

**BÚSQUEDA DISTRIBUIDA DE NÚMEROS PERFECTOS USANDO ICE Y EL MODELO
MAESTRO-CLIENTE-TRABAJADORES**

- Daniel Esteban Arcos Cerón A00400760
- Hideki Tamura Hernández A00348618
- Luna Catalina Martínez Vásquez A00401964
- Renzo Fernando Mosquera Daza A00401681

1. DESCRIPCIÓN DEL PROBLEMA: El presente proyecto tiene como finalidad optimizar la búsqueda de **números perfectos** dentro de un rango determinado, mediante la implementación de un sistema distribuido que permita repartir eficientemente la carga de procesamiento entre múltiples nodos de cómputo. En este contexto, se define un número perfecto como aquel que es igual a la suma de sus divisores propios positivos, excluyéndose a sí mismo. Por ejemplo, el número **28** cumple esta condición, ya que **1 + 2 + 4 + 7 + 14 = 28**.

Desde el punto de vista computacional, la verificación de números perfectos representa una tarea costosa, ya que requiere evaluar todos los posibles divisores de cada número dentro del rango. Esta operación presenta una complejidad de orden $O(n\sqrt{n})$, lo que implica un crecimiento significativo del tiempo de ejecución a medida que se incrementa el tamaño del intervalo.

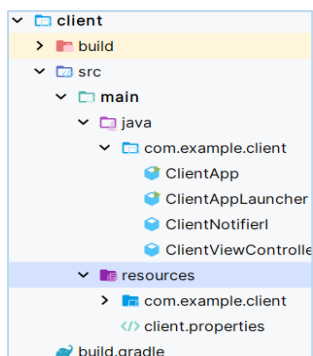
Para abordar este desafío, el proyecto propone un enfoque basado en paralelización, dividiendo el trabajo entre múltiples nodos que operan de manera concurrente. Esta estrategia permite distribuir la carga computacional, reducir los tiempos de espera y escalar el procesamiento de forma más eficiente.

En este contexto, se adopta una arquitectura distribuida del tipo **Cliente-Maestro-Trabajadores**, implementada utilizando la plataforma **ICE** (Internet Communications Engine). Donde ICE proporciona un marco robusto para la comunicación entre procesos distribuidos, facilitando la definición de interfaces mediante el lenguaje **Slice** y permitiendo la ejecución de llamadas remotas **asíncronas**.

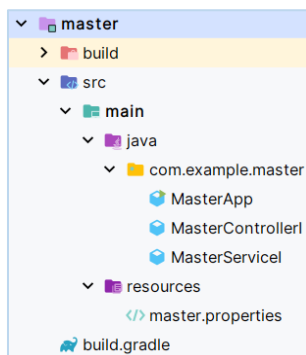
2. ARQUITECTURA GENERAL DEL SISTEMA DISTRIBUIDO: Esta sección presenta la organización interna del sistema desde un enfoque estructural, describiendo los módulos que lo integran y la manera en que se comunican entre sí para lograr una ejecución distribuida eficiente.

En nuestro caso, el sistema está conformado por **cuatro módulos principales**:

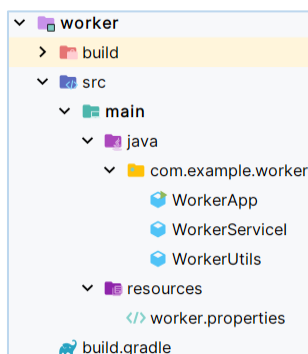
- **Cliente (client/src/...):** Permite al usuario configurar y lanzar solicitudes de búsqueda a través de una interfaz gráfica.



- **Maestro (master/src/...):** Nodo central encargado de dividir el trabajo, coordinar los resultados y controlar el flujo global.



- **Trabajadores (worker/src/...):** Procesos independientes que ejecutan subrangos de búsqueda de forma paralela.

