Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

Trabajo Práctico Final: DeliriOS

Organización del Computador II

Marzo 2014

Apellido y Nombre	LU	E-mail
Silvio Vileriño	106/12	svilerino@gmail.com
Ezequiel Gambaccini	715/13	ezequiel.gambaccini@gmail.com

Repositorio del Proyecto:
`https://github.com/Izikiel/intel_multicore`

Índice

1. Introduccion

El objetivo de este trabajo práctico fue inicialmente experimentar con la arquitectura intel, realizando un microkernel de 64 bits inicializando multinúcleo. Más tarde en el desarrollo del proyecto se decidió extender el alcance del mismo, realizando algunos experimentos para analizar las posibles mejoras de rendimiento de algoritmos que se pueden paralelizar en varios núcleos, asimismo se realizaron varios enfoques diferentes en la sincronizacion entre núcleos: espera activa haciendo polling a memoria versus sincronizacion con interrupciones inter-núcleo que evitan los accesos al bus de memoria. La ganancia que esperamos obtener, es una reducción considerable en el overhead que genera el manejo de varios hilos sobre sistemas operativos utilizando librerias como por ejemplo pthreads, y de esta forma poder determinar si podría aprovecharse de mejor manera el hardware disponible para resolver problemas de mayor tamaño que el que nos permiten las librerias actuales para multihilo.

El informe estará dividido en secciones, cada una describiendo una parte del trabajo.

2. Desarrollo: Inicialización y contexto del sistema

2.1. Estructura de carpetas: Compilación, Linkeo y Scripts

- **ap_code:** Contiene el código de los Application processors, tanto de inicialización desde modo protegido a modo nativo x64 como parte del código de los algoritmos implementados para los experimentos.
- **ap_startup_code:** Contiene el código de inicializacion de modo real a modo protegido de los Application processors
- **bsp_code:** Contiene el código de booteo del kernel principal de nivel 2, parte del código de los algoritmos implementados, y el encendido por parte del BSP de los Application Processors.
- **common_code:** Contiene el código común al Bootstrap Processor y los Application Processors.
- **grub_init:** Contiene scripts y archivos de configuracion de grub, en la subcarpeta src se encuentra el primer nivel de booteo que realiza el pasaje entre la máquina en estado mencionado en la especificación multiboot al segundo nivel de booteo del BSP.
- **run.sh:** Script para compilación y distribución del tp.
- **macros:** Esta carpeta contiene macros utilizadas en el código.
- **informe:** Contiene el informe del trabajo práctico en formato Latex y PDF.
- **include:** Contiene las cabeceras de las librerias incluidas en el código.

El tp está compilado en varios ejecutables, de esta forma cargamos algunas partes como módulos de grub. Hay scripts encargados de compilar todo lo necesario, cada módulo tiene su Makefile y el comando make es llamado oportunamente por los scripts en caso de ser necesario.

El linkeo está realizado con linking scripts en las carpetas donde sea necesario, cada sección esta alineada a 4k por temas de compatibilidad, asimismo hay símbolos que pueden ser leidos desde el codigo si es necesario saber la ubicación de estas secciones en memoria. Los módulos de 32 bits estan compilados en formato elf32 y los de 64 bits en binario plano de 64 bits, esto se debe a temas de compatibilidad de grub al cargar modulos y kernels.

Para correr el trabajo practico unicamente hace falta tipear $./run.sh - r$ en consola y se recompilará automaticamente para luego ejecutarse en bochs.

2.2. Integración con grub y división en módulos

Utilizamos grub en el nivel mas bajo del booteo para lograr un contexto inicial estable y que posibilitara la ejecución del trabajo practico desde un pendrive usb.

Grub permite iniciar un kernel por medio de una especificación publicada en la web, en la que se detalla un contrato que debe cumplir tanto el kernel a iniciar como grub, entre ellas, un header que debe contener el ejecutable del kernel para ser identificado por grub como un sistema operativo, y por otro lado, las obligaciones que debe cumplir grub al entregarle el control a dicho kernel, es decir, un contexto determinado: selectores de código y datos válidos, registros de propósito general con valores específicos, etc.
(`http://www.gnu.org/software/grub/manual/multiboot/multiboot.html`)

Esta revisión de grub no permite cargar ejecutables compilados en 64 bits de manera sencilla, por este motivo decidimos que lo mejor era utilizar a una herramienta provista por grub, llamada carga de módulos, que nos permitió cargar distintas partes no críticas del sistema por encima del primer mega de memoria y tener en un mapa de memoria provisto por grub, las posiciones exactas en la RAM de dichos módulos.

Junto con la carga de módulos y el booteo del BSP, se prepara el entorno para el booteo en etapas de los AP.

- **Niveles de booteo del BSP y preparación del entorno de inicio de los AP:**

1. Grub inicializa la máquina a un estado conocido y otorga el control al loader de nivel 1 del BSP pasándole por parámetros estructuras de grub con información del sistema.
   a) Se realizan validaciones requeridas por la especificación multiboot, verificación de firmas, etc.
   b) Se obtienen de la metadata provista por grub las posiciones de memoria donde estan cargados los módulos, ellos son:

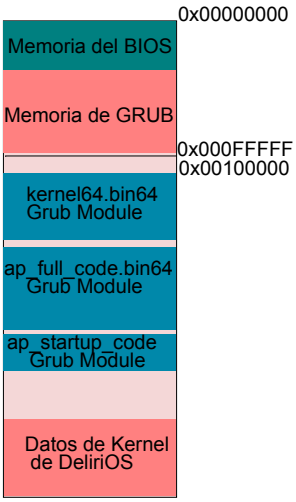
   kernel64.bin64:
   Módulo compilado en formato binario plano de 64 bits que contiene el segundo nivel de booteo del BSP.

   ap_full_code.bin64:
   Módulo compilado en formato binario plano de 64 bits que contiene el código del segundo nivel de booteo de los Application Processors.

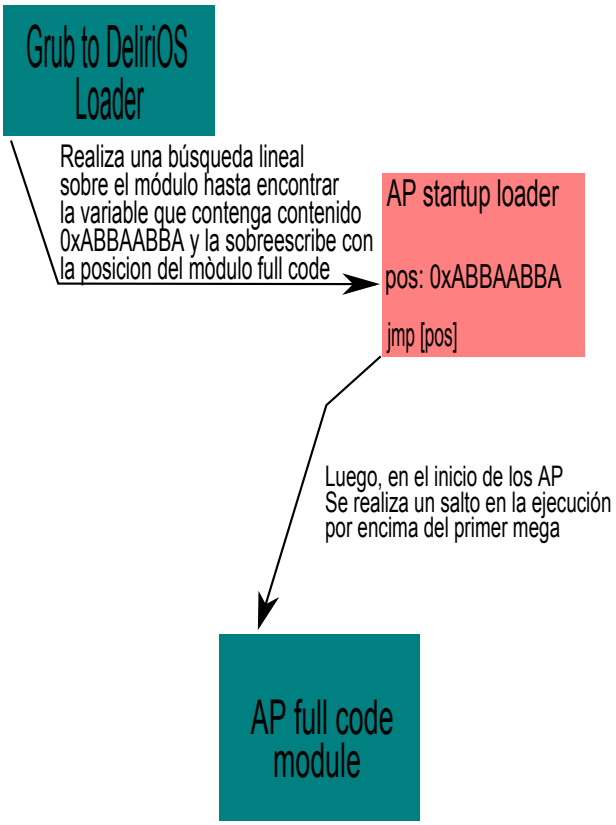
   ap_startup_code:
   Módulo compilado en formato binario de 32 bits que contiene el código de inicio del AP en modo real y el primer stage de booteo a modo protegido.

   c) Los Application Processors inician en modo real, es por esto que deben comenzar su ejecución por debajo del primer mega de memoria principal. Por este motivo se copia el módulo ap_startup_code a una dirección arbitraria alineada a página 0x2000.
   d) Como los AP comienzan su ejecución por debajo del primer mega, esto hace posible la superposicion de código con grub y otras estructuras del kernel, para evitarlo, minimizamos el tamaño del startup de modo real del ap, haciendo lo antes posible un salto a un loader de nivel 2 por encima del mega, este loader de nivel 2 es ap_full_code.bin64.
   e) Al tener el Application Processor que hacer un salto entre los dos niveles de booteo, se le inyecta al primer módulo la direccion donde esta cargado el segundo nivel de booteo.
   f) Finalmente, se realiza un salto en la ejecución a donde comienza el modulo kernel64.bin64 donde el BSP finaliza la inicializacion del contexto hasta modo largo de 64 bits y enciende a los demás núcleos del sistema.

