии WinAPI:

#include <windows.h>

#include <iostream>

DWORD WINAPI HelloWorld(void \*thr\_param){

int index\_thread = \*((int\*)thr\_param);

std::cout << index\_thread << ":" << "Hello World!" << std::endl;

return 0;

}

void main(){

DWORD\* array\_id\_of\_threads;

HANDLE\* array\_of\_threads;

int\* index\_thread; // индексы потоков

int num\_of\_threads;

std::cin >> num\_of\_threads;

array\_id\_of\_threads = new DWORD[num\_of\_threads];

array\_of\_threads = new HANDLE[num\_of\_threads];

index\_thread = new int[num\_of\_threads];

for (int i = 0; i < num\_of\_threads; i++)

index\_thread[i] = i;

for (int i = 0; i < num\_of\_threads; i++)

array\_of\_threads[i] = CreateThread(NULL,

0,

HelloWorld,

&index\_thread[i],

0,

&array\_id\_of\_threads[i]);

WaitForMultipleObjects(num\_of\_threads, array\_of\_threads, true, INFINITE);

for (int i = 0; i < num\_of\_threads; i++)

CloseHandle(array\_of\_threads[i]);

}

В этой программе каждый поток выводит свой индекс и строчку "Hello World!".

2.2. OpenMP

2.2.1. Краткое описание технологии OpenMP

Обычным подходом при организации вычислений для многопроцессорных вычислительных систем с общей памятью является создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений, конечно же, достаточно ограничены. Поэтому второй подход является преобладающим.

В последнее время активно развивается подход к разработке параллельных программ, когда программист организует параллельные вычисления в своей программе при помощи тех или иных внеязыковых средств языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы. При этом исходный текст программы остается неизменным, и по нему, в случае отсутствия препроцессора, компилятор построит исходный последовательный программный код. При применении препроцессора директивы параллелизма заменяются на некоторый дополнительный программный код, который, будучи скомпилированным, будет выполняться несколькими потоками, число которых задано в этих директивах. Такой подход является основой для технологии OpenMP – наиболее широко применяемая в системах с использованием общей памяти. Директивы данной технологии используются для выделения параллельных блоков, в которых содержится последовательный код. Этот код может исполняться несколькими потоками [5].

В итоге, при использовании данной технологии, программа представляется в виде последовательных и параллельных участков кода.

Для работы с OpenMP требуется подключить библиотеку <omp.h>, а также в свойствах проекта (созданного в Microsoft Visual Studio) включить поддержку OpenMP.

2.2.2. Описание директив OpenMP

Общий формат директив OpenMP представляется следующим образом:

#pragma omp <имя\_директивы> [<опция>,[[,]<опция>] ...]

Начальная часть директивы (#pragma omp) является фиксированной, вид директивы определяется ее именем (имя\_директивы), каждая директива может сопровождаться произвольным количеством опций (их наличие не обязательно).

Работа с OpenMP начинается со следующей директивы (для выделения параллельных фрагментов программы):

#pragma omp parallel [<опция>, ...]

<блок\_программы>

Здесь приведен список опций директивы parallel:

1. if (scalar\_expression)
2. private (list)
3. shared (list)
4. default (shared | none)
5. firstprivate (list)
6. reduction (operator: list)
7. num\_threads (scalar\_expression)

Поясним их назначение:

if (scalar\_expression) – если scalar\_expression есть true, то указанный под директивой блок программы распараллеливается, иначе выполняется последовательным образом.

private (list) – указывается список переменных, которые будут личными для каждого потока.

shared (list) – указывается список переменных, которые будут общими для каждого потока (переменные будут общими и без указания этой опции по умолчанию)

default (shared | none) – отмена правила вхождения переменных по умолчанию в распараллеливаемый блок программы.

firstprivate (list) – указанные параметры инициализируются значениями, которые были получены до момента выполнения распараллеливаемого участка кода.

reduction (operator: list) – указанные в списке параметры инициализируются значениями, полученными после исполнения распараллеливаемого блока кода таким образом, что локальные переменные (указанные в list) каждого потока "объединяются" с помощью указанного оператора operator . Результаты таких объединений присваиваются на выходе соответствующим переменным в list .

num\_threads (scalar\_expression) – указывается число потоков, которые будут выполнять указанный блок кода. Их число задается на основе результата, выдаваемого выражением scalar\_expression .

Программный код блока директивы parallel по умолчанию исполняется всеми потоками. Данный способ распараллеливания эффективен в том случае, когда нужно выполнить одни и те же действия многократно. В рамках данного подхода в OpenMP наряду с обычным повторением в потоках одного и того же программного кода – как в директиве parallel – можно осуществить разделение итеративно-выполняемых действий в циклах для непосредственного указания, над какими данными должны выполняться соответствующие вычисления. Такая возможность является тем более важной, поскольку во многих случаях именно в циклах выполняется основная часть вычислительно-трудоемких вычислений.

Для распараллеливания циклов в OpenMP применяется директива for:

#pragma omp for [<опция> ...]

