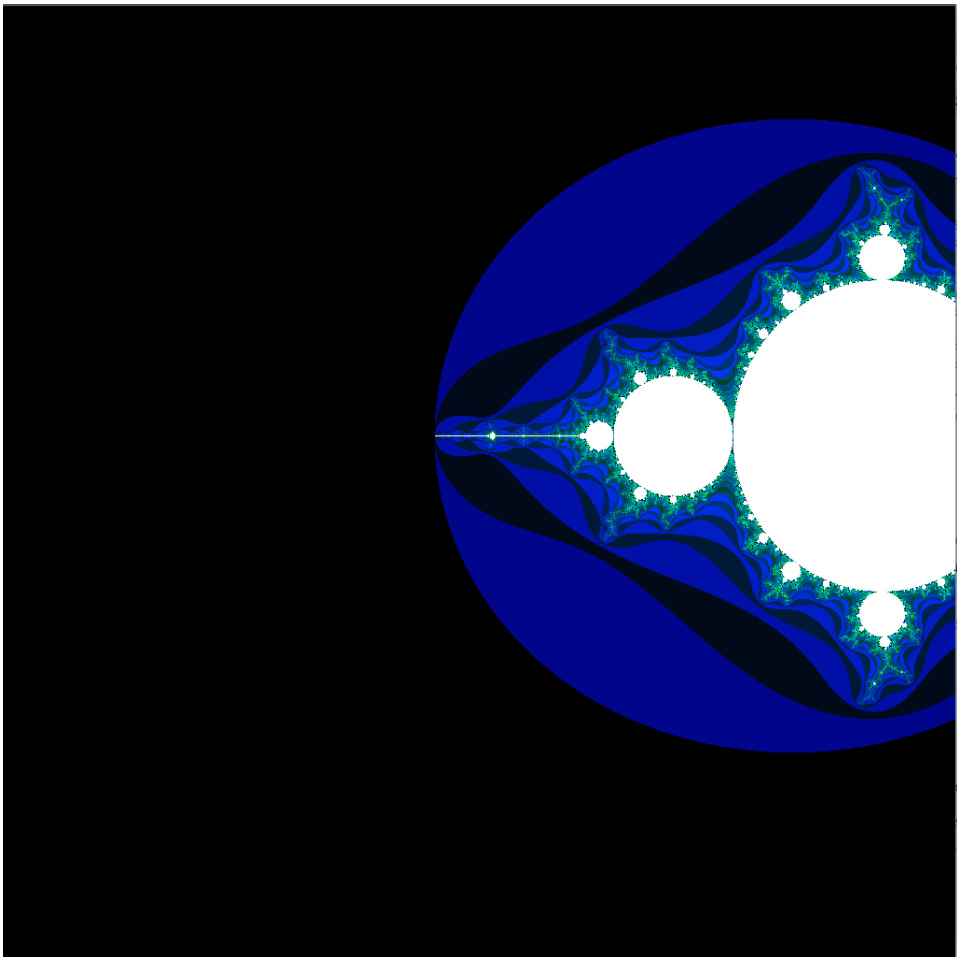
PAR Laboratory Assignment
Lab 3: Analysis of parallel strategies: the computation of the Mandelbrot set



UNIVERSITAT POLITÈCNICA DE CATALUNYA
FACULTAD D'INFORMÀTICA DE BARCELONA

Nuria Lou Enseñat Balaguer (nuria.lou.ensenat)
Albert Gómez Triunfante (albert.gomez.triunfante)

INDEX

Introduction

In this laboratory assignment we are going to explore different parallel strategies of iterative and recursive task decompositions of the computation of the Mandelbrot set. The Mandelbrot set is an iconic fractal structure defined in the complex plane, generated by successive iterations of a quadratic function. Recognised for its visually complex patterns, this figure represents an interesting and common challenge in parallel processing due to its iterative nature and the high degree of computing that each point in the image requires.

The main objective of this experiment is to evaluate how the different task decomposition techniques learnt in class, both iterative and recursive, affect the generation of this fractal in its efficiency and parallelism. We will use Tareador as our analysis tool to identify data dependencies and possible synchronisation problems, in addition to measuring load balance and concurrency potential in different strategies.

This detailed analysis will identify the best parallel strategy to maximise the performance and commuting efficiency of the Mandelbrot set. Please note that to make the problem doable, the maximum number of steps is limited: if that number of steps is reached, then the point p is said to belong to the Mandelbrot set.

