UNIVERSIDAD AUTÓNOMA DE MADRID

MASTER'S DEGREE IN RESEARCH AND INNOVATION IN COMPUTATIONAL INTELLIGENCE AND INTERACTIVE SYSTEMS (I2-ICSI)

Numerical and Data-Intensive Computing (Course 2022/23)

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

Computación a Gran Escala (Curso 2022/23)

Laboratory 2: OPENMP (Parallel loops)

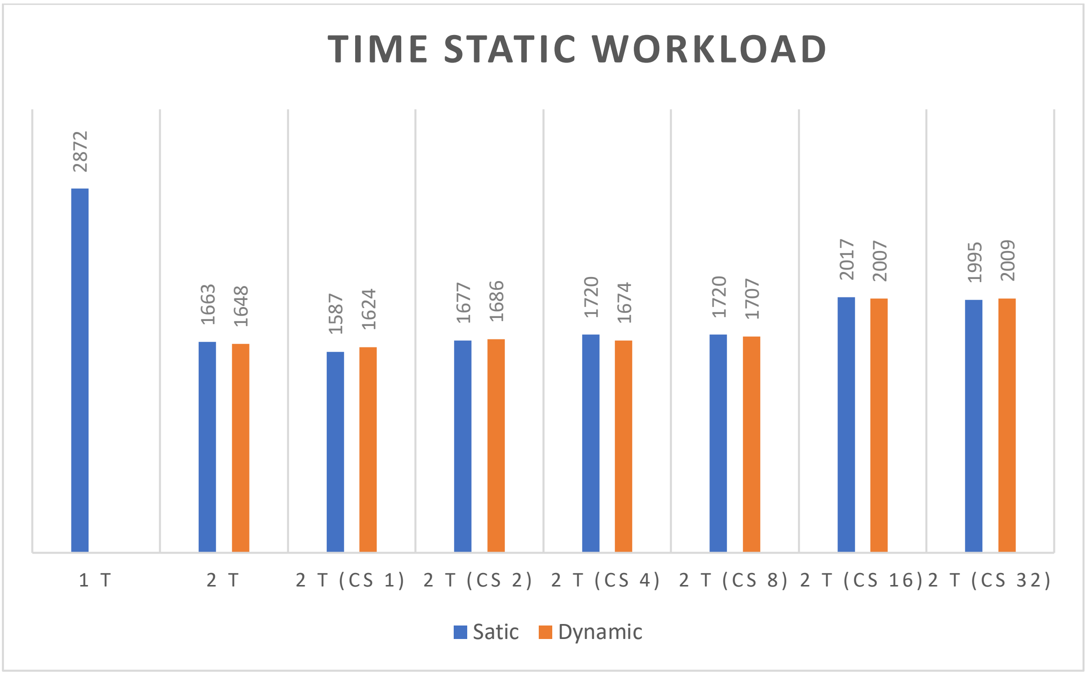
Deadline: November 3rd

Silvio Baratto
Tobias Ganzmann

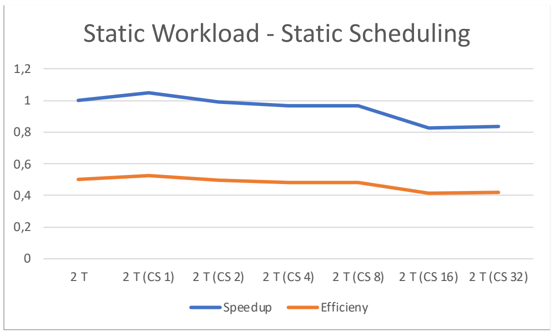
Table of Contents

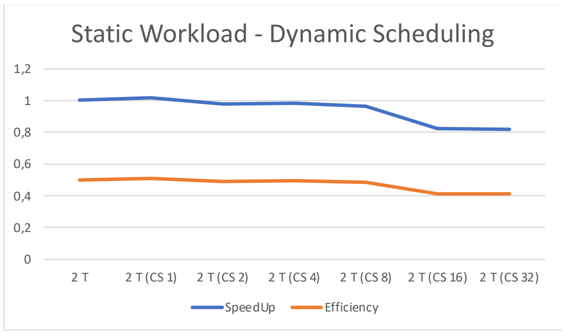
Compulsory Assignment I

Static Workload



Static scheduling pre-plans the execution of the code and always distributes the chunks to threads in the same order. If no chunk-size is given, the iterations are divided into as many chunks as threads available. Here the overhead is low and with two threads the execution is basically just cut in half. The best result (even though, differences are small) can be seen with with chunk size one, providing the lowest Wall Time and therefore highest efficiency and speedup. Increasing the chunk-size on static scheduling does show an increase of Wall Time, however with low chunk sizes the increase as well as the speedup is not significantly worse. This is due to higher chunk-sizes reducing the flexibility of the program and not being able to adapt to the demands of the unique iterations.



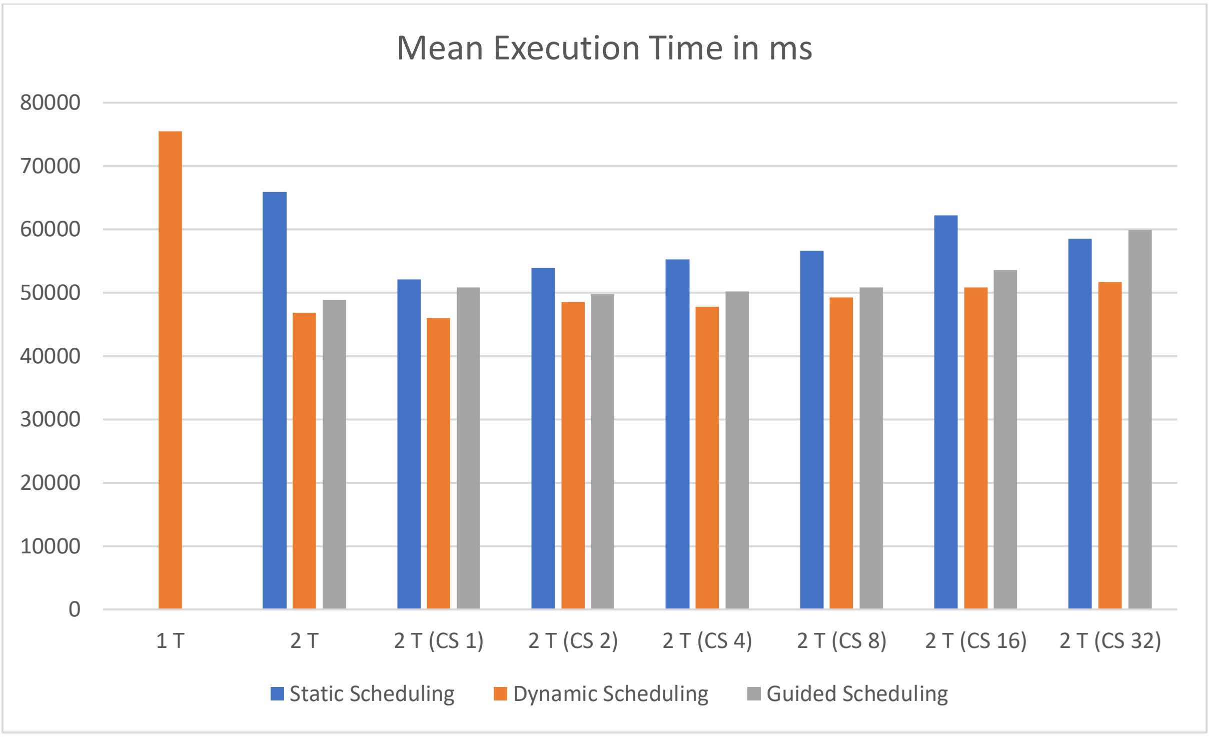


With dynamic scheduling the order of the chunk assignment to the threads is not fixed. Also the standard chunk-size definition is different, as the default value here is 1, making the two commands

