به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

برنامه نویسی موازی

گزارش کار پروژه‌ی دوم

موازی سازی در سطح نخ (OpenMP)

نام و نام خانوادگی	امیرحسین ثمودی – آرمین قاسمی
شماره دانشجویی	810100198 – 810100108
تاریخ ارسال گزارش	1403/8/29

فهرست گزارش سوالات

سوال اول: Mandelbrot Set

در این سوال قصد داریم نواحی در صفحه اعداد مختلط که با قرار دادن C در آنها و اعمال معادله $z = z^2 + c$ بر آن (با مقدار اولیه z=0) همگرا میشود را بیابیم که به Mandelbrot Set شناخته میشود.

پیاده سازی سریال:

در پیاده سازی سریال ابتدا با توجه به مقدار اولیه C که برای ZoomIn و ZoomOut شدن تصویر خروجی مشخص میشود یک بازه در راستای محور های حقیقی و موهومی برای تغییرات C در نظر میگیریم:

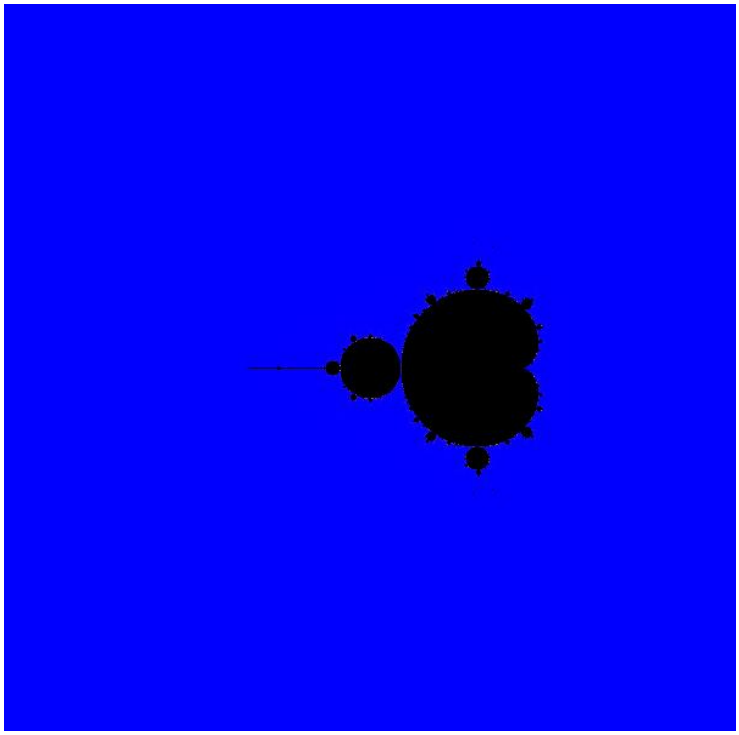
```
double realMin = -2.0 * initialC/4, realMax = 1.0 * initialC/4;
double imagMin = -1.5 * initialC/4, imagMax = 1.5 * initialC/4;
cv::Mat image(height, width, CV_8UC3);
```

سپس برای تمام پیکسل هایی از تصویر خروجی که در OpenCV باز کرده ایم مقدار متناظر حقیقی و موهومی C را بدست میآوریم (دوحلقه تو در تو برای در بر گرفتن همه نقاط C داریم). سپس به ازای هر نقطه C الگوریتم را که همان فرمول ذکر شده در بالا هست را به تعداد iteration ها تکرار میکنیم (در صورتی که اندازه Z از 2 بزرگتر شود ثابت میشود که دیگر رابطه همگرا نمیشود و لوپ را متوقف مینماییم.