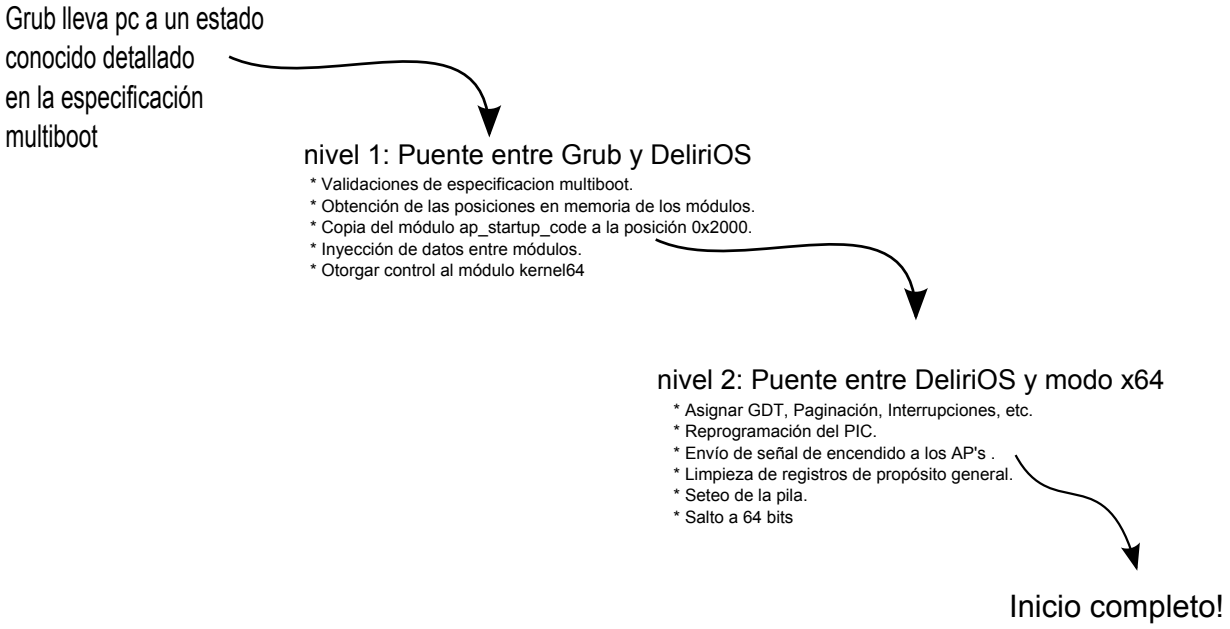
2.2.1. Booteo e integración con grub: Mapa de memoria: Memoria baja y modulos en memoria alta



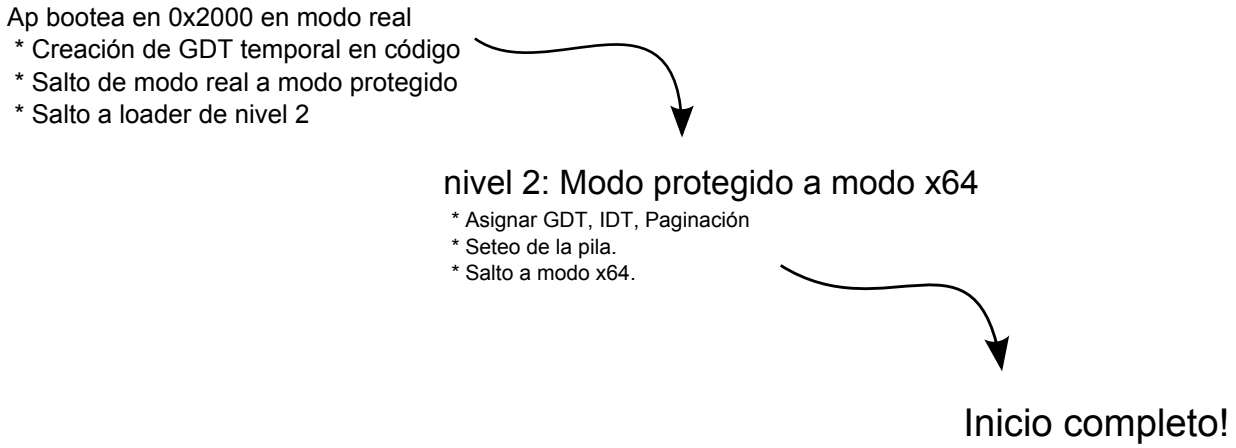
2.2.2. Booteo e integración con grub: Diagrama de interacción entre los niveles de Booteo e inyección de datos



2.2.3. Niveles de booteo del BSP



2.2.4. Niveles de booteo del AP



2.3. Inicialización del Bootstrap Processor: Pasaje desde modo protegido hacia modo legacy x64

2.3.1. Modo Legacy x64: Modelo de segmentación

La especificación multiboot nos asegura que estamos en modo protegido, pero no tenemos la certeza de tener asignada una GDT válida, es por esto que asignamos una GDT con 3 descriptores de nivel 0, una para datos y otras dos de código, esta diferenciación de descriptores de código es necesaria para realizar los jump far para pasar de modo real hacia modo protegido y de modo protegido-compatibilidad x64 hacia modo largo x64.

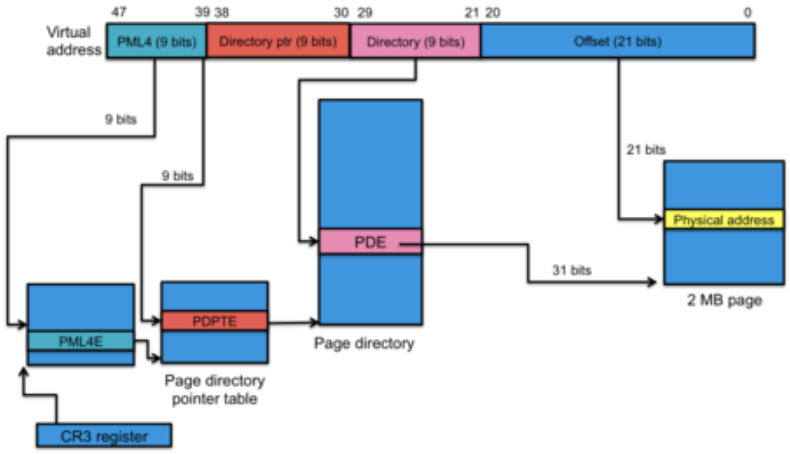
Indice	Descriptor
0	Descriptor nulo
1	Código nivel 0 para 32 bits
2	Código nivel 0 para 64 bits
3	Datos nivel 0 para 32 y 64 bits

2.3.2. Modo Legacy x64: Modelo de paginación de los primeros 4gb

Se utilizó un modelo de paginación en identity mapping donde se cubren los primeros 64 GB de memoria. El modo de paginación elegido fue IA32e en 3 niveles con páginas de 2 megas, es importante remarcar que como para pasar a modo largo de 64 bits es obligatorio tener paginación activa, el mapeo de la memoria virtual fue realizado en 2 etapas, en la primera se mapearon unicamente los primeros 4gb pues desde modo protegido puedo direccionar como máximo hasta 4gb y luego desde modo largo, se completa el esquema de paginacion a 64gb agregando las entradas necesarias a las estructuras.

Esquema de paginación IA32-e:

- **PML4:** 512 entries disponibles de 8 bytes de ancho cada una. Solo fue necesario instanciar la primera entrada de la tabla.
- **PDPT:** 512 entries disponibles de 8 bytes de ancho cada una. Solo fue necesario instanciar las primeras 64 entradas de esta tabla.
- **PDT:** 32768 entries disponibles de 8 bytes de ancho cada una. Se instancian en modo protegido 2048 entradas para cubrir los primeros 4gb y luego desde modo largo se completan las 30720 entradas restantes completando 64 gb.



Luego de establecer estas estructuras, realizamos una comprobación vía *cpuid* para saber si está disponible modo x64, de verificarse esta comprobación, encendemos los bits del procesador correspondientes para habilitar dicho modo.

2.4. Inicialización del Bootstrap processor: Pasaje a modo largo x64 nativo

Para pasar de modo compatibilidad a modo nativo de 64 bits, es necesario realizar un salto largo en la ejecucion a un descriptor de la GDT de código de 64 bits.

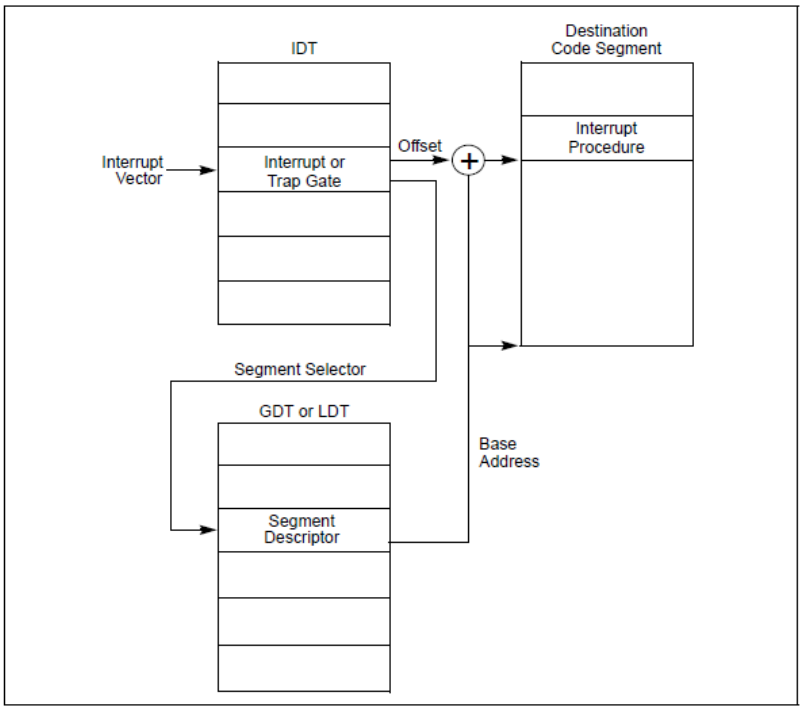
Luego de realizar el salto al segmento de código de x64 de la GDT establecemos un contexto seguro con los registros en cero, seteamos los selectores correspondientes de la GDT y establecemos los punteros a una pila asignada al BSP.

2.4.1. Modo Largo x64: Extensión de paginación a 64 gb

En este punto ya podemos direccionar arriba de los 4gb, entonces completamos las entradas en las estructuras de paginación para completar el mapeo hasta 64gb.

