






Проектное задание

Теверовский Михаил, ИВТ-11М

Используемые программы и характеристики ПК:

1. ПО: VisualStudio 2017 и Intel Parallel Studio 2019;
2. ОС: Windows 7 Professional
3. Процессор: Intel core i5-4200 CPU, 2.50 GHz, 2 ядра, 4 потока

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

Задание 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"*.

Добавленные участки кода:

1) `void low_level_fragmentation::VerticalSplitter(const Box& box, boxes_pair& vertical_splitter_pair):`

```
void low_level_fragmentation::VerticalSplitter(const Box& box, boxes_pair&
vertical_splitter_pair)
{
    double del = 2.0;
    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 / del;
    newheight1 = height;

    Box leftbox(newleft1, newtop1, newwidth1, newheight1);

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

2) `void low_level_fragmentation::HorizontalSplitter(const Box& box, boxes_pair& horizontal_splitter_pair):`

```
double del = 2.0;
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;
newheight1 = height / del;
Box topbox(newleft1, newtop1, newwidth1, newheight1);

newleft2 = x_min;
newtop2 = y_min + height / del;
newwidth2 = width;
newheight2 = height / del;
Box bottombox(newleft2, newtop2, newwidth2, newheight2);
horizontal_splitter_pair.first = topbox;

horizontal_splitter_pair.second = bottombox;

```

3) `void low_level_fragmentation::GetNewBoxes(const Box& box, boxes_pair& new_pair_of_boxes):`

```

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);

```

4) `int low_level_fragmentation::ClasifyBox(const min_max_vectors& vects):`

```

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;
}

if (count == vects.second.size())
    return 0;

if (vects.first[0] == 0 && vects.second[0] == 0)
    return 2;

return 3;

```

5) `void low_level_fragmentation::GetBoxType(const Box& box):`

```

void low_level_fragmentation::GetBoxType(const Box& box)
{
    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; }

        case 1: {not_solution.push_back(box); break; }

        case 2: {boundary.push_back(box); break; }

        case 3:

```

```

        {
            GetNewBoxes(box, new_pair_of_boxes);
            temporary_boxes.push_back(new_pair_of_boxes.first);
            temporary_boxes.push_back(new_pair_of_boxes.second);

            break;
        }
    }
}

```

6) `void high_level_analysis::GetSolution():`

```

void high_level_analysis::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);
        buf_boxes.assign(temporary_boxes.begin(), temporary_boxes.end());

        current_boxes = buf_boxes;
        buf_boxes.clear();
        for(int j = 0; j < current_boxes.size(); ++j)
            GetBoxType(current_boxes[j]);
    }
}

```

7) `void WriteResults(const char* file_names[]):`

```

void WriteResults(const char* file_names[])
{
    double x_min, y_min, width, height;
    vector <Box> temp;

    std::ofstream fsolution(file_names[0]);
    std::ofstream fboundary(file_names[1]);
    std::ofstream fnot_solution(file_names[2]);

    //solution.move_out(temp);
    temp.assign(solution.begin(), solution.end());

    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();
    //boundary.move_out(temp);
    temp.assign(boundary.begin(), boundary.end());
    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();
    //not_solution.move_out(temp);
    temp.assign(not_solution.begin(), not_solution.end());
    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();
}

```

А также файл NUCovering.cpp, содержащий функцию main:

```

#include "fragmentation.h"
#include <locale.h>

/// параметры начальной прямоугольной области
/*
const double g_l1_max = 12.0;
const double g_l2_max = g_l1_max;
const double g_l1_min = 8.0;
const double g_l2_min = g_l1_min;
const double g_l0 = 5.0;

/// точность аппроксимации рабочего пространства
const double g_precision = 0.25;
*/

int main()
{
    setlocale(LC_ALL, "Rus");

    high_level_analysis main_object;
    high_resolution_clock::time_point start = high_resolution_clock::now();
    main_object.GetSolution();
    high_resolution_clock::time_point finish = high_resolution_clock::now();
    duration<double> duration = (finish - start);


    cout << "Precision: " << g_precision << endl;
    cout << "Duration: " << duration.count() << " seconds" << endl;

    const char* out_files[3] = { "Solution.txt", "Boundary.txt", "Not_Solution.txt" };
    WriteResults(out_files);

