# High Performance Computing
# Departamento de Ingeniería en Informática
# LAB2 : SIMD-vectorial + SIMD-hebras

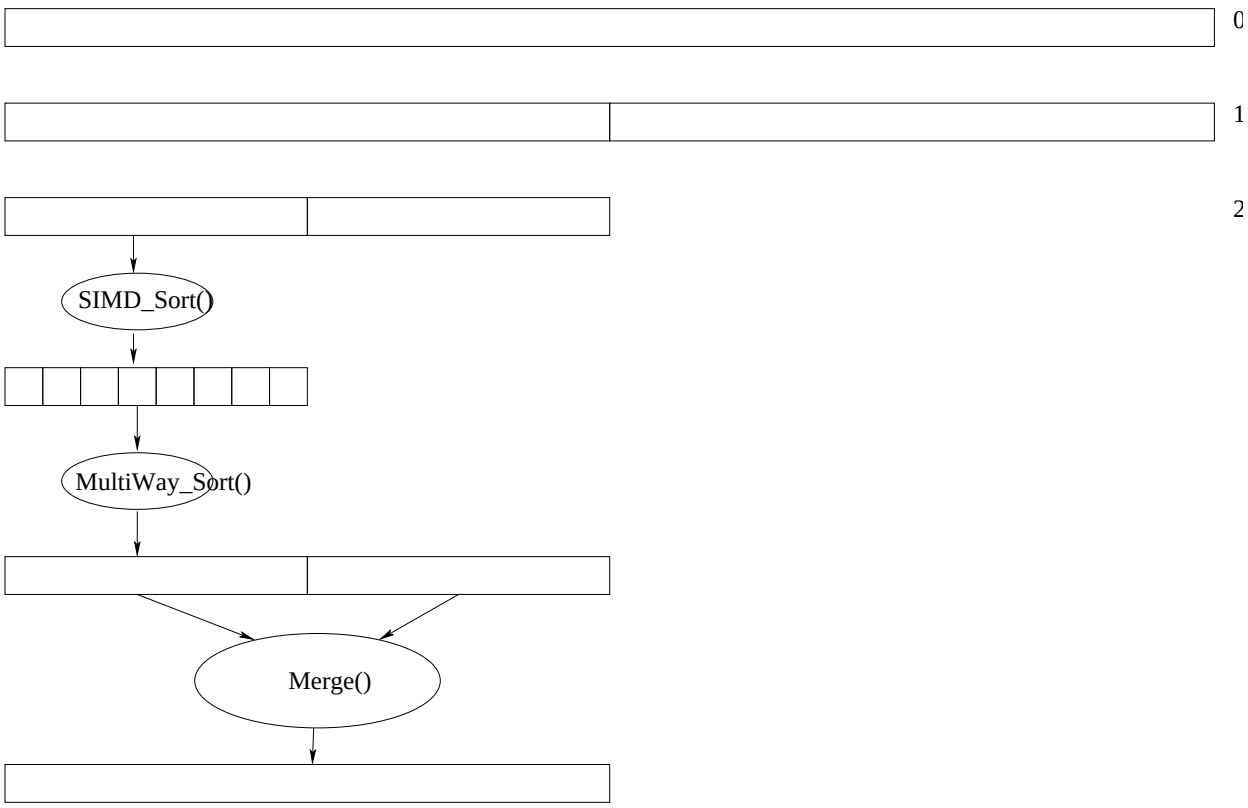
## 1 Objetivo

El objetivo de este laboratorio es usar dos tecnologías de programación paralela para ordenar una lista de números. Específicamente, usaremos SIMD vectorial (MMX) junto con SIMD con OpenMP, para implementar ordenamiento de números.

## 2 Algoritmo general

La siguiente figura describe los pasos más importantes del algoritmo, el cual utiliza las funciones `SIMD_Sort()` `Multiway_Sort()` y `Merge()`.

1. Consideramos que el arreglo de entrada se encuentra en el nivel 0 del árbol de recursividad.
2. En cada nivel del árbol, se debe crear tareas OpenMP para ordenar los dos subarreglos correspondientes.
3. Este paso se repite hasta que se alcance un nivel máximo de profundidad.
4. En el nivel más profundo, el algoritmo utiliza `SIMD_Sort()` (ordenamiento vectorial) para producir listas ordenadas de 16 elementos.
5. Estas listas son ordenadas usando un multi-way merge `Multiway_Sort()` basada en una cola de prioridad.
6. Luego, en el nivel superior del arbol se mezclan las dos listas ordenadas con un merge simple (2-way). Note que este trabajo es realizado por la tarea que creo las dos tareas hijas.
7. El algoritmo continúa hasta retornar al nivel 0 con la lista ordenada.




## 2.1 El Heap binario

Un **Heap** es una cola de prioridad tal que siempre el elemento del frente de la cola tiene la prioridad máxima (por ejemplo el valor menor). Un **heap binario** es un heap implementado como un árbol binario, que satisface:

1. Es un árbol completo, es decir todos lo niveles del árbol, excepto posiblemente el último, están completos.

2. Cada nodo del árbol contiene un valor menor o igual (mayor o igual) al de sus hijos.

Por ejemplo, la Figura 1 muestra un heap binario con números enteros. Note que ningún recorrido del árbol puede producir una secuencia ordenada. Sin embargo, por la propiedad de heap, la raíz del árbol siempre contiene el valor menor de todos los nodos.

Un heap tiene dos métodos asociados:

1. `Insert()`: que inserta un nuevo elemento en la cola

2. `Delete()`: que saca el elemento menor de la cola, es decir el de la raíz.

Ambas operaciones deben garantizar que las dos propiedades de un heap binario se mantengan.

## 2.2 Insert

Esta operación consiste de dos pasos, primero de la inserción del elemento "al final del árbol", y luego un *heapify-up* que mueve el elemento recién insertado a una posición del árbol tal que se satisfaga la propiedad de heap. Estos pasos son:

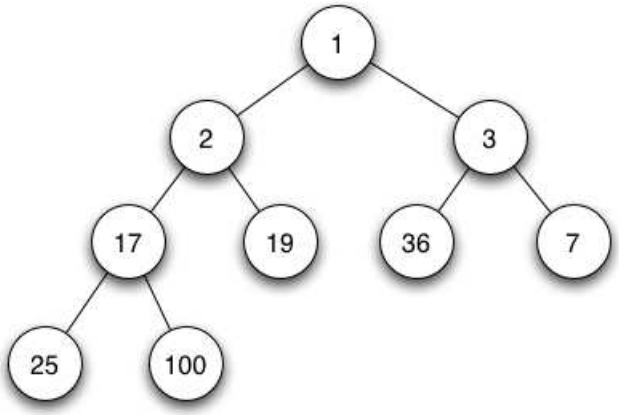



Figure 1: Heap binario.

1. Insertar el elemento al final del árbol.
2. Comparar el elemento con su padre. Si el padre es menor o igual que el hijo, el árbol es un heap y termina el heapfiy-up.
3. Si es menor que el padre, se intercambian las posiciones entre padre e hijo.
4. Se continúa este procedimiento hasta que la operación termine o se alcance la raíz.

### 2.3 Delete

Con la propiedad de heap, se garantiza que siempre en la raíz está el elemento menor del heap. Cuando este elemento es removido del heap, se debe realizar un *heapify-down* para que el elemento menor del árbol suba hasta la raíz. EL procedimiento es el siguiente:

1. Eliminar (y retornar) el elemento de la raíz
2. Poner en la raíz el último elemento del heap
3. Comparar la raíz con sus dos hijos; si el padre es menor que ambos, el heapify-down termina.
4. Si no, intercambiar el padre con el menor de sus dos hijos.
5. Se continúa este procedimiento hasta que la operación termine o se alcance una hoja del árbol.

## 3 Heap-based Multiway Merge Sort

El heap binario será utilizado para el multiway merge sort. Una vez que una hebra haya finalizado de formar secuencias ordenadas de 16 elementos, generará la secuencia ordenada usando un heap binario. Inicialmente, debe inicializar el heap insertando un elemento de cada lista. Luego, cuando saque el elemento menor del heap, debe insertar un nuevo elemento al heap, de la lista a la que pertenecía el

elemento que recien salió del heap. **Este elemento se inserta en la raíz del árbol, seguido de un *heapify-down***. De esta forma, intentamos mantener las listas aproximadamente del mismo largo durante el merge. Note que a medida que se crea la secuencia ordenada, es posible que las sublistas se vacíen. Usted, debe preocuparse de considerar estos caso de borde.

# 4 Los programas y la lista

Usted debe implementar este programa en lenguaje C, sobre sistema operativo Linux. El programa debe ejecutarse de la siguiente forma:

```
$ ./sort -i desordenada.raw -o ordenada.raw -N num_elementos -d debug -l nivel_arbol -h num_he
```

donde las opciones indican lo siguiente:

- `-i`: archivo binario con la lista de entrada desordenados
- `-o`: archivo binario de salida con la lista ordenada
- `-N`: largo de la lista
- `-l`: número de niveles de recursividad
- `-h`: número de hebras

Para este programa, usaremos siempre listas con números flotantes (32 bits) y de largo $2^4 \times 2^n$. Los números en los archivos de entrada y de salida deben estar almacenados en formato binario.

# 5 Mediciones

Ejecute su programa y mida el tiempo de ejecución y speed-up con 2, 3, 4, y 5 niveles de recursividad y varios número de hebras.

# 6 Entregables

Tarree, comprima y envíe a `fernando.rannou@usach.cl` al menos los siguientes archivos:

1. `Makefile`: archivo para make que compila los programas
2. `sort.c`: archivo con el código. Puede incluir otros archivos fuentes. algoritmos.
3. `mediciones.pdf`: archivo con gráficos tiempo de ejecución y speedup, indicando el largo del arreglo.

# Fecha de entrega (hard-deadline): Viernes 22 de noviembre antes de las 24:00 hrs.