2.4.2. Modo Largo x64: Inicialización del PIC - Captura de excepciones e interrupciones

Enviamos señales al PIC para reprogramarlo de forma tal en la que atienda las interrupciones enmascarables. Luego, asignamos una IDT que captura todas las excepciones e interrupciones y de ser necesario, realiza las acciones correspondientes con su ISR asociada. Particularmente las excepciones son capturadas y mostradas en pantalla y se utiliza la interrupcion de reloj para sincronizacion y esperas, las demás interrupciones son ignoradas.



Interrupt Procedure Call

2.5. Modo Largo x64: Mapa de memoria del kernel

2.5.1. Estructuras de paginación en memoria

Estructura	Posición en memoria
PML4T	0x0000000000740000
PDPT	0x0000000000841000
PDT	0x0000000000942000

2.5.2. Pilas asignadas a cada procesador

Núcleo	Posición de la pila
BSP	0x0000000000400000
AP1	0x0000000000500000
AP2	0x0000000000600000
AP3	0x0000000000700000
...	...
AP15	0x0000000001600000

2.5.3. Variables globales y constantes del sistema

Descripción	Posición en memoria
static_variable_area	0x0000000000200000
start_address	0x0000000000200000
start_merge_address	0x0000000000200001
sleep_address	0x0000000000200002
start_copy_address	0x0000000000200003
number_of_cores_address	0x0000000000200004
seed_address	0x0000000000200006
array_len_address	0x0000000000200010
done_address	0x0000000000200020
finish_copy_address	0x0000000000200030
TEN_MEGA	0x0000000000a00000
MAX_PROCESSOR	8
temp_address	0x0000000001e00000
array_start_address	temp_address + MAX_PROCESSOR * TEN_MEGA

2.6. Multicore: encendido de los AP's

Una vez inicializado el bsp, se procede a inicializar el resto de los procesadores del sistema.
Para lograr esto, primero es necesario que encontrar una estructura de datos llamada MP Floating Pointer Structure, que contiene la información sobre: los demas procesadores; apic (advanced programable interrupt controller); y el I/O apic (análogo al apic, pero se encarga además de rutear interrupciones de input/output a los lapic's).

2.6.1. Búsqueda de la estructura MP Floating Pointer

Esta estructura puede estar en diferentes lugares de memoria, inicialmente se debe realizar la búsqueda dentro del primer kilobyte de la ebda (extended bios data area), de no encontrarse allí, se procede a buscar entre los 639K-640K de memoria. Para encontrar la tabla se busca dentro de esas areas de memoria la firma de la MP Floating Pointer Structure, la cual es "_MP_".

2.6.2. Comprobación de checksum de la estructura

Una vez encontrada una estructura con esa firma, es necesario realizar una comprobacion de checksum para verificar que la misma sea válida. De no ser válida la estructura encontrada, se continúa buscando dentro de las areas de memoria previamente mencionadas, al agotarse las áreas mencionadas sin éxito en la buqueda, se concluye que el sistema no soporta multicore.

2.6.3. Habilitacion de ICMR y Local APIC

El sistema luego es configurado usando la MP Configuration Table, extraída de la MP Floating Pointer Structure, que contiene en diferentes entradas la información acerca de los diferentes dispositivos utilizados en multicore. Una vez parseadas estas entradas, si la maquina tiene ICMR, se habilita ese registro, y luego se procede a inicializar el apic local del bsp, seteando el vector de interrupciones espurias (interrupciones por ruido de hardware, etc) y habilitando el bit de enable dentro de los registros del local apic. (Registros del local apic especificados en Table 10-1 Local APIC Register Address Map, manual 3 de intel capitulo 10).

2.6.4. Inicializacion de los AP's

Una vez finalizada la inicializacion del local apic, se debe pedir a la BIOS que setee el warm reset vector a la dirección donde esta localizado el inicio de modo real de los aps($0x2000$). Luego de este paso, se procede a encender los aps usando la información obtenida por la MP Configuration Table.

El proceso de encendido los aps consiste en preparar 3 estructuras de interrupt command register (registro del apic usado para ipis) para luego enviar las ipis especificas de inicio. El primer icr se setea con el delivery mode de INIT, que sea una interrupción por nivel y con la dirección del ap a iniciar. El segundo icr se setea con el delivery mode de INIT_DASSERT, que sea una interrupción por flanco, y que sea enviado a todos los procesadores. El tercer icr se setea con el delivery mode STARTUP, la dirección fisica de la página de inicio de ejecución del ap shifteado 12 a derecha, y la dirección del ap.

Luego de tener listos los icr, se procede a mandar las ipis de INIT e INIT_DASSERT, luego se espera unos 10 milisegundos, verificando previamente que las ipis se hayan enviado correctamente. Luego de la espera, se procede a enviar la ipi de STARTUP, se espera 10 ms, se verifica que se haya enviado correctamente, se la vuelve a enviar, se realiza otra espera de 10 ms, y se vuelve a verificar.

Una vez terminado este proceso, se puede asumir que el ap fue encendido, y se continúa el encendido el resto de los aps encontrados en la MP Configuration Table.

Nota 1: El proceso de inicializacion de los aps se encuentra especificado en detalle en el manual de intel volumen 3, capitulo 8, subsección 4, MULTIPLE-PROCESSOR (MP) INITIALIZATION.

Nota 2: Dado que se realizan experimentos con un máximo de 2 cores, se limita el encendido de los Application Processors a uno, que junto con el Bootstrap Processor son 2 núcleos.

Nota 3: Los identificadores de nucleo no necesariamente son valores numéricos consecutivos.

2.7. Multicore: inicialización de modo real a modo nativo x64 de los AP's

Como vimos en la sección anterior, los Application Processors comienzan su ejecución en una posición otra por debajo del primer mega en modo real, nosotros necesitamos hacer saltar la ejecucion a una posicion conocida por encima del mega que no se solape con estructuras del kernel ni otros modulos, la solucion que propusimos es un booteo por etapas.

2.7.1. Booteo por niveles: Modo real a modo protegido y modo protegido en memoria alta

En este primer nivel el núcleo se encuentra en modo real, se inicializa una GDT básica en el mismo código y se salta a modo protegido, esto es necesario para poder direccionar posiciones de memoria por encima del primer mega.

Recordando secciones anteriores, cuando el BSP iniciaba el primer nivel de booteo preparaba el contexto de los APS para inicializar en niveles, en este proceso se inyecta en el código del primer nivel de booteo del AP la posicion de memoria donde esta el segundo nivel de booteo por encima del mega.

Luego resta unicamente realizar el salto al segundo bootloader con un jump para continuar el booteo del AP, de manera similar al BSP pasamos luego a modo nativo de 64 bits.
Notemos que por ejemplo la línea A20 ya esta habilitada, y algunas estructuras del kernel que fueron inicializadas por el BSP son comúnes a todos los núcleos, como por ejemplo la GDT y la estructura jerárquica de paginación, que son directamente asignadas a los registros del núcleo correspondiente con punteros a ellas.
Para el manejo de interrupciones y excepciones, se inicializa una IDT para los Application Processors, además es necesario habilitar los local-apic de cada AP, en caso contrario, no podríamos utilizar interrupciones inter-procesador(IPIS).

Como el número de application processors puede ser variable a priori, cuando un núcleo comienza su ejecución, es obtenido su código de identificación dentro del procesador y luego se obtiene una posición de memoria única para cada núcleo con el fin de poder asignar los punteros de la pila RSP y RBP, de ser necesaria la reserva de memoria para alguna otra estructura única por núcleo se puede utilizar este recurso.

3. Desarrollo: Algoritmos implementados

En esta sección, con el contexto del sistema inicializado completamente, describiremos los algoritmos paralelizados implementados y sus resultados.

3.1. Sorting de arreglos

3.1.1. Conjuntos de numeros pseudoaleatorios utilizados para los experimentos

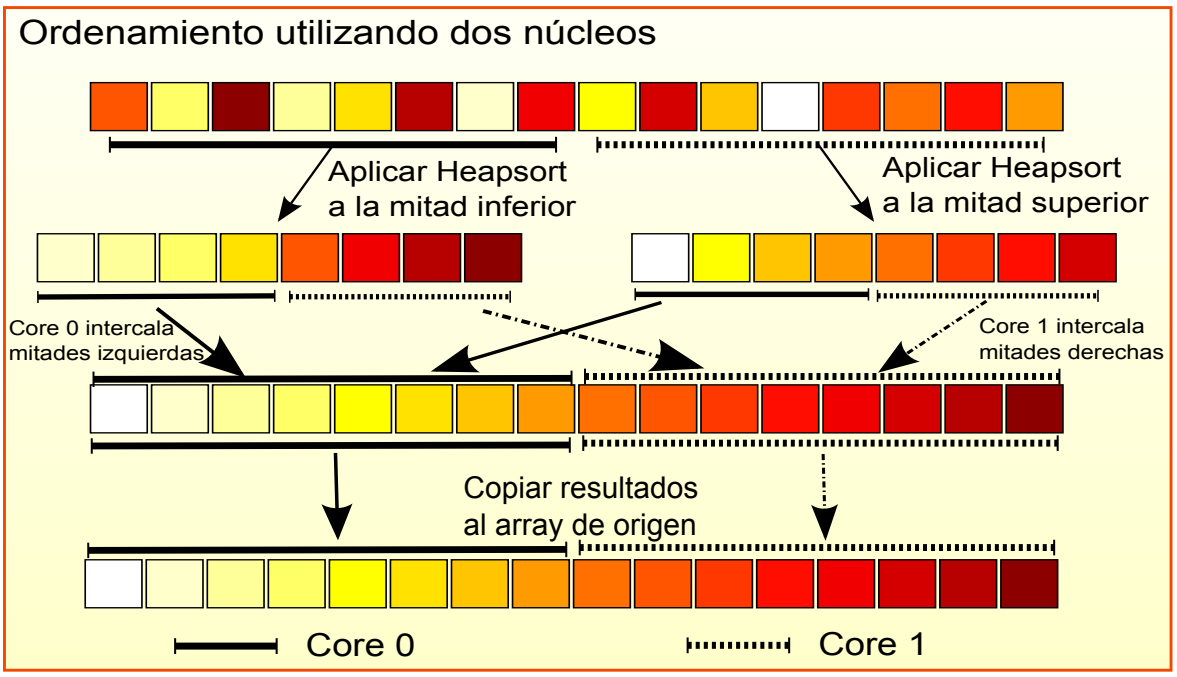
Para este experimento se realizaron pruebas con diferentes longuitudes de arreglo, estas siempre siendo una potencia de 2.
Las muestras de datos se generaron utilizando un generador de numeros pseudo-aleatorios congruencial lineal con la misma semilla inicial. Asimismo luego de todos los ordenamientos de distinto tamaño de entrada y modo de procesamiento, se realiza una verificación de correctitud del algoritmo de ordenamiento para detectar posibles errores en el procesamiento o sincronización de los núcleos.

3.1.2. Implementación con un unico núcleo

En este experimento se realiza un heap-sort sobre todos los arreglos de distinto tamaño.

3.1.3. Implementación con dos cores: Paralelización del algoritmo

Se optó en este caso por un algoritmo propio que combinara varios ordenamientos conocidos con el objetivo de hacer mas facil la paralelización del experimento. En particular lo que se realiza es partir el arreglo en 2 mitades, donde cada core realiza heapsort sobre su mitad asignada y luego una intercalación de los resultados con un algoritmo tambien paralelo en el cual un core realiza el merge de los maximos y el otro el merge de los minimos, luego resta copiar los resultados y concluimos.



3.1.4. Implementación con dos cores: Sincronización con espera activa

Para abordar el problema de la sincronización la primera implementacion fue basada en espera activa, en la que basicamente existen flags en los cuales los distintos núcleos indican su estado actual entre sí actualizando los diferentes flags, igualmente esto no es simétrico, hay 2 roles definidos, maestro y esclavo, el primero da las órdenes y espera la señal de finalización de las operaciones asignadas al núcleo esclavo. El algoritmo funciona de la siguiente manera, siendo el bsp el master y el ap el slave:

1. Cuando el master empieza a ejecutar el algoritmo, el slave esta chequeando todo el tiempo una variable para ver si puede iniciar o no el trabajo.

2. El master setea el flag de inicio del sort y empieza a ordenar su mitad del arreglo, mientras que el slave sale del loop de verificacion y empieza a ordenar su mitad del arreglo.

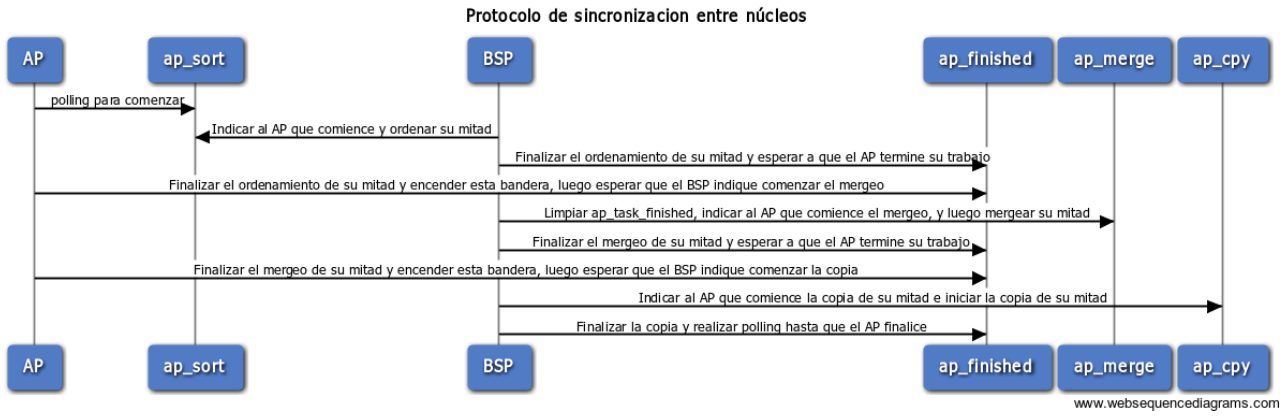
3. Una vez que el master termina, se queda esperando a que el slave setee el flag de que termino la etapa en la que estaba. El slave cuando termina su etapa setea el flag que avisa que termino y se pone a chequear en un loop un flag para que le autorice a pasar a la siguiente etapa.

4. Cuando el master verifica que el slave termino, resetea los flags utilizados hasta ahora, y setea el flag de inicio de la etapa de merge, luego de lo cual se pone a realizar merge de su mitad. El slave, por su parte, una vez que verifica que puede pasar a la siguiente etapa, se pone hacer merge con su mitad, seteando un flag para avisar que termino.

5. El master, luego de verificar que el slave termino, setea el flag para iniciar la última etapa del algoritmo, que es la de copiar los resultados de los diferentes merge al arreglo original. Cada core por su lado copia los resultados de sus respectivos merge al array original. El slave cuando termina le avisa al master y se queda loopeando esperando la señal para arrancar el algoritmo de 0.

6. Una vez que el master verifico que el slave termino, resetea todos los flags utilizados hasta ahora, y sale de la función.

7. Cuando se terminan todas las pruebas, se setea una variable global para avisarle al slave que puede salir de la función y pasar a modo de bajo consumo.



3.1.5. Implementación con dos cores: Sincronización con inter-processor interrupts

En esta implementación, el slave se encuentra en modo de bajo consumo, y lo que hace el master es mandarle interrupciones específicas de acuerdo a la etapa en la que esta del algoritmo. En total son 3 interrupciones diferentes, las cuales hacen que el slave ordene, mergee y copie respectivamente.

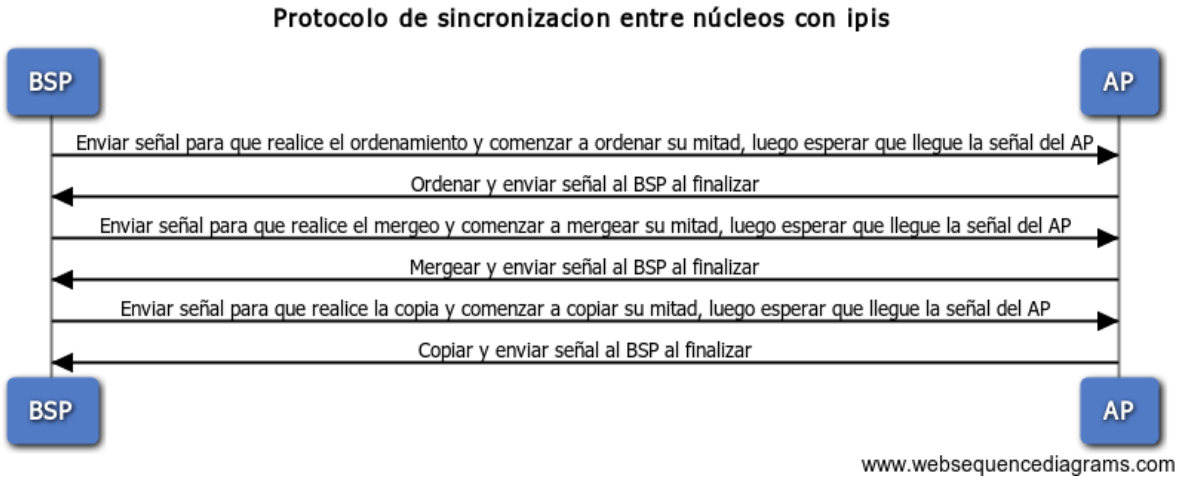
En este caso el protocolo de sincronización para las tres etapas es:

1. El slave espera que el master envie una interrupcion para comenzar la etapa correspondiente.

2. El master realiza su trabajo asignado en dicha etapa, y al finalizar setea su registro RAX en cero y setea una variable bandera en 1 dando aviso de que finalizó y luego realiza polling al registro RAX mientras el valor sea cero.

3. El slave espera que el master finalice leyendo la variable bandera, esperando que su valor sea 1. Cuando esto ocurre, es enviada una interrupcion al master, y pasa a modo bajo consumo, es decir, se haltea.

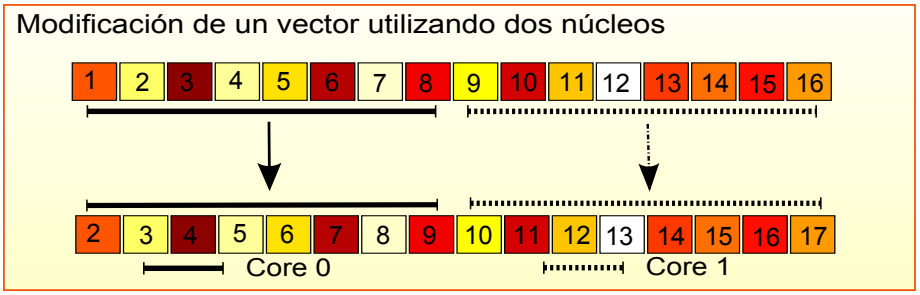
4. El master recibe la interrupcion y en la ISR setea su registro RAX en 1, haciendo que salga del ciclo del item 2 para pasar a la siguiente etapa del algoritmo o finalizar segun corresponda.


Protocolo de sincronizacion entre núcleos con ipis
BSP
AP
Enviar señal para que realice el ordenamiento y comenzar a ordenar su mitad, luego esperar que llegue la señal del AP
Ordenar y enviar señal al BSP al finalizar
Enviar señal para que realice el mergeo y comenzar a mergear su mitad, luego esperar que llegue la señal del AP
Mergear y enviar señal al BSP al finalizar
Enviar señal para que realice la copia y comenzar a copiar su mitad, luego esperar que llegue la señal del AP
Copiar y enviar señal al BSP al finalizar
BSP
AP
www.websequencediagrams.com

3.2. Modificación de elementos de un arreglo

3.2.1. Saturación del canal de memoria

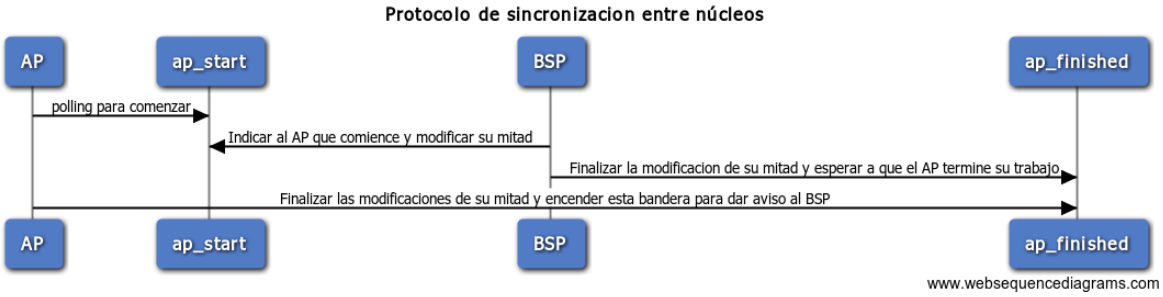
Con el objetivo de analizar el impacto de los accesos a memoria de multiples núcleos realizamos un experimento en el cual cada uno de los dos núcleos accede a una mitad del arreglo concurrentemente modificando los valores, realizando un incremento en uno del valor correspondiente.



3.2.2. Sincronizacion entre núcleos

En este algoritmo se realizo una sincronización sencilla de espera activa con roles master y slave, el protocolo es:

1. El master setea una bandera indicando al slave que comience su trabajo.
2. El slave, que estaba realizando polling sobre la bandera del punto anterior comienza su trabajo, al finalizar, escribe en otra bandera indicando que finalizo.
3. El master, luego de finalizar su trabajo, hace polling sobre una bandera en espera de la finalización del slave. Al salir de esta espera, concluye el algoritmo.



3.3. Fast Fourier Transform

4. Resultados

En esta sección se plasmarán los resultados obtenidos en las pruebas de las secciones anteriores, asimismo se darán detalles tecnicos de los equipos donde fueron realizados los experimentos.

4.1. Lectura e interpretación de resultados por pantalla

El sistema luego de realizar la inicialización, comienza a arrojar por pantalla, los resultados(medidos en ticks) de los distintos algoritmos (separados por columna) aplicados sobre distintos tamaños de datos(separados por filas). Sobre estos datos obtenidos en una variedad de máquinas reales, se realizaron documentos, análisis y gráficos y serán plasmados debajo.

4.2. Resultados: Forma de medición

Las mediciones se realizaron utilizando los registros del procesador que contienen la cantidad de ticks transcurridos. Los puntos del código elegidos para comenzar y finalizar la medición son antes y despues de llamar a los métodos de test, por ejemplo:

- Comenzar medicion
- Ejecutar sort_single_core
- Finalizar medicion

Notemos que estas formas de medición no solo analizan el rendimiento del algoritmo propiamente dicho, sino además los tiempos de sincronización requeridos para cada implementación. (ver gráficos de los protocolos de sincronización), esto nos permitirá ver como en algunos casos, donde el tamaño de la entrada de datos es relativamente pequeña, la mejora obtenida en el multiprocesamiento es licuada por los tiempos extras requeridos para la sincronización, dándonos además un cross-over, el cual nos indica a partir de que tamaño de entrada comienza a ser mas ventajoso utilizar multiprocesamiento, permitiendonos si lo desearamos, realizar algoritmos mas inteligentes que dependiendo de los datos de entrada utilicen la implementación que tenga el mejor rendimiento.

4.3. Resultados: Arquitectura de las máquinas utilizadas

Se realizaron pruebas sobre los siguientes equipos:

1. Intel® Pentium® Processor T4200
   - CPU Clock: 2000Mhz
   - Front Side Bus: 800Mhz
   - Cores Number: 2
   - L1 Caché: 2 x 32KB instruction, 2 x 32KB data
   - L2 Caché: Shared 1MB
   - Litografía: 45nm
2. Intel® Xeon® Processor E5345
   - CPU Clock: 2333Mhz
   - Front Side Bus: 1333Mhz
   - Cores Number: 4
   - L1 Caché: 4 x 32KB instruction, 4 x 32KB data
   - L2 Caché: 2 x 4MB shared
   - Litografía: 65nm
3. Intel® Pentium® Processor G2030
   - CPU Clock: 3000Mhz
   - Direct Media Interface: 5 GT/s

- Cores Number: 2
- L1 Caché: 2 x 32KB instruction, 2 x 32KB data
- L2 Caché: 2 x 256K
- L3 Caché: 3 MB (Intel Smart Cache)
- Litografía: 22nm

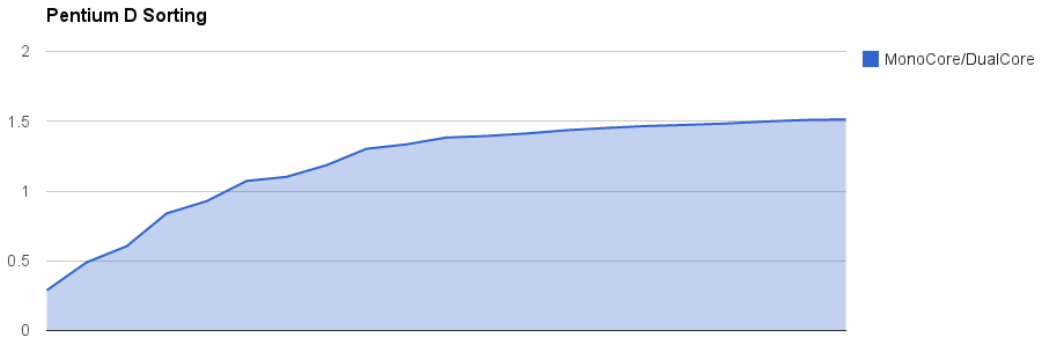
4. Intel® Core™ i7-920 Processor
   - CPU Clock: 2666Mhz
   - Bus Speed: 4.8 GT/s QPI (2400 MHz)
   - Cores Number: 4
   - L1 Caché: 4 x 32KB instruction, 4 x 32KB data
   - L2 Caché: 4 x 256K
   - L3 Caché: 8 MB (Intel Smart Cache) Shared
   - Litografía: 45nm
5. Intel® Core™ i5-2500K Processor
   - CPU Clock: 3300Mhz 3700Mhz turbo
   - Direct Media Interface: 5 GT/s
   - Cores Number: 4
   - L1 Caché: 4 x 32KB instruction, 4 x 32KB data
   - L2 Caché: 4 x 256K
   - L3 Caché: 6 MB (Intel Smart Cache) Shared
   - Cache latency:
     - 4 (L1 cache)
     - 11 (L2 cache)
     - 25 (L3 cache)
   - Litografía: 32nm
6. Intel® Core™2 Quad Processor Q6600
   - CPU Clock: 2400Mhz
   - FSB: 1066Mhz
   - Cores Number: 4
   - L1 Caché: 4 x 32 KB 8-vias asociativa por conjuntos instrucciones, 4 x 32 KB 8-vias asociativa por conjuntos datos
   - L2 Caché: 2 x 4 MB 16-vias asociativas (cada L2 cache es compartida por los 2 cores)
   - Litografía: 65nm

Nota: Todos los equipos tienen al menos 2 núcleos y su set de instrucciones es de 64 bits.

4.4. Análisis de resultados

4.4.1. Intel® Pentium® Processor T4200 - Sorting

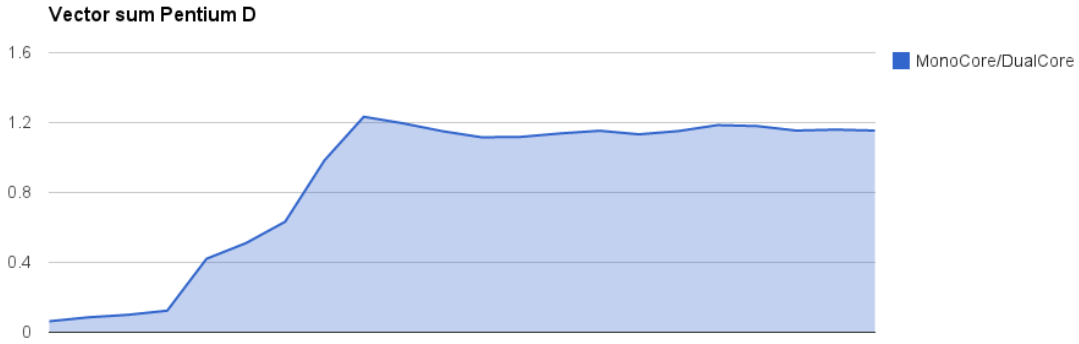
Elements	MonoCore Ticks	DualCore Ticks	MonoCore/DualCore Ratio
2	13880	48100	0.288
4	24270	49570	0.489
8	55400	91690	0.604
16	141500	168570	0.839
32	327920	353760	0.926
64	796010	742520	1.072
128	1849630	1679460	1.101
256	4320740	3646330	1.184
512	9934510	7628720	1.302
1024	22071820	16545320	1.334
2048	48763360	35257470	1.383
4096	106738920	76572280	1.393
8192	231258040	163781130	1.411
16384	498651230	347409350	1.435
32768	1068017890	735754710	1.451
65536	2273209280	1551108110	1.465
131072	4829745290	3277073490	1.473
262144	10221826090	6889769040	1.483
524288	21634980190	14443509820	1.497
1048576	45669905810	30263300000	1.509
2097152	96048564890	63479415710	1.513



En esta configuración podemos observar un cross-over en 64 elementos a partir de donde comienza a mejorar la performance utilizando dos cores. Para la mayor cantidad de elementos obtenemos una mejora aproximada del 50 %.

