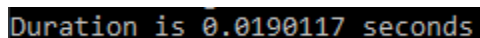


Отчёт по проектному заданию

Петрова О.А. группы ИВТ-12М

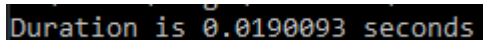
1. Ознакомьтесь со статьей [The non-uniform covering approach to manipulator workspace assessment.pdf](#).
2. Скачайте следующие файлы: [box.h](#), [box.cpp](#), [fragmentation.h](#), [fragmentation.cpp](#), [NUCovering.cpp](#). В этих файлах представлен предлагаемый каркас разрабатываемого проекта. Ознакомьтесь с содержимым каждого файла. После выполнения *n.1*. Вашей задачей является написание определений тех функций проекта, в теле которых представлен комментарий *"// необходимо определить функцию"*.
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.

После выполнения всех вышеуказанных требований для последовательной программы выполним её в режиме debug и режиме release:



```
Duration is 0.0190117 seconds
```

Рисунок 1. Результат выполнения последовательной программы в режиме Debug



```
Duration is 0.0190093 seconds
```

Рисунок 2. Результат выполнения последовательной программы в режиме Release

После выполнения программы будут сформированы текстовые файлы с данными (Solution.txt, Not_Solution.txt, Boundary.txt), путь к которым нужно указать в скрипте [PrintWorkspace.m](#).

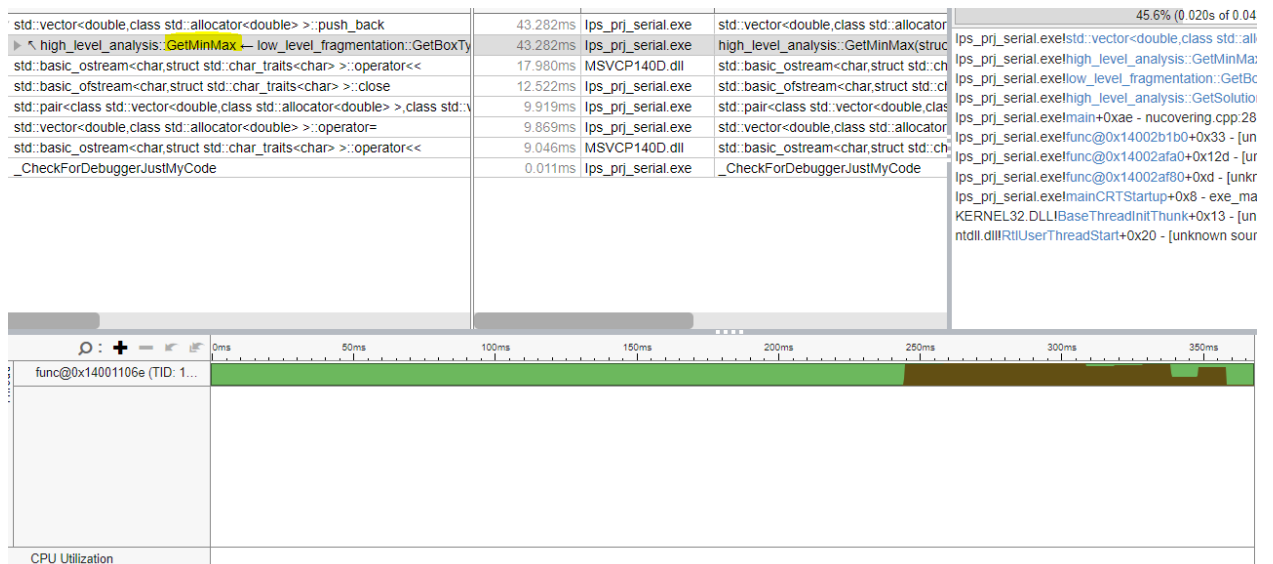


Рисунок 5. Результат работы Amplifier XE вкладка Bottom-up

Из рисунка 5 видно, что функция `GetMinMax()` действительно имеется в списке.

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*.

