МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий

(Наименование института полностью)

Кафедра Прикладная математика и информатика

(наименование кафедры/департамента/центра полностью)

09.03.03 Прикладная информатика

(код и наименование направления подготовки, специальности)

Корпоративные информационные системы

(направленность(профиль) / специализация)

**КУРСОВАЯ РАБОТА**

по дисциплине (учебному курсу)

(наименование дисциплины (учебного курса))

На тему Многопоточная реализация игры "Жизнь"

Группа ПИб-1903а

Студент Ерофеев А.В.

Руководитель

Оценка

Дата

(подпись руководителя)

Тольятти 2021

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий

(наименование института полностью)

Кафедра Прикладная математика и информатика

(наименование кафедры/департамента/центра полностью)

**ЗАДАНИЕ**

**на выполнение курсовой работы**

Студент Ерофеев Александр Владимирович

1.Тема Многопоточная реализация игры "Жизнь"

2. Срок сдачи студентом законченной курсовой работы Декабрь 2021 г.

3. Исходные данные к курсовой работе

4. Содержание курсовой работы (перечень подлежащих разработке вопросов, разделов)

Глава 1 Постановка задачи на исследование

1.1 Практическая значимость игры «Жизнь»

1.2 Обзор методов реализации игры «Жизнь»

1.3 Реализация последовательной программы игры «Жизнь»

Глава 2 Проектирование и разработка параллельной программы

2.1 Обзор технологий разработки параллельного программного обеспечения

2.2 Разработка параллельного алгоритма

2.3 Реализация параллельной программы игры «Жизнь» с использованием технологий

OpenMP

2.4 Реализация параллельной программы игры «Жизнь» с использованием технологий C++ thread

Глава 3 Анализ эффективности параллельных алгоритмов и разработанного программного обеспечения

3.1 Теоретическое исследование эффективности алгоритма игры «Жизнь»

3.2 Разработка методики оценки эффективности

3.3 Проведение эксперимента на эффективность и анализ результатов

5. Ориентировочный перечень графического и иллюстративного материала

6. Рекомендуемые учебно-методические материалы

7. Дата выдачи задания «15» сентября 2021г.

Руководитель курсовой работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (И.О.Фамилия)

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 4](#_Toc91235920)

[Глава 1 Постановка задачи на исследование 5](#_Toc91235921)

[1.1 Практическая значимость игры «Жизнь» 5](#_Toc91235922)

[1.2 Обзор методов реализации игры «Жизнь» 5](#_Toc91235923)

[1.3 Реализация последовательной программы игры «Жизнь» 6](#_Toc91235924)

[Глава 2 Проектирование и разработка параллельной программы 13](#_Toc91235925)

[2.1 Обзор технологий разработки параллельного программного обеспечения 13](#_Toc91235926)

[2.2 Разработка параллельного алгоритма 14](#_Toc91235927)

[2.3 Реализация параллельной программы игры «Жизнь» с использованием технологий OpenMP 15](#_Toc91235928)

[2.4 Реализация параллельной программы игры «Жизнь» с использованием технологий C++ thread 18](#_Toc91235929)

[Глава 3 Анализ эффективности параллельных алгоритмов и разработанного программного обеспечения 22](#_Toc91235930)

[3.1 Теоретическое исследование эффективности алгоритма игры «Жизнь» 22](#_Toc91235931)

[3.2 Разработка методики оценки эффективности 23](#_Toc91235932)

[3.3 Проведение эксперимента на эффективность и анализ результатов 23](#_Toc91235933)

[ЗАКЛЮЧЕНИЕ 27](#_Toc91235934)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ И ИСТОЧНИКОВ 28](#_Toc91235935)

[ПРИЛОЖЕНИЕ А 29](#_Toc91235936)

# **ВВЕДЕНИЕ**

В наши дни в программировании достаточно часто можно встретить необходимость вести работу с задачами, требующими большой объем вычислений. В такой ситуации последовательный способ выполнения алгоритма приводит к большим затратам по времени и ресурсам компьютера. Для решения этой проблемы используется такое свойство современных машин, как многопоточность – возможность работать с алгоритмом, разбивая процесс выполнения на несколько потоков, которые будут выполняться параллельно. Данное решение позволяет гораздо эффективнее использовать ресурсы вычислительной машины, и, соответственно, сократить время выполнения программы.

Объектом исследования является многопоточная реализация игры «Жизнь». Предметом исследования в данном случае является проектирование, разработка, и анализ работы многопоточных алгоритмов.

Цель курсовой работы: разработать параллельную реализацию программы клеточного автомата для проведения сравнительного анализа методов разбиения алгоритма на потоки.

Для достижения поставленной цели необходимо выполнить следующие задачи:

* провести анализ предметной области;
* определить методы решения задачи;
* разработать параллельные программы;
* провести анализ эффективности разработанных алгоритмов.

Работа состоит из трех разделов, в первой главе определяется задача на последующее исследование, рассматриваются различные методы ее решения. Во второй главе производится обзор доступных технологий для разработки параллельных алгоритмов, описывается процесс разработки и реализации параллельных программ. В третьей главе описывается процесс анализа эффективности многопоточных алгоритмов.

