UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

DESARROLLO DE TÉCNICAS DE PROCESAMIENTO PARALELO A NIVEL DE LENGUAJE ASSEMBLER PARA EL PROCESADOR RISC-V

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELÉCTRICO

JOSÉ MANUEL GIRALT ÁLVAREZ

PROFESOR GUÍA:
FRANCISCO RIVERA SERRANO

MIEMBROS DE LA COMISIÓN:
CESÁR AZURDIA MEZA
PABLO PALACIOS JÁTIVA

SANTIAGO DE CHILE
2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO
POR: JOSÉ MANUEL GIRALT ÁLVAREZ
FECHA: 2021
PROF. GUÍA: FRANCISCO RIVERA SERRANO

DESARROLLO DE TÉCNICAS DE PROCESAMIENTO PARALELO A NIVEL DE LENGUAJE ASSEMBLER PARA EL PROCESADOR RISC-V

El siguiente trabajo muestra la paralelización del algoritmo de ordenamiento *insertion sort* de forma que pueda funcionar utilizando múltiples procesadores RISC-V, con tal de reducir el tiempo de ejecución necesario para su funcionamiento.

Las experiencias realizadas en este trabajo fueron divididas en dos partes, la primera consistiendo en la implementación del paralelismo propiamente tal utilizando la API *OpenMP* y la modificación del programa básico en C original según sea necesario. Mientras que la segunda consistió en reemplazar secciones claves del programa resultante de la experiencia anterior por código *assembler* equivalente. Durante ambas experiencias se utilizó el tiempo de ejecución, medido desde que inicia el proceso de ordenamiento hasta el fin de dicho proceso, como parámetro de estudio.

La implementación de las experiencias se realizó utilizando el simulador *gem5*, que permite simular sistemas con distintos niveles de complejidad y detalle, mientras que su aspecto más importante es su precisión a nivel de ciclo, lo que permite utilizarlo para medir los tiempos de ejecución de manera realista y consistente.

Los resultados obtenidos indicaron que es posible generar un programa que, mediante el paralelismo entre múltiples procesadores, logre reducir el tiempo de ejecución necesario para ordenar un arreglo aleatorio, el que además fuese escalable, significando que el programa es capaz de trabajar con una cantidad creciente de procesadores y reducir el tiempo de ejecución acordemente. Mientras que de la segunda experiencia se desprendió que si bien el lenguaje *assembler* logró reducir el tiempo de ejecución considerablemente, el cambio por si solo no fue mas eficiente que utilizar las opciones del compilador para optimizar el programa.

Quiero dedicar este trabajo a mi familia,
que me apoyó durante todo este trayecto e
hicieron posible llegar a este momento.

Tabla de Contenido

Introducción **1**

1. Marco teórico y estado del arte **4**
1.1. Marco teórico . . . 4
1.1.1. Definiciones básicas . . . 4
1.1.2. El procesador . . . 5
1.1.3. La ISA RISC-V . . . 5
1.1.4. El lenguaje *Assembler* . . . 6
1.1.5. La consistencia de memoria . . . 7
1.1.6. El módulo *Atomic* . . . 7
1.1.7. Métodos de sincronización . . . 9
1.1.8. Secciones paralelizables . . . 10
1.1.9. Complejidad temporal . . . 10
1.1.10. *SpeedUp* . . . 11
1.1.11. *CPU Time* y *Wall Time* . . . 11
1.1.12. Insertion sort . . . 11
1.1.13. Freedom studio . . . 12
1.1.14. *gem5* . . . 12
1.1.15. *OpenMP* . . . 14
1.2. Estado del arte . . . 15
1.2.1. Programa básico en assembler . . . 15
1.2.2. *Insertion sort* adaptativo . . . 16
1.2.3. *Insertion sort* de dos elementos . . . 17

2. Metodología **18**
2.1. Análisis y consideraciones previas . . . 18
2.1.1. Análisis preliminar de *Insertion sort* . . . 18
2.1.2. Formalización del problema . . . 18
2.1.3. Resultados esperados . . . 19
2.2. Metodología de trabajo . . . 19
2.2.1. Parámetros de trabajo . . . 19
2.2.2. Estrategia de medición . . . 20
2.2.3. Procedimiento propuesto . . . 21
2.2.4. Ejemplo de aplicación . . . 23
2.2.5. Implementación del programa . . . 24

3. Resultados y discusión **26**
3.1. Factores de la configuración . . . 26
3.1.1. *Overheads* de *OpenMP* . . . 26
3.1.2. Dispersión de los datos . . . 27
3.1.3. Nivel de optimización en compilación del programa . . . 28
3.2. Resultados de simulaciones . . . 29
3.2.1. Según tamaño del arreglo . . . 29
3.2.2. Según número de procesadores . . . 30
3.2.3. Efecto de lenguaje *assembly* . . . 31
3.2.4. Rendimiento CPU DerivO3 . . . 32
3.2.5. Rendimiento en peor caso . . . 32
3.2.6. Comparación multifactorial . . . 33
3.3. Discusión de resultados . . . 35
3.3.1. Rendimiento solo con paralelismo . . . 35
3.3.2. Sobre *assembly* y optimización . . . 35

Conclusión **36**
4.1. En base a resultados de la experiencia . . . 36
4.2. Trabajo futuro . . . 37

Bibliografía **38**

Appendices **40**

A. Conceptos **41**
A.1. Acrónimos utilizados . . . 41
A.2. Paralelismo y Concurrencia . . . 42
A.3. Niveles de memoria . . . 42
A.4. *Assembler* en linea y extendido . . . 42
A.5. Optimización en GCC . . . 43

B. Programa utilizado **44**
B.1. Programa en C . . . 44
B.2. Conversión a *assembly* . . . 49
B.2.1. Primer *loop* . . . 49
B.2.2. Segundo *loop* . . . 50
B.2.3. Barreras . . . 51

C. Instrucciones RISC-V **53**
C.1. Formato base de instrucciones . . . 53
C.2. Módulos estándares comunes de RISC-V . . . 54
C.2.1. Módulo I . . . 54
C.2.2. Módulo M . . . 55
C.2.3. Módulo A . . . 56
C.2.4. Módulos F y D . . . 56
C.2.5. Módulo C . . . 57

Índice de Tablas

1.1. Módulos estándar de RISC-V 6
1.2. Instrucciones introducidas en el modulo, w indica que son instrucciones para palabras de 32 bits (*single-word*), las instrucciones son análogas para *double-words* de 64 bits. 8

3.1. Aproximación del tiempo dedicado exclusivamente a *overheads* para distintos tamaños. 27
3.2. Variación de resultados para CPU del tipo TimingSimpleCPU y DerivO3CPU 27
3.3. Tiempos de ejecución promedio para distintos niveles de optimización para los programas en C 28
3.4. Evolución del tiempo de ejecución promedio según tamaño 29
3.5. Tiempo de ejecución para distinta cantidad de procesadores 30
3.6. Comparación de tiempos ejecución para distintas implementaciones del programa 31
3.7. Comparación de tiempos ejecución para distintas implementaciones del programa para el procesador DerivO3. 32
3.8. Rendimiento de las 3 variantes del programa en los peores casos. 33
3.9. Evolución del tiempo de ejecución promedio según tamaño 34

Índice de Ilustraciones

2.1. Diagrama de de operación del algoritmo 22

3.1. Parámetros obtenidos de la instrucción *Barrier* para la configuración utilizada en las simulaciones, con 5 procesadores. 27
3.2. Variación del tiempo de ejecución según la optimización del compilar. 28
3.3. Representación gráfica de los resultados. 29
3.4. Tiempo promedio de ejecución para distinta cantidad de procesadores. 31
3.5. Tiempo promedio de ejecución para distintas configuraciones del sistema. . . 32
3.6. Tiempo promedio de ejecución para variantes del programa en su peor caso. 33
3.7. Tiempos de ejecución método tradicional y multiprocesador. 34

A.1. Sintaxis de *assembler* extendido 42

C.1. Sintaxis de instrucciones en RISC-V. Los valores en la parte superior indican la posición de bits 54
C.2. Instrucciones del módulo I, en particular de RV32I, la variante más básica [1, p.17] 55
C.3. Instrucciones del módulo M [1, p.48] 55
C.4. Instrucciones del módulo A [1, p.64] 56
C.5. Instrucciones de los módulos F y D [1, p.53] 57
C.6. Instrucciones del módulo C [1, p.69] 58

Introducción

No es difícil darse cuenta que cada año los computadores disponibles comercialmente mejoran tanto en capacidades como su velocidad. Esto es principalmente atribuible a los avances tecnológicos que ocurren constantemente en el sistema [2].

Quizás el avance mas relevante para esto es la reducción del tamaño de los transistores utilizados en circuitos integrados, actualmente en los 7 nm [3], lo que le permite a la industria producir mejores componentes sin comprometer el tamaño del producto final, para nuestro caso, procesadores.

Esta tendencia no es nueva, pues es un avance que va de la mano con la computación desde sus etapas más tempranas. La evidencia más clara de ello es la llamada *"Ley de Moore"*, observación hecha en el año 1965 por Gordon Moore, la que predijo que cada año se duplicaría la cantidad de transistores que se pueden colocar en un circuito integrado [4].

Y así sucedió durante las décadas siguientes, solo en los años recientes se ha notado una disminución en la proporción que se progresa en este ámbito. Efectivamente, a medida que se reduce el tamaño de estos componentes, surgen nuevos problemas que generan una saturacíon de los circuitos, como lo es por ejemplo la capacidad de disipación de calor para la misma superficie.

Con esta limitante física se vuelve necesario encontrar otras formas de aumentar el rendimiento de los sistemas, aquí es donde entran en juego las ISA (*Instruction Set Architecture*), que se pueden pensar como la abstracción de un computador, ya que son las instrucciones mas básicas que puede realizar y, por lo tanto, definen qué es lo que pueden hacer y cómo lo hacen.

Los principales desarrolladores de ISA han participado en el mercado desde las décadas de 1970- 1980, como es el caso de AMD e Intel [5], y durante este tiempo han implementado una política de desarrollo incremental, es decir, se han centrado en añadir nuevas características y funciones a los modelos antiguos, manteniendo su vigencia, pero también aumentando su valor comercial, obligando a los usuarios a obtener el producto completo sin importar qué tan básico es su objetivo.

En contraste, en la década del 2010 se presentaron las ISA RISC-V, las que además de ser *open-source* (implicando que su implementación no requiere del pago de licencias) utiliza una implementación modular para sus diversas funciones. Lo que significa que si uno quisiera, por ejemplo, utilizar multiplicación/ división de enteros, debería instalar la extensión que incluye dichas instrucciones, lo mismo ocurre para el caso de operaciones decimales y, de esta forma, un usuario puede continuar agregando módulos hasta tener las funcionalidades deseadas en su sistema.

Uno de estos módulos, específicamente la variedad *Atomic*, está orientado a facilitar el paralelismo entre procesadores, una de las formas de mejorar las capacidades de computación al aprovechar de mejor manera la presencia de múltiples procesadores, tan común en estos tiempos, pero no siempre utilizada apropiadamente por los programas. Esta ventaja la logra teniendo funciones que atienden a los problemas más típicos al intentar utilizar este acercamiento, principalmente derivados del uso de memoria cuando es compartida por varios procesadores.

Es por eso que en este trabajo se pretende utilizar las capacidades de paralelismo que nos entrega RISC-V para mejorar la implementación a nivel de *assembler* de algún programa ya existente, y de esta manera lograr reducir el tiempo de computación necesario para su ejecución.

A pesar de esto, tampoco se quiere crear un programa que utilice una cantidad desproporcionada de CPU, por lo que el objetivo final es lograr un equilibrio entre mejorar el tiempo de ejecución y las CPU necesarias para lograrlo.

Con lo anterior, el objetivo de este trabajo es reducir el tiempo de ejecución del proceso de ordenamiento *insertion sort* mediante la división de la operación entre múltiples procesadores, lo que es posible mediante el módulo *Atomic* de RISC-V.

En los capítulos siguientes se sigue el proceso del desarrollo de esta solución, más específicamente:

- El capítulo 1 está dedicado a describir en mayor detalle los conceptos relevantes para este trabajo, como lo son la ISA utilizada, las herramientas que se ocuparán y que variantes de *insertion sort* existen en la literatura.

- Para el capítulo 2 se detallará el programa candidato a mejorar, así como los puntos particulares en donde se pueden realizar los cambios apropiados y su desarrollo.

- En el capítulo 3 se presentarán los resultados obtenidos a partir de las experiencias realizadas, para luego analizar sus implicaciones sobre el objetivo original.

- Las conclusiones obtenidas a partir de la experiencia se detallarán en el capitulo homónimo .

Capítulo 1

Marco teórico y estado del arte

1.1. Marco teórico

1.1.1. Definiciones básicas

Para entender cómo operan las funciones del módulo *Atomic* hace falta definir algunos conceptos comúnmente usados en la literatura

- *Hart*: es un *thread* de *hardware* en un procesador RISC-V, y que desempeña alguna función compartiendo recursos con *threads* similares.

- Carrera de datos: evento que ocurre cuando 2 accesos a memoria de distintos *threads* intentan acceder a la misma dirección, uno tras otro, y por lo menos uno de ellos es una escritura. Si ambos accesos no están sincronizados adecuadamente es posible que el resultado final dependa de cuál acción se ejecutó primero.

- Instrucción *FENCE*: una instrucción que permite sincronizar una escritura a memoria de instrucciones con la etapa de búsqueda de instrucciones (*instruction-fetch*) dentro de un mismo *hart*. En otras palabras se asegura que si un *store* es visible para un *hart*, también será visible para sus *instruction-fetches*

- Atomicidad: una instrucción se considera atómica cuando no existen interrupciones entre que se lee y se escribe a la memoria, ni siquiera desde otros procesadores.

- Direcciones de memoria alineadas naturalmente: Una palabra en la memoria de n bytes se considera naturalmente alineada cuando su dirección en la memoria es múltiplo de n.

- *Instruction fetch*: Primera etapa de la ejecución de una instrucción, en donde se carga la instrucción indicada por el contador de programa a una memoria temporal para ser decodificada y ejecutada.

- ISA: Es el conjunto de instrucciones que conforman el abstracto de un computador,

definiendo que tipos de operaciones puede realizar y como lo hace.

- Sección Crítica: Es la parte de un programa (en programación paralela) en donde se accede a uno de los recursos compartidos con otros procesos. En general si varios de estos procesos intentan acceder a dichos recursos simultáneamente se producen resultados erróneos y/o inesperados, por lo que se hace necesario restringir el acceso de forma tal que solo pueda haber un proceso a la vez trabajando con el.

1.1.2. El procesador

Un procesador es definido formalmente por la RAE como [6]: "Unidad funcional de una computadora u otro dispositivo electrónico, que se encarga de la búsqueda, interpretación y ejecución de instrucciones". A grandes rasgos un procesador es el circuito de un sistema encargado de traducir las instrucciones recibidas (el programa) a operaciones aritméticas y lógicas para que puedan ser ejecutadas.

Por lo general, el procesador es asociado con la "Unidad central de procesamiento" (CPU por sus siglas en Inglés), pero no esta limitado a ella, pues también juegan un papel importante en unidades de procesamiento de gráficos (GPU), tarjetas de sonido y prácticamente de cualquier sistema o subsistema que deba interactuar con datos externos y/o componentes periféricos.

La forma de operación y capacidades de cada procesador dependen estrictamente del set de instrucciones (ISA) en el que está basado, pues este último es el modelo abstracto que define como se comporta el código de máquina cuando se ejecute en la implementación.