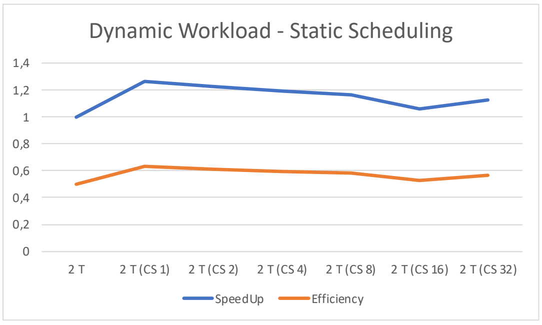
```
#pragma omp parallel for schedule( dynamic)
#pragma omp parallel for schedule( dynamic, 1 )
```

identical in effect. For the Static Workload Package hardly any difference can be seen between the two scheduling modes. The reason for this is, that the iterations more evenly sized and hardly profit from a dynamic assignment to the available threads.

Dynamic Workload

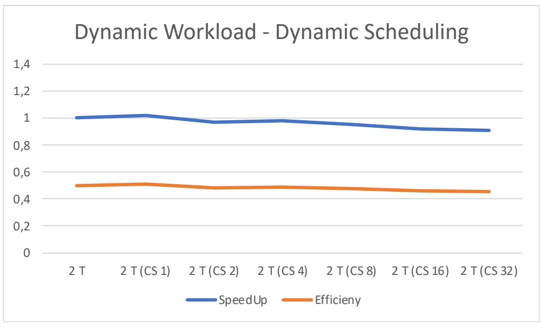
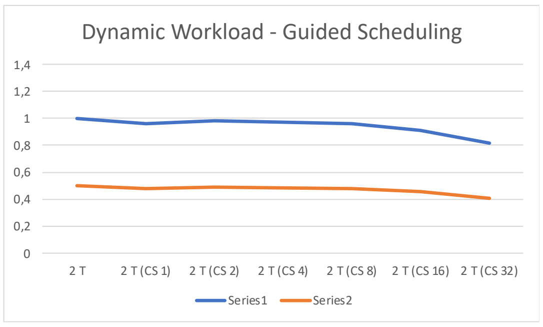


This observation is different for the execution of the dynamic workload. Here we can clearly see the static scheduling producing the worst wall times. As the limit in the function is pseudo-randomly generated, work package assignment is more dependent on the actual demand of the iteration and benefits therefore from flexible assignments which are not provided in the static scheduling. As with the static workload though the improvement of setting small chunk-sizes can be noticed.



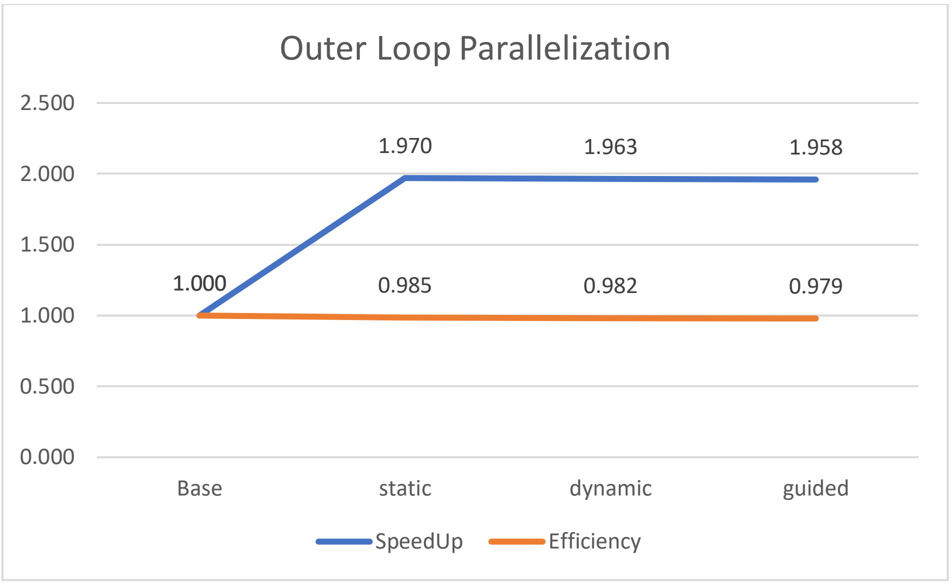
On average the second-best performance can be observed by that guided scheduling. With the approach the chunk size is dependent on the number of available threads and the remaining unassigned iterations. The remaining iterations are divided by the threads for each time a new thread begins, meaning for two threads the chunk-size halves each time a new chunk is assigned. Here the chunk sizes come into play as it represents the lower limit of the size a chunk can have. Therefore, a lower chunk-size is more efficient for the guided scheduling as it allows the chunk to shrink further and be more flexible with the assignment.

