



# Trabajo Teórico: Herramientas de Análisis de Rendimiento – Intel VTune Profiler



**Pablo Rodríguez Solera / Juan Mena Patón**

## **Computadores Avanzados**

- ➔ Prof. Serafín Benito Santos.
- ➔ Escuela Superior de Informática.
- ➔ Universidad de Castilla – La Mancha

# Índice:

Trabajo Teórico: Intel VTune Profiler.....	3
1. Introducción:.....	3
2. Intel VTune Profiler:.....	3
2.1 Instalación:.....	3
2.2 Archivos necesarios.....	4
2.3 En la aplicación.....	5
2.3.1 Crear un proyecto.....	5
2.3.2 Ejecutando un análisis.....	7
2.3.3 Tipos de análisis.....	8
3. OpenMP - Mandelbrot.....	8
3.1 Explicación del programa.....	8
3.1.1 Funciones.....	8
3.1.2 Versión paralela.....	9
3.2 Análisis con la herramienta.....	9
3.2.1 Secuencial.....	10
3.2.2 Paralelo.....	10
3.2.3 Comparación de Cores.....	12
4. MPI - Renderizado distribuido.....	13
4.1 Funciones.....	13
4.2 Explicación del programa.....	13
4.3 Análisis con la herramienta.....	14
4.4 Mejora del programa.....	16
5. Conclusión.....	18

# Trabajo Teórico: Intel VTune Profiler.

## 1. Introducción:

El análisis del rendimiento de una aplicación paralela puede resultar muy interesante debido a toda la información que podemos obtener del mismo.

Con la información podemos identificar cuellos de botella dentro de nuestra aplicación, entender que uso del hardware disponible hace nuestra aplicación, los diferentes consumos que conlleva etc. Y en consecuencia, mejorar así o reparar los posibles fallos o puntos de mejora que identifiquemos.

Como primer caso de estudio, vamos a utilizar la herramienta Intel Vtune Profiler, anteriormente conocida como Intel Vtune Amplifier. Esta herramienta nos va a permitir analizar aplicaciones con componente paralelo tanto MPI como OpenMP y visualizar diferentes datos de la ejecución.

## 2. Intel VTune Profiler:

Esta herramienta está diseñada por intel y nos permite realizar muchas funciones en relación con lo explicado más arriba.

### 2.1 Instalación:

El primer paso es instalarla. Está disponible para la mayoría de sistemas operativos. En nuestro caso hemos decidido instalarla en un Linux nativo de 64 bits (debian).

Para poder descargarla hace falta una licencia, nosotros hemos obtenido la licencia de estudiante y desde la página de intel, solo hace falta registrarse y te permite descargarla según tu distribución.

Una vez descargada, simplemente descomprimos el fichero y nos encontramos 2 archivos de instalación, una versión para consola de comandos y otra con interfaz gráfica.

Nosotros hemos escogido la opción con interfaz gráfica debido a que es un proceso más fácil e intuitivo.

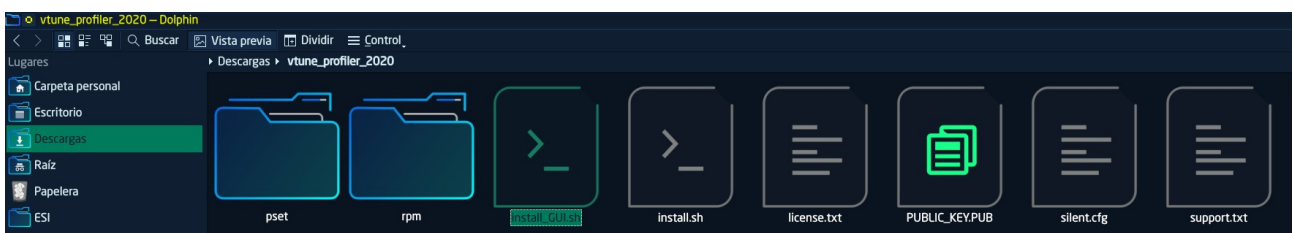


Figura 1: Contenido de la carpeta al descomprimir el fichero.

Para realizar la instalación es recomendable realizarla con permisos de superusuario, ya que aunque nos deja hacerlo como usuario normal nos avisa de que es recomendable realizarlo como superusuario para evitar fallos por permisos en alguna creación de ficheros o carpetas.

Al ejecutar el instalador se nos abre una ventana clásica de instalación guiada por pasos:

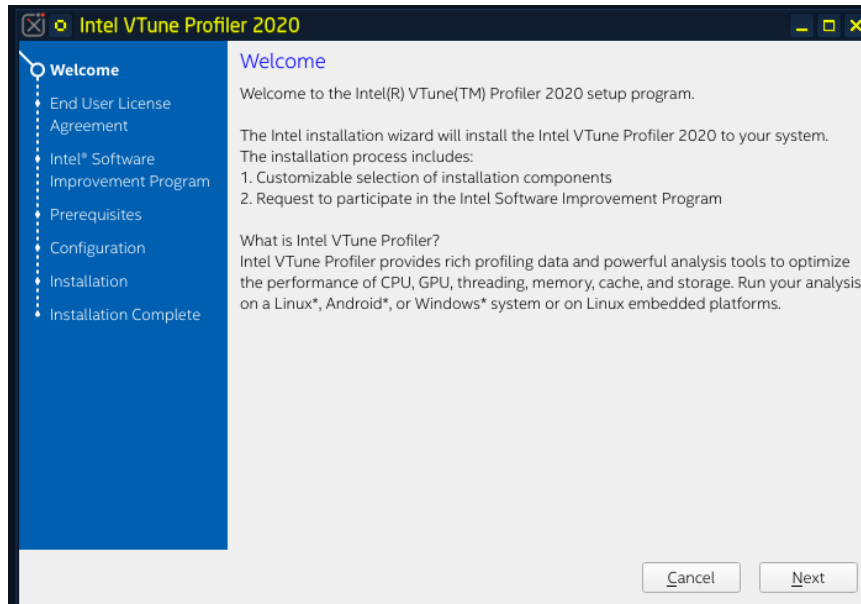


Figura 2: Ventana de instalación Vtune.