1.1.3. La ISA RISC-V

Como se mencionó con anterioridad, RISC-V es una ISA modular, lo que significaba que las instrucciones base son bastante limitadas, necesitando de extensiones para poder obtener mayor funcionalidad. Pero en la realidad existen 4 ISA básicas de RISC-V, las que se diferencian entre sí principalmente en el tamaño de direcciones que soportan. El par principal son RV32I y RV64I, las que soportan direcciones de 32 y 64 bits, respectivamente, un subtipo del primero es RV32E, ISA pensada para soportar aplicaciones más pequeñas y que posee una cantidad reducida de registros. Finalmente la ISA RV128I, una variación futura que teóricamente soportaría direcciones de 128 bits.

La principal lógica tras esta separación es que de esa forma se puede optimizar cada ISA según se necesite sin preocupaciones de que genere conflictos de compatibilidad con el resto de ISA base, aunque esto también tiene la desventaja de complicar el diseño de *hardware* que permita emular una ISA si utiliza otra variante.

Los módulos adicionales de RISC-V están pensados para ser personalizados y configurados según sea necesario en cada implementación, por lo que las extensiones se pueden dividir en las categorías estándar y no estándar, diferenciando entre aquellas que son definidas por la fundación RISC-V y las que no. Dentro de las estándar se encuentran las extensiones indicadas en la tabla 1.1 dada a continuación.

Módulo	**Descripción rápida**
I	Es el módulo básico de operaciones con enteros, operaciones de control y manejo de memoria, además de incluir los registros correspondientes.
M	introduce las operaciones de multiplicación y división de enteros.
A	módulo de operaciones de memoria atómicas.
F	añade registros de punto flotante (decimales) y operaciones de puntos flotante de precisión simple.
D	expande los registros de F y añade instrucciones para punto flotante de doble precisión.
C	incluye instrucciones comunes en forma compacta para poder operar con 16 bits.

Tabla 1.1: Módulos estándar de RISC-V

Uno puede saber cuales ISA incluye la implementación con la que se está trabajando según las letras que proceden la base en su nombre, por ejemplo, un procesador RV32IMC incluiría los módulos M y C además de la base de 32 bits.

1.1.4. El lenguaje *Assembler*

Assembler es un lenguaje de programación de bajo nivel, que tiene la particularidad de ser bastante similar a las instrucciones en lenguaje de máquina para la arquitectura que se esté programando, por lo que las instrucciones específicas entregadas por el lenguaje son completamente dependientes de la ISA con la que se esté trabajando.

Esto en contraste a los lenguajes de alto nivel, que son más universales por lo que tienden a ser más compatibles entre distintas arquitecturas, pero requieren de un interprete o un compilador para convertirse en lenguaje de máquina si se pretende programar con ellos [7].

Esto significa que *assembly* posee la ventaja de poder controlar con un mayor nivel de detalle las acciones del sistema, permitiendo una mejor optimización de las operaciones, a costas de la portabilidad del programa.

1.1.5. La consistencia de memoria

Es el modelo que restringe que valor puede entregar una lectura de memoria, el que normalmente corresponde al último valor escrito en la misma, pero como en sistemas multiprocesadores cada uno de ellos tiene su propio orden de programa, por lo que la definición de 'último' no será universal. Es decir, el modelo es el que define cual valor será el 'último' para todos los procesos que estén participando y leyendo [8].

La elección de un modelo a utilizar, realizada al momento de planificar y crear un procesador, va a impactar directamente su programabilidad y rendimiento futuro, además de afectar la portabilidad de los programas creados para dicho sistema, debido a los posibles problemas de compatibilidad entre implementaciones que utilicen distintos modelos.

RISC-V emplea el modelo de consistencia RVWMO (RISC-V *Weak Memory Ordering*) [9, p.83], diseñado para ser flexible y de alto rendimiento, mientras que el módulo *atomic* le permite soportar el modelo RC_{sc} [9, p.47](modelo *RELEASE CONSISTENCY* manteniendo la consistencia secuencial). El modelo *RELEASE CONSISTENCY* emplea varias distinciones entre las operaciones de memoria, la primera es entre operaciones ordinarias y especiales (cuáles son de datos y cuáles de sincronización, respectivamente). Estas últimas se vuelven a dividir entre *sync* y *nsync* dependiendo si son utilizadas efectivamente o no para sincronizar. Y finalmente, las operaciones *sync* son separadas entre *acquire* y *release*, siendo *acquire* una operación de lectura con la que se obtiene acceso a un espacio de memoria compartida, y *release* una operación de escritura que concede el acceso a dicha memoria compartida.

Por su parte, la consistencia secuencial impone 2 reglas para asegurar el orden: mantener el orden de programa entre operaciones del mismo procesador, y mantener un único orden secuencial entre todas las operaciones. Otra forma de ver la consistencia secuencial es, según Lamport [10] (citado en [11], p.67): "Si el resultado de cualquier ejecución es la misma a la que se obtendría realizando todas las operaciones de todos los procesadores de forma secuencial, y el orden de las operaciones de cada procesador individual en dicha secuencia es la misma al orden especificado por el programa".

De esta forma el modelo RC_{sc} se preocupa de mantener la consistencia secuencial entre las operaciones especiales (definidas por el modelo *RELEASE CONSISTENCY*), lo que se logra distinguiendo las operaciones apropiadas según corresponda para garantizar el mejor orden de programa entre un par de operaciones.

1.1.6. El módulo *Atomic*

Como ya se mencionó anteriormente, el módulo "A" posee instrucciones que apuntan a trabajar con la memoria atómicamente con tal de dar soporte a la sincronización entre múltiples *harts* que se estén ejecutando en el mismo espacio de memoria. Estos 2 tipos de instruccio-

nes son el dúo *load-reserved / store-conditional*, y las instrucciones AMO (*Atomic Memory Operations*). Una lista completa de estas se puede ver en la siguiente tabla.

Instrucción	**Descripción**
lr.w	load-reserved
sc.w	Store-conditional
amoswap.w	SWAP
amoadd.w	ADD
amoxor.w	XOR
amoand.w	AND
amoor.w	OR
amomin.w	Mínimo
amomax.w	Máximo
amominu.w	Mínimo sin signo
amomaxu.w	Máximo sin signo

Tabla 1.2: Instrucciones introducidas en el modulo, w indica que son instrucciones para palabras de 32 bits (*single-word*), las instrucciones son análogas para *double-words* de 64 bits.

Ambos tipos de instrucciones proveen distintos tipos de atomicidad, por un lado las AMO son variantes de instrucciones ya existentes pero ejecutadas atómicamente por si mismas, por el otro, el uso del par *load / store* forma una instrucción atómica entre ambas, con *load-reserved* leyendo y reservando un espacio de memoria, y *store conditional* guardando un valor en la memoria solo si dicha reserva no fue violada, asegurando que se haya seguido la atomicidad deseada.

El principal motivo para incluir ambas variedades de instrucciones es el de contar sus beneficios, las instrucciones AMO escalan de mejor manera para sistemas multiprocesador y son más convenientes al momento de comunicarse con dispositivos I/O, principalmente debido a que realizan la lectura y escritura es un único paso. Por su parte, el par *load / store* permite implementar la operación *compare-and-swap*, la cual a su vez permite implementar cualquier otra operación de sincronización para *single-word*, pero que no puede implementarse directamente en el procesador RISC-V, pues comprometería la simplicidad de su diseño.

Por su parte, todas las instrucciones AMO incluyen dos bits *acquire* y *release* (*aq* y *rl*, respectivamente) que apuntan a mantener la consistencia del sistema (como se mencionó durante la descripción de consistencia de memoria), y que afectan como es vista la instrucción dentro de su mismo *hart*. Si el bit *aq* esta activo, entonces ninguna operación de memoria que venga después de esta en su mismo *hart* puede ocurrir antes de ella. De manera similar, si el bit *rl* está activo, los demás *harts* no observarán esta AMO antes de que el resto de accesos a memoria que le preceden y que sean del mismo *hart*. De esta forma, si ambos bits están activos la instrucción se vuelve secuencialmente consistente, es decir, no puede ser reordenada con otras operaciones de memoria del mismo *hart*.

Estos 2 bits pueden ser utilizados para proteger las secciones críticas de una programa, al utilizar una AMO con *aq* antes de dicha sección, y otra con *rl* al finalizarla.

1.1.7. Métodos de sincronización

Como es de esperar, la principal forma de evitar las carreras de datos es impedir que dos o más procesos intenten acceder a la misma dirección de memoria al mismo tiempo; este proceso es llamado sincronización.

Actualmente existen 5 técnicas básicas para lograrla [12], las que se discuten brevemente a continuación:

- *lock*: Un proceso se reserva el acceso al espacio de memoria denegando el acceso a cualquier otro proceso que intente acceder, hasta que termine la operacion y libere la reserva. Por lo general funciona mediante una llave única que debe ser compartida entre todos los procesos existentes.

- *read-modify-write*: En este caso todos los procesos se ejecutan con normalidad al entrar a la sección critica, pero leen el valor en la memoria antes y después de dicha ejecución. Si ambos valores coinciden, significa que el proceso fue ejecutado "atómicamente" y se continúa con el proceso, de lo contrario se reintenta la operación de la sección crítica.

- *transaction*: Este método requiere de una memoria adaptada específicamente para poder ejecutarla, la que delimita los espacios de código que deben ser ejecutados atómicamente, por lo general hay un *manager* que detecta las carreras de datos mientras ocurren y cancela uno de los procesos al momento.

- *copy-on-write*: Es una técnica relativamente nueva, en la que se permite que varios procesos puedan acceder al recurso compartido, siempre y cuando no se actualicen sus contenidos, y creando una nueva copia si la hay.

- *read-copy-update*: Similar al caso anterior, pero considera solo instrucciones de lectura, y necesita de un sistema que actualice atómicamente los valores de memoria de forma tal que cualquier lectura pendiente al momento de la actualización lea el valor anterior, mientras que las futuras vean el valor actualizado.

Cabe mencionar que no es necesario que estos métodos sean implementados por su cuenta, existen varios modelos que utilizan combinaciones de ellos. Además de que el impacto efectivo sobre el rendimiento de un programa en particular va a depender de la plataforma con que se implemente.

Una forma mas compleja de sincronización son las barreras, las que funcionan evitando que los *harts* continúen ejecutándose mas allá de una cierta sección del código hasta que el

resto de *harts* alcance el mismo punto.

1.1.8. Secciones paralelizables

Como es de esperarse, no todos los problemas se pueden paralelizar completamente, por lo que lo común es dividirlos en secciones que sí lo sean. En particular una sección sería paralelizable si su operación no depende de resultados anteriores (como en un proceso secuencial).

El principal ejemplo de secciones paralelizables son aquellos *loops* con cierto nivel de independencia entre sus partes, los que se pueden dividir mediante el proceso llamado *loop unrolling*.

Además de dicho método hay dos enfoques distintos sobre la forma en que se debiera desenrollar la operación; la selección de cuál depende de qué tipos de independencia poseen los ciclos:

Si un ciclo del *loop* no es directamente dependiente de un ciclo anterior, entonces es posible separar el *loop* de forma tal que cada ciclo se ejecute independientemente en un proceso distinto y de manera simultanea.

Si las secuencias de operaciones dentro del *loop* pueden ser divididas en varias etapas, entonces es posible crear un *pipeline* en que distintos procesos cubran distintas etapas, permitiendo una ejecución más fluida de cada ciclo.

1.1.9. Complejidad temporal

Es una de las medidas utilizadas para describir la velocidad de ejecución de un algoritmo, en donde se utiliza, como medida, la cantidad de operaciones básicas que realiza durante el proceso, pero como el tiempo de ejecución de un mismo programa va a variar, por lo que comúnmente (y para este documento) se utiliza como referencia el peor caso, que es el tiempo que demora el proceso en ejecutarse para cierto *input* donde es necesario ejecutar la mayor cantidad de operaciones posible para resolverlo.

Este sistema ocupa la notación *big O* para diferenciar tiempos, por ejemplo, una función que sea $O(n)$ significa que para un ínput de tamaño n, o un arreglo de n elementos, se deben realizar n operaciones para resolverlo. Mientras que otra función de complejidad $O(1)$ ejecuta un número constante de operaciones, independiente del tamaño del *input*.

Si bien es relativamente sencillo identificar la velocidad, ocupando este sistema cuando el algoritmo es lineal; no ocurre lo mismo para el caso de programación paralela, por lo que se

vuelve necesario ocupar otra forma de medir el cambio en la velocidad.

1.1.10. *SpeedUp*

Esta es otra manera de medir el cambio de eficiencia entre 2 métodos distintos de ejecutar un algoritmo es el *SpeedUp*, definido como [13]:

$$S_N = \frac{\tau_1}{\tau_N}$$

Donde los subíndices indican la cantidad de procesos trabajando en el algoritmo simultáneamente, y τ es el tiempo de ejecución real de dicho programa.

De esta forma el *speedup* será la razón con la cual se medirán las mejoras en este estudio.

1.1.11. *CPU Time* y *Wall Time*

Al momento de medir el tiempo de ejecución de cualquier programa existen dos términos que se pueden medir: *Wall Time* es el tiempo real que se mide entre dos puntos designados por el usuario, mientras que *CPU Time* es el tiempo en que una CPU se encuentra trabajando, por lo que no considera aspectos como tiempos de espera con otros componentes del sistema, como lo seria una interacción con puertos I/O.

Para el caso de este proyecto es necesario estudiar el tiempo real que tarda el programa en ejecutarse, incluyendo el tiempo adicional necesario para sincronizar los múltiples procesadores, o esperando en una barrera, por lo que se debe trabajar con el primero.

La forma de obtenerlo depende del sistema operativo con que se esté trabajando. La utilizada para este trabajo será especificada más adelante en el documento.

1.1.12. Insertion sort

Insertion sort es un algoritmo de ordenamiento para un arreglo en donde el ordenamiento se realiza un componente a la vez, por lo que su efectividad tiende a caer a medida que se utiliza sobre arreglos mas largos, pero que presenta varias ventajas en otros aspectos. Como el hecho de necesitar una cantidad de memoria extra fija para poder llevarse a cabo, y el de poder operar en linea, es decir, ordenar un arreglo a medida que se expande en tiempo real.

La forma básica en que se programa esta operación se muestra a continuación [1, p. 26] (presentada en C para facilitar la comprensión):

```
1         void insertion_sort(long a[], size_t n){
2             for (size_t i = 1, j; i < n; i++) {
3             long x = a[i];
4             for (j = i; j > 0 && a[j-1] > x; j--) {
5                 a[j] = a[j-1];
6                 }
7             a[j] = x;
8             }
9         }
10
```

Como se aprecia el algoritmo utiliza dos *loops* anidados, donde el único recurso utilizado son los valores del arreglo, los que son sobre escritos una vez por cada paso del *loop* interno, y una vez más al salir del mismo. Además, el arreglo se divide en dos secciones, una ordenada, y otra desordenada.

1.1.13. Freedom studio

Freedom studio es una herramienta de desarrollo basada en la IDE *Eclipse* creada por SiFive y pensada para la programación de procesadores RISC-V, más específicamente de aquellos utilizados por las placas que la misma empresa desarrolla [14].

Al momento de comenzar este estudio el único modelo de procesador multinucleo disponible para trabajar/ simular era el modelo u54mc, de 4 procesadores, y presente en la placa *Hifive Unleashed*, la que fue descontinuada durante este año, siendo reemplazada por la placa *Hifive Unmatched* y su procesador multinucleo u74mc.

1.1.14. *gem5*

El simulador *gem5* es una plataforma *open source* modular para el estudio de arquitectura computacional. Si bien su objetivo principal, cuando fue desarrollado hace 15 años, estaba limitado a la área ya mencionada, actualmente es ocupado también en la industria para el desarrollo y análisis de sistemas computacionales [15].

Actualmente *gem5* funciona principalmente en sistemas operativos Linux y Mac OS, habiendo un soporte muy limitado para Windows.

Algunas de sus principales características, y las razones de porque se utilizan en este proyecto, son:

- Es un simulador con precisión a nivel de ciclos, lo que significa que las simulaciones realizadas con él son hechas de acuerdo a cada ciclo de la arquitectura y configuración que se esté trabajando. Esto lo separa de simuladores como QEMU (el simulador por defecto en *freedom Studio*), cuya prioridad es la simulación de una ISA de la manera más rápida posible, lo que reduce su funcionalidad para objetivos como los nuestros, que requieren un análisis temporal del sistema [16, p.1] [17].

  Por otra parte, hacer una simulación mediante un sistema equivalente utilizando RTL también es una forma de obtener resultados precisos a nivel de ciclos, pero este tipo de soluciones tienden a requerir mucho más trabajo por parte del usuario al momento de realizar cambios en la lógica, limitando su utilidad como medio de investigación para casos donde se requieran modificaciones rápidas de lo que se pretenda estudiar.

- *gem5* dispone de múltiples modelos de CPU, con distintos tipos de funcionalidades y precisión según sea requerido por lo que se esté estudiando. Cada uno de los modelos disponibles presentan sus propias variantes para casos aún más específicos. A continuación se describen las categorías de estas:

  - *SimpleCPU*: Son CPU sin un *pipeline*, por lo que se alejan bastante de la realidad al momento de ejecutarse. Dentro de esta categoría destacan 2 subtipos: *AtomicSimpleCPU* y *TimingSimpleCPU*, la primera siendo un modelo en que todas las operaciones y accesos a memoria se realizan instantáneamente, lo que a su vez la convierte en una de las CPU con las que se puede simular lo más rápido posible, mientras que el segundo tipo simula los tiempos de accesos a la memoria, lo que da un poco más de precisión al momento de analizar sus simulaciones.

  - *MinorCPU*: Aparte de simular los accesos a memoria, este modelo también simula un *pipeline* de 4 etapas del tipo *in order* (las instrucciones son ejecutadas por la CPU en el mismo orden en que están presentes en el programa), además de ser un procesador superescalar genérico, lo que le da la habilidad de ejecutar más de una instrucción por ciclo de reloj.

  - *DerivO3CPU*: Similarmente, esta CPU simula un *pipeline out of order* (ejecuta instrucciones en la medida que los recursos lo permitan); el resto de características son bastante similares con la *MinorCPU*.

  - *BaseKVMCPU*: Esta CPU utiliza la interfaz de linux KVM para ejecutar las instrucciones directamente en *hardware*, en vez de simularlas, por lo que entrega los resultados más rápidos de esta lista, pero su disponibilidad es bastante limitada debido a que requiere que el *host* ejecutando *gem5* y la CPU simulada utilicen la misma ISA.

- Actualmente, el simulador soporta varias ISA, con distintos niveles de detalle, como por ejemplo ARM, X86 y RISC-V. Aunque el soporte real al momento de utilizar *gem5* puede variar según el ambiente desde donde se ejecute. En el caso particular de RISC-V, actualmente solo se soporta el modo usuario, teniendo como trabajo pendiente la

arquitectura privilegiada.

- El simulador dispone de 2 modos de funcionamiento: *Syscall Emulation* y *Full System* (SE y FS respectivamente). En modo FS *gem5* simula todo el *hardware* del sistema objetivo y ejecuta el núcleo sin realizar modificaciones, de forma similar a como operan las máquinas virtuales. En contraste, en modo SE solo se simulan las CPU y memorias del sistema, lo que conlleva la limitante de que solo se pueden simular programas en modo usuario.

- Es posible configurar múltiples parámetros de la máquina que se esta simulando, para lograr que el rendimiento de este sea lo mas similar posible al equipo objetivo. Algunos de estos parámetros son la velocidad del reloj interno, distintos niveles de cache, la cantidad de memoria total disponible, y la capacidad / distribución de los conectores entre componentes.

- En caso de no disponer de las especificaciones de un modelo de un sistema en particular, *gem5* contiene una configuración de ejemplo que utiliza valores por defecto, pero fácilmente reconfigurables al momento de ejecutar *gem5* mediante la consola del sistema.

En particular, para poder simular sistemas con múltiples procesadores se dispone de dos opciones, *pthreads* y *OpenMP*.

1.1.15. *OpenMP*

OpenMP es una API que permite la programación paralela con memoria compartida para plataformas que utilicen C/C++ o Fortran. su funcionamiento se basa principalmente en directivas de compilador y rutinas de sus propias librerías.

El paralelismo obtenido al usar *OpenMP* sigue el modelo *fork-join*, lo que implica que un programa se puede dividir en cualquier momento en varios hilos que se ejecuten de forma concurrente o paralela, para luego volver a unirse y continuar ejecutándose de manera secuencial.

Si bien *OpenMP* permite utilizar una diversa cantidad de directivas para controlar el flujo del programa, para nuestra implementación se ocupan principalmente las siguientes:

- *Parallel*: Define la sección paralela del programa, por defecto se generan tantos *threads* como los definidos por la variable de entorno **OMP_NUM_THREADS** (que se debe editar fuera del programa), pero uno puede sobre escribir dicho valor junto a esta declaración. Además, cada uno de los *threads* generados van a ejecutar la región completa del programa, a menos que uno indique lo contrario, lo que se puede lograr separando tareas según su numero de *thread*, u ocupando la directiva *single*.

- *Single*: Indica una sección de código que solo puede ser ejecutada por un *thread*, pero no especifica cual debe hacerlo.

- *Barrier*: Hace que todos los *threads* ejecutándose esperen hasta que todos ellos lleguen a esta directiva.

Las demás directivas disponibles s cumplen roles similares, definiendo si una sección se debe ejecutar atómicamente, limitar su ejecución solo al *thread* maestro, dividir un *loop for* entre todos los *threads* disponibles, o si dicha ejecución se debe hacer de forma ordenada.

Cabe indicar que cada directiva utilizada conlleva un costo de computación que afecta la velocidad de ejecución de un programa, llamada *Overhead*.

1.2. Estado del arte

Para propósitos de este trabajo son de interés las formas alternativas de ejecutar *insertion sort* como un programa monoproceso, y las variaciones que existen del mismo y que buscan aumentar su eficacia en situaciones donde las implementaciones tradicionales tienden a perderla.

A continuación se describen algunas de estas variantes, aunque no necesariamente serán incluidas al momento de realizar las *benchmarks*.

1.2.1. Programa básico en assembler

La implementación básica (ejecutada en un solo proceso) del algoritmo para una ISA RISC-V que se tiene considerada para este estudio es [1, p.29]:

```
1   # RV32I (19 instrucciones, 76 bytes, o 52 bytes con RVC)
2   # a1 es n, a3 apunta a a[0], a4 es i, a5 es j, a6 es x
3       0: 00450693   addi  a3,a0,4    # a3 apunta a a[i]
4       4: 00100713   addi  a4,x0,1    # i = 1
5
6   Outer Loop:
7       8: 00b76463   bltu  a4,a1,10   # si i < n, saltar a Continue Outer loop
8
9   Exit Outer Loop:
10      c: 00008067   jalr  x0,x1,0    # retornar de funcion
11
12  Continue Outer Loop:
```

```
        10: 0006a803   lw     a6,0(a3)    # x = a[i]
        14: 00068613   addi   a2,a3,0     # a2 apunta a a[j]
        18: 00070793   addi   a5,a4,0     # j = i

    Inner Loop:
        1c: ffc62883   lw     a7,-4(a2)   # a7 = a[j-1]
        20: 01185a63   bge    a6,a7,34    # si a[j-1] <= a[i],
                                             saltar a Exit Inner Loop
        24: 01162023   sw     a7,0(a2)    # a[j] = a[j-1]
        28: fff78793   addi   a5,a5,-1    # j--
        2c: ffc60613   addi   a2,a2,-4    # decrementar a2 para apuntar a a[j]
        30: fe0796e3   bne    a5,x0,1c    # si j != 0, saltar a Inner Loop

    Exit Inner Loop:
        34: 00279793   slli   a5,a5,0x2   # multiplicar a5 por 4
        38: 00f507b3   add    a5,a0,a5    # a5 es ahora el byte address de a[j]
        3c: 0107a023   sw     a6,0(a5)    # a[j] = x
        40: 00170713   addi   a4,a4,1     # i++
        44: 00468693   addi   a3,a3,4     # incrementar a3 para apuntar a a[i]
        48: fc1ff06f   jal    x0,8        # saltar a Outer Loop

```

Este código se considerará como la referencia básica para los análisis futuros, pues apunta a la implementación en los mismos sistemas.

1.2.2. *Insertion sort* adaptativo

Una variante del algoritmo encontrado en la literatura [18], que busca aumentar la adaptabilidad de *Insertion sort* (la capacidad de un algoritmo de aprovechar el orden original de sus partes).

Esto lo logra añadiendo una búsqueda binaria al algoritmo, que identifica preliminarmente la posición del elemento que se está ordenando y, además, implementa una expansión bidireccional sobre la sección ordenada del arreglo, reduciendo así la cantidad de comparaciones necesarias para ubicar este nuevo elemento.

Mediante estos cambios se logra reducir la complejidad a $O(n^2/4)$ para el peor caso, lo que también se traduce en una reducción importante de la cantidad de comparaciones y operaciones *shift* realizadas para ordenar arreglos aleatorios.

El principal costo de estas mejoras son un aumento importante en la complejidad del código utilizado, sumado a la necesidad de ocupar una cantidad de memoria adicional igual al tamaño del arreglo a ordenar.

1.2.3. *Insertion sort* de dos elementos

Esta variante de *Insertion sort* [19] apunta a reducir la cantidad de ciclos que deben ejecutarse, esto mediante la inserción de 2 elementos por cada ciclo realizado, lo que implica la necesidad de añadir una comparación entre ambos valores a insertar, así como del espacio para almacenarlas.

De forma similar a la variación anterior, este algoritmo mantiene el aumento de complejidad del código, y también mantiene la complejidad cuadrática ($O(n^2)$) del algoritmo, pero logra reducir el tiempo total de operación para ordenar un arreglo, aunque sin la necesidad de ocupar memoria adicional dependiente del tamaño del arreglo.

Capítulo 2

Metodología

2.1. Análisis y consideraciones previas

2.1.1. Análisis preliminar de *Insertion sort*

Al observar el funcionamiento del algoritmo en 1.1.12 se desprende que, si hay múltiples procesos realizando distintos ciclos del *loop* externo simultáneamente, el único caso en que se podría producir una carrera de datos es cuando varios de ellos se encuentran operando sobre el mismo índice del arreglo ya sea leyendo el valor actual para poder comparar valores o sobrescribiéndolos, por lo que cada escritura dentro del arreglo implica una sección crítica independiente.

Además, también se puede observar que en su forma base *Insertion Sort* tiene una complejidad de $O(n^2)$, también llamada cuadrática. Y en cuanto a la memoria adicional necesaria para funcionar, solo se requiere espacio para un único valor extra (el tamaño de memoria real necesaria va a depender del tipo de número que se este almacenando), sin importar el tamaño del arreglo original.

En base a lo anterior el problema se aproximará en una primera instancia buscando la paralelización del ciclo externo y procurando la atomicidad de los 2 casos en que se escribe sobre el arreglo, o que ambas no interfieran entre sí.

2.1.2. Formalización del problema

Como ya se mencionó anteriormente, el objetivo de este trabajo es lograr una mejora de rendimiento para una operación de *insertion sort* ejecutada en un ambiente de RISC-V mediante paralelismo entre múltiples procesadores.

En particular al implementar paralelismo entre procesadores se espera reducir el tiempo

de ejecución total de la operación (comprobándolo mediante el *speedUp*), a cambio de un aumento en la memoria y registros necesarios para almacenar y ejecutar todo el proceso, que son una consecuencia común de la utilización de *loop unrolling*, sumado a los recursos necesarios para utilizar y sincronizar apropiadamente múltiples procesadores.

Realizando un balance entre estos 2 parámetros y los beneficios obtenidos al variar la cantidad de procesadores involucrados se busca encontrar la cantidad óptima de estos.

Debido a que las instrucciones específicas en *assembler* dependen directamente de la ISA para la que se este programando, por lo que para este trabajo la nueva implementación desarrollada solamente podrá implementarse directamente en procesadores que utilicen la misma ISA, o que por lo menos incluyan los módulos necesarios para poder ejecutarla.

El principal enfoque de este trabajo es la paralelización del *software* del sistema, dándole un especial énfasis a la identificación de secciones críticas del proceso, y la mantención de la atomicidad en la medida que se vuelva necesario. Aunque también se vuelve necesario crear un programa en un lenguaje de mayor nivel (en este caso C) que ejecute nuestra operación como subrutina y coordine apropiadamente múltiples procesadores para su operación.

Con esto en mente, el procedimiento general a seguir es la creación de ambas partes del *software* y asegurarse de que cumplan con sus respectivas funciones, a partir de donde se realizarán pruebas de rendimiento que servirán como base para las pruebas, las que consistirían principalmente en *benchmarks* realizadas mediante el mismo *software* para variadas combinaciones de parámetros involucrados, tales como el largo del arreglo, o sus contenidos. Este proceso se repetirá con las otras variables del programa, como distinta cantidad de procesadores, con tal de obtener un punto de comparación adecuado.

2.1.3. Resultados esperados

El principal resultado que se espera obtener es una reducción del tiempo de ejecución para el método de *insertion sort*, en especial para arreglos de mayor tamaño, que son aquellos donde este algoritmo de ordenamiento pierde potencia, todo esto limitando la cantidad de procesadores adicionales necesarios para la tarea según sea razonable.

2.2. Metodología de trabajo

2.2.1. Parámetros de trabajo

Para el desarrollo de este trabajo se trabajará con la versión 19.0.0.0 de *gem5* (25 de febrero del año 2020), la idea original del trabajo era utilizar un sistema lo más similar posible al ambiente que entrega la placa *Hifive unleashed* y su procesador *u54mc*, cuyas características

principales son sus 4 CPU del tipo *in order*. Lo que en *gem5* significaría utilizar una CPU del tipo *MinoRCPU*, que como ya se mencionó, corresponde a un superescalar del tipo *in order*, por lo que sería el modelo ideal para la tarea. Sin embargo, actualmente existe un *bug* que impide utilizar *threads* correctamente con dicho modelo de CPU (existe un parche para equipos que utilicen *python 3*)[20]. Razón por la cual se optó por modelar un sistema mas genérico, utilizando para ello el modelo *TimigSimpleCPU*, no obstante, para tener una referencia adicional se realizará un pequeño número de mediciones utilizando la CPU *DerivO3*. Aunque los resultados, en ambos, casos no pueden ser comparados directamente debido a la naturaleza superescalar de la segunda.

En cuanto al resto de la configuración utilizada para tomar datos, se utilizó el archivo de configuración por defecto para el modo SE, junto a los siguiente parámetros:

- Memoria compartida de 8192 MB.
- Cache de nivel L1, con 64 kb de memoria de datos, y 16 para instrucciones.
- Cantidad de CPU variable según la medición.
- Un reloj de sistema de 10GHz

2.2.2. Estrategia de medición

Para el estudio se realizaron pruebas de rendimiento del programa, utilizando distintas configuraciones tanto de *input*, como del ambiente de prueba, esto con tal de ver como escala el rendimiento a medida que participa una mayor cantidad de procesadores, o cuando se debe ordenar arreglos de mayor tamaño. Utilizando el algoritmo original de 1 procesador como referencia.