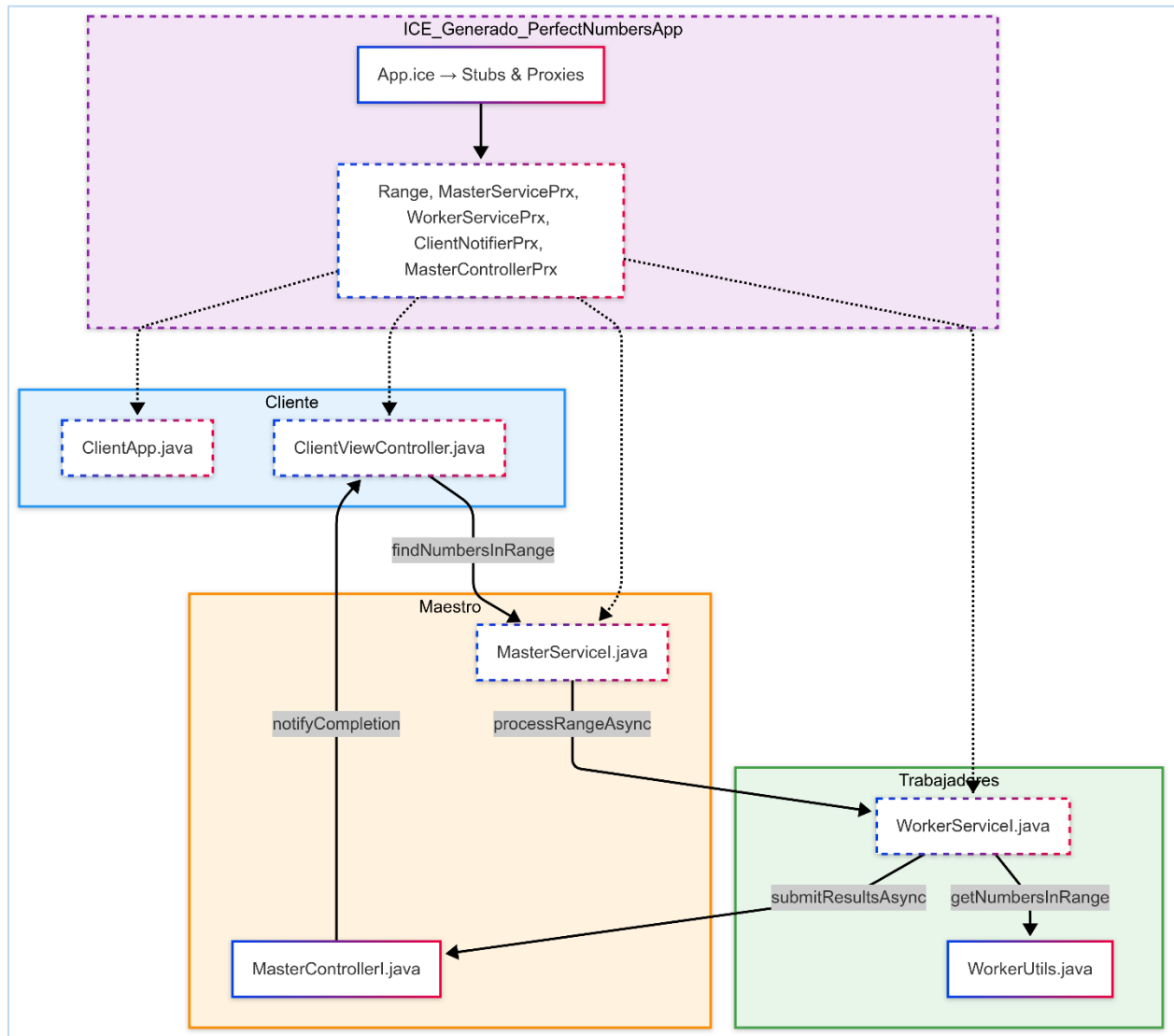


- **Interfaces ICE y estructuras auxiliares (PerfectNumbersApp/build/...):** Este módulo contiene todos los stubs generados automáticamente a partir del archivo **App.ice**.

```
App.ice x
41
42 // Interfaz que el Maestro invoca en cada Worker para procesar un subrango
43 interface WorkerService {
44     // AMD para no bloquear al Maestro
45     // subRangeToProcess: rango a procesar
46     // masterCallbackProxy: proxy al MasterController para enviar resultados
47     // workerId: ID único de la sub-tarea
48     ["amd"] void processSubRange(Range subRangeToProcess,
49                                 MasterController* masterCallbackProxy,
50                                 string workerId);
51 };
52
53 // Interfaz principal del Maestro, usada por el Cliente
54 interface MasterService {
55     // Inicia la búsqueda de perfectos en un rango
56     // jobRange: rango completo a analizar
57     // clientNotifierProxy: proxy al ClientNotifier para devolver resultados
58     // numWorkersToUse: cuántos Workers usar
59     void findPerfectNumbersInRange(Range jobRange,
60                                   ClientNotifier* clientNotifierProxy,
61                                   int numWorkersToUse);
62
63     // Permite a un Worker registrarse con el Maestro
64     void registerWorker(WorkerService* workerProxy);
65
66     // Consulta el número de Workers activos (responden a ping)
67     int getActiveWorkerCount();
68 };
69
```

Cada módulo cumple una función específica y se comunica a través de interfaces generadas a partir del archivo **App.ice**. Aunque este se encuentra en la raíz del proyecto, su compilación se gestiona desde el módulo **PerfectNumbersApp**, encargado de generar los **proxies** y **stubs** mediante **slice2java**. Esto permite una integración fluida entre componentes y asegura una comunicación remota coherente y eficiente.

En este sentido, el siguiente de bloques permite visualizar los componentes y el flujo general del sistema:



1. El **cliente** recibe del usuario el rango de búsqueda y el número de trabajadores deseado.
2. El **maestro** divide dicho rango en subrangos y los asigna a los trabajadores disponibles.
3. Cada **trabajador** procesa su subrango de forma independiente y reporta los resultados encontrados.
4. El **maestro** consolida los resultados y se los notifica al **cliente**.

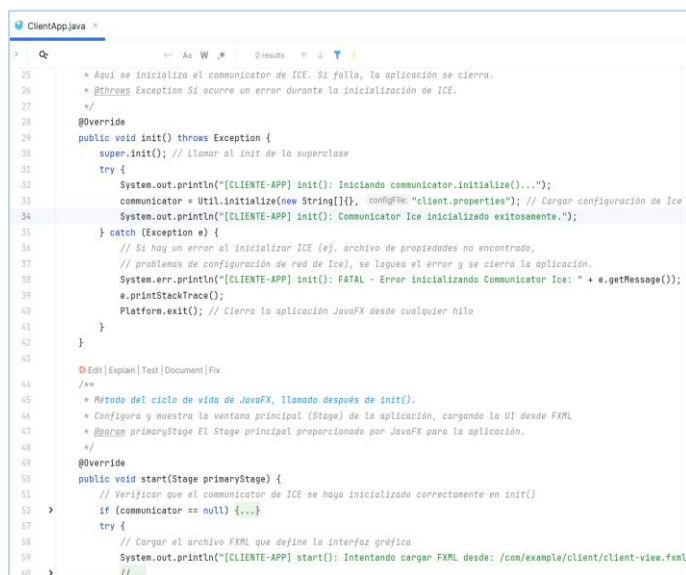
Este diseño modular favorece la escalabilidad, ya que nuevos trabajadores pueden integrarse sin modificar el cliente ni el maestro. De hecho, como la comunicación entre componentes se realiza de manera asíncrona, se permite una ejecución concurrente sin bloqueos entre procesos.

3. DETALLE DEL DISEÑO CLIENTE-MAESTRO-TRABAJADORES CON ICE: En nuestra implementación, la interacción entre los componentes se articula mediante interfaces distribuidas definidas en el archivo **App.ice**, ubicado en la raíz del proyecto. Este archivo describe cuatro interfaces clave que estructuran la comunicación remota:

- **MasterService:** Permite al cliente enviar solicitudes al maestro.
- **ClientNotifier:** Usada por el maestro para notificar al cliente una vez finalizado el procesamiento.
- **WorkerService:** Define la lógica que los trabajadores deben implementar para recibir y ejecutar tareas.
- **MasterController:** Funciona como **callback** para que los trabajadores envíen sus resultados al maestro.

CLIENTE: El cliente, cuya lógica principal se encuentra en **ClientApp.java**, presenta una interfaz gráfica (**JavaFX**) que permite al usuario especificar el rango numérico y el número de trabajadores. Internamente:

- Establece una conexión con el **maestro** mediante el proxy **MasterServicePrx**.
- Publica un adaptador (**ClientNotifierI.java**) para recibir los resultados de manera asíncrona.
- Configura la lógica de envío de solicitud y recepción de resultados en **ClientViewController.java**.



```

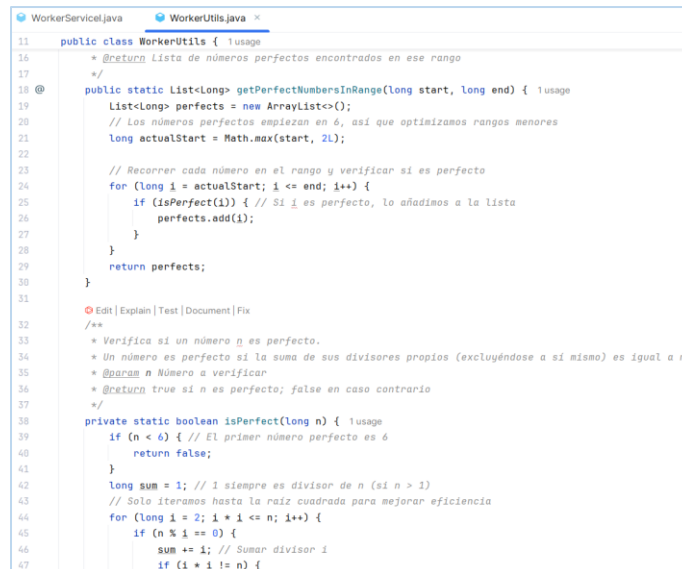
25  * Aquí se inicializa el comunicador de ICE. Si falla, la aplicación se cierra.
26  * @throws Exception Si ocurre un error durante la inicialización de ICE.
27  */
28  @Override
29  public void init() throws Exception {
30      super.init(); // Llamar al init de la superclase
31      try {
32          System.out.println("[CLIENTE-APP] init(): Inicializando comunicador.initialize(...)");
33          communicator = Util.initialize(new String[]{}); // Cargar configuración de Ice
34          System.out.println("[CLIENTE-APP] init(): Comunicador Ice inicializado exitosamente.");
35      } catch (Exception e) {
36          // Si hay un error al inicializar ICE (ej. archivo de propiedades no encontrado,
37          // problemas de configuración de red de Ice), se loguea el error y se cierra la aplicación.
38          System.err.println("[CLIENTE-APP] init(): FATAL - Error inicializando Comunicador Ice: " + e.getMessage());
39          e.printStackTrace();
40          Platform.exit(); // Cierra la aplicación JavaFX desde cualquier hilo
41      }
42  }
43
44  /**
45   * Método del ciclo de vida de JavaFX, llamado después de init().
46   * Configura y muestra la ventana principal (Stage) de la aplicación, cargando la UI desde FXML
47   * @param primaryStage El Stage principal proporcionado por JavaFX para la aplicación.
48   */
49  @Override
50  public void start(Stage primaryStage) {
51      // Verificar que el comunicador de ICE se haya inicializado correctamente en init()
52      if (communicator == null) {
53          try {
54              // Cargar el archivo FXML que define la interfaz gráfica
55              System.out.println("[CLIENTE-APP] start(): Intentando cargar FXML desde: /com/example/client/client-view.fxml");
56          }
57      }
58  }

```

Gracias a este diseño, el cliente puede iniciar el flujo de procesamiento sin depender de respuestas sincrónicas, lo que garantiza una experiencia fluida y no bloqueante.

MAESTRO: El maestro es el nodo coordinador del sistema. Su comportamiento está implementado principalmente en la clase **MasterServiceI.java**, que se encarga de:

- Gestionar las solicitudes del cliente y dividir el rango de búsqueda en subrangos balanceados.
- Invocar de forma asíncrona a los trabajadores a través de **WorkerServicePrx**.
- Registrar trabajadores dinámicamente mediante el método **registerWorker(...)**.
- Publicar un adaptador (**MasterControllerI.java**) que implementa la interfaz **MasterController** y permite consolidar los resultados recibidos de los trabajadores.



```

11 public class WorkerUtils { // usage
12
13     * @return Lista de números perfectos encontrados en ese rango
14     */
15     @
16     public static List<Long> getPerfectNumbersInRange(long start, long end) { // usage
17         List<Long> perfects = new ArrayList<>();
18         // Los números perfectos empiezan en 6, así que optimizamos rangos menores
19         long actualStart = Math.max(start, 2L);
20
21         // Recorrer cada número en el rango y verificar si es perfecto
22         for (Long i = actualStart; i <= end; i++) {
23             if (isPerfect(i)) { // Si i es perfecto, lo añadimos a la lista
24                 perfects.add(i);
25             }
26         }
27         return perfects;
28     }
29 }
30
31
32 /**
33  * Verifica si un número n es perfecto.
34  * Un número es perfecto si la suma de sus divisores propios (excluyéndose a sí mismo) es igual a n.
35  * @param n Número a verificar
36  * @return true si n es perfecto; false en caso contrario
37  */
38 private static boolean isPerfect(long n) { // usage
39     if (n < 6) { // El primer número perfecto es 6
40         return false;
41     }
42     long sum = 1; // 1 siempre es divisor de n (si n > 1)
43     // Solo iteramos hasta la raíz cuadrada para mejorar eficiencia
44     for (Long i = 2; i * i <= n; i++) {
45         if (n % i == 0) {
46             sum += i; // Sumar divisor i
47             if (i * i != n) {

```

Este diseño garantiza que el maestro no mantenga conexiones bloqueantes ni dependencias rígidas con los trabajadores o el cliente.

TRABAJADORES: Cada trabajador implementa la lógica para recibir tareas del maestro y devolver los resultados en **WorkerServiceI.java**, donde:

- Publica su servicio local para ser registrado por el maestro.
- Permanece a la espera de tareas, las cuales le llegan mediante el método **processSubRangeAsync(...)**.
- Una vez asignado un subrango, calcula los números perfectos usando la clase utilitaria **WorkerUtils.java**.
- Reporta los resultados de vuelta al maestro mediante el **callback MasterControllerPrx**.

```

WorkerService.java
WorkerUtils.java

18 public class WorkerService implements WorkerService { 2 usages
19     * Retorna CompletionStage completado cuando termine de procesar y notificar
20     */
21
22     @Override
23     @Usage
24     public CompletionStage<Void> processSubRangeAsync(
25         Range subRangeToProcess,
26         MasterControllerPrx masterCallBackProxy,
27         String workerJobId, // ID específico para esta tarea, asignado por el Maestro
28         Current current) {
29
30         // Mostrar en consola el subrange que se va a procesar
31         System.out.println("[" + workerJobId + "] Recibido subrange: [" + subRangeToProcess.start + ", " + subRangeToProcess.end + "]);
32
33         // Ejecutar el cálculo en un hilo separado para no bloquear el servidor Ice
34         return CompletableFutura.runAsync(() -> {
35             // Medir tiempo de cálculo local
36             long calculationStartTime = System.currentTimeMillis();
37             // Obtener lista de números perfectos en el rango indicado
38             List<Long> foundPerfectNumbersList = WorkerUtils.getPerfectNumbersInRange(
39                 subRangeToProcess.start, subRangeToProcess.end);
40             long calculationEndTime = System.currentTimeMillis();
41             long workerProcessingTimeMillis = calculationEndTime - calculationStartTime;
42
43             // Convertir la lista a un array primitivo para enviar a través de Ice
44             long[] perfectNumbersArray = foundPerfectNumbersList.stream().mapToLong(l -> l).toArray();
45
46             // Mostrar resultados y tiempo de cálculo en consola
47             System.out.println("[" + workerJobId + "] Números encontrados: " + Arrays.toString(perfectNumbersArray) +
48                 ", Tiempo de cálculo ESTE SUBRANGE: " + workerProcessingTimeMillis + " ms.");
49
50             // Si el proxy al Maestro es válido, enviar los resultados
51             if (masterCallBackProxy != null) {
52                 try {
53                     // Enviar resultados al Maestro
54                 } catch (LocalException e) {
55                     // Error de comunicación local
56                 }
57             }
58         });
59     }
60 }

```

Esta arquitectura permite que los trabajadores operen de manera desacoplada: cada uno es independiente y responde únicamente cuando es requerido.