4.4.2. Intel® Pentium® Processor T4200 - Vector modification

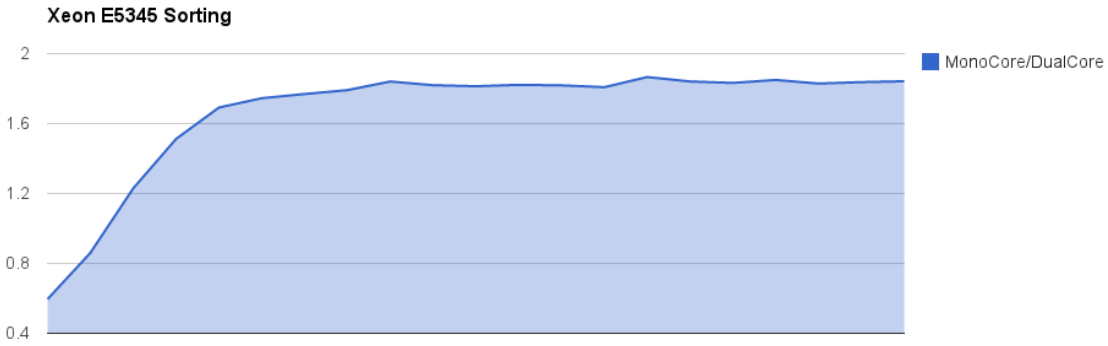
Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	6528	103472	0.063
4	8128	94672	0.085
8	10656	106480	0.100
16	14032	113328	0.123
32	32688	77680	0.420
64	60736	119088	0.510
128	118432	187120	0.632
256	237984	241904	0.983
512	471664	382400	1.233
1024	958848	801840	1.195
2048	1862816	1618736	1.150
4096	3616320	3241024	1.115
8192	6855104	6129552	1.118
16384	13412768	11785552	1.138
32768	26840704	23272048	1.153
65536	53739680	47434224	1.132
131072	107571760	93427920	1.151
262144	216317472	182467152	1.185
524288	430617120	364908528	1.180
1048576	860450928	745400768	1.154
2097152	1730369088	1491881600	1.159
4194304	3462956448	3000064512	1.154



Podemos observar como, incluso utilizando dos núcleos, la saturación del canal de memoria licua la performance mejorada del multiprocesamiento.

4.4.3. Intel® Xeon® Processor E5345 - Sorting

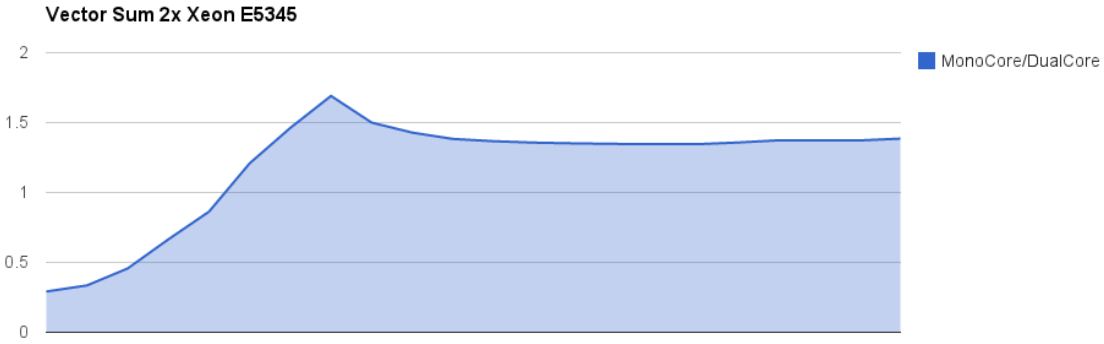
Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	20657	34692	0.595
4	40565	47131	0.860
8	90559	73661	1.229
16	208635	137970	1.512
32	485933	287224	1.691
64	1096592	628369	1.745
128	2490712	1407854	1.769
256	5545589	3095624	1.791
512	12350142	6709311	1.840
1024	26958904	14815346	1.819
2048	58405928	32200161	1.813
4096	126362572	69358674	1.821
8192	271480069	149265949	1.818
16384	580168379	320822040	1.808
32768	1265098373	677863235	1.866
65536	2663953124	1447008150	1.841
131072	5597340728	3054159843	1.832
262144	11904206349	6435623397	1.849
524288	24800933899	13558975689	1.829
1048576	52289342196	28461202042	1.837
2097152	109516365196	59447473463	1.842



Dadas las mejores prestaciónes de caché y arquitectura de esta configuración, vemos que se obtiene un 84 % de mejora al realizar el multiprocesamiento al ordenar muestras de 2097152 elementos. El cross-over donde conviene utilizar multicore es muy bajo, desde 8 elementos ya es mas rápido el ordenamiento dual core.

4.4.4. Intel® Xeon® Processor E5345 - Vector modification

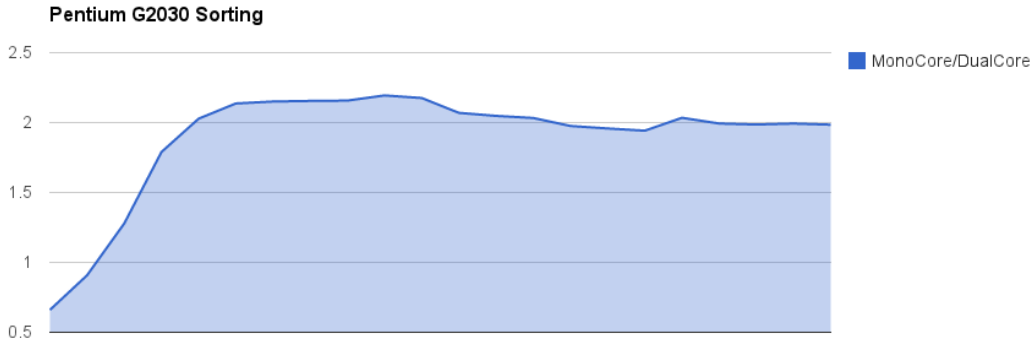
Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	4900	16800	0.291
4	5831	17388	0.335
8	8211	17976	0.456
16	12985	19537	0.664
32	23198	26880	0.863
64	42511	35168	1.208
128	81543	55769	1.462
256	159663	94430	1.690
512	316463	210994	1.499
1024	630154	441224	1.428
2048	1256241	908292	1.383
4096	2509745	1836184	1.366
8192	5014814	3696385	1.356
16384	10030118	7422128	1.351
32768	20053355	14880299	1.347
65536	40102146	29786883	1.346
131072	80194310	59598644	1.345
262144	161790321	119193046	1.357
524288	327379871	238416367	1.373
1048576	656828004	478496431	1.372
2097152	1309034153	953788577	1.372
4194304	2619897161	1891070293	1.385



Nuevamente los accesos a memoria afectan la performance del multiprocesamiento.

4.4.5. Intel® Pentium® Processor G2030 - Sorting

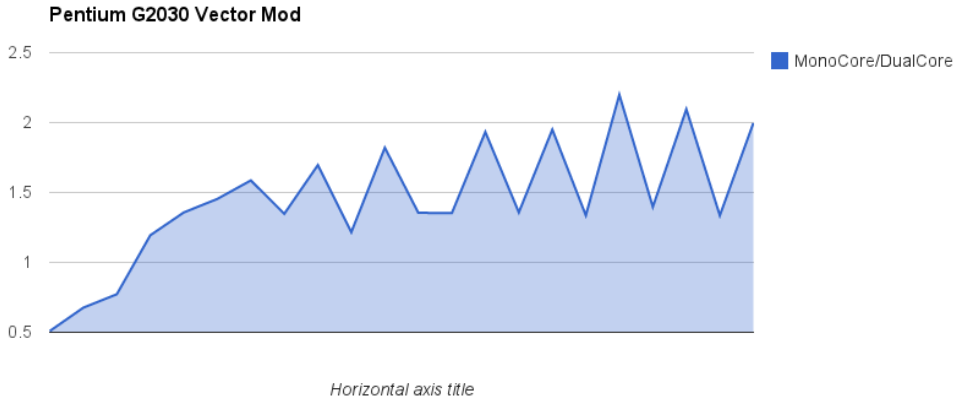
Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	15180	22996	0.660
4	30704	33812	0.908
8	70096	54804	1.279
16	186620	104288	1.789
32	436216	215084	2.028
64	1011620	473460	2.136
128	2311748	1074916	2.150
256	5156888	2393136	2.154
512	11414660	5290992	2.157
1024	25167592	11469944	2.194
2048	54610712	25100072	2.175
4096	118042804	57037548	2.069
8192	252518012	123304440	2.047
16384	536777312	264044108	2.032
32768	1097052680	555262936	1.975
65536	2322030308	1185902828	1.958
131072	4932183084	2539728536	1.942
262144	10447157964	5135427448	2.034
524288	21599239404	10835600988	1.993
1048576	45271465128	22793636612	1.986
2097152	94724393340	47526712420	1.993
4194304	197880816576	99686391488	1.985



Esta arquitectura nueva, nos sorprendió con un crossover de 8 elementos en el cual mejora la performance y a partir de 32 elementos aproximadamente se duplica el rendimiento utilizando multicore!