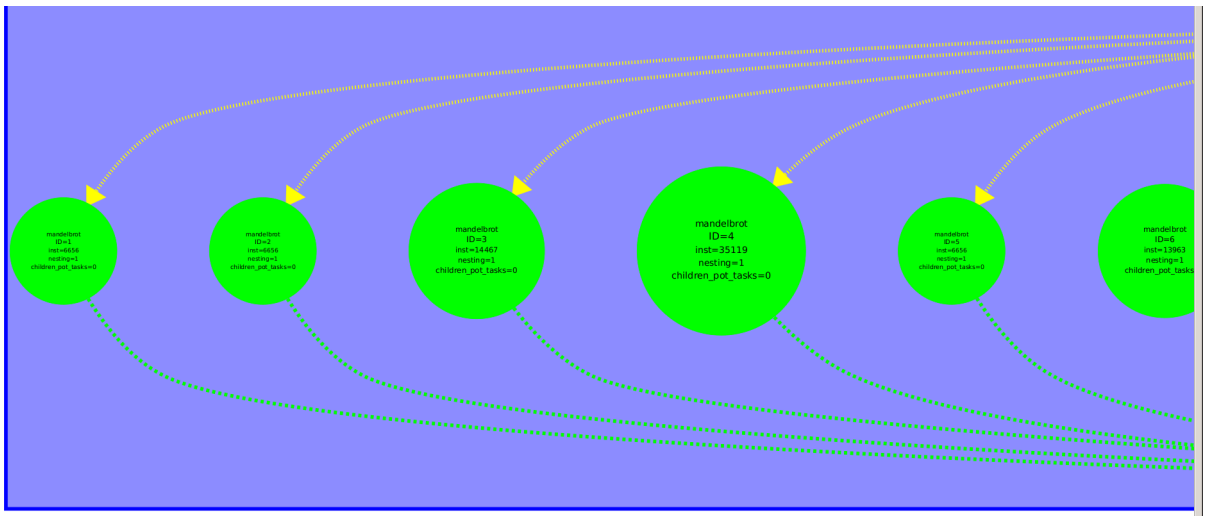
Iterative task decomposition analysis

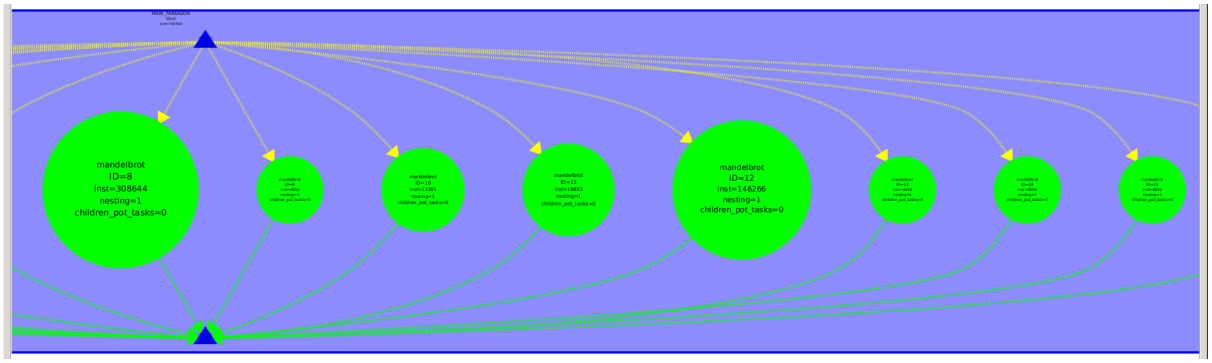
In this first section we will analyse different iterative task decompositions.

Original

Code: mandel-seq-iter-tar.c

Tareador TDG (with dependencies)





Dependence analysis

This code is embarrassingly parallelizable, therefore there are no dependencies between tasks.

Tareador TDG (without "data sharing" dependencies)

As mentioned in the previous section, the original code is already parallelised, therefore there are no "data sharing" dependencies and the Tareador TDG looks exactly the same in both cases.

T_∞ analysis

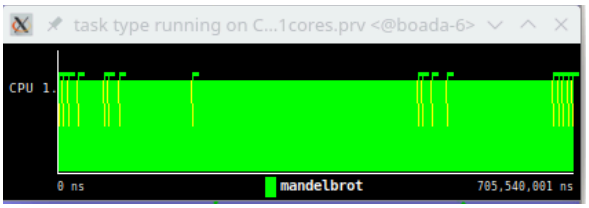


Figure 1: task running on 1 core.

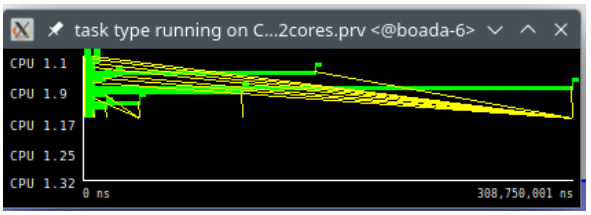


Figure 2: task running on 32 cores.

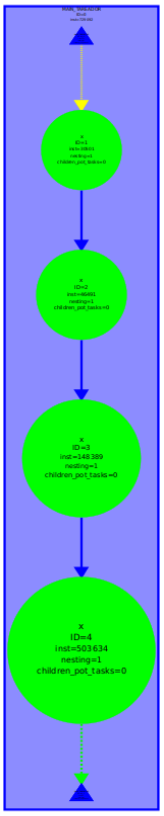
As it can be observed in the above images, in which we used 32 processors to simulate the parallelized program, it doesn't need that many to reach its maximum parallelism.

