ERCEMAPI

Escola Regional de Computação do Ceará, Maranhão e Piauí

LIVRO DE MINICURSOS

XIII EDIÇÃO – 2025

Organizadores

ANTONIO FERNANDO LAVAREDA JACOB JUNIOR

ATSLANDS REGO DA ROCHA

EDUILSON LÍVIO NEVES DA COSTA CARNEIRO

REALIZAÇÃO

SBC

Sociedade Brasileira de Computação

ORGANIZAÇÃO

UEMA

Uema

UNIVERSIDADE ESTADUAL DO MARANHÃO

UFMA

A VIDA É COMBATE

Universidade Federal do Maranhão

APOIO

FAPEMA

Fundação de Amparo à Pesquisa e ao Desenvolvimento Científico e Tecnológico do Maranhão

ERCEMAPI
Escola Regional de Computação do Ceará, Maranhão e Piauí

XIII Escola Regional de Computação do Ceará, Maranhão e Piauí

04 e 05 de dezembro de 2025

São Luís - Maranhão

LIVRO DE MINICURSOS

ERCEMAPI 2025

ORGANIZAÇÃO DO LIVRO:

ANTONIO FERNANDO LAVAREDA JACOB JUNIOR

ATSLANDS REGO DA ROCHA

EDUILSON LÍVIO NEVES DA COSTA CARNEIRO

Porto Alegre

Sociedade Brasileira de Computação – SBC

2026

Dados Internacionais de Catalogação na Publicação (CIP)

E65 Escola Regional de Computação do Ceará, Maranhão e Piauí (13. : 04 – 05 setembro 2025 : São Luís)

Minicursos da ERCEMAPI 2025 [recurso eletrônico] / organização: Antonio Fernando Lavareda Jacob Junior; Atslands Rego da Rocha; Eduilson Lívio Neves da Costa Carneiro. Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2025.

48 p. : il. : PDF ; 2 MB

Modo de acesso: World Wide Web. Inclui bibliografia

ISBN 978-85-7669-664-3 (e-book)

1. Computação – Brasil – Evento. 2. . 3. Análise de dados. I. Jacob Junior, Antonio Fernando Lavareda. II. Rocha, Atslands Rego da. III. Carneiro, Eduilson Lívio Neves da Costa. IV. Sociedade Brasileira de Computação. V. Título.

CDU 004(063)

Ficha catalográfica elaborada por Annie Casali – CRB-10/2339

Biblioteca Digital da SBC – SBC OpenLib

Sociedade Brasileira de Computação
Av. Bento Gonçalves, 9500
Setor 4 | Prédio 43.412 | Sala 219 | Bairro Agronomia Caixa Postal 15012 | CEP 91501-970
Porto Alegre - RS
(51) 99252-6018
sbc@sbc.org.br

Secretarias Regionais

Anselmo Cardoso de Paiva (Secretário SBC Maranhão - UFMA)

Atslands Rego da Rocha (Secretária SBC Ceará - UFC)

Eduilson Livio Neves da Costa Carneiro (Secretário SBC Piauí – UFDPar)

Coordenação Geral do Evento

Anselmo Cardoso de Paiva

Ewaldo Eder Carvalho Santana

Comitê do Programa de Minicursos

Antonio Fernando Lavareda Jacob Junior

Atslands Rego da Rocha

Eduilson Lívio Neves da Costa Carneiro

MENSAGEM DA COORDENAÇÃO GERAL DA ERCEMAPI 2025

As Escolas Regionais são eventos voltados aos estudantes de graduação e pós-graduação cujo objetivo principal é aproximar e facilitar a presença dos estudantes em eventos da área de Computação. A Escola Regional de Computação Ceará, Maranhão e Piauí (ERCEMAPI) tem como objetivo disseminar o conhecimento técnico e científico sobre temas e assuntos de vanguarda na área de Computação. As Escolas Regionais são eventos promovidos pela Sociedade Brasileira de Computação (SBC). A ERCEMAPI busca consolidar-se como um evento de referência nos estados do Ceará, Maranhão e Piauí, além de possuir importante papel na promoção da ciência e da pesquisa em Computação na região Nordeste. O evento também contribui para o fortalecimento e integração dos Programas de Pós-Graduação da região.

A XIII edição da ERCEMAPI foi realizada nos dias 04 e 05 de dezembro de 2025, na cidade de São Luís, Maranhão, sendo organizada pela Universidade Estadual do Maranhão (UEMA) em parceria com a Universidade Federal do Maranhão (UFMA). Nesta edição, o evento contou com 52 trabalhos submetidos, dos quais 19 foram selecionados para apresentação, distribuídos em sessões técnicas que contemplaram diferentes áreas da Computação.

A programação científica incluiu duas atividades de minicursos e três palestras convidadas, que abordaram temas atuais e relevantes da área. A palestra de abertura, intitulada *"Da Fonologia aos Mecanismos de Falha: Fenômenos Linguísticos do Português Brasileiro que Desafiam Modelos de ASR"*, foi proferida por Arthur Pereira Santana (Telus Digital). A segunda palestra, *"Inteligência Artificial Agêntica: Origens, Estado Atual e Caminhos para o Futuro"*, foi apresentada por Leandro Marinho (UFCG). Encerrando o evento, a palestra *"Ecossistemas de Software na Indústria"* foi ministrada por Rodrigo Pereira dos Santos (UNIRIO).