<цикл\_for>

Опции директивы for:

1. schedule (type [,chunk])
2. ordered
3. private (list)
4. shared (list)
5. firstprivate (list)
6. lastprivate (list)
7. reduction (operator: list)

Поясним значение ранее не рассмотренных опций:

schedule (type [,chunk]) – при помощи этой опции можно управлять распределением итераций цикла между потоками. Поле type опции schedule может принимать следующие значения:

* static – статический способ распределения итераций до начала выполнения цикла. Если поле chunk не указано, то итерации делятся поровну между потоками. При заданном значении chunk итерации цикла делятся на блоки размера chunk и эти блоки распределяются между потоками до начала выполнения цикла.
* dynamic – динамический способ распределения итераций. До начала выполнения цикла потокам выделяются блоки итераций размера chunk (если поле chunk не указано, то полагается значение chunk = 1). Дальнейшее выделение итераций (также блоками размера chunk) осуществляется в зависимости от завершения потоками своих ранее назначенных итераций.
* guided – управляемый способ распределения итераций. Данный способ близок к предшествующему варианту. Отличие состоит в том, что начальный размер блоков итераций определяется в соответствии с некоторым параметром среды реализации OpenMP, а затем уменьшается экспоненциально (следующее значение chunk есть некоторая доля предшествующего значения) при каждом новом выделении блока итераций. При этом получаемый размер блока итераций не должен быть меньше значения chunk (если поле chunk не указано, то полагается значение chunk = 1).

ordered – используется в случае, если для ряда действий в цикле необходимо сохранить первичный порядок вычислений, который соответствует последовательному выполнению итераций в последовательной программе (например, это может пригодиться для печати элементов матрицы упорядоченно по строкам).

Для синхронизации действий потоков над общими данными можно использовать критические секции (блоки программного кода, которые могут выполняться только одним потоком в каждый конкретный момент времени). При попытке входа в критическую секцию, которая уже исполняется одним из потоков, все другие потоки приостанавливаются. Как только критическая секция освобождается занявшим ее потоком, один из приостановленных потоков (если такой имеется) активизируется для выполнения критической секции.

При помощи директивы critical объявляется критическая секция:

#pragma omp critical

<блок\_кода>

2.2.3. Пример использования OpenMP

Ниже представлен код простой программы, демонстрирующей использование технологии OpenMP:

#include <omp.h>

#include <iostream>

#define num\_of\_threads 4; // число потоков

void main(){

int thread\_num; // номер потока

#pragma omp parallel num\_threads(num\_of\_threads) private(thread\_num)

{

thread\_num = omp\_get\_thread\_num();

std::cout << thread\_num << ":" <<"Hello world! " << std::endl;

}

}

Каждый из 4-х потоков выведет свой номер и строчку "Hello world! " .

2.3. Библиотека <thread> [6]

2.3.1. Краткое описание библиотеки <thread>

Стандартная поддержка параллелизма за счет многопоточности – вещь новая для языка C++. Только новый стандарт С++11 позволяет писать многопоточный код, не прибегая к платформенно-зависимым расширениям (например рассмотренная ранее технология WinAPI).

В стандартную библиотеку <thread> включены классы для управления потоками, защиты разделяемых данных, синхронизации операций между потоками. Эта библиотека предлагает высокоуровневые абстракции и средства, позволяющие писать программисту многопоточный код проще, с меньшим количеством ошибок и, не особо вдаваясь в детали устройства потоков. Естественно из-за такого абстрагирования возможно снижение производительности выполнения кода. Но, как правило, это не критично в силу того, что производительность снижается на достаточно малую величину [6].

Для работы с этими потоками требуется подключить библиотеку <thread>.

2.3.2. Основные функции библиотеки <thread>

Для создания потока достаточно воспользоваться конструктором класса thread:

thread t(SomeFunction, Param);

SomeFunction – функция, которая исполняется потоком.

Param – адрес параметра, который передается потоку для выполнения функции SomeFunction.

Чтобы дождаться завершения потока, следует вызвать функцию join() у соответствующего потока. В отличии от технологии WinAPI, в технологии с использованием библиотеки <thread> допускается использование числа потоков большего, чем 64.

Также в отличии от WinAPI, программисту не нужно заботиться об удалении дескрипторов потоков. Это делается автоматически деструктором класса thread.

2.3.2. Пример использования библиотеки <thread>

Ниже представлена простая программа с использованием C++ библиотеки <thread>:

#include <thread>

#include <iostream>

#define num\_of\_threads 4

void HelloWorld(void\* thr\_param)

{

int index\_thread = \*((int\*)thr\_param);

std::cout << index\_thread << ":" << "Hello World!" << std::endl;

}

void main(){

thread\* array\_of\_thread = new thread[num\_of\_threads];

int\* index\_thread = new int[4];

for(int i = 0; i < num\_of\_threads; i++)

index\_thread[i] = i;

for(int i = 0; i < num\_of\_threads; i++)

array\_of\_threads[i] = thread::thread(HelloWorld, &index\_thread[i]);

for(int i = 0; i < num\_of\_threads; i++)

array\_of\_threads[i].join();

}

# 3. Вычисление числа Пи

3.1. Постановка задачи вычисления числа Пи

Требуется вычислить число Пи методом прямоугольников с использованием следующего интеграла:

Площадь фигуры, ограниченной кривой и линиями и , будет численно равна значению числа Пи.

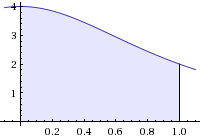


Рисунок 1. График функции

Метод прямоугольников заключается в разбиении участка под кривой (рис. 1) на заданное число прямоугольников и суммировании площадей этих прямоугольников, что дает требуемый результат вычислений. Чем больше прямоугольников, тем больше точность вычислений.

3.2. Последовательная версия решения задачи

Последовательная версия заключается в суммировании площадей прямоугольников. Ниже представлен код этой версии:

double value\_of\_function(double x)

{

return 4.0 / (1.0 + x \* x);

}

double Calculate\_pi(int num\_of\_rectangles)

{

double pi = 0.0;

double step = 1.0 / num\_of\_rectangles;

double curr\_arg = 0;

double sum = 0.0;

for (int i = 0; i < num\_of\_rectangles; i++)

{

curr\_arg = step \* (double)(i + 0.5);

sum += value\_of\_function(curr\_arg);

}

pi = sum \* step;

return pi;

}

3.3. Параллельная версия решения задачи

Пусть было проведено разбиение области (рис. 1) на n прямоугольников. Общая идея распараллеливания вычисления числа Пи состоит в том, что каждому потоку, участвующему в процессе вычисления, отводятся номера прямоугольников из множества , которые отдельно для каждого потока определяются по формуле:

*,* где

– номер текущего потока,

*i* –номер текущей итерации основного цикла вычислений,

– число потоков, участвующих в вычислении.

При окончании подсчета каждым потоком, происходит объединение полученных результатов. Конечный результат будет численно равен числу Пи.

3.3.1. Реализация на WinAPI

На вход программе подаются следующие параметры:

num\_of\_threads – число потоков,

num\_of\_rectangles –число прямоугольников.

Далее главный поток создает потоки для вычисления частичных сумм и ожидает их завершения:

for (int i = 0; i < num\_of\_threads; i++)

array\_of\_threads[i] = CreateThread(NULL, 0, Calculate\_pi, &index\_thread[i], 0,&array\_id\_of\_threads[i]);

WaitForMultipleObjects(num\_of\_threads, array\_of\_threads, true, INFINITE);

Здесь index\_thread[i] – номер i-го потока и он передается в качестве параметра функции Calculate\_pi, представленной ниже:

DWORD WINAPI Calculate\_pi(void \*thr\_param)

{

int start = \*((int\*)thr\_param);

int i = 0;

double step = 1.0 / (double)(num\_of\_rectangles);

double arg;

double sum = 0.0;

for (i = start; i < num\_of\_rectangles; i = i + num\_of\_rectangles)

{

arg = step \* (i + 0.5);

sum += 4.0 / (1.0 + arg \* arg); // Считаем площадь прямоугольника

}

array\_of\_local\_summ[start] = sum \* step;

return 0;

}

По завершении вычислений в созданных потоках, главный поток выполняет очистку таблицы дескрипторов потоков и выполняет объединение частичных сумм, полученных при выполнении потоками функции Calculate\_pi:

for (int i = 0; i < num\_of\_threads; i++)

CloseHandle(array\_of\_threads[i]);

for (int i = 0; i < num\_of\_threads; i++)

pi += array\_of\_local\_summ[i];

Здесь переменная pi содержит итоговый результат.

3.3.2.Реализация с использованием библиотеки <thread>

На вход программе подаются следующие параметры:

num\_of\_threads – число потоков,

num\_of\_rectangles – число прямоугольников.

Далее главный поток создает потоки для вычисления частичных сумм и ожидает их завершения при помощи метода join():

for (int i = 0; i < num\_of\_threads; i++)

mas\_of\_threads[i] = thread::thread(CalculationPi, &index\_thread[i]);

for (int i = 0; i < num\_of\_threads; i++)

mas\_of\_threads[i].join();

Здесь index\_thread[i] и функция CalculationPi организованы так же, как в реализации версии вычисления числа Пи с использованием технологии WinAPI.

По окончании вычислений в созданных потоках, главный поток выполняет суммирование полученных частичных сумм:

for (int i = 0; i < num\_of\_threads; i++)

pi += rez\_of\_threads[i];

Переменная pi содержит итоговый результат.

3.3.3.Реализация с использованием OpenMP

На вход программе подаются следующие параметры:

num\_of\_threads – число потоков,

num\_of\_rectangles – число прямоугольников.

Далее при достижении главным потоком директивы #pragma omp parallel, создается num\_of\_threads потоков, которые выполняют подсчет частичных сумм согласно алгоритму планирования static (Результаты работы потоков содержатся в переменной sum и по окончанию работы потоков выполняется объединение этих результатов в один при помощи reduction(+: sum)):

#pragma omp parallel num\_threads(num\_of\_threads) private(arg)

{

#pragma omp for schedule(static, chunk) reduction(+: sum)

for (i = 0; i < num\_of\_intervals; i++)

{

arg = step \* (double)(i - 0.5);

sum += 4.0 / (1.0 + arg \* arg);

}

}

pi = step \* sum;

Значение переменной pi является итоговым ответом.

# 4. Умножение матрицы на вектор

4.1. Постановка задачи умножения матрицы на вектор

Пусть дана матрица и вектор (без ограничения общности рассмотрим матрицу размером 3х3 и вектор размером 3х1):

Умножение матрицы на вектор производится по следующему правилу: каждая строка матрицы умножается на вектор . Число столбцов в матрице должно совпадать с числом строк в векторе . Приведенные выше матрица и вектор удовлетворяют этому условию, следовательно, вычисление произведения допускается:

Далее мы будем рассматривать только умножение квадратной матрицы на вектор.

4.2. Последовательная версия решения задачи

Последовательная версия вычисления произведения заключается в последовательном умножении строк матрицы на вектор столбец . Ниже представлен код функции последовательного умножения матриц:

void Multiplication()

{

double SumJ = 0.0;

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

SumJ += Matr[i \* Size + j] \* Vec[j];

Res[i] = SumJ;

SumJ = 0.0;

}

}

4.3. Параллельная версия решения задачи

Основная идея параллельной версии умножения матрицы на вектор заключается в том, что каждому потоку отводится некоторое число строк матрицы , и соответственно каждый из них выполняет скалярные произведения между выданными потоку строками и вектором столбцом . После этого результаты, полученные потоками, объединяются и получается итоговый результат вычисления произведения.

4.3.1. Реализация с использованием WinAPI

На вход программе подаются следующие параметры:

num\_of\_threads – число потоков,

Size – размер квадратной матрицы.

Далее главный поток создает потоки для вычисления скалярных произведений и ожидает их завершения. Каждому потоку отводится строк матрицы , а если это выражение не делится нацело, то остаток от деления передается последнему по номеру потоку. Ниже представлен фрагмент функции main:

for (int i = 0; i < num\_of\_threads; i++)

array\_of\_threads[i] = CreateThread(NULL, 0, Multiplication\_pp, &index\_thread[i], 0, &array\_id\_of\_threads[i]);

WaitForMultipleObjects(num\_of\_threads, array\_of\_threads, true, INFINITE)

pp\_alg\_time\_end = omp\_get\_wtime();

pp\_work\_time = pp\_alg\_time\_end - pp\_alg\_time\_start;

for (int i = 0; i < num\_of\_threads; i++)

CloseHandle(array\_of\_threads[i]);

Функция Multiplication\_pp, которая фигурирует в выше представленном коде, имеет следующую реализацию (в эту функцию передается номер потока - именно по нему возможно определить, какие номера строк должен использовать поток для вычисления скалярных произведений):

DWORD WINAPI Multiplication\_pp(void \*thr\_param)

{

int index = \*((int\*)thr\_param);

double SumJ = 0.0;

int start = index \* Work\_for\_threads;

int end = index == num\_of\_threads - 1 ? Size : start + Work\_for\_threads;

for (int i = start; i < end; i++)

{

for (int j = 0; j < Size; j++)

SumJ += Matr[i \* Size + j] \* Vec[j];

Res\_pp[i] = SumJ;

SumJ = 0.0;

}

}

По завершению вычислений созданных потоков, главный поток выполняет чистку таблицы дескрипторов потоков, а массив Res\_pp содержит итоговый результат умножения матрицы на вектор (Res\_pp объявлен глобально).

4.3.2. Реализация с использованием библиотеки <thread>

На вход программе подаются следующие параметры:

num\_of\_threads – число потоков,

Size – размер квадратной матрицы.

Далее главный поток создает потоки для вычисления скалярных произведений и ожидает их завершения при помощи метода join():

for (int i = 0; i < num\_of\_threads; i++)

array\_of\_threads[i]=thread::thread(Multiplication\_pp, &index\_thread[i]);

for (int i = 0; i < num\_of\_threads; i++)

array\_of\_threads[i].join();

Здесь index\_thread и Multiplication\_pp организованы так же, как и в версии WinAPI. Итоговый результат умножения содержится в массиве Res\_pp, который заполняется потоками при выполнении функции Multiplication\_pp.

4.3.3. Реализация с использованием OpenMP

На вход программе подаются следующие параметры:

num\_of\_threads – число потоков,

Size – размер квадратной матрицы.

Далее при достижении главным потоком директивы #pragma omp parallel, создается num\_of\_threads потоков, которые выполняют вычисление скалярных произведений (здесь по умолчанию задан алгоритм планирования распараллеливания цикла static):

#pragma omp parallel num\_threads(num\_of\_threads) firstprivate(SumJ)

{

// Распараллеливание цикла

#pragma omp for private(j)

for (i = 0; i < Size; i++)

{

for (j = 0; j < Size; j++)

SumJ += Matr[i \* Size + j] \* Vec[j];

Res\_pp[i] = SumJ;

SumJ = 0;

}

}

По окончанию работы умножения потоками, Res\_pp будет содержать итоговый результат.

# 5. Сортировка массива

5.1. Постановка задачи сортировки массива

Дан неупорядоченный массив, состоящий из N элементов типа double:

Требуется реализовать последовательную и параллельную версии сортировки данного массива. В рамках поставленной задачи будут рассматриваться пузырьковая сортировка и сортировка слиянием [7].

5.2. Последовательная версия решения задачи

Рассмотрим пузырьковую сортировку методом чет-нечетных перестановок (одна из модификаций пузырьковой сортировки) [[8](http://www.intuit.ru/studies/courses/1156/190/lecture/4958?page=2)]. Основная идея заключается в следующем: в алгоритм сортировки вводятся два разных правила выполнения итераций метода – в зависимости от четности или нечетности номера итерации для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами. Таким образом, на всех нечетных итерациях сравниваются пары (при четном n), а на четных итерациях обрабатываются элементы . После n-кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

Программная реализация последовательной версии пузырьковой сортировки методом чет-нечетных перестановок представлена ниже:

void BubbleSort\_seq(int\* arr)

{

for (int i = 0; i < Size; i++)

if (i % 2 == 1) // нечетная итерация

{

for (int j = 0; j < Size / 2 - 1; j++)

if (arr[2 \* j + 1] > arr[2 \* j + 2])

Swap(arr[2 \* j + 1], arr[2 \* j + 2]);

if (Size % 2 == 1) // если нечетный размер, то нужно сравнить последнюю пару

if (arr[Size - 2] > arr[Size - 1])

Swap(arr[Size - 2], arr[Size - 1]);

}

else // четная итерация

for (int j = 0; j < Size / 2; j++)

if (arr[2 \* j] > arr[2 \* j + 1])

Swap(arr[2 \* j], arr[2 \* j + 1]);

}

Рассмотрим последовательную версию сортировки слиянием. Суть данного метода заключается в том, что сортируемый массив разбивается на части практически одного размера (размеры этих массивов отличаются не более, чем на единицу). Для каждой из получившихся частей вызывается этот же алгоритм сортировки слиянием. После упорядочивания элементов, два упорядоченных массива объединяются в один упорядоченный.

Разбиение массива выполняется рекурсивно и это разбиение происходит до тех пор, пока размер массива не станет равным единице (массив с одним элементом является упорядоченным).

Программная реализация последовательной версии сортировки слиянием представлена ниже:

void MergeSort\_seq(int left, int right, int\* mas)

{

int s;

if (left < right)

{

s = (left + right) / 2;

MergeSort\_seq(left, s, mas);

MergeSort\_seq(s + 1, right, mas);

Merge\_seq(left, s, right, mas);

}

}

void Merge\_seq(int left, int s, int right, int\* mas)

{

int tmp1 = left;

int tmp2 = s + 1;

int tmp3 = 0;

while (tmp1 <= s && tmp2 <= right)

{

if (mas[tmp1] < mas[tmp2])

mas\_merge\_seq[tmp3++] = mas[tmp1++];

else

mas\_merge\_seq[tmp3++] = mas[tmp2++];

}

while (tmp1 <= s)

mas\_merge\_seq[tmp3++] = mas[tmp1++];

while (tmp2 <= right)

mas\_merge\_seq[tmp3++] = mas[tmp2++];

for (tmp3 = 0; tmp3 < right - left + 1; tmp3++)

mas[left + tmp3] = mas\_merge\_seq[tmp3];

}

MergeSort\_seq выполняет разбиение массива mas (с индексом начала left и индексом конца right) на две части. Две упорядоченных части массива сливаются при помощи функции Merge\_seq.

5.3. Параллельная версия решения задачи

Рассматриваемая ниже версия параллельной сортировки работает только в случае, когда число потоков является степенью двойки.

Для распараллеливания сортировки используем простое слияние. Идея параллельной реализации с использованием простого слияния заключается в выполнении следующих шагов:

* Каждый поток сортирует свою часть исходного массива.
* После сортировки частей массива выполняется слияние этих частей.

Пусть дан неупорядоченный массив, состоящий из 10 элементов. Предположим, что для сортировки массива было выделено 4 потока. Тогда каждому потоку выдается элемента исходного массива для выполнения первого этапа параллельной версии сортировки, а остаток от этого деления передается последнему по номеру потоку (т.е. первые 3 потока будут иметь в распоряжении 2 элемента исходного массива, а 4-ый будет иметь 4 элемента):

На 1-м этапе можно использовать одну из ранее рассмотренных последовательных версий сортировок (пузырьковая сортировка или сортировка слиянием) для упорядочивания частей исходного массива. Тогда после исполнения 1-го этапа всеми потоками получаем следующий результат:

После исполнения этого этапа, 2-й и 4-й потоки завершают свою работу.

Изобразим схематично работу 2-го этапа параллельной версии сортировки. Основная суть работы этого этапа заключается в том, что соседние потоки выполняют слияние своих упорядоченных частей в большую упорядоченную часть (а именно поток, который находится "слева", выполняет слияние). Конкретно в нашем случае, 1-й поток с элементами выполнит слияние с элементами 2-го потока, а 3-й поток с элементами ( выполнит слияние с элементами 4-го потока, т.е. получаем:

После исполнения этого этапа 3-й по номеру поток завершает свою работу, а 1-й поток с элементами выполняет слияние с элементами 3-го потока. После исполнения этого этапа получаем результат:

5.3.1.Реализация с использованием WinAPI

На вход программе подаются следующие параметры:

num\_of\_threads – число потоков,

Size – размер неупорядоченного массива.

Далее главный поток создает потоки для сортировки заданного массива и ожидает их завершения. Каждому потоку отводится элементов массива, а если это выражение не делится нацело, то остаток от деления передается последнему по номеру потоку. Ниже представлен фрагмент функции main (параллельная версия сортировки с использованием пузырьковой сортировки для сортировки частей исходного массива каждым потоком):

if (PowerOfTwo(num\_of\_threads))

{

for (int i = 0; i < num\_of\_threads; i++)

array\_of\_threads[i] = CreateThread(NULL, 0, BubbleSort\_pp, &index\_thread[i], 0, &array\_id\_of\_threads[i]);

WaitForMultipleObjects(num\_of\_threads, array\_of\_threads, true, INFINITE);

for (int i = 0; i < num\_of\_threads; i++)

CloseHandle(array\_of\_threads[i]);

}

Для сортировки массива mas\_pp\_bubble используется функция BubbleSort\_pp. Реализация функции BubbleSort\_pp представлена ниже:

DWORD WINAPI BubbleSort\_pp(void\* thr\_param)

{

int index\_thread = \*((int\*)(thr\_param));

int work = Size / num\_of\_threads;

int start\_pos = work \* index\_thread;

int end\_pos = start\_pos + work;

int exponent = log2(num\_of\_threads);

if (index\_thread == num\_of\_threads - 1)

end\_pos = Size;

for (int i = 0; i < end\_pos - start\_pos; i++)

for (int j = start\_pos; j < end\_pos - i - 1; j++)

if (mas\_pp\_bubble[j] > mas\_pp\_bubble[j + 1])

Swap(mas\_pp\_bubble[j], mas\_pp\_bubble[j + 1]);

if (index\_thread == num\_of\_threads - 1)

end\_pos = Size - 1;

if (index\_thread % 2 != 0)

ThreadIsFinish[index\_thread] = true;

for (int i = 0; i < exponent; i++)

if (index\_thread % (int)pow(2, i + 1) == 0)

MergeSplit\_right(index\_thread, start\_pos, end\_pos, work, i);

return 0;

}

Для достижения синхронизированного выполнения параллельной версии сортировки предлагается добавить следующий механизм синхронизации:

volatile bool\* ThreadIsFinish;

Массив ThreadIsFinish имеет размер, равный num\_of\_threads. Если ThreadIsFinish[i] = true, то поток с индексом i завершил свою работу, иначе поток i все еще выполняет свою работу (изначально ThreadIsFinish проинициализирован значениями false). Спецификатор volatile означает, что ThreadIsFinish не будет оптимизироваться компилятором и указывает на то, что ThreadIsFinish будет меняться несколькими потоками.

В функции BubbleSort\_pp каждый из потоков сортирует свои части исходного массива. После этого, согласно изложенному выше алгоритму параллельной версии сортировки, каждый второй поток завершает свою работу, а остальные потоки выполняют функцию MergeSplit\_right (для выполнения слияния упорядоченных массивов), реализация которой представлена ниже:

void MergeSplit\_right(int index\_thread, int start\_pos, int& end\_pos, int work, int i)

{

int start\_pos\_right = (index\_thread + (int)pow(2, i)) \* work;

int end\_pos\_right = start\_pos\_right + work \* pow(2, i) - 1;

if (index\_thread + (int)pow(2, i + 1) == num\_of\_threads)

end\_pos\_right = Size - 1;

int tmp\_1 = start\_pos;

int tmp\_2 = start\_pos\_right;

int tmp\_3 = 0;

int size\_merge\_mas = end\_pos\_right - start\_pos + 1;

int\* merge\_mas = new int[size\_merge\_mas];

/\* Синхронизация \*/

while (!ThreadIsFinish[index\_thread + (int)pow(2, i)]); // Ожидаем поток, с которым будем сливать массивы

/\* MERGE \*/

while (tmp\_1 < end\_pos && tmp\_2 <= end\_pos\_right)

if (mas\_pp\_bubble[tmp\_1] < mas\_pp\_bubble[tmp\_2])

merge\_mas[tmp\_3++] = mas\_pp\_bubble[tmp\_1++];

else

merge\_mas[tmp\_3++] = mas\_pp\_bubble[tmp\_2++];

while (tmp\_1 < end\_pos)

merge\_mas[tmp\_3++] = mas\_pp\_bubble[tmp\_1++];

while (tmp\_2 <= end\_pos\_right)

merge\_mas[tmp\_3++] = mas\_pp\_bubble[tmp\_2++];

/\* SPLIT \*/

for (tmp\_3 = 0; tmp\_3 < size\_merge\_mas; tmp\_3++)

mas\_pp\_bubble[start\_pos + tmp\_3] = merge\_mas[tmp\_3];

end\_pos = end\_pos\_right + 1; // У этого потока массив увеличился до размера size\_merge\_mas => необходимо изменить end\_pos

if (index\_thread % (int)pow(2, i + 1) == 0 && index\_thread % (int)pow(2, i + 2) != 0)

ThreadIsFinish[index\_thread] = true; // То этот поток больше работать не будет, он выполнил свою задачу

delete[] merge\_mas;

}

Функция MergeSplit\_right идентична по смыслу функции Merge\_seq. Работа потоков синхронизируется следующим образом (механизм активного ожидания):

while (!ThreadIsFinish[index\_thread + (int)pow(2, i)]);

где i - текущий номер итерации цикла в функции BubbleSort\_pp. Тем самым исключается ситуация гонки потоков, а именно, например, когда 1-й поток начал выполнять функцию MergeSplit\_right, а 2-й поток еще не успел отсортировать свою часть исходного массива.

После завершения всеми потоками функции BubbleSort\_pp, mas\_pp\_bubble является отсортированным.

Время работы параллельной версии сортировки можно уменьшить за счет сортировки потоками частей исходного массива с помощью последовательной версии сортировки слиянием. Реализация такой версии (функция MergeSort\_pp) практически ничем не отличается от той, что представлена выше, за исключением того, что части исходного массива будут сортироваться при помощи сортировки слиянием.

5.3.2. Реализация с использованием библиотеки <thread>

На вход программе подаются следующие параметры:

num\_of\_threads – число потоков,

Size – размер неупорядоченного массива.

Далее главный поток создает потоки для сортировки данного массива и ожидает их завершения при помощи метода join(). Каждому потоку отводится элементов исходного массива, а если это выражение не делится нацело, то остаток от деления передается последнему по номеру потоку. Ниже представлен фрагмент функции main:

if (PowerOfTwo(num\_of\_threads))

{

for (int i = 0; i < num\_of\_threads; i++)

mas\_of\_threads[i] = thread::thread(BubbleSort\_pp, &index\_thread[i]);

for (int i = 0; i < num\_of\_threads; i++)

mas\_of\_threads[i].join();

}

Для сортировки массива mas\_pp\_bubble используется функция BubbleSort\_pp, реализация которой аналогична реализации в технологии WinAPI. Это же замечание касается и функции MergeSort\_pp, в которой части исходного массива сортируются потоками при помощи последовательной версии сортировки слиянием.

# 6. Сравнение технологий

Сравнение технологий будет проводиться согласно критериям, описанным в постановке задачи. Ниже представлена таблица, позволяющая сравнить рассмотренные ранее технологии по двум критериям: "Простота использования" и "Простота освоения технологии".

Таблица 1.Сравнение технологий по двум критериям

|  |  |  |
| --- | --- | --- |
|  | Простота использования | Простота освоения технологии |
| Последовательная версия | + | + |
| WinAPI | -+ | +- |
| OpenMP | +- | + |
| Библиотека <thread> | +- | + |

Очевидно, что последовательная версия проще в разработке и изучении, так как она не требует дополнительных усилий для организации вычислений, а нацелена только на прямое исполнение алгоритма. Последовательная версия зачастую эффективна при малых значениях размеров некоторых входных данных (например, массив).