The best performance shows dynamic scheduling, making use of the flexible thread assignment once again. Even though the differences are not as big, with small chunk-sizes the best results can be observed. Especially with big chunk-sizes this scheduling technique works more efficient than the others.

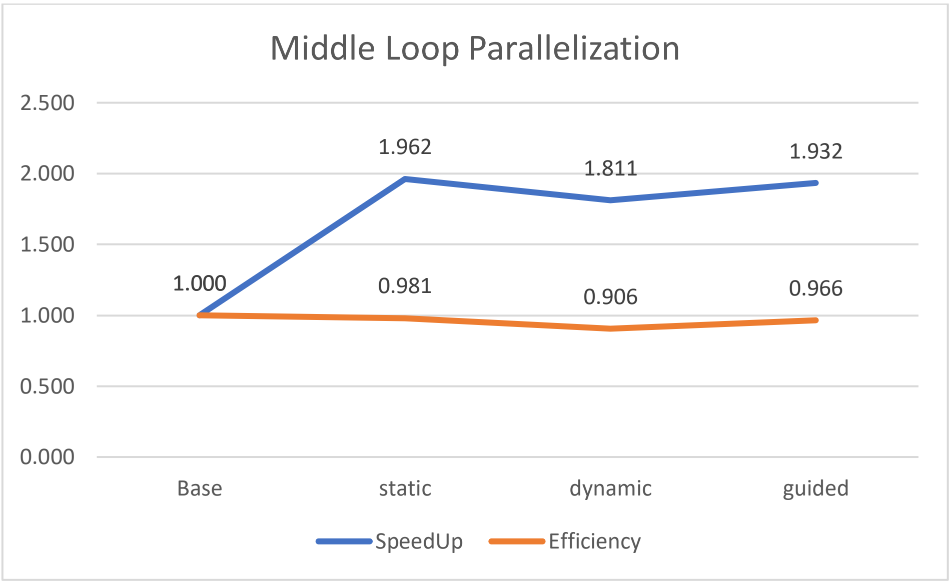

Dynamic Workload - Dynamic Scheduling
1,4
1,2
1
0,8
0,6
0,4
0,2
0
2 T
2 T (CS 1)
2 T (CS 2)
2 T (CS 4)
2 T (CS 8)
2 T (CS 16)
2 T (CS 32)
SpeedUp
Efficieny

Dynamic Workload - Guided Scheduling
1,4
1,2
1
0,8
0,6
0,4
0,2
0
2 T
2 T (CS 1)
2 T (CS 2)
2 T (CS 4)
2 T (CS 8)
2 T (CS 16)
2 T (CS 32)
Series1
Series2

Compulsory Assignment II

For task2.c we tend to naïve matrix multiplication again. For the base case we test the wall time several times and take the average (as for all other runs as well). Trying different scheduling methods for the outer loop does not indicate a significant difference; the static scheduling does pose a slight lead in SpeedUp and Efficiency though. As the type of operation is always the same and each single iteration does not take much time there is no point in trying to schedule the operations differently. With bigger matrices though we would expect a bigger difference between static and dynamic scheduling.