As sessões técnicas desta edição contemplaram trabalhos em diversas áreas, incluindo Ciência de Dados, Aprendizado de Máquina, Engenharia de Software, Empreendedorismo e Inovação, Revisão Sistemática da Literatura e Visualização de Dados, refletindo a diversidade e a qualidade das pesquisas desenvolvidas na região.

A primeira edição da Escola Regional de Computação dos Estados do Ceará, Maranhão e Piauí (ERCEMAPI) ocorreu em 2007, em Fortaleza (CE). A segunda edição foi realizada em São Luís (MA). Em 2009, a terceira edição ocorreu em Parnaíba (PI). Em 2010, a ERCEMAPI foi realizada em Sobral (CE), contando com 258 participantes de

diversas instituições de ensino superior do país e 41 trabalhos apresentados. Em 2011, o evento foi realizado pela UFPI, em parceria com o IFPI, no estado do Piauí, reunindo pesquisadores de diversas instituições brasileiras. Em 2012, a sexta edição ocorreu em São Luís (MA), em conjunto com a IV Jornada de Informática do Maranhão (JIM 2012).

Após um período de interrupção, o evento retornou em 2019, sendo sediado no Centro de Ciências Exatas da Universidade Federal do Maranhão (UFMA), com participação de estudantes e professores de diversos estados do país. Em 2020, a escola foi organizada pela comunidade do Piauí em formato virtual devido à pandemia, tendo Pesquisa e Empreendedorismo como temas centrais. Em 2021, o evento foi realizado novamente em formato online pela UFC – Campus de Quixadá. Em 2022, a organização ficou a cargo do Instituto Federal do Maranhão (IFMA), UEMA, UFMA e Centro Universitário UNDB. Em 2024, a ERCEMAPI foi sediada em Parnaíba (PI), no campus da Universidade Federal do Delta do Parnaíba (UFDPar). A edição de 2025, realizada em São Luís (MA), dá continuidade à trajetória de fortalecimento da ERCEMAPI como espaço de integração científica e acadêmica na região.

A realização desta edição contou com apoio da Fundação de Amparo à Pesquisa e ao Desenvolvimento Científico e Tecnológico do Maranhão (FAPEMA).

Coordenação Geral da ERCEMAPI 2025

Anselmo Cardoso de Paiva – Universidade Federal do Maranhão (UFMA)
Ewaldo Eder Carvalho Santana – Universidade Estadual do Maranhão (UEMA)

SINOPSE EM PORTUGUÊS

Este livro reúne os minicursos apresentados na Escola Regional de Computação Ceará, Maranhão e Piauí (ERCEMAPI 2025), abordando temas relacionados à inteligência artificial baseada em agentes e à análise de desempenho de aplicações paralelas. Os capítulos discutem conceitos fundamentais e aspectos práticos do desenvolvimento de sistemas computacionais, incluindo arquiteturas de agentes baseadas em Grandes Modelos de Linguagem (LLMs) e métodos para avaliação da escalabilidade e eficiência em ambientes de computação paralela. O primeiro capítulo apresenta a construção de chatbots e agentes inteligentes no paradigma multiagente utilizando os frameworks LangChain e LangGraph, relacionando conceitos da teoria clássica de agentes com sua implementação em aplicações modernas. O segundo capítulo aborda técnicas de perfilamento e análise da escalabilidade de aplicações paralelas, discutindo métricas como speedup, eficiência e escalabilidade, além do uso da ferramenta Parallel Scalability Suite (PaScal Suite) para coleta e visualização de dados de desempenho. O livro oferece, assim, um material de apoio para estudantes, pesquisadores e profissionais interessados em inteligência artificial e computação de alto desempenho.

SINOPSE EM INGLÊS

This book gathers the tutorials presented at the Regional School of Computing of Ceará, Maranhão and Piauí (ERCEMAPI 2025), addressing topics related to agent-based artificial intelligence and performance analysis of parallel applications. The chapters discuss fundamental concepts and practical aspects of developing computational systems, including agent architectures based on Large Language Models (LLMs) and methods for evaluating scalability and efficiency in parallel computing environments. The first chapter presents the construction of chatbots and intelligent agents using the multi-agent paradigm with the LangChain and LangGraph frameworks, relating concepts from classical agent theory to their implementation in modern applications. The second chapter discusses profiling and scalability analysis of parallel applications, addressing metrics such as speedup, efficiency, and scalability, as well as the use of the Parallel Scalability Suite (PaScal Suite) for collecting and visualizing performance data. The book therefore serves as supporting material for students, researchers, and professionals interested in artificial intelligence and high-performance computing.

SUMÁRIO

Capítulo 1

Agentes de IA: Construção de Chatbot no Paradigma Multiagente com LangChain e LangGraph

Gutemberg P. A. da Silva, Jailson C. Zarur, Luis H. M. Queiroz, Raimundo S. Moura

Abstract

This chapter presents a theoretