

Программное обеспечение, использованное при выполнении: Visual Studio 2017, IPS 2019, Windows 10 – 64x

Процессор – четырехъядерный Intel Core i5 8250U с частотой 1.6 ГГц

1. Ознакомьтесь со статьей [The non-uniform covering approach to manipulator workspace assessment.pdf](#).

2. Скачайте следующие файлы: [box.h](#), [box.cpp](#), [fragmentation.h](#), [fragmentation.cpp](#), [NUCovering.cpp](#). В этих файлах представлен предлагаемый каркас разрабатываемого проекта. Ознакомьтесь с содержимым каждого файла. После выполнения **п.1**. Вашей задачей является написание определений тех функций проекта, в теле которых представлен комментарий *"// необходимо определить функцию"*.

Определение функции **VerticalSplitter**

```
double x_min, y_min, width, height, newleft1, newtop1, newwidth1, newheight1, newleft2,
newtop2, newwidth2, newheight2;
    box.GetParameters(x_min, y_min, width, height);

    newleft1 = x_min;
    newtop1 = y_min;
    newwidth1 = width / 2.0;
    newheight1 = height;
    Box leftbox(newleft1, newtop1, newwidth1, newheight1);

    newleft2 = x_min + width / 2.0;
    newtop2 = y_min;
    newwidth2 = width / 2.0;
    newheight2 = height;
    Box rightbox(newleft2, newtop2, newwidth2, newheight2);
    vertical_splitter_pair.first = leftbox;
    vertical_splitter_pair.second = rightbox;
```

Определение функции **HorizontalSplitter** выглядит аналогично.

Определение функции **GetNewBoxes**:

```
double width, height;
    box.GetWidthHeight(width, height);

    if (abs(width) > abs(height))
        VerticalSplitter(box, new_pair_of_boxes);
    else
        HorizontalSplitter(box, new_pair_of_boxes);
```

Определение функции **ClasifyBox**:

```
int count = 0;

    for (int i = 0; i < vects.second.size(); i++)
    {
        if (vects.second[i] < 0)
            count += 1;
        if (vects.first[i] > 0)
            return 1; // not solution -> return 1
    }

    if (count == vects.second.size())
```

```

        return 0; // solution -> return 0

    if (vects.first[0] == 0 && vects.second[0] == 0)
        return 2; // boundary -> return 2

    return 3; // new boxes -> return 3

```

Определение функции **GetBoxType**:

```

min_max_vectors min_max_vecs;
boxes_pair new_pair_of_boxes;

GetMinMax(box, min_max_vecs);
int res = ClasifyBox(min_max_vecs);

switch (res)
{
    case 0: {solution->push_back(box); break; } // solution
    case 1: {not_solution->push_back(box); break; } // not solution
    case 2: {boundary->push_back(box); break; } // boundary
    case 3:
    {
        GetNewBoxes(box, new_pair_of_boxes); // new boxes
        temporary_boxes->push_back(new_pair_of_boxes.first);
        temporary_boxes->push_back(new_pair_of_boxes.second);
        break;
    }
}

```

Определение функции **GetSolution**:

```

current_box = Box(-g_l1_max, 0, g_l2_max + g_l0 + g_l1_max, __min(g_l1_max, g_l2_max));
std::vector<Box> current_boxes;
temporary_boxes->push_back(current_box);

int level = FindTreeDepth();
for (int i = 0; i < (level + 1); ++i)
{
    temporary_boxes.move_out(buf_boxes);
    current_boxes = buf_boxes;
    buf_boxes.clear();
    for(int j = 0; j < current_boxes.size(); ++j)
        GetBoxType(current_boxes[j]);
}

```

Определение функции **WriteResults**:

```

double x_min, y_min, width, height;
vector <Box> temp;

std::ofstream fsolution(file_names[0]); // создаём объект класса ofstream для
записи и связываем его с файлом solution.txt
std::ofstream fboundary(file_names[1]);
std::ofstream fnot_solution(file_names[2]);

solution.move_out(temp);
for (int i = 0; i < temp.size(); i++)
{
    temp[i].GetParameters(x_min, y_min, width, height);
}

```

```


        fsolution << x_min << " " << y_min << " " << width << " " << height <<
'\n';
    }

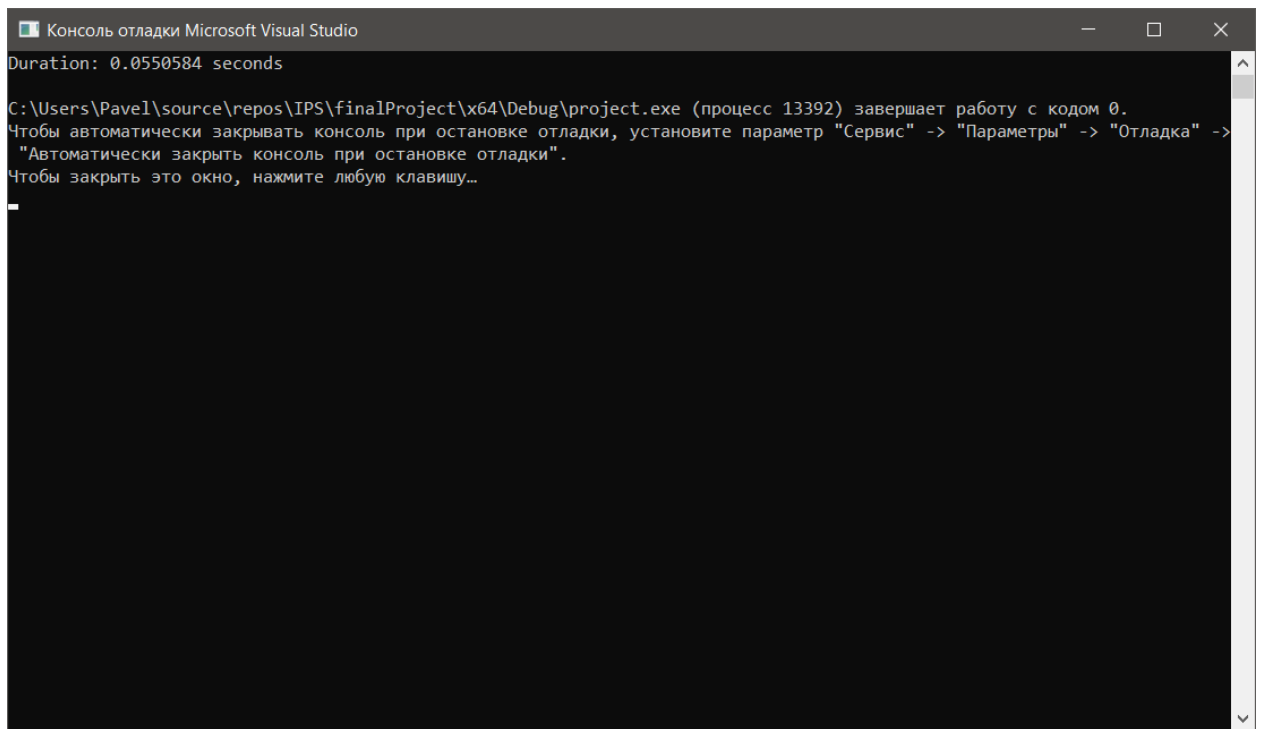
    temp.clear(); // очищаем временный вектор
    boundary.move_out(temp);
    for (int i = 0; i < temp.size(); i++)
    {
        temp[i].GetParameters(x_min, y_min, width, height);
        fboundary << x_min << " " << y_min << " " << width << " " << height <<
'\n';
    }

    temp.clear(); // очищаем временный вектор
    not_solution.move_out(temp);
    for (int i = 0; i < temp.size(); i++)
    {
        temp[i].GetParameters(x_min, y_min, width, height);
        fnot_solution << x_min << " " << y_min << " " << width << " " << height
<< '\n';
    }

    fsolution.close();
    fboundary.close();
    fnot_solution.close();