T_1 (ns)	T_∞ (ns)	Parallelism	Load Unbalance (Yes/No) Why?
705540001ns	308750001ns	2.285	Yes, because of task size

Original -d

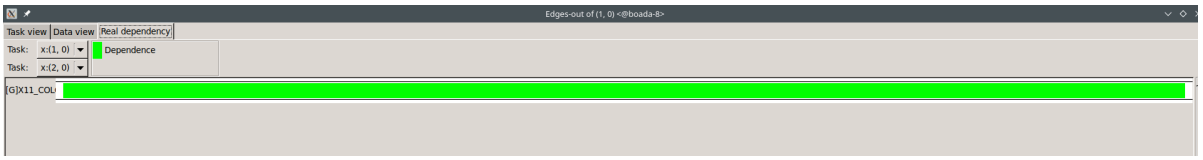
Code: mandel-seq-iter-tar.c

Tareador TDG (with dependencies)



Dependence analysis

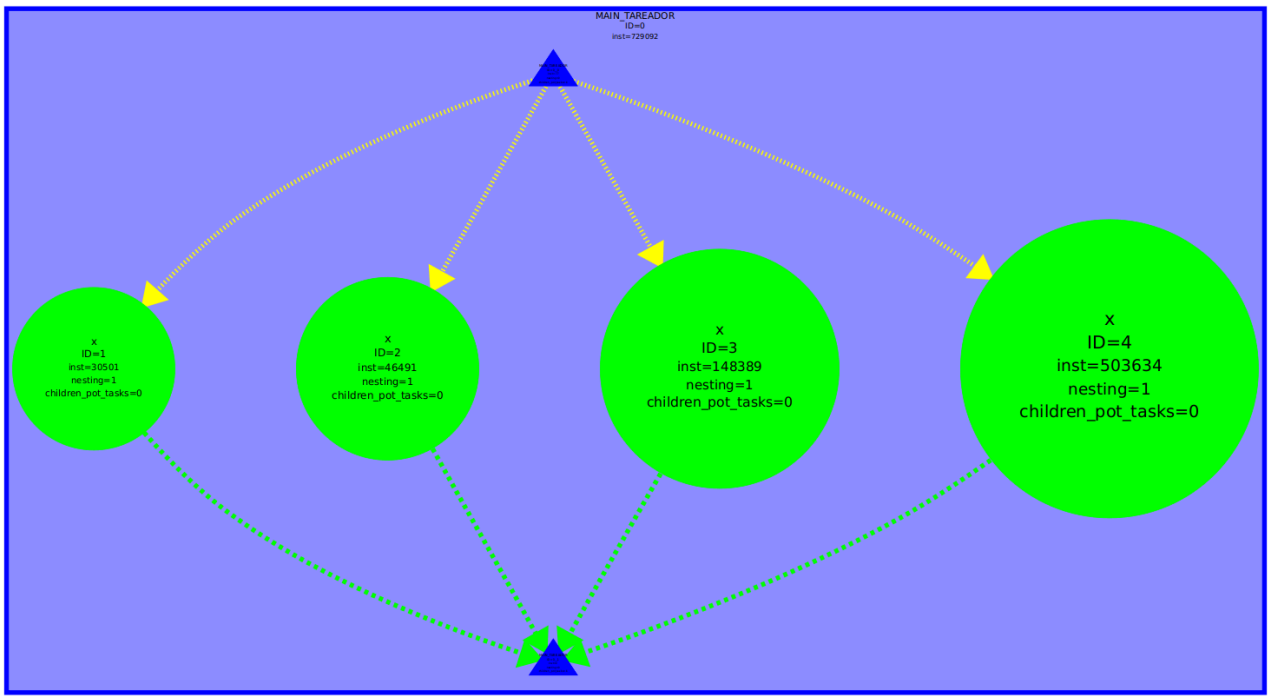
We check with Tareador's real dependency tool which is the variable causing all the dependencies.




```
if (setup_return == EXIT_SUCCESS) {
        tareador_disable_object(&X11_COLOR_fake);
        XSetForeground(display, gc, color);
       XDrawPoint(display, win, gc, px, py);
        tareador_enable_object(&X11_COLOR_fake);
}
```

As it can be observed in the above image and code, there is one particular variable that is causing all the dependencies of the code. Thus, we use tareador_disable_object and tareador_enable_object in the code snippet where it causes the dependencies to ensure it stops causing the dependency.

Tareador TDG (without "data sharing" dependencies)



T_∞ analysis

Figure 1: task running on 1 core.

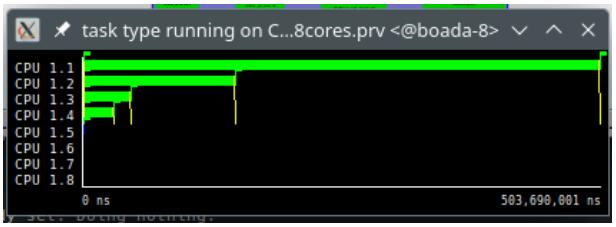


Figure 2: task running on 8 cores.