Next we apply parallelization to the middle loop. As can be seen below, static scheduling again performs best with minimal margin. It is noteworthy though that dynamic scheduling shows a slight dip in both performance indicators.



Executing the parallelization on the inner loop leads to a failure in calculation as the output does not match the target output. This is due to type of operation in the inner lop:

```
C[i][j] += A[i][k] * B[k][j];
```

As the inner loop concerns *k*, with parallel execution we split up the addition operation resulting in the wrong outcome.

	Outer Loop				Middle Loop			
	Base	**static**	**dynamic**	**guided**	**Base**	**static**	**dynamic**	**guided**
	Wall Time	**Wall Time**	**Wall Time**	**Wall Time**	**Wall Time**	**Wall Time**	**Wall Time**	**Wall Time**
	640	325	329	328	640	328	353	334
	644	323	327	326	644	326	354	331
	640	323	328	328	640	326	353	330
	641	327	323	328	641	327	355	325
	639	328	325	326	639	326	354	338
Average	**641**	**325**	**326**	**327**	**641**	**327**	**354**	**332**
SpeedUp	**1,000**	**1,970**	**1,963**	**1,958**	**1,000**	**1,962**	**1,811**	**1,932**
Efficiency	**1,000**	**0,985**	**0,982**	**0,979**	**1,000**	**0,981**	**0,906**	**0,966**

Concerning the ideal execution time, we can see from the table above that in general parallelizing the outer loop is more successful. However, as there are hardly any differences in execution time, recommending a static scheduling for the outer loop results in the fastest time but only by 1ms difference. Consequently, we propose following code:

```
void Mul1Par( char A[N][N], char B[N][N], int C[N][N] )
{
   int i, j, k;

   #pragma omp parallel for schedule(static)
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
     {
      C[i][j] = 0;

      for (k=0; k<N; k++)
         C[i][j] += A[i][k] * B[k][j];
     }
}
```

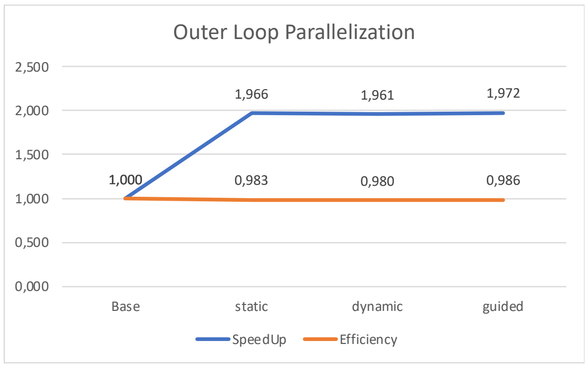
Optional Assignment A

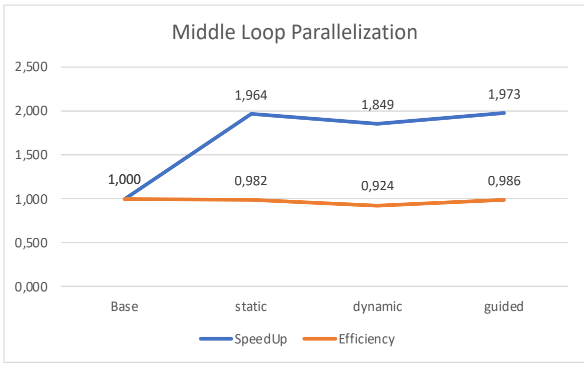
Shown below is the optimized code from Assignment 1 with the "ideal" parallelization:

```
void Mul2Par( char A[N][N], char B[N][N], int C[N][N] )
{
   int i, j, k;

   #pragma omp parallel for schedule(guided)
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
     {
      C[j][i] = 0;

      for (k=0; k<N; k++)
         C[j][i] += A[j][k] * B[k][i];
     }
}
```

As can be seen from the table though, again there is no clear difference between the most times or in comparison to Mult1Par. Guided scheduling shows minimal improvement (1ms), which could be due to random differences. Again, dynamic scheduling shows a slight dip in performance, probably due to overhead.