Vamos siguiendo los pasos que nos va describiendo el instalador, como elegir la carpeta de destino o informarnos de lo que se va a instalar.

Antes de proceder con la instalación tenemos que aceptar los términos de usuario y si queremos dar información a intel sobre nuestras pruebas.

Después se lleva a cabo la instalación y al finalizar si todo ha ido como debería, nos informa de que ya está instalada y podemos comenzar a utilizarla.

## 2.2 Archivos necesarios

Para poder realizar el análisis de nuestros programas con VTune no es necesaria ninguna configuración especial.

Por ejemplo en el caso de un programa con paralelismo OpenMP, simplemente necesitamos el archivo fuente con el código del programa y el programa compilado con la opción de depuración, es decir con el flag -g (a parte de los necesarios para compilar el programa). De esta forma un ejemplo sería:

```
➔ gcc prueba.c -o prueba -fopenmp -g
```

## 2.3 En la aplicación

Si estamos en una distribución linux, la instalación por defecto se realiza en la carpeta opt de la raíz. La instalación incluye un archivo .desktop para poder incluirla en nuestro registro de aplicaciones, pero no lo hace por defecto por lo que para iniciarla debemos entrar en la carpeta de instalación. Entramos y nos encontramos con el fichero ejecutable dentro de la carpeta de 64 o 32 bits según la arquitectura de nuestro equipo.

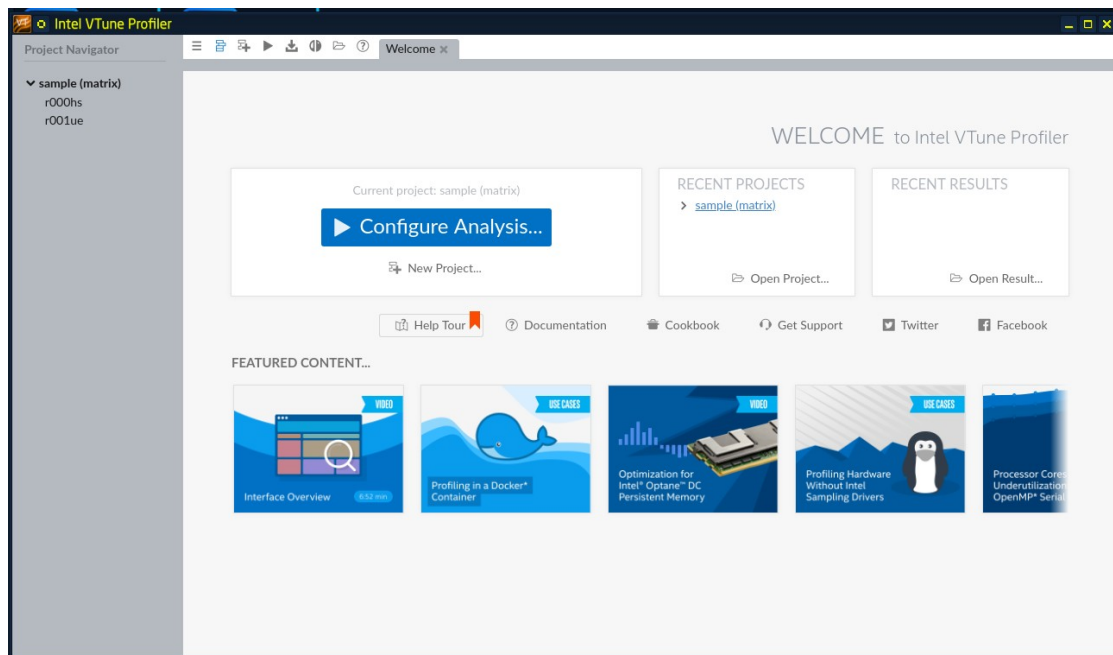


Figura 3: Pantalla de bienvenida de VTune

Iniciamos el programa y nos encontramos con la pantalla de bienvenida.

### 2.3.1 Crear un proyecto

Crear un proyecto con VTune es fácil con la interfaz gráfica. Dentro de la ventana de bienvenida nos encontramos con el botón de crear un nuevo proyecto, al pulsarlo, se abre una ventana como la siguiente:

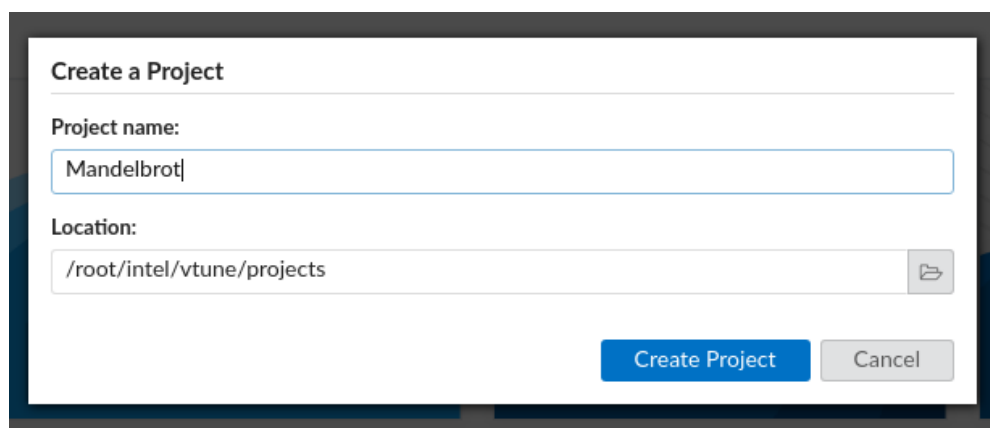


Figura 4: Creación de un proyecto

Ponemos nombre a nuestro proyecto y lo creamos.



Una vez se crea se nos muestra una pantalla con información del proyecto y algunas opciones y configuraciones que podemos realizar sobre el mismo.

Lo primero que tenemos que hacer es añadir el fichero compilado de la aplicación que queremos analizar.


Para ello seleccionamos el fichero en la sección “Application”.

Si nuestro archivo necesita algún parámetro especial en su ejecución debemos añadirlo en la sección “Application parameters”. En nuestro caso solo es un fichero con datos de entrada que necesita nuestra aplicación.

Application:

/home/s723/ESI/CUARTO/(ARCO) ARQUITECTURA DE COMPUTADORES/LABORATORIO/POpenMP-Mand  

Application parameters:

entrada| 

☒ Use application directory as working directory

Advanced ▶

Figura 5: Añadir la aplicación al proyecto.

Al VTune Profiler le hace falta también el archivo con el código del programa sin compilar para poder informarnos sobre las funciones dentro del mismo. Para dárselo tenemos que irnos a la opción “Search Sources/Binaries” que se encuentra abajo junto al botón de iniciar.

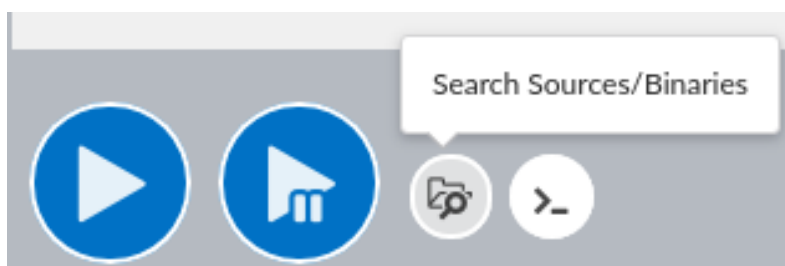


Figura 6: Botón de añadir binarios.

Aquí podemos ver los botones del programa que nos permiten ejecutar un análisis, además, esta sección nos permite añadir fuentes o binarios, en nuestro caso queremos añadir los fuentes, así que entramos en la sección “Sources” y pulsamos a la derecha el botón de buscar carpeta.

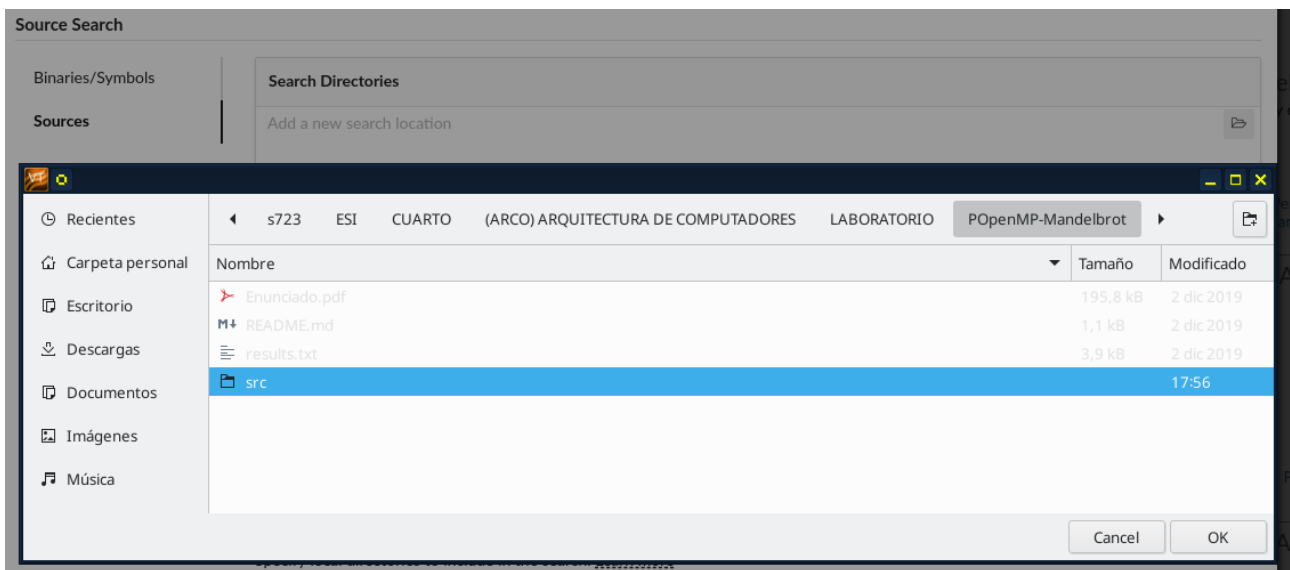


Figura 7: Añadimos la carpeta con los fuentes del programa.

Aquí buscamos la carpeta que contiene los archivos de nuestro programa y la seleccionamos. VTune se encarga solo de buscar que archivo corresponde a nuestro ejecutable, así que seleccionamos la carpeta src entera.

Una vez tenemos todo preparado, solo nos queda ejecutar el análisis para empezar a ver los resultados y la información sobre nuestra aplicación.

### 2.3.2 Ejecutando un análisis

Después de tener todo configurado, podemos lanzar el análisis. La herramienta nos permite lanzarlo en local o en algún equipo remoto, por ejemplo mediante el protocolo ssh de linux, pero en nuestro caso lo ejecutamos en local.

Para empezar el análisis, simplemente tenemos que pulsar el botón de “Play” azul que tenemos abajo a la izquierda. Una vez lo pulsemos el programa empezará a ejecutar nuestra aplicación.

Una vez ejecutemos, podremos ver como se va ejecutando el análisis y las fases que va recorriendo hasta finalizar.

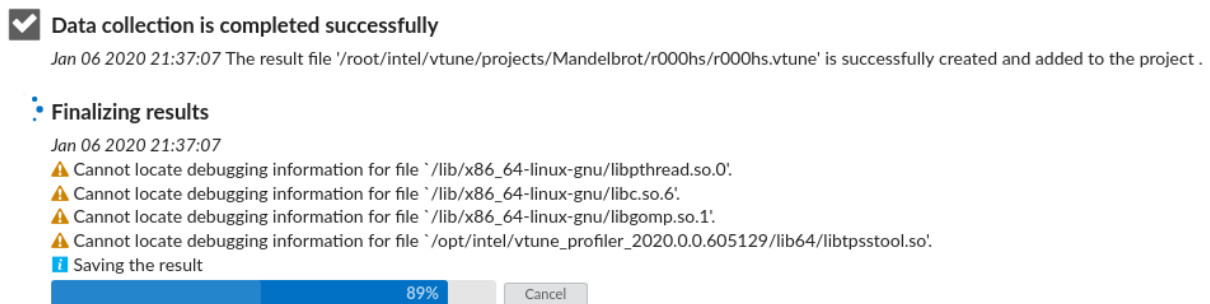


Figura 8: Ejecutando el análisis.