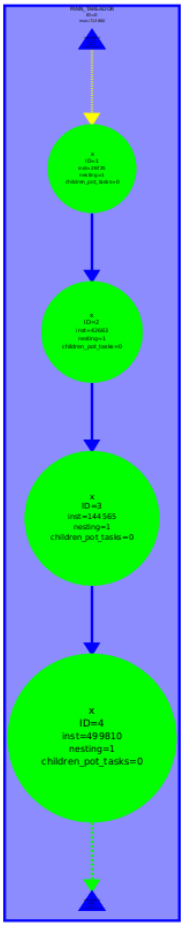
We can see in the above image, the comparison between T1 and T∞. We simulated with 8 processors, but only 4 were used by Tareador to reach its max parallelism, that is due to Load Unbalance.

T_1 (ns)	T_∞ (ns)	Parallelism	Load Unbalance (Yes/No) Why?
729092001	503690001	1.45	Yes, because of task size

Original -h

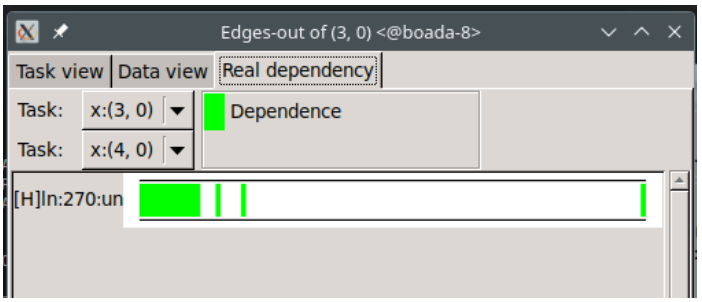
Code: mandel-seq-iter-tar.c

Tareador TDG (with dependencies)



Dependence analysis

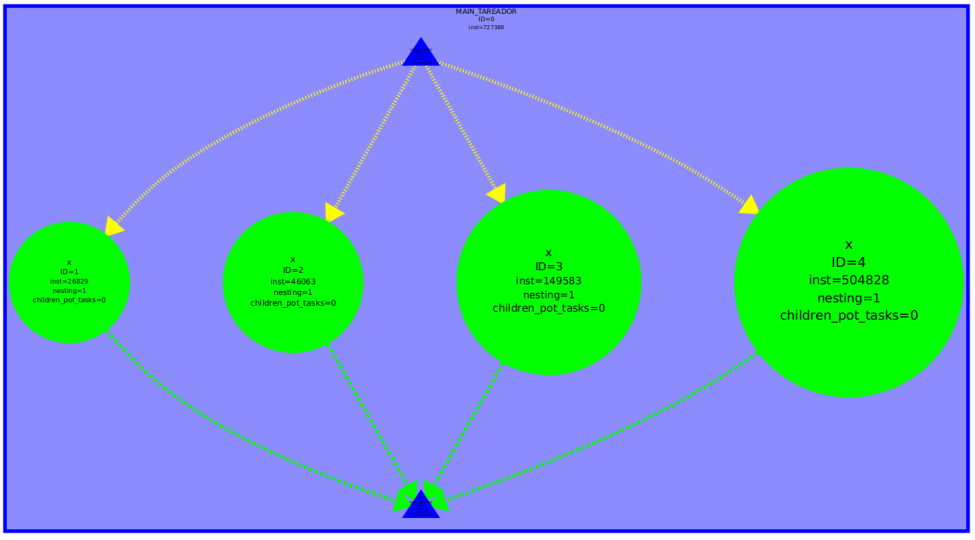
We check with Tareador's real dependency tool which is the variable causing all the dependencies.




```
if (output2histogram)
    tareador_disable_object(&histogram[M[y][x]-1]
);
    histogram[M[y][x] - 1]+=(TILE*TILE);
    tareador_enable_object(&histogram);
if (output2histogram)
    tareador_disable_object((&histogram[M[py][px]-1]
);
    histogram[M[py][px] - 1]++;
    tareador_enable_object(&histogram);
```

As it can be observed in the above image and code, there is one particular object, histogram, that is causing all the dependencies of the code. Thus, we use tareador_disable_object and tareador_enable_object in the code snippet where it causes the dependencies to ensure it stops causing the dependency.

Tareador TDG (without "data sharing" dependencies)



T_∞ analysis

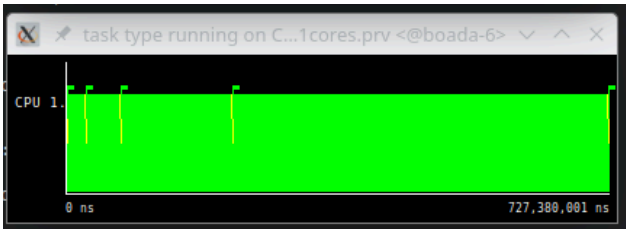


Figure 1: task running on 1 core.

