IMPLEMENTACIÓN DE BENCHMARK EN CONTENEDORES

Martínez Medina, Daniel. 200180871
Murillo Galindo, Nicole. 200181355

Informe de implementación, Grupo Presencial 2288, Universidad del Norte

Augusto Salazar Silva
Febrero 10 de 2025

RESUMEN

El presente informe tiene como objetivo general detallar y explicar la implementación de un sistema de evaluación comparativa basado en contenedores, creado para comparar el desempeño de distintos lenguajes de programación al solucionar un problema matemático. El proyecto emplea estructuras de Docker de multi-etapa y compose de contenedores para generar un ambiente de pruebas automatizado, reproducible y compatible.

SELECCIÓN DEL PROBLEMA E IMPLEMENTACIÓN

Para este benchmark, implementamos un algoritmo de búsqueda de raíces con bisección en cinco lenguajes de programación diferentes: Python, Java, JavaScript, Rust y C++. Se eligió el método de bisección ya que es un algoritmo que es fácilmente escalable en cuanto a complejidad, es decir, dependiendo de la funcione que se le dé, esta puede convertirse en una tarea computacionalmente exigente, haciéndolo ideal para la comparación de rendimiento en un benchmark.

Cada implementación en los diferentes lenguajes sigue estos pasos:

1. Ejecuta el mismo algoritmo del método de bisección
2. Mide el tiempo de ejecución en milisegundos
3. Escribe los resultados en un archivo de salida estandarizado llamado `execution_time.txt`

DISEÑO DE LA ARQUITECTURA

La solución emplea una arquitectura containerizada de dos capas:

- **Contenedor Interior (Ejecución de los benchmarks en cada lenguaje)**
  El contenedor interior utiliza un enfoque de dockerfile multi-etapa para:
  1. Ejecutar cada implementación de lenguaje de forma independiente
  2. Recolectar resultados de tiempo
  3. Agregar resultados en una etapa final

El proceso multi-etapa sigue esta secuencia:

1. Etapa Python
    a. Utiliza imagen base Python 3.12.8
    b. Instala la biblioteca SymPy para operaciones matemáticas
    c. Ejecuta el algoritmo de bisección
    d. Genera salida de tiempo en execution_time.txt
2. Etapa JavaScript
    a. Utiliza el entorno Node.js 18
    b. Ejecuta la implementación en JavaScript puro
    c. Registra el tiempo de ejecución
3. Etapa Java
    a. Utiliza Amazon Corretto JDK 21
    b. Compila y ejecuta la implementación en Java
    c. Mide el tiempo de ejecución usando System.nanoTime()
4. Etapa C++
    a. Utiliza el compilador GCC 12
    b. Compila y ejecuta la implementación en C++
    c. Registra mediciones precisas de tiempo
5. Etapa Rust
    a. Utiliza el compilador Rust 1.72
    b. Implementa el método de bisección con seguridad de memoria
    c. Registra el tiempo de ejecución
6. Etapa Final
    a. Utiliza Alpine Linux ligero
    b. Recopila archivos de tiempo de todas las etapas anteriores
    c. Agrega resultados en una única salida

- **Contenedor Exterior (Docker Compose)**
  El contenedor exterior maneja la automatización del flujo de trabajo general:
    1. Clona el repositorio que contiene todas las implementaciones
    2. Construye el contenedor interior
    3. Ejecuta la suite de benchmark
    4. Devuelve todos los resultados

  Este contenedor utiliza:

    - Imagen Docker más reciente con soporte para Docker-in-Docker (DinD)
    - Git para la clonación del repositorio
    - Montaje del socket Docker para la ejecución de contenedores anidados

DETALLES DE IMPLEMENTACIÓN

- Configuración de Docker Compose
  La configuración de Docker Compose proporciona:
  - Montaje de volumen del socket Docker
  - Modo privilegiado para funcionalidad DinD
  - Nombrado de contenedor para fácil referencia
  - Configuración del contexto de construcción
- Gestión del Código Fuente
  Todo el código fuente se mantiene en un repositorio público de GitHub, organizado con:
  - Directorios separados para cada implementación de lenguaje
  - Dockerfiles individuales para cada etapa
  - Formato de salida común para consistencia de resultados