### 2.3.3 Tipos de análisis

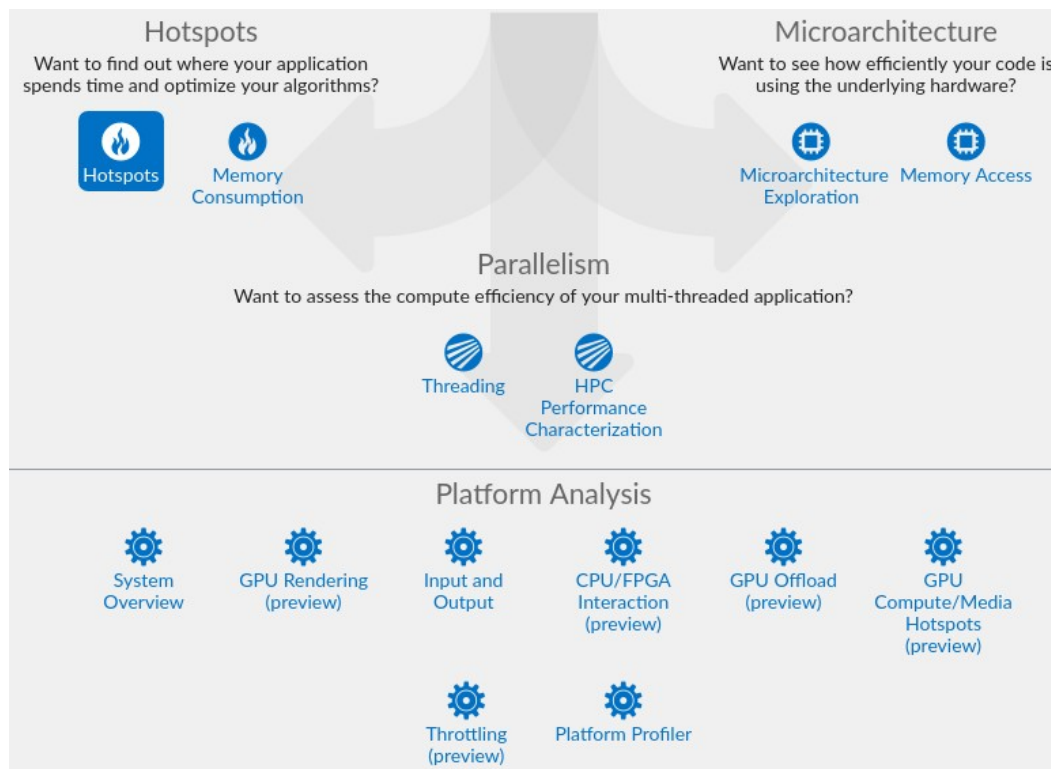


Figura 9: Tipos de análisis.

La herramienta Intel VTune Profiler nos da la opción de realizar muchos tipos de análisis predefinidos que trae de por sí. Por ejemplo, analizar los puntos calientes, el paralelismo, el uso de memoria etc:

## 3. OpenMP - Mandelbrot

Para realizar los análisis de rendimiento, vamos a utilizar como aplicación la correspondiente a la práctica final de Arquitectura de Computadores de este curso, que se corresponde con una aplicación que busca números complejos que correspondan a los fractales dentro del conjunto de Mandelbrot.

Este programa está escrito en C y tenemos varias versiones que vamos a analizar.

En primer lugar vamos a realizar un análisis de la versión con el programa secuencial y después con las versiones del programa paralelo.

### 3.1 Explicación del programa

#### 3.1.1 Funciones

El programa como ya hemos explicado busca números complejos dentro del conjunto Mandelbrot y después imprime una serie de imágenes del conjunto.



Vamos a realizar la explicación de las funciones del programa así como de las comunicaciones que se producen entre los procesos en la versión paralela.

- `main()`: La función principal del programa, se encarga de abrir el fichero con los límites de cada una de las imágenes en las cuales se va a realizar la búsqueda del conjunto y que queremos generar.
  - Lee el tipo de imagen, si es circular o rectangular.
  - Calcula la anchura y la altura de la imagen en píxeles.
  - Calcula la coordenada (x,y) y decide si se incluye en el conjunto mandelbrot.
  - Determina el color del píxel y lo imprime en el archivo de imagen que ha creado.
- `mandel_val()`: Es la función que se determina si el número complejo (x,y) a comprobar tiene posibilidades de pertenecer al conjunto de Mandelbrot o rotundamente no pertenece al mismo. Esta función es en la que se busca mejorar el rendimiento en la versión paralela del programa.

### **3.1.2 Versión paralela**

En la versión paralela, tenemos que crear una serie de variables tanto públicas como privadas para cada proceso, además, utilizamos las directivas correspondientes de OpenMP para paralelizar la sección de código que necesitamos.

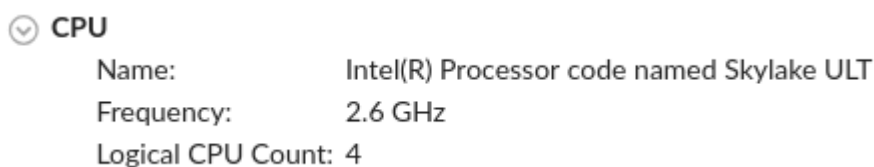
En este caso se paraleliza la directiva `for` que va recorriendo todos los píxeles y comprobando que números del plano (x,y) corresponden al conjunto.

En esta versión, simplemente hacemos uso de la directiva `parallel`, separamos las regiones de la imagen que le tocan a cada proceso y le indicamos que las variables `x` e `y` correspondientes a los píxeles son privadas.

En este caso la comunicación entre procesos es inexistente, puesto que el proceso primitivo calcula que región le toca a cada proceso hijos y estos solo modifican información dentro de su región por lo que no se comunican entre ellos.

## **3.2 Análisis con la herramienta**

Vamos a realizar el análisis del programa con la herramienta explicada, como tenemos 3 versiones, vamos a separar los análisis de cada una de ellas, en este caso con un procesador de 4 núcleos, información que también nos da la herramienta.