También se pretende observar el efecto de utilizar lenguaje *assembly* para ejecutar las secciones mas importantes del programa, como lo son los *loops* donde se utiliza la mayoría del tiempo de sincronización, o las barreras utilizadas en medio de estos con tal de eliminar el *Overhead* derivado de utilizar la directiva *barrier* de *OpenMP*.

Para medir dicho rendimiento (mediante el tiempo de ejecución) se utiliza la función de *OpenMP* **omp_get_wtime()**, que entrega el tiempo real de ejecución del programa en segundos (*walltime*), medido desde el momento desde que el primer procesador comienza a ordenar los primero elementos, hasta que los últimos valores son ordenados secuencialmente.

Para cada configuración se realizaron múltiples iteraciones con arreglos pseudoaleatorios, a las cuales se les midió el tiempo de ejecución individualmente, para luego promediar los tiempos de dicha configuración, que corresponden a los ilustrados en la sección de resultados. La cantidad de iteraciones realizadas para cada configuración no son iguales entre sí, princi-

palmente debido al limitado tiempo de simulación disponible y como este aumenta al utilizar arreglos con miles de elementos.

Los arreglos utilizados en las pruebas están compuestos por números enteros, positivos y únicos, que corresponden a todos los valores desde 0 hasta el antecesor del tamaño del arreglo (por ejemplo, un arreglo de 1000 valores contiene todos los elementos entre 0 y 999 inclusive); ordenados de manera aleatoria para cada iteración del problema. La única excepción a esto es cuando se está midiendo el rendimiento del programa para el peor caso posible, que corresponde a un arreglo originalmente ordenado de mayor a menor, ya que en cada iteración se debe recorrer completamente la sección ordenada para lograr ubicar el nuevo elemento.

2.2.3. Procedimiento propuesto

La idea principal es diseñar un algoritmo multiprocesador que sea efectivamente paralelo y no solamente concurrente, además de ser escalable para una cantidad indeterminada de procesadores, por lo que el programa buscado debe dar un suficiente nivel de flexibilidad en dicho aspecto.

La idea básica de este algoritmo es que si se dispone de X procesadores, entonces que cada uno pueda ordenar un elemento del arreglo objetivo dentro de la misma iteración del *loop* externo de la función, reduciendo la cantidad total de iteraciones necesarias para ordenar todo el arreglo.

En particular, los pasos que sigue el programa para ordenar un arreglo durante cada iteración es el siguiente:

1. Se seleccionan los siguientes X elementos del arreglo a ordenar para la iteración actual.

2. A cada procesador/*thread* le es asignado uno de estos elementos según su ID en el sistema, en orden de menor a mayor. Ambas operaciones anteriores se hacen de manera secuencial por parte del procesador principal. Desde este punto se inicia la sección paralela propiamente tal.

3. Cada *thread* realiza un pseudo-*insertion sort* de su respectivo elemento, omitiendo los elementos que se ordenan simultáneamente, con la diferencia de que en vez de alterar el arreglo original con el proceso, los números mayores al propio son guardados en un arreglo único al *thread* actual.

4. Una vez que el *thread* termina de ordenar, guarda el último índice registrado, tal que sea visible por los demás *threads*.

5. Todos los *threads* sobrescriben una sección del arreglo original según los contenidos del

arreglo propio, esta sección corresponde a los números ubicados entre el índice guardado propio del *thread* y el del siguiente (o el fin de la sección ordenada si se trata del último procesador).

6. Continuar iterando hasta que resten menos de X elementos a ordenar, los que son ordenados secuencialmente por un único procesador.

Otra forma de verlo es el diagrama de la figura 2.1 a continuación:

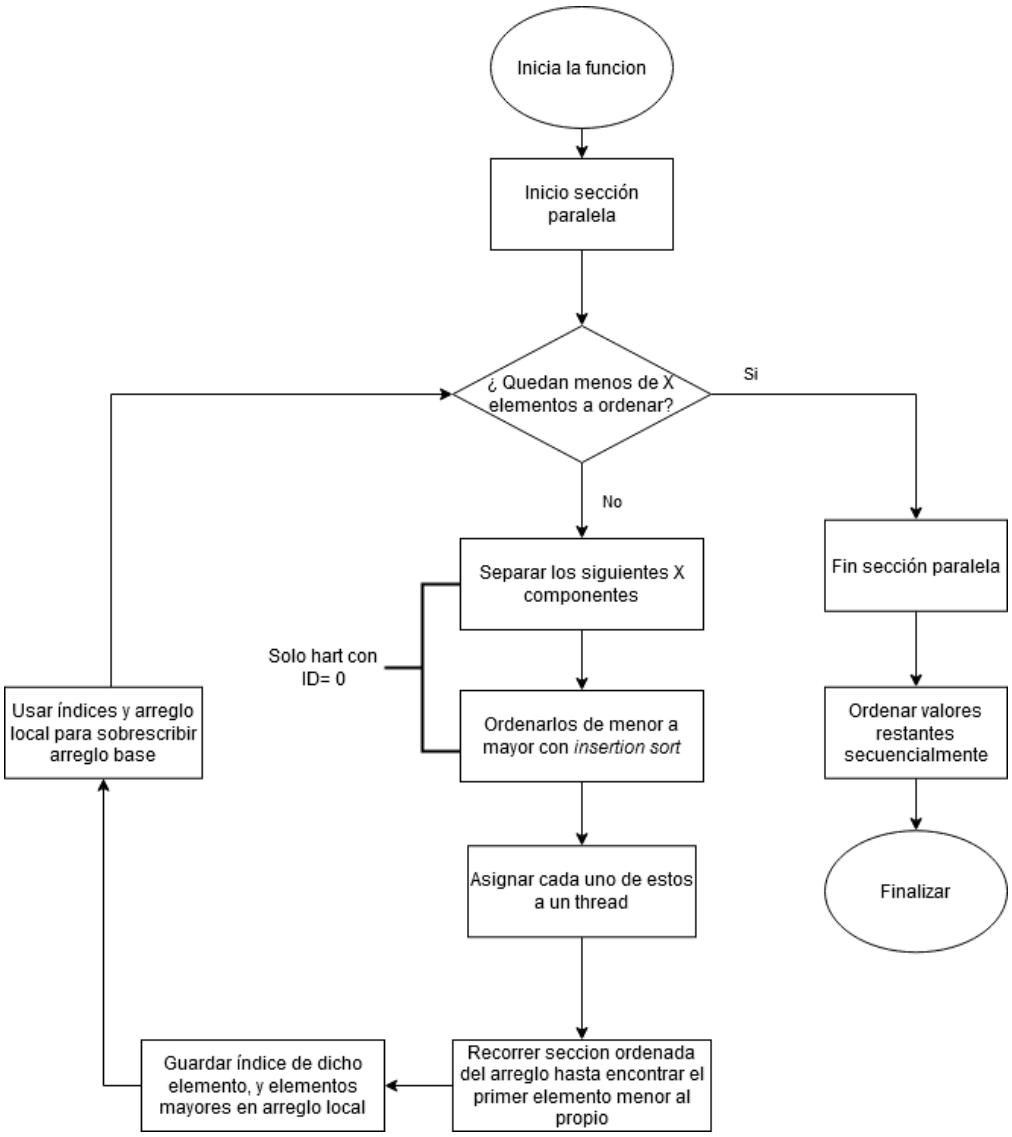

Figura 2.1: Diagrama de de operación del algoritmo

Mediante este método se debe recorrer el arreglo 2 veces por cada iteración. La primera en el paso 3 para que cada *thread* encuentre la posición que corresponde para el número a ordenar y, una segunda, al momento de sobrescribir el arreglo original durante el paso 5.

Lo que resulta es que esta metodología debiera tener una complejidad de $O(2n^2/X)$, por lo que este método es capaz de mejorar el tiempo real de ejecución del programa solo si se asignan más de 2 procesadores a la tarea (asumiendo una sincronización perfecta sin el costo temporal de utilizar cualquier función/ variable auxiliar).

2.2.4. Ejemplo de aplicación

A modo de ejemplo, a continuación se muestra el paso a paso de como se ordenaría una secuencia de números utilizando el algoritmo diseñado.

En este ejemplo, el algoritmo ya se encuentra en ejecución, con 3 procesadores (I, II y III) trabajando en ello, y el estado actual del arreglo base es:

$$1, 3, 4, 6, 7, 8, 10, 2, 5, 9$$

donde los elementos que se ordenarán, durante esta iteración, corresponden a 2, 5 y 9. Los que se asignan a cada procesador según su ID en orden de menor a mayor, por lo que en este caso

$$\begin{array}{cc} I & ordena\ 2 \\ II & ordena\ 5 \\ III & ordena\ 9 \end{array}$$

Una vez asignados los elementos, cada procesador recorre la parte ordenada del arreglo guardando los elementos mayores al propio en su arreglo local, de forma que el estado de los 3 arreglos al finalizar es:

$$\begin{array}{ccc} I & => & 10_0, 8_1, 7_2, 6_3, 4_4, 3_5, 2_6 \\ II & => & 10_0, 8_1, 7_2, 6_3, 5_4 \\ III & => & 10_0, 9_1 \end{array}$$

(Donde los subíndices indican su posición dentro del arreglo)

Con los arreglos locales listos, los procesadores pasan a sobrescribir simultáneamente el arreglo base en la sección que cubre entre su elemento asignado y el del siguiente procesador. De forma que el arreglo es sobrescrito de la siguiente forma:

$$\begin{array}{ccccccc} 1 & & & & & & \\ & 2, 3, 4 & & & & & \\ & & & 5, 6, 7, 8 & & & \\ & & & & & 9, 10 & \\ - & I & & II & & III & \end{array}$$

De esta manera el arreglo base al final de esta iteración es

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

2.2.5. Implementación del programa

A continuación se muestra un extracto del *script* utilizado, en específico, la función encargada de ordenar el arreglo, omitiendo las funciones auxiliares orientadas a crear el arreglo aleatorio, o verificar que el arreglo se haya ordenado apropiadamente.

```
void Ordenar(int n){
i=1;
int x;
int num_harts;
int arr [10][n];
int vals[10] = {0,0,0,0,0,0,0,0,0,0};        // contenedor de valores a ordenar
int auxs[11] = {0,0,0,0,0,0,0,0,0,0,0};      // contenedor de subindices
#pragma omp parallel num_threads(5) firstprivate(i) private(j)
{
  num_harts = omp_get_num_threads();
  int ID = omp_get_thread_num();
  for (i,j; i < (n-(num_harts-1)); i+=num_harts) {

    #pragma omp barrier
    // Ordena los valores
    if(ID==0){
      for(int m =0;m<num_harts;m++){
        vals[m]=arreglo[i+m];
      }

        for (int i= 1,j; i < num_harts; i++) {
          int x = vals[i];
        for (j = i; j > 0 && vals[j-1] > x; j--) {
          vals[j] = vals[j-1];
        }
        vals[j] = x;
      }
      auxs[num_harts]=i;
    }
    #pragma omp barrier

    for(j=i;j>0 && arreglo[j-1]>vals[ID];j--){
      arr[ID][i-j]=arreglo[j-1];
    }
    arr[ID][i-j] = vals[ID];
    auxs[ID]=j;

    #pragma omp barrier
    int m=0;

    for(m=0;m<(auxs[ID+1]-auxs[ID]+1);m++){
      arreglo[auxs[ID]+m+ID] = arr[ID][i-auxs[ID]-m];
    }

```

```
45      #pragma omp barrier
46
47    }
48  }
49  //Ordenar secuencialmente los elementos faltantes.
50  x = n%num_harts;
51  if (x!=1){
52    for(i;i<n;i++){
53      x=arreglo[i];
54      for (j = i; j > 0 && arreglo[j-1] > x; j--) {
55        arreglo[j] = arreglo[j-1];
56      }
57      arreglo[j] = x;
58    }}
59  }
```

Para entender el código hace falta aclarar algunas partes de él que no se muestran en el extracto anterior, pero que se ejecutan o declaran previamente:

- Si bien teóricamente el código es escalable indefinidamente, esta versión esta pensada con un límite superior de 10 *threads*, como se puede desprender del largo de los arreglos auxiliares *vals*, *auxs*, y la cantidad de filas de *arr*. Además, al utilizar *num_threads(5)*, al momento de declarar la sección paralela, se está limitando la cantidad de *threads* a 5, sin importar si se encuentran más disponibles.

- *arreglo* es el arreglo desordenado que se busca ordenar de menor a mayor con esta función.

- i y j son enteros definidos fuera de la función. Todos los *threads* comparten y utilizan el mismo valor base de i (por la declaración *firstprivate(i)*), mientras que j es única para cada uno.

- El arreglo *vals* contiene los siguientes X elementos de arreglo, que son los próximos a ordenar, donde X es el número de *threads*. Ellos se ordenan de menor a mayor antes de proceder a ordenar *arreglo*.

- *auxs* contiene los índices del último valor ordenado por cada *thread*, mientras que su último elemento contiene el valor de i de la iteración correspondiente.

- La matriz *arr* se usa para que cada *thread* pueda hacer el pseudo-*insertion sort* de manera independiente del resto, ya que solo un *thread* puede acceder a una cierta fila de ella.

- En caso de que la cantidad de valores restantes de *arreglo* a ordenar sea menor que los X *threads*, dichos elementos se ordenan secuencialmente mediante un *insertion sort* tradicional.

Capítulo 3

Resultados y discusión

En este capitulo se procede a ilustrar los resultados obtenidos desde las simulaciones, algunos parámetros de la configuración que influyeron en estos y, finalmente, se analizarán los resultados obtenidos y se discutirán si cumplen los objetivos planteados.

3.1. Factores de la configuración

Para poder comprender apropiadamente los resultados obtenidos del proceso, primero se deben conocer aquellos factores externos que son parte de la implementación del problema, y que tienen la capacidad de afectar el rendimiento de este.

3.1.1. *Overheads* de *OpenMP*

Como se mencionó en la sección 1.1.15, cada directiva de *OpenMP* conlleva un cierto costo temporal al ser llamadas por el programa. En este caso, la implementación final utiliza 4 *barriers* por cada iteración que realiza, por lo que su presencia e impacto no puede ser ignorado. Con tal de determinar su impacto sobre el programa se utilizó la *benchmark* desarrollada por el centro de procesamiento paralelo de Edimburgo (EPCC) [21]

Los autores recomiendan utilizar los resultados promedio para 10 - 20 ejecuciones de su *benchmark* con tal de obtener un resultado mas realista, ya que es común la obtención de resultados variables, por lo que los resultados presentados a continuación en la figura 3.1 corresponden al promedio de 20 mediciones.

```
-----------------------------------------------------
Computing BARRIER time using 2560 reps

Sample_size       Average     Min         Max          S.D.          Outliers
 20                1.222822   1.222780   1.222966    0.000040      1

BARRIER time     = 1.222822 microseconds +/- 0.000078
BARRIER overhead = 1.119817 microseconds +/- 0.000081

-----------------------------------------------------
```

Figura 3.1: Parámetros obtenidos de la instrucción *Barrier* para la configuración utilizada en las simulaciones, con 5 procesadores.