	Outer Loop				Middle Loop			
	Base	static	dynamic	guided	Base	static	dynamic	guided
	Wall Time	Wall Time	Wall Time	Wall Time		Wall Time	Wall Time	Wall Time
	639	332	325	326	640	326	348	326
	639	325	326	324	644	328	345	328
	639	325	331	326	640	329	345	322
	640	326	325	325	641	324	346	323
	640	322	327	324	639	324	349	325
Average	**639**	**326**	**327**	**325**	**641**	**326**	**347**	**325**
SpeedUp	**1,000**	**1,966**	**1,961**	**1,972**	**1,000**	**1,964**	**1,849**	**1,973**
Efficiency	**1,000**	**0,983**	**0,980**	**0,986**	**1,000**	**0,982**	**0,924**	**0,986**



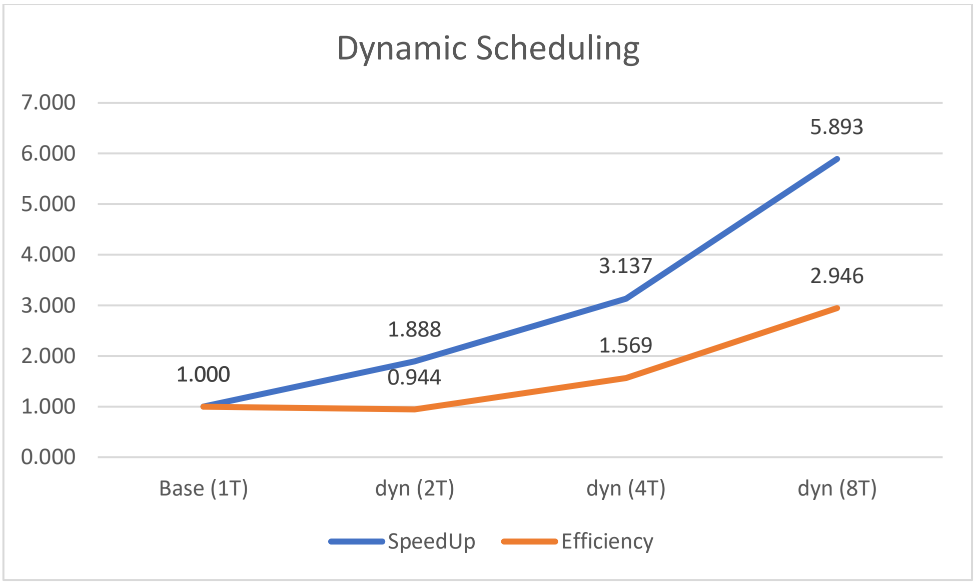


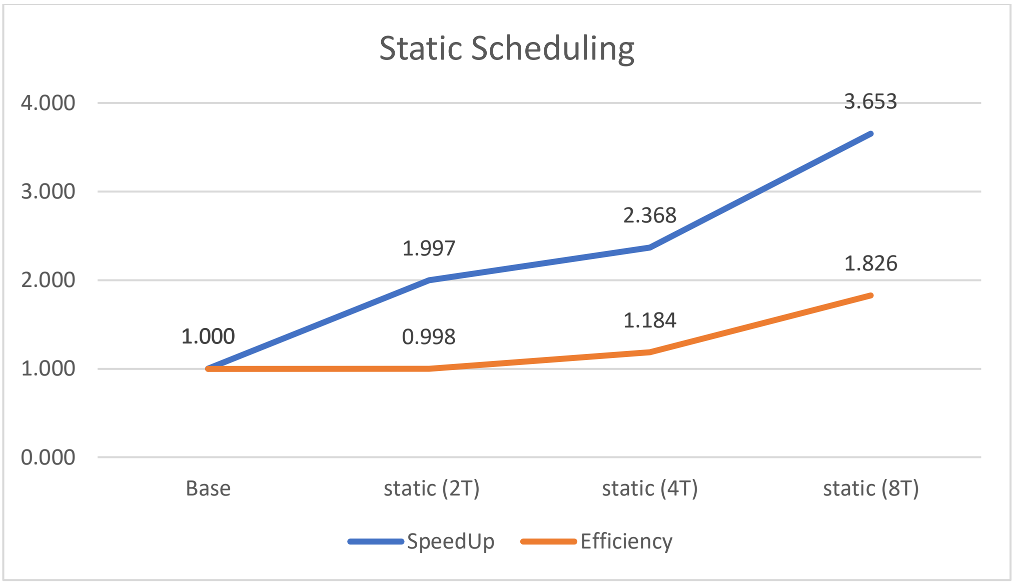
Optional Assignment B

Task3.c creates a fractal image. We try to find the best optimization for this code. To do that we optimize the function we use to calculate the fractals by parallelizing the for-loops (using “collapse):

```
//loop through every pixel
  #pragma omp parallel for schedule(static) collapse(2)
  for(y = 0; y < h; y++)
    for(x = 0; x < w; x++)
    {
      //calculate the initial real and imaginary part of z, based on the pixel location and zoom and
        Position values
      pr = 1.5 * (x - w / 2) / (0.5 * zoom * w) + moveX;
      pi = (y - h / 2) / (0.5 * zoom * h) + moveY;
      newRe = newIm = oldRe = oldIm = 0; //these should start at 0,0
      //"i" will represent the number of iterations
      //start the iteration process
      for(i = 0; i < maxIterations; i++)
      {