*Figura 10: Procesador que analiza el mandelbrot.*

### 3.2.1 Secuencial

En primer lugar, realizamos el análisis de la versión secuencial, es decir, no existe paralelismo. Como podemos ver con las imágenes a continuación, que resultan clave:

Elapsed Time: 84.544s  
CPU Time: 84.468s  
Total Thread Count: 1  
Paused Time: 0s

Parallelism: 25.0%

Figura 12: Solo un 25% de Paralelismo, 1/4 de los procesadores al ser secuencial.

Figura 11: El tiempo de ejecución es muy grande al ser secuencial.

Al ser una versión secuencial, todo el trabajo del programa queda realizado por un solo núcleo, que realiza todo el trabajo y carga el peso de la función principal que como podemos observar es la que lleva la mayoría del tiempo de ejecución:

Process / Function / Thread / Call Stack	Effective Time by Utilization
	Idle Poor Ok Ideal Over
mandelbrot_secuencial	84.468s
main	0.080s
mandel_val	83.712s

Figura 13: Función principal en versión secuencial

A continuación, hemos realizado un análisis "Threading", que nos permite ver en que puntos, en este caso objetos, el programa pasa más tiempo estancado debido a la poca paralelización.

En la siguiente imagen, vemos que los objetos que más tiempo pasan esperando en alguna función son los streams que se usan para imprimir las imágenes que generamos a partir del programa.

Sync Object / Function / Call Stack	Wait Time by Utilization	Wait Count
	Idle Poor Ok Ideal Over	
Stream cuadrado2.ppm 0x1be03f11	177.311ms	1,512
Stream cuadrado1.ppm 0x1be03f11	117.279ms	877
Stream 0xf7669f4d	87.512ms	638
Stream rectangulo.ppm 0x1be03f11	30.323ms	259

Figura 14: Objetos que más tiempo esperan en versión secuencial.

### 3.2.2 Paralelo

Ahora vamos a analizar la primera versión del programa paralelo y, nada más empezar el análisis ya vamos observando que hay una mejoría, el análisis tarda menos, debido a que el tiempo de ejecución es menor como vemos nada más obtener los resultados:

Elapsed Time: 4.371s  
CPU Time: 10.900s  
Total Thread Count: 4  
Paused Time: 0s

Parallelism: 62.3%

Figura 16: También, obviamente mejora el porcentaje de paralelismo.

Figura 15: Mejoría de tiempo en la versión paralela.

Con estos datos podemos ver el potencial que nos ofrece el paralelismo, pero para profundizar en el análisis, centrémonos en la función `mandel_val()` que es la que como hemos dicho tiene el peso superior del programa:

mandel_val	10.814s
mandelbrot_para (TID: 28945)	0.282s
mandelbrot_para (TID: 28968)	2.030s
mandelbrot_para (TID: 28969)	4.220s
mandelbrot_para (TID: 28970)	4.282s

Figura 17: Función `mandel_val()` en versión paralela.

Aquí vemos como en este caso, la función se ha repartido entre 4 procesos, uno por núcleo, de forma que mejora el tiempo del programa.

Otra de las opciones que nos ofrece el programa es ver dentro del mismo código un desglose de los datos de uso de los procesadores por cada función.

Source	CPU Time: Self	Effective Time by Utilization			
		Idle	Poor	Ok	Ideal
int mandel_val(double x, double y, int max_iter){					
int j = 1;					
double temp_real, cr, p_real = x;					
double ci, p_imag = y;					
cr = x*x; ci = y*y; // cr+ci es el cuadrado del módulo de x + i.y					
while (((cr+ci)<=4) && (j < max_iter)){	3.946s	0.0%	2.2%	32.8%	3.8%
temp_real = cr - ci + x;	0.626s	0.0%	0.4%	5.3%	0.5%
p_imag = 2 * p_real * p_imag + y;	0.602s	0.0%	0.1%	5.4%	0.4%
p_real = temp_real;	0.934s	0.0%	0.3%	8.1%	0.8%
j++; // z(j) = p_real + i * p_imag	0.726s	0.0%	0.4%	5.9%	0.8%
cr = p_real * p_real; ci = p_imag * p_imag;	3.206s	0.0%	1.2%	27.2%	3.1%
}					
if((cr+ci) <= 4) return -1; else return j;	0.008s	0.0%	0.0%	0.1%	0.0%
};					

Figura 18: Datos de la función `mandel_val()`

Aquí analizamos la función `mandel_val()` que es la principal.

Podemos ver el uso de 1, 2-3 o los 4 cores que realiza cada una de las instrucciones o partes de la función

A continuación, hemos realizado un análisis de tipo “MicroArchitecture Exploration” que nos ha permitido ver como se comporta el programa en la microarquitectura de este equipo.

El siguiente diagrama muestra un montón de datos en relación a esto:

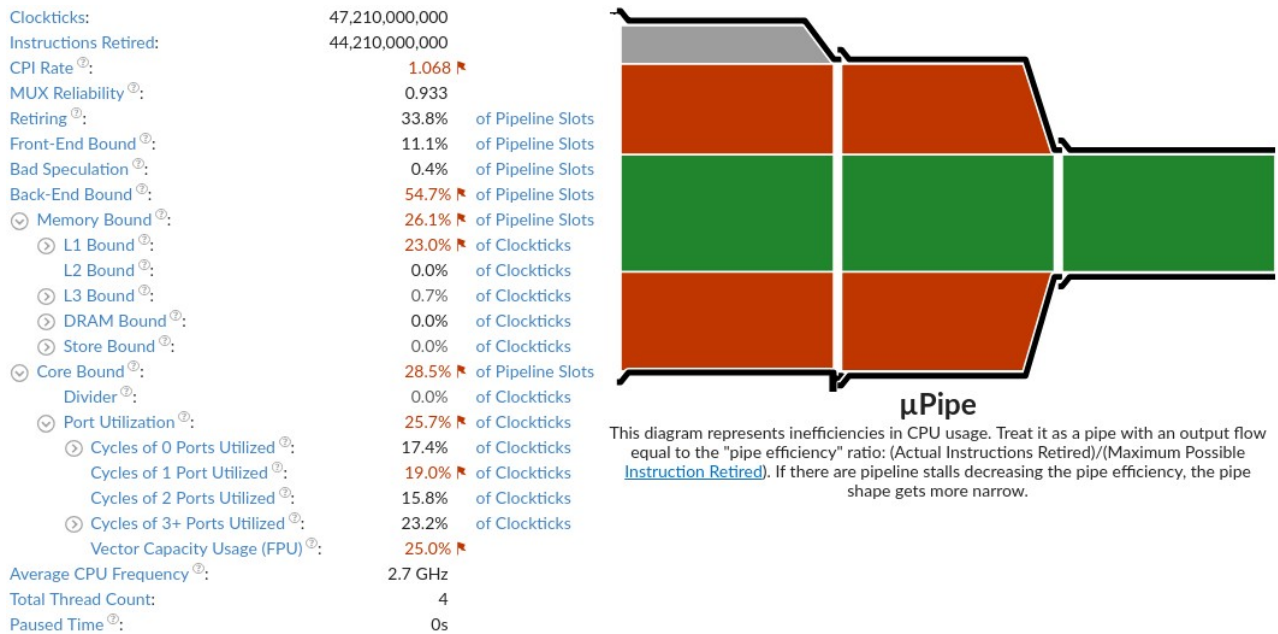


Figura 19: Uso de Microarquitectura en Mandelbrot Paralelo.

Podemos ver los ciclos de reloj, instrucciones, CPI, diferentes datos sobre la memoria o la media de la frecuencia de CPU.

### 3.2.3 Comparación de Cores

Vamos a comparar por último los histogramas de las versiones secuencial y paralela del programa. Observamos la clara diferencia del uso de procesadores en cada una de ellas, donde la secuencial no aprovecha paralelismo alguno mientras la paralela sí:

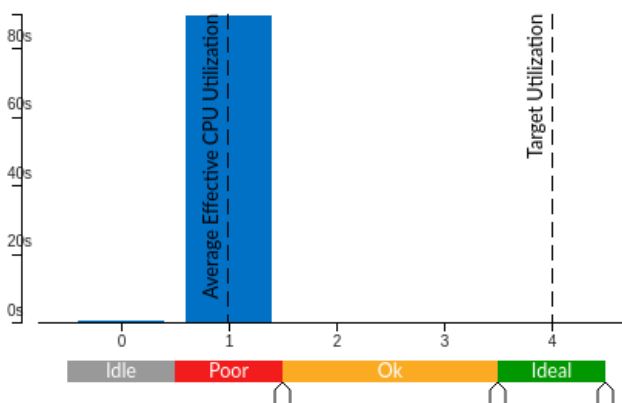


Figura 21: Histograma de la versión secuencial.

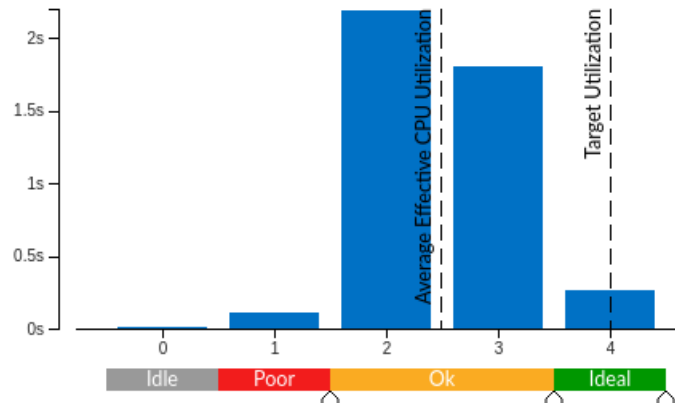


Figura 20: Histograma de la versión paralela.

De este modo terminamos el análisis de la parte del programa Mandelbrot y damos paso a un nuevo análisis que explorará el potencial de esta herramienta en caso de MPI.

## **4. MPI - Renderizado distribuido**

Otro ejemplo que vamos a usar es un renderizado de imágenes distribuido en el que vamos a utilizar MPI en lugar de OpenMP.

Este programa está escrito en C al igual que el Mandelbrot.

En este caso solamente tenemos una versión paralela que es la que vamos a analizar.

Para este análisis, vamos a utilizar un procesador con 8 núcleos.

### **4.1 Funciones**

Para este programa, como ya hemos dicho, vamos a hacer uso de MPI y para entender mejor el funcionamiento del mismo vamos a explicar sus funciones.

- `initX()`: Es la función que se encarga de crear la ventana en la cuál se irán mostrando los píxeles de la imagen una vez calculados.
- `dibujaPunto()`: Es la función que se encarga de ir imprimiendo los píxeles por pantalla una vez que le lleguen al proceso encargado de ello.
- `main()`: La función principal del programa, en ella se crean los procesos encargados de realizar el renderizado y reparten las tareas para cada uno de ellos.

### **4.2 Explicación del programa**

Este programa va a tener 2 tipos de procesos, un tipo que solamente tendrá acceso a la pantalla pero no al disco y otro tipo que tendrá los accesos contrarios, si tendrá acceso al disco pero no a la pantalla.

Inicialmente solamente lanzaremos un proceso que va a tener acceso a la pantalla de gráficos. Él mismo será el encargado de lanzar a los demás procesos que tendrán el acceso al disco.

Esto se hace utilizando la directiva `MPI_Comm_spawn` que se usa para crear varias instancias de una sola aplicación MPI.

Una vez hecho esto, este proceso se queda esperando mediante un `MPI_Recv` a que los demás procesos le manden los datos que irá imprimiendo en pantalla a través de la función `dibujaPunto()`.

Los procesos lanzados mediante `MPI_Comm_spawn` son los encargados de leer de forma paralela los datos del archivo `datos.dat`. Una vez hecho esto, se encargarán de ir enviando los píxeles al proceso que tiene el acceso a la pantalla para que los vaya mostrando.

Lo primero que hace cada proceso es calcular cuantas filas del archivo se le van a asignar, cuantos elementos va a procesar y donde empieza y donde termina su parte del fichero.

Una vez calculado esto, abre el fichero utilizando la directiva `MPI_File_open` y accede a su parte asignada mediante `MPI_File_set_view`.