Como se puede observar el *overhead* obtenido para la configuración utilizada corresponde a cerca de 1.12 μs por cada llamado realizado, los que suma un total de 4.48 μs por cada iteración que deba hacer el programa. Para darle una perspectiva a esta lectura, en la tabla 3.1 se estima el tiempo total gastado en *overheads* para un sistema de 5 procesadores y largo variante. Recordando por supuesto que como la seccion del programa que involucra las barreras es paralela, cada CPU ocupará el tiempo indicado dentro de una misma iteración.

Cantidad de elementos en el arreglo	Número de iteraciones necesarias	Tiempo total aproximado (μs)
100	19	85.12
500	99	443.52
1000	199	891.52
10000	1999	8955.52
30000	5999	26875.52

Tabla 3.1: Aproximación del tiempo dedicado exclusivamente a *overheads* para distintos tamaños.

3.1.2. Dispersión de los datos

Con motivo similar al que llevó a promediar los resultados de la *benchmark*, en la tabla 3.2 a continuación se muestran los resultados promedio de 1000 iteraciones del programa sobre un mismo arreglo de 5000 elementos y 5 CPU, con tal de observar las variaciones en los resultados para 2 tipos de CPU distintas.

	TimingSimpleCPU	DerivO3CPU
Prom	0,018228038	0,0035897
Desviacion	1,20167E-06	1,658E-06
Minimo	0,018228	0,00358
Maximo	0,018266	0,003595

Tabla 3.2: Variación de resultados para CPU del tipo TimingSimpleCPU y DerivO3CPU

3.1.3. Nivel de optimización en compilación del programa

Si bien para todas las mediciones realizadas durante este trabajo utilizaron programas compilados con el nivel de optimización por defecto del compilador GCC (-O0), con tal de tener una referencia del impacto que causaría su variación sobre los tiempos de ejecución, se realizaron mediciones de rendimiento utilizando distintos niveles de esta.

Por lo que la tabla 3.3 muestra como varía la velocidad del programa para ordenar un arreglo específico de 5000 componentes (en particular el arreglo correspondiente al peor caso) con 5 procesadores para distintos niveles de optimización al compilar.

Nivel optimización	Tiempo 1 CPU (s)	Tiempo 5 CPU (s)
O0	0,626038	0,4377527
O1	0,163346	0,0996473
O2	0,138386	0,0946181
O3	0,138386	0,0996646

Tabla 3.3: Tiempos de ejecución promedio para distintos niveles de optimización para los programas en C

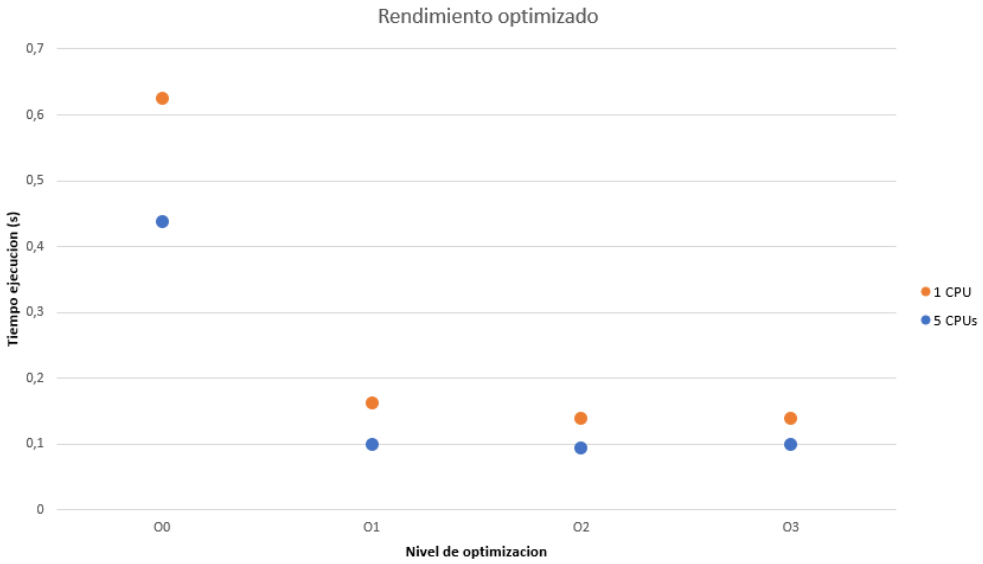

Figura 3.2: Variación del tiempo de ejecución según la optimización del compilar.

3.2. Resultados de simulaciones

3.2.1. Según tamaño del arreglo

Con tal de comprobar la evolución del tiempo de ejecución necesario, en la medida que aumenta el tamaño de los arreglos a ordenar, se midieron dichos tiempos con arreglos de distintos tamaños dentro del rango de los 500 a 30000 elementos, tanto para el programa estándar como para la versión multiprocesador (utilizando 5 CPU).

Los resultados se muestran en la tabla a y gráfico a continuación:

Largo arreglo	Tiempo 1 CPU (s)	Tiempo 5 CPU (s)	*SpeedUp*
500	0,00325	0,00285	1,140593
1000	0,012724	0,010602	1,200051
5000	0,313881	0,248396	1,263629
10000	1,251743	0,942925	1,327511
15000	2,877229	2,246692	1,280651
20000	5,099454	3,987368	1,278902
25000	7,951421	6,200885	1,282304
30000	11,45325	9,140584	1,253011

Tabla 3.4: Evolución del tiempo de ejecución promedio según tamaño

Figura 3.3: Representación gráfica de los resultados.

3.2.2. Según número de procesadores

Para comprobar la complejidad temporal teórica del algoritmo, en lo que respecta a su reducción mediante la presencia de más procesadores, se midió el tiempo de ejecución del programa con arreglos de 1000 elementos, con distintas configuraciones en el número de procesadores participantes.

La tabla 3.5 y figura 3.4 muestran dichos resultados para un rango entre 2 y 50 procesadores utilizando el programa creado, mientras que el resultado mostrado para 1 procesador corresponde a aquel obtenido utilizando el método tradicional de *insertion sort*, a manera de como referencia.

La medición marcada $*$ utiliza un arreglo de 1001 elementos, con tal de comprobar el impacto sobre el rendimiento de la sección secuencial final de la función. En el caso particular de 40 *harts* un arreglo de 1000 elementos genera la sección secuencial más larga posible, mientras que 1001 no la requiere.

Número de procesadores	Tiempo ejecución (s)	*SpeedUp*
1	0,012724	1
2	0,025896	0,491331
3	0,017546	0,725131
4	0,013285	0,957718
5	0,010522	1,209283
6	0,008831	1,440836
7	0,007598	1,67455
8	0,006694	1,900681
9	0,005784	2,19995
10	0,005473	2,324708
11	0,005024	2,532382
12	0,004524	2,812519
13	0,004386	2,900895
14	0,003996	3,18425
15	0,003868	3,289528
20	0,003375	3,76968
30	0,002562	4,96616
40	0,003143	4,048022
40*	0,002032	6,262366
50	0,002526	5,038012

Tabla 3.5: Tiempo de ejecución para distinta cantidad de procesadores

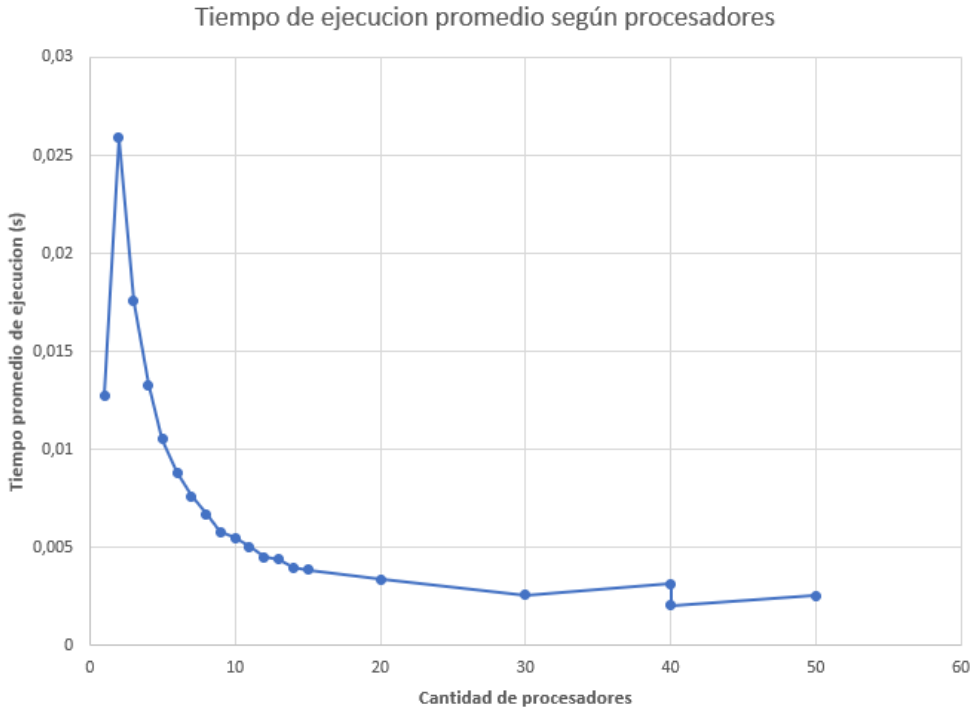

Figura 3.4: Tiempo promedio de ejecución para distinta cantidad de procesadores.

3.2.3. Efecto de lenguaje *assembly*

Para comprobar el efecto de reemplazar secciones claves del código original en C por su equivalente en *assembly*, se comparó el tiempo de ejecución promedio entre la versión estándar del programa multiprocesador con la versión donde los *loops* principales y barreras fueron reemplazados por código en dicho lenguaje, ambos utilizando 5 CPU para ordenar arreglos de 1000 elementos. El detalle de los cambios sobre el código básico mostrado en 2.2.5 se puede encontrar en la sección B.2 del anexo.

La tabla 3.6 muestra brevemente los resultados particulares, con el *SpeedUp*, utilizando la variante en C como base de la comparación.

Versión del programa	Tiempo (s)
C	0,010602464
ASM	0,003407266
SpeedUp	3,111721832

Tabla 3.6: Comparación de tiempos ejecución para distintas implementaciones del programa

3.2.4. Rendimiento CPU DerivO3

Con tal de verificar la consistencia de los resultados en simulaciones mas complejas se repitió la sección anterior, pero utilizando una CPU del tipo DerivO3. Cabe recordar que debido a la naturaleza superescalar de estas CPU, los tiempos de ejecución no se pueden comparar apropiadamente con los que se han mostrado hasta el momento en las secciones anteriores.

Version del programa	Tiempo (s)
C	0,00232668
ASM	0,000628971
SpeedUp	3,699184774

Tabla 3.7: Comparación de tiempos ejecución para distintas implementaciones del programa para el procesador DerivO3.

En el gráfico de la figura 3.5 se muestran los resultados de las dos secciones anteriores con tal de ilustrar las diferencias entre los lenguajes *assembler* y C, y entre los dos tipos de CPU utilizadas. Donde grupo 1 se refiere al par de implementaciones utilizando C, mientras que grupo 2 se refiere a las versiones con reemplazos en *assembly*.

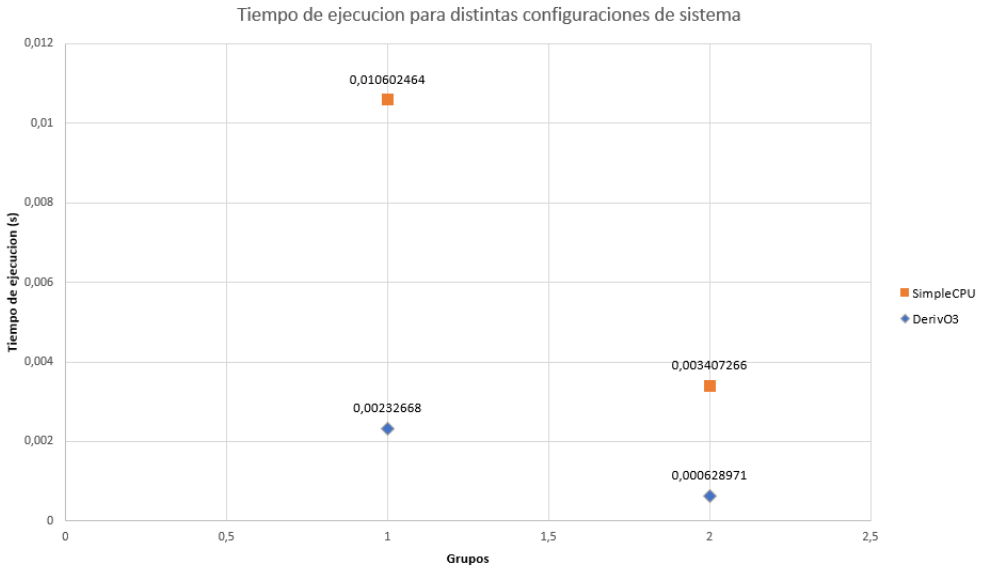

Figura 3.5: Tiempo promedio de ejecución para distintas configuraciones del sistema.

3.2.5. Rendimiento en peor caso

Si bien hasta el momento todos los resultados mostrados corresponden a los tiempos de ejecución de arreglos aleatorios, es apropiado ver el comportamiento en los peores casos del algoritmo, que corresponde al tiempo de ejecución máximo para el programa.

Razón por la cual se realizaron mediciones para distintos tamaños del arreglo y programas, por lo que se compara el rendimiento entre los programas en C y *assembly* de 5 CPU, con el programa estándar de 1 procesador en C al ordenar los peores arreglos posibles de 1000, 5000 y 10000 elementos.

Tamaño del arreglo	Tiempo 1 CPU (s)	Tiempo 5 CPU C (s)	Tiempo 5 CPU ASM (s)	*SpeedUp* C	*SpeedUp* ASM
1000	0,02521	0,0182318	0,005521	1,3827324	4,566147437
5000	0,626038	0,4377527	0,1363741	1,4301171	4,590590149
10000	2,502073	1,7423237	0,5396107	1,436055	4,636810575

Tabla 3.8: Rendimiento de las 3 variantes del programa en los peores casos.

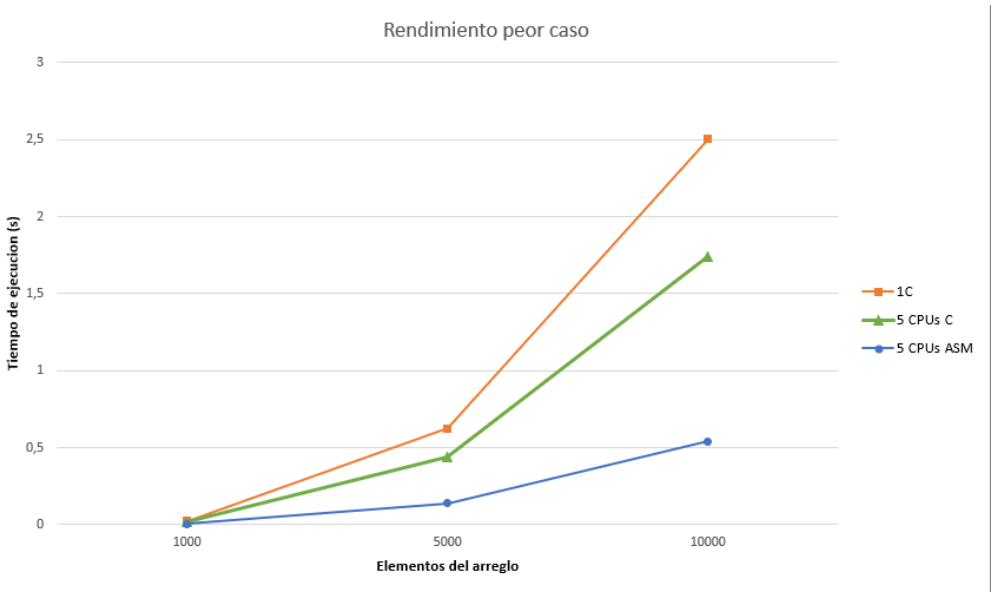

Figura 3.6: Tiempo promedio de ejecución para variantes del programa en su peor caso.