```

3. Реализация последовательной версии программы, определяющей рабочее пространство планарного робота, по предложенному в статье из **п.1.** алгоритму. Функция **WriteResults()** должна записывать значения параметров box-ов в выходные файлы в следующем порядке: **x_min, y_min, width, height, '\n'**. На выходе из программы должно получиться 3 файла. Определите время работы последовательной версии разработанной программы в двух режимах: **Debug** и **Release**. Сделайте скрины консоли, где отображается время работы для обоих случаев. Вставьте скрины в отчет к проекту, дав им соответствующие названия. Постройте полученное рабочее пространство, используя скрипт **MATLAB PrintWorkspace.m** . Сохраните изображение рабочего пространства. Вставьте его в отчет, назвав соответствующим образом.

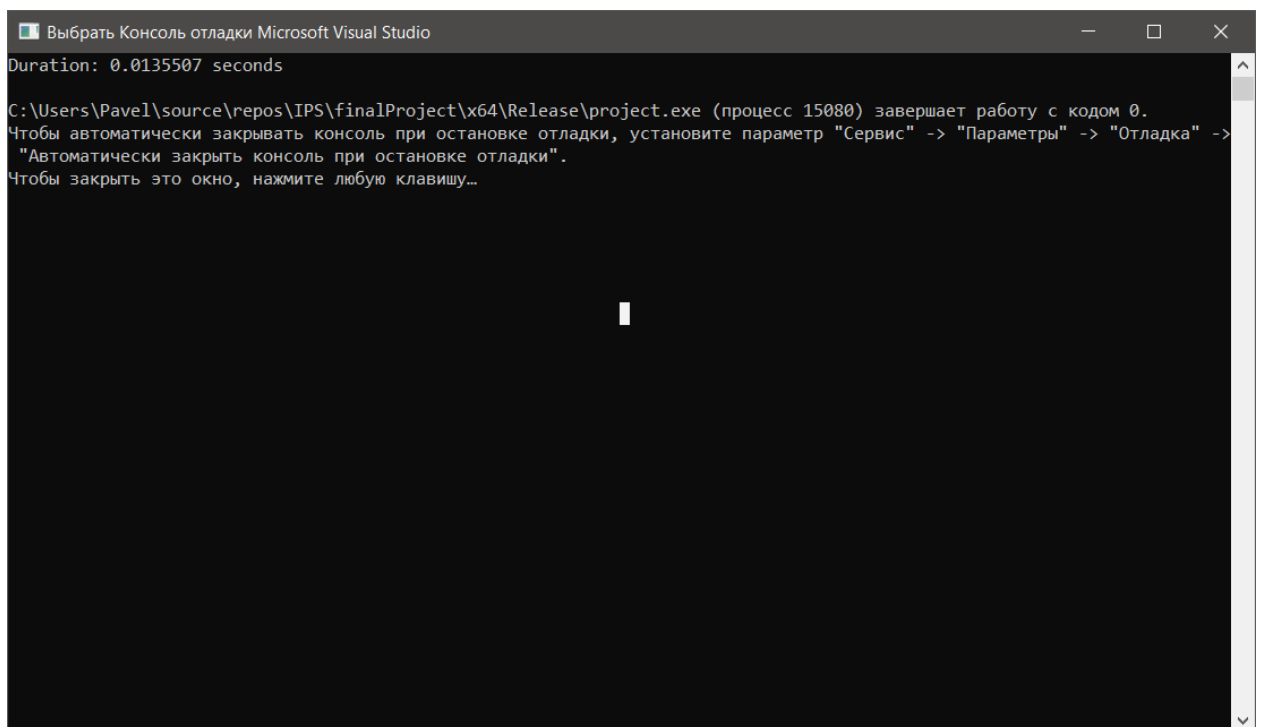


Консоль отладки Microsoft Visual Studio

Duration: 0.0550584 seconds

C:\Users\Pavel\source\repos\IPS\finalProject\x64\Debug\project.exe (процесс 13392) завершает работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, установите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Чтобы закрыть это окно, нажмите любую клавишу...

Рисунок 1 – Пример запуска программы на версии Debug x64



Выбрать Консоль отладки Microsoft Visual Studio

Duration: 0.0135507 seconds

C:\Users\Pavel\source\repos\IPS\finalProject\x64\Release\project.exe (процесс 15080) завершает работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, установите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Чтобы закрыть это окно, нажмите любую клавишу...

