**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.pdfПредварительный просмотр документа](https://canvas.instructure.com/courses/1844563/files/87817255/download?wrap=1).

2. Скачайте следующие файлы: [box.hПредварительный просмотр документа](https://canvas.instructure.com/courses/1844563/files/87817219/download?wrap=1), [box.cppПредварительный просмотр документа](https://canvas.instructure.com/courses/1844563/files/87817236/download?wrap=1), [fragmentation.h](https://canvas.instructure.com/courses/1844563/files/87817256/download?wrap=1" \o "fragmentation.h)[Предварительный просмотр документа](https://canvas.instructure.com/courses/1844563/files/87817256/download?wrap=1), [fragmentation.cppПредварительный просмотр документа](https://canvas.instructure.com/courses/1844563/files/87817228/download?wrap=1), [NUCovering.cppПредварительный просмотр документа](https://canvas.instructure.com/courses/1844563/files/87817257/download?wrap=1). 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.GetWidhtHeight(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;  }  } |

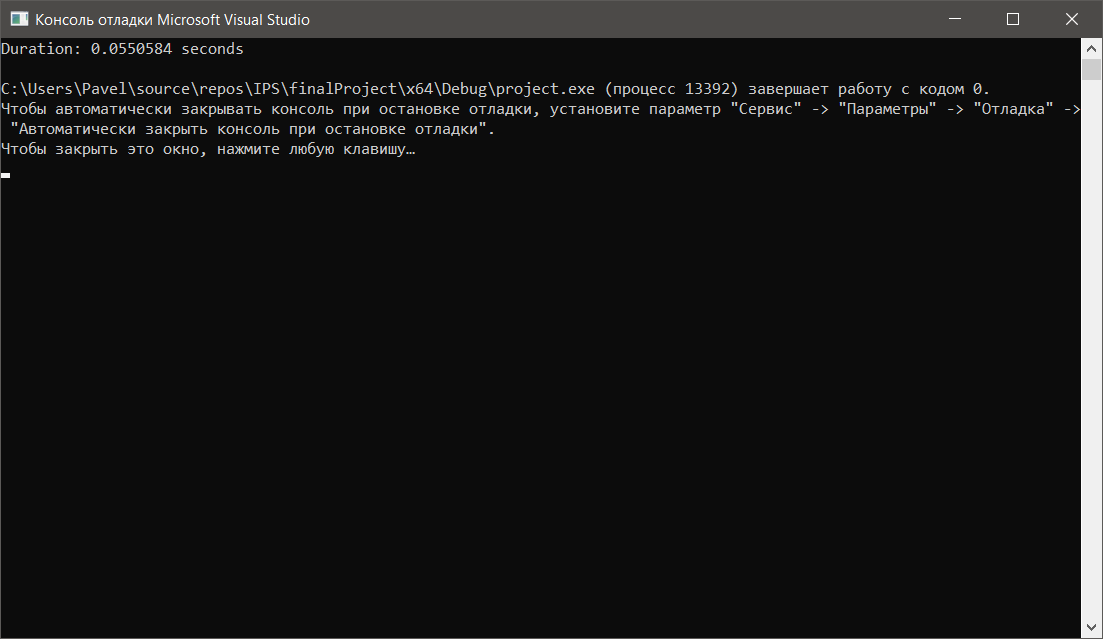
Определение функции **GetSolution**:

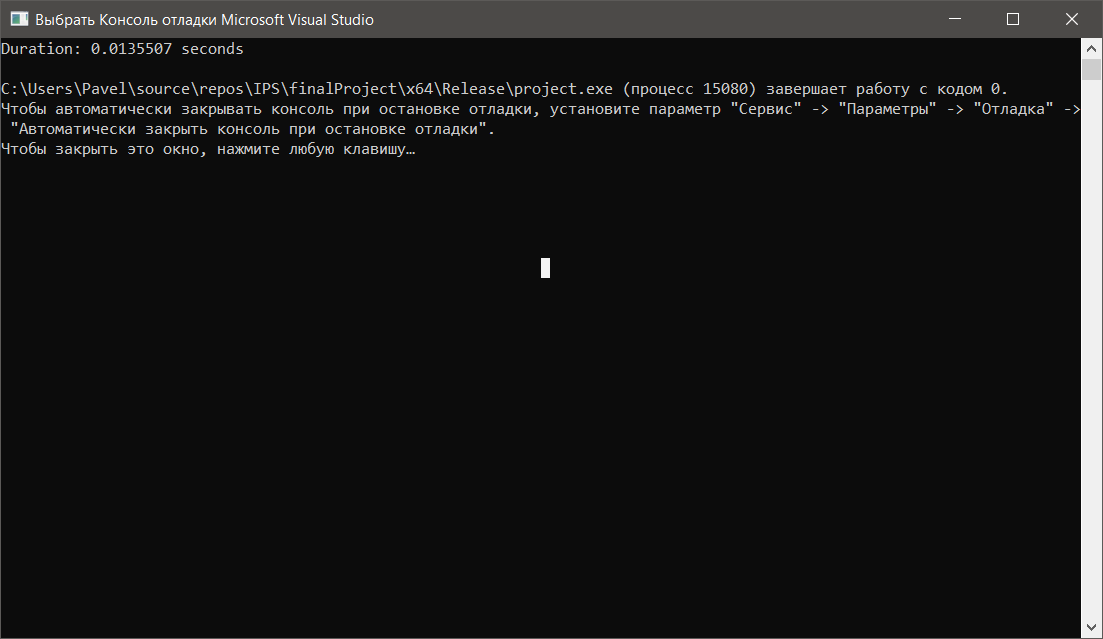
|  |
| --- |
| current\_box = Box(-g\_l1\_max, 0, g\_l2\_max + g\_l0 + g\_l1\_max, \_\_min(g\_l1\_max, g\_l2\_max));  std::vector<Box> current\_boxes;  temporary\_boxes->push\_back(current\_box);  int level = FindTreeDepth();  for (int i = 0; i < (level + 1); ++i)  {  temporary\_boxes.move\_out(buf\_boxes);  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](https://canvas.instructure.com/courses/1844563/files/87817241/download?wrap=1" \o "PrintWorkspace.m)[Предварительный просмотр документа](https://canvas.instructure.com/courses/1844563/files/87817241/download?wrap=1). Сохраните изображение рабочего пространства. Вставьте его в отчет, назвав соответствующим образом.

