Cristian Dávila
Enrique González

4.1 Reading activity

In this first part of the report we first briefly describe the basic formulation for the Mandelbrot set and then its implementation in the sequential code available (mandel-serial.c). The Mandelbrot set is a particular set of points, in the complex domain which is an extension of the real domain. The complex domain can be represented as the sum between a real and an imaginary number. These points belong to the set if the sequence $\{z_0 = 0, z_{n+1} = z^2_n + c\}$ is bounded. For example, we know that c=-1 belongs to the Mandelbrot set, because his sequence is bounded (0,-1,0,-1,..). However, if c=2, we obtain an unbounded sequence (0, 2, 6, 38,..).

The simplest algorithm for drawing a picture of the Mandelbrot set is the following. We discretize the complex space in a set of points and each point corresponds to a pixel in a two dimensional plot. To color any such pixel, let c (represented by the complex variable c at the mandel-serial.c code) be the midpoint of that pixel. Now we begin to calculate the critical/divergent points of the set, starting from the critical point 0 (represented by 'z' at *mandel-serial.c*). We check that the sum between the square of the real component and the square of the imaginary component is lower than the lengthsq variable (that variable has the value of the module).

When it is the case that is bigger, we can see this point doesn't belong to the Mandelbrot set, then we'll color it according to the iteration, which indicates the 'k' variable. If It's in the set, we continue iterating until reaching the limit, represented by 'maxiter'. Then we've checked our point belongs to the Mandelbrot set (aproximately), therefore it'll be white colored.

References: Wikipedia (http://es.wikipedia.org/wiki/Conjunto_de_Mandelbrot)

4.2 Task granularity analysis

1. Which are the two most important common characteristics of the task graphs generated for the two task granularities (Row and Point) for mandel-tareador?

La primera característica és que cap dels dos tenen dependencies en el codi 'mandel', i per tant obre la porta a una paralelització.
La segona característica és que no estan balancejades, tenen diferents granularitats, perquè les iteracions no triguen el mateix temps. Això ho indica la mida dels nodes dintre del graf: els nodes tenen mida diferent, és a dir, diferent nombre d'iteracions, degut a que el número d'iteracions per saber si pertany al conjunt és variable.

2. Which section of the code is causing the serialization of all tasks in mandeld-tareador?

```
#if _DISPLAY_
        /* Scale color and display point  */
        long color = (long) ((k-1) * scale_color) + min_color;
        if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
        }
```

En aquesta secció de codi els diferents threads creats intenten accedir simultàniament a les mateixes variables, per tant accedeixen també a llibreries compartides, i els accessos a memoria es solapen i modifiquen les variables accedides per altres threads, per tant la visualització no es pot dur a terme si no tractem la serialització en aquesta part del codi.
Per fer-ho, situarem aquesta zona de codi com una zona d'exclusió mútua, amb
#pragma omp critical { ..Codi.. }

4.3 OpenMP execution analysis

1- For each parallelization strategy of the non-graphical version, complete the following table with the execution time for different loop schedules and number of threads, reasoning about the results that are obtained.

Per obtenir els resultats, hem utilitzat 10.000 iteracions. Les taules estan expresades en segons. Cada taula representa els temps obtinguts per cada estrategia de paralelització.
En les gràfiques, l'eix 'y' representa el temps en segons, i l'eix 'x' representa el nombre de threads.

Row

Num. threads	static	static,1	dynamic,1	guided,1
1	1.16942	1.17056	1.18143	1.18742
2	3.09281	3.07886	3.04577	3.09356
4	3.08783	1.53534	1.5218	3.35898
6	3.01694	1.08067	1.06617	2.30561
8	2.56359	0.812345	0.800572	1.66234
10	2.19875	0.683248	0.675653	1.43547
12	1.96797	0.568685	0.563656	1.24876