Рисунок 2 – Пример запуска программы на версии Release x64

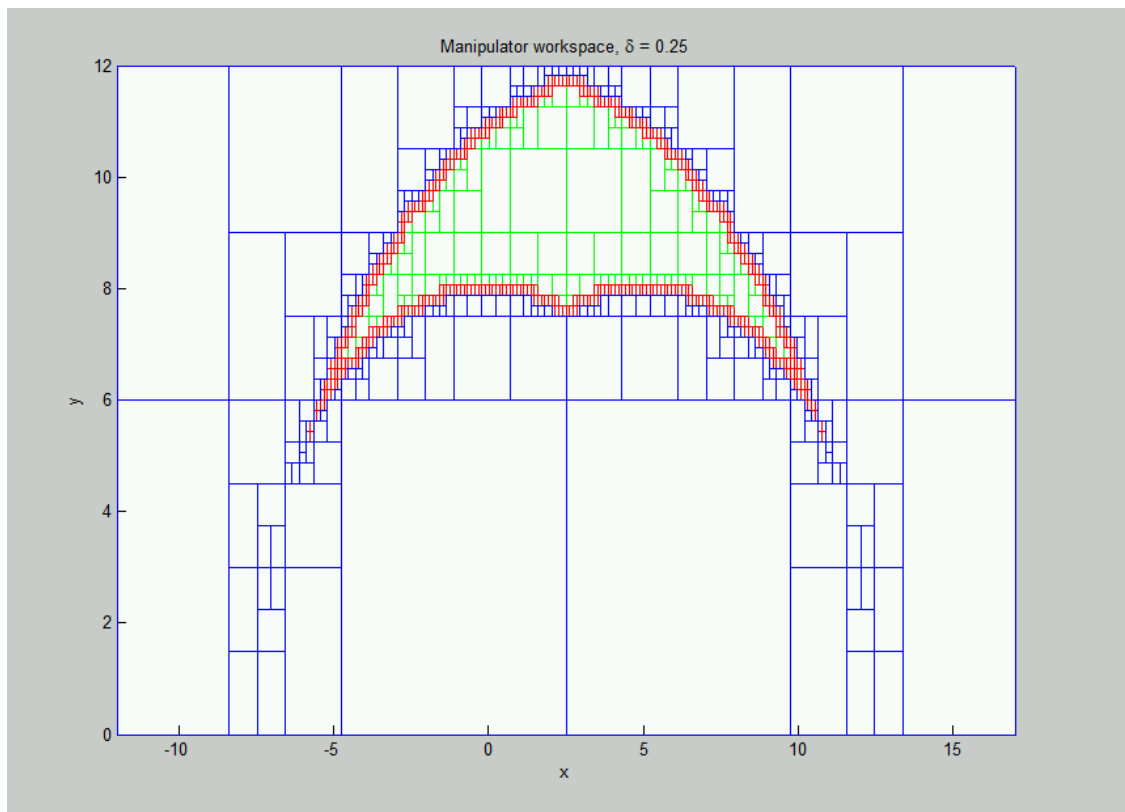


Рисунок 3 – Полученное рабочее пространство при $\delta = 0.25$

4. Использование **Amplifier XE** в целях определения наиболее часто используемых участков кода. Для этого закомментируйте строки кода, отвечающие за запись результатов в выходные файлы, выберите **New Analysis** из меню **Amplifier XE** на панели инструментов, укажите тип анализа **Basic Hotspots**, запустите анализ. Сделайте скрин окна результатов анализа и вкладки **Bottom-up**. В списке, представленном в разделе **Top Hotspots** вкладки **Summary** должна фигурировать функция **GetMinMax()**.