A continuación lee del archivo el valor de los píxeles utilizando `MPI_File_read` y los guarda en un array. Una vez que los tiene guardados, les aplica el filtro que hayamos seleccionado y los guarda en un buffer junto con las coordenadas x e y del pixel.

Ese buffer es el que le va a mandar, utilizando la primitiva `MPI_Bsend` para asegurarnos que los datos llegan, al proceso que posee acceso a la pantalla para que los muestre.

El archivo que se proporciona para trabajar con este programa (`datos.dat`) es un fichero de 400 filas por 400 columnas de puntos. Cada uno de estos puntos están formados por 3 unsigned char que corresponden a los valores R, G y B.

### 4.3 Análisis con la herramienta

Debido a la poderosa funcionalidad de la herramienta Intel VTune solo vamos a ver algunas de sus funciones ya que sino la extensión del trabajo sería demasiada.

Para este ejemplo vamos a ver datos del programa así como también su rendimiento.

En la pestaña *Summary* podemos observar los siguientes parámetros y métricas.



Figura 9: Tiempos de ejecución MPI.

En esta imagen podemos ver el tiempo de ejecución del programa que es el *Elapsed Time*.

El valor de *CPU Time* es el tiempo en el cuál la CPU está activamente ejecutando el programa. Mientras que el *Effective Time* es el tiempo de CPU que se emplea para ejecutar el código del usuario.

*Spin Time* es el tiempo tiempo de espera durante el cuál la CPU está ocupada. Como podemos ver está marcada y esto significa que es un aspecto crítico del programa que debemos solucionar.

*Total Thread Count* es el número de llamadas o directivas que se ejecutan durante el programa.

### Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time <sup>②</sup>
PMPI_Bsend	libmpi.so.20	5.552s
XAllocColor	libX11.so.6	5.088s
XFlush	libX11.so.6	2.232s
MPI_Init	libmpi.so.20	0.436s
MPI_Recv	libmpi.so.20	0.430s
[Others]		1.382s

\*N/A is applied to non-summable metrics.

Figura 10: Hotspots MPI.

Aquí podemos ver que llamadas o instrucciones son las que más tiempo consumen de nuestra aplicación. En este caso vemos que la instrucción MPI\_Bsend es la que más tiempo consume.

### Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

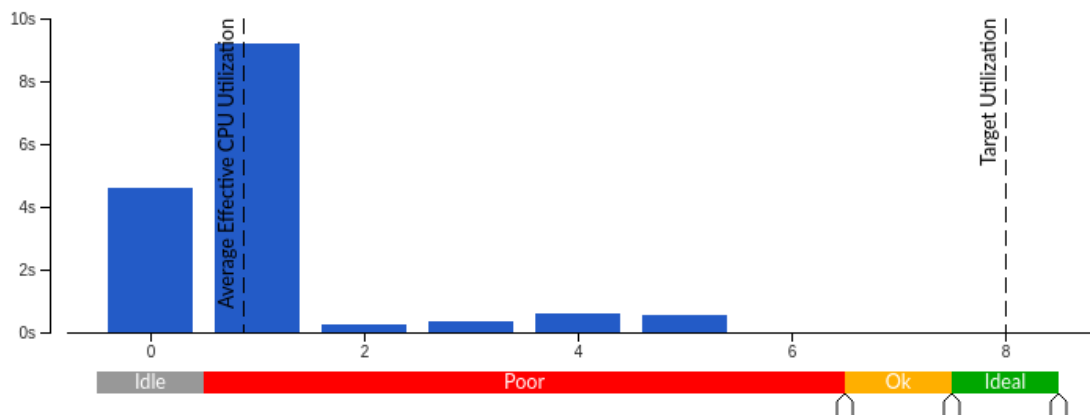


Figura 11: Histograma MPI.

Con esta imagen podemos tener una vista general de cuanto tiempo han estado funcionando los distintos procesos de forma simultánea. Aquí vemos que disponemos de 8 CPUs para ejecutar nuestro programa por lo que esto puede ser una manera de mejorarlo ya que podemos observar que con nuestro código solamente se utilizan 5 CPUs.



Grouping: Function / Call Stack					
Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
PMPI_Bsend	5.552s	libmpi.so.20	PMPI_Bsend		0x59280
XAllocColor	5.088s	libX11.so.6	XAllocColor		0x22eb0
XFlush	2.232s	libX11.so.6	XFlush		0x1f730
MPI_Init	0.436s	libmpi.so.20	MPI_Init		0x68240
MPI_Recv	0.430s	libmpi.so.20	MPI_Recv		0x704a0
MPI_File_read	0.408s	libmpi.so.20	MPI_File_read		0x86a20
orte_daemon	0.148s	libopen-rte.so.20	orte_daemon		0x32710
PMPI_File_close	0.136s	libmpi.so.20	PMPI_File_close		0x82a20
OS_BARESYSCALL_DoCallAsmIntel64Li	0.135s	libc-dynamic.so	OS_BARESYSCALL_DoCallAsmIntel64Linux		0x72d40
__sprintf	0.100s	libc.so.6	__sprintf	sprintf.c	0x65000
ompi_mpi_finalize	0.086s	libmpi.so.20	ompi_mpi_finalize		0x47cf0
XDrawPoint	0.072s	libX11.so.6	XDrawPoint		0x1d600
MPI_Comm_spawn	0.050s	libmpi.so.20	MPI_Comm_spawn		0x5f450
[Outside any known module]	0.040s		[Outside any known module]		0
MPI_File_open	0.030s	libmpi.so.20	MPI_File_open		0x857a0

Figura 11: Llamadas y directivas MPI.

En la pestaña *Bottom-up* podemos ver todas las llamadas y directivas que se ejecutan en el programa junto con su tiempo de CPU, el módulo al cuál pertenece y su dirección de memoria.

Además podemos agrupar los datos de muchas otras maneras según nos interese la forma de mostrar los datos. Para ello solamente tenemos que elegir otra forma de agrupación en el campo “grouping”.