Pixel

Num. threads	static	static,1	dynamic,1	guided,1
1	1.17659	1.17209	1.17038	1.19943
2	3.09644	3.09633	3.10164	3.09357
4	3.09379	1.56279	1.56055	3.33922
6	2.95687	1.08422	1.09608	2.31768
8	2.55009	0.828377	0.830423	1.65022
10	2.18775	0.685712	0.702069	1.43912
12	1.93155	0.569555	0.590152	1.20831

Tipus de loop schedules:

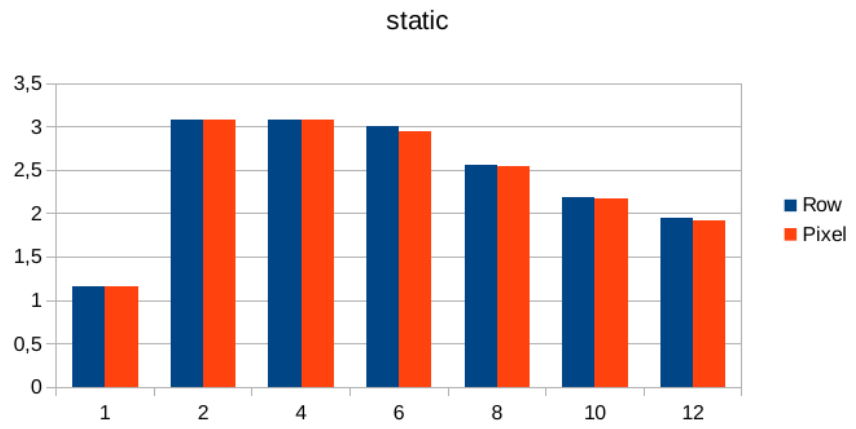
Static: El nombre d'iteracions es reparteixen entre els diferents threads, aproximadament de forma: iteracions/nombre threads. Aquests blocs es reparteixen seguint la política de planificació Round Robin.

Static, N: El nombre d'iteracions es reparteixen en blocs de mida N. Els blocs es reparteixen de la mateixa forma que en static.

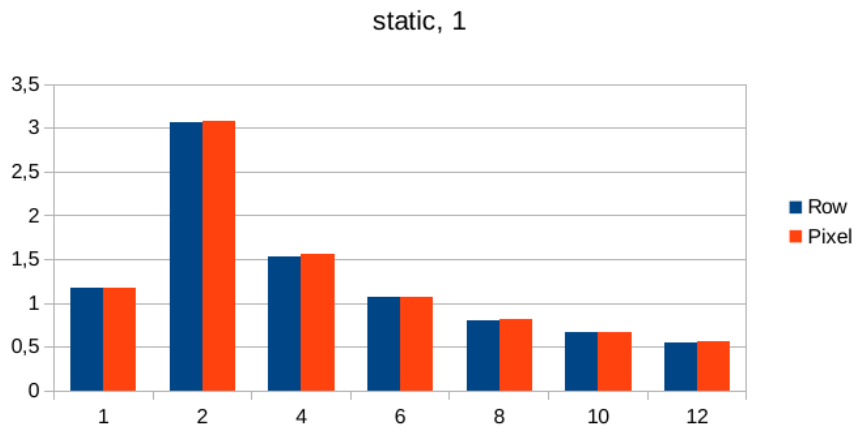
Dynamic, N: A mida que els threads acaben la seva execució, agafen blocs de mida N.

Guided, N: Es una variant de Dynamic. La mida dels blocs, que és el nombre d'iteracions restants dividit entre el nombre de threads, va disminuint segons els threads demanen més blocs, però tindràn mida N com a mínim.