3.2.6. Comparación multifactorial

Hasta esta sección, solo se han realizado simulaciones en las que varía un único factor del sistema, con el objetivo de analizar el impacto específico de dicho cambio, pero hasta el momento no se ha comprobado como se componen estos factores, por lo que en esta última experiencia se realiza una comparación entre los tiempos de ejecución del *insertion sort* original (utilizando las mediciones obtenidas en 3.2) y la versión paralela, con *assembly* de 8 procesadores, comparando, una vez más, el comportamiento para arreglos de distinto tamaño.

Largo arreglo	Tiempo 8 CPU (s)	*SpeedUp*
1000	0,002318	5,489889
5000	0,047291	6,637245
10000	0,183135	6,835072
15000	0,409666	7,023357
20000	0,724657	7,037063
25000	1,12494	7,068307
30000	1,620415	7,068098

Tabla 3.9: Evolución del tiempo de ejecución promedio según tamaño

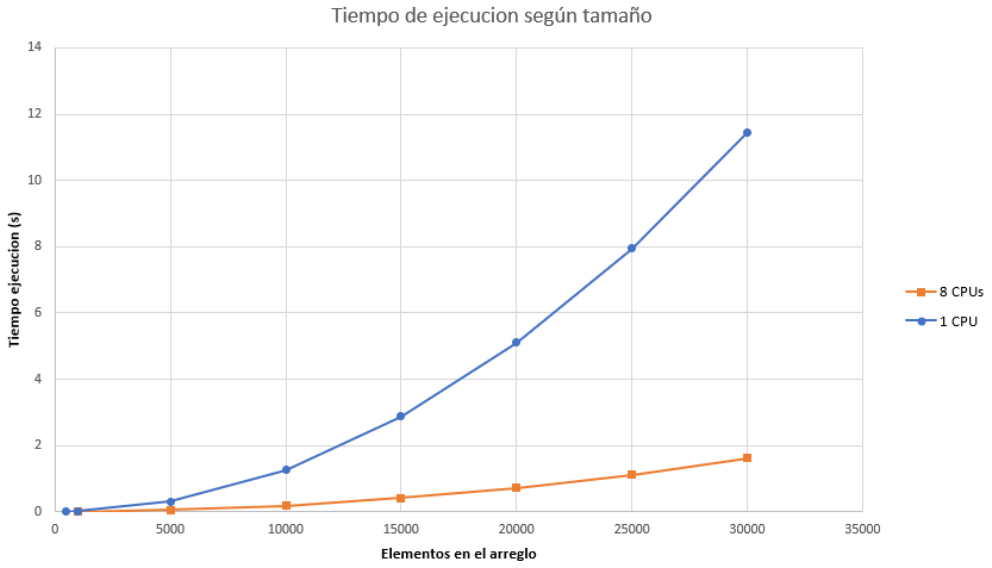

Figura 3.7: Tiempos de ejecución método tradicional y multiprocesador.

3.3. Discusión de resultados

3.3.1. Rendimiento solo con paralelismo

Con los resultados obtenidos en la sección 3.2 se puede observar que el *SpeedUp* final no varía de manera significativa según el tamaño del arreglo trabajado, mientras que, al introducir código *assembler* en el programa, la variación se vuelve más aparente, aunque con niveles decrecientes.

En cuanto a la cantidad de procesadores participantes, el *SpeedUp* aumenta en la medida que se agregan más procesadores para trabajar en el programa, como es de esperar, pero el rendimiento solo supera al método estándar cuando se incluyen al menos 5 procesadores, siendo necesarios 9 para lograr reducir el tiempo de ejecución a la mitad. Este *SpeedUp* continúa aumentando en la medida que se continúan agregando CPU, con la consideración de que su aumento implica también que aumente el tamaño de la sección secuencial que se realiza al final de la función, hasta el punto en que llega a afectar notoriamente el rendimiento de la operación completa al utilizar una cantidad cercana a 20 CPU, como es el caso mostrado de 40 procesadores ordenando arreglos de 1000 y 1001 elementos, que corresponden a los tamaño máximos y mínimos de dicha sección, respectivamente.

3.3.2. Sobre *assembly* y optimización

Los reemplazos realizados con código *assembler* lograron aumentar el *speedUp* obtenido de manera significativa sobre su equivalente paralelo en C bajo configuraciones similares, logrando valores cerca de tres veces mayores.
Por lo que al combinar los efectos de este cambio con el obtenido al aumentar la cantidad de procesadores, que trabajan en el problema, se obtienen niveles de *speedUp* como los alcanzados en la sección 3.2.6.

Por otra parte, el *speedUp* alcanzado mediante este proceso se vuelve equivalente al logrado con la variante en C del programa clásico de 1 CPU compilado con niveles de optimización mayores (O2 y O3). Aunque de acuerdo a las mediciones realizadas con 8 CPU, es posible de todas formas reducir el tiempo de ejecución mediante la inclusión de más procesadores. Cabe recordar que el uso de *assembler* en linea limita la capacidad de optimización del compilador. Por lo que las versiones ASM que se están comparando fueron compiladas con el nivel de optimización por defecto O0.

Conclusión

4.1. En base a resultados de la experiencia

Si bien los resultados obtenidos en esta experiencia demuestran que el paralelismo logra reducir en manera considerable el tiempo de ejecución de *insertion sort*, esto conlleva un costo mayor en los recursos necesarios para poder operar, lo que significa la pérdida de portabilidad y la independencia de memoria adicional necesaria, principales ventajas del algoritmo original.

Por otra parte, el hecho de necesitar dicha cantidad de memoria adicional y al menos 5 procesadores para lograr un aumento en el rendimiento sobre el método tradicional (sin optimizar mediante el compilador), significa que no hubiera sido posible obtener resultados positivos de haber implementado el algoritmo paralelo con la idea original del proyecto, recordando que el procesador U54mc solo posee 4 procesadores.

En cuanto a la implementación de *assembly* en secciones clave del programa, su inclusión es dependiente de las capacidades de optimización que presente el compilador utilizado en la experiencia particular, por ejemplo, el compilador GCC utilizado durante este proyecto logra un mejor rendimiento, pero si no se tiene acceso a un compilador similar, entonces es conveniente implementar *assembly* para aumentar considerablemente el rendimiento.

Estos factores que limitan su aplicación sobre *hardware*, sumados al hecho de que la mayor parte del *insertion sort* tradicional se puede considerar una sección crítica con dependencia entre cada iteración, nos lleva a considerar que dicho algoritmo no es el mejor candidato para ser paralelizado, con el fin de ser utilizado para su propósito de manera realista, ya que los recursos necesarios impide que sea implementado en sistemas simples, como el *insertion sort* original, mientras que los sistemas que sí pueden implementarlo también pueden utilizar algoritmos mas eficientes.

Pero lo anterior no significa que no se haya cumplido el objetivo de este proyecto, que era mejorar el rendimiento de *insertion sort* mediante el uso de paralelismo entre múltiples procesadores y el lenguaje *assembly*.

4.2. Trabajo futuro

Como este trabajo se vio limitado a un ambiente de simulaciones todavía queda el proceso de validar los resultados utilizando una plataforma que implemente la arquitectura RISC-V, como lo serían las placas de *SiFive* mencionadas con anterioridad, adaptando la configuración de la simulación y el programa acordemente, con tal que la ejecución sea lo más similar posible al caso real en lo particular. Recordando que la configuración utilizada para las simulaciones presentadas corresponden a un caso genérico de procesadores y memoria, sin simular el *pipeline* de los primeros.

En forma paralela a lo anterior se debe comprobar el comportamiento del algoritmo frente a distintas configuraciones del simulador *gem5*, en especial para las CPU *In Order* y *Out of Order*, así como el modo de simulación *Full System*. Esto para lograr obtener simulaciones mas realistas en el futuro para cada caso de estudio particular.

En cuanto a posibles cambios sobre el algoritmo y programa propiamente tal, la sección de iteraciones secuenciales que se ejecuta, en caso de que resten menos elementos desordenados en el arreglo que la cantidad de procesadores, daña de manera notoria la escalabilidad de todo el programa. Razón por la que debiera ser reemplazado por una variante del resto de iteraciones donde se utilice la cantidad justa de procesadores necesarios, con tal de que dicho resto del arreglo también sea ordenado de manera paralela.

De forma similar se debiera ajustar el programa para reducir la memoria adicional necesaria por cada procesador para poder operar, de forma tal que al menos requiera una cantidad de memoria adicional independiente del tamaño del arreglo objetivo, como la versión original. Lo que permitiría implementarlo en sistemas más limitados en aquel aspecto.

Bibliografía

[1] D. A. Patterson and A. Waterman, *Guía Práctica de RISC-V: El Atlas de una Arquitectura Abierta*. 1ª ed., (2018. May. 4). [En linea]. Disponible en: `http://riscvbook.com/spanish/`.

[2] D. A. Patterson and J. L. Hennessy, *Arquitectura de computadores: Un enfoque cuantitativo*. 1*st* ed. Madrid: MacGraw-Hill, 2002.

[3] TSMC, "7nm technology," Taiwan Semiconductor Manufacturing Company Limited. [En linea]. Disponible en: `https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_7nm`. [Consultado: 04-Jan-2021].

[4] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[5] S. P. Morse, "The intel 8086 chip and the future of microprocessor design," *Computer*, vol. 50, no. 4, pp. 8–9, 2017.

[6] Diccionario de la lengua española 23ª ed. 2020., *procesador*. [En linea] Real Academia Española. Disponible en: `https://dle.rae.es`. [Consultado 8 Aug 2020].

[7] B. Archer, *Assembly Language For Students*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pp. 15–26, 1990.

[9] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*. 2019. [En linea]. Disponible: `https://riscv.org/technical/specifications/`.

[10] Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[11] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[12] V. Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," *SIGPLAN Not.*, vol. 50, p. 1–10, Jan. 2015.

[13] "Efficiency of parallel algorithm," sdsc.edu, 2021, [En linea]. Disponible en: `https://www.sdsc.edu/~majumdar/thesis/node47.html`. [Consultado: 03-Jan-2021].

[14] "Freedom studio version 2019.03 - sifive," [En linea]. Disponible en: `https://www.sifive.com/blog/freedom-studio-version-2019.03`. [Consultado: 03-Jan-2021].

[15] "gem5: The gem5 simulator system," [En linea]. Disponible en: `https://www.gem5.org/`. [Consultado: 03-Jan-2021].

[16] T. Ta, L. Cheng, and C. Batten, "Simulating multi-core risc-v systems in gem5," in *Second Workshop on Computer Architecture Research with RISC-V*, 2018.

[17] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, 2012.

[18] K. Nenwani, V. Mane, and S. Bharne, "Enhancing adaptability of insertion sort through 2-way expansion," in *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, pp. 843–847, 2014.

[19] Wang Min, "Analysis on 2-element insertion sort algorithm," in *2010 International Conference On Computer Design and Applications*, vol. 1, pp. V1–143–V1–146, 2010.

[20] "Revert "cpu: fix how a thread starts up in minorcpu"," 2020. [En linea]. Disponible en: `https://gem5-review.googlesource.com/c/public/gem5/+/18604`. [Consultado 23-Dec-2020].

[21] M. Bull, "Measuring synchronisation and scheduling overheads in openmp," *Proceedings of First European Workshop on OpenMP*, pp. 99–105, 1999.

[22] "Extended asm (using the gnu compiler collection (gcc))," Gcc.gnu.org, 2020, [En linea]. Disponible en: `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`. [Consultado 28-Dec-2020].

[23] "Optimize options (using the gnu compiler collection (gcc))," Gcc.gnu.org, 2020, [En linea]. Disponible en: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`. [Consultado 28-Dec-2020].

[24] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified.* 2019. [En linea]. Disponible: `https://riscv.org/technical/specifications/`.

Appendices

Apéndice A

Conceptos

Si bien los conceptos principales del proyecto fueron explicados dentro del documento principal, hay algunas aclaraciones sobre términos relacionados al mismo, peor no necesariamente involucrados en la experiencia, por lo que se explican a continuación.

A.1. Acrónimos utilizados

1. API (*application programming interface*): Es una interfaz que permite interacciones entre múltiples aplicaciones de *software*, definiendo la manera en que se realizan estas mismas mediante la colección de funciones y subrutinas que posee.

2. ASM (*assembler*): abreviación de *assembly* .

3. CPU (*central processing unit*): Circuito electrónico encargado de ejecutar las instrucciones de un programa.

4. GPU (*graphics processing unit*): Circuito electrónico especializado en la creación de imágenes a ser mostradas en un *display*.

5. IDE (*integrated development environment*): Aplicación que ofrece funciones y servicios orientados a facilitar la programación y desarrollo de *software*.

6. ISA (*instruction set architecture*): Conjunto de instrucciones que conforman el modelo abstracto de un computador.

7. RTL (*Register-transfer level*): Nivel de abstracción utilizado para representar un circuito de la manera mas precisa posible mediante la modelación del flujo de señales.

A.2. Paralelismo y Concurrencia

La programación concurrente se basa en el uso de múltiples *threads* con tal de cumplir una tarea en específico mediante la división de la carga que cada uno debe ejecutar, pero ello no siempre implica que dichas secciones se ejecuten al mismo tiempo ni ciclo de reloj, ya sea por restricciones de la máquina que lo esté ejecutando, o que exista una sección crítica entre los múltiples *threads* que lo impida.

Dicho esto, la programación paralela es el subgrupo de la programación concurrente en que aquellos *threads* si se ejecutan simultáneamente para obtener el resultado deseado.

A.3. Niveles de memoria

Uno de los principales problemas que presentan las memorias cache es el *tradeoff* entre latencia y *hit rate* (el porcentaje de accesos a la cache en que se encuentra lo buscado), con caches de mayor tamaño mostrando un aumento de ambas respecto a cache mas compactas. Es por ello que se suele utilizar una combinación de varias cache de distinto tamaño con tal de equilibrar los costos y beneficios, de esa forma si no se logra un *hit* al revisar el primer nivel de cache (L1), se continua buscando en el segundo nivel, que tendrá un mayor tamaño, y así sucesivamente hasta encontrar lo que se busca en algún nivel de cache, o se deba acceder a la memoria interna.

A.4. *Assembler* en linea y extendido

El compilador GCC ofrece una combinacion de características que permiten ocupar el lenguaje *assembly* dentro de *scripts* de otro lenguaje.
El primero es *assembler* en linea, que consiste en un bloque de código *assembly*, en C existen dos formas de llamarlo: *asm()* y *__asm__()*.

Por otra parte, *assembler* extendido es una herramienta/ sintaxis del compilador GCC que permite utilizar variables de C directamente sobre código *assembler* en linea [22], utilizando la siguiente sintaxis:

```
asm asm-qualifiers ( AssemblerTemplate
                     : OutputOperands
                     : InputOperands
                     : Clobbers
                     : GotoLabels)
```

Figura A.1: Sintaxis de *assembler* extendido

Cuyas partes se explican a continuación:

- *Qualifiers*: calificativos que se utilizan para informar al compilador que efectos puede tener el código en *assembly* y como debiese tratarlo.
- *AssemblerTemplate*: Es una lista de *strings* que contiene el código *assembly* que se ejecutarán, mas los símbolos que representan los *inputs*, *outputs* y *gotos*. Los detalles se explicarán mas adelante.
- *OutputOperands*: Son las variables de C que serán modificadas por el código, se separan entre sí por comas.
- *InputOperands*: Son las variables C que solo serán leidas desde el código.
- *Clobbers*: Una lista de valores cambiados por el código, pero que estan fuera de los *outputs*.
- *GotoLabels*: Lista de las etiquetas de otras secciones de código *assembly* hacia donde esta sección asm podría saltar.