ROOFLINE	Function Call Sites and Loops	Performance Issues	CPU Time		Type
			Total Time	Self Time	
	f _scrt_common_main_seh		0,035s 100,0%	0,000s	Function
	f main		0,035s 100,0%	0,000s	Function
	high_level_analysis::GetSolution		0,021s	0,000s	Function
	f WriteResults		0,014s	0,000s	Function
	[loop in high_level_analysis::GetSolution at fragmentation.cpp:282]	1 System fun...	0,011s	0,000s	Scalar
	[loop in high_level_analysis::GetSolution at fragmentation.cpp:282]		0,011s	0,000s	Scalar
	f low_level_fragmentation::GetBoxType		0,011s	0,000s	Function
	f std::vector<Box, std::allocator<Box> >::~vector		0,010s	0,000s	Function
	f std::vector<struct Box, class std::allocator<struct Box> >::_Tidy		0,010s	0,000s	Function
	f std::allocator<struct Box>::deallocate		0,010s	0,000s	Function
	f std::_Deallocate		0,010s	0,000s	Function
	f std::vector<struct std::Box, class std::allocator<struct Box> >::~vector		0,010s	0,010s	Function

Source	Top Down	Code Analytics	Assembly	Recommendations	Why No Vectorization?
--------	----------	----------------	----------	-----------------	-----------------------

File: fragmentation.cpp:282 high_level_analysis::GetSolution

Line	Source
276	min_max_vecs.first = g_min;
277	min_max_vecs.second = g_max;
278	}
279	
280	
281	void high_level_analysis::GetSolution()
282	{
283	current_box = Box(-g_l1_max, 0, g_l2_max + g_l0 + g_l1_max, __min(g_l1_max, g_l2_max));
284	std::vector<Box> current_boxes;
285	temporary_boxes.push_back(current_box);
286	std::vector<Box> test;
287	

Рисунок 6. Результат работы Parallel Advisor

File: fragmentation.cpp:282 high_level_analysis::GetSolution			
Line	Source	Total Time	%
290	for (int i = 0; i < (level + 1); ++i)		
291	{		
292	current_boxes.assign(temporary_boxes.begin(), temporary_boxes.end());		
293	temporary_boxes.clear();		
294	for (int j = 0; j < current_boxes.size(); ++j)		
295	GetBoxType(current_boxes[j]);		
296	}	9,958ms	
297	}		
298			
299	//-----		
300	// функция WriteResults() записывает параметры полученных box-ов (относясь к пространству, к граничной области и ко множеству, не являющемуся решением)		
301			

Рисунок 7. Результат работы Parallel Advisor

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

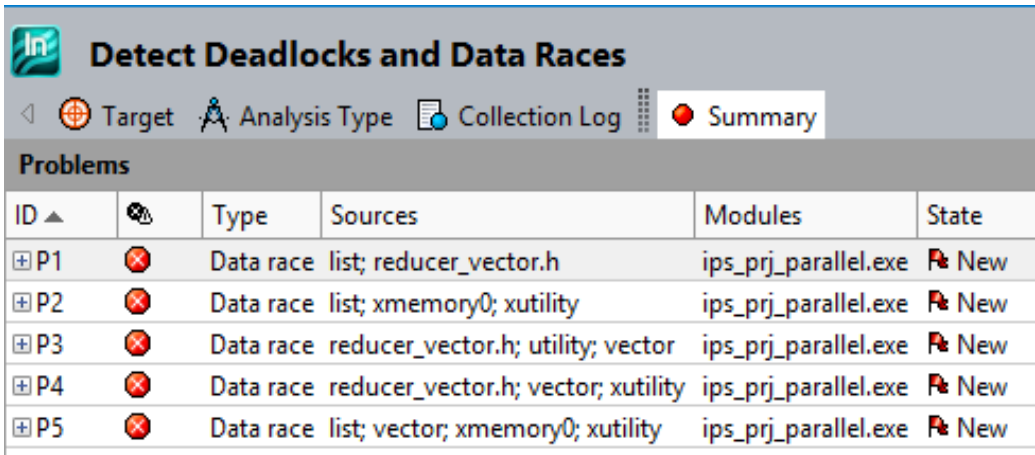
дальнейшему анализу). Вам необходимо ввести параллелизм во внутренний цикл. Тогда следует подумать о возможности независимого обращения к векторам *solution*, *not_solution*, *boundary*, *temporary_boxes*. Для этого предлагается использовать *reducer* векторы *Intel Cilk Plus*, вместо обычных *std::vector*’ов.