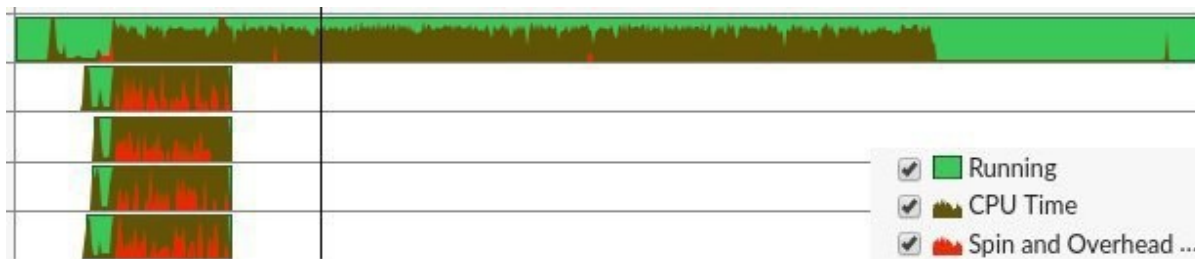


Figura 12: Ejecución procesos MPI.

Esta herramienta también nos permite ver el comportamiento de los distintos procesos con todos sus tiempos.

En la pestaña *Platform* podemos ver esta información pero a mayor escala y de una forma más clara.

## 4.4 Mejora del programa

Como hemos visto en el histograma de utilización, hay CPUs que no se utilizan y por tanto el programa no es todo lo eficiente que podría ser.

Para cambiar esto vamos a modificar el código haciendo que cuando el proceso que tiene el acceso a disco cree a los demás, en lugar de lanzar 4 procesos hagamos que lance 7. Con esto podremos tener a todas las CPUs ocupadas y el rendimiento y el tiempo de ejecución del programa debería mejorar.





Figura 12: Tiempos de ejecución MPI versión mejorada.

Con este cambio, vemos que todos los tiempos se reducen debido a que ahora tenemos más CPUs y por lo tanto menos datos que procesar por cada una.

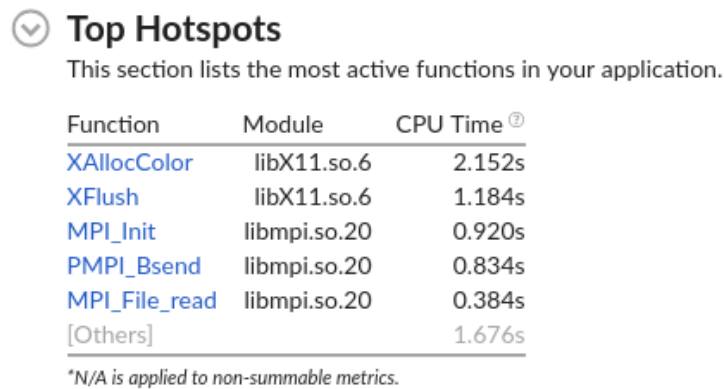


Figura 13: Hotspots MPI versión mejorada.

En cuanto a las llamadas o instrucciones más utilizadas vemos que ahora la directiva MPI\_Bsend ya no es la que más tiempo consume, esto es el cambio más significativo que podemos observar ya que pasamos de consumir 5.5 segundos a reducirlos a 0.8 segundos.

Esto se debe a que pese a que tenemos las mismas llamadas a MPI\_Bsend, ahora al tener mayor paralelismo se pueden realizar más operaciones en el mismo tiempo.

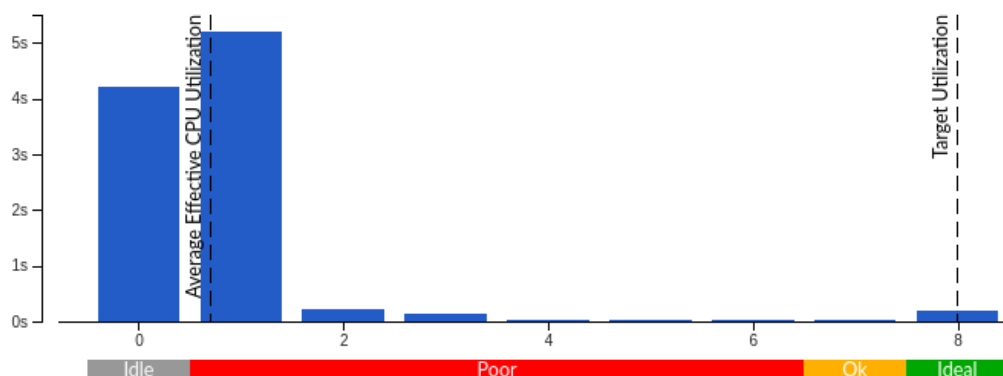


Figura 14: Histograma MPI versión mejorada.

En el histograma podemos comprobar que aunque no es durante mucho tiempo, debido al funcionamiento de nuestro programa, las 8 CPUs funcionan durante algún tiempo de forma simultánea.



Figura 15: Comparativa ejecución efectiva MPI.

Al realizar este cambio lo que también observamos es que tenemos una utilización efectiva de CPU menor y esto se debe a que al utilizar 8 CPUs en lugar de 5, estas se utilizan durante menor tiempo y esto hace que la eficiencia baje aunque nuestro programa tenga menor tiempo de ejecución.

## 5.Conclusión

Como podemos observar, con la herramienta Intel VTune Profiler, podemos observar a simple vista muchos datos e información sobre nuestras aplicaciones que nos permiten comprender más rápido y de forma más intuitiva como funciona el paralelismo dentro de las mismas y como podríamos mejorarlo o incluso implementarlo en caso de que no existiera.

Este programa nos brinda una infinidad de oportunidades, tiene un montón de tipos de análisis y muchas secciones que nos ayudan con la interpretación de los resultados, desde por ejemplo ver cuanto tiempo tarda cada función del código hasta cuantos cores del procesador estamos utilizando pasando por el uso de la microarquitectura.

Como aclaración final, los resultados que hemos expuesto en este trabajo pueden variar según el equipo donde se ejecute el programa y las distintas circunstancias en las que se lance el programa, ya que no es lo mismo un equipo con 8 núcleos que uno con 4 como hemos visto, y tampoco es igual un equipo que solo ejecute este programa a la hora del análisis como uno que se encuentre realizando otras tareas al mismo tiempo.