Para el caso de este proyecto solo se utilizaron los *inputs*, *outputs* para trabajar, ya que cada bloque de *assembly* solo salta a secciones de si mismo, y el principal valor que cambia (y que es una variable) son los contadores de los *loops*.

Para incluir las variables de C en las listas se deben llamar como %(*número de la variable en las listas*), lo que les asigna un registro dentro del procesador, con el cual se puede trabajar directamente en las instrucciones de *assembly*.

A.5. Optimización en GCC

El compilador GCC ofrece varios niveles de optimización, mencionados en el documento principal, cada una de estas *flags* implica una cantidad de optimizaciones específicas que se ejecutan al ser utilizada, todas con la finalidad de mejorar el rendimiento o el tamaño del programa a cambio del aumento en el tiempo de compilación, la capacidad de *debugging* y la independencia de cada declaración del programa [23].

Apéndice B

Programa utilizado

Durante el texto principal se detalló la sección principal del programa utilizado, que es donde se utiliza la mayor cantidad del tiempo de ejecución total, lo que significa que se omitieron las funciones auxiliares de el, como por ejemplo la función generadora de valores aleatorios, a la que verificaba que el arreglo fuera ordenado correctamente.

Por lo que en las siguientes secciones se muestra el programa completo utilizado, junto a las variables pertinentes utilizando la versión en C de la implementación, aunque también se detallarán la conversión a lenguaje *assembler* de las funciones principales en las que siguen.

B.1. Programa en C

La versión completa del programa en C paralelo utilizado para las mediciones se muestra a continuación, los comentarios incluidos describen el objetivo de cada función, o alguna sección importante de ella.

```
1
2  #include "omp.h"                          // Libreria de OpenMP
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  int arreglo[1000];                        // Arreglo base a ordenar
8  int arreglo1[1000];                       // Arreglo con valores fijos
9  int targ [1000] ;                         // Arreglo objetivo (ordenado)
10
11
12
13 int repeticiones = 1000;                  // repeticiones del proceso
14 long unsigned int ciclos=0;
15
16 void main(void);
```

```
17 int main_main(void);
18 int Revisar(void);
19 void Printer(void);
20 void ArregloReset(void);
21 void longCount(void);
22 void Randomizer(int);
23 void OrdenarX(int);
24 void GenerarTarg(int);
25 int Worst_maker(int);
26
27 int a=0;
28 int porte;                              // largo del arreglo
29 int i;
30 int j;
31 int startCount;
32 double start, end;                      // valores de tiempos
33 double elapsed;                         // tiempo de ejecucion
34 omp_lock_t my_lock;                     // lock de OpenMP
35
36
37 void main(){
38   omp_init_lock(&my_lock);             // inicializa el lock
39   main_main();
40   return;
41 }
42
43 // Funcion principal del programa
44 int main_main(void) {
45   int x = sizeof(targ)/sizeof(targ[0]);   // calcular largo del arreglo
46   porte = x;
47   GenerarTarg(x);                          // Generar arreglo objetivo
48
49     // En cada iteracion se restaura o aleatoriza el arreglo base
50     // para luego ser ordenado. Esto se hace tantas veces
51     // como indica repeticiones
52   for(int ii=0; ii<repeticiones; ++ii){
53     //Randomizer(porte);                // aleatoriza arreglo
54     Worst_maker(porte);                 // ajusta arreglo al peor caso
55     start = omp_get_wtime();            // se marca el tiempo de partida
56
57     OrdenarX(x);                        // ordena el arreglo
58
59     end = omp_get_wtime();              // marca el tiempo de termino
60     elapsed = (end - start);            // calcula el tiempo transcurrido
61
62
63     int Revision = Revisar();           // revisa el arreglo
64     printf("%d; %d ;%d; %f \n",Revision,x, ii,elapsed);
65     if(Revision!=1){Printer();}         // indica si se ordeno bien
66   }
```

```
67
68    omp_destroy_lock(&my_lock);      // colapsa el lock
69    return 0;                        // retorna
70  }
71
72  // Funcion que revisa si arreglo fue ordenado apropiadamente
73  // compara los valores de arreglo con targ elemento a elemento
74  int Revisar(){
75    int x = sizeof(arreglo)/sizeof(arreglo[0]);
76    int y = 1;
77      for(int i=0; i<x;i++){
78        if(arreglo[i]==targ[i]){
79          y=y*1;}
80        else {
81          y=0;
82        }
83    }
84    return y; // 0 indica algun error al ordenar arreglo
85  }
86
87  // Genera el arreglo targ, correspondiente a los valores
88  // ordenados de menor a mayor
89  void GenerarTarg(int n){
90    for (int i=1;i<n;i++){
91      targ[i] = i;
92    }
93    return;
94  }
95
96  // Imprime arreglo de forma completa
97  void Printer(){
98    omp_set_lock(&my_lock);
99    int x = sizeof(arreglo)/sizeof(arreglo[0]);
100   for(int i=0; i<x;i++){
101     printf("%d ", arreglo[i]);
102   }
103   printf("\n");
104   omp_unset_lock(&my_lock);
105 }
106
107 // Copia arreglo1 sobre arreglo
108 void ArregloReset(){
109   a=0;
110   int x = sizeof(arreglo)/sizeof(arreglo[0]);
111   for(int i=0; i<x;i++){
112       arreglo[i]= arreglo1[i];
113   }
114 }
115
116 // Convierte arreglo en el peor caso para su largo
```

```
117 int Worst_maker(int nn){
118   int xx = 0;
119   omp_set_lock(&my_lock);
120   for(int jaja=(nn-1); jaja>=0;jaja--){
121     arreglo[xx] = jaja;
122     xx++;
123   }
124   omp_unset_lock(&my_lock);
125   return 0;
126 }
127
128 // Aleatoriza los valores de arreglo
129 void Randomizer(int n){
130   int max = n-1;
131   int x;
132   int indice;
133
134   for(int i=0; i<n;i++){
135       arreglo[i]= targ[i];
136   }
137
138   while(max>0){
139     indice = rand() %(max + 1-0) + 0;
140     x = arreglo[indice];
141     arreglo[indice] = arreglo[max];
142     arreglo[max] = x;
143     max--;
144 }}
145
146
147 // Funcion que ordena arreglo
148 void OrdenarX(int n){
149   a=0;
150   i=1;
151   int x;
152   int num_harts;
153   int arr [10][n];                              // matriz de arreglos unicos
      para cada hart
154   int vals[10] = {0,0,0,0,0,0,0,0,0,0}  ;    // arreglo con los valores a
      ordenar
155   int auxs[11] = {0,0,0,0,0,0,0,0,0,0,0}  ; // arreglo con los indices de cada
       hart
156
157   // Inicio seccion paralela
158   #pragma omp parallel num_threads(5) firstprivate(i) private(j)
159   {
160     num_harts = omp_get_num_threads();     // obtiene numero de harts
161     int ID = omp_get_thread_num();         // obtiene ID del hart
162     for (i,j; i < (n-(num_harts-1)); i+=num_harts) {
163
```

```
164       #pragma omp barrier
165
166       // Ordena los siguietes valores con Insertion sort
167       if(ID==0){
168         for(int m =0;m<num_harts;m++){
169           vals[m]=arreglo[i+m];
170         }
171
172         for (int i= 1,j; i < num_harts; i++) {
173           int x = vals[i];
174           for (j = i; j > 0 && vals[j-1] > x; j--) {
175             vals[j] = vals[j-1];
176           }
177           vals[j] = x;
178         }
179         auxs[num_harts]=i;
180         omp_set_lock(&my_lock);
181         omp_unset_lock(&my_lock);
182       }
183
184       #pragma omp barrier
185
186       // Busca posiciones de los nuevos valores en el arreglo
187       for(j=i;j>0 && arreglo[j-1]>vals[ID];j--){
188         arr[ID][i-j]=arreglo[j-1];
189       }
190       arr[ID][i-j] = vals[ID];
191       auxs[ID]=j;
192
193       #pragma omp barrier
194       int m=0;
195
196       // Sobrescribe el arreglo de base con el nuevo orden
197       for(m=0;m<(auxs[ID+1]-auxs[ID]+1);m++){
198         arreglo[auxs[ID]+m+ID] = arr[ID][i-auxs[ID]-m];
199       }
200
201       #pragma omp barrier
202
203     }
204   }
205
206   //Ordena cualquier elemento restante secuencialmente
207   x = n%num_harts;
208   if (x!=1){
209     for(i;i<n;i++){
210       x=arreglo[i];
211       for (j = i; j > 0 && arreglo[j-1] > x; j--) {
212         arreglo[j] = arreglo[j-1];
213       }
```

```
214         arreglo[j] = x;
215       }}
216 }
```

B.2. Conversión a *assembly*

Como fue mencionado anteriormente, solo se reemplazaron por *assembler* las secciones más importantes del programa, por lo que a continuación se muestra como se reemplazó cada *loop* y directiva. Por claridad de va a incluir el llamado a *assembler* extendido tal como se utiliza dentro del programa.

Cabe recordar que estos reemplazos ocurren dentro de la sección paralela del programa, por lo que las variables j, ID y m son locales, mientras que los arreglos son compartidos en la memoria, y que la declaración *volatile* se utiliza para que el compilador no modifique la sección durante la optimización.

B.2.1. Primer *loop*

La primera mitad de la iteración consiste en que cada procesador realice su *insertion sort* localmente.

En C corresponde a:

```
1     for(j=i;j>0 && arreglo[j-1]>vals[ID];j--){
2       arr[ID][i-j]=arreglo[j-1];
3     }
4     arr[ID][i-j] = vals[ID];
5     auxs[ID]=j;
```

Mientras que su conversión a *assembly* es:

```
1   asm volatile(
2     "add %0,x0,%1;"          // j = i
3     "li t0,4;"               // t0 = 4
4     "mul t1,t0,%2;"          // t1 = ID*4
5     "add t1,t1,%5;"          // t1 = *vals+ID
6     "lw t3,0(t1);"           // t3 = vals[ID]
7
8     "Inicio:;"
9       "mul t1, t0,%0;"         // t1 = 4*j
10      "add t1, t1,%4;"         // t1 = *arreglo + j
11      "lw t2, -4(t1);"         // t2 = arreglo[j-1]
12      "ble t2, t3, Fin;"       // arreglo[j-1] <= vals[ID] ---> Fin
13      "blez %0, Fin;"          // j=<0 ---> Fin
14
15      "mul t1,%2,%3;"          // t1 = ID*n
16      "sub t4,%1,%0;"          // t4 = i-j
17      "add t4,t1,t4;"          // t4 = ID*n + (i-j)
```

```
      "mul t1,t0,t4;"          // t1 = (ID*n+i-j)*4
      "add t1,t1,%6;"          // t1 = *arr + (ID*n+i-j)*4 = *arr[ID][i-j]
      "sw t2, 0(t1);"          // arr[ID][i-j] = arreglo[j-1]
      "addi %0,%0,-1;"         // j--
      "j Inicio;"

    "Fin: ;"
      "mul t1,%2,%3;"          // t1 = ID*n
      "sub t4,%1,%0;"          // t4 = i-j
      "add t4,t1,t4;"          // t4 = ID*n + (i-j)
      "mul t1,t0,t4;"          // t1 = (ID*n+i-j)*4
      "add t1,t1,%6;"          // t1 = *arr + (ID*n+i-j)*4 =*arr[ID][i-j]
      "sw t3,0(t1);"           // arr[ID][i-j] = vals[ID]
      "mul t1,t0,%2;"          // t1 = ID*4
      "add t1,%7,t1;"          // t1 = *auxs + ID*4 = *auxs[ID]
      "sw %0,0(t1);"           // auxs[ID] = j

  : "+rm" (j)
  : "r" (i), "r" (ID), "r"(n), "r" (ptr_arreglo), "r" (ptr_vals),
    "r" (ptr_arr), "r" (ptr_auxs)
);
```

B.2.2. Segundo *loop*

Este es el *loop* encargado de actualizar el arreglo original con el nuevo orden de elementos, basados en los resultados locales del *loop* anterior.

```
    int m=0;
    for(m=0;m<(auxs[ID+1]-auxs[ID]+1);m++){
    arreglo[auxs[ID]+m+ID] = arr[ID][i-auxs[ID]-m];
  }
```

Que equivale a:

```
  asm volatile(
    "li t0, 4;"          // t0 = 4
    "li %0, 0;"          // m = 0

    "Inicio2:;"
      "mul t2,t0,%2;"      // t2 = ID*4
      "add t3,%7,t2;"      // t3 = *auxs[ID] = *auxs + ID*4
      "lw t4,4(t3);"       // t4 = auxs[ID+1]
      "lw t5,0(t3);"       // t5 = auxs[ID]
      "sub t4,t4,t5;"      // t4 = auxs[ID+1] - auxs[ID]
      "addi t4,t4,1;"      // t4 = auxs[ID+1] - auxs[ID] +1
      "bge %0,t4,Fin2;"    // m >= auxs[ID+1] - auxs[ID] +1 ---> Fin

      "add t4,t5,%0;"      // t4 = auxs[ID] + m
      "add t4,t4,%2;"      // t4 = auxs[ID] + m + ID
      "mul t4,t4,t0;"      // t4 = (auxs[ID] + m + ID)*4
      "add t4,t4,%4;"      // t4 = *arreglo[auxs[ID]+m+ID]
      "mul t1,%2,%3;"      // t1 = ID*n
      "sub t2,%1,t5;"      // t2 = i - aux[ID]
      "sub t2,t2,%0;"      // t2 = i - aux[ID] -m
```

```
      "add t1,t1,t2;"      // t1 = ID*n + (i - aux[ID] -m)
      "mul t1,t1,t0;"      // t1 = (ID*n + (i - aux[ID] -m))*4
      "add t1,t1,%6;"      // t1 = * arr[ID][i-auxs[ID]-m]
      "lw t1,0(t1);"       // t1 = arr[ID][i-auxs[ID]-m]
      "sw t1,0(t4);"       // arreglo[auxs[ID]+m+ID] = arr[ID][i-auxs[ID]-m]
      "addi %0,%0,1;"      // m++
      "j Inicio2;"
    "Fin2: ;"
    : "+rm" (m)
    :  "r" (i), "r" (ID), "r"(n), "r" (ptr_arreglo), "r" (ptr_vals),
       "r" (ptr_arr), "r" (ptr_auxs)
  );
```

B.2.3. Barreras

En el caso particular de las barreras, más que reemplazar una función propia del programa, se está reemplazando la directiva *barrier* de *OpenMP* con el objetivo de reducir los tiempos de *overhead* necesarios cada vez que se requería sincronizar las CPU. Por lo que solo se está reemplazando siguiente línea de código.

```
    #pragma omp barrier
```

Como a diferencia de las secciones anteriores no es obvio lo que se está reemplazando, a continuación se explicará la lógica de las nuevas barreras:

- Cada barrera tiene un contador propio de ellas (ubicado dentro del arreglo *barrierVal*).
- Al llegar a la barrera cada *thread* incrementa atómicamente (mediante *locks*) el valor del contador respectivo en 1.
- Los *threads* esperan a que el contador alcance un valor igual al número de *threads* involucrados, indicativo de que todos han alcanzado la barrera en particular.
- Una vez que todos los *threads* están en la barrera se continúa con la ejecución del programa.