        //remember value of previous iteration
        oldRe = newRe;
        oldIm = newIm;
        //the actual iteration, the real and imaginary part are calculated
        newRe = oldRe * oldRe - oldIm * oldIm + pr;
        newIm = 2 * oldRe * oldIm + pi;
        //if the point is outside the circle with radius 2: stop
        if((newRe * newRe + newIm * newIm) > 4) break;
      }

    if(i == maxIterations)
        color(img, w, x, y, 0, 0, 0); // black
      else
      {
        z = sqrt(newRe * newRe + newIm * newIm);
        brightness = 256. * log2(1.75 + i - log2(log2(z))) / log2((double)maxIterations);
        color(img, w, x, y, brightness, brightness, 255);
      }
    }
```

This code is the result of testing several options; static and dynamic scheduling with different thread counts. We tried to parallelize the inner loop as well but without success because OpenMP throws an error when trying to parallelize a loop that contains a break statement. In the end we use dynamic scheduling because the workload is variable. Therefore the flexible thread usage is to be preferred, as opposed to static scheduling which works better with less variable workloads.


Dynamic Scheduling
7.000
6.000
5.000
4.000
3.000
2.000
1.000
0.000
1.000
1.888
0.944
3.137
1.569
5.893
2.946
Base (1T)
dyn (2T)
dyn (4T)
dyn (8T)
SpeedUp
Efficiency


Static Scheduling
4.000
3.000
2.000
1.000
0.000
1.000
1.997
0.998
2.368
1.184
3.653
1.826
Base
static (2T)
static (4T)
static (8T)
SpeedUp
Efficiency

