



Софийски Университет “Св. Климент Охридски”
Факултет по Математика и Информатика

Паралелен тест на Манделброт
Оценки на товара при динамично и
статично балансиране

Радослав Каратанев 3MI0800036

Научни Ръководители:

проф. д-р Васил Цунижев
ас. Христо Христов

София, 2024

Съдържание

1. Цел на курсовия проект и въведение в задачата

- a. Цел на курсовия проект
- b. Какво е „тест на Манделброт“?
- c. Представяне на подобни източници
- d. OpenMP vs MPI

2. Проектиране

- a. Библиотека за визуализация
- b. Реализация със статично балансиране
 - 1) Функционален анализ
 - 2) Технологичен анализ
- c. Реализация със статично циклично балансиране
 - 1) Функционален анализ
 - 2) Технологичен анализ
- d. Реализация със динамично централизирано балансиране
 - 1) Функционален анализ
 - 2) Технологичен анализ
- e. Реализация със разпределено динамично балансиране
 - 1) Функционален анализ
 - 2) Технологичен анализ

3. Тестване

4. Заключение

5. Източници

1. Цел на Курсовия Проект и Въведение в Задачата

а. Цел на курсовия проект

Целта на този курсов проект е да се проучи натоварването при изпълнение на теста на Манделброт. Ще се анализира влиянието на статичното и динамичното балансиране върху постигнатото ускорение, като ще се направи сравнение между тях. Ще бъдат разгледани въпроси, свързани с грануларността, тоест как отделните задачи са разпределени спрямо големината си и броя на използваните нишки. Ще се изследва и адаптивността към L1 D-Cache, с цел да се установи дали размерът на всяка задача може да бъде намален достатъчно, за да се побере изцяло в L1 D-Cache.

Структурата на курсовия проект ще следва следните етапи:

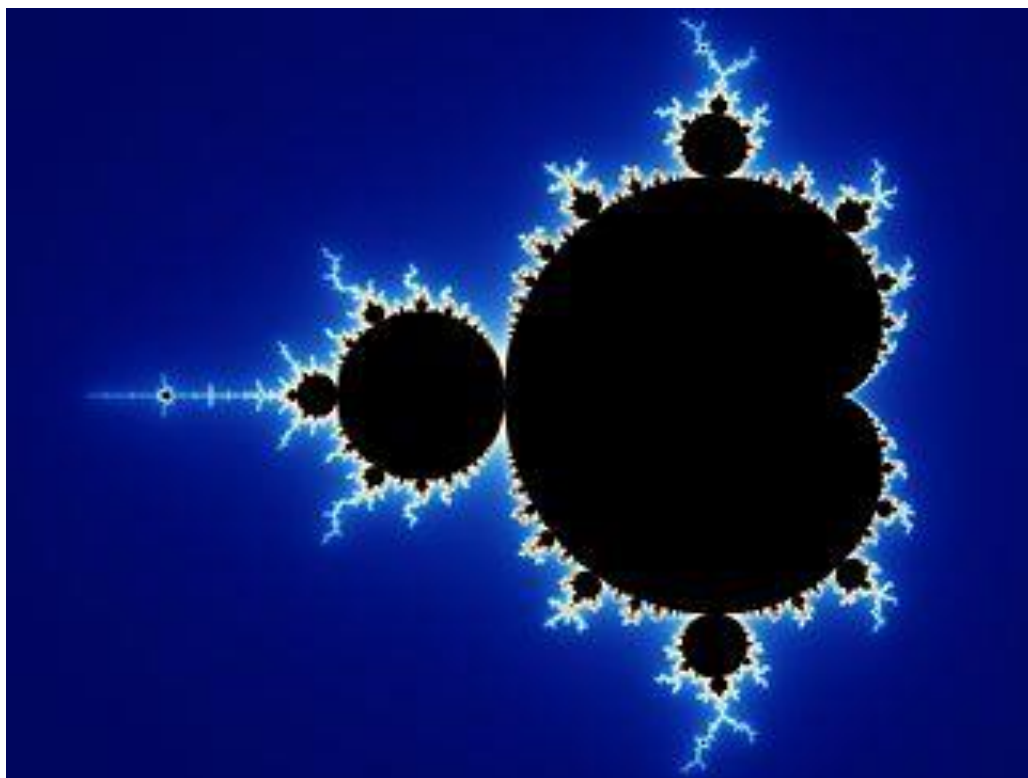
- **Обяснение:** Ще обясним точно каква е целта на курсовия проект.
- **Анализ:** Ще проучим и представим източници, които са максимално близки до поставената задача. Тези източници ще бъдат анализирани, за да се извлекат добрите и лошите практики, които ще следваме или избягваме в следващите етапи. Също ще разгледаме използваните технологии, езици и инструменти, необходими за решаването на проблема.
- **Проектиране:** В тази секция ще разгледаме въпросите по реализацията на проблема. Функционалностите на крайния продукт ще бъдат представени чрез "ръководство за потребителя", описващо необходимите входни параметри и начина на въвеждането им. Ще бъдат сравнени статичното и динамичното балансиране, за да се покаже кой вариант е по-подходящ и какви са предимствата и недостатъците на всеки. За двата случая ще се представят UML диаграми на последователността, които ще илюстрират стъпките на изпълнение на програмата и действията на различните класове. Ще се опишат и всички зависимости под формата на библиотеки, използвани от софтуера.
- **Тестване:** В тази секция ще бъде представен тестов план, включващ множество тестови случаи, целящи да изследват поведението на програмата при различни входни параметри. Целта е да се изчисли ускорението (speedup) в различни ситуации. Резултатите ще бъдат