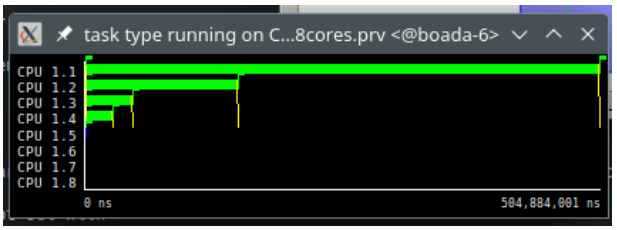


Figure 2: task running on 8 cores.

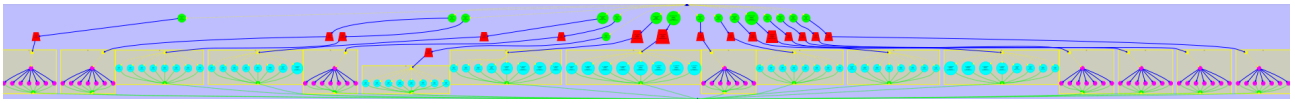
We can see in the above image, the comparison between T1 and T_∞. We simulated with 8 processors, but only 4 were used by Tareador to reach its max parallelism, that is due to Load Unbalance.

T_1 (ns)	T_∞ (ns)	Parallelism	Load Unbalance (Yes/No) Why?
727380001	504884001	1,44	Yes, because of task size

Finer grain

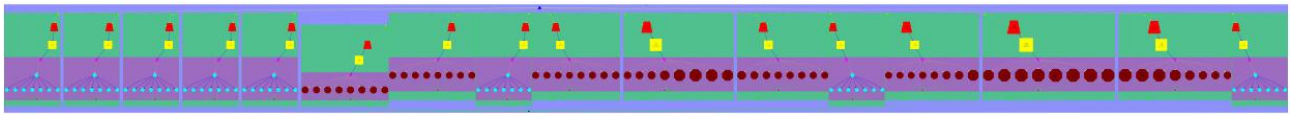
Code: mandel-seq-iter-tar-finer.c

Tareador TDG (with dependencies)

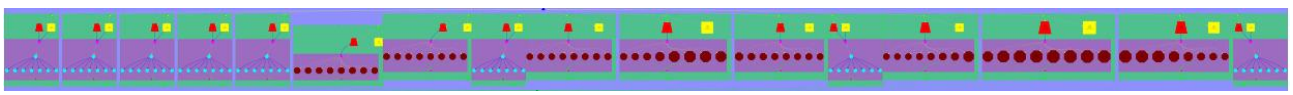


(This is an extremely large graph due to the enormous amount of tasks that are created, so we decided to zoom it out rather than just showing part of the TDG).

Dependence analysis



Tareador TDG (without "data sharing" dependencies)



T_∞ analysis

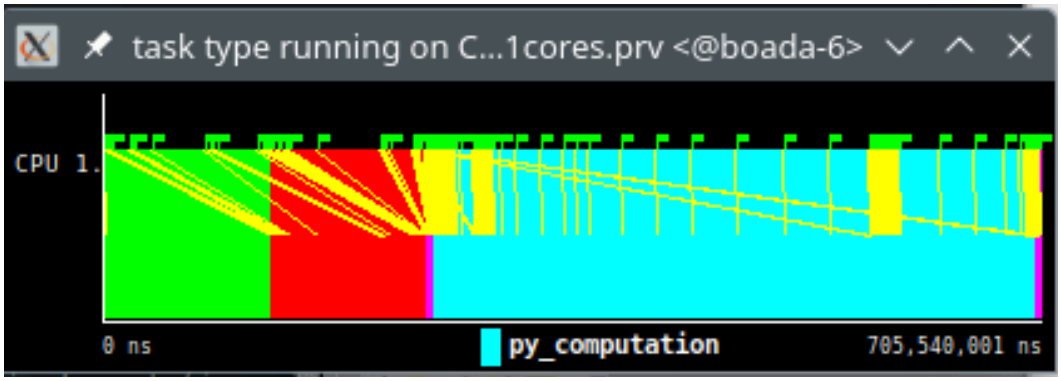


Figure 1: task running on 1 core.