  
Рисунок 1 – Пример запуска программы на версии Debug x64

  
Рисунок 2 – Пример запуска программы на версии Release x64

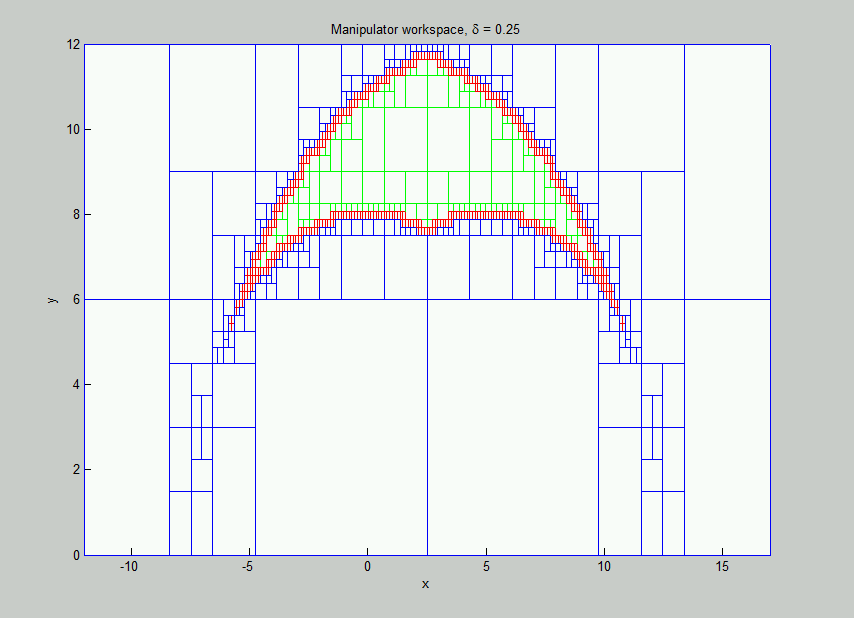


Рисунок 3 – Полученное рабочее пространство при delta = 0.25

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()***.

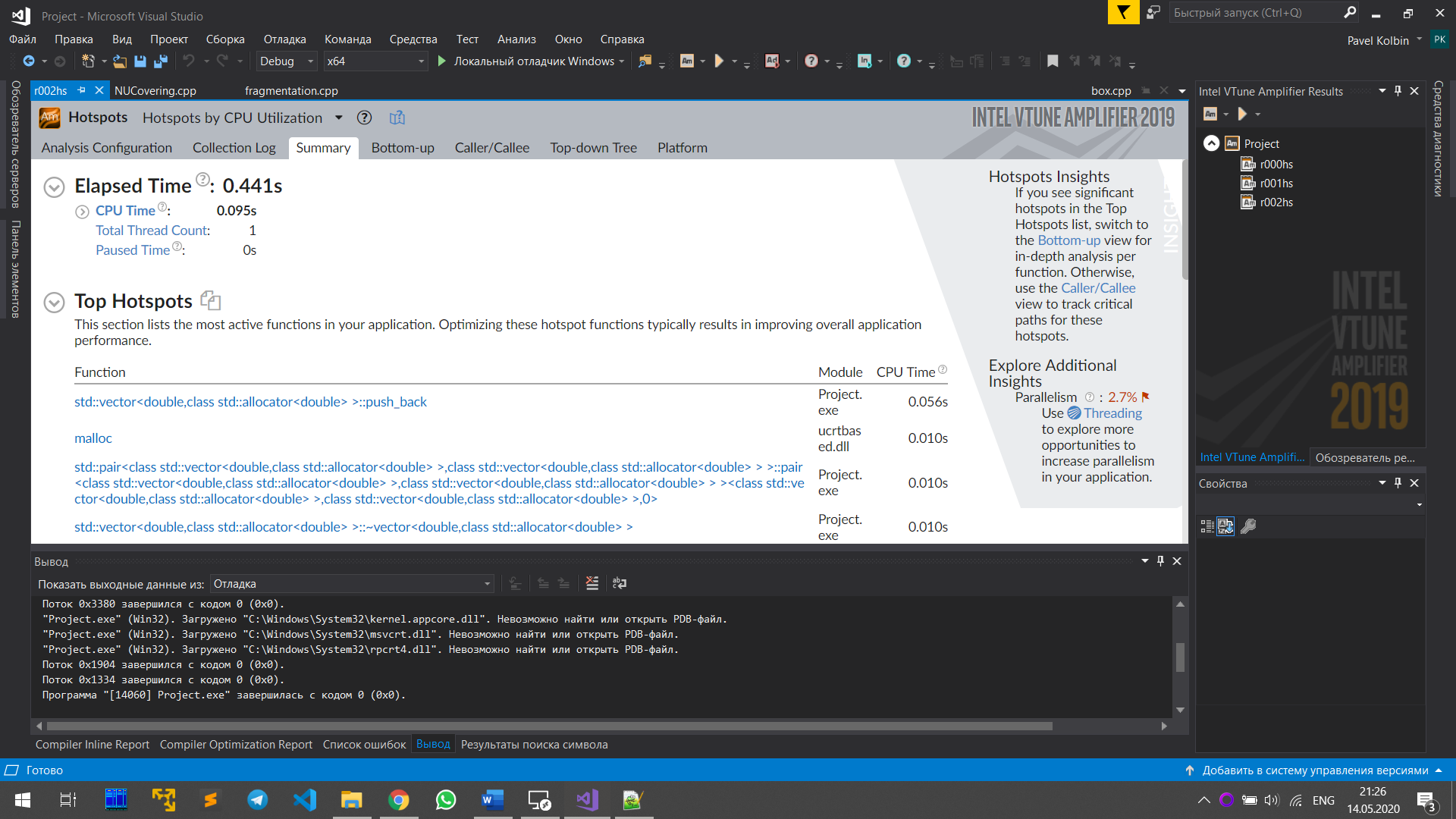


Рисунок 4 – результаты Amplifier XE

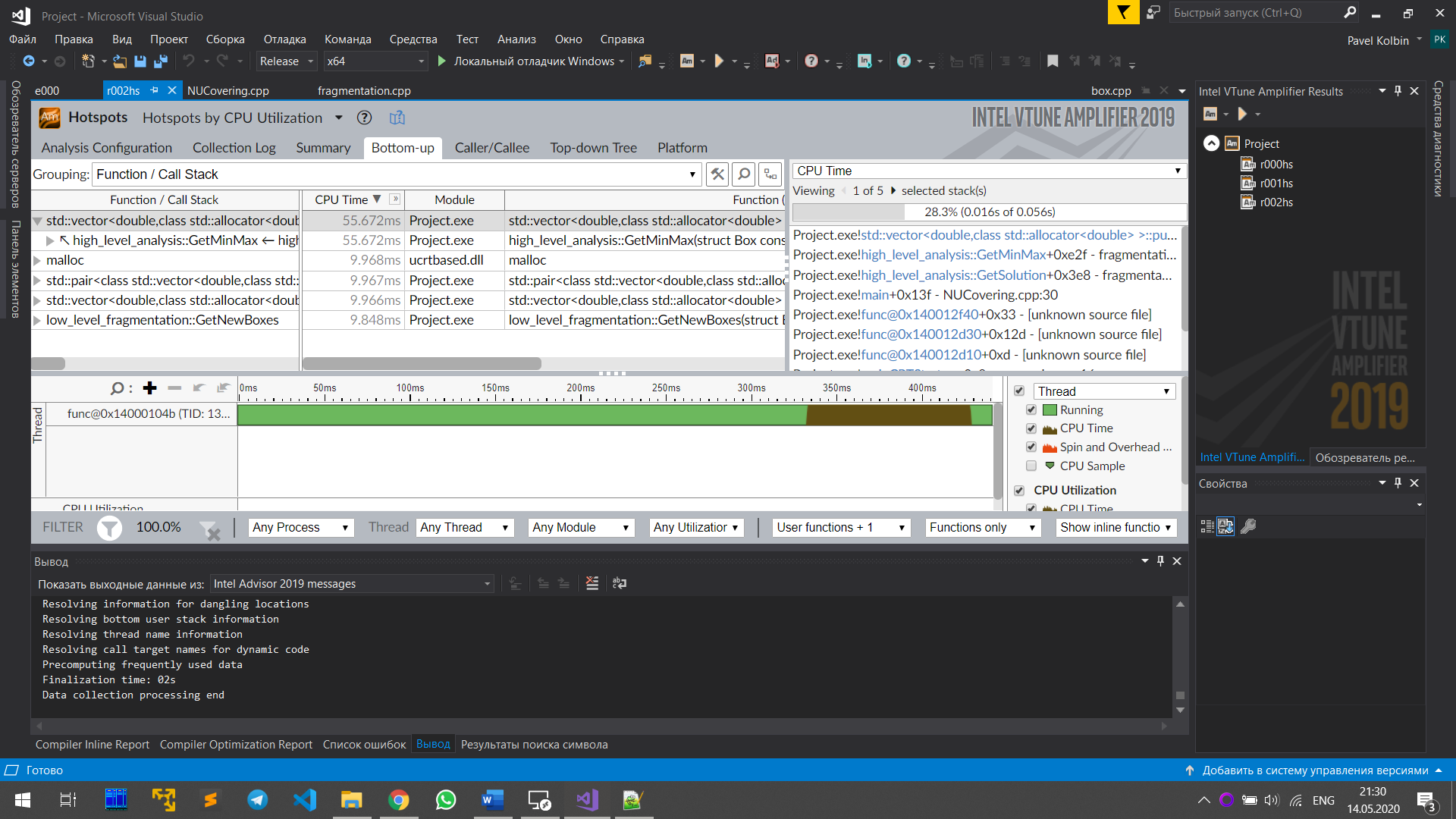
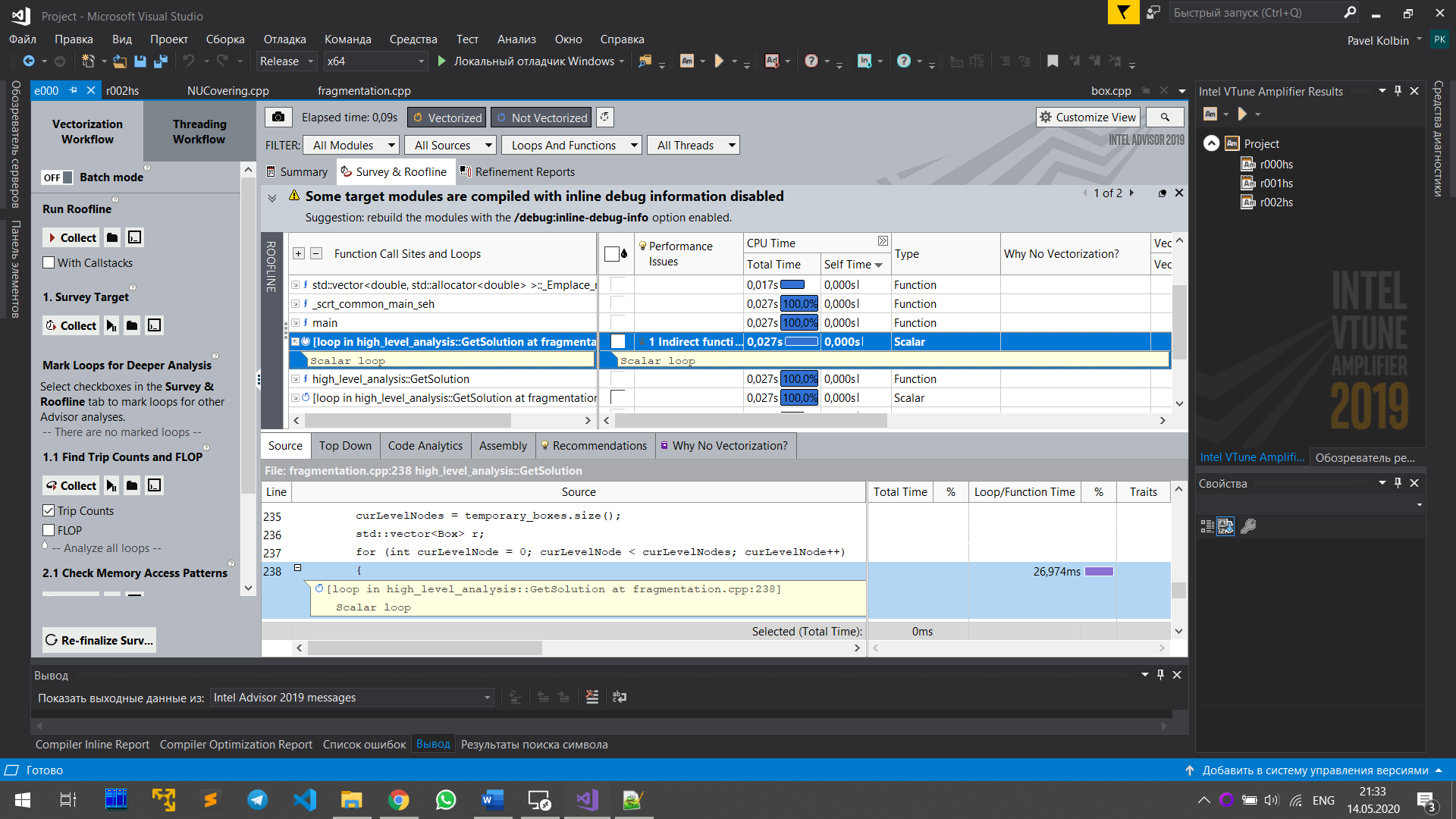


Рисунок 5 – результаты Amplifier XE вкладка Bottom-Up

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 ***С\С++*** 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 окончанию 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.