    return 0;
}

```

Задание 3: Реализация последовательной версии программы, определяющей рабочее пространство планарного робота, по предложенному в статье из *n.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.

Скорости выполнения в режиме:

1) Debug x64:

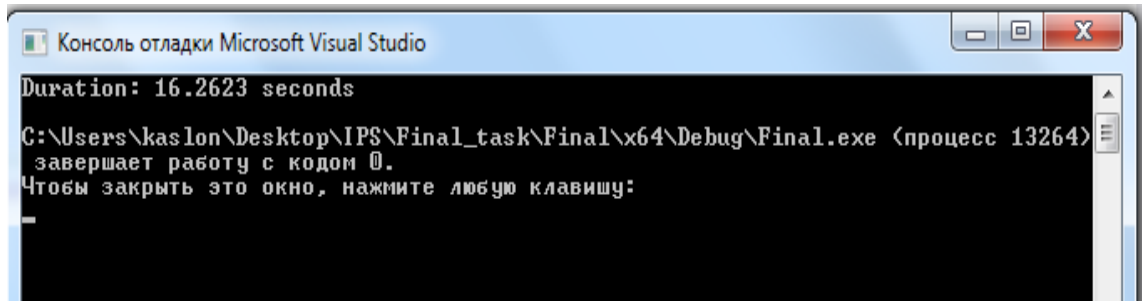


Рисунок 1. – Скриншот результата в режиме Debug

2) Release x64:

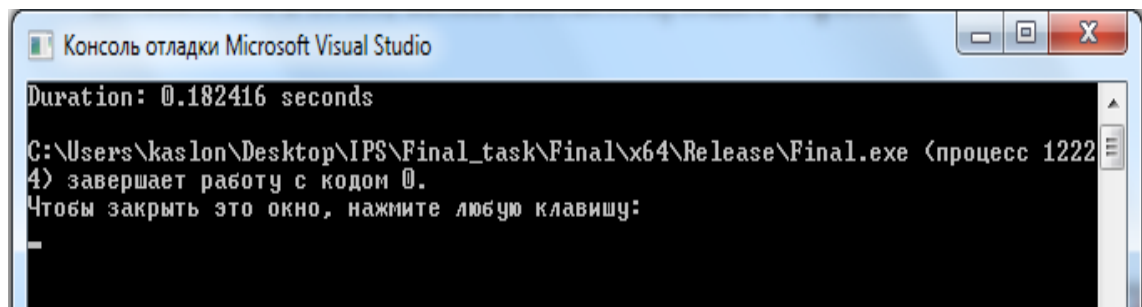


Рисунок 2. – Скриншот результата в режиме Release

Анализ полученных результатов:

Получили более 16 секунд в режиме Debug против 0.18 секунд в режиме Release.

Возможно это связано с тем, что оптимизация ещё не отключалась.

Файл [PrintWorkspace.m](#):

```
% Необходимо указать полный путь к считываемому файлу.
% A - матрица, содержащая значения параметров box'ов,
% являющихся решением исходной системы неравенств
A = dlmread('Solution.txt');
figure();
for i = 1:1:size(A,1)
    line([A(i,1) A(i,1)+A(i,3)], [A(i,2), A(i,2)], 'color', 'green');
    hold on;
    line([A(i,1) A(i,1)], [A(i,2), A(i,2)+A(i,4)], 'color', 'green');
    line([A(i,1)
A(i,1)+A(i,3)], [A(i,2)+A(i,4), A(i,2)+A(i,4)], 'color', 'green');
    line([A(i,1)+ A(i,3), A(i,1)+
A(i,3)], [A(i,2), A(i,2)+A(i,4)], 'color', 'green');
end
axis([-12 17 0 12]);

% Необходимо указать полный путь к считываемому файлу.
```

```

% В - матрица, содержащая значения параметров box'ов,
% находящихся на границе между множеством решений исходной системы
% и множеством, не являющимся решением
B = dlmread('Boundary.txt');
for i = 1:1:size(B,1)
    line([B(i,1) B(i,1)+B(i,3)], [B(i,2),B(i,2)], 'color','red');
    line([B(i,1) B(i,1)], [B(i,2),B(i,2)+B(i,4)], 'color','red');
    line([B(i,1) B(i,1)+B(i,3)], [B(i,2)+B(i,4),B(i,2)+B(i,4)], 'color','red');
    line([B(i,1)+ B(i,3),B(i,1)+
B(i,3)], [B(i,2),B(i,2)+B(i,4)], 'color','red');
end

% Необходимо указать полный путь к считываемому файлу.
% С - матрица, содержащая значения параметров box'ов,
% не являющихся решением исходной системы неравенств
C = dlmread('Not_Solution.txt');
for i = 1:1:size(C,1)
    line([C(i,1) C(i,1)+C(i,3)], [C(i,2),C(i,2)]);
    line([C(i,1) C(i,1)], [C(i,2),C(i,2)+C(i,4)]);
    line([C(i,1) C(i,1)+C(i,3)], [C(i,2)+C(i,4),C(i,2)+C(i,4)]);
    line([C(i,1)+ C(i,3),C(i,1)+ C(i,3)], [C(i,2),C(i,2)+C(i,4)]);
end