визуализирани чрез графики и таблици, които ще послужат за формулиране на изводи относно разглежданите проблеми. В частта за внедряването ще се покажат изображения, резултат от програмата.

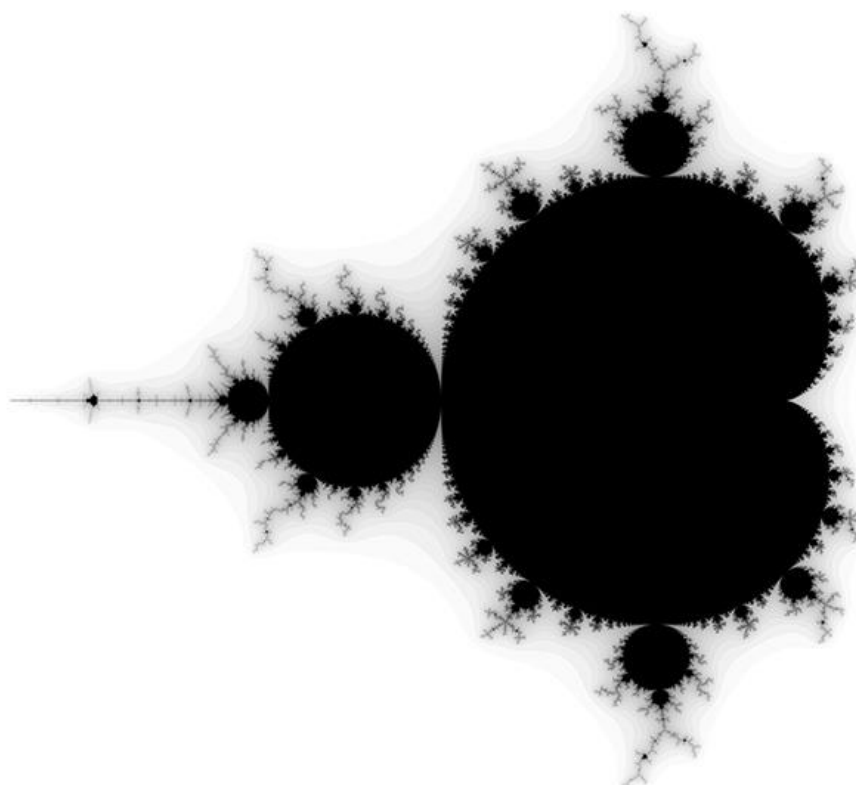
в. Какво е „тест на Манделброт“?

Тестът на Манделброт се отнася до компютърната симулация и визуализация на множеството на Манделброт, което е една от най-известните фрактални геометрични структури. За да разберем същността на теста на Манделброт, нека разгледаме няколко ключови аспекта:

- **Множеството на Манделброт** е дефинирано в комплексната равнина и се състои от точки, които *не* клонят към безкрайност при многократното прилагане на определена рекурсивна функция.
- **Функцията, която се използва за генерирането на множеството на Манделброт**, е $z_{n+1} = z_n^2 + c$, където z и c са комплексни числа. Започвайки от $z_0 = 0 + 0i$, функцията се прилага многократно. В този проект сме изследвали резултата при прилагане $n = 100, 500, 1000$ пъти.
- **Критерий за принадлежност** – За всяка точка c в комплексната равнина, ние следим последователността от стойности на z_n . Ако тази стойност остава ограничена (не клони към безкрайност), тогава точката c принадлежи на множеството на Манделброт. Ако последователността клони към безкрайност, то точката c *не* принадлежи на множеството.
- **Визуализация** - Резултатът от теста на Манделброт обикновено се визуализира чрез изобразяване на точки в комплексната равнина. За точки, които принадлежат на множеството, се използва един цвят (в нашия случай бяло), а за точки, които не принадлежат, се използват различни цветове в зависимост от скоростта на разпадане (след колко итерации стойностите на z_n клонят към безкрайност). Това създава характерния фрактален образ с богата структура и самоподобие на различни мащаби.