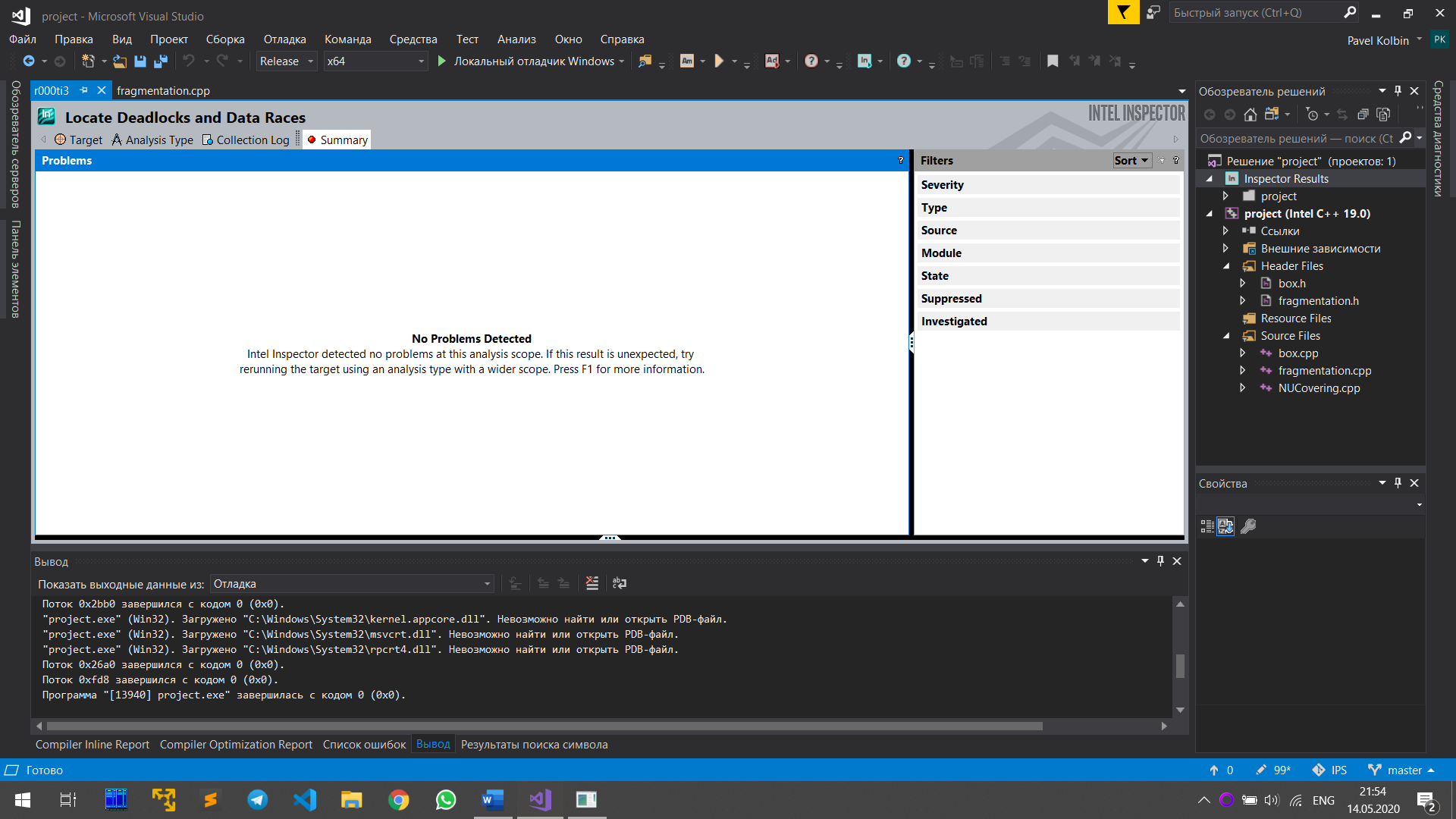


Рисунок 7 – Результаты работы Threading Error Analysis

Как видно гонок данных нет, значит используем reducer правильно

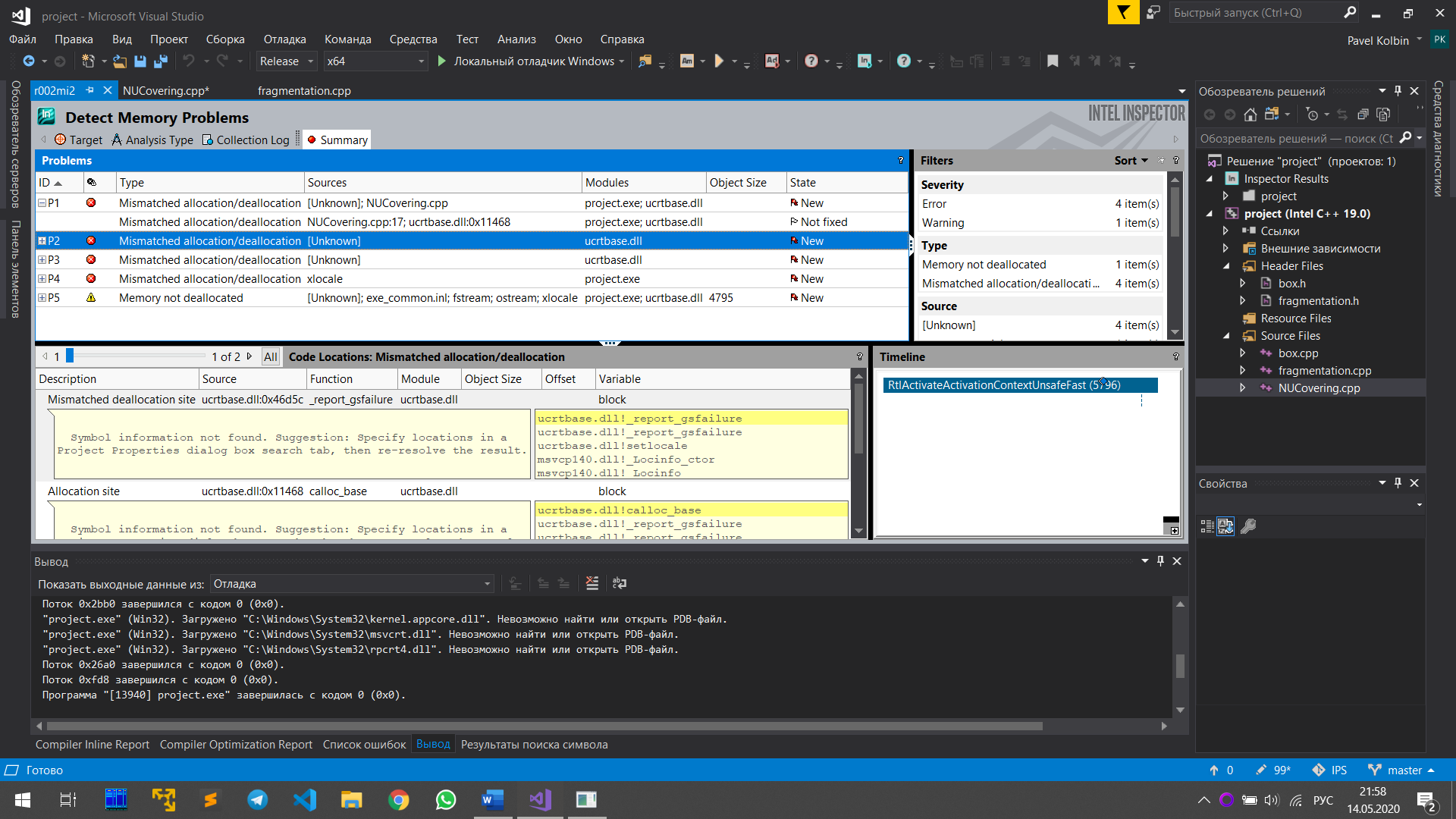


Рисунок 8 – Результаты работы Memory Error Analysis