# **Глава 1 Постановка задачи на исследование**

## 1.1 Практическая значимость игры «Жизнь»

Игра жизнь была создана Джоном Конвеем в 1970 году, и является одним из вариантов клеточного автомата. Несмотря на то, что игра состоит всего из двух достаточно простых правил, она уже более пятидесяти лет привлекает внимание ученых. Игра жизнь и ее модификации внесли влияние на многие точные науки, такие как математика, информатика и физика.

Кроме того, игра «Жизнь» повлияла даже на такие далекие от математики науки, как, например биология и бактериология. Достигнуто это было за счет того, что некоторые версии игры с особыми условиями могли достаточно точно воспроизводить процесс размножения и мутаций бактерий. В физиологии игра тоже нашла свой отклик – рождение и смерть клеток напоминает возникновение и затухание нейронный импульсов.

Во всех описанных случаях в игре приходится взаимодействовать с большим объемом данных, а это значит, что распараллеливание ее алгоритмов позволит увеличить производительность программного обеспечения.

## 1.2 Обзор методов реализации игры «Жизнь»

Действия игры происходят в так называемой «Вселенной» – размеченной на клетки плоскости, которая может быть бесконечна, ограниченной, или замкнутой. Клетка может иметь два состояния – быть либо живой, либо мертвой. Клетка имеет восемь соседей, в зависимости от состояния которых будет определяться состояние уже самой клетки. Состояние клеток на момент начала игры называется первым поколением, на основе которого рассчитываются последующие поколения по определенным правилам:

* если у мертвой клетки есть трое живых соседей, то эта клетка оживает;
* если у живой клетки есть двое или трое живых соседей, то клетка не изменяется. Если соседей больше трех или меньше двух, клетка погибает от перенаселения или от одиночества соответственно.

Игра завершается в случае, если на поле не осталось ни одной живой клетки, либо если при очередной смене поколений ни одна из клеток не меняет своего состояния, либо, если состояние клеток текущего поколения повторит состояние одного из предыдущих поколений.

В компьютерной реализации игровое поле, как правило, на замкнутой плоскости – нижняя граница соединена с верхней, а левая – с правой. Существует несколько вариантов реализации смены поколений:

* последовательный просмотр всех клеток и подсчет для них соседей, определяя состояние каждой клетки;
* составляется отдельный список для просмотра в следующем поколении – в такие списки попадают только те клетки, соседи которых меняют свое состояние. Этот вариант отличается более высокой скоростью расчёта состояния клеток в следующем поколении.

Для последующей разработки был выбран первый вариант – он крайне прост в реализации.

## 1.3 Реализация последовательной программы игры «Жизнь»

Поставим задачу разработать последовательную программу работы игры «Жизнь» для последующего распараллеливания.

Смена поколения будет реализована в классе «cellular\_automaton». Класс будет иметь три приватных поля – двумерные векторы типа char «current\_state\_» и «next\_state\_» – текущее и следующее состояния игрового поля соответственно. Третий двумерный вектор хранит координаты относительного положения соседей отдельной клетки. Векторы для реализации выбраны не случайно, они потокобезопасны, что поможет в дальнейшей разработке параллельных программ. Сам алгоритм смены поколений выглядит следующим образом:

* «current\_state\_» копируется в «next\_state\_»;
* осуществляется поэлементный обход current\_state\_, для каждого элемента ведется подсчет живых соседей;
* элементу массива «next\_state\_», соответствующего текущему элементу «cur­rent\_state\_» выставляется состояние в соответствии с количеством живых соседей по правилам, описанным в пункте 1.2;
* «next\_state\_» копируется в «current\_state\_».

За подсчет и установку состояния отдельной клетки будет отвечать приватный метод «set\_cell\_state», в качестве входных данных принимающий положение клетки в векторе в виде номеров строки и столбца. Его работа отражена в блок-схеме на рисунке 1:

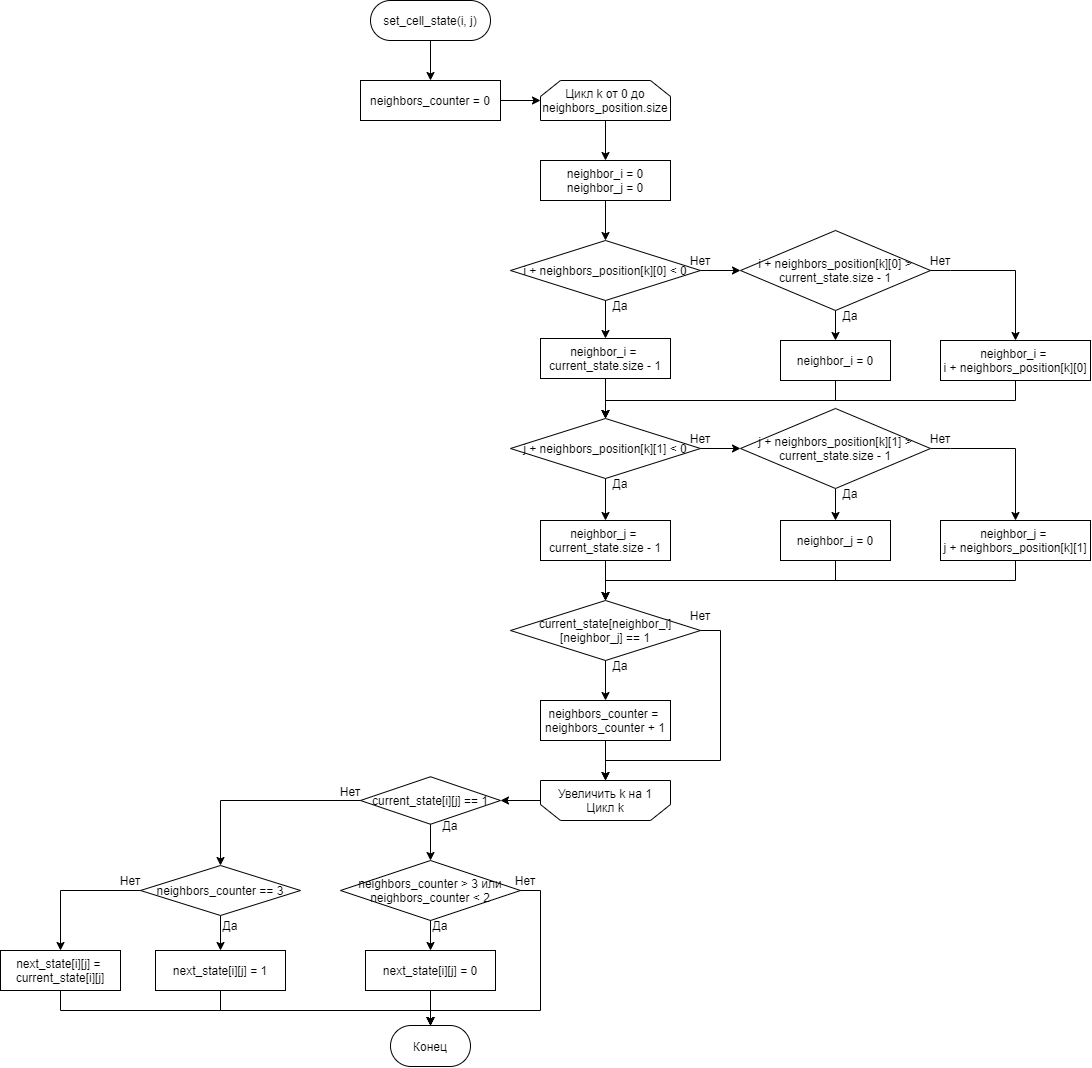


Рисунок 1 – Блок-схема, демонстрирующая работу метода «set\_cell\_state»

Смена поколения реализована в открытом методе «make\_step». В нем осуществляется поэлементный обход двумерного массива «current\_state», где для каждой клетки вызывается метод «set\_cell\_state», описанный ранее. Блок-схема данного метода представлена на рисунке 2:

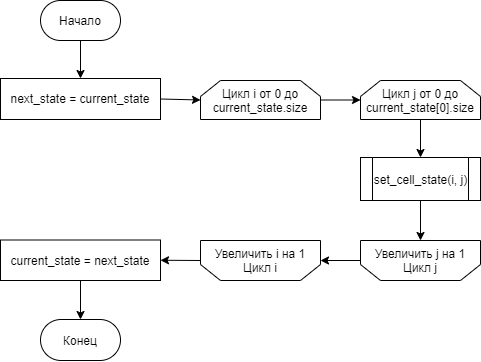


Рисунок 2 – Блок-схема, демонстрирующая работу метода «make\_step»

Помимо этих методов, будет присутствовать перегрузка метода «set\_cell\_state», которая, помимо положения элемента в векторе, принимает еще и аргумент типа bool, на основе которого будет проставлено состояние клетки. Блок-схема этого метода представлена на рисунке 3:

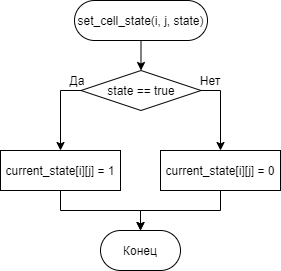


Рисунок 3 – Блок-схема, демонстрирующая работу перегрузки метода «set\_cell\_state»

Еще один метод – «get\_current\_state» – является геттером приватного поля «current\_state», и потребуется при дальнейшем тестировании.

На основе описанной последовательности действий и блок-схемах реализуем класс «cellular\_automaton» на языке C++.

Перейдем к главной функции main. В константах «row\_counter» и «column\_counter» содержится размерность игрового поля. Затем инициализируется объект класса «cellular\_automaton», и случайным образом задается начальная конфигурация игрового поля через метод «set\_initial\_state». Затем, метод «make\_step» выполняется в бесконечном цикле, который прерывается при вводе символа «q». На рисунке 4 представлен листинг функции main:

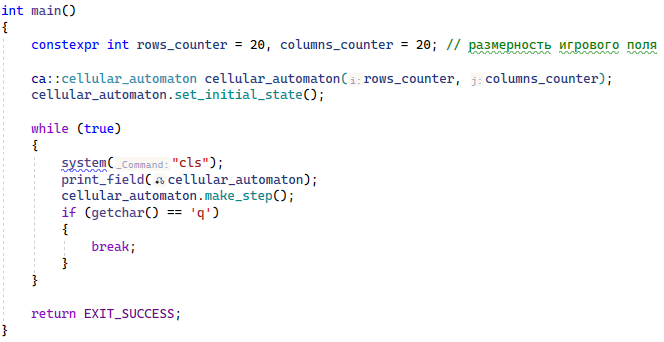


Рисунок 4 – скриншот, демонстрирующий листинг функции main

Полный листинг последовательной программы представлен в приложении А.

Для тестирования программы на правильность выполнения была создана функция «print\_field», который выводит на консоль игровое поле. Алгоритм работы данной функции:

* вызов метода «get\_current\_state» для получения двумерного вектора, содержащего состояние игрового поля на текущей итерации;
* осуществление поэлементного обхода получившегося вектора, если элемент вектора равен 1 (клетка жива), выводится «#», если 0 (клетка мертва) – «\*»;

На рисунке 5 представлен листинг «print\_field»:

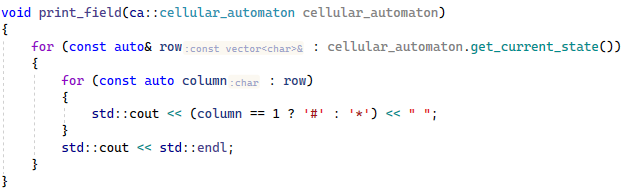


Рисунок 5 – скриншот, демонстрирующий листинг функции «print\_field»

Для проведения теста на поле был добавлен «планер» – фигура с повторяющимся состоянием, движущаяся бесконечно. После достижения четвертого поколения, «планер» переходит в состояние первого поколения, и так по кругу. Размер поля – 10 на 10.

Таблица 1.1 – результаты тестирования последовательной программы

|  |  |  |  |
| --- | --- | --- | --- |
| Шаг № | Результат работы программы | Ожидаемый результат | Правильность результата |
| 1 |  |  | Верно |
| 2 |  |  | Верно |
| 3 |  |  | Верно |
| 4 |  |  | Верно |
| 5 |  |  | Верно |

Как видно из предоставленных результатов, состояние «планера» на них соответствует ожидаемым, на основе чего можно сделать вывод, что программа работает корректно.

Помимо этого, необходимо замерить время выполнения программы. На таблице 1.2 демонстрируются результаты теста:

Таблица 1.2 – результаты замера времени выполнения последовательной программы

|  |  |  |
| --- | --- | --- |
| № теста | Размерность массива | Время выполнения, мс |
| 1 | 100x100 | 5 мс |
| 2 | 500x500 | 122 мс |
| 3 | 1000x1000 | 475 мс |
| 4 | 2000x2000 | 1918 мс |

Таким образом, была разработана и протестирована программа, реализующая смену поколений клеток в игре «Жизнь».

# **Глава 2 Проектирование и разработка параллельной программы**

## 2.1 Обзор технологий разработки параллельного программного обеспечения

На данный момент существует достаточно много технологий, предназначенных для разработки параллельных программ. У всех них есть свои достоинства и недостатки, которые и будут рассмотрены в данном разделе.

Первой рассматриваться будет библиотека OpenMP – стандарт для написания параллельных алгоритмов на таких языках, как C, C++ и Fortran. Данная библиотека реализует параллельные вычисления с помощью многопоточности, в которой ведущий поток создает ведомые, между которыми и распределяется задача.

Достоинства OpenMP:

* последовательное распараллеливание программы. Это позволяет значительно упростить процесс написания алгоритма и его отладку;
* простота использования – нет необходимости переписывать последовательную программу с нуля, достаточно просто добавить OMP-директивы;

Недостатки:

* организация взаимодействия потоков через общие переменные. Такой подход приводит к возникновению трудно обнаружимых ошибок. Кроме того, инструменты для поиска и отладки таких ошибок крайне малочисленны или отсутствуют вовсе.

Следующей будет рассмотрена библиотека, которая входит в стандартный набор библиотек языка C++ – thread.

Перечень достоинств данной библиотеки:

* кроссплатформенность – программа, написанная с использованием данной библиотеки будет запускаться на большинстве существующих машин;
* передача нескольких параметров при инициализации потока;
* активное комьюнити – можно довольно легко найти необходимую информацию о тех или иных аспектах разработки с помощью данной библиотеки.

Каких-либо серьезных недостатков у данной технологии нет, разве что можно выделить относительную сложность в применении по сравнению с тем же OpenMP.

Еще один инструмент для разработки параллельных программ – MPI (The Message Passing Interface). Это хорошо стандартизированный механизм для построения параллельных программ в модели обмена сообщениями.

Плюсы применения MPI:

* кроссплатформенность;
* позволяет создавать хорошо масштабируемые параллельные программы.

Минусы:

* затруднительность проведения отладки из-за одновременного исполнения нескольких программных ветвей.

Для проведения анализа эффективности было решено выбрать библиотеки OpenMP и threads. Первая была выбрана за счет удобства применения в алгоритмах, где необходимо распараллеливать циклы и общую простоту. Вторая входит в стандартный набор библиотек C++, а значит, не требует никаких дополнительных настроек, и обладает достаточным количеством документации.

## 2.2 Разработка параллельного алгоритма

Программа, разработанная в пункте 1.3 корректно выполняет свою работу, но все еще имеет ряд недостатков. Алгоритм строится на поэлементном обходе двумерного массива, а это значит, что чем больше размер массива, тем больше времени будет затрачено. В данном случае имеется возможность снизить нагрузку на вычислительную машину и увеличить скорость выполнения программы, прибегнув к распараллеливанию процесса обработки массива.

Алгоритм распараллеливания для данной задачи достаточно прост:

* разделить массив состояний на части, количество которых равняется числу задействованных потоков;
* запустить каждую часть на обработку в отдельный поток.