4.4.6. Intel® Pentium® Processor G2030 - Vector modification

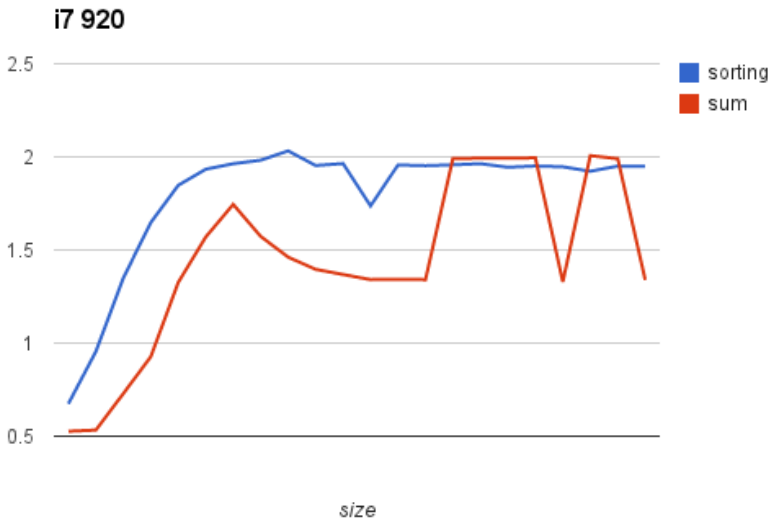
Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	3844	7552	0.509
4	5260	7780	0.676
8	7996	10356	0.772
16	15036	12592	1.194
32	24732	18220	1.357
64	46612	32060	1.453
128	87408	55104	1.586
256	173268	128556	1.347
512	336340	198352	1.695
1024	682580	561132	1.216
2048	1356376	745136	1.820
4096	2959568	2184308	1.354
8192	6134040	4536436	1.352
16384	12473160	6453392	1.932
32768	24960936	18386228	1.357
65536	49965760	25624672	1.949
131072	99945548	74779612	1.336
262144	199857864	90879264	2.199
524288	399539592	285954068	1.397
1048576	799210112	381478298	2.095
2097152	1598536888	1196662984	1.335
4194304	3196868172	1601236716	1.996



A pesar de como afectan los accesos a memoria, en algunos casos vemos que se rompen los esquemas y se obtienen performances del doble versus procesamiento mono-núcleo.

4.4.7. Intel® Core™ i7-920 Processor - Sorting - Vector modification

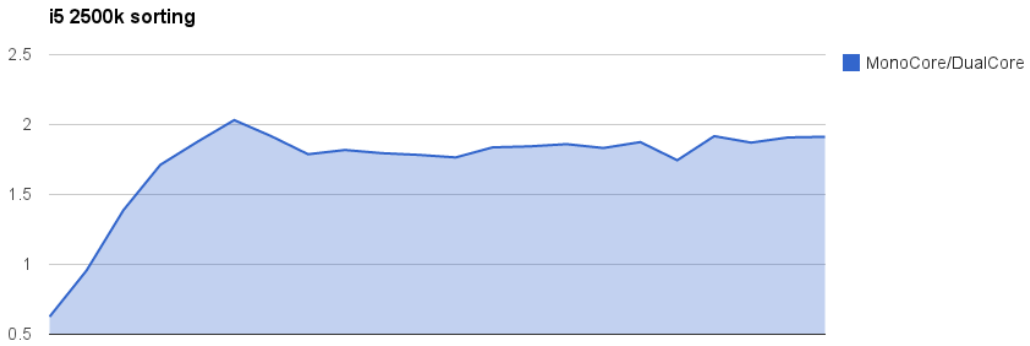
Elements	Sort Mono	Sort Dual	Vector Mono	Vector Dual	Sort Ratio	Vector Mod Ratio
2	17668	26204	4912	9332	0.674	0.526
4	36888	38648	5692	10680	0.954	0.532
8	84260	62436	8524	11696	1.349	0.728
16	192596	116900	15160	16356	1.647	0.926
32	463176	250716	26856	20256	1.847	1.325
64	1062000	549240	50600	32240	1.933	1.569
128	2427412	1236440	97260	55740	1.963	1.744
256	5389400	2719116	192860	122600	1.982	1.573
512	11997744	5903640	381276	260856	2.032	1.461
1024	25525608	13065240	761760	545620	1.953	1.396
2048	55905976	28463616	1519960	1110780	1.964	1.368
4096	120254264	69237000	3031636	2260796	1.736	1.340
8192	259149012	132442380	6051280	4509000	1.956	1.342
16384	554582312	284005796	12106900	9029180	1.952	1.340
32768	1199133428	612601980	24212436	12154760	1.957	1.992
65536	2547819264	1297609520	48417380	24300516	1.963	1.992
131072	5385534272	2769704056	96870560	48611840	1.944	1.992
262144	11395843660	5840527220	193900280	97191900	1.951	1.995
524288	23990990684	12323090580	398251576	299541500	1.946	1.329
1048576	50408173440	26217339456	792920300	395143060	1.922	2.006
2097152	105680965696	54178541700	1578620280	793110860	1.950	1.990
4194304	221064067796	113389632780	3176314036	2371198400	1.949	1.339



Podemos observar aqui que en arquitecturas mas nuevas de intel, se va mejorando la performance promedio obtenida, cada vez mas cercana a duplicar la performance single core, a pesar de esto, los accesos a memoria siguien siendo un problema en algunos casos.

4.4.8. Intel® Core™ i5-2500K Processor - Sorting

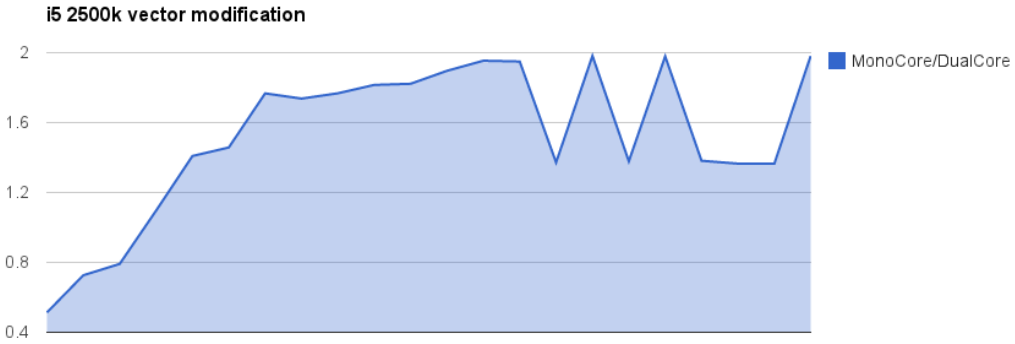
Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	15384	24580	0.625
4	31660	33112	0.956
8	74324	53544	1.388
16	171640	100236	1.712
32	406144	216452	1.876
64	932468	458840	2.032
128	2110364	1100608	1.917
256	4475012	2502924	1.787
512	9904452	5447160	1.818
1024	21732200	12102956	1.795
2048	47346996	26551652	1.783
4096	102885844	58287424	1.765
8192	222250916	120969120	1.837
16384	476876272	258581740	1.844
32768	1019406688	548013260	1.860
65536	2173616224	1186136300	1.832
131072	4633428528	2471420552	1.874
262144	9056811460	5190092924	1.745
524288	20854472344	10876582315	1.917
1048576	43923320368	23475758752	1.871
2097152	94551848940	49553049284	1.908
4194304	198157210936	103593610452	1.912



Dado que este procesador es cercano al i7, podemos observar una performance similar respecto a los ratios de mejora en multiprocesamiento, tanto en los ordenamientos, como en la modicación de vectores en al siguiente sección.

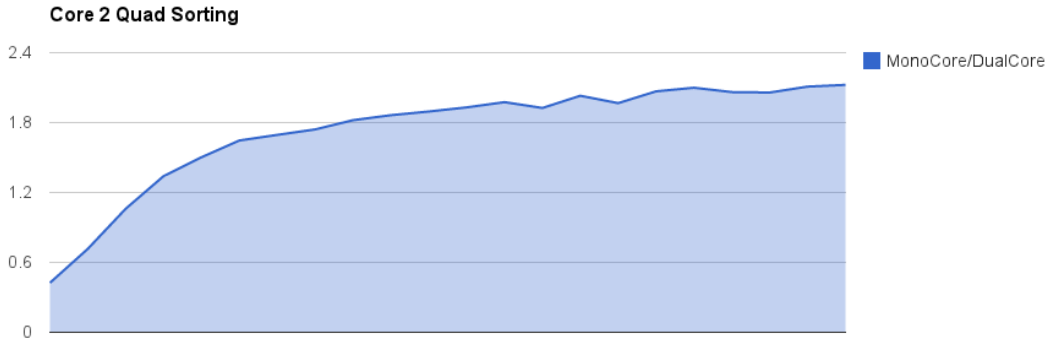
4.4.9. Intel® Core™ i5-2500K Processor - Vector modification

Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	3544	6900	0.513
4	5288	7280	0.726
8	7808	9868	0.791
16	13296	12132	1.095
32	23920	16980	1.408
64	47044	32276	1.457
128	92228	52192	1.767
256	182716	105168	1.737
512	363804	205808	1.767
1024	716228	394484	1.815
2048	1433180	786560	1.822
4096	2870572	1513800	1.896
8192	5694568	2914168	1.954
16384	11378960	5836656	1.949
32768	22432852	16351152	1.371
65536	45314096	22864312	1.981
131072	90536832	65673712	1.378
262144	181141232	91528392	1.979
524288	362184776	262148544	1.381
1048576	724287244	530532932	1.365
2097152	1448749484	1061714556	1.364
4194304	2897714296	1462866292	1.980