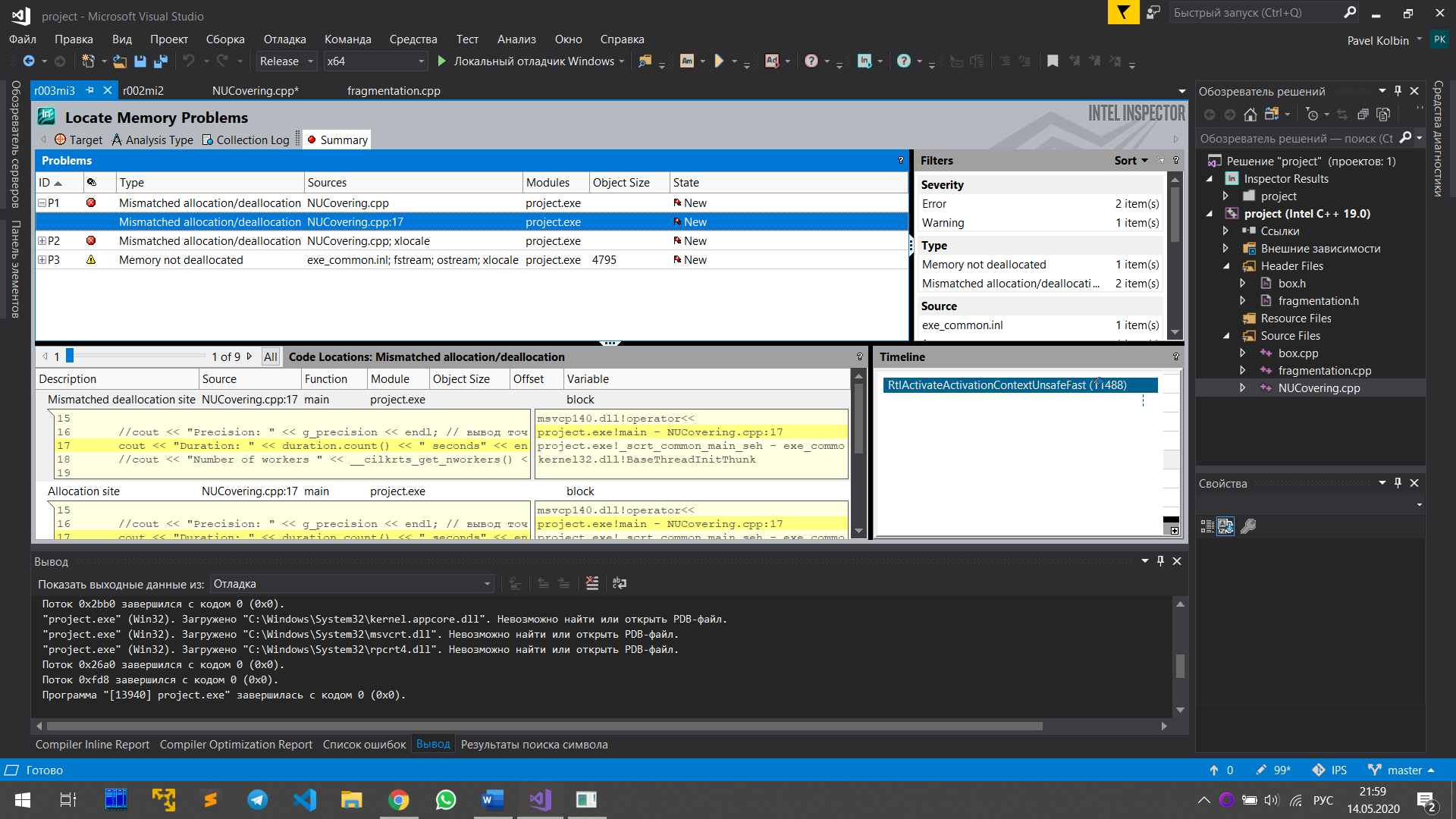


Рисунок 9 – Результаты работы Memory Error Analysis

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 *Х* - 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 *Х*, 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.