Generación de proxies y estructura de soporte en ICE: A partir del archivo **App.ice**, ICE genera automáticamente todas las clases necesarias para la comunicación distribuida, incluyendo los proxies (***_Prx**), implementaciones base (***_I**) y estructuras auxiliares como **Range**. Todas estas clases se agrupan dentro del módulo **perfectNumbersApp**, generado por el módulo **PerfectNumbersApp**.

Por otro lado, las interfaces están anotadas con **["amd"]** (**Asynchronous Method Dispatch**), lo que permite ejecutar las operaciones de manera completamente asíncrona. Esto resulta fundamental para mantener la concurrencia, escalar con múltiples nodos y evitar cuellos de botella en la ejecución distribuida.

4. EXPLICACIÓN DEL MECANISMO DE DISTRIBUCIÓN DE RANGO Y COORDINACIÓN: Una vez que el cliente envía una solicitud de cálculo, el maestro asume la responsabilidad de distribuir la carga de trabajo entre los trabajadores disponibles y de consolidar los resultados. Este flujo de operación se implementa principalmente en las clases **MasterServiceI.java** y **MasterControllerI.java**.

4.1. DISTRIBUCIÓN DE SUBRANGOS

El proceso se inicia con la invocación del método **findPerfectNumbersInRange(...)** por parte del cliente, el cual recibe:

- El rango numérico a evaluar (**Range**)
- El proxy de notificación del cliente (**ClientNotifierPrx**)
- El número de trabajadores solicitados

Con esta información, el maestro ejecuta los siguientes pasos:

1. **Consulta de disponibilidad:** Identifica los trabajadores registrados utilizando su lista local de proxies (**WorkerServicePrx**).
2. **División del rango:** Divide el intervalo total en subrangos balanceados, ajustando dinámicamente el tamaño según la cantidad de trabajadores activos. Se garantiza que cada subrango tenga al menos un número.
3. **Asignación de tareas:** A cada trabajador se le asigna un subrango mediante una invocación asíncrona al método **processSubRangeAsync(...)**. Junto con el subrango, se le transmite:
 - El objeto **Range** correspondiente,
 - Un proxy de retorno al maestro (**MasterControllerPrx**),
 - Y un identificador único de tarea.

Este proceso se realiza de manera completamente asíncrona. En caso de que haya más trabajadores que subrangos, los nodos excedentes simplemente no reciben tarea, evitando así bloqueos innecesarios.

4.2. COORDINACIÓN Y RECOLECCIÓN DE RESULTADOS

Cada trabajador, al finalizar su subrango, ejecuta **submitWorkerResultsAsync(...)** para notificar al maestro. Esta invocación contiene:

- El identificador del worker (**workerId**)
- El subrango procesado

- El listado de números perfectos encontrados
- El tiempo de ejecución individual

Estas notificaciones son gestionadas por la clase `MasterControllerI.java`, que:

- Acumula los resultados parciales por cada trabajador,
- Sincroniza la recepción de respuestas utilizando un **`CountDownLatch`**,
- Y ordena los resultados finales para consolidarlos.

Una vez recibidas todas las respuestas o alcanzado un límite de tiempo, el maestro calcula el tiempo total de ejecución y comunica los resultados finales al cliente mediante **`notifyJobCompletionAsync(...)`**.

4.3. ROBUSTEZ Y TOLERANCIA A FALLOS

El sistema contempla escenarios adversos como:

- Trabajadores no disponibles o desconectados
- Rangos inválidos o vacíos
- Fallas de red o tiempos de espera prolongados (**`timeout de 10 minutos`**)

Estas situaciones son controladas mediante validaciones y mecanismos de recuperación implementados en diversos métodos como **`resetForNewJob(...)`** o **`handleWorkerFailureOrNoTask(...)`**. Gracias a esto, el sistema mantiene su funcionalidad incluso ante fallos parciales, garantizando una ejecución confiable y continua.

5. RESULTADOS EXPERIMENTALES Y ANÁLISIS DE RENDIMIENTO: Con el fin de evaluar el rendimiento, la escalabilidad y la eficiencia del sistema distribuido propuesto, se llevó a cabo un experimento controlado en el que se manipularon dos factores clave: el tamaño del rango de búsqueda y el número de trabajadores disponibles para el procesamiento paralelo.

5.1. DISEÑO EXPERIMENTAL

Factor 1 – Tamaño del rango de búsqueda: Este factor representa la cantidad de números naturales consecutivos sobre los cuales se realiza la verificación de números perfectos. En nuestro caso, se utilizaron los siguientes rangos: **[1 – 10,000]**, **[1 – 100,000]**, **[1 – 1,000,000]** y **[1 – 10,000,000]**. Estos niveles permiten observar el comportamiento del sistema en escenarios de baja, media y alta carga computacional.

Factor 2 – Número de trabajadores disponibles: Este factor determina el grado de paralelismo del sistema distribuido. En este caso, se consideraron los siguientes niveles: **1, 3, 5, 8 y 10 trabajadores**. Esto permite medir el impacto que tiene el aumento de recursos en la reducción del tiempo de procesamiento.

Para cada combinación de estos factores, se midió la **variable de respuesta principal**, correspondiente al **tiempo total de ejecución (ms)**, registrado por el cliente desde el envío de la solicitud hasta la recepción del resultado final. Estos tiempos fueron almacenados en el archivo **tiempos_ejecucion.txt**.

Además, se calcularon dos métricas fundamentales para analizar el desempeño del sistema distribuido:

<ul style="list-style-type: none"> Aceleración: 	Indica la mejora en tiempo de procesamiento que se logra gracias al paralelismo. Donde T_1 es el tiempo de ejecución total con un solo trabajador (referencia) y T_n es el tiempo de ejecución total con n trabajadores.
$A_n = \frac{T_1}{T_n}$	
<ul style="list-style-type: none"> Eficiencia: 	Indica qué tan útil resulta, en promedio, cada trabajador adicional. Donde E_n representa la eficiencia del sistema al usar n trabajadores, y su valor se encuentra entre 0 y 1 (o en porcentaje multiplicando por 100).
$E_n = \frac{A_n}{n} = \frac{T_1}{n \cdot T_n}$	