Пример за множество на Манделброт^[1]



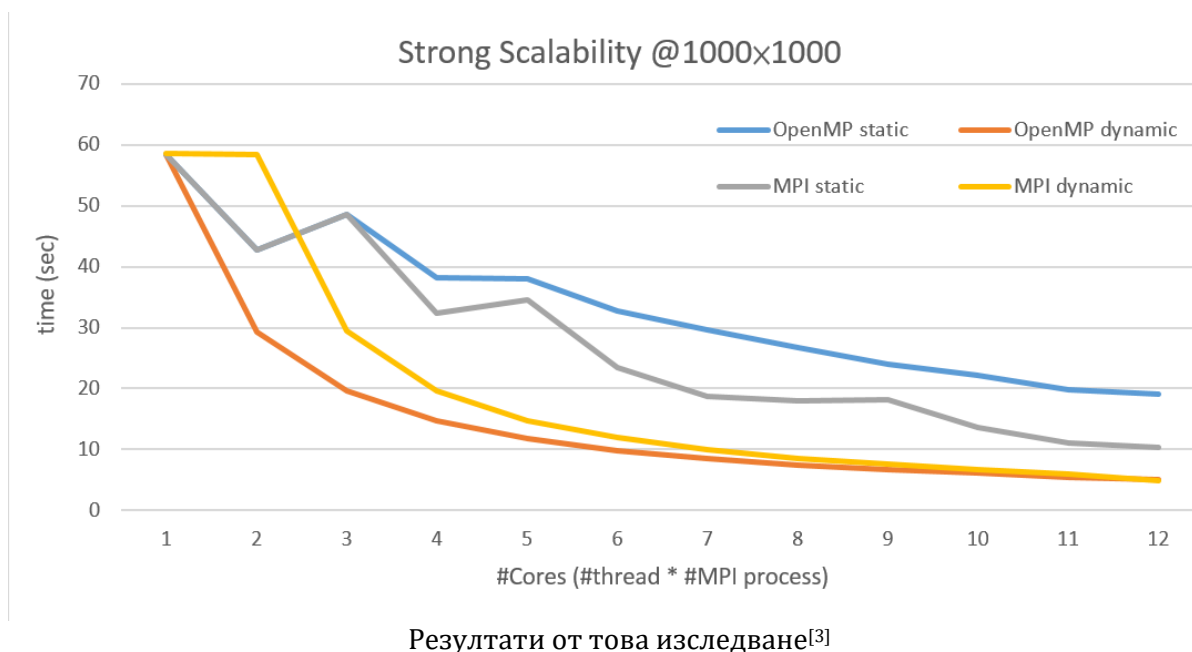
Пример за множество на Манделброт^[2]

с. Представяне на подобни източници

В този параграф ще представим проекта „Parallel Mandelbrot Set Calculation“[3].

Този проект има за цел да визуализира множеството на Манделброт чрез използване на паралелни изчислителни техники в C++: OpenMP и MPI. Проектът включва генериране на изображение на множеството, като се използват OpenMP за паралелизация на изчисленията на многонишкови системи и MPI за разпределение на задачите между множество изчислителни възли:

- Определяне на областта в комплексната равнина за изчисление на множеството на Манделброт
- Тестване със статично балансиране
- Тестване със динамично балансиране



Както можем да забележим от графиката, във всички примери, броят използвани нишки е обратно пропорционален с времето, което отнема за да се визуализира множеството на Манделброт, като динамичното балансиране с OpenMP consistently е по-бързо от останалите. Друго нещо, което можем да забележим е, че колкото повече нишки използваме, толкова по-малко ускорение ще получим от това да добавим още 1 нишка.

d. OpenMP vs MPI

Това е добър момент да обясним какво точно представляват OpenMP и MPI, и какви са разликите между тях.

Кратко описание на OpenMP:

- Позволява добавяне на паралелизъм в съществуващ код чрез специални директиви (препроцесорни команди), които указват кои части от кода трябва да се изпълняват паралелно.
- Автоматично разпределя работата между множество нишки, които работят паралелно на различни нива на процесора.
- Всички нишки споделят една и съща адресна памет, което улеснява достъпа до общи данни.