Lo que se implementa como:

```
  asm volatile(
    "li t0,1;"                      // t0 = 1
    "spinlock_acq_2:;"
    "amoswap.w.aq t1,t0,(%0);"      // adquirir lock
    "bnez  t1,spinlock_acq_2;"      // reintentar si fallo

    "lw t2,4(%1);"                  // t2 = barrierVal[1] (la barrera de
                                       esta barrera particular)
    "addi t2,t2,1;"                 // barrierVal[1] ++
    "sw t2,4(%1);"                  // Actualizar barrierVal[1] en la
                                        memoria
    "amoswap.w.rl x0,x0,(%0);"      // liberar lock

      "barrierWait_2: ;"
    "lw t2,4(%1);"                  // t2 = barrierVal[1]
    "blt t2,%2,barrierWait_2;"      // repetir si no estan todos
```

```
15
16      :"+rm" (pspinlock)
17      : "r" (pbarrierVal), "r" (num_harts)
18    );
```

Apéndice C

Instrucciones RISC-V

La documentación de RISC-V divide la arquitectura en 2 categorías: privilegiada y no privilegiada [24, p.2]. La primera siendo la orientada hacia las funciones privilegiadas (aquellas que pueden modificar el *kernel*), funcionamiento de sistemas operativos y componentes periféricos, mientras que la segunda categoría esta orientada al resto de funciones, las que pueden ser ejecutadas por cualquier usuario de forma segura. Esta separación existe para generar distintos niveles de acceso a los recursos de *software* y protegerlos.

En las siguientes secciones se muestra en mayor detalle las instrucciones de los módulos relevantes para el proyecto, los que caen en la categoría de arquitectura no privilegiada, asi como la estructura general de las instrucciones en RISC-V.

C.1. Formato base de instrucciones

Las instrucciones estándar de RISC-V tienen un tamaño uniforme de 32 bits, pero según su funciones son divididas en 6 grande categorías:

- R: operaciones entre registros
- I: operaciones con inmediatos cortos y *loads*
- S: operaciones de *store*
- B: operaciones con *branches* (saltos condicionales)
- U: operaciones con inmediatos largos
- J: saltos incondicionales

Estas categorías siguen los siguientes formatos [1, p.17]:

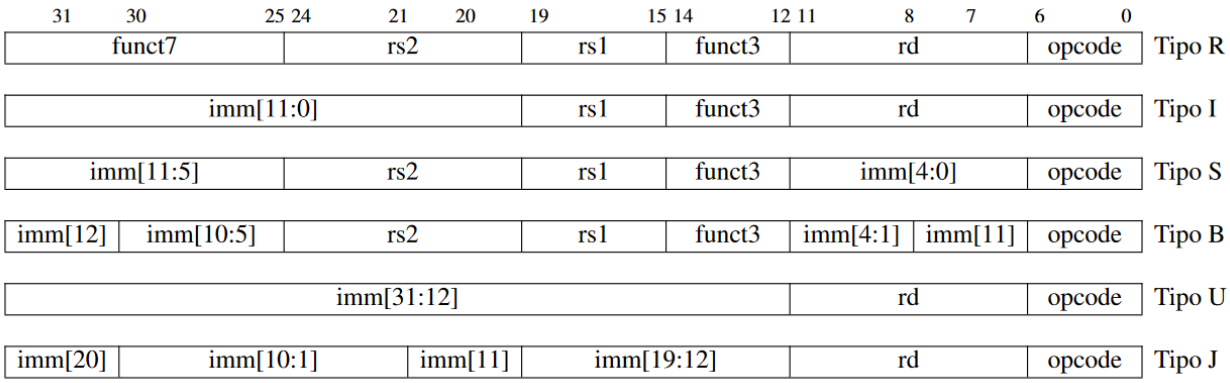

Figura C.1: Sintaxis de instrucciones en RISC-V. Los valores en la parte superior indican la posición de bits

Donde las funciones se separan en distintas secciones:

- rsX: registro fuente
- rd: registro de destino
- functX: seleccionan el tipo de operación
- imm[X]: posición del bit X del valor inmediato
- opcode: bits que especifican la función a ocupar

La única excepción a esta regla son las operaciones *control status register* (CSR), que utilizan una variante del tipo I.

C.2. Módulos estándares comunes de RISC-V

C.2.1. Módulo I

Este es el modulo básico de operaciones con enteros, por lo que contiene una diversa cantidad de operaciones necesarias para funcionar, como lo son operaciones de memoria, saltos condicionales e incondicionales. El detalle de las instrucciones se muestra en la imagen C.2, donde las letras subrayadas son las que componen el llamado de la función, mientras que se utilizan para mostrar distintas variantes de ella. Por ejemplo, para llamar a la función *load byte unsigned* se utiliza *lbu*.

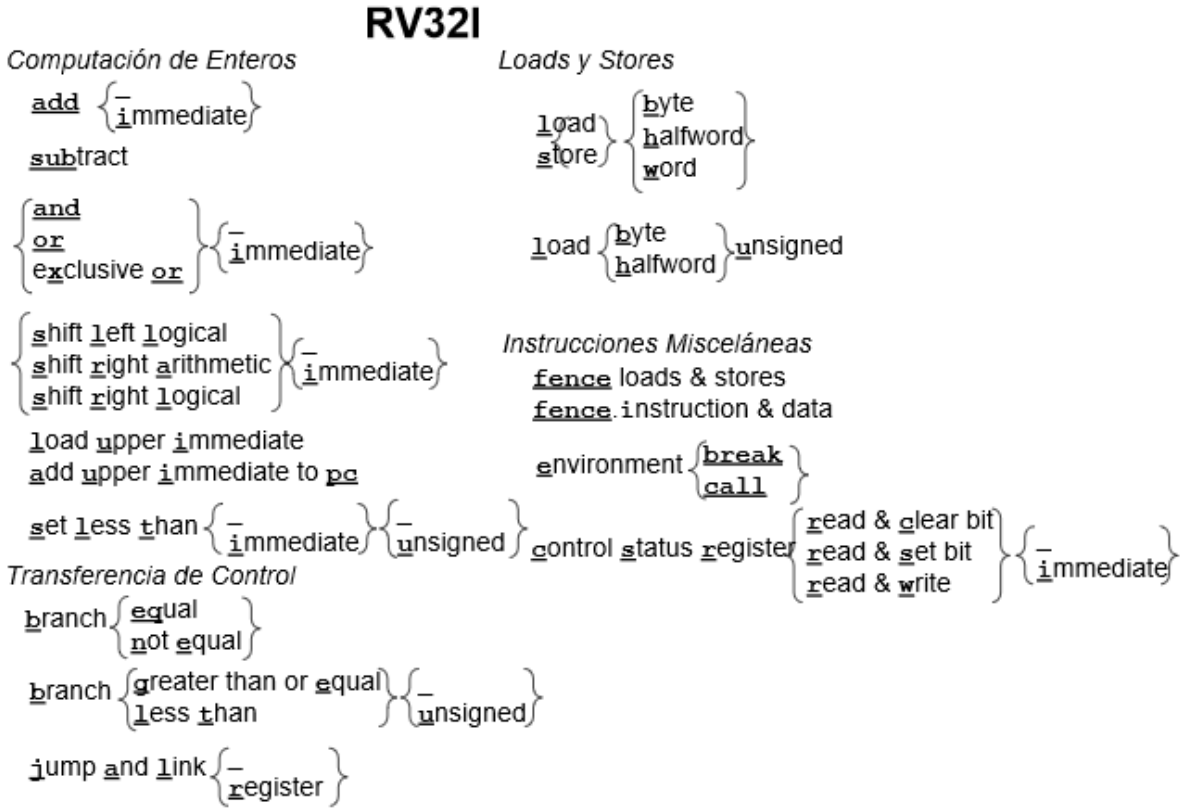

Figura C.2: Instrucciones del módulo I, en particular de RV32I, la variante más básica [1, p.17]

C.2.2. Módulo M

Módulo especifico de multiplicación y división,

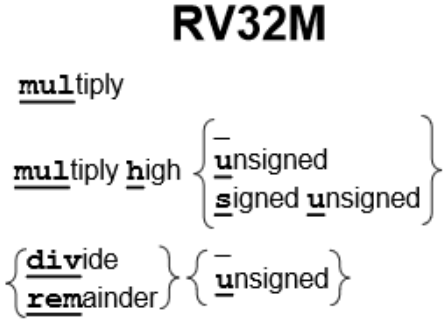

Figura C.3: Instrucciones del módulo M [1, p.48]

C.2.3. Módulo A

Módulo dedicado a las operaciones atómicas entre *harts*, y que entregan soporte al modelo de consistencia de memoria RC_{sc}

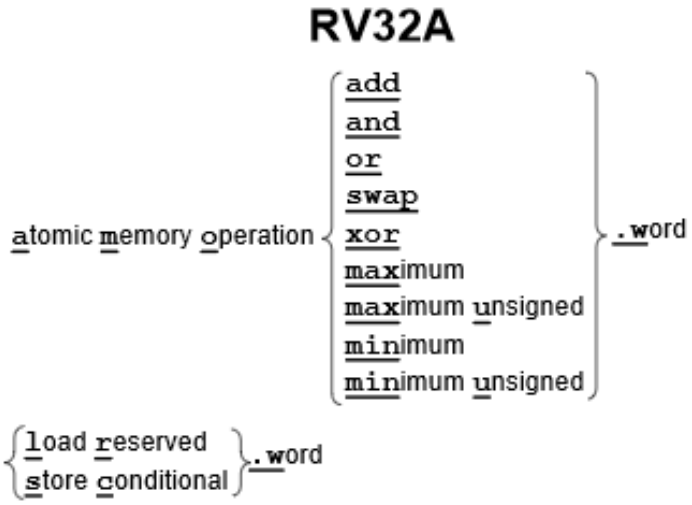

Figura C.4: Instrucciones del módulo A [1, p.64]

C.2.4. Módulos F y D

Ambos módulos entregan funcionalidades con punto flotante, variando entre la precisión simple y doble respectivamente, siguiendo el formato IEEE 754-2008.
Estos módulos utilizan un set de registros específicos llamados *f*, en vez de los registros estándar *x*. Si bien ambos estos registros no son intercambiables estos módulos presentan funciones capaces de transferir valores entre ambos.

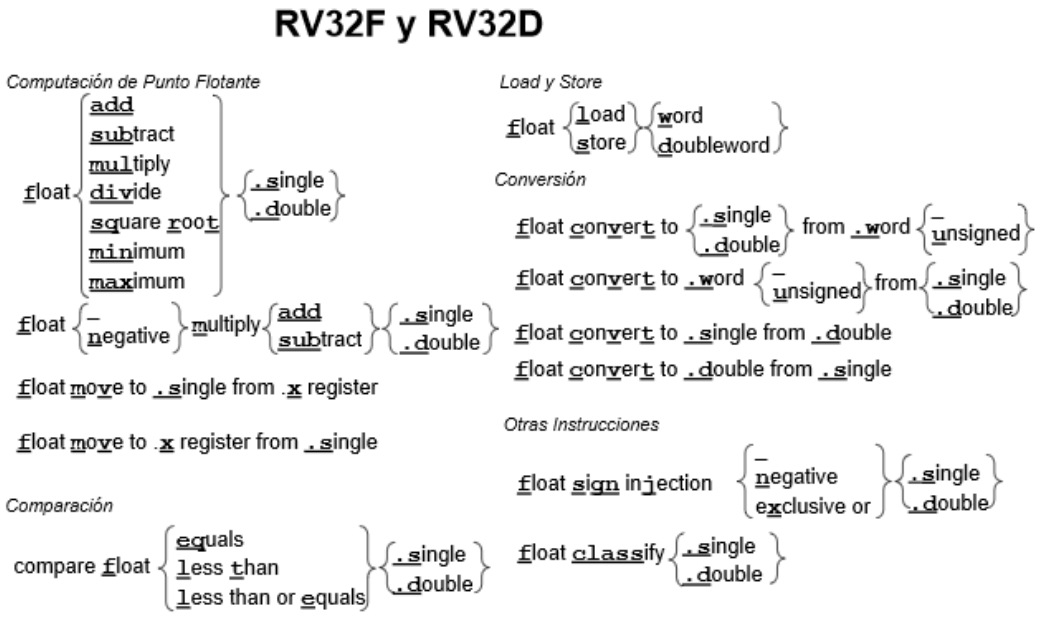

Figura C.5: Instrucciones de los módulos F y D [1, p.53]

C.2.5. Módulo C

Este módulo introduce funciones compactas de 16 bits, las que son asociadas a una única función de 32 bits por parte del ensamblador y el *linker*

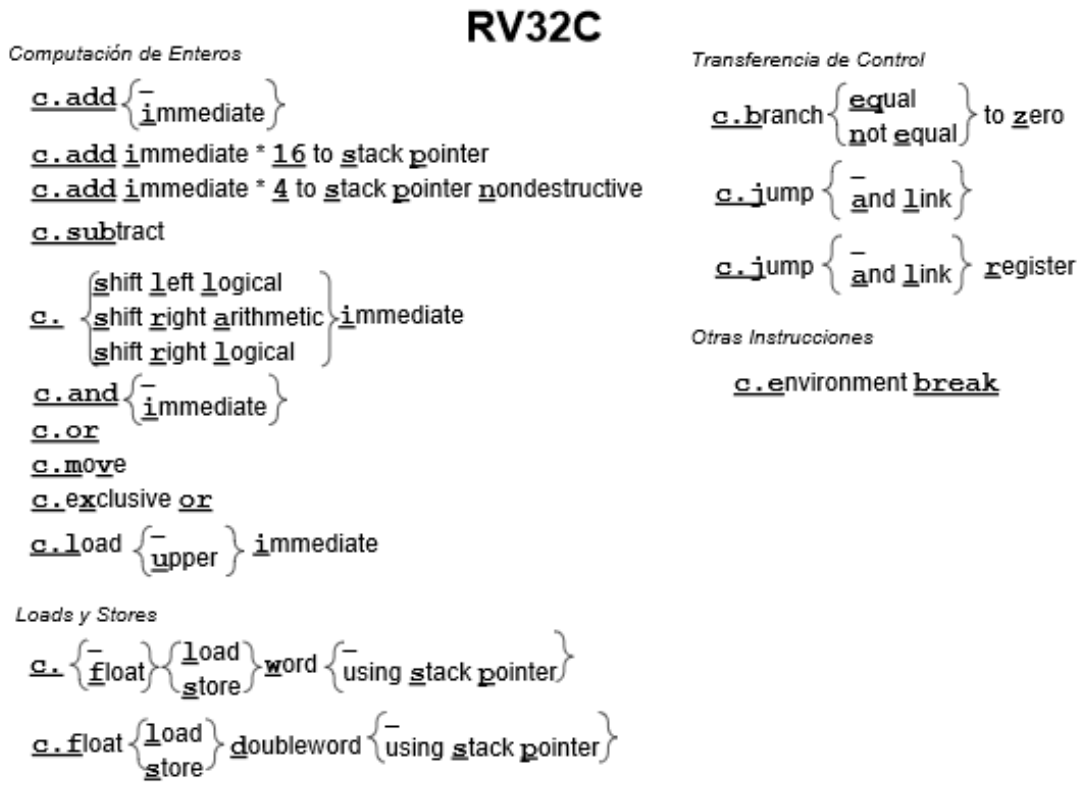

Figura C.6: Instrucciones del módulo C [1, p.69]