При использовании средств WinAPI или библиотеки <thread> требуется заботиться о синхронизации данных (в частности, избегать ситуации гонок потоков). Для этого необходимо использовать механизмы синхронизации (мьютексы, семафоры и т.д.) в целях защиты критических секций, а также создавать структуры данных, которые будут содержать индивидуальные данные для каждого потока. Также, программист ответственен за создание и уничтожение потоков, тогда как в OpenMP это делается автоматически.

OpenMP предоставляет достаточно простой интерфейс по управлению потоками (и их параметрами) для распараллеливания алгоритмов. Многие действия, которые нужно осуществить программисту в WinAPI собственноручно, в OpenMP реализованы неявно (например, в WinAPI нужно ждать завершения выполнения всех потоков с помощью функции WaitForMultipleObjects, тогда как в OpenMP в конце параллельной секции ожидание завершения вычислений всех потоков происходит автоматически).

В разделах ниже будет проведено сравнение технологий по критерию эффективности выполнения поставленных нами ранее задач. Вычисления будут производиться на ПК с одним процессором, у которого есть 4 ядра.

6.1. Результаты выполнения задачи вычисления числа Пи

Ниже представлены результаты вычисления числа Пи при разном числе потоков (обозначим их за P) и при нескольких значениях количества прямоугольников N для разбиения.

Таблица 2.Время работы последовательной версии

|  |  |  |  |
| --- | --- | --- | --- |
| Число прямоугольников | N=10000 | N=1000000 | N=100000000 |
| Время выполнения, сек | 0.00019 | 0.01199 | 0.50657 |

Таблица 3.Время работы версии WinAPI

|  |  |  |  |
| --- | --- | --- | --- |
|  | N=10000 | N=1000000 | N=100000000 |
| P =2 | 0.00889 | 0.01407 | 0.26106 |
| P =3 | 0.00909 | 0.01463 | 0.23389 |
| P =4 | 0.00977 | 0.01259 | 0.16479 |

Таблица 4.Время работы версии OpenMP

|  |  |  |  |
| --- | --- | --- | --- |
|  | N=10000 | N=1000000 | N=100000000 |
| P =2 | 0.00734 | 0.01175 | 0.24952 |
| P =3 | 0.00753 | 0.01178 | 0.20754 |
| P =4 | 0.00834 | 0.01263 | 0.19992 |

Таблица 5.Время работы версии с использованием библиотеки <thread>

|  |  |  |  |
| --- | --- | --- | --- |
|  | N=10000 | N=1000000 | N=100000000 |
| P =2 | 0.01241 | 0.01852 | 0.26311 |
| P =3 | 0.01421 | 0.01813 | 0.18068 |
| P =4 | 0.01504 | 0.01783 | 0.16621 |

Проведем анализ получившихся результатов. При малых N наиболее эффективнее будет использовать последовательную версию вычисления числа Пи. При больших N появляется линейное ускорение от числа потоков. Согласно данным из таблиц выше можно утверждать, что все рассматриваемые технологии (для распараллеливания поставленной задачи) выполнили вычисления практически за одинаковые промежутки времени, но технология с использованием библиотеки <thread> решила эту задачу чуть быстрее. В этой задаче производилось распараллеливание по вычислениям.

6.2. Результаты выполнения задачи умножения матрицы на вектор

Ниже представлены результаты умножения матрицы на вектор при разном числе потоков (обозначим их за P) и при нескольких значениях размера Size для квадратной матрицы и вектора.

Таблица 6.Время работы последовательной версии

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Size=1000 | Size=10000 | Size=13500 |
| Время выполнения, сек | 0.00251 | 0.12302 | 0.20394 |

Таблица 7.Время работы версии WinAPI

|  |  |  |  |
| --- | --- | --- | --- |
|  | Size=1000 | Size=10000 | Size=13500 |
| P =2 | 0.01080 | 0.15180 | 0.20340 |
| P =3 | 0.00993 | 0.08044 | 0.11951 |
| P =4 | 0.01063 | 0.07396 | 0.11420 |

Таблица 8.Время работы версии OpenMP

|  |  |  |  |
| --- | --- | --- | --- |
|  | Size=1000 | Size=10000 | Size=13500 |
| P =2 | 0.01013 | 0.09402 | 0.16272 |
| P =3 | 0.00915 | 0.07258 | 0.13048 |
| P =4 | 0.01003 | 0.07711 | 0.11735 |

Таблица 9.Время работы версии с использованием библиотеки <thread>

|  |  |  |  |
| --- | --- | --- | --- |
|  | Size=1000 | Size=10000 | Size=13500 |
| P =2 | 0.01293 | 0.07730 | 0.14159 |
| P =3 | 0.01418 | 0.06962 | 0.11384 |
| P =4 | 0.01481 | 0.06697 | 0.10680 |

Проведем анализ получившихся результатов. При малых N наиболее эффективнее будет использовать последовательную версию умножения матрицы на вектор. При больших N появляется линейное ускорение от числа потоков. Согласно данным из таблиц выше можно утверждать, что все рассматриваемые технологии (для распараллеливания поставленной задачи) выполнили вычисления практически за одинаковые промежутки времени, но технология с использованием библиотеки <thread> решила эту задачу чуть быстрее. В этой задаче производилось распараллеливание по данным.