Кратко описание на MPI:

- Процесите обменят данни чрез изпращане и получаване на съобщения, което прави възможно паралелното изпълнение на задачи на различни машини.
- Всеки процес има своя собствена адресна памет и комуникацията между тях става чрез изпращане на съобщения.
- Позволява паралелизация както на една машина с много ядра, така и на съвкупност от машини.

Въпреки това, че двете технологии се използват за паралелно програмиране, имат и доста разлики:

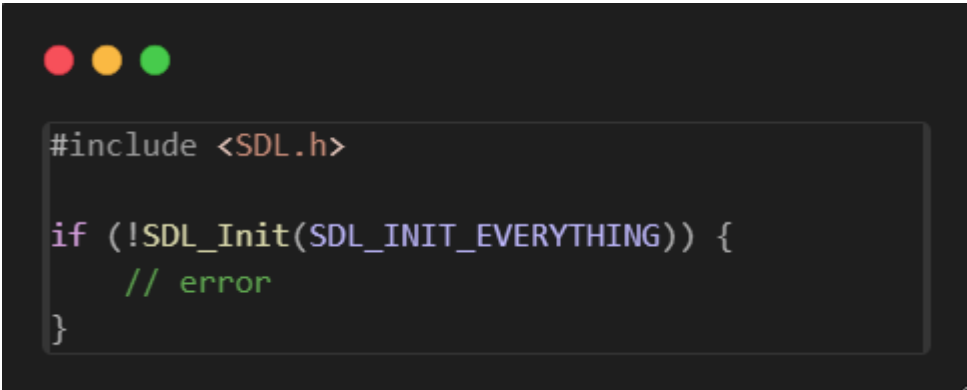
OpenMP	MPI
Използва споделена памет, където всички нишки имат достъп до една и съща област в паметта	Използва разпределена памет, където всеки процес има своя собствена памет и комуникацията става чрез съобщения
По-подходящ за паралелизация на една машина с множество ядра	По-подходящ за разпределени системи и съвкупност от машини
По-лесен за използване в съществуващи програми чрез добавяне на директиви	Изисква по-сложен код за управление на комуникацията между процесите

2. Проектиране

а. Библиотека за визуализация – SDL2^[4]

SDL2 (Simple DirectMedia Layer 2) е библиотека, която предоставя ниско ниво на достъп до аудио, клавиатура, мишка, джойстик и графичен хардуер чрез OpenGL и Direct3D. Използва се главно за разработка на игри, но ние ще я използваме единствено за визуализация на множеството на Манделброт. Тази библиотека се използва както следва:

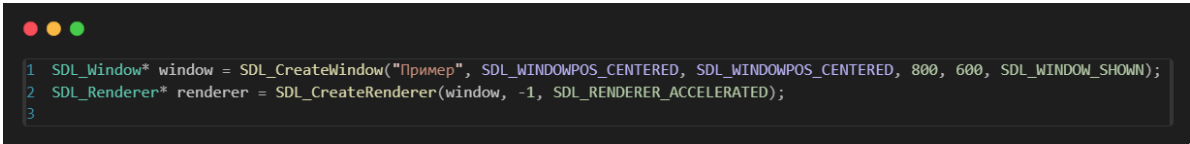
1) Инициализация

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light-colored font and is as follows:

```
#include <SDL.h>

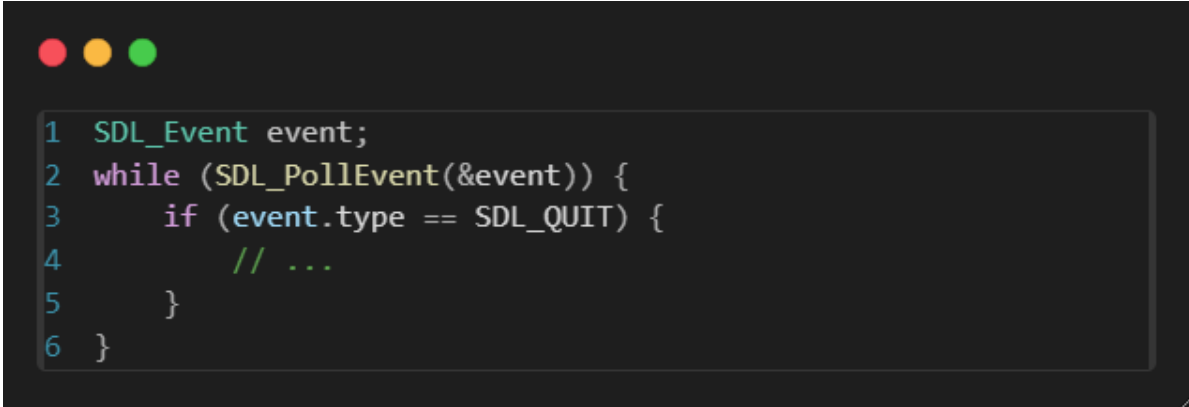
if (!SDL_Init(SDL_INIT_EVERYTHING)) {
    // error
}
```