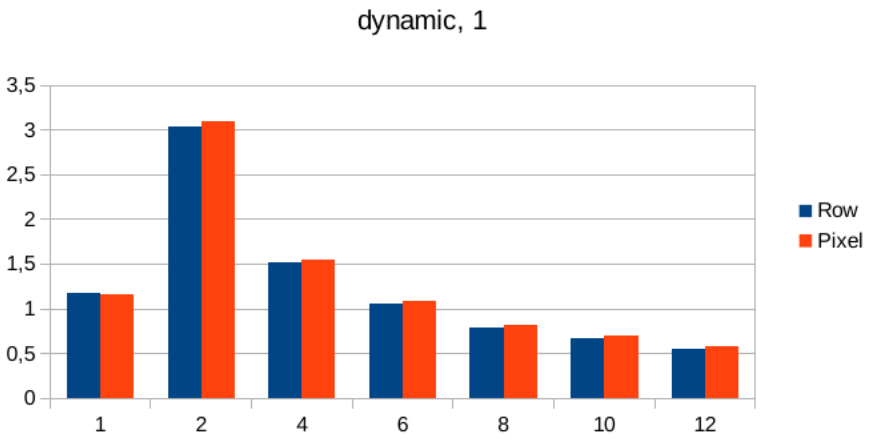
Resultats obtinguts:



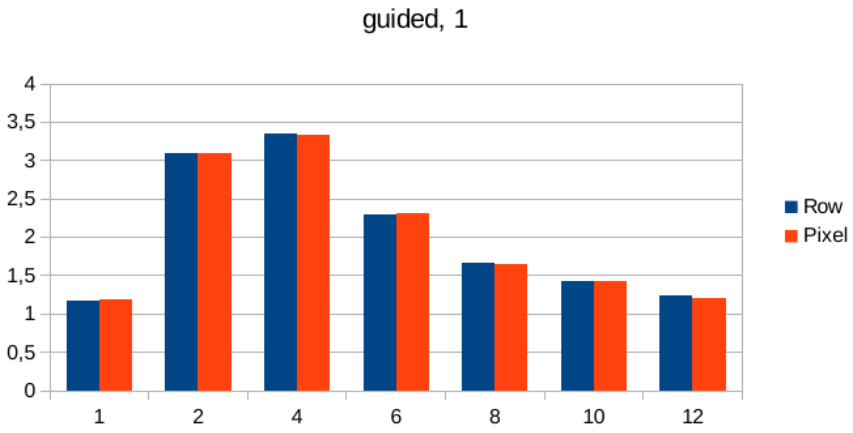
Static: Utilitzant '**static'**, tenim threads que tenen més càrrega que d'altres. Això és degut a que els blocs en els quals es reparteixen les iteracions son de mida "iteracions/nombre threads", per tant els threads executen blocs d'aquesta mida, fent que alguns threads tinguin més càrrega que altres, obligant a que els que tenen menys càrrega els esperin. El loop shedule **'static'** és la més lenta comparada amb les altres 3.



Static, 1: Amb **‘static,1’**, el nombre d’iteracions es reparteixen entre més blocs, obtenint blocs més petits. Així és més fàcil repartir equitativament (Roun-Robin) els blocs entre els threads i aconseguir equilibrar la càrrega. Amb aquest loop shedule, aconseguim que els threads acabin amb temps més semblant i obtenim millors resultats que amb **‘static’** i **‘guided,1’**. El balanç de càrrega aconseguit és molt semblant al de **‘dynamic,1’**. En la gràfica podem veure que amb 2 threads, obtenim l’overhead més gran, degut a les dependències entre els blocs, però amb 12 threads obtenim el millor temps, perquè l’òptima repartició de càrrega feta amb els blocs permet fer més treball paral·lel, tot i que tinguem dependències.



Dynamic,1: Amb **‘dynamic,1’,** repartim dinàmicament la càrrega de treball entre els threads, segons van acabant, i el nombre de blocs és superior al de threads (això és degut a que els blocs son de mida més petita que amb **‘static’**). Així aconseguim que els threads finalitzin amb temps similars (la carrega de treball és semblant en tots els threads). Els temps obtinguts son molt similars als obtinguts amb **‘static,1’**, per això, les dues gràfiques son pràcticament iguals.