Duration is 0.00489043 seconds

Рисунок 8. Результат работы параллельной программы

В результате видно, что параллельная программа выполняется быстрее чем последовательная.

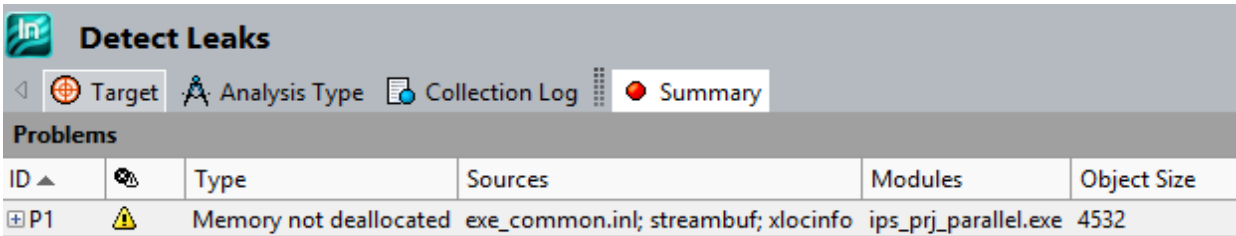
7. Определение ошибок после введения параллелизации. Запустите анализы *Inspector XE: Memory Error Analysis* и *Threading Error Analysis* на различных уровнях (*Narrowest*, *Medium*, *Widest*). Приложите к отчету скрины результатов запуска перечисленных анализов. Исправьте обнаруженные ошибки, приложите новые скрины результатов анализов, в которых ошибки отсутствуют. *Примечание:* "глюки" *Intel Cilk Plus* исправлять не нужно.



The screenshot shows the Intel Inspector XE interface with the title "Detect Deadlocks and Data Races". The "Summary" tab is selected. A table lists five detected data race problems (P1-P5).

ID	Type	Sources	Modules	State
P1	Data race	list; reducer_vector.h	ips_prj_parallel.exe	New
P2	Data race	list; xmemory0; xutility	ips_prj_parallel.exe	New
P3	Data race	reducer_vector.h; utility; vector	ips_prj_parallel.exe	New
P4	Data race	reducer_vector.h; vector; xutility	ips_prj_parallel.exe	New
P5	Data race	list; vector; xmemory0; xutility	ips_prj_parallel.exe	New

Рисунок 9. Результат работы Inspector XE



The screenshot shows the Intel Inspector XE interface with the title "Detect Leaks". The "Summary" tab is selected. A table lists one detected memory leak problem (P1).

ID	Type	Sources	Modules	Object Size
P1	Memory not deallocated	exe_common.inl; streambuf; xlocinfo	ips_prj_parallel.exe	4532