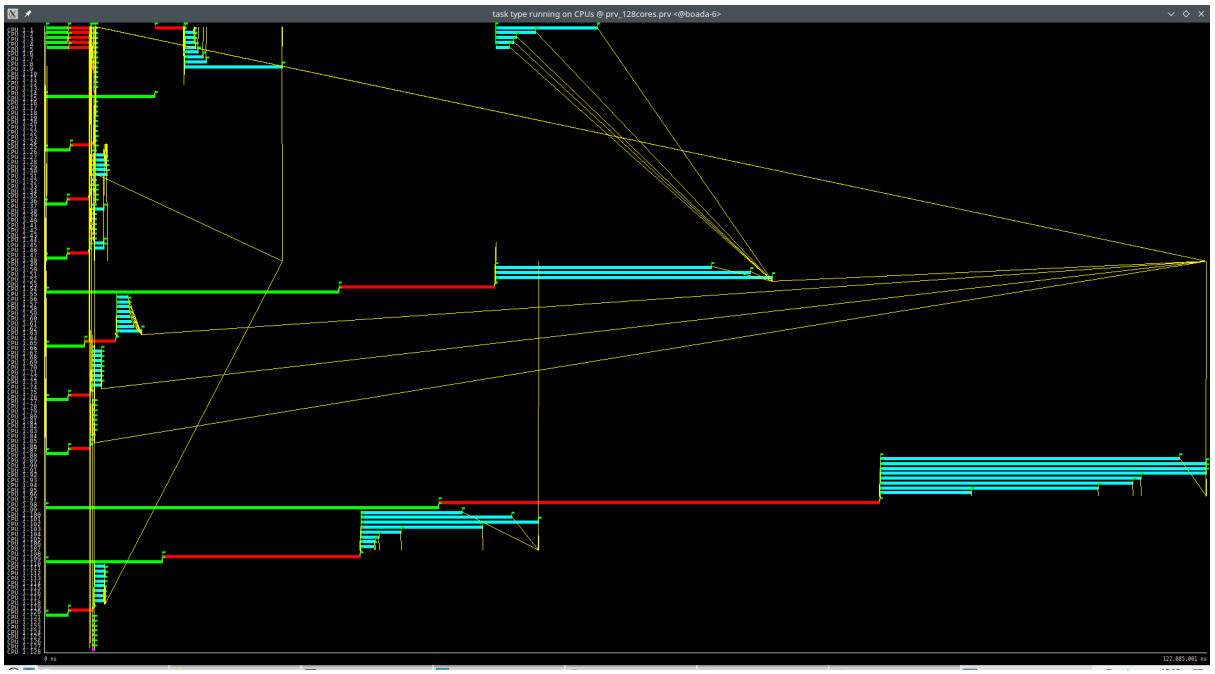


Figure 2: task running on 128 cores.

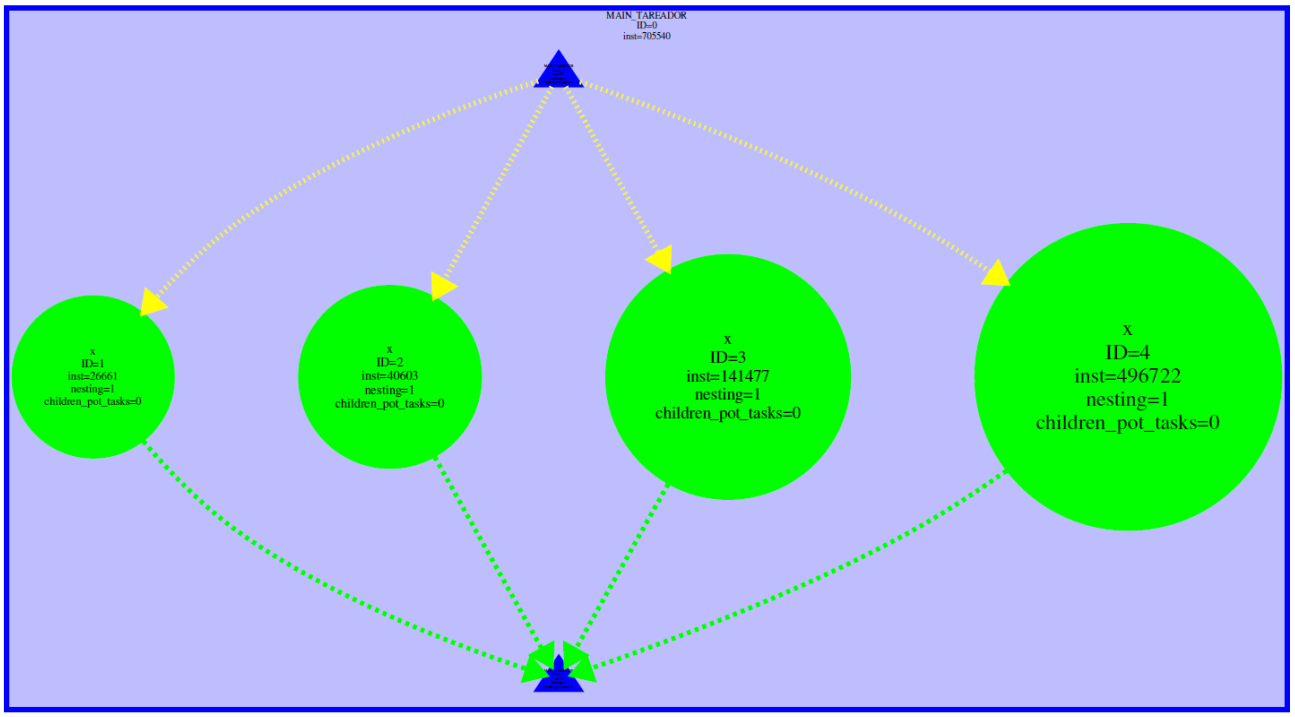
We can see in the above image, the comparison between T1 and T∞. We simulated with 128 processors, and due to the fact that there are so many fine grained tasks almost all of them were used. Still, we can observe a really high Load Unbalance.