Guided,1: Amb **‘guided,1’,** obtenim temps millors que amb **‘static’**, però no arribem als temps obtinguts amb **‘static,1’** i **‘dynamic,1’**. Això és degut a que **‘guided,1’** reparteix els blocs segons els threads vagin acabant, igual que **‘dynamic,1’**, però decrementant la mida dels blocs durant l’execució (com més execució hem fet, queda menys treball, dividint aquest treball en blocs mes petits, podem equilibrar millor la carrega), i aconseguint eliminar una part del desbalanceig de carrega entre els threads, però sense ser tan eficaç com **‘static,1’** i **‘dynamic,1’**. Això és degut que en **‘guided,1’** esperem a que acabin els threads amb blocs més grans per balancejar la càrrega, i ens produeix un overhead bastant gran, mentre que en **‘static,1’** i **‘dynamic,1’** segons acaben els threads, els hi donem més treball. En la gràfica podem veure com l’overhead més gran obtingut és amb 2 i 4 threads, mentre que els millors temps s’apropen al temps sense paral·lelització, però amb un petit overhead.

2- For each parallelization strategy, complete the following table with the information extracted from the Extrae instrumented executions with 8 threads and analysis with Paraver, reasoning about the results that are obtained.

Row

	static	static,1	dynamic,1	guided,1
Running average time per thread	767,142,036.12 ns	803,710,607.75 ns	807,879,625.88 ns	779,598,176.12 ns
Execution unbalance (average time divided by maximum time)	0.30	0.99	1.00	0.46
SchedForkJoin (average time per thread or time if only one does)	2,541,841,519 ns	11,720,011 ns	82,286.75 ns	193,392,044.50 ns

Pixel

	static	static,1	dynamic,1	guided,1
Running average time per thread	765,812,694.75 ns	840,793,730 ns	869,994,499.12 ns	779,957,284.50 ns
Execution unbalance (average time divided by maximum time)	0.30	0.93	0.93	0.47
SchedForkJoin (average time per thread or time if only one does)	318,827,940 ns	25,842,277.38 ns	41,439,885 ns	203,496,717.75 ns

Amb l'estratègia Row, la càrrega és més fàcil de repartir en blocs, i obtenim un millor balanceig de càrrega (amb **'static,1'** la càrrega està balancejada, i amb **'dynamic,1'**, està pràcticament balancejada). A més, amb més threads, és més fàcil de balancejar.

En canvi, amb l'estratègia Pixel, els resultats son pitjors. És lògic sabent que existeix un desbalanceig bastant gran de càrrega. Això és degut que a l'hora de paral·lalitzar, alguns pixels poden tenir més iteracions que d'altres dintre del bucle intern (en aquest bucle comprovem si un punt està dins del conjunt de Mandelbrot). Per tant, la càrrega de treball per cada thread pot variar, depenen de les iteracions que fa cada punt. Podem veure una gran diferencia entre **'static,1'** i **'dynamic,1'**, això ens indica que les diferències de temps son més degudes a la repartició de la carrega entre threads, que als problemes causats per l'estratègia de paral·lelització.

Codi font

Per a l'estratègia de Row vam fer servir:

```
1.  /* Calculate points and save/display */
2.    #pragma omp parallel for schedule(static)
3.    for (int row = 0; row < height; ++row) {
4.       for (int col = 0; col < width; ++col) {
5.           ...
```

Per a l'estratègia de Point vam fer servir:

```
1.  /* Calculate points and save/display */
2.    #pragma omp parallel for collapse(2) schedule(static)
3.    for (int row = 0; row < height; ++row) {
4.       for (int col = 0; col < width; ++col) {
5.           ...
```

Per a tractar la serialització que es produia al mandeld-tareador vam fer servir:

```
1.  #if _DISPLAY_
2.          /* Scale color and display point  */
3.          #pragma omp critical
4.          {
5.             long color = (long) ((k-1) * scale_color) + min_color;
6.               if (setup_return == EXIT_SUCCESS) {
7.                    XSetForeground (display, gc, color);
8.                    XDrawPoint (display, win, gc, col, row);
9.             }
10.         }
11.         ...
```