Для хранения значений в массиве состояния была был выбран такой тип данных, как вектор. Вектор является потокобезопасным, что облегчит процесс дальнейшей разработки параллельных программ. С элементами вектора можно безопасно вести одновременную работу с нескольких потоков.

В данном случае нет смысла использовать блокировки потоков, поскольку, следуя описанному ранее алгоритму, не предполагается одновременный доступ нескольких потоков к одному и тому же элементу вектора.

## 2.3 Реализация параллельной программы игры «Жизнь» с использованием технологий OpenMP

В случае реализации параллельной программы с использованием технологии OpenMP нет необходимости переписывать код последовательной программы с нуля, достаточно добавить в определенные места необходимые директивы. Для этого будет создан новый метод «make\_step\_omp», к который будет скопировано содержимое метода «make\_step» класса «cellular\_automa-ton». Перед каждым из вложенных циклов будет добавлена строка «#pragma omp parallel for», указывающая, что надо разделить данный цикл по итерациям между потоками. Внесенные изменения отображены на рисунке 8:

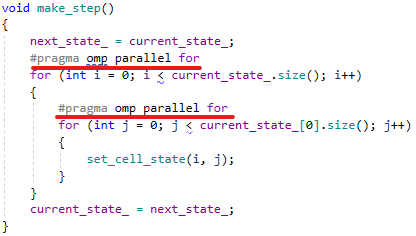


Рисунок 6 – скриншот, демонстрирующий изменения, внесенные в метод «make\_step»

Перейдем к тестированию программы. Условия тестирования будут соответствовать тем, что использовались при тестировании последовательной программы в пункте 1.3, но с одним отличием – будет использоваться другая фигура под названием «легкий космический корабль», тоже с повторяющимся состоянием через каждые 4 поколения. Результаты тестирования представлены на таблице 2.1:

Таблица 2.1 – результаты тестирования параллельной программы, написанной с использованием технологии OpenMP

|  |  |  |  |
| --- | --- | --- | --- |
| Шаг № | Результат работы программы | Ожидаемый результат | Правильность результата |
| 1 |  |  | Верно |
| 2 |  |  | Верно |
| 3 |  |  | Верно |
| 4 |  |  | Верно |
| 5 |  |  | Верно |

Помимо этого, необходимо замерить ускорение параллельной программы по отношению у последовательной. Скорость выполнения последовательной программы взята из таблицы 1.3. Результаты замером представлены на таблице 2.2.

Таблица 2.2 – результаты замера ускорения параллельной программы по отношению к последовательной

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № теста | Размерность массива | Время выполнения (последовательная программа), мс | Время выполнения (параллельная программа), мс | Ускорение, мс |
| 1 | 100x100 | 5 мс | 1 мс | 4 мс |
| 2 | 500x500 | 122 мс | 20 мс | 102 мс |
| 3 | 1000x1000 | 475 мс | 104 мс | 371 мс |
| 4 | 2000x2000 | 1918 мс | 358 мс | 1560 мс |

На основе полученных результатов, можно сделать вывод, что программа работает корректно.

## 2.4 Реализация параллельной программы игры «Жизнь» с использованием технологий C++ thread

В отличие от реализации на OpenMP, преобразование последовательного алгоритма в параллельный с использованием C++ thread потребует несколько более крупных изменений.

Первым шагом в класс «cellular\_automaton» будут добавлены два приватных поля – «threads\_» и «part\_». Первое поле является вектором, содержащим элементы типа thread, количество которых будет задаваться через главный конструктор. Фактически, количество элементов этого вектора будет соответствовать количеству потоков, которые будут задействованы при выполнении программы. Назначение поля «part\_» будет описано чуть позже.

Следующим шагом содержимое метода «make\_step» будет скопировано в метод «make\_step\_threads», где и будет изменено для работы с C++ thread. Проход по двумерному вектору будет вынесен в отдельный вспомогательный метод, который будет называться «thread\_make\_step». Метод будет принимать два параметра целочисленного типа – «from» и «to». Эти параметры отражают промежуток строк, который будет обрабатывать поток, в котором будет вызван этот метод.

Вернемся к методу «make\_step». Через цикл будет проходить инициализация и запуск потоков. Каждый поток будет запускать метод «thread\_make\_step», и передавать ему границы, в которых данный поток будет работать с массивом состояний «current\_state\_». Тут и необходимо поле «part\_». Инициализируется оно следующим образом: количество строк в массиве состояний делится на количество задействованных потоков, и приводится к целочисленному типу. Нижняя граница разбиения массива «from» формируется посредством умножения счетчика «i» на поле «part\_». Верхняя граница вычисляется чуть сложнее – если «i» равно разности размера вектора «threads\_» и единицы, то верхняя граница будет равна размеру «current\_state\_». Если это условие не удовлетворяется, то верхняя граница будет равна сумме «from» и «part\_». Скриншот, на котором представлен фрагмент описанного выше кода, можно увидеть на рисунке 11:

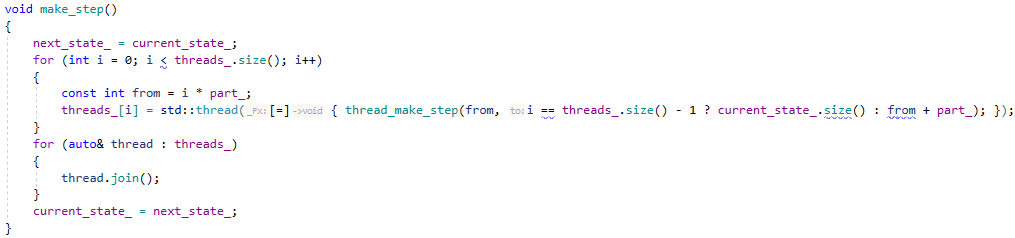


Рисунок 7 – скриншот, демонстрирующий метод «make\_step», использующий технологию thread

После того, как все вычисления будут выполнены, потоки завершаются в отдельном цикле через команду join.

Далее перейдем к тестированию программы. Условия теста соответствуют условиям теста последовательной программы, но используется другая фигура – «средний космический корабль». Результаты тестирования представлены на таблице 2.3:

Таблица 2.3 – результаты тестирования последовательной программы

|  |  |  |  |
| --- | --- | --- | --- |
| Шаг № | Результат работы программы | Ожидаемый результат | Правильность результата |
| 1 |  |  | Верно |
| 2 |  |  | Верно |
| 3 |  |  | Верно |
| 4 |  |  | Верно |
| 5 |  |  | Верно |

Так же, необходимо провести замеры ускорения выполнения, чтобы убедиться, что разбиение на потоки работает правильно. Количество потоков, используемое в данном тесте, максимальное, что доступно на компьютере, на котором проводился тест – двенадцать. Скорость выполнения последовательного алгоритма взята из таблицы 1.2.

Таблица 2.4 – результаты замера ускорения параллельной программы по отношению к последовательной

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № теста | Размерность массива | Время выполнения последовательной программы, мс | Время выполнения параллельной программы, мс | Ускорение, мс |
| 1 | 100x100 | 5 мс | 2 мс | 3 мс |
| 2 | 500x500 | 122 мс | 35 мс | 87 мс |
| 3 | 1000x1000 | 475 мс | 118 мс | 357 мс |
| 4 | 2000x2000 | 1918 мс | 366 мс | 1152 мс |

На основе проведенного тестирования можно сделать вывод, что программа работает корректно.

# **Глава 3 Анализ эффективности параллельных алгоритмов и разработанного программного обеспечения**

## 3.1 Теоретическое исследование эффективности алгоритма игры «Жизнь»

Использование параллельных алгоритмов в определенных случаях способствует более эффективному распределению ресурсов вычислительной машины и ускорению работы программы. Но, порой, можно встретить ситуации, в которых программа, реализованная с использованием параллелизма, будет работать хуже, чем ее последовательный вариант. Во избежание подобного сначала обычно проводится теоретическое исследование эффективности алгоритма.

Поскольку процесс распараллеливания с использованием технологий OpenMP и C++ Threads достаточно схож, будет верно утверждать, что результаты последующих вычислений будут в равной степени справедливы для обоих методов.

Сложность алгоритма смены поколения в игре «Жизнь», согласно источникам O(n2), составляет F1(n) = n2. Таким образом, время выполнения будет равно T1(n) = n2\* t, t –время выполнения одной операции. Количество выполняемых операций для алгоритма не изменится, поскольку разработанные алгоритмы не увеличивают или изменяют последовательность операций.

В разработанном алгоритме вычислительные операции распределяются между потоками, из чего можно сделать вывод, что время выполнения параллельного алгоритма можно описать следующей формулой: Tp(n)= (n2 \* t) / p*.*

Вычисления коэффициентов ускорения и эффективности:

1. Rp(n) = T1(n) / Tp(n) = (n2\* t) / ((n2 \* t) / p) = p;
2. Ep(n)=Rp(n)/p = p / p = 1.

Из результата вычислений можно сделать вывод, что коэффициент ускорения должен быть равен числу потоков. В теории, данный алгоритм распараллеливания должен быть очень эффективен.

## 3.2 Разработка методики оценки эффективности

В данном разделе будет проведено исследование, задачей которого является оценка эффективности алгоритма смены поколений в игре «Жизнь».

Для проведения оценки эффективности будут использованы три программы, введем их обозначения:

* последовательная программа смены поколений в игре «Жизнь» (0);
* параллельная программа смены поколений в игре «Жизнь» с использованием технологии C++ threads (1);
* параллельная программа смены поколений в игре «Жизнь» с использованием технологии OpenMP (2).

Эксперименты проведены на стационарном компьютере с процессором AMD Ryzen 5 частота которого составляет 3,4 ГГц, 6 ядер, 12 потоков. Операционная система – Windows 11.

Варьируемый параметр – число потоков p варьируется от 2 до 12 с шагом 2.

Постоянные параметры – начальная конфигурация поля, сгенерированная в начале выполнения программы и размерность массива состояний, которая равна 4000000 элементов.

Измеряемая величина – время выполнения одного шага T (секунд).

Расчётные величины:

* коэффициентs ускорения параллельных вычислений R1 = T0/T1, R2 = T0/T2;
* коэффициент эффективности распараллеливания E1 = R1/p, E2 = R2/p.