NOTA: Para ejecutar el experimento, fue necesario realizar algunas modificaciones en las **direcciones IP** de los clientes y workers. Además, se agregó la ruta de **ICE** en los equipos y se configuró el entorno correspondiente con **Java 17**, tal como se había definido previamente.

```

</> client.properties x
1  # --- Archivo: client/src/main/resources/client.properties ---
2
3  # Configura el endpoint para recibir notificaciones del Maestro
4  # Cambia 'localhost' por la IP de este cliente si el Maestro está en otra máquina
5  ClientNotifierAdapter.Endpoints=default -h localhost
6
7  # Proxy para conectarse al servicio del Maestro
8  # 'localhost' es la IP o hostname del Maestro si no está en la misma máquina
9  MasterService.Proxy=MasterService:default -h localhost -p 10000
10 # Para la sala de laboratorio:
11 # MasterService.Proxy=MasterService:default -h 192.168.131.31 -p 10000

```

```

</> worker.properties x
1  # --- Archivo: worker/src/main/resources/worker.properties ---
2  # Configuración del adaptador Worker (escucha en la interfaz local)
3  # Cambia 'localhost' por la IP alcanzable si el Maestro está en otra máquina
4  WorkerAdapter.Endpoints=default -h localhost
5  # Para el salón de laboratorios
6  # WorkerAdapter.Endpoints=default -h xxx.xxx.xxx.xxx
7
8  # Proxy para conectarse al servicio Maestro
9  # 'localhost' es la IP o hostname del Maestro
10 MasterService.Proxy=MasterService:default -h localhost -p 10000
11 # Para el salón de laboratorio
12 # MasterService.Proxy=MasterService:default -h 192.168.131.31 -p 10000
13
14 # Opciones de trazas de Ice: desactivadas para red y protocolo
15 Ice.Trace.Network=0
16 Ice.Trace.Protocol=0
17
18 # Mostrar advertencias de conexiones de Ice
19 Ice.Warn.Connections=1
20
21 # Configuración para timeouts en milisegundos
22 Ice.Default.ConnectTimeout=5000

```

```

build.gradle (:PerfectNumbersApp) x
1  // PerfectNumbersDistributed/PerfectNumbersApp/build.gradle
2  plugins {
3      id 'java' // Permite compilar los stubs generados por ICE en este módulo
4      id 'com.zeroc.gradle.ice-builder.slice' // Plugin para procesar archivos .ice y genera
5  }
6
7  // Configuración específica para la compilación de Slice a Java
8  slice {
9      java { Java it ->
10         // Indica dónde está el archivo App.ice que define las interfaces
11         files = [file('../App.ice')]
12         // Aquí se podrían agregar opciones extra, como args = ['--verbose']
13     }
14     // Para la sala de laboratorios que necesitamos la ruta específica de ICE
15     // iceHome = file("/opt/Ice-3.7.6")
16 }
17
18 // ⚠ Este subproyecto solo genera la librería de stubs de Ice, no es una app ejecutable
19 // La dependencia com.zeroc:ice:3.7.10 se hereda del build.gradle principal del proyecto

```

5.2. PROCEDIMIENTO EXPERIMENTAL

1. Primero, se prepararon las condiciones necesarias para la ejecución del experimento en el laboratorio de computación (lo mencionado en la nota anterior), utilizando un total de **12 equipos**.
2. Posteriormente, se puso en marcha el sistema siguiendo un orden específico: primero se ejecutó el **Master**, luego los **10 Workers**, y finalmente el **Cliente**.
3. Una vez iniciado el sistema, se interactuó desde el cliente, ingresando el rango correspondiente y el número de trabajadores deseado por cada escenario.
4. Se recopilaron los tiempos de ejecución reportados por el **cliente**, los cuales fueron registrados de forma automática en el archivo **tiempos_ejecucion.txt**.
5. Cada configuración fue repetida **tres veces**, con el objetivo de mitigar el efecto de variaciones no controladas y obtener un **promedio representativo**:

Rango de Búsqueda	Nº Workers	Repetición 1 (ms)	Repetición 2 (ms)	Repetición 3 (ms)
[1 – 10,000]	1	77	30	27
[1 – 100,000]	1	78	75	75
[1 – 1,000,000]	1	1515	1498	1499
[1 – 10,000,000]	1	45724	45720	45710
[1 – 10,000]	3	110	24	25
[1 – 100,000]	3	61	49	46
[1 – 1,000,000]	3	692	687	701
[1 – 10,000,000]	3	20796	20810	20798
[1 – 10,000]	5	146	23	19
[1 – 100,000]	5	52	43	47
[1 – 1,000,000]	5	438	437	437
[1 – 10,000,000]	5	12957	12961	12958
[1 – 10,000]	8	91	19	19
[1 – 100,000]	8	43	30	28
[1 – 1,000,000]	8	309	307	304
[1 – 10,000,000]	8	9144	9153	9143
[1 – 10,000]	10	78	27	25
[1 – 100,000]	10	39	34	35
[1 – 1,000,000]	10	236	237	237
[1 – 10,000,000]	10	6669	6670	6667

6. Finalmente, los resultados obtenidos se consolidaron en una tabla resumen, en la cual se calcularon tanto el tiempo promedio de ejecución para cada configuración como las métricas correspondientes de aceleración y eficiencia.

Rango de Búsqueda	N° Workers	Tiempo Promedio (ms)	Aceleración (ms)	Eficiencia (ms)
[1 – 10,000]	1	44,67	N/A	N/A
[1 – 100,000]	1	76	N/A	N/A
[1 – 1,000,000]	1	1504	N/A	N/A
[1 – 10,000,000]	1	45718	N/A	N/A
[1 – 10,000]	3	53	0,84	0,28
[1 – 100,000]	3	52	1,46	0,49
[1 – 1,000,000]	3	693,33	2,17	0,72
[1 – 10,000,000]	3	20801,33	2,20	0,73
[1 – 10,000]	5	62,67	0,71	0,14
[1 – 100,000]	5	47,33	1,61	0,32
[1 – 1,000,000]	5	437,33	3,44	0,69
[1 – 10,000,000]	5	12958,67	3,53	0,71
[1 – 10,000]	8	43	1,04	0,13
[1 – 100,000]	8	33,67	2,26	0,28
[1 – 1,000,000]	8	306,67	4,90	0,61
[1 – 10,000,000]	8	9146,67	5,00	0,62
[1 – 10,000]	10	43,33	1,03	0,10
[1 – 100,000]	10	36	2,11	0,21
[1 – 1,000,000]	10	236,67	6,35	0,64
[1 – 10,000,000]	10	6668,67	6,86	0,69

5.3. ANÁLISIS E INTERPRETACIÓN DE RESULTADOS: Al analizar los resultados experimentales, se identificaron varios patrones relevantes. En primer lugar, se observó que en todas las configuraciones el tiempo de la **primera ejecución** fue mayor que el de las repeticiones subsiguientes. Este comportamiento se atribuye a **la carga inicial del sistema**, la cual incluye la activación de servicios, el establecimiento de conexiones de red y la sincronización entre procesos distribuidos. Una vez completadas estas fases en la primera ejecución, los recursos del sistema ya se encuentran activos o en caché, lo que explica la mejora significativa en los tiempos de las repeticiones posteriores.

