Trabalho Final

Redes de Computadores - QXD0021

Heric da Silva Cruz
hericsilvaho@gmail.com

Universidade Federal do Ceará

10 de julho de 2025

Algoritmos Genéticos

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Desenvolvidos por John Holland em 1975.
- Inspirados nos processos de evolução natural Trabalham com uma população de soluções candidatas (cromossomos) e usam operadores de seleção, cruzamento e mutação.
- Usados para encontrar soluções aproximadas para problemas NP-Completos e outros.

Algorithm 1 Basic GENETIC ALGORITHM

```
initialize population
repeat
    repeat
        crossover
        mutation
        phenotype mapping
        fitness computation
    until population complete
    selection of parental population
until termination condition
```

Pseudocódigo de um algoritmo genético simplificado.

O Desafio da Execução

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- A natureza estocástica dos Algoritmos Genéticos significa que cada execução pode gerar uma solução final diferente.

- Para obter um conjunto robusto e **diversificado** de boas soluções para um mesmo problema, é essencial executar o algoritmo **múltiplas vezes**.

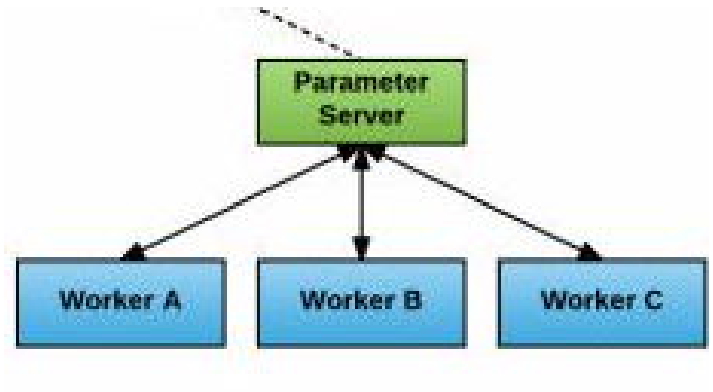
- Embora uma única execução possa ter um tempo "razoável", a necessidade de dezenas ou centenas de execuções sequenciais transforma o processo em uma tarefa extremamente demorada.

A Solução: Kambo-Hive

A solução é dividir as múltiplas execuções do algoritmo, distribuindo a carga de trabalho em uma arquitetura **Cliente-Servidor**:

- **Host (Servidor):** Gerencia e distribui a fila de tarefas a serem processadas.

- **Workers (Clientes):** Solicitam tarefas ao Host, executam o algoritmo de forma independente e devolvem o resultado.



O Host distribui tarefas para múltiplos Workers, que as processam em paralelo.

Requisitos do Projeto

A Solução foi desenvolvida segundo os requisitos do problema e incluindo os requisitos para a pontuação extra:

- **Disponibilidade do Código:**
  Versionamento e publicação no GitHub.

- **Pesquisa Automática de Host:**
  Workers encontram o servidor na rede local via broadcast.

- **Distribuição Configurável:**
  O Host permite alterar a estratégia de envio de tarefas (FIFO, LIFO, etc).

- **Relatório de Uso Detalhado:**
  Geração de um arquivo JSON com estatísticas da execução.

Passo 1: A Requisição do Worker

O ciclo se inicia com um Worker ativo solicitando uma tarefa ao Host.

- O Worker envia uma mensagem **`RequestTask`**, que contém apenas seu próprio ID único para identificação.

```rust
#[derive(Debug, Serialize, Deserialize)]
pub enum Request {
    RequestTask { worker_id: Uuid },
    ReportResult { worker_id: Uuid, result: TaskResult },
    Heartbeat { worker_id: Uuid },
}
```

A enum `Request` define as possíveis ações de um Worker.

Passo 2: A Atribuição da Tarefa

Ao receber a requisição, o Host consulta sua fila de tarefas pendentes e responde conforme a estratégia de distribuição configurada.

- Se houver trabalho, o Host envia a mensagem **`AssignTask`**.
- Essa mensagem carrega um objeto **`Task`**, que contém todas as informações que o Worker precisa para executar o algoritmo.