title('Manipulator workspace, \delta = 0.25');
xlabel('x'); ylabel('y');

```

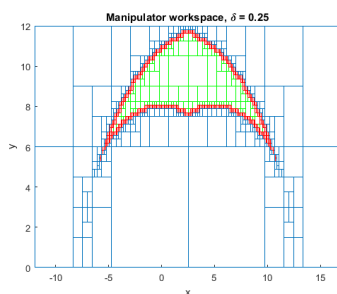


Рисунок 3. – Скриншот полученного рабочего пространства по заданию 3

Задание 4: Использование *Amplifier XE* в целях определения наиболее часто используемых участков кода. Для этого прокомментируйте 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()*.

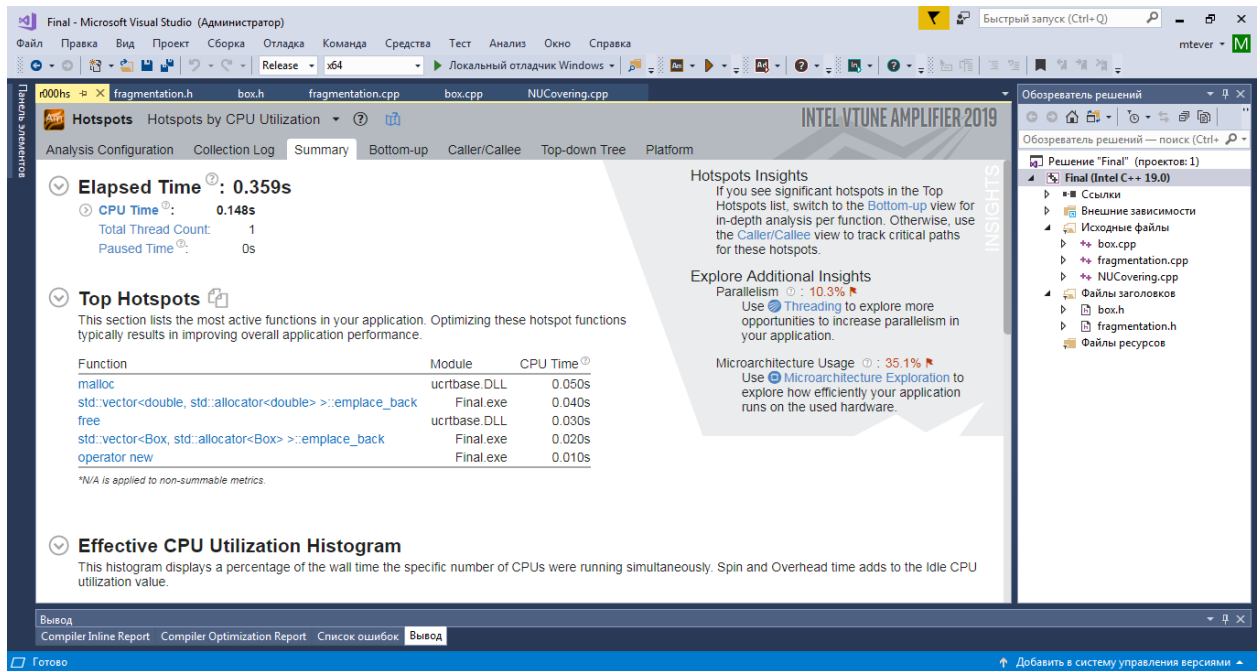


Рисунок 4. – Скриншот результатов Amplifier XE. Summary

В окне Bottom-up как раз фигурирует функция *GetMinMax()*:

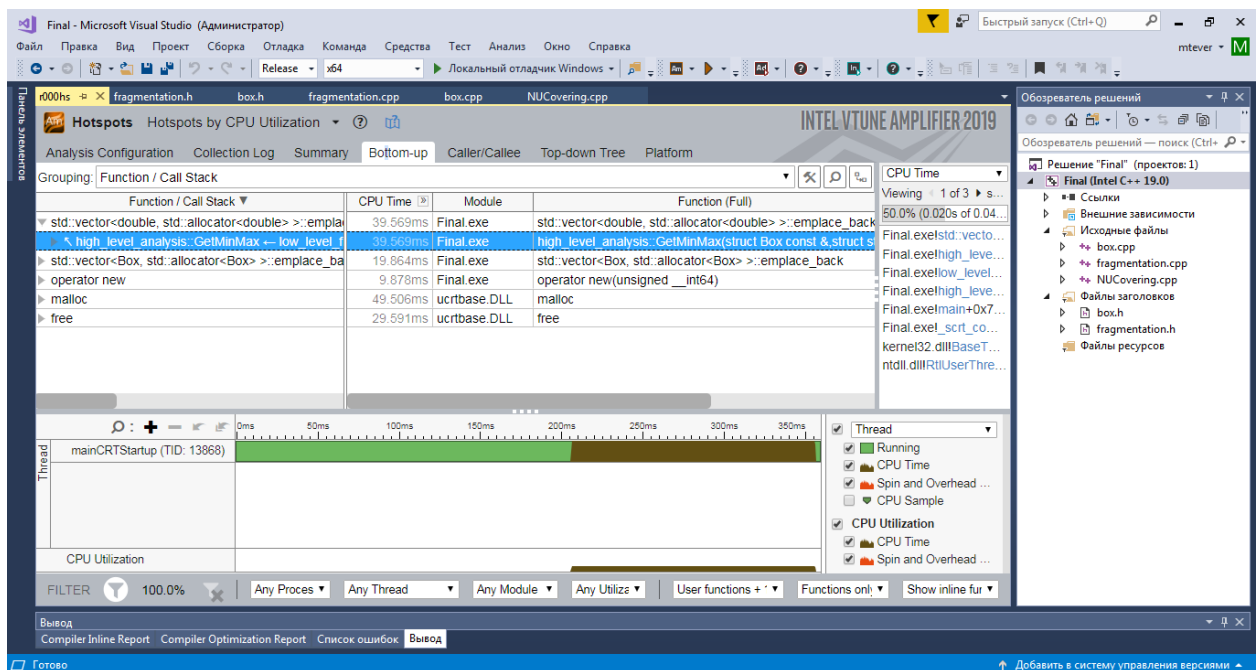


Рисунок 5. – Скриншот результатов Amplifier XE. Bottom-up