Rango de Búsqueda	N° Workers	Repetición 1 (ms)	Repetición 2 (ms)	Repetición 3 (ms)
[1 – 10,000]	1	77	30	27
[1 – 10,000]	3	110	24	25
[1 – 10,000]	5	146	23	19
[1 – 10,000]	8	91	19	19
[1 – 10,000]	10	78	27	25

Por otro lado, se evidenció un comportamiento inesperado en los dos primeros niveles de carga ([1 – 10,000] y [1 – 100,000]). En estos casos, el tiempo de ejecución promedio con 10 trabajadores no fue inferior al registrado con 8 trabajadores, lo cual contradice la expectativa teórica de que un mayor número de nodos debería reducir el tiempo de procesamiento total.

Rango de Búsqueda	N° Workers	Repetición 1 (ms)	Repetición 2 (ms)	Repetición 3 (ms)
[1 – 10,000]	8	91	19	19
[1 – 100,000]	8	43	30	28
[1 – 10,000]	10	78	27	25
[1 – 100,000]	10	39	34	35

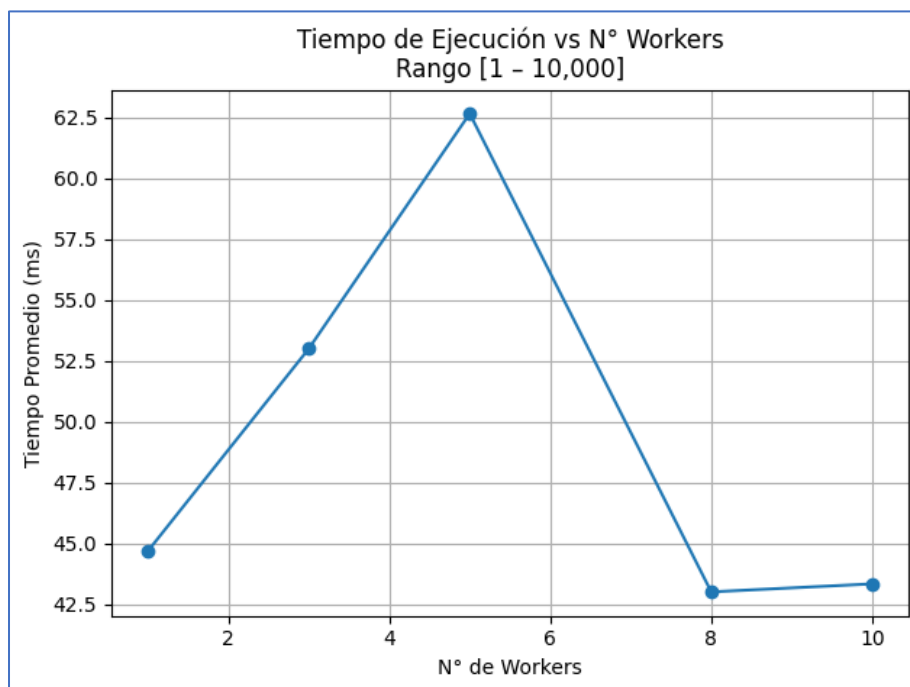
Una explicación plausible para este fenómeno es que, en tareas computacionalmente ligeras, el sobre costo de coordinación incurrido por el maestro al gestionar un número elevado de trabajadores puede superar los beneficios del paralelismo adicional. Aunque los workers no muestran demoras significativas en su ejecución individual, el proceso de consolidar los resultados provenientes de 10 nodos impone una carga adicional sobre el maestro, generando una pérdida de eficiencia. Esta situación cambia a partir del tercer rango, [1 – 1,000,000], donde la carga de trabajo es suficiente para que los beneficios de una mayor paralelización superen el **overhead** de coordinación.

Así mismo, se realiza un análisis global de las **métricas de rendimiento calculadas**:

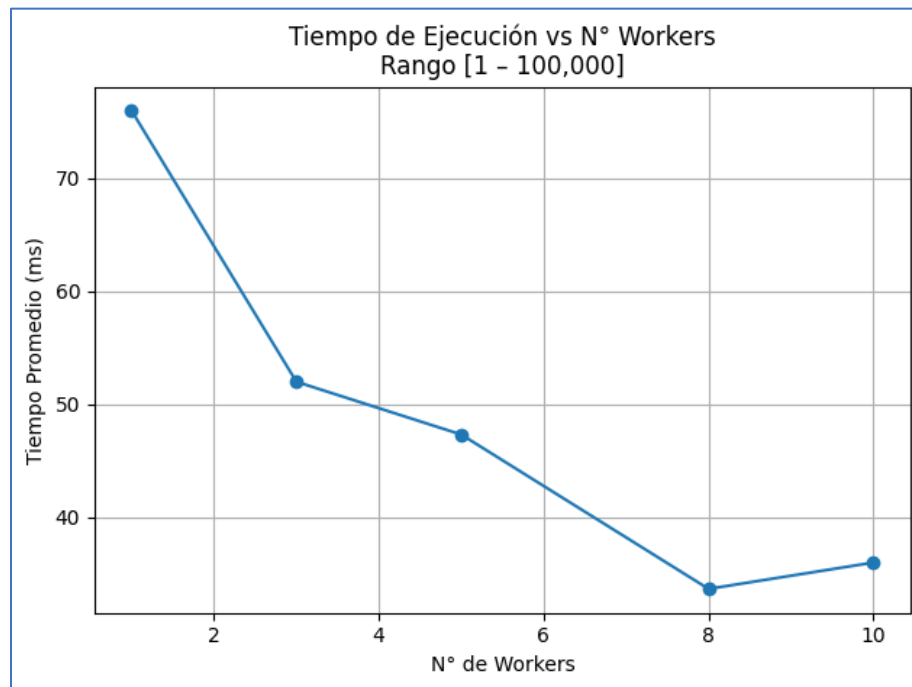
Aceleración: Los resultados muestran que el sistema logra mejorar considerablemente su rendimiento con el incremento del número de trabajadores, particularmente a medida que se incrementa la carga computacional. La aceleración es moderada en tareas pequeñas, pero se vuelve más pronunciada en rangos amplios. Por ejemplo, en el caso del rango [1 – 10,000,000], se alcanza una aceleración de **6.86** al utilizar **10 trabajadores**, lo que representa una reducción significativa del tiempo de ejecución respecto a la ejecución secuencial. Este comportamiento indica que el sistema aprovecha adecuadamente el paralelismo, especialmente cuando la división del trabajo resulta proporcional al esfuerzo requerido.