Таким образом, были разработаны методики оценки эффективности

## 3.3 Проведение эксперимента на эффективность и анализ результатов

Пример вывода программ при проведении эксперимента представлен на рисунке 12:

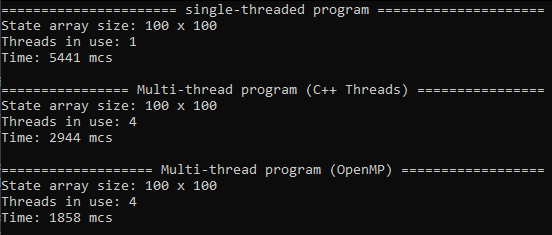


Рисунок 8 – скриншот, демонстрирующий пример вывода программы при тестировании

На таблице 3.1 представлены результаты времени тестирования параллельных программ. Помимо этого, в таблицах так же указаны значения коэффициентов ускорения и эффективности параллельных вычислений, формулы расчёта которых представлены в пункте 3.2.

Таблица 3.1 – результаты тестирования параллельных алгоритмов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во потоков | Обозначение программы | Время выполнения, с | Коэффициент ускорения, R | Коэффициент эффективности, E |
| 1 | 0 | 2,209629 | - | - |
| 2 | 1 | 1,069045 | 2,0669 | 1,0335 |
| 2 | 1,054157 | 2,0961 | 1,0481 |
| 4 | 1 | 0,611266 | 3,6148 | 0,9037 |
| 2 | 0,595953 | 3,7077 | 0,9269 |
| 6 | 1 | 0,480657 | 4,5971 | 0,7662 |
| 2 | 0,495365 | 4,4606 | 0,7434 |
| 8 | 1 | 0,431124 | 5,1253 | 0,0539 |
| 2 | 0,425039 | 5,1987 | 0,0531 |
| 10 | 1 | 0,347418 | 6,3601 | 0,6360 |
| 2 | 0,346149 | 6,3835 | 0,6383 |
| 12 | 1 | 0,357647 | 6,1782 | 0,0298 |
| 2 | 0,346717 | 6,3730 | 0,0289 |

Зависимость времени выполнения от количества потоков показано на рисунке 9:

****

Рисунок 9 – график, отображающий зависимость времени выполнения от кол-ва потоков

Зависимость коэффициента ускорения от числа потоков показана на рисунке 10:



Рисунок 10 – график, отображающий зависимость коэффициента ускорения от числа потоков

Зависимость коэффициента эффективности от числа потоков показана на рисунке 11:



Рисунок 11 – график, отображающий зависимость коэффициента эффективности от числа потоков

Проанализировав полученные данные, можно сделать вывод, что параллельная реализация превосходит последовательную в зависимости от числа потоков в 2-6 раз. Кроме того, на практике результаты оказались выше, чем рассчитывалось, ведь предполагалось, что коэффициент ускорения будет примерно равен количеству потоков.

Если сравнивать многопоточные программы, то небольшой выигрыш по скорости наблюдался у программы (2) по отношению к программе (1), но, выигрыш настолько небольшой, что можно считать его погрешностью.

# **ЗАКЛЮЧЕНИЕ**

Целью курсовой работы была разработка параллельных алгоритмов для проведения анализа эффективности, и, впоследствии, выбор более удобного и эффективного.

Для исследования были выбраны две технологии распараллеливания программ – OpenMP и C++ Threads. Были разработаны и протестированы последовательная, и две параллельные программы. Так же, была рассчитана теоретическая эффективность алгоритма.

Распараллеливание последовательного алгоритма смены поколений в игре «Жизнь» показало себя крайне эффективно, позволив ускорить выполнение программы в 2-6 раз в зависимости от количества потоков.

Помимо прочего стало ясно, что нет особой разницы в применении распараллеливания с помощью технологий OpenMP и C++ Threads. Пускай OMP и был более выигрышным по времени, но этот выигрыш был настолько мал, что мог считаться погрешностью. В итоге, оба метода показали себя достаточно эффективно.

Таким образом, можно утверждать, что поставленная цель курсовой работы достигнута, обозначенные задачи выполнены.

# **СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ И ИСТОЧНИКОВ**

1. Гантерот, К. Оптимизация программ на C++. Проверенные методы повышения производительности. Руководство / К. Гантерот : пер. с англ. – Москва : «Диалектика-Вильямс», 2019. – 400 с. : – ISBN 978-5-907144-58-3. – Текст : непосредственный.
2. Уильямс, Э. Параллельное программирование на С++ в действии. Практика разработки многопоточных программ / Э. Уильямс : пер. с англ. – Москва : «ДМК Пресс», 2012. – 672 с. : – ISBN 978-5-94074-448-1. – Текст : непосредственный.
3. Антонов, А. Технологии параллельного программирования MPI и OpenMP / А. Антонов – Москва : «Издательство МГУ», 2012. – 344 с. : ISBN 978-5-211-06343-3. – Текст : непосредственный.
4. Федотов, И. Параллельное программирование. Модели и приемы / И. Федотов – Москва : «Солон-Пресс», 2017. – 390 с. : ISBN 978-5-91359-222-4. – Текст : непосредственный.
5. Васильев, А. Программирование на C++ в примерах и задачах / А. Васильев – Москва : «Эксмо», 2021. – 368 с. – ISBN 978-5-699-87445-3. – Текст : непосредственный.
6. Posch, M. Mastering C++ Multithreading. Write robust, concurrent, and parallel applications / M. Posch – «Packt Publishing», 2017. – 244 p. – ISBN 9781787121706 (eng.)
7. Meyers, S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 / S. Meyers – « O'Reilly Media, Incorporated», 2014. – 334 p. – ISBN 9781491903995 (eng.)