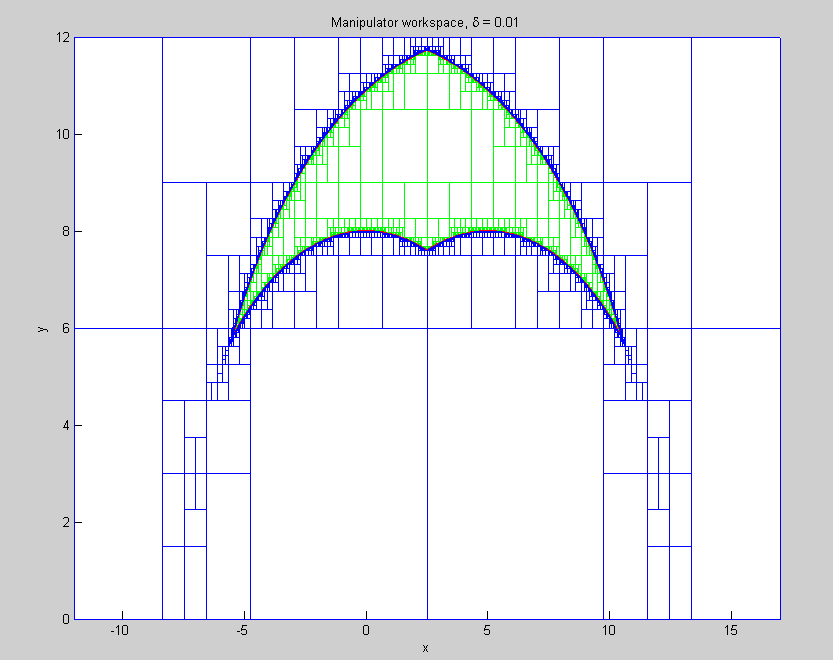


Рисунок 10 – Полученное рабочее пространство при delta = 0.01

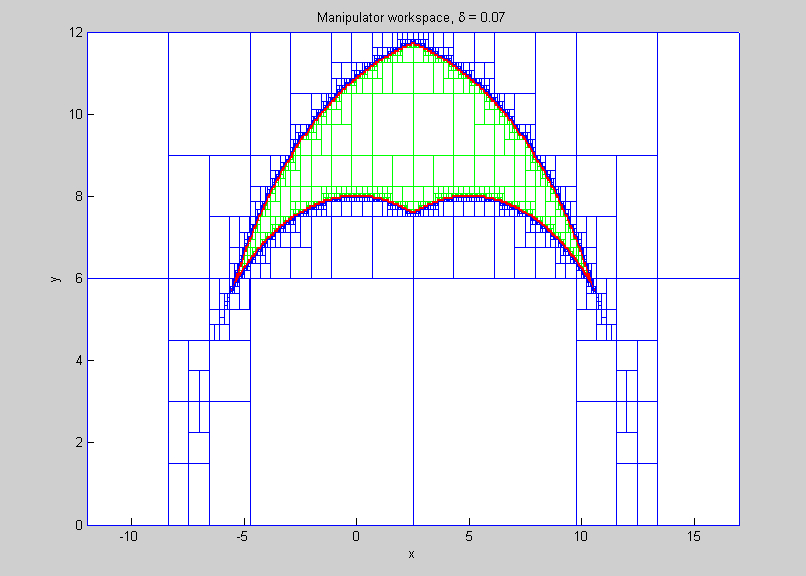


Рисунок 11 – Полученное рабочее пространство при delta = 0.07

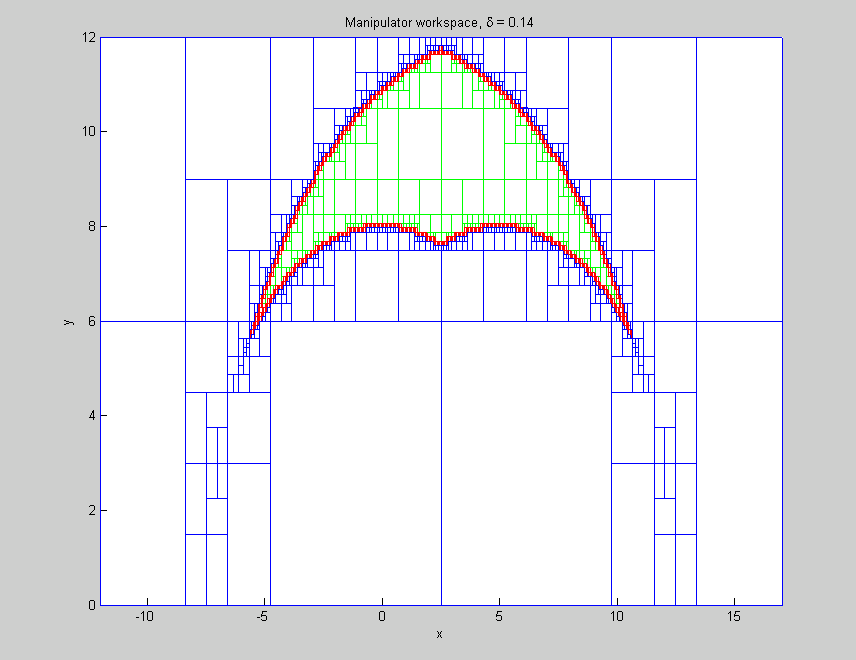


Рисунок 12 – Полученное рабочее пространство при delta = 0.14

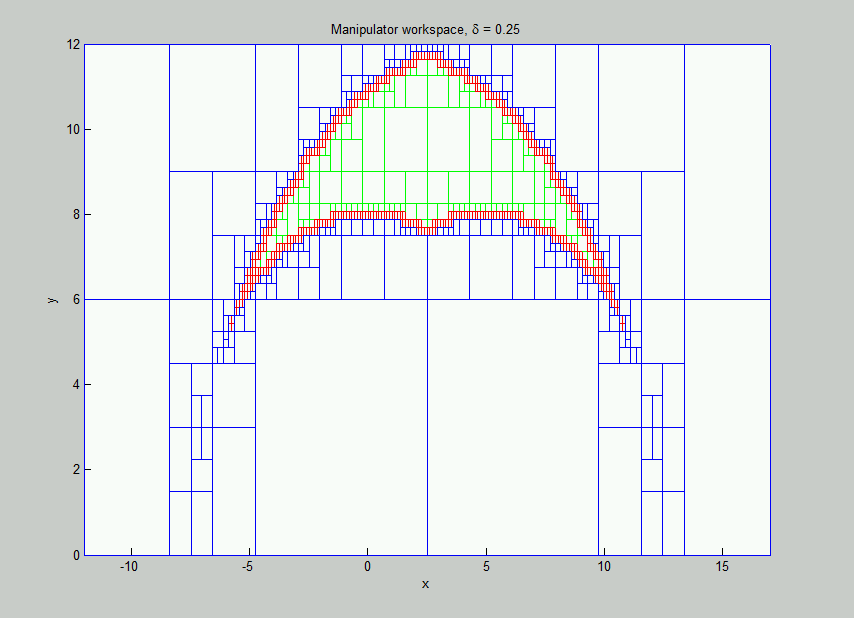


Рисунок 13 – Полученное рабочее пространство при delta = 0.25