Eficiencia: En contraste, la eficiencia presenta un comportamiento decreciente a medida que aumenta el número de trabajadores, lo cual es un fenómeno habitual en sistemas distribuidos. Si bien la aceleración mejora, el rendimiento por trabajador no escala linealmente debido al **overhead de coordinación** y la consolidación de resultados. En tareas pequeñas, este efecto es más notorio, alcanzando eficiencias inferiores al **30%** en muchas configuraciones. Sin embargo, en cargas mayores, la eficiencia tiende a estabilizarse en valores superiores al **60%**, mostrando que el sistema logra distribuir la carga de forma más efectiva conforme se incrementa la complejidad del problema.

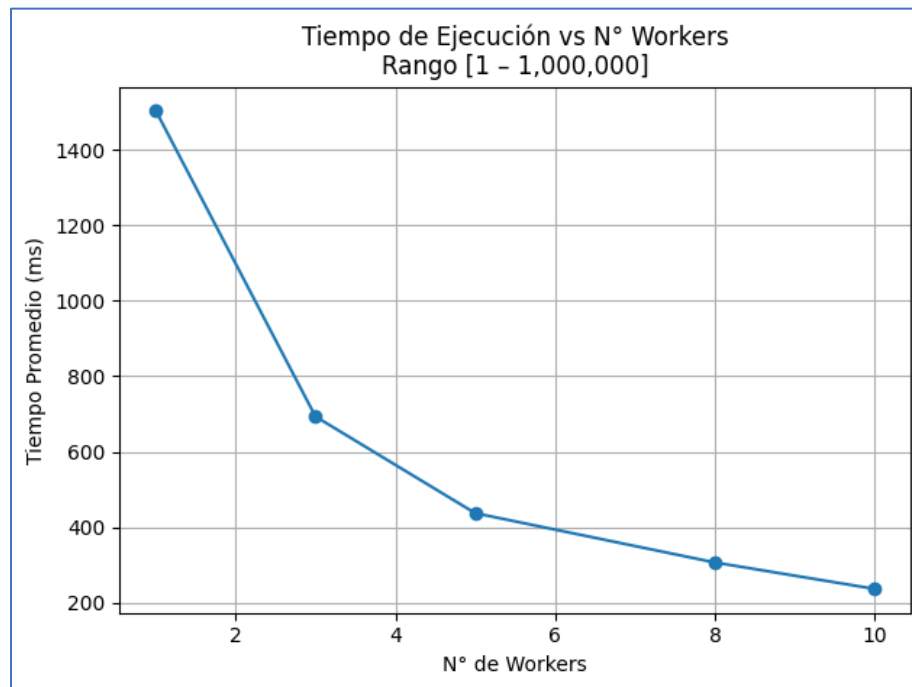
Interpretación por Rango de Búsqueda: A continuación, se presenta un análisis detallado de los resultados obtenidos del tiempo promedio para cada rango, acompañado de representaciones gráficas que ilustran la relación entre el número de workers y el tiempo promedio de ejecución. La implementación en Python de cada gráfico se encuentra en el siguiente [Colab](#):



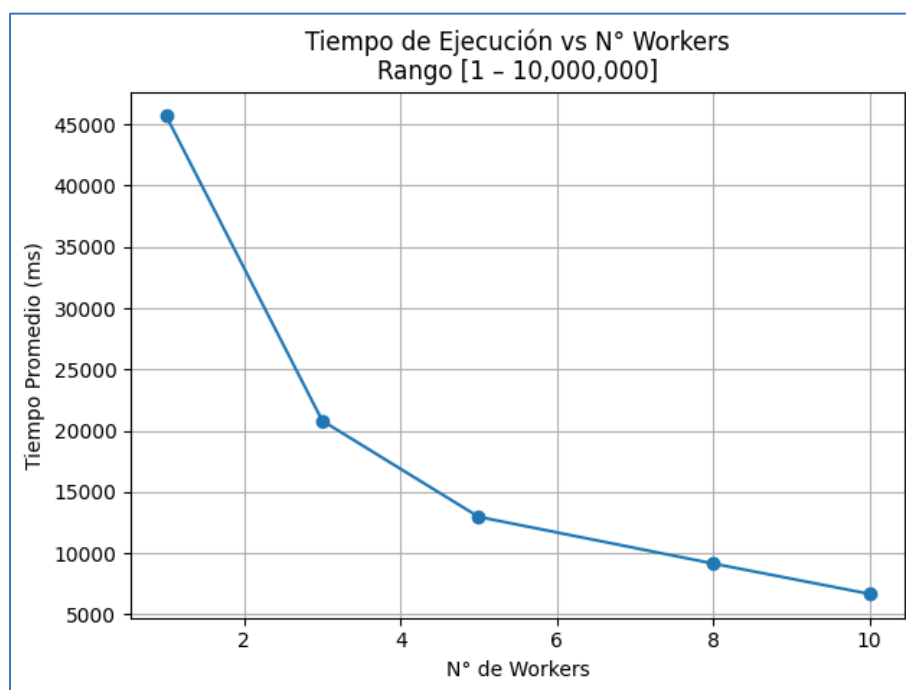
[1 – 10,000]: Este rango corresponde a una **carga computacional muy baja**. Como muestran los resultados, el tiempo de ejecución no disminuye de manera consistente con el aumento en el número de workers. De hecho, se observa un pico inesperado en los 5 trabajadores, lo cual sugiere que el **overhead** asociado a la coordinación, así como los costos de comunicación entre nodos, superan los beneficios del paralelismo en tareas pequeñas. A partir de 8 workers, el sistema vuelve a estabilizarse, pero sin lograr mejoras significativas frente a configuraciones más simples. En este escenario, el uso de múltiples nodos resulta ineficiente y puede incluso penalizar el rendimiento.



[1 – 100,000]: Con una carga ligeramente mayor, se evidencia una **mejora más clara** en los tiempos de ejecución a medida que se incrementa el número de workers, especialmente hasta 8 nodos. Sin embargo, al llegar a 10 workers, el rendimiento decrece levemente. Esto refuerza la hipótesis de que, aunque ya se empieza a aprovechar el paralelismo, el costo de consolidar resultados entre muchos nodos aún impacta negativamente cuando la carga no es suficientemente grande. El sistema es funcional, pero la eficiencia comienza a desestabilizarse en este punto.