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Task {
    pub id: Uuid,
    pub graph_id: String,
    pub run_number: u32,
    pub ag_config: String,
}

```

A struct `Task` encapsula os dados de uma única execução.

Passo 3: O Relato do Resultado

Após finalizar o processamento, o Worker envia seu resultado de volta para o Host.

- A mensagem **`ReportResult`** carrega um objeto **`TaskResult`**.
- Esta estrutura contém o fitness da solução, o tempo de processamento, e outros detalhes da execução.
- O Host armazena esses dados para o relatório final e confirma o recebimento.

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct TaskResult {
    pub task_id: Uuid,
    pub graph_id: String,
    pub worker_id: Uuid,
    pub fitness: f64,
    pub solution_data: Vec<u8>,
    pub interations_run: u32,
    pub processing_time_ms: u64,
}
```

A struct `TaskResult` com os dados da solução encontrada.

O Cérebro do Sistema: O Host

Além de distribuir tarefas, o Host gerencia todo o ciclo de vida da execução através de três componentes principais:

- **TaskManager:**
  Controla a fila de tarefas, o status de cada uma (pendente, atribuída, concluída) e implementa as diferentes **estratégias de distribuição** como *FIFO*, *LIFO* e *RANDOM*.

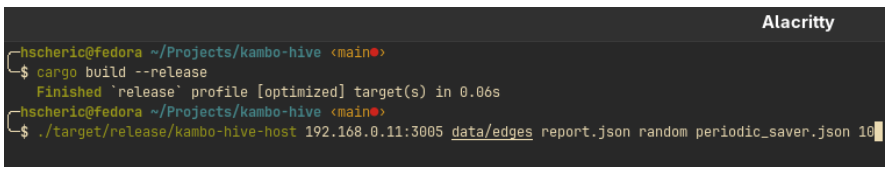
- **ResultAggregator:**
  Recebe e armazena todos os `TaskResult` enviados pelos workers, centralizando os dados para a análise final, utiliza um ArcMutex para fazer isso.

- **PeriodicSaver:**
  De forma assíncrona, salva o progresso dos resultados em um arquivo. Isso garante a **"tolerância a falhas"**, permitindo a recuperação dos dados caso o Host seja interrompido.

Kambo-Hive em Execução

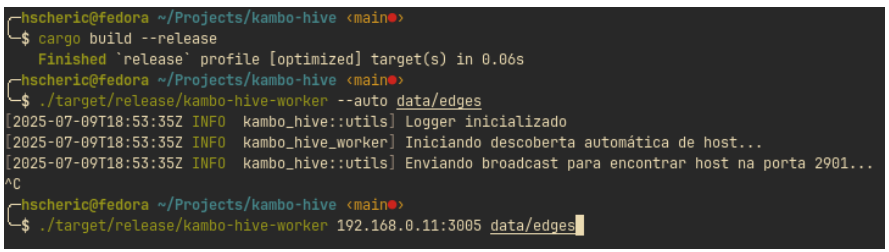
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Host Inicializado



Host carregando as tarefas e aguardando conexões.

Workers em Execução



Workers se conectam, solicitam e processam tarefas em paralelo.

Análise de Desempenho: O Relatório Final

Ao final da execução (ou ao ser interrompido), o Host gera um relatório detalhado em formato JSON, consolidando todos os dados da operação.

Este relatório permite analisar:

- O desempenho individual de cada Worker (tarefas resolvidas, tempo médio)..
- Um resumo geral da saúde da execução (total de tarefas, falhas, etc.).

Conclusão e Contexto do Projeto

Primeiramente **Kambo-Hive** cumpriu todos os requisitos propostos para o trabalho acadêmico. O código-fonte completo está publicamente disponível no GitHub:

`github.com/hscHeric/kambo-hive`

Uma curiosidade: por que "reinventar a roda"?

- Já existem soluções consolidadas para computação paralela e distribuída, como OpenMP e MPI.

- Construir o sistema do zero permitiu explorar na prática conceitos que o uso de uma biblioteca pronta abstrairia, como:
  - O design de um protocolo de aplicação sobre TCP/IP.
  - Gerenciamento de estado concorrente com Tokio e Mutex.
  - Mecanismos de descoberta de serviço em rede local (UDP Broadcast).