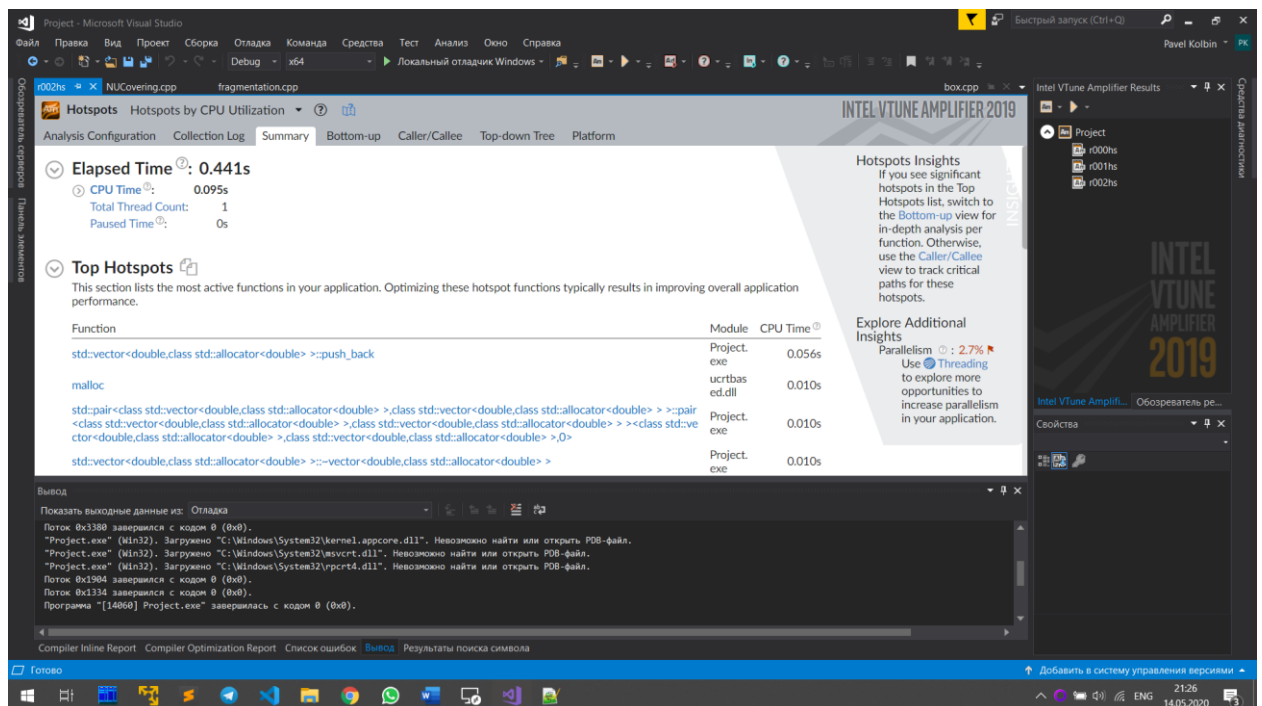


Рисунок 4 – результаты Amplifier XE

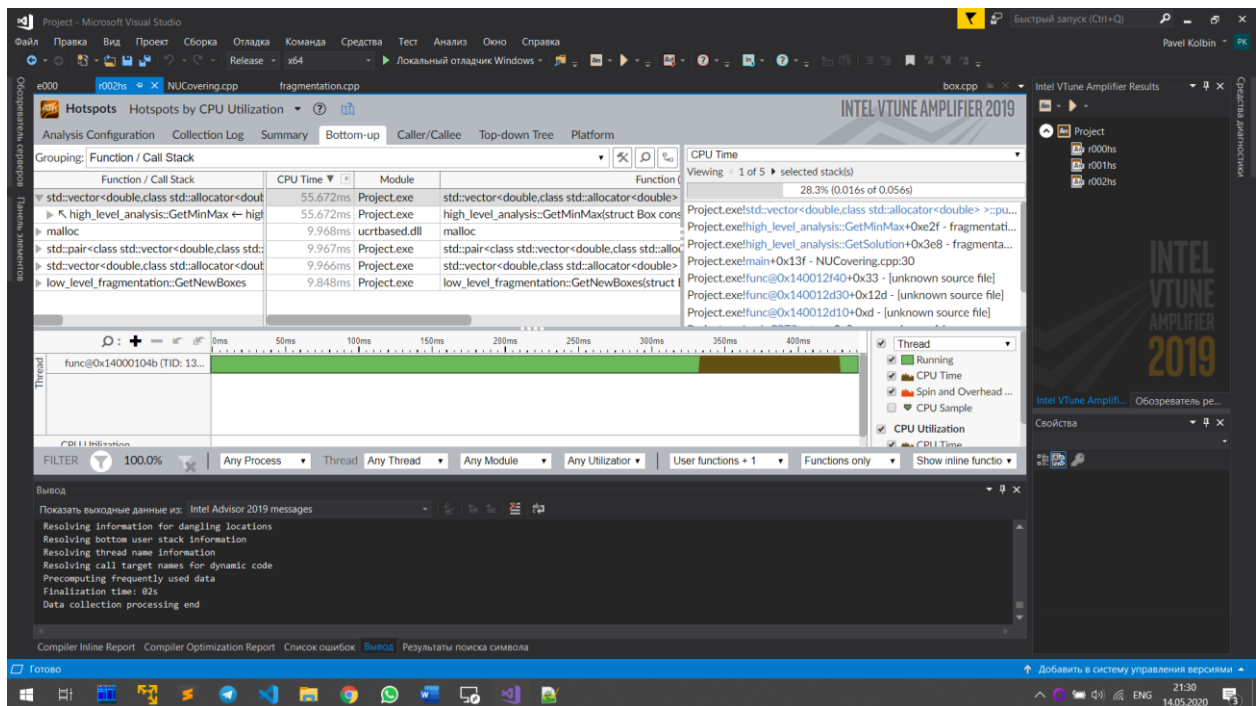


Рисунок 5 – результаты Amplifier XE вкладка Bottom-Up

Как видно действительно, наибольшее время затрачивает функция **GetMinMax()**

5. Использование **Parallel Advisor** с целью определения участков кода, которые требуют наибольшего времени исполнения. Переведите проект в режим **Release** и отключите всякую оптимизацию. Для этого следует выбрать свойства проекта, во вкладке **C/C++** перейти в раздел **Оптимизация**, в пункте меню **“Оптимизация”** выбрать **Отключено (/Od)**. Далее выберем **Parallel Advisor** на панели инструментов **Visual Studio** и запустим **Survey Analysis**. По окончании анализа Вы должны увидеть, что наибольшее время затрачивается в цикле функции **GetSolution()**, двойным кликом по данной строке отчета можно перейти к участку исходного кода и увидеть, что имеется в виду цикл, в котором на каждой итерации вызывается функция **GetBoxType()**. Сделайте скрины результатов **Survey Analysis**, сохраните их, добавьте в отчет. Вернитесь в режим **Debug**.

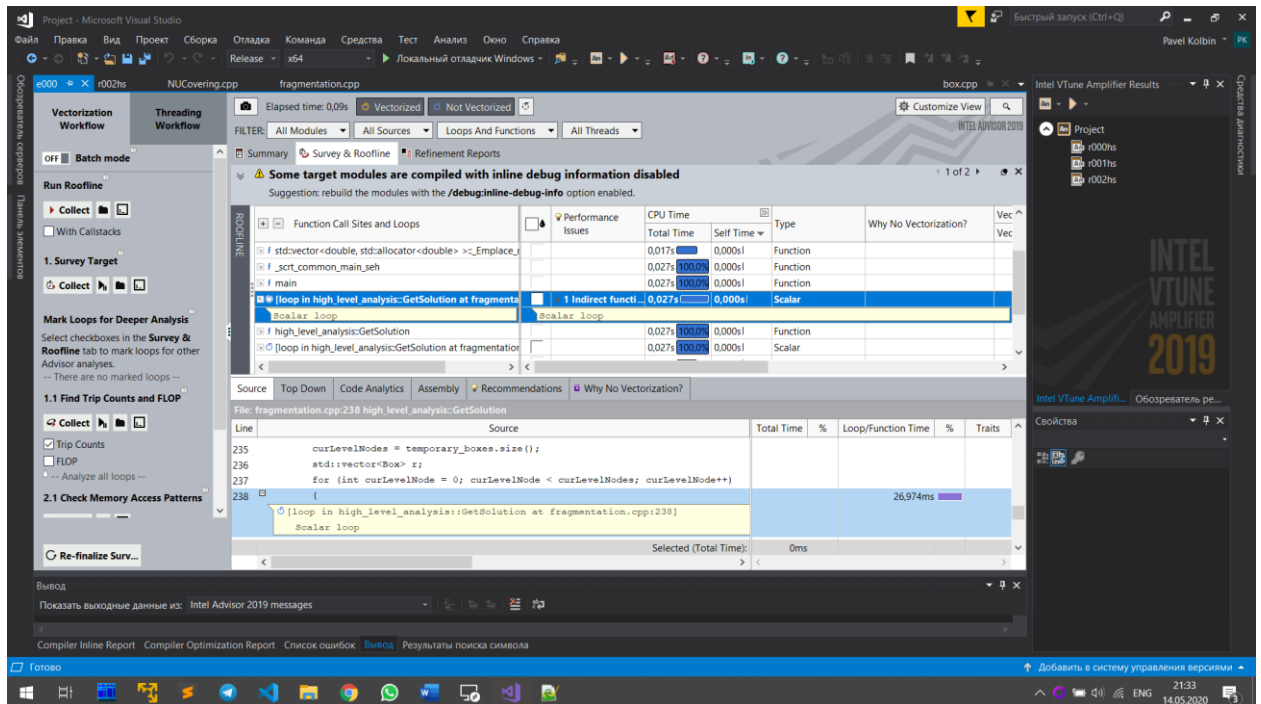


Рисунок 6 – Результат работы Survey Analysis

Как видно, действительно наибольшее время затрачивается в функции **GetSolution** в цикле for.

6. Введение параллелизма в программу. В текущей (последовательной) реализации программы, в функции **GetSolution()** должны фигурировать два вложенных цикла. Внешний цикл проходит по всем уровням двоичного дерева разбиения. В рамках внутреннего цикла происходит перебор всех box-ов текущего уровня разбиения и определение типа box-а (является он частью рабочего пространства либо не является, лежит он на границе или подлежит дальнейшему анализу). Вам необходимо ввести параллелизм во внутренний цикл. Тогда следует подумать о возможности независимого обращения к векторам **solution**, **not_solution**, **boundary**, **temporary_boxes**. Для этого предлагается использовать **reducer** векторы **Intel Cilk Plus**, вместо обычных **std::vector** ов.