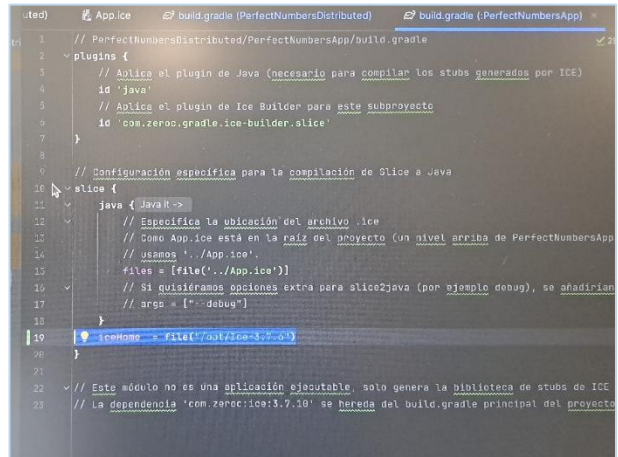
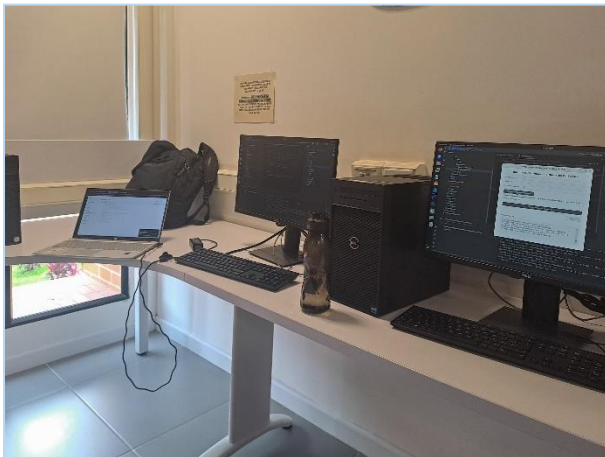
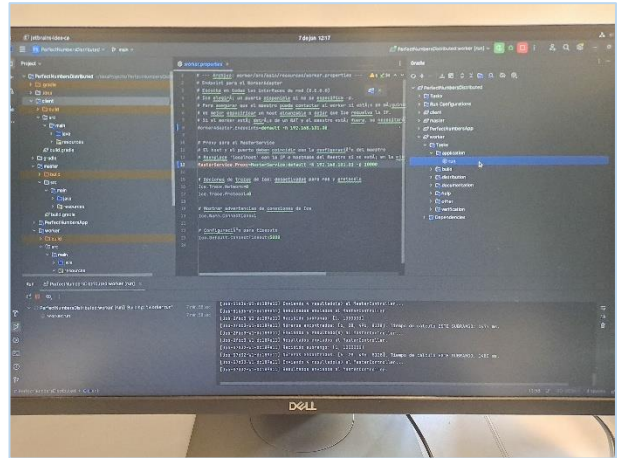
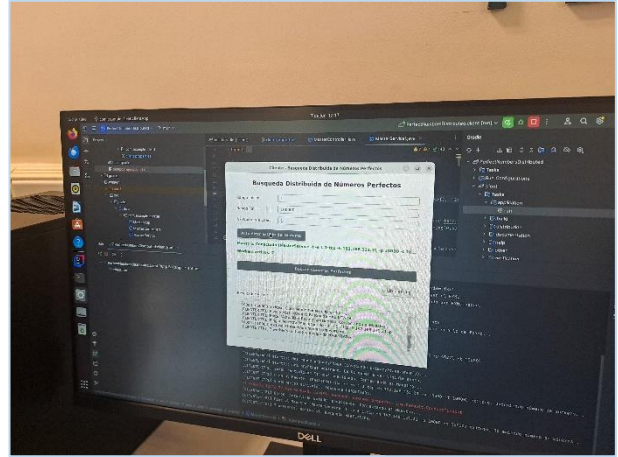
[1 - 1,000,000]: En este rango, el sistema comienza a mostrar un comportamiento de escalabilidad más definido y positivo. El tiempo de ejecución disminuye de forma sostenida con el incremento de workers, sin retrocesos abruptos. A partir de este volumen de datos, la tarea se vuelve lo suficientemente compleja como para que el maestro distribuya efectivamente la carga, y el overhead de coordinación queda amortizado por los beneficios del paralelismo. El uso de 10 workers demuestra ser el más eficiente, logrando el mejor desempeño registrado hasta este punto.



[1 - 10,000,000]: Este rango representa una carga altamente demandante y revela un comportamiento ideal de escalabilidad. A medida que se incrementa el número de workers, se observa una disminución constante y significativa en el tiempo de ejecución. No se presentan retrocesos, lo que indica que el sistema está operando en su rango óptimo de paralelismo. La eficiencia se mantiene elevada, y el uso de 10 trabajadores permite aprovechar al máximo los recursos distribuidos. Este escenario valida plenamente la arquitectura propuesta.

6. ANEXOS Y/O EVIDENCIAS:

A continuación, se muestran algunas fotografías tomadas durante la puesta en marcha y ejecución del laboratorio de búsqueda distribuida de números perfectos.



7. CONCLUSIONES Y POSIBLES MEJORAS:

Los resultados obtenidos muestran que la arquitectura distribuida Cliente-Maestro-Trabajadores implementada con ICE logra cumplir su propósito el cual es: **mejorar el rendimiento del cálculo de números perfectos al escalar la carga entre múltiples nodos de ejecución.**

En cargas pequeñas (como el rango [1 – 10,000]), el sistema presenta limitaciones propias al paralelismo, como el **overhead de coordinación**, que puede incluso degradar el rendimiento al utilizar demasiados workers. No obstante, a medida que la complejidad del problema crece, el sistema comienza a escalar de manera efectiva. A partir del tercer rango [1 – 1,000,000], se observan beneficios evidentes al aumentar el número de nodos, tanto en la reducción del tiempo de ejecución como en la mejora de métricas como la aceleración y la eficiencia.

Con 10 millones de números por analizar, el sistema demuestra su mayor potencial: **el uso de 10 workers permite alcanzar una aceleración cercana a 7x respecto a la ejecución secuencial, manteniendo una eficiencia superior al 60%.** Esto valida el diseño asincrónico y desacoplado del sistema, en el cual los componentes se comunican sin bloqueos, y donde la consolidación de resultados es manejada eficazmente a través de **callbacks**.

Además, la implementación muestra robustez en escenarios adversos: la tolerancia a workers desconectados, la detección de rangos inválidos, y los mecanismos de timeout permiten mantener la continuidad del sistema incluso ante fallos parciales.

Entre las posibles mejoras que se podrían explorar se incluyen:

- **Balance dinámico de carga:** Actualmente los subrangos son asignados de forma equitativa, pero no se considera que algunos valores son más costosos de procesar que otros. Un reparto adaptativo podría mejorar aún más la eficiencia.
- **Reintentos ante fallos de workers:** Si un nodo falla al procesar un subrango, no se intenta reasignar esa tarea a otro nodo. Incluir esta capacidad mejoraría el sistema.
- **Configuración dinámica de direcciones IP:** En la versión actual, las direcciones de los nodos deben ser configuradas manualmente en los archivos de propiedades. Incorporar un mecanismo de descubrimiento automático o asignación dinámica simplificaría la ejecución del sistema, especialmente en entornos con múltiples nodos o direcciones **IPs** cambiantes.

En resumen, el proyecto constituye una base funcional y muy sólida para la búsqueda distribuida de números perfectos, mostrando beneficios claros en escenarios de alta carga. Su arquitectura modular y extensible facilita futuras mejoras tanto en funcionalidad como en escalabilidad.