# **ПРИЛОЖЕНИЕ А**

Полный листинг cellular\_automaton.h:

#include <vector>

#include <thread>

namespace ca

{

class cellular\_automaton

{

public:

cellular\_automaton(int, int, int);

cellular\_automaton(const cellular\_automaton&);

void make\_step();

void make\_step\_omp();

void make\_step\_threads();

std::vector<std::vector<char>> get\_current\_state();

void set\_initial\_state();

private:

std::vector<std::vector<char>> current\_state\_;

std::vector<std::vector<char>> next\_state\_;

std::vector<std::vector<char>> neighbors\_position\_ = {

{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}

};

std::vector<std::thread> threads\_;

const int part\_;

void set\_cell\_state(int, int);

};

}

Полный листинг cellular\_automaton.cpp:

#include "cellular\_automaton.h"

#include <random>

#include <omp.h>

ca::cellular\_automaton::cellular\_automaton(const int row\_size,

const int column\_size,

const int threads\_counter)

: current\_state\_(

std::vector<std::vector<char>>(row\_size,

std::vector<char>(column\_size, 0))),

next\_state\_(

std::vector<std::vector<char>>(row\_size,

std::vector<char>(column\_size, 0))),

threads\_(threads\_counter),

part\_(static\_cast<int>(current\_state\_.size() / threads\_.size()))

{

omp\_set\_num\_threads(threads\_counter);

}

ca::cellular\_automaton::cellular\_automaton(const ca::cellular\_automaton& cellular\_automaton)

: current\_state\_(cellular\_automaton.current\_state\_),

next\_state\_(cellular\_automaton.next\_state\_),

threads\_(cellular\_automaton.threads\_.size()), part\_(cellular\_automaton.part\_)

{ }

void ca::cellular\_automaton::make\_step()

{

next\_state\_ = current\_state\_;

for (int i = 0; i < static\_cast<int>(current\_state\_.size()); i++)

{

for (int j = 0; j < static\_cast<int>(current\_state\_[0].size()); j++)

{

set\_cell\_state(i, j);

}

}

current\_state\_ = next\_state\_;

}

void ca::cellular\_automaton::make\_step\_omp()

{

next\_state\_ = current\_state\_;

#pragma omp parallel for

for (int i = 0; i < static\_cast<int>(current\_state\_.size()); i++)

{

#pragma omp parallel for

for (int j = 0; j < static\_cast<int>(current\_state\_[0].size()); j++)

{

set\_cell\_state(i, j);

}

}

current\_state\_ = next\_state\_;

}

void ca::cellular\_automaton::make\_step\_threads()

{

next\_state\_ = current\_state\_;

for (size\_t t = 0; t < threads\_.size(); t++)

{

const int from = static\_cast<int>(t) \* part\_;

const int to = t == threads\_.size() - 1

? static\_cast<int>(current\_state\_.size())

: from + part\_;

threads\_[t] = std::thread([=]

{

for (int i = from; i < to; i++)

{

for (int j = 0; j < static\_cast<int>(current\_state\_[0].size()); j++)

{

set\_cell\_state(i, j);

}

}

});

}

for (auto& thread : threads\_)

{

thread.join();

}

current\_state\_ = next\_state\_;

}

std::vector<std::vector<char>> ca::cellular\_automaton::get\_current\_state()

{

return current\_state\_;

}

void ca::cellular\_automaton::set\_initial\_state()

{

std::random\_device device;

std::mt19937 generator(device());

const std::uniform\_int\_distribution<int> distribution(0, 100);

for (auto& row : current\_state\_)

{

for (size\_t i = 0; i < current\_state\_[0].size(); i++)

{

row[i] = distribution(generator) % 2 ? 1 : 0;

}

}

}

void ca::cellular\_automaton::set\_cell\_state(const int row, const int column)

{

int neighbors\_counter = 0;

for (auto& position : neighbors\_position\_)

{

if (current\_state\_[row + position[0] < 0 ? current\_state\_.size() - 1 :

row + position[0] > static\_cast<int>(current\_state\_.size()) - 1 ? 0 : row + position[0]]

[column + position[1] < 0 ? static\_cast<int>(current\_state\_[0].size()) - 1 :

column + position[1] > static\_cast<int>(current\_state\_[0].size()) - 1 ? 0 : column + position[1]] == 1)

{

neighbors\_counter++;

}

}

next\_state\_[row][column] = current\_state\_[row][column] == 1 && (neighbors\_counter > 3 || neighbors\_counter < 2) ? 0 :

neighbors\_counter == 3 ? 1 : current\_state\_[row][column];

}

Полный листинг main.cpp:

#include <iostream>

#include "cellular\_automaton.h"

void print\_field(ca::cellular\_automaton& cellular\_automaton)

{

for (const auto& row : cellular\_automaton.get\_current\_state())

{

for (const auto column : row)

{

std::cout << (column == 1 ? '#' : '\*') << " ";

}

std::cout << std::endl;

}

}

int main()

{

constexpr int row\_size = 20, column\_size = 20;

ca::cellular\_automaton cellular\_automaton(row\_size, column\_size, 8);

cellular\_automaton.set\_initial\_state();

while (true)

{

system("cls");

print\_field(cellular\_automaton);

cellular\_automaton.make\_step();

if (getchar() == 'q')

{

break;

}

}

return EXIT\_SUCCESS;

}