Задание 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*.

1) Отключение оптимизации:

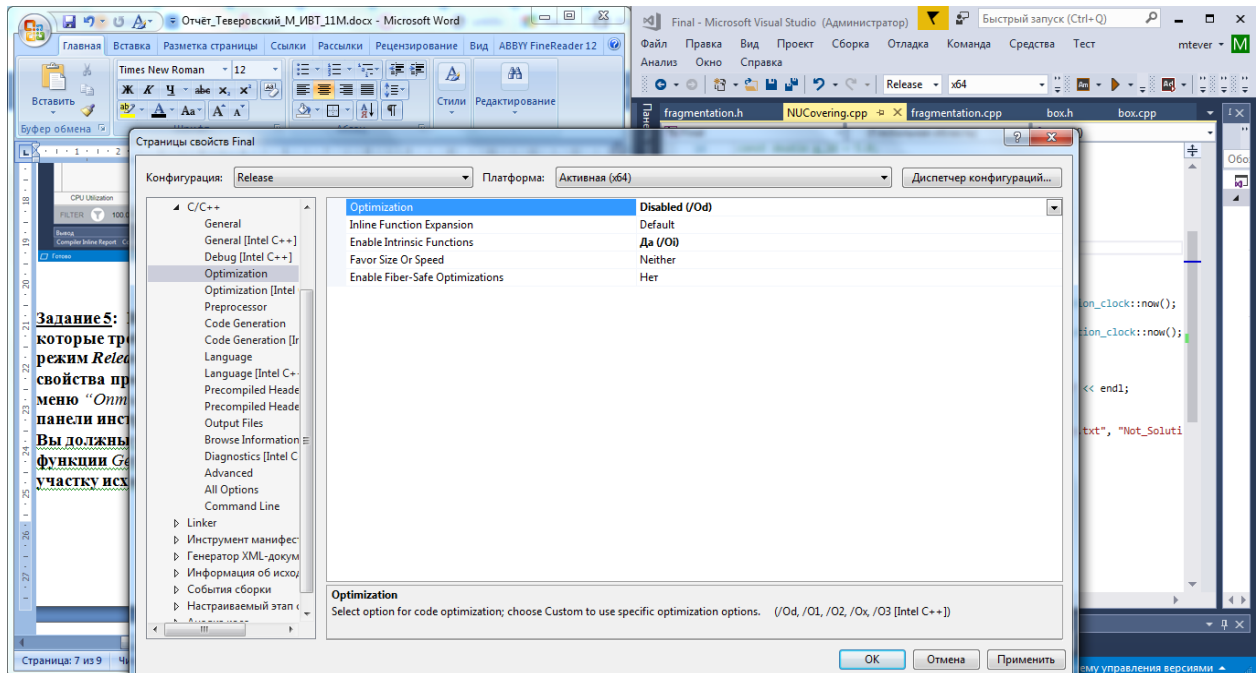


Рисунок 6. – Настройки проекта

2) Parallel Advisor:

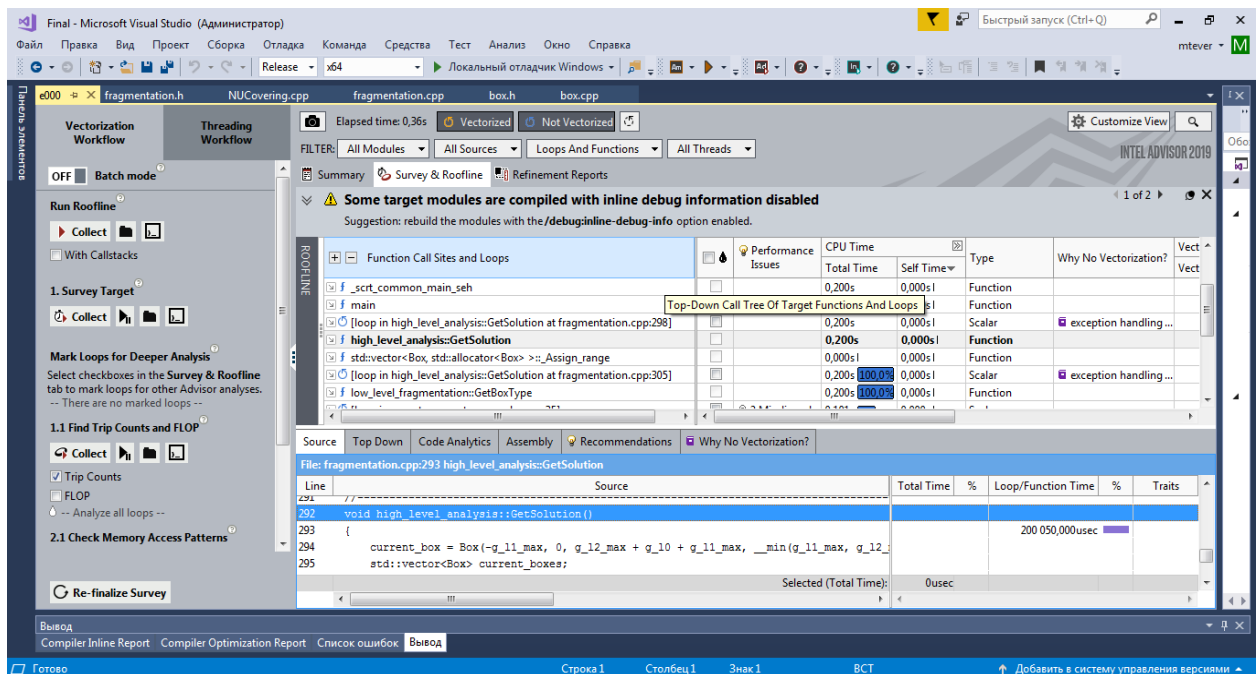


Рисунок 7. – Скриншот результатов Parallel Advisor

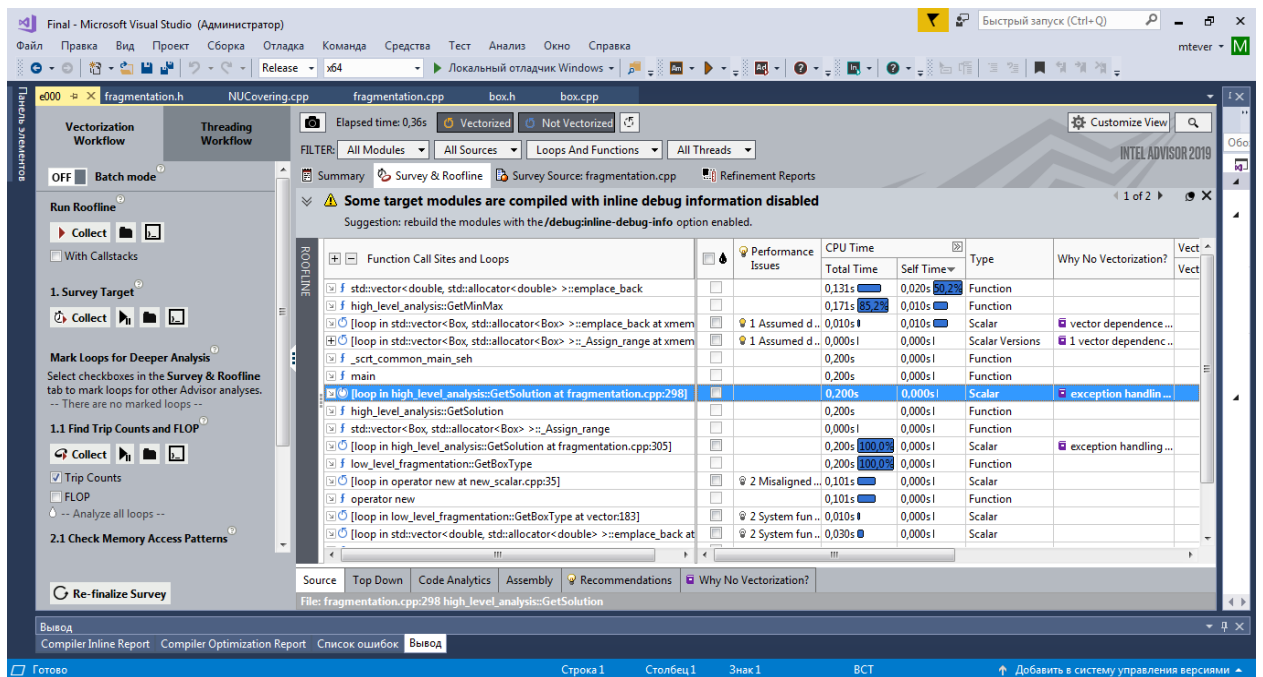


Рисунок 8. – Скриншот результатов Parallel Advisor

Анализ полученных результатов:

Действительно, наибольшее время затрачивается на цикл `GetSolution()`. Необходимо введение параллелизма.

Задание 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.

Изменённый участок кода в файле `fragmentation.cpp`:

1) reduced:

```
std::vector<Box> buf_boxes;
cilk::reducer< cilk::op_vector<Box> > solution;
cilk::reducer< cilk::op_vector<Box> > not_solution;
cilk::reducer< cilk::op_vector<Box> > boundary;
cilk::reducer< cilk::op_vector<Box> > temporary_boxes;
```

2) `GetSolution()`:

```
void high_level_analysis::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);
    //buf_boxes.assign(temporary_boxes.begin(), temporary_boxes.end());

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

```

Время выполнения программы в результате:

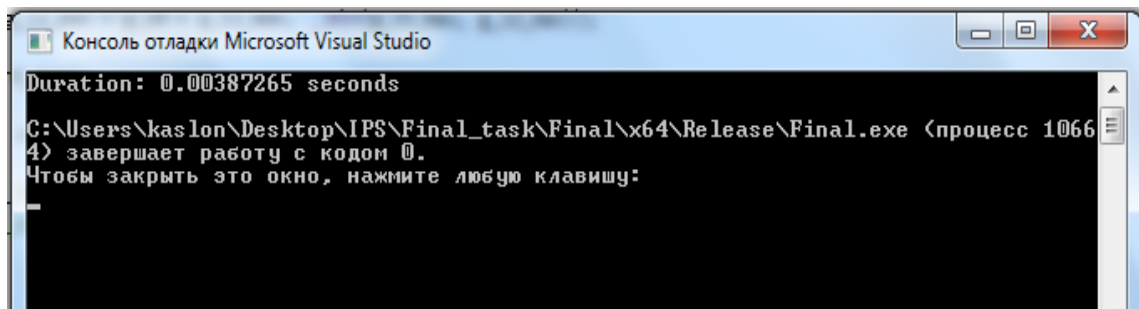


Рисунок 8. – Время выполнения кода с параллелизмом

*Время выполнения сократилось до 3 миллисекунд, тем не менее необходимы проверки на наличие гонок данных и иных ошибок при введении параллелизма в код.