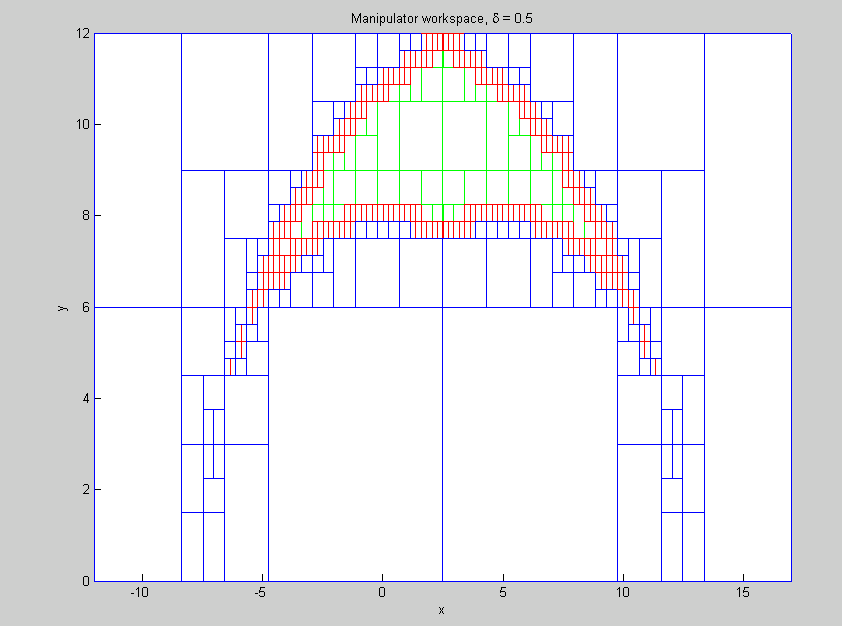


Рисунок 14 – Полученное рабочее пространство при delta = 0.5

Результаты зависимости delta и времени работы программы приведены в таблице ниже

|  |  |  |
| --- | --- | --- |
| **Количество потоков** | **delta** | **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 выявление 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 delta. Чем меньше дельта, тем дольше работает программа.