Рисунок 10. Результат работы Inspector XE

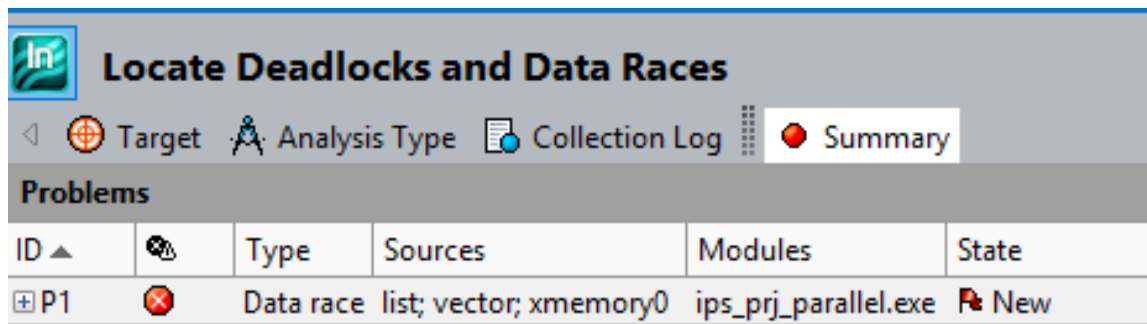


Рисунок 11. Результат работы Inspector XE

Из результатов анализа видно, что ошибки имеются, но они связаны с самой библиотекой Cilk, и на данный ошибки мы, как пользователи, повлиять никак не можем. К счастью данные ошибки не влияют на результат работы программы.

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.

В моём случае *X* равен 2, т.к. на моём ЦПУ только 2 ядра, поэтому возможны только 2 варианта: 1 и 2 ядра.

```
Number of workers 1
Duration is 0.016592 seconds
```

Рисунок 12. Результат вычисления программы с 1 ядром

```
Number of workers 2
Duration is 0.00282991 seconds
```

Рисунок 13. Результат вычисления программы с 2 ядрами

Из результатов видно, что программа с 2 ядрами выполняется быстрее, чем программа с 1 ядром.

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

```
Number of workers 2
Duration is 0.00935703 seconds
```

Рисунок 14. Результат выполнения программы при $\delta=0.1$

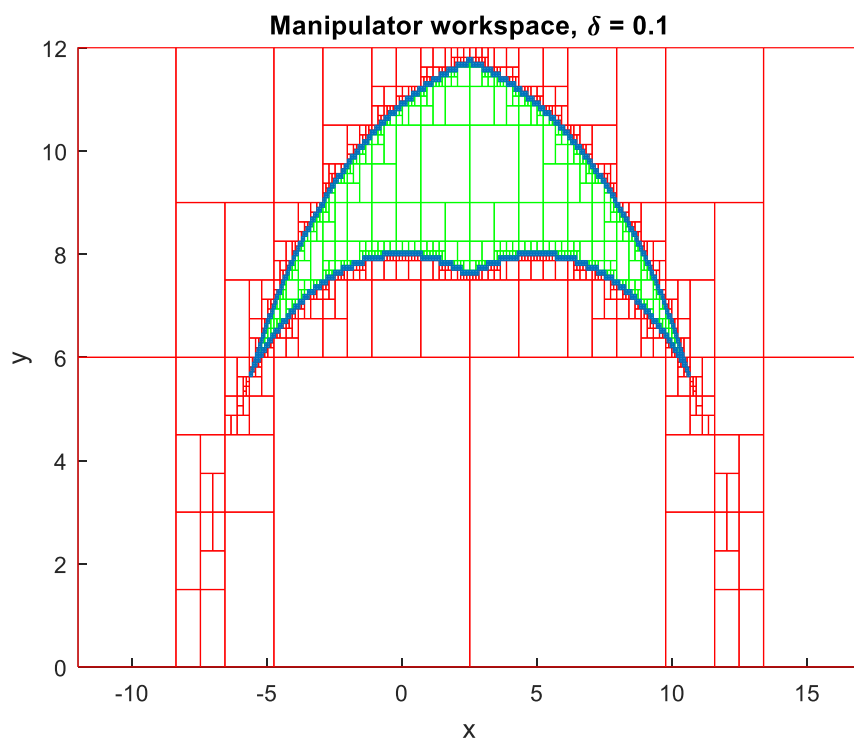


Рисунок 15. Результат выполнения скрипта при $\delta=0.1$

```
Number of workers 2
Duration is 0.714435 seconds
```

Рисунок 16. Результат выполнения программы при $\delta=0.001$

Выполнить скрипт не удалось, т.к. мне не хватило оперативной памяти для получения картинки, и операционная система успешно убила процесс вычисления Matlab.

Вывод: из результатов выполнения программы следует что её время исполнения напрямую зависит от точности.