Задание 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.

1) *Threading Error Analysis*

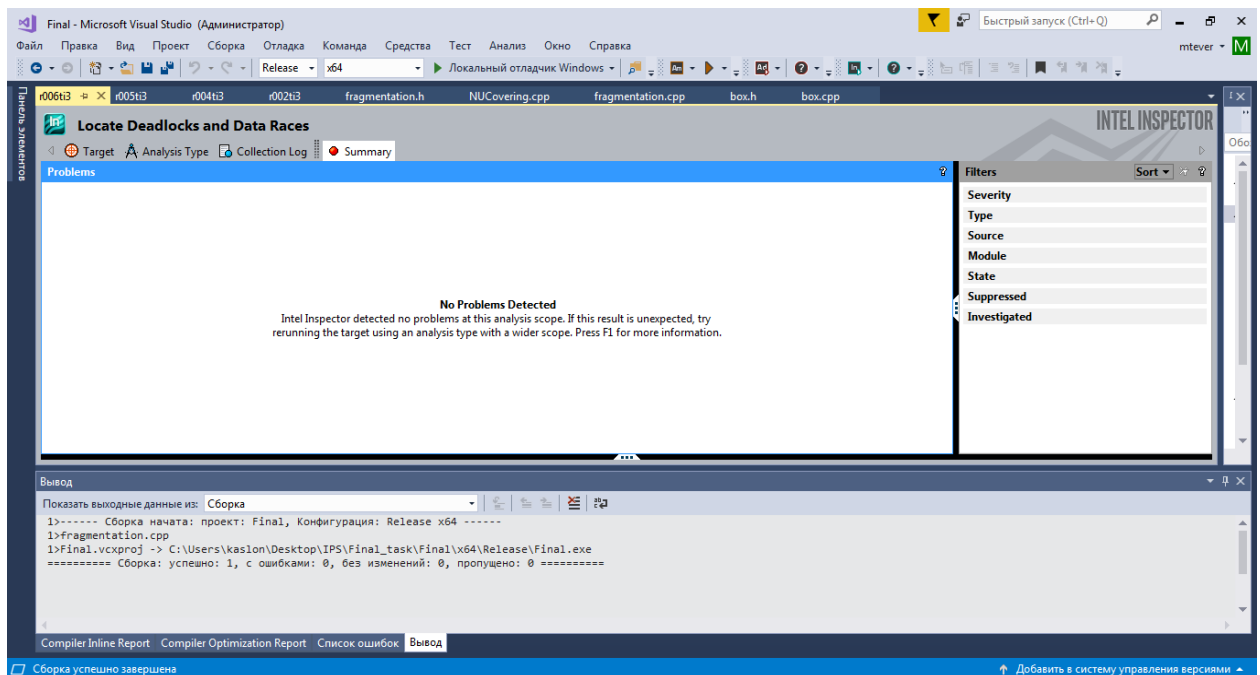


Рисунок 9 – Результаты работы Inspector XE.ThreadingErrorAnalysis

Анализ полученных результатов:

Гонок данных обнаружено не было, значит reducer используется верно

2) Memory Error Analysis

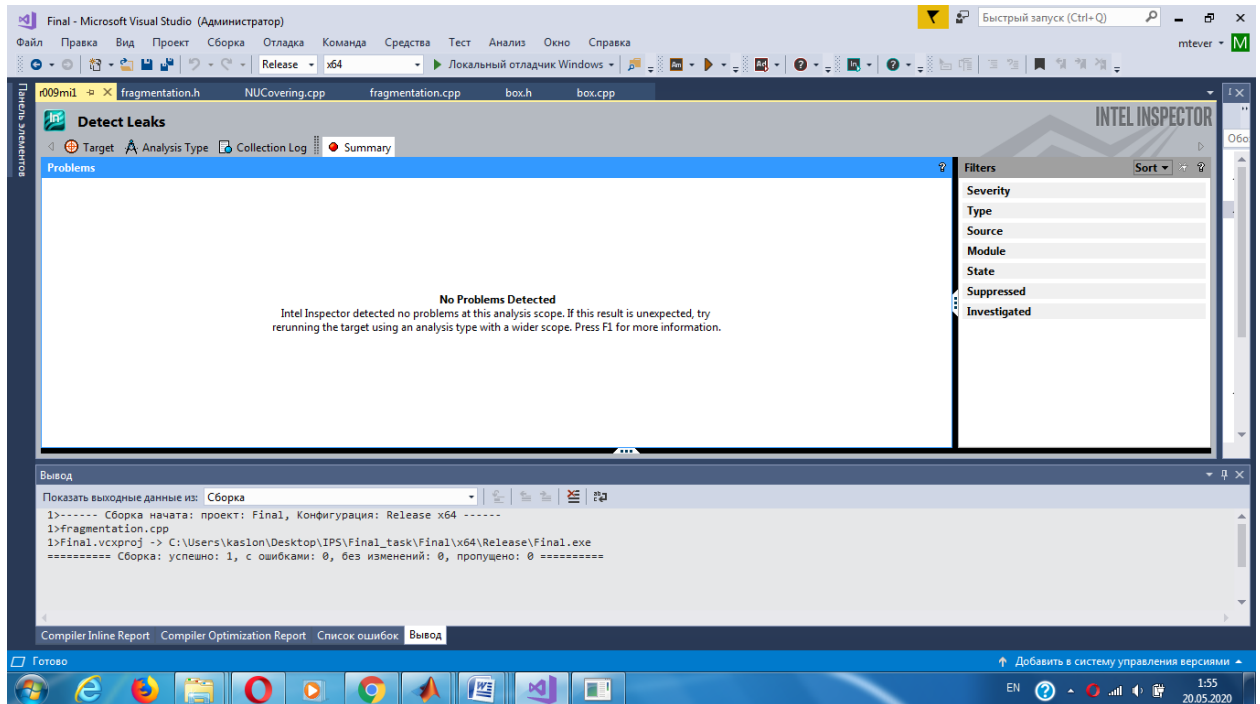


Рисунок 10 – Результаты работы Inspector XE. Detected Leaks

3) Locate Memory Problems

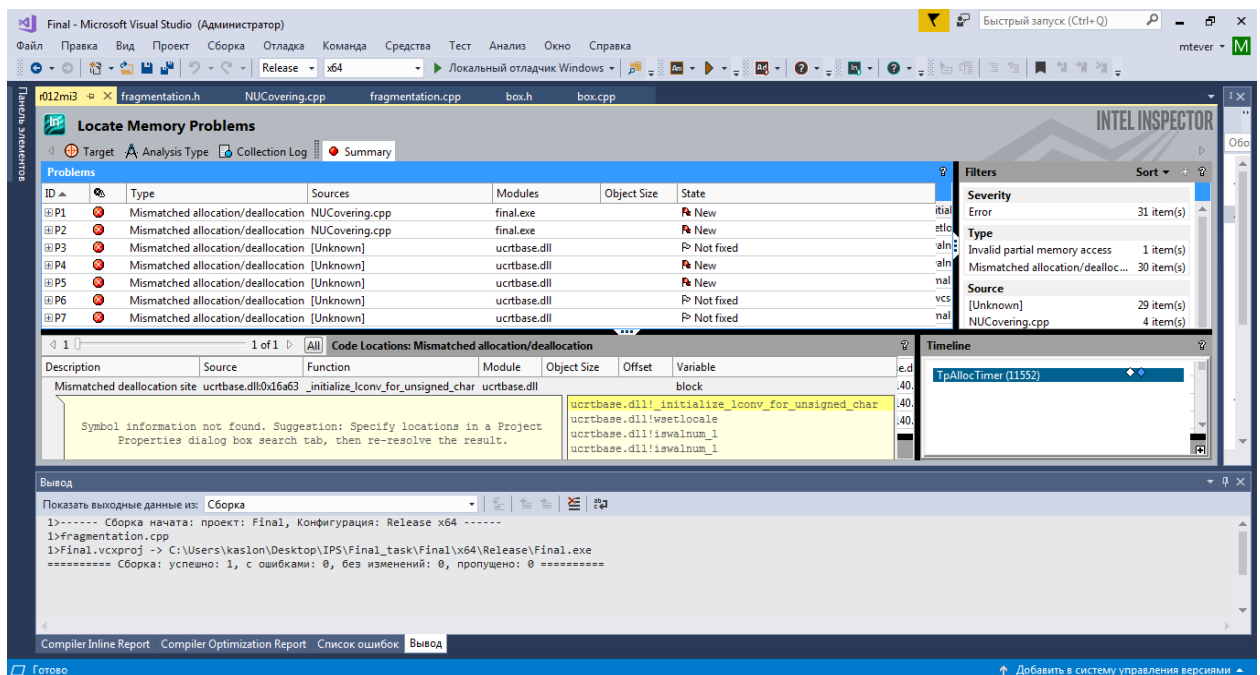


Рисунок 11 – Результаты работы Inspector XE. Locate Memory Problems

Задание 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 заголовочный 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.

Изменённый файл *NUCovering.cpp* :