6.3. Результаты выполнения задачи сортировки массива

Ниже представлены результаты умножения матрицы на вектор при разном числе потоков (обозначим их за P) и при нескольких значениях размера Size для квадратной матрицы и вектора.

Таблица 10. Время работы последовательных версий

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Size=1000 | Size=10000 | Size=100000 | Size=1000000 |
| Пузырьковая сортировка | 0.00186 | 0.09768 | 9.14151 | >1000 |
| Сортировка слиянием | 0.00019 | 0.00214 | 0.02319 | 0.11644 |

Таблица 11.Время работы версии WinAPI (BubbleSort\_pp)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Size=1000 | Size=10000 | Size=100000 | Size=1000000 |
| P =2 | 0.06009 | 0.04996 | 3.77016 | 392.666 |
| P =4 | 0.00427 | 0.01717 | 1.04984 | 104.109 |

Таблица 12.Время работы версии WinAPI (MergeSort\_pp)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Size=1000 | Size=10000 | Size=100000 | Size=1000000 |
| P =2 | 0.00271 | 0.00355 | 0.01566 | 0.07760 |
| P =4 | 0.00406 | 0.00509 | 0.01144 | 0.04694 |

Таблица 13.Время работы BubbleSort\_pp с использованием библиотеки <thread>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Size=1000 | Size=10000 | Size=100000 | Size=1000000 |
| P =2 | 0.00190 | 0.05259 | 3.70497 | 391.492 |
| P =4 | 0.00783 | 0.02095 | 1.02206 | 104.181 |

Таблица 13.Время работы MergeSort\_pp с использованием библиотеки <thread>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Size=1000 | Size=10000 | Size=100000 | Size=1000000 |
| P =2 | 0.00553 | 0.00415 | 0.01598 | 0.08266 |
| P =4 | 0.00492 | 0.00616 | 0.01237 | 0.05796 |

Проведем анализ получившихся результатов. При малых N наиболее эффективнее будет использовать последовательные версии сортировок массива, причем последовательная версия сортировки слиянием, как и ожидалось, отработала быстрее, чем пузырьковая. При больших N параллельные версии сортировок работают быстрее по сравнению с последовательной. Последовательная версия пузырьковой сортировки и BubbleSort\_pp сравнению не поддаются, так как BubbleSort\_pp использует действия (помимо сортировки потоками своих частей исходного массива с помощью последовательной версии сортировки), которые присутствуют в сортировке слиянием и соответственно из-за этого ускорение получилось большим, чем линейное от числа потоков. Для корректного сравнения последовательной и параллельной версии сортировок рассматривались сортировка слиянием (последовательная версия) и MergeSort\_pp, в которой каждый поток сортирует свои части исходного массива при помощи использования последовательной версии сортировки слиянием. Ускорение, даваемое параллельной версией, должно быть меньше линейного, так как на каждом этапе выполнения функции MergeSort\_pp завершают свою работу половина потоков (от текущего числа работающих потоков). Эти рассуждения подтверждаются на практике (согласно результатам в таблицах выше). Все рассматриваемые технологии (для распараллеливания поставленной задачи) выполнили вычисления практически за одинаковые промежутки времени, но технология WinAPI решила эту задачу чуть быстрее.

# 7. Заключение

В рамках данной работы были освоены технологии разработки параллельных программ для систем с общей памятью(WinAPI, OpenMP и библиотека <thread>). Были реализованы последовательные и параллельные версии программ с использованием перечисленных технологий для решения задач: вычисление числа Пи, умножение матрицы на вектор и сортировка массива. Было замерено время их работы и на основе полученных результатов проведено сравнение технологий по следующим критериям: простота использования, простота освоения технологии и эффективность выполнения поставленной задачи. На основе результатов сравнения технологий можно сделать следующие выводы:

* Все ранее рассмотренные технологии решают поставленные задачи практически за одинаковый промежуток времени.
* Как и предполагалось, последовательная версия работает быстрее при малом количестве данных, но медленно при большом, по сравнению с параллельной.
* Так как все технологии показали практически одинаковое время исполнения рассмотренных задач, то в рамках постановки задачи лучшей технологией является та, которая проста в использовании и изучении. Согласно результатам сравнения технологий по двум критериям (таблица 1) лучшими технологиями оказались OpenMP и использование библиотеки <thread>. OpenMP обладает большим количеством функциональных возможностей, нежели библиотека <thread>, поэтому предпочтительнее и эффективнее будет пользоваться технологией OpenMP.

# 8. Литература

1. Функция CreateThread

<https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682453(v=vs.85).aspx>, апрель 2016.

1. Функция WaitForMultipleObjects

[https://msdn.microsoft.com/ruru/library/windows/desktop/ms687025(v=vs.85).aspx](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms687025(v=vs.85).aspx) ,апрель 2016.

1. Функция CloseHandle [https://msdn.microsoft.com/ruru/library/windows/desktop/ms724211(v=vs.85).aspx](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms724211(v=vs.85).aspx), апрель 2016.
2. Литвиненко А.Н. Технология программирования на C++. Win32 API-приложения: учебное пособие. СПб.: БХВ-Петербург, 2010.
3. **Параллельное программирование с использованием OpenMP** <http://www.hpcc.unn.ru/?doc=489>, апрель 2016.
4. Энтони Уильямс. Параллельное программирование на C++ в действии. ДМК пресс: 2012.
5. Кнут Д, Искусство программирования, том 3. Сортировка и поиск, 2-е издание. -М. : Издательский дома "Вильямс", 2000.
6. Описание алгоритма сортировки чет-нечетной http://www.intuit.ru/studies/courses/1156/190/lecture/4958?page=2, май 2016.