T_1 (ns)	T_∞ (ns)	Parallelism	Load Unbalance (Yes/No) Why?
705540001	122085001 (128proc)	5.78	Yes, because of task size

Column

Code: mandel-seq-iter-tar-column.c

Tareador TDG (with dependencies)



Dependence analysis

This is an embarrassingly parallel code, with no dependencies between tasks.

```
if (output2histogram) {
        tareador_disable_object(&histogram[M[y][x]-1]);
        histogram[M[y][x] - 1]+=(TILE*TILE);
        tareador_enable_object(&histogram[M[y][x]-1]);
}
```

Tareador TDG (without "data sharing" dependencies)

As mentioned in the previous section, the original code is already parallelised, therefore there are no "data sharing" dependencies and the Tareador TDG looks exactly the same in both cases.

T_∞ analysis

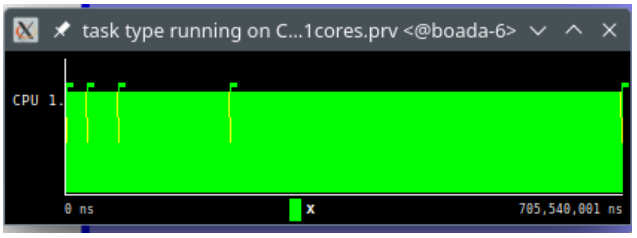


Figure 1: task running on 1 core.

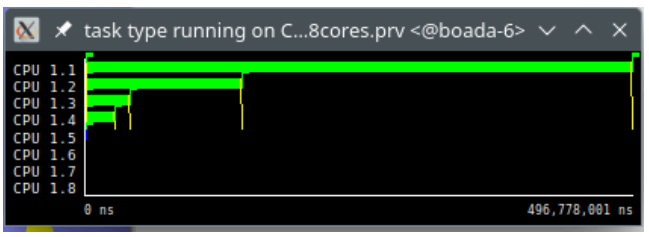


Figure 2: task running on 8 cores.

We can see in the above image, the comparison between T_1 and T_∞. We simulated with 8 processors, but only 4 were used by Tareador to reach its max parallelism, that is due to Load Unbalance.

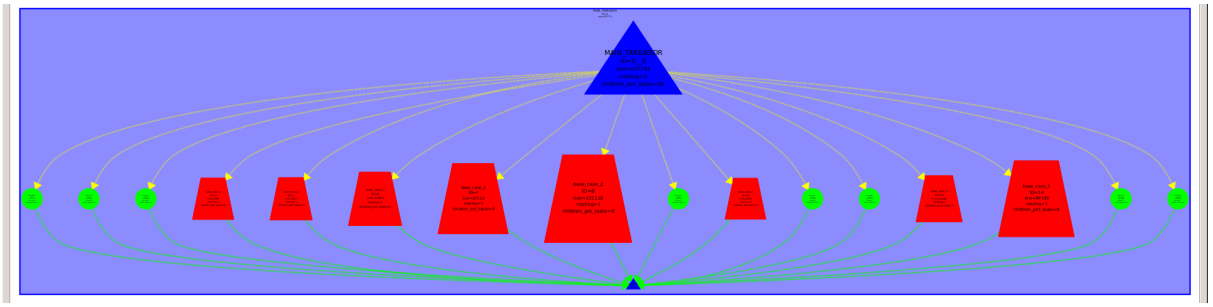
T_1 (ns)	T_∞ (ns)	Parallelism	Load Unbalance (Yes/No) Why?
705540001	496778001 (8proc)	1.42	Yes, because of task size

Recursive task decomposition analysis

Leaf

Code: mandel-seq-rec-leaf.c

Tareador TDG (with dependencies)



Dependence analysis

This code is embarrassingly parallelizable, therefore there are no dependencies between tasks.

Tareador TDG (without "data sharing" dependencies)

As mentioned in the previous section, the original code is already parallelised, therefore there are no "data sharing" dependencies and the Tareador TDG looks exactly the same in both cases.

T_∞ analysis

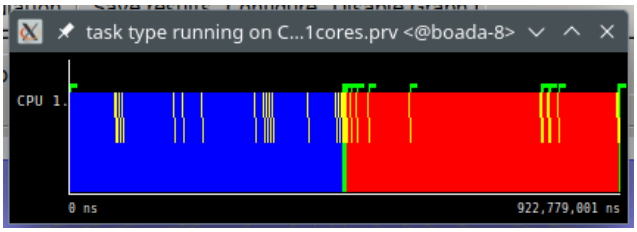


Figure 1: task running on 1 core.

Figure 2: task running on 32 cores.

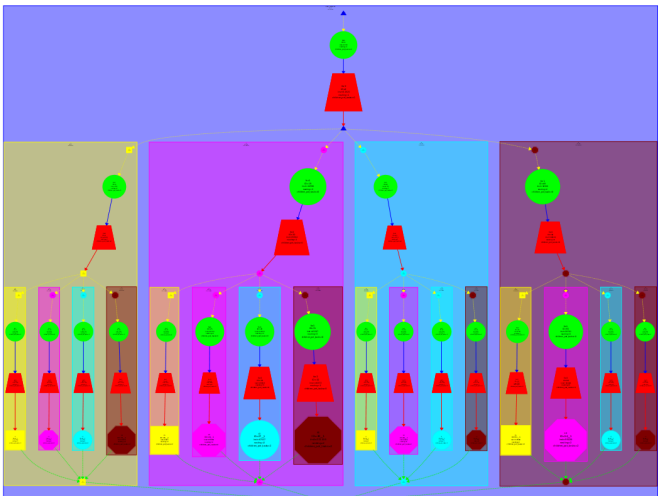
We can see in the above image, the comparison between T_1 and T_∞. We simulated with 32 processors, but only 16 were used by Tareador to reach its max parallelism, that is due to Load Unbalance.

T_1 (ns)	T_∞ (ns)	Parallelism	Load Unbalance (Yes/No) Why?
922779001	547331001	1.69	Yes, because of task size

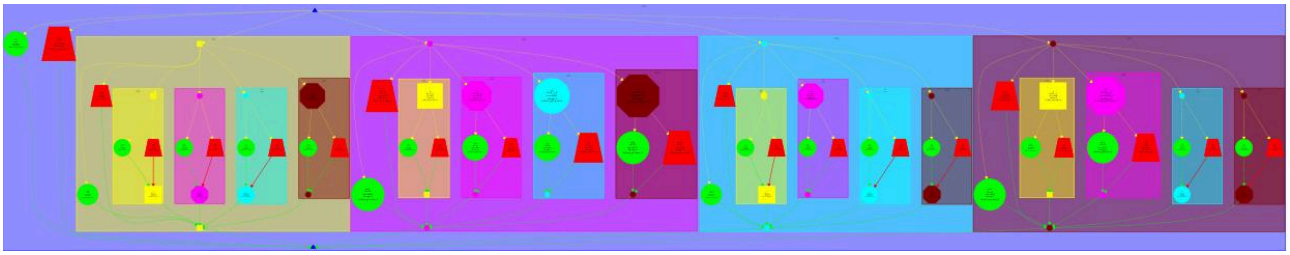
Tree

Code: mandel-seq-rec-tree.c

Tareador TDG (with dependencies)



Dependence analysis



T_∞ analysis

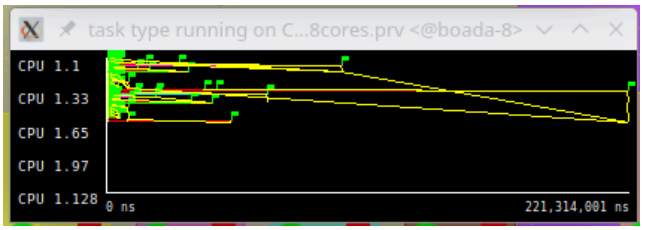


Figure 1: task running on 1 core.

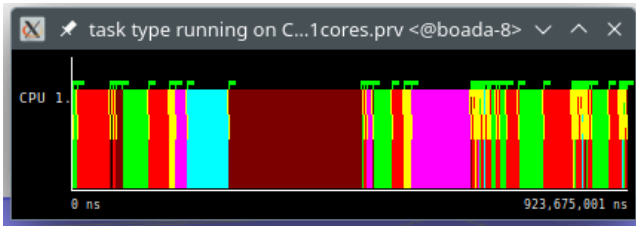


Figure 2: task running on 128 cores.

We can see in the above image, the comparison between T_1 and T_∞. We simulated with 128 processors, but only 64 were used by Tareador to reach its max parallelism, that is due to Load Unbalance.

T_1 (ns)	T_∞ (ns)	Parallelism	Load Unbalance (Yes/No) Why?
923675001	221314001	4,17	Yes, because of task size

Summary of the different performances

Task decomposition	**Strategy**	**Run arguments**	**T_1 (ns)**	**T_∞ (ns)**	**Parallelism**	**Load Unbalance (Yes/No) Why?**
Iterative	**Original**		705540001ns	308750001ns	2.285	Yes, because of task size
	Original	-d	729092001	503690001	1.45	Yes, because of task size
	Original	-h	727380001	504884001	1,44	Yes, because of task size
	Finer grain		705540001	122085001	5.78	Yes, because of task size
	Column		705540001	496778001	1.42	Yes, because of task size
Recursive	**Leaf**		922779001	547331001 (32proc)	1.69	Yes, because of task size
	Tree		923675001	221314001 (128proc)	4,17	Yes, because of task size