```
#include "fragmentation.h"
#include <locale.h>
#include <cilk/cilk_api.h>

/// параметры начальной прямоугольной области
/*
const double g_l1_max = 12.0;
const double g_l2_max = g_l1_max;
const double g_l1_min = 8.0;
const double g_l2_min = g_l1_min;
const double g_l0 = 5.0;

/// точность аппроксимации рабочего пространства
const double g_precision = 0.25;
*/

int main()
{
    setlocale(LC_ALL, "Rus");
    __cilkrts_end_cilk(); __cilkrts_set_param("nworkers", "2");

    high_level_analysis main_object;
    high_resolution_clock::time_point start = high_resolution_clock::now();
    main_object.GetSolution();
    high_resolution_clock::time_point finish = high_resolution_clock::now();
    duration<double> duration = (finish - start);

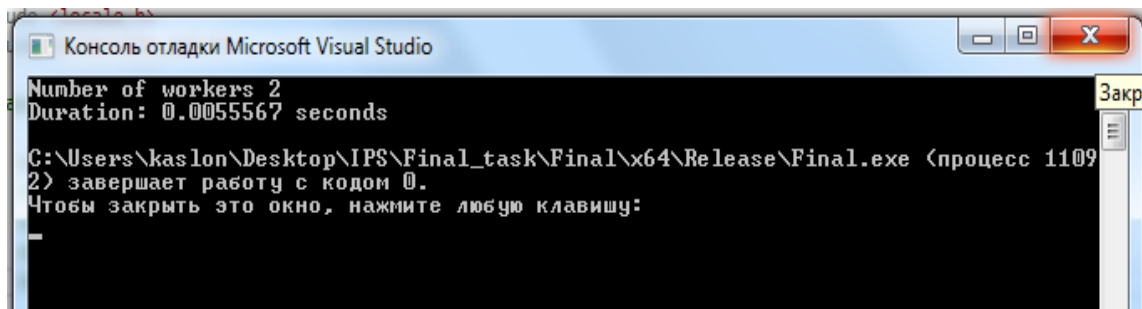
    cout << "Number of workers " << __cilkrts_get_nworkers() << endl;
    cout << "Duration: " << duration.count() << " seconds" << endl;

    const char* out_files[3] = { "Solution.txt", "Boundary.txt", "Not_Solution.txt" };
    WriteResults(out_files);

    return 0;
}
```

К сожалению, в моём ноутбуке всего лишь 2 ядра, поэтому ниже представлены 2 попытки запуска:

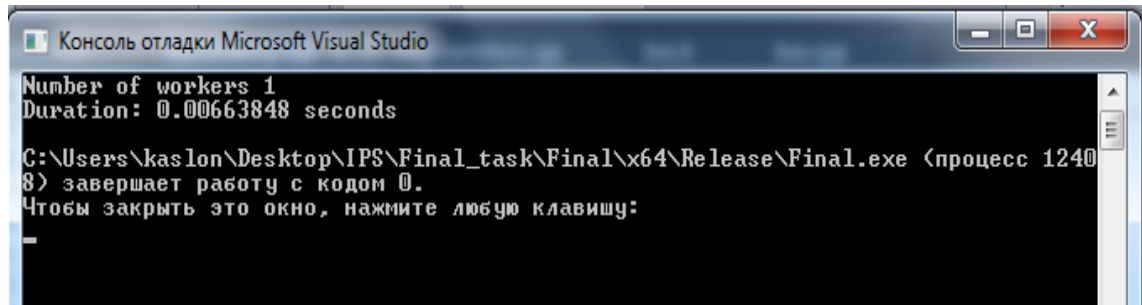
1) При 2 ядрах:



```
Консоль отладки Microsoft Visual Studio
Number of workers 2
Duration: 0.0055567 seconds
C:\Users\kaslon\Desktop\IPS\Final_task\Final\x64\Release\Final.exe (процесс 11092) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
-
```

Рисунок 12 – Скорость выполнения при использовании 2 ядер

2) При 1 ядре:



```
Консоль отладки Microsoft Visual Studio
Number of workers 1
Duration: 0.00663848 seconds
C:\Users\kaslon\Desktop\IPS\Final_task\Final\x64\Release\Final.exe (процесс 12408) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
-
```

Рисунок 12 – Скорость выполнения при использовании 1 ядра

Анализ полученных результатов:

Количество ядер	2	1
Скорость выполнения	0.0055567	0.00663848

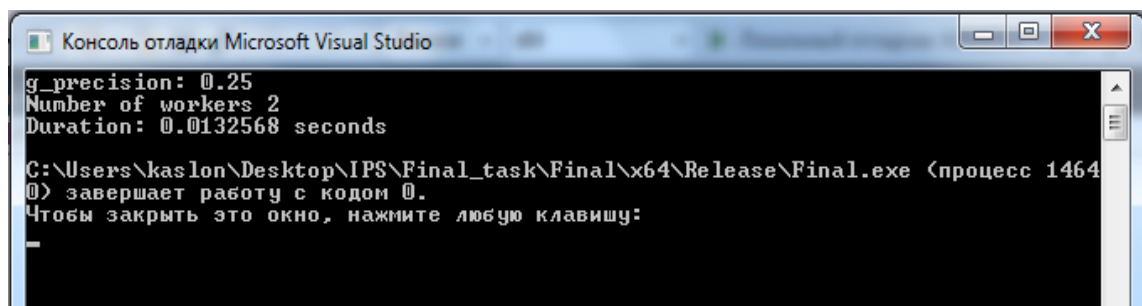
Таблица 1 – Сравнение скорости выполнения

Примерно на 1 миллисекунду выполнение при использовании 2 ядер быстрее

Задание 10: Визуализация полученного решения. Поэкспериментируйте со входными параметрами программы и отобразите несколько версий полученного рабочего пространство робота. Рисунки приложите к отчету.

Для сравнения везде будем использовать 2 ядра.

1) При $\delta = 0.25$:



```
Консоль отладки Microsoft Visual Studio
g_precision: 0.25
Number of workers 2
Duration: 0.0132568 seconds
C:\Users\kaslon\Desktop\IPS\Final_task\Final\x64\Release\Final.exe (процесс 14640) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
-
```

Рисунок 13 – Результат при $\delta = 0.25$

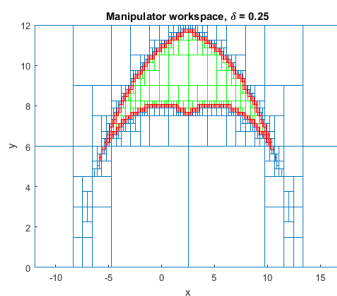


Рисунок 14 – Результат при $\delta = 0.25$

2) При $\delta = 0.5$:

```
Консоль отладки Microsoft Visual Studio
g_precision: 0.5
Number of workers 2
Duration: 0.00712168 seconds
C:\Users\kaslon\Desktop\IPS\Final_task\Final\x64\Release\Final.exe (процесс 12508)
> завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
_
```

Рисунок 15 – Результат при $\delta = 0.5$

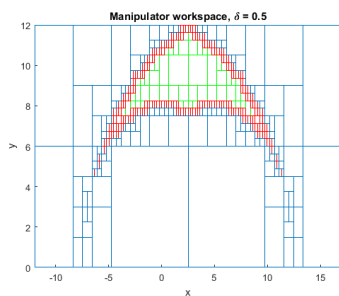


Рисунок 16 – Результат при $\delta = 0.5$

3) При $\delta = 0.1$:

```
Консоль отладки Microsoft Visual Studio
g_precision: 0.1
Number of workers 2
Duration: 0.0337873 seconds
C:\Users\kaslon\Desktop\IPS\Final_task\Final\x64\Release\Final.exe <процесс 15316> завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
_
```

Рисунок 17 – Результат при $\delta = 0.1$

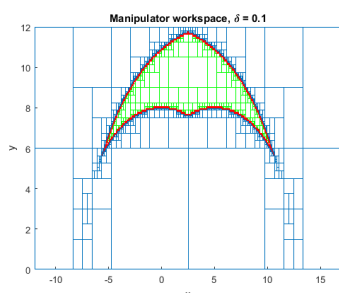


Рисунок 18 – Результат при $\delta = 0.1$

delta	0.25	0.5	0.1
Количество ядер	2	2	2
Время выполнения	0.0132568	0.00712168	0.0337873

Таблица 2 – Сравнение результатов

Следовательно: чем больше значение параметра δ , тем быстрее выполняется программа.

Выводы:

- 1) Были реализованы необходимые функции для модели работы планарного робота
- 2) В ходе выполнения задания были протестированы результаты выполнения последовательного выполнения программы на различных версиях и замерялось время выполнения
- 3) С помощью пакета MATLAB было построено рабочее пространство для последовательной программы
- 4) С помощью инструментов IPS: Amplifier XE и Survey Analise был проанализирован последовательный код для введения параллелизма
- 5) В код был введён параллелизм

- 6) С помощью Amplifier XE и ParallelAdvisor была проверена правильность работы кода с параллельными участками выполнения
- 7) Было измерено время выполнения программы в зависимости от количества используемых ядер процессора
- 8) Было измерено время выполнения программы при использовании 2 ядер в зависимости от параметра delta.