4.4.10. Intel® Core™2 Quad Processor Q6600 Sorting

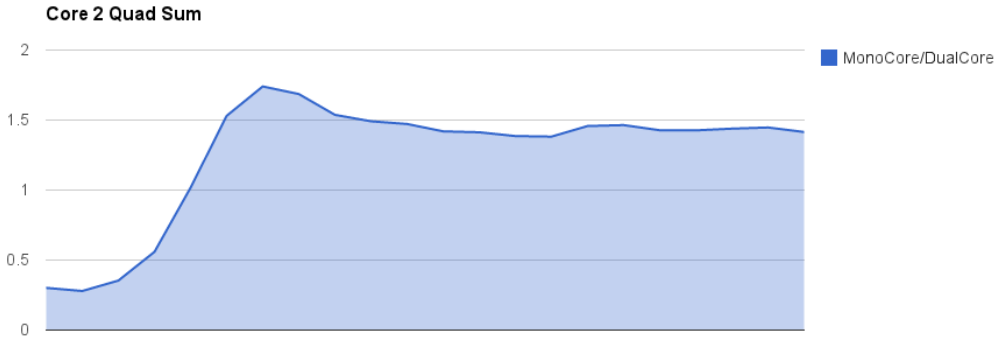
Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	6885	16218	0.424
4	13277	18530	0.716
8	32011	30183	1.060
16	76364	56958	1.340
32	181024	120487	1.502
64	415990	252671	1.646
128	949399	560303	1.694
256	2121048	1218075	1.741
512	4742838	2604884	1.820
1024	10433809	5599078	1.863
2048	22765235	12013475	1.894
4096	49466056	25624066	1.930
8192	107050318	54191427	1.975
16384	230695245	119820352	1.925
32768	492634993	242634115	2.030
65536	1054223219	535990263	1.966
131072	2249860461	1087838729	2.068
262144	4773926137	2274182829	2.099
524288	10087422167	4892424806	2.061
1048576	21270642148	10335385475	2.058
2097152	44667373988	21181628542	2.108
4194304	93526794057	44040620699	2.123



Esta arquitectura, a pesar de ser mas antigua de los ix, nos muestra una buena performance utilizando multicore, de aproximadamente el doble, la mejora respecto de la cantidad de elementos en aumento es casi instantanea, a partir de 8 elementos.

4.4.11. Intel® Core™2 Quad Processor Q6600 Vector modification

Elements	MonoCore Ticks	DualCore Ticks	MonoCore Ticks/DualCore Ticks
2	1717	5695	0.301
4	1938	6945	0.279
8	2456	6945	0.353
16	5491	9826	0.558
32	13745	13515	1.017
64	28016	18309	1.530
128	56151	32266	1.740
256	115379	68434	1.685
512	231107	150314	1.537
1024	459255	307930	1.491
2048	918060	623645	1.472
4096	1835176	1292867	1.419
8192	3682583	2606517	1.412
16384	7390283	5332679	1.385
32768	14743522	10669702	1.381
65536	29898164	20515192	1.457
131072	59790649	40836465	1.464
262144	119581808	83774122	1.427
524288	239160938	167648356	1.426
1048576	478114256	332301831	1.438
2097152	957119720	661399399	1.447
4194304	1913099785	1352226444	1.414

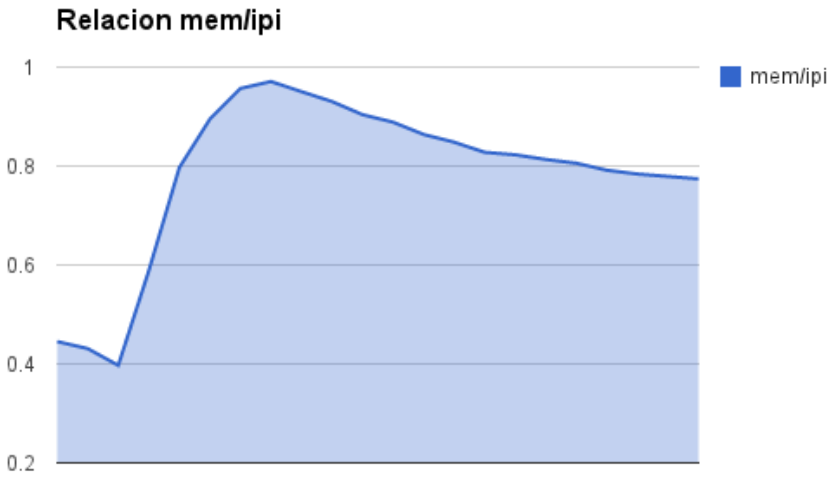


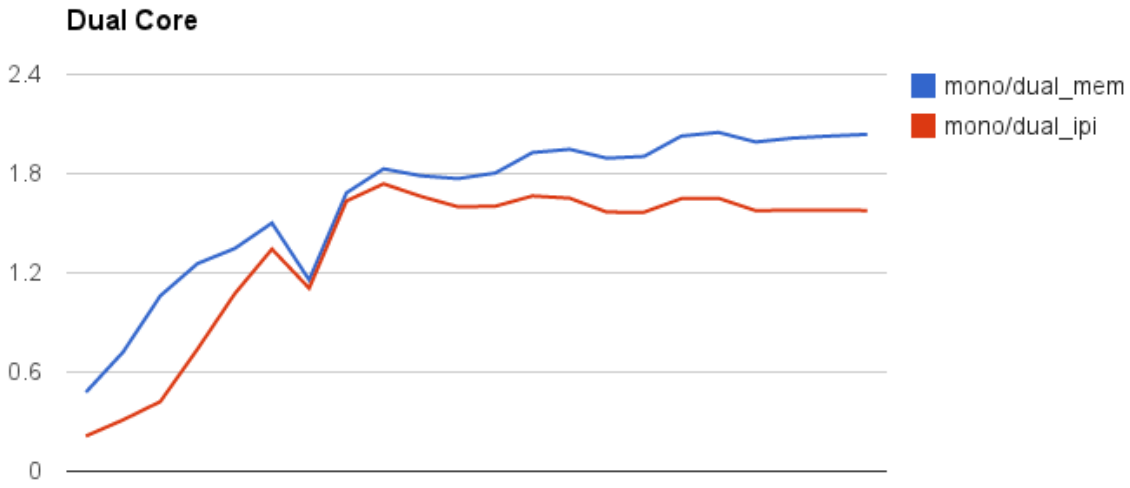
Los accesos a memoria, a los cuales compiten los 2 cores, afectan el rendimiento, aunque utilizar dos cores sigue dando aproximadamente un 40 % de mejoría.

4.4.12. Intel® Pentium® Processor T4200 - Sorting usando Inter-processor interrupts

Para intentar reducir los accesos a memoria, implementamos la sincronización entre núcleos mediante interrupciones entre procesadores en lugar de realizar espera activa a memoria. Los resultados fueron curiosos, dado que dió peor la performance respecto a la sincronización con variables en memoria RAM. Creemos que esto se debe a las velocidades del bus de memoria y el bus de sistema y la memoria caché.

Elements	Sort Single	Sort Dual Mem	Sort Dual Ipi	Ratio M/I	Single/DMem	Single/DIpi
2	26336	55088	123712	0.445	0.478	0.212
4	51136	70896	164496	0.430	0.721	0.310
8	114832	108224	272560	0.397	1.061	0.421
16	268832	213984	363696	0.588	1.256	0.739
32	613520	455376	571488	0.796	1.347	1.073
64	1429152	951840	1063760	0.894	1.501	1.343
128	2524576	2182176	2280608	0.956	1.156	1.106
256	7816560	4642912	4782544	0.970	1.683	1.634
512	17833216	9749392	10258304	0.950	1.829	1.738
1024	37187824	20810160	22367904	0.930	1.787	1.662
2048	79244560	44773584	49540224	0.903	1.769	1.599
4096	171885520	95302656	107242560	0.888	1.803	1.602
8192	369363168	191632704	221836560	0.863	1.927	1.665
16384	781349776	401270912	473079168	0.848	1.947	1.651
32768	1670361776	882252640	1065852080	0.827	1.893	1.567
65536	3568050464	1873549072	2277199056	0.822	1.904	1.566
131072	7538692864	3718695296	4572378768	0.813	2.027	1.648
262144	15870953952	7744017984	9611314224	0.805	2.049	1.651
524288	33266571168	16705027472	21105461872	0.791	1.991	1.576
1048576	69926858336	34701409440	44269003616	0.783	2.015	1.579
2097152	146290056320	72153753040	92620047520	0.779	2.027	1.579
4194304	305621530000	149990136352	193768043584	0.774	2.037	1.577





4.4.13. Intel® Pentium® Processor G2030 - Sorting usando Inter-processor interrupts

TODO: TEST IT!

5. Conclusión Final

Como conclusión de este trabajo, pensamos que la performance obtenida al utilizar dual core a un nivel muy bajo para algoritmos paralelizables no es para nada despreciable, dado que obtuvimos mejoras entre 50 % y 100 % dependiendo de la arquitectura del hardware donde se realizaron los experimentos, esto es una mejor performance que utilizando la infraestructura de threads de lenguajes de alto nivel. Restaría investigar mas aplicaciónes en las cuales se pueda aprovechar este modelo ultraeficiente de cómputo y realizar pruebas con sistemas operativos reales que nos permitan verificar que el overhead creado por estos sea suficientemente apreciable como para decidir utilizar nuestra herramienta.