Добавим reducer

```

// вектор, содержащий box-ы, являющиеся частью рабочего пространства
cilk::reducer< cilk::op_vector<Box> > solution;
// вектор, содержащий box-ы, не являющиеся частью рабочего пространства
cilk::reducer< cilk::op_vector<Box> > not_solution;
// вектор, содержащий box-ы, находящиеся на границе между "рабочим" и "нерабочим"
пространством
cilk::reducer< cilk::op_vector<Box> > boundary;
// вектор, хранящий box-ы, анализируемые на следующей итерации алгоритма
cilk::reducer< cilk::op_vector<Box> > temporary_boxes;

```

И также распараллелим внутренний цикл функции **GetSolution**

```

cilk_for(int j = 0; j < current_boxes.size(); ++j)
    GetBoxType(current_boxes[j]);

```

7. Определение ошибок после введения параллелизации. Запустите анализы Inspector **XE: Memory Error Analysis** и **Threading Error Analysis** на различных уровнях

(**Narrowest, Medium, Widest**). Приложите к отчету скрины результатов запуска перечисленных анализов. Исправьте обнаруженные ошибки, приложите новые скрины результатов анализов, в которых ошибки отсутствуют. *Примечание:* "глюки" **Intel Cilk Plus** исправлять не нужно.

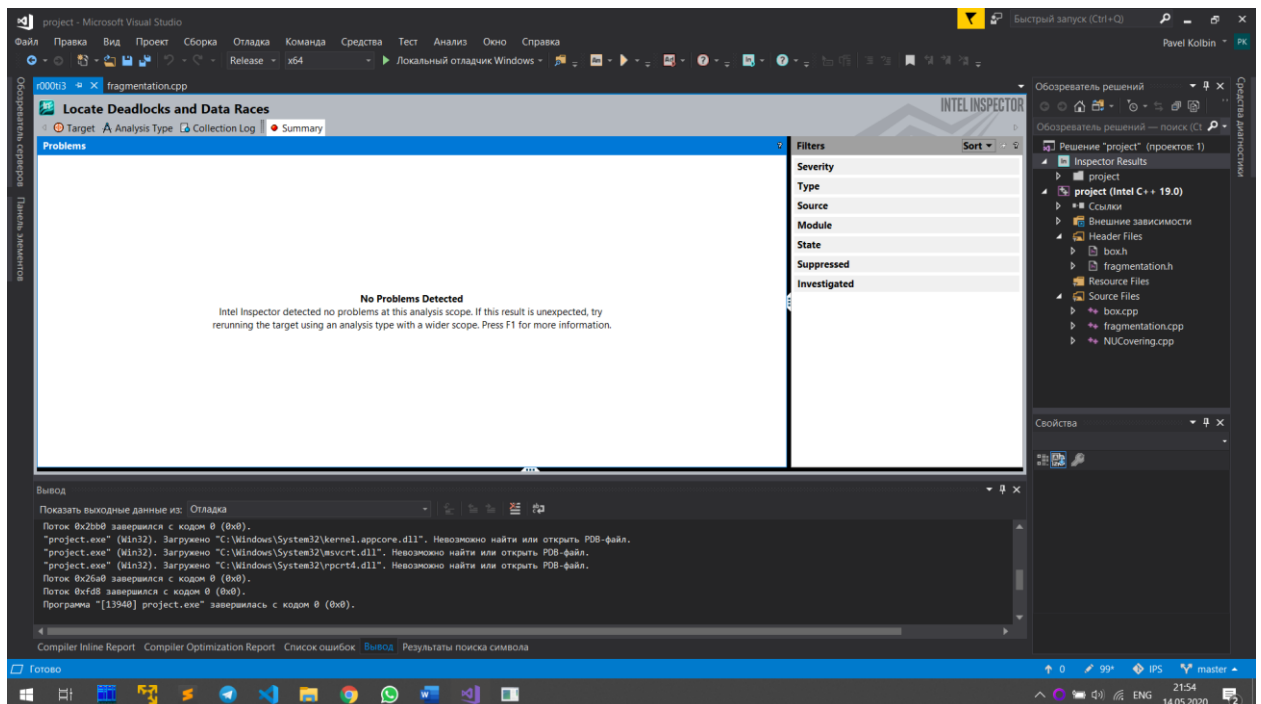


Рисунок 7 – Результаты работы Threading Error Analysis

Как видно гонок данных нет, значит используем reducer правильно

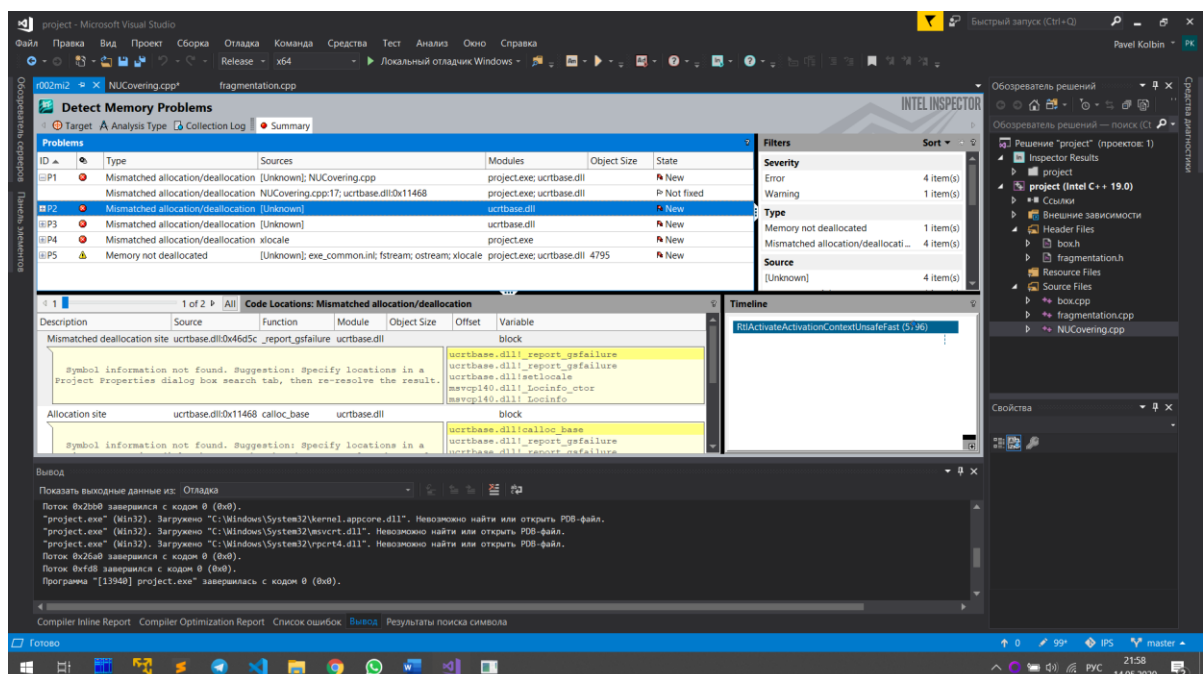


Рисунок 8 – Результаты работы Memory Error Analysis

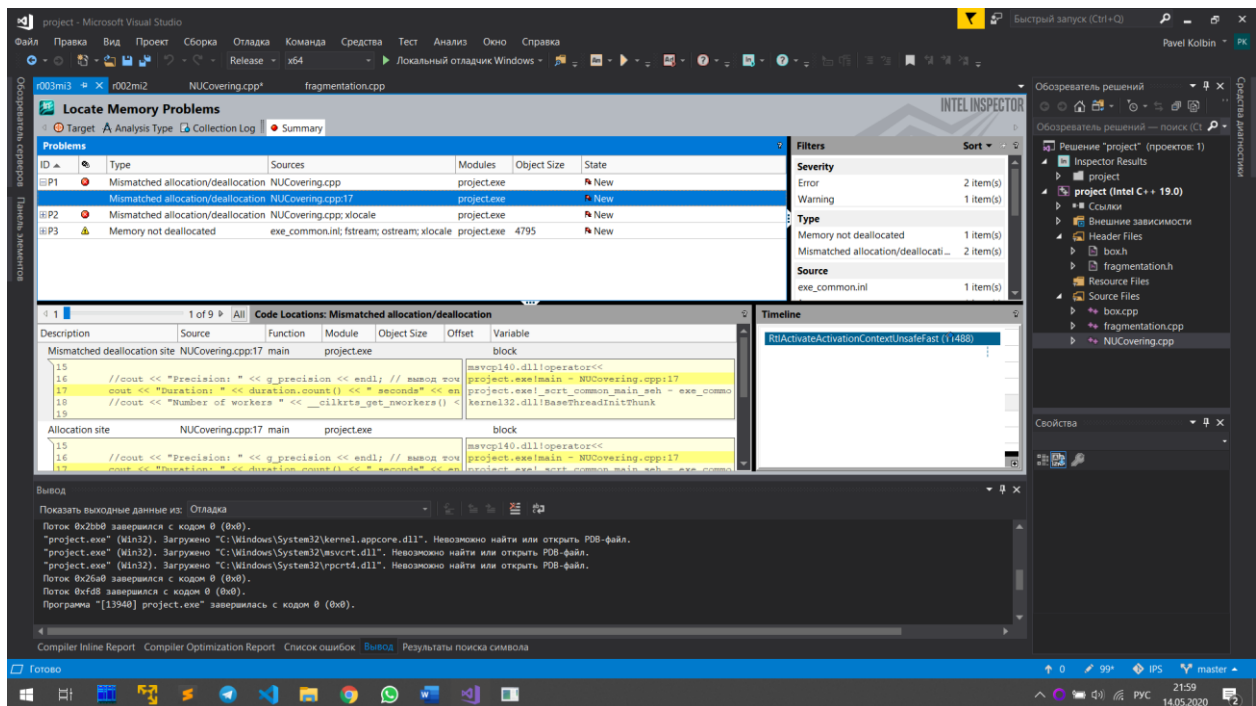


Рисунок 9 – Результаты работы Memory Error Analysis

Как видно на скриншотах выше – серьезных ошибок нет.

8. Работа с **Cilk API**. По умолчанию параллельная программа, использующая Cilk запускается на количестве потоков равных количеству ядер вашего компьютера. Для управления количеством вычислителей необходимо добавить заголовочных файл `#include <cilk/cilk_api.h>` и действовать следующим образом: в исполняемом файле `NUCovering.cpp` перед созданием объекта **main_object** класса **high_level_analysis** необходимо вставить следующие строки кода: `__cilkrts_end_cilk(); __cilkrts_set_param("nworkers", "X");` Здесь **X** - отвечает за количество вычислителей, на которых будет запускаться исходная программа. Это число может быть от 1 до **N**, где **N** - количество ядер в Вашей системе. Изменяя **X**, запускайте программу и фиксируйте время ее выполнения, каждый раз сохраняйте скрины консоли, где должно быть отображено количество вычислителей (`cout << "Number of workers" << __cilkrts_get_nworkers() << endl;`) и время работы программы.

Тестировать будем на версии программы Release x64. Зависимость времени выполнения приведена в таблице ниже.

Количество потоков	Время выполнения, с
1	0.0155468
2	0.0116153
3	0.0108832
4	0.0102423

Как видно прирост есть, но он почти незаметен, особенно в случае 3 и 4 ядер.

9. Визуализация полученного решения. Поэкспериментируйте со входными параметрами программы и отобразите несколько версий полученного рабочего пространство робота. Рисунки приложите к отчету

Построим полученное рабочее пространство при разных значения delta.