Optional Assignment C

The script algi.c is used for image processing. Initial testing shows a wall time of 1792ms on average. We tried to parallelize the functions that concern reading and writing of pixels because those are the most called. This results in a small improvement of performance (→ 1729ms on average, resulting in a SpeedUp of 1,036 and an efficiency of 51,8%) and can be seen in the code examples below. However, no significant improvements to the execution time can be made because the parts of the code that take the most time (like 'Create_Label_Array()' or 'BMP_Write_Gray_Image()') cannot be parallelized. This is due to the way these functions work. The sequence observed here is not dividable into smaller instances and is therefore not benefiting from parallelization.

```
void Binarize_Gray_Image( tGrayImage *src, tGrayPixel threshold, tGrayImage *dst )
{
   int r, c;
   tGrayPixel pixel;

   #pragma omp parallel for schedule(guided) collapse(2) private(r,c, pixel)
   for (r=0; r<src->rows; r++)
   for (c=0; c<src->cols; c++)
   {
      BMP_Read_Gray_Pixel( src, r, c, &pixel );
      if (pixel > threshold) pixel = 255;
      else pixel = 0;
      BMP_Write_Gray_Pixel( dst, r, c, pixel );
   }
}

…

void Or_Gray_Images( tGrayImage *src1, tGrayImage *src2, tGrayImage *dst )
{
   int r, c;
   tGrayPixel pixel1, pixel2;

   #pragma omp parallel for schedule(guided) collapse(2) private(pixel1, pixel2)
   for (r=0; r<src1->rows; r++)
   for (c=0; c<src1->cols; c++)
   {
      BMP_Read_Gray_Pixel( src1, r, c, &pixel1 );
      BMP_Read_Gray_Pixel( src2, r, c, &pixel2 );

      pixel1 = pixel1 | pixel2;
      BMP_Write_Gray_Pixel( dst, r, c, pixel1 );
   }
}

…
```

```
void Graylevel_Histogram( tGrayImage *src, int graylevels, int *histo )
{
/* Compute the histogram of a gray image of the given number of
   gray levels */

    int i;
    int r, c;
    tGrayPixel pixel;

    for (i=0; i<graylevels; i++) histo[i] = 0;

    #pragma omp parallel for schedule(guided) collapse(2) shared(histo) private(pixel)
    for (r=0; r<src->rows; r++)
    for (c=0; c<src->cols; c++)
    {
        BMP_Read_Gray_Pixel( src, r, c, &pixel );
        histo[ pixel ]++;
    }
}

…

void Conditional_Graylevel_Histogram( tGrayImage *src, tGrayImage *cnd, int graylevels, int
*histo )
{
/* Compute the histogram of a gray image of the given number of
   gray levels only considering the pixels that are equal to 255
   in the cns image */

    int i;
    int r, c;
    tGrayPixel pixel, flag;

    for (i=0; i<graylevels; i++) histo[i] = 0;

    #pragma omp parallel for schedule(guided) collapse(2) private(r,c,pixel,flag, histo)
    for (r=0; r<src->rows; r++)
    for (c=0; c<src->cols; c++)
    {
        BMP_Read_Gray_Pixel( src, r, c, &pixel );
        BMP_Read_Gray_Pixel( cnd, r, c, &flag );

        if (flag == 255) histo[ pixel ]++;
    }
}
```

```
void Binary_Dilation( tGrayImage *src, tGrayImage *dst )
{
   int r, c, rr, cc;
   int r1, r2, c1, c2;
   char flag;
   tGrayPixel pixel;

   #pragma omp parallel for schedule(guided) collapse(2) private(r,c,pixel)
   for (r=0; r<src->rows; r++)
   for (c=0; c<src->cols; c++)
   {
     BMP_Read_Gray_Pixel( src, r, c, &pixel );
     BMP_Write_Gray_Pixel( dst, r, c, pixel );
   }
….

void Binary_Erosion( tGrayImage *src, tGrayImage *dst )
{
   int r, c, rr, cc;
   int r1, r2, c1, c2;
   char flag;
   tGrayPixel pixel;

   #pragma omp parallel for schedule(guided) collapse(2) private(r, c, pixel)
   for (r=0; r<src->rows; r++)
   for (c=0; c<src->cols; c++)
   {
     BMP_Read_Gray_Pixel( src, r, c, &pixel );
     BMP_Write_Gray_Pixel( dst, r, c, pixel );
   }
…
```