2) Създаване на прозорец и рендерер

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light-colored font and is as follows:

```
1 SDL_Window* window = SDL_CreateWindow("Пример", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800, 600, SDL_WINDOW_SHOWN);
2 SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
3
```

3) Обработка на събития

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light-colored font and is as follows:

```
1 SDL_Event event;
2 while (SDL_PollEvent(&event)) {
3     if (event.type == SDL_QUIT) {
4         // ...
5     }
6 }
```


4) Рисуване

```
1 SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);  
2 SDL_RenderClear(renderer);  
3  
4 SDL_RenderPresent(renderer);
```

5) Освобождаване на ресурси

```
1 SDL_DestroyRenderer(renderer);  
2 SDL_DestroyWindow(window);  
3 SDL_Quit();
```

След изпълнението на тези стъпки, на екрана на потребителя излиза нов прозорец, с име „Пример“ (поддържа кирилица), на който трябва да нарисуваме множеството на Манделброт. За целта, ние ще променяме 4) с желаните ни цвят и позиция на екрана.

в. Реализация със статично балансиране

1) Функционален анализ

В този подход, ние разделяме цялото изображение на толкова части, колкото нишки сме заделили за обработката му. Това намаля времето за генериране, но не е оптимален подход. Някои нишки извършват повече изчисления от други, тоест времето ни за генериране на цялото изображение е ограничено от времето, нужно на най-натоварената нишка.

2) Технологичен анализ

```

1 void* workerFunction(void* arg) {
2     int threadID = *(int*)arg;
3     int width = WIDTH;
4     int height = HEIGHT;
5     double scale_real = 3.5 / width;
6     double scale_imag = 2.0 / height;
7
8     int chunkSize = HEIGHT / NUM_THREADS;
9     int startY = threadID * chunkSize;
10    int endY = (threadID == NUM_THREADS - 1) ? HEIGHT : startY + chunkSize;
11
12    for (int y = startY; y < endY; ++y) {
13        for (int x = 0; x < width; ++x) {
14            std::complex<double> c((x - width / 2) * scale_real - 0.5, (y - height / 2) * scale_imag);
15            int iterations = mandelbrot(c);
16
17            int color = 255 * iterations / MAX_ITERATIONS;
18            pixelBuffer[y][x] = color;
19        }
20    }
21
22    return nullptr;
23 }

```

Това е функцията, която разделя цялото изображение на части. Функцията *mandelbrot(std::complex<double> c)*, получава като параметър комплексно число, и ни казва дали е в границите на множеството на Манделброт при дадения брой итерации (*MAX_ITERATIONS*). Спрямо резултатът от тази функция, се определя какъв да е цвета на съответния пиксел, и се запазва в матрица.

```

1 void renderMandelbrot() {
2     pthread_t workers[NUM_THREADS];
3     int threadIDs[NUM_THREADS];
4
5     for (int i = 0; i < NUM_THREADS; ++i) {
6         threadIDs[i] = i;
7         pthread_create(&workers[i], nullptr, workerFunction, &threadIDs[i]);
8     }
9
10    for (int i = 0; i < NUM_THREADS; ++i) {
11        pthread_join(workers[i], nullptr);
12    }
13 }

```

Горното е функцията, създаваща нужния брой нишки. Използваме библиотеката *<pthread>* от *STL*.

с. Реализация със статично циклично балансиране

Няма да описваме този подход подробно поради това, че е аналогичен на предишния. Отново разделяме изображението на определен брой части, като разликата тук е, че частите са повече от броя нишки, водейки до по-фина гранулярност. Всяка нишка обработва предварително определени части от изображението. Пример:

Нишка 1 обработва отрязък 1, 5, 9, 13..

Нишка 2 обработва отрязък 2, 6, 10, 14...

Нишка 3 обработва отрязък 3, 7, 11, 15...

Нишка 4 обработва отрязък 4, 8, 12, 16...

d. Реализация със динамично централизирано балансиране

1) Функционален анализ

Този подход отново разделя изображението на малки части, повече на брой от броя използвани нишки. Ако допуснем, че са заделени n на брой нишки за визуализацията, то $n-1$ от тях таботят от изчисленията. Останалата нишка върши работата на „разпределител“. Когато даден „работник“ нишка завърши със своя отрязък от изображението, „разпределителят“ го пренасочва до първия свободен отрязък за изчисление. Това би следвало да доведе до големи ускорения при изчисленията, като общата работа е разделена по-ефективно и с по-фина гранулярност.

2) Технологичен анализ

Тук функцията ни за разпределение на работниците е значително по-сложна. Главната идея на останалата част от кода е същата като в предишните примери. Разликата тук е, че е нужен „мютекс“ за разпределението на нишките от разпределителя. Това се налага, защото има случаи, в които 2 нишки завършат работа по своя участък по едно и също време. Това може да създаде проблеми при разпределителната нишка, което ще се отрази негативно на нужното време за финалната визуализация.

```
1 void* workerFunction(void* arg) {
2     while (true) {
3         Task task;
4
5         // Lock and pull a task from the queue
6         pthread_mutex_lock(&queueMutex);
7         while (taskQueue.empty()) {
8             pthread_cond_wait(&queueCV, &queueMutex);
9         }
10
11        task = taskQueue.front();
12        taskQueue.pop();
13        pthread_mutex_unlock(&queueMutex);
14
15        // Process the task
16        processTask(task);
17
18        // Exit condition
19        pthread_mutex_lock(&queueMutex);
20        if (taskQueue.empty()) {
21            pthread_mutex_unlock(&queueMutex);
22            break;
23        }
24        pthread_mutex_unlock(&queueMutex);
25    }
26
27    return nullptr;
28 }
```

е. Реализация със разпределено динамично балансиране

1) Функционален анализ

При този подход отново използваме разделяне на изображението на малки, но многобройни части. Тук всичките ни използвани нишки са „работници“. Пазим глобална променлива *currentTaskIndex*, към която се обръща дадена нишка, след като завърши работа по своя участък от изображението. Тази променлива казва коя е първата свободна (и все още неизчислена) част от изображението, след което четящата нишка започва работа по нея.

2) Технологичен анализ

Естествено, тук също ни е нужен „мютекс“ в случай, че 2 нишки се опитат да достъпят променливата по едно и също време

```
1 void* workerFunction(void* arg) {
2     while (true) {
3         Task task;
4
5         // Lock and get the next task index
6         pthread_mutex_lock(&taskMutex);
7         if (currentTaskIndex >= tasks.size()) {
8             pthread_mutex_unlock(&taskMutex);
9             break;
10        }
11        task = tasks[currentTaskIndex];
12        currentTaskIndex++;
13        pthread_mutex_unlock(&taskMutex);
14
15        // Process the task
16        processTask(task);
17    }
18
19    return nullptr;
20 }
```

3. Тестване

Тестването на всичките изброени варианти от 2. е извършено както и на сървъра gmi.yaht.net, както и на настолен компютър с хардуер:

Device specifications

Device name	DESKTOP-ULNGVBK
Processor	Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz 2.70 GHz
Installed RAM	8,00 GB
Device ID	67D12080-0BFD-4834-B88B-1DDAB00CFBAA
Product ID	00326-10000-00000-AA041
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

На операционната система Windows 10.

Легенда за таблицата:

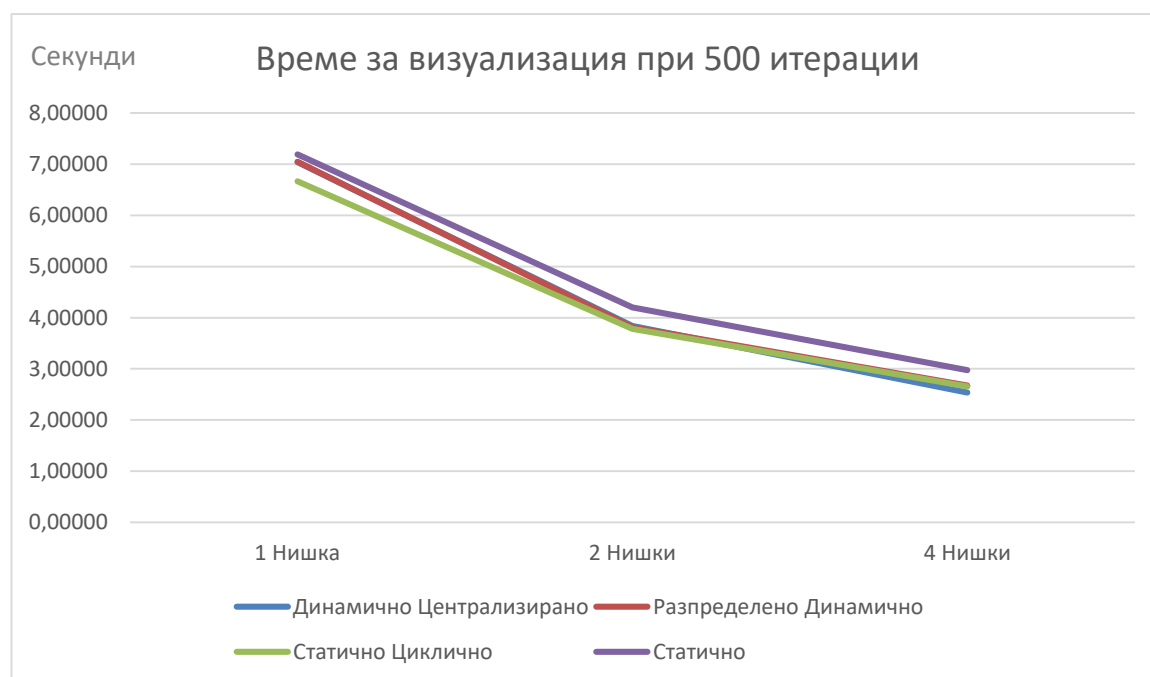
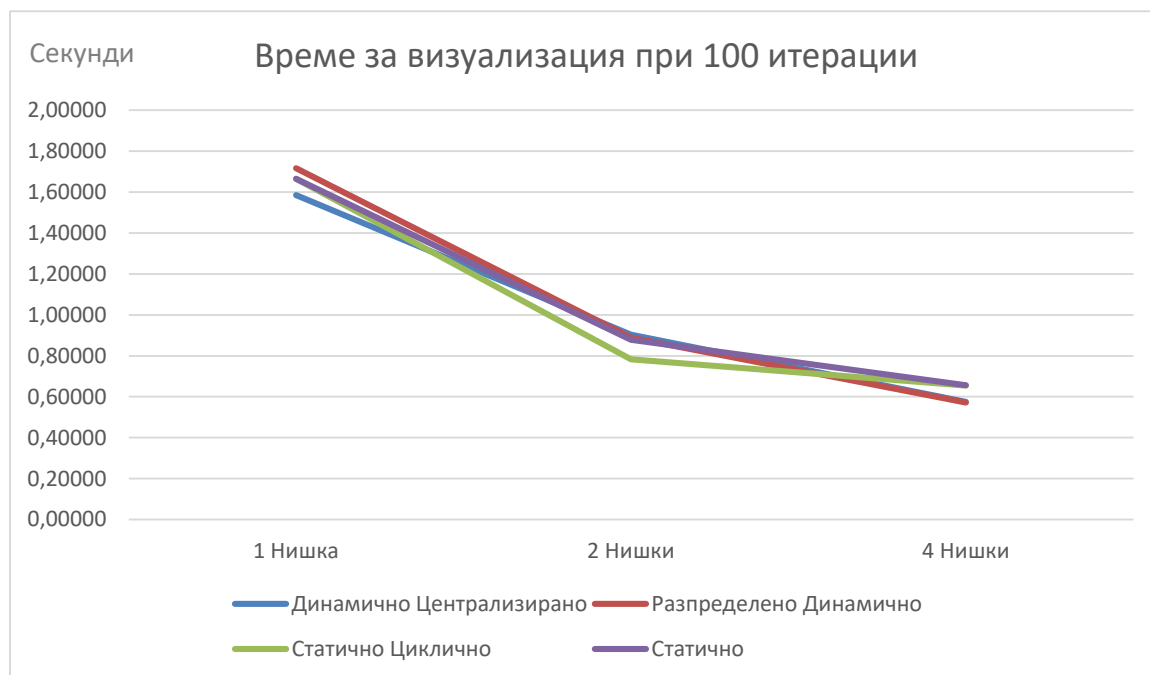
- **“N итерации”** – Броят проверени итерации, за програмата да разбере дали дадената точка се намира в множеството на Манделброт
- **„Размер N”** – Размерът на отделните очастъци, които трябва да опработни дадена нишка
- **„1 2 4”** – Броят нишки, използван в съответния тест

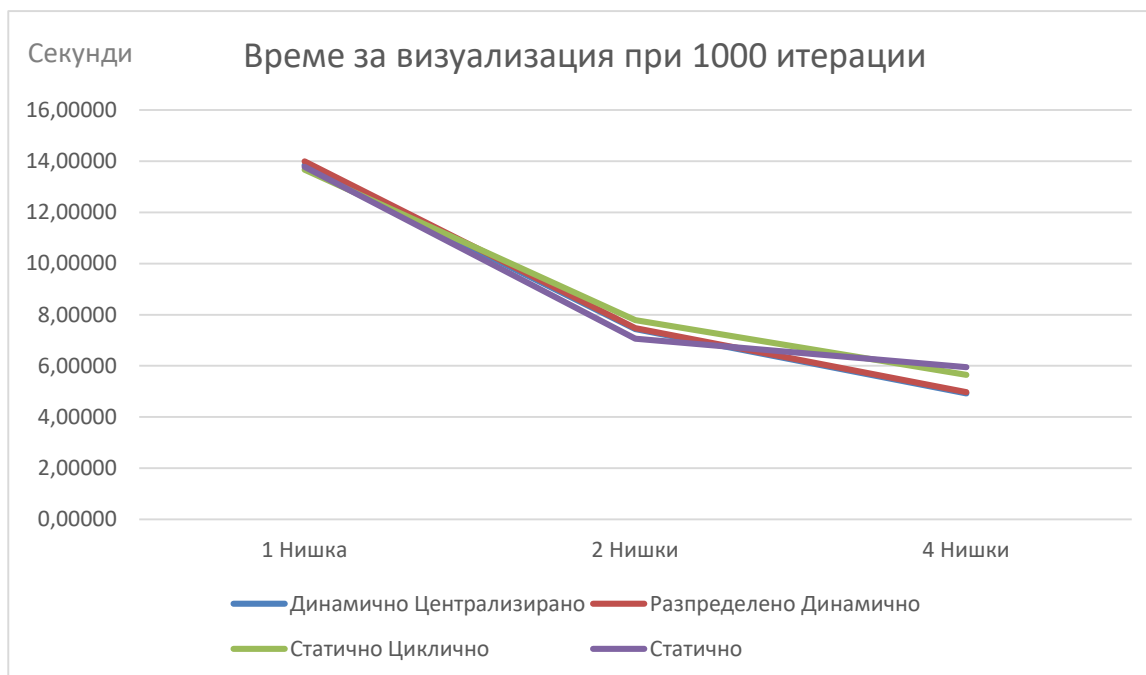
Следните данни са измерени на настолния компютър:

			1	2	4
100 итерации	Динамично Централизирано	Размер 10	1.69492	0.899847	0.445
		Размер 50	1.65146	0.926369	0.459
		Размер 100	1.69651	0.911227	0.456002
		Размер 200	1.68563	0.905517	0.590987
		Размер 400	1.77151	0.874977	0.921959
	Разпределено Динамично	Размер 10	1.65251	0.86551	0.447997
		Размер 50	1.76134	0.858035	0.458002
		Размер 100	1.73047	0.864376	0.477001
		Размер 200	1.80942	0.874046	0.610279

		Размер 400	1.62707	0.988001	0.869081
	Статично Циклично	-	1.66432	0.78253	0.65402
	Статично	-	1.66528	0.878251	0.656026
500 итерации	Динамично Централизирано	Размер 10	7.02843	3.77618	1.94811
		Размер 50	6.92717	3.76063	2.04779
		Размер 100	7.21333	3.80519	1.99463
		Размер 200	6.97698	3.89261	2.83072
		Размер 400	7.04732	3.96435	3.86976
	Разпределено Динамично	Размер 10	7.25261	3.66977	2.00503
		Размер 50	7.02902	3.80501	2.008
		Размер 100	7.00407	3.72144	2.045
		Размер 200	7.01949	3.94438	3.27688
		Размер 400	6.93523	3.93271	4.04199
	Статично Циклично	-	6.66432	3.78253	2.65402
	Статично	-	7.19333	4.20329	2.97492
1000 итерации	Динамично Централизирано	Размер 10	13.584	7.12534	3.78769
		Размер 50	13.5703	7.17719	3.97164
		Размер 100	13.6987	7.22222	3.81953
		Размер 200	13.828	7.95817	5.42743
		Размер 400	14.536	7.72972	7.62944
	Разпределено Динамично	Размер 10	13.9302	7.31071	3.87211
		Размер 50	14.2414	7.22939	3.83063
		Размер 100	13.9563	7.25006	3.83398
		Размер 200	13.8694	7.64663	5.60344
		Размер 400	13.9954	7.91113	7.71831
	Статично Циклично	-	13.66432	7.78253	5.65402
	Статично	-	13.7758	7.06768	5.94342

- **Графика на ускорението при по-голям брой нишки**





4. Заключение

От таблицата и графиките ясно може да видим, че при всички варианти увеличаването на броя нишки драстично ускорява процеса. Нещо интересно е това, че вида балансиране няма толкова голямо значение, колкото очаквахме. Във всички случаи, времето за рисуване на цялото изображение е близо до еднакво между видовете балансиране.

5. Източници

- [1] https://en.wikipedia.org/wiki/Mandelbrot_set
- [2] <https://paulbourke.net/fractals/mandelbrot/>
- [3] <https://github.com/e-bug/parallel-mandelbrot-set>
- [4] <https://www.libsdl.org/>