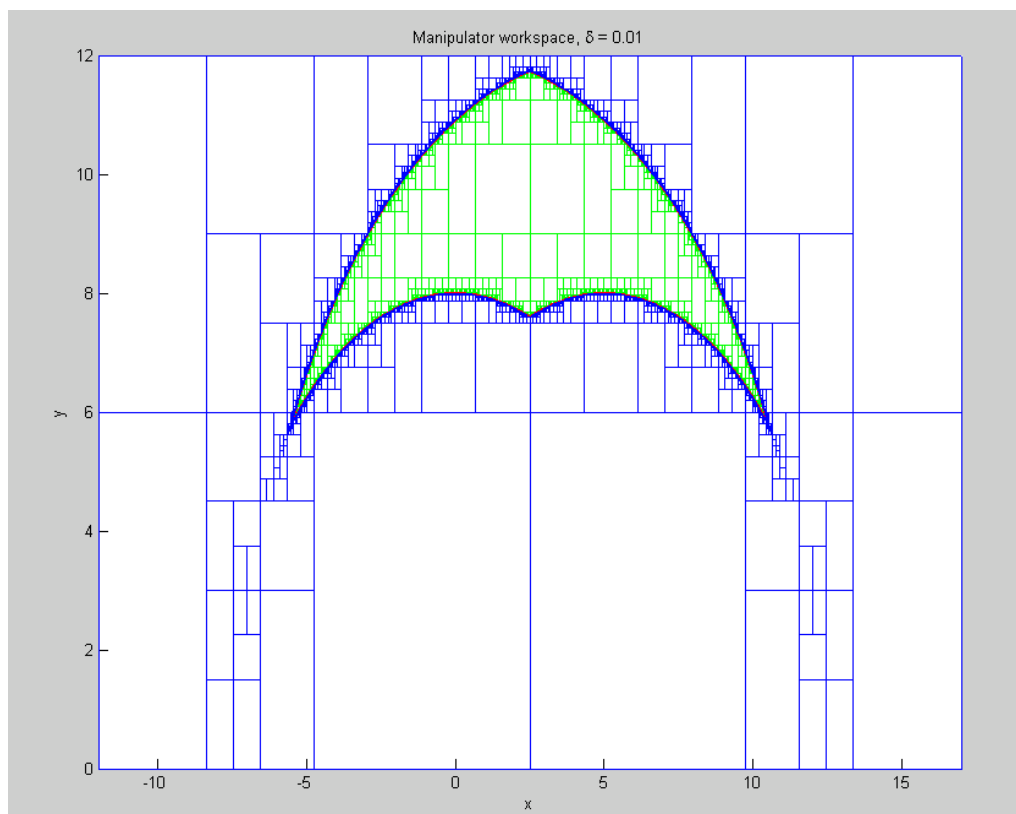


Рисунок 10 – Полученное рабочее пространство при $\delta = 0.01$

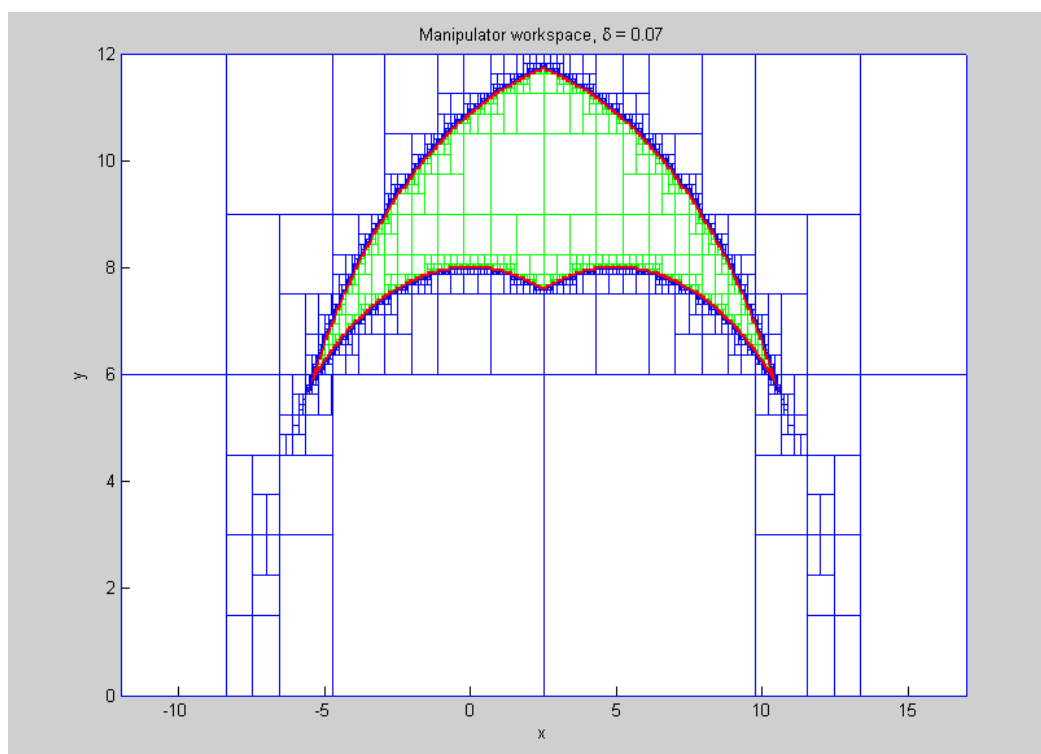


Рисунок 11 – Полученное рабочее пространство при $\delta = 0.07$

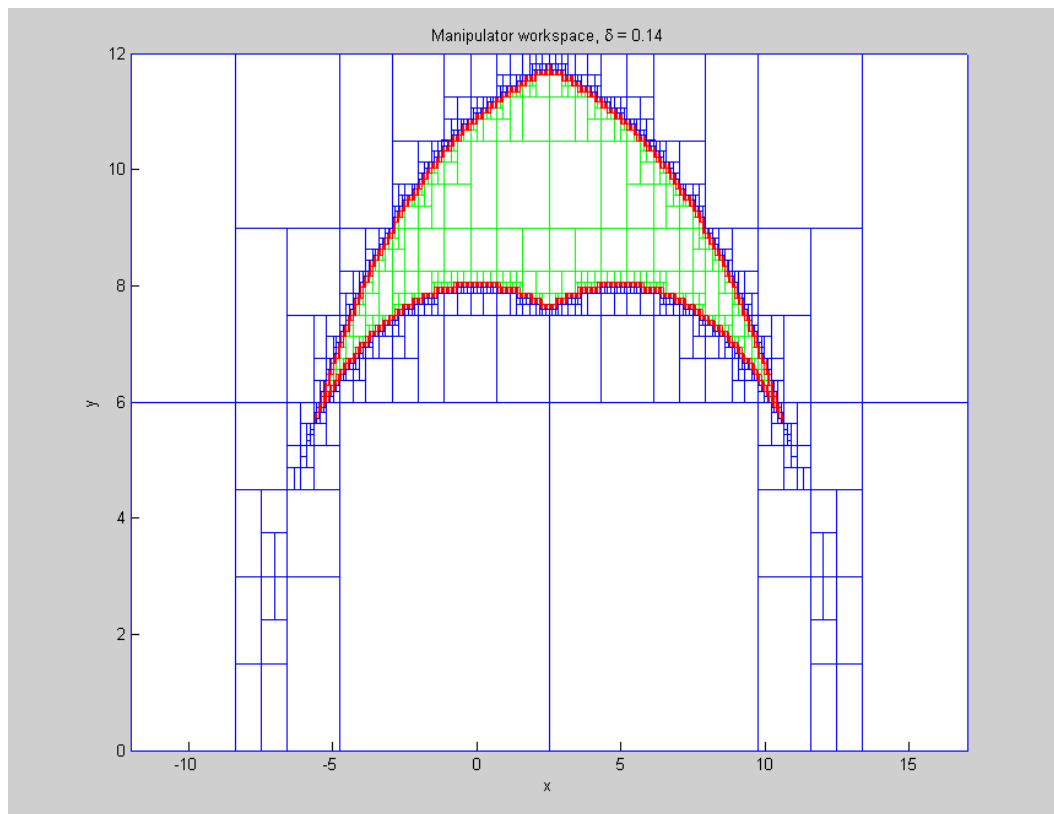


Рисунок 12 – Полученное рабочее пространство при $\delta = 0.14$

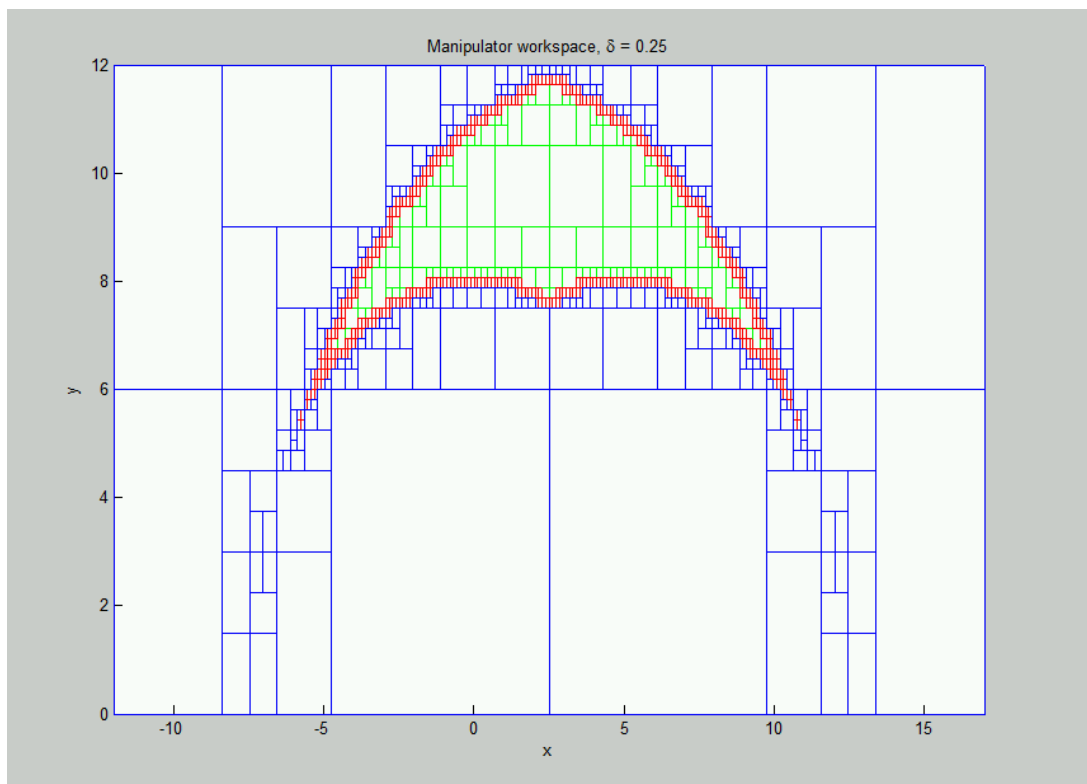


Рисунок 13 – Полученное рабочее пространство при $\delta = 0.25$

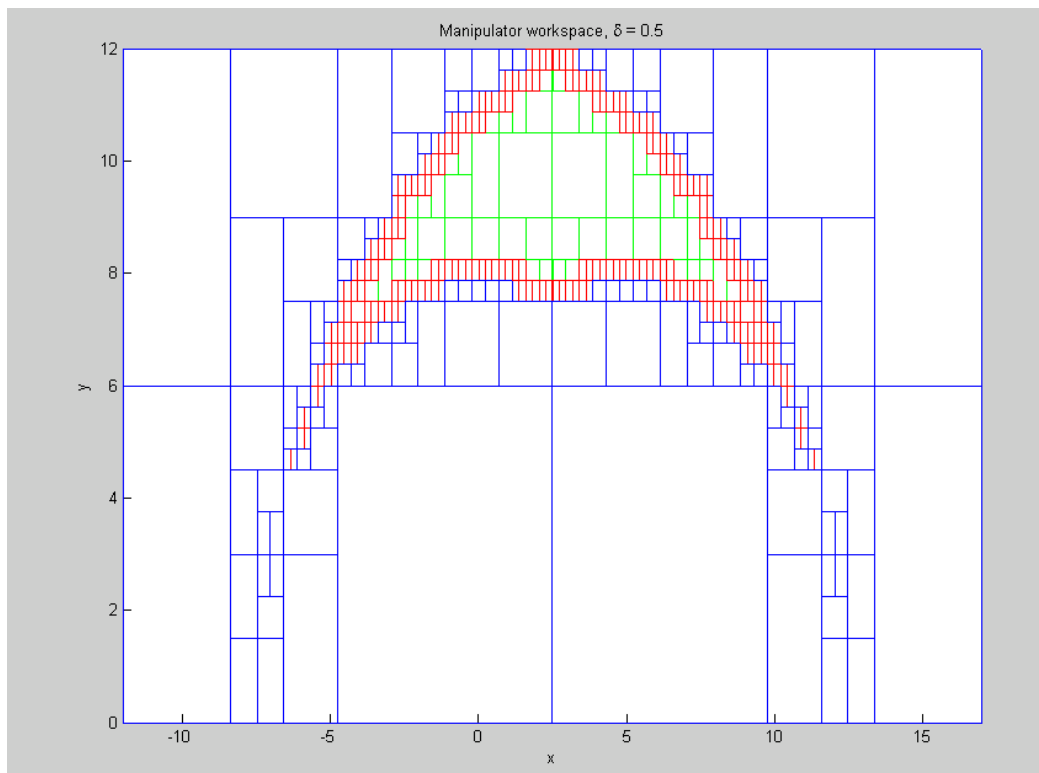


Рисунок 14 – Полученное рабочее пространство при $\delta = 0.5$

Результаты зависимости δ и времени работы программы приведены в таблице ниже

Количество потоков	δ	Время выполнения, с
4	0.01	0.105832
4	0.07	0.0260641
4	0.14	0.0190976
4	0.25	0.0137692
4	0.5	0.0106267

Вывод: в данном проекте были реализованы функции необходимые для реализации планарного робота, проверена работоспособность проекта на различных версиях Debug, Release, измерено время работы программы. Также было построено полученное рабочее пространство с помощью Matlab. Были использованы программы Parallel Advisor и Amplifier XE для выявления самых время затратных мест в программе, чтобы снизить время выполнения программы, был введен параллелизм. Программа была проанализирована на наличие утечек памяти и гонок данных, время выполнения программы уменьшилось. Затем был проведен анализ времени выполнения программы в зависимости от количества задействованных ядер, а также был проведен анализ зависимости времени выполнения программы от δ . Чем меньше дельта, тем дольше работает программа.