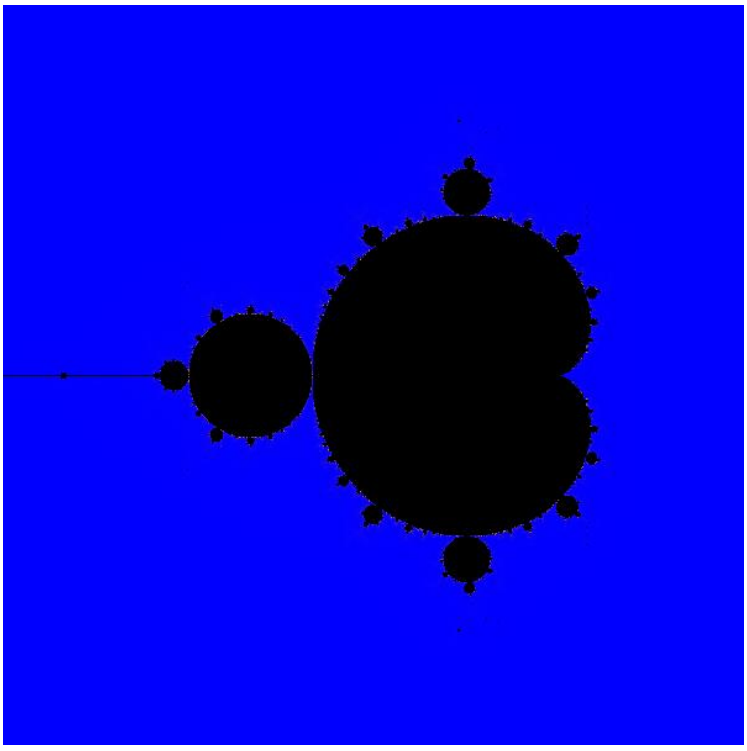
سپس با توجه به تعداد iteration ها و تعیین 3 رنگ R, G, B میتوان طیف خروجی را رسم کرد. به این صورت که هر چه تعداد تکرار ها به بیشینه iteration ها نزدیک تر باشد یعنی با احتمال خیلی بالاتری آن نقطه اولیه برای C موجب همگرا شدن Z میشود(مولفه های رنگی به مشکی نزدیک تر شده) و هر چه iteration کوچک تر و به صفر نزدیک تر باشد مولفه Blue بزرگتر شده و آن پیکسل به رنگ آبی در می آید. کد الگوریتم اصلی به صورت زیر میباشد:

```
for (int y = 0; y < height; ++y) {
    imag = imagMin + (y / static_cast<double>(height)) * (imagMax - imagMin);
    for (int x = 0; x < width; ++x) {
        real = realMin + (x / static_cast<double>(width)) * (realMax - realMin);
        c = std::complex<double>(real, imag);
        z = 0;

        iteration = 0;
        for (iteration = 0; iteration < maxIterations; ++iteration) {
            z = z * z + c;
            if (std::abs(z) > 2.0) {
                break;
            }
        }

        if (iteration == maxIterations) {
            image.at<cv::Vec3b>(y, x) = cv::Vec3b(0, 0, 0);
        }
        else {
            r = static_cast<int>(255 * (iteration / static_cast<double>(maxIterations)));
            g = static_cast<int>(128 * (iteration / static_cast<double>(maxIterations)));
            b = 255 - r;
            image.at<cv::Vec3b>(y, x) = cv::Vec3b(b, g, r);
        }
    }
}
```

درنهایت تصویر خروجی به صورت زیر در می آید:



خروجی برنامه سریال برای c=8j



خروجی برنامه سریال برای c=4j

مشاهده میکنیم که با افزایش C اولیه Zoom Out اتفاق می افتد

پیاده سازی موازی:

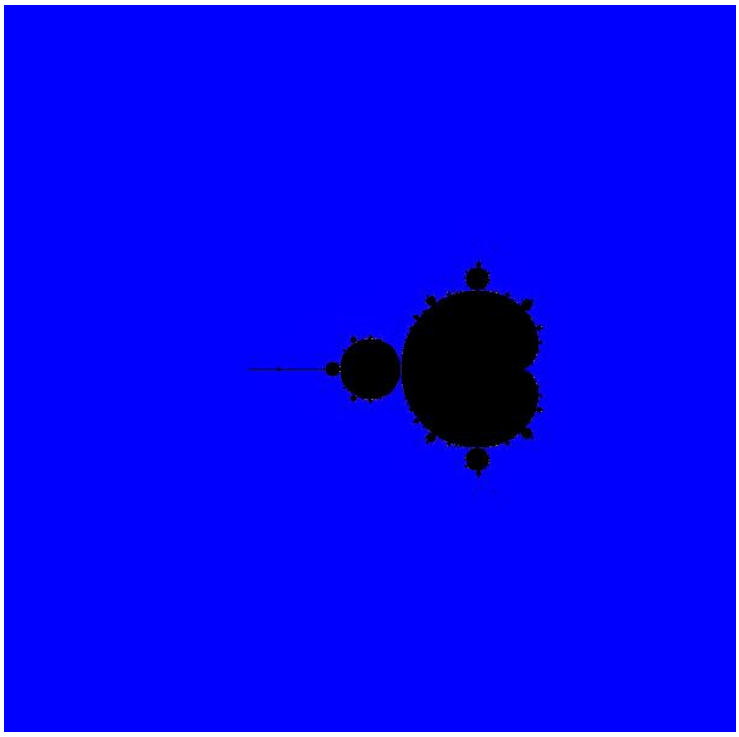
برای موازی سازی این برنامه در سطح Thread از ساختار های OpenMP استفاده میکنیم. با کمک Directive زیر اینکار را انجام میدهیم.

```
#pragma omp parallel for schedule(dynamic) num_threads(6) private(real, imag,
c, z, iteration, r, g, b) shared(image, realMin, realMax, imagMin, imagMax,
                                        width, height, maxIterations)
```

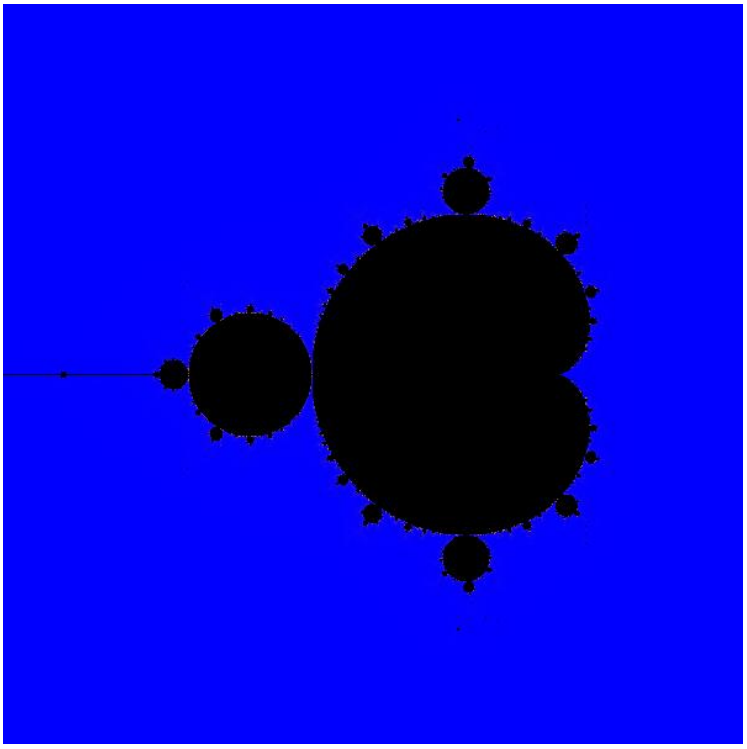
توجه شود که در اینجا با محاسبه SpeedUp در چند حالت Scheduling مختلف به این نتیجه رسیدیم که Dynamic Scheduling علارغم سربار زمانی که نسبت به حالت Static دارد افزایش سرعت بهتری میدهد (البته استفاده از Static با سایز پک های کوچک حدود 5 نیز SpeedUp مشابهی را منجر میشد).

تعداد Thread ها نیز که 6 مشخص شده با توجه به سیستم فعلی که کد روی آن اجرا شده است میباشد که Efficiency مناسبی را ارائه میداد.

درنهایت تصویر خروجی به صورت زیر در می آید:



خروجی برنامه موازی برای c=8j



خروجی برنامه موازی برای c=4j

مشاهده میکنیم که خروجی ها مشابه حالت سریال است. همچنین در اینجا نیز با افزایش C اولیه Zoom Out اتفاق می افتد.

محاسبه SpeedUp:

برای بدست آوردن SpeedUp میانگین چندین بار خروجی گرفتیم:

```
Mandelbrot Set image saved done.
Serial clock cycles: 5499017652
Mandelbrot Set parallel done.
Parallel clock cycles: 1286834784
Speedup: 4.27329
```

```
Mandelbrot Set image saved done.
Serial clock cycles: 5752729444
Mandelbrot Set parallel done.
Parallel clock cycles: 1497149464
Speedup: 3.84246
```

```
Mandelbrot Set image saved done.
Serial clock cycles: 5572512241
Mandelbrot Set parallel done.
Parallel clock cycles: 1322887941
Speedup: 4.21238
```

```
Mandelbrot Set image saved done.
Serial clock cycles: 5412292056
Mandelbrot Set parallel done.
Parallel clock cycles: 1274230866
Speedup: 4.2475
```

$$Average\ SpeedUp = 4.1439$$

مشاهده میکنیم که با استفاده از 6 Thread به SpeedUp حدود 4.14 رسیدیم که مطلوب است.

سوال دوم: Julia Set

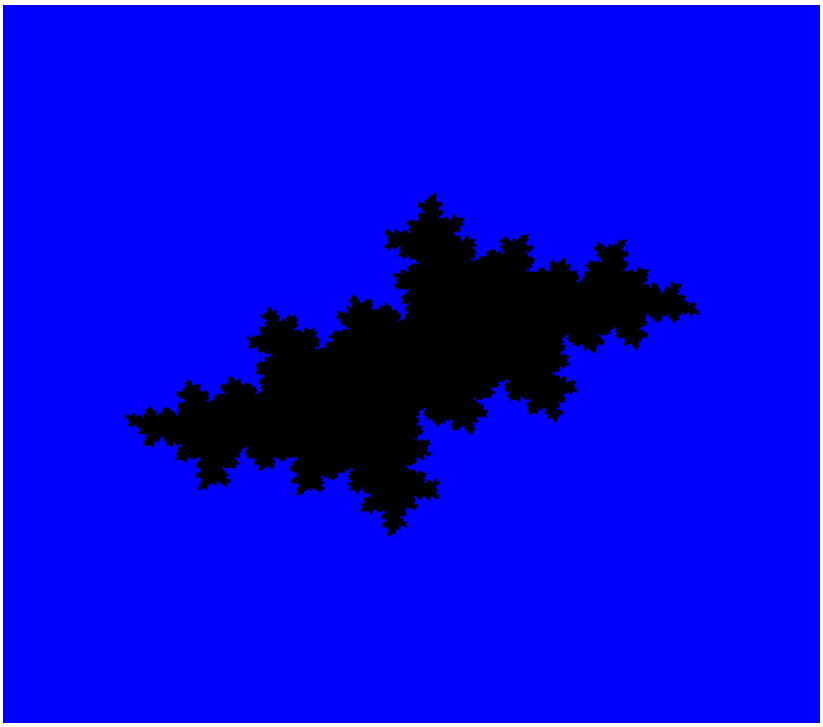
این سوال مانند سوال اول است با این تفاوت که مقدار C را ثابت قرار میدهیم و با اعمال معادله $z = z^2 + c$ بر آن نقاطی که منجر به همگرا شدن Z میشود را میابیم که به نام Julia Set شناخته میشود.

پیاده سازی سریال:

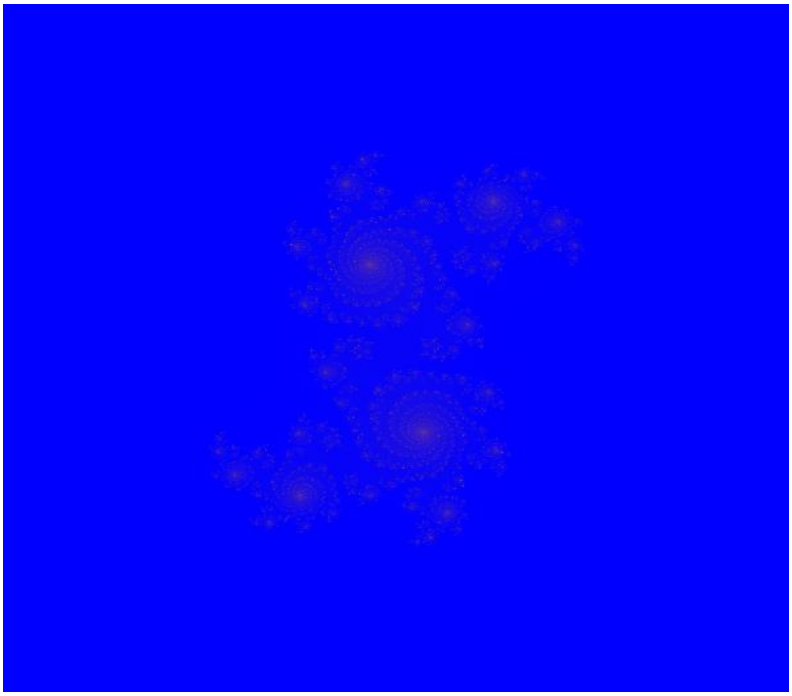
در پیاده سازی سریال مانند الگوریتم سوال اول عمل میکنیم با این تفاوت که پیکسل هایی از صفحه را که به مقادیر حقیقی و موهومی متناظرشان مپ میکنیم را برابر مقدار اولیه Z میگذاریم.(مقدار اولیه C را نیز ابتدای الگوریتم به صورت ثابت مشخص میکنیم. کد آن به صورت زیر میباشد:

```cpp
void JuliaSetSerial(int width, int height, int maxIterations, const std::string& filename, std::complex<double> c) {
    double realMin = -2.0, realMax = 2.0;
    double imagMin = -2.0, imagMax = 2.0;
    cv::Mat image(height, width, CV_8UC3);

    double real, imag;
    std::complex<double> z;
    int iteration;
    int r, g, b;

    for (int y = 0; y < height; ++y) {
        imag = imagMin + (y / static_cast<double>(height)) * (imagMax - imagMin);
        for (int x = 0; x < width; ++x) {
            real = realMin + (x / static_cast<double>(width)) * (realMax - realMin);
            z = std::complex<double>(real, imag);

            iteration = 0;
            for (iteration = 0; iteration < maxIterations; ++iteration) {
                z = z * z + c;
                if (std::abs(z) > 2.0) {
                    break;
                }
            }
            if (iteration == maxIterations) {
                image.at<cv::Vec3b>(y, x) = cv::Vec3b(0, 0, 0); // Black
            }
            else {
                r = static_cast<int>(255 * (iteration / static_cast<double>(maxIterations)));
                g = static_cast<int>(128 * (iteration / static_cast<double>(maxIterations)));
                b = 255 - r;
                image.at<cv::Vec3b>(y, x) = cv::Vec3b(b, g, r); // Blue
            }
        }
    }
```

خروجی برنامه سریال به ازای C های مختلف به صورت زیر میباشد:



خروجی برنامه سریال Julia Set با C =-0.5+0.5j



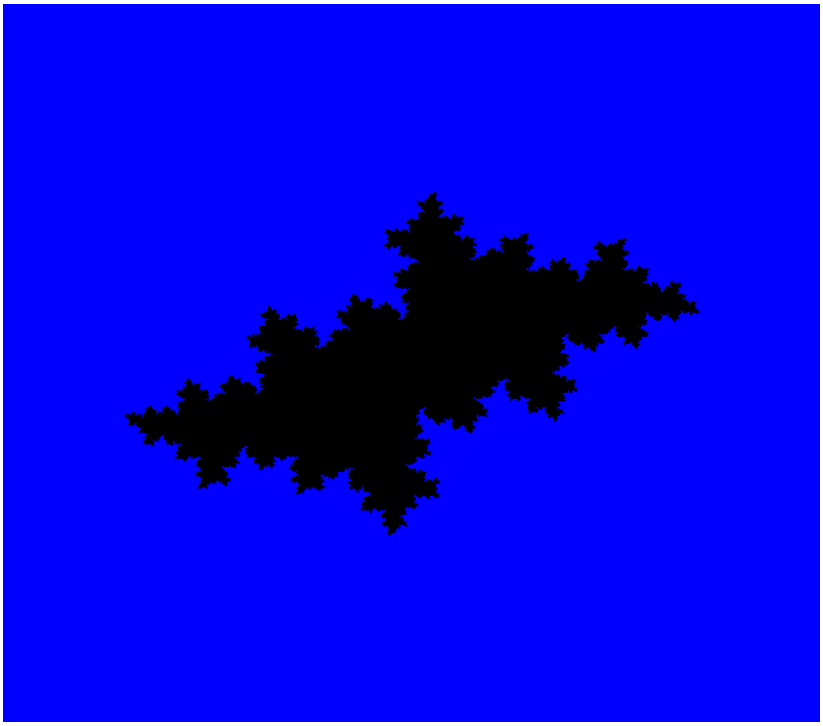
خروجی برنامه سریال Julia Set با C =0.355+0.355j

پیاده سازی موازی:

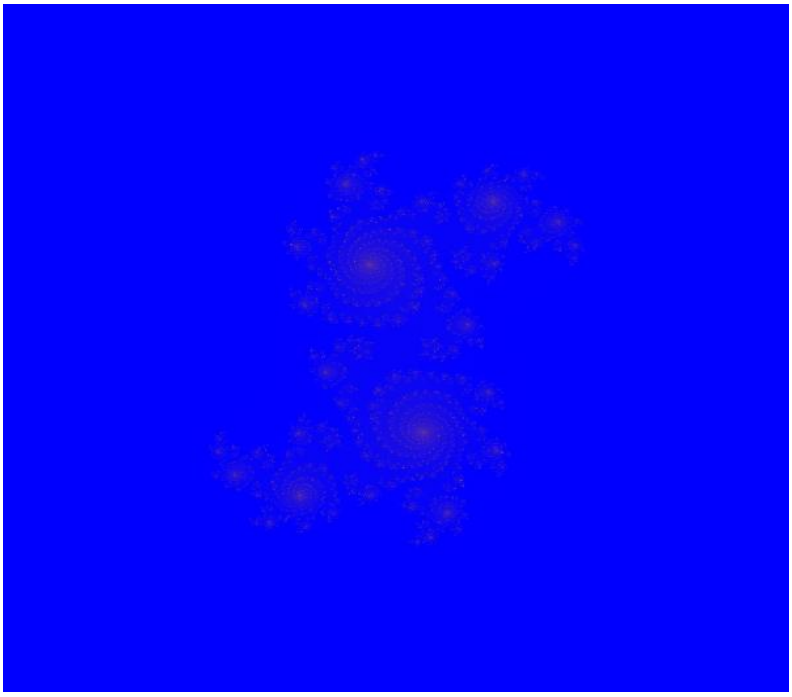
پیاده سازی موازی نیز دقیقا مانند سوال قبل میباشد (از یک بدنه پارالل با Loop WorkSharing Construct) استفاده نموده ایم. در این سوال نیز توجه شود که با محاسبه SpeedUp در چند حالت Scheduling مختلف به این نتیجه رسیدیم که Dynamic Scheduling علارغم سربار زمانی که نسبت به

حالت Static دارد افزایش سرعت بهتری میدهد (البته استفاده از Static با سایز پک های کوچک حدود 5 نیز SpeedUp مشابهی را منجر میشد).

خروجی ها به صورت زیر میباشند (مشابه حالت سریال):



خروجی برنامه موازی Julia Set با C =-0.5+0.5j



خروجی برنامه موازی Julia Set با C =0.355+0.355j

محاسبه SpeedUp:

برای بدست آوردن SpeedUp میانگین چندین بار خروجی گرفتیم:

```
Julia Set image done.
Serial clock cycles: 1496205142
Julia Set parallel image done.
Parallel clock cycles: 379725262
Speedup: 3.94023
```

```
Julia Set image done.
Serial clock cycles: 1450889300
Julia Set parallel image done.
Parallel clock cycles: 378512565
Speedup: 3.83313
```

```
Julia Set image done.
Serial clock cycles: 1411396840
Julia Set parallel image done.
Parallel clock cycles: 331361413
Speedup: 4.25939
```

```
Julia Set image done.
Serial clock cycles: 1499665088
Julia Set parallel image done.
Parallel clock cycles: 381773422
Speedup: 3.92815
```

```
Julia Set image done.
Serial clock cycles: 1492692964
Julia Set parallel image done.
Parallel clock cycles: 378940823
Speedup: 3.93912
```

$$Average\ SpeedUp = 3.98$$

مشاهده میشود که با استفاده از 6 Thread به SpeedUp حدود 4 رسیدیم.

سوال سوم: تخمین pi با روش Monte Carlo

تابع محاسبه عدد TT با روش Monte Carlo به صورت سریال:

```
double monteCarloPiSerial(int numPoints, double radius) {
    int pointsInsideCircle = 0;

    // Random number generator for serial computation
    mt19937 generator(std::random_device{}());
    uniform_real_distribution<double> distribution(-radius, radius);

    for (int i = 0; i < numPoints; ++i) {
        double x = distribution(generator);
        double y = distribution(generator);

        if (x * x + y * y <= radius * radius) {
            ++pointsInsideCircle;
        }
    }
```

```
    return 4.0 * pointsInsideCircle / numPoints;
}
```

در این تابع تعداد نقاط (numPoints) و شعاع دایره (معادل با نصف ضلع مربع : radius) به عنوان پارامتر ورودی دریافت شده است.

متغییر pointsInsideCircle تعداد نقاط داخل دایره را میشمارد که در ابتدا مقدار اولیه صفر را گرفته است.

```
    mt19937 generator(std::random_device{}());
    uniform_real_distribution<double> distribution(-radius, radius);
```

این دو خط برای تولید اعداد تصادفی(با کیفیت بالا) به کار میروند. دلیل استفاده از این روش هماهنگی کد سریال با کد موازی است. در کد موازی برای بهتر درست کردن اعداد رندوم بهتر است از این روش استفاده کنیم که در بخش کد موازی ذکر خواهد شد.

در حلقه for اعداد تصادفی در بازه مورد نظر تولید خواهند شد و اگر درون دایره افتاده باشند (این مورد را از رابطه فیثاغورس میفهمیم) pointsInsideCircle یک واحد افزایش خواهد یافت.

نهایتا pointsInsideCircle تقسیم بر numPoints می شود و در 4 ضرب میشود و به عنوان عدد π تخمین زده شده برگردانده میشود.

تابع محاسبه عدد π با روش Monte Carlo به صورت موازی:

```
double monteCarloPiParallel(int numPoints, double radius) {
    int pointsInsideCircle = 0;

    #pragma omp parallel
    {
        // Create a thread-local random number generator
        mt19937 generator(std::random_device{}() + omp_get_thread_num());
        uniform_real_distribution<double> distribution(-radius, radius);
        int localCount = 0;

        #pragma omp for
        for (int i = 0; i < numPoints; ++i) {
            double x = distribution(generator);
            double y = distribution(generator);

            if (x * x + y * y <= radius * radius) {
                ++localCount;
            }
        }

        #pragma omp atomic
```

```
        pointsInsideCircle += localCount;
    }

    return 4.0 * pointsInsideCircle / numPoints;
}
```

در این تابع هم مراحل اولیه مانند حالت سریال طی میشود.

دقت کنیم که دو خط :

```
        mt19937 generator(std::random_device{}() + omp_get_thread_num());
        uniform_real_distribution<double> distribution(-radius, radius);
```

درون بدنه parallel قرار دارند این عمل به این دلیل است که مطمئن شویم thread ها اعداد تصادفی مختص خود را تولید میکنند.

سپس هر thread برای خود یک localCount تعریف می کند و و نقاط تصادفی که خودش تولید کرده و داخل دایره افتاده را میشمارد .

نهایتا در یک atomic construct تمام localCount ها را باهم جمع می کنیم و pointsInsideCircle را بدست می آوریم و برمیگردانیم.

تابع main():

```
int main() {
    int numPoints;
    double radius;

    cout << "Enter the number of points to generate: ";
    cin >> numPoints;

    cout << "Enter the radius of the circle: ";
    cin >> radius;

    // Measure serial execution time
    double startSerial = __rdtsc() ;
    double estimatedPiSerial = monteCarloPiSerial(numPoints, radius);
    double endSerial = __rdtsc() ;

    double serialTime = endSerial - startSerial;

    // Measure parallel execution time
    double startParallel = __rdtsc();
    double estimatedPiParallel = monteCarloPiParallel(numPoints, radius);
    double endParallel = __rdtsc();
```

```
    double parallelTime = endParallel - startParallel;

    // Calculate Speedup
    double speedup = serialTime / parallelTime;

    // Print results
    cout << "Estimated value of Pi (Serial): " << estimatedPiSerial <<
endl;
    cout << "Estimated value of Pi (Parallel): " << estimatedPiParallel <<
endl;
    cout << "Time taken (Serial): " << serialTime << " cc" << endl;
    cout << "Time taken (Parallel): " << parallelTime << " cc" << endl;
    cout << "Speedup: " << speedup << endl;

    return 0;
}
```

در تابع main کار خاصی انجام نمی دهیم و صرفا توابه سریال و موازی را صدا میزنیم و زمان اجرای آنها را اندازه گیری و گزارش میکنیم.

محاسبه SpeedUp:

```
Enter the number of points to generate: 999999
Enter the radius of the circle: 9
Estimated value of Pi (Serial): 3.14173
Estimated value of Pi (Parallel): 3.14123
Time taken (Serial): 1.32253e+09 cc
Time taken (Parallel): 2.91681e+08 cc
Speedup: 4.53415
```

```
Enter the number of points to generate: 30000
Enter the radius of the circle: 5
Estimated value of Pi (Serial): 3.12947
Estimated value of Pi (Parallel): 3.14307
Time taken (Serial): 1.08137e+08 cc
Time taken (Parallel): 2.40791e+07 cc
Speedup: 4.49088
```

```
Enter the number of points to generate: 10000
Enter the radius of the circle: 1
Estimated value of Pi (Serial): 3.1244
Estimated value of Pi (Parallel): 3.1216
Time taken (Serial): 7.34095e+07 cc
Time taken (Parallel): 1.8728e+07 cc
Speedup: 3.91978
```

```
Enter the number of points to generate: 100000
Enter the radius of the circle: 1000
Estimated value of Pi (Serial): 3.14904
Estimated value of Pi (Parallel): 3.14168
Time taken (Serial): 1.48629e+08 cc
Time taken (Parallel): 6.17572e+07 cc
Speedup: 2.40667
```

```
Enter the number of points to generate: 50000
Enter the radius of the circle: 10
Estimated value of Pi (Serial): 3.1408
Estimated value of Pi (Parallel): 3.14768
Time taken (Serial): 8.90637e+07 cc
Time taken (Parallel): 2.56803e+07 cc
Speedup: 3.46818
```

```
Enter the number of points to generate: 1000
Enter the radius of the circle: 2
Estimated value of Pi (Serial): 3.128
Estimated value of Pi (Parallel): 3.232
Time taken (Serial): 4.00257e+07 cc
Time taken (Parallel): 1.21952e+07 cc
Speedup: 3.28208
```

$$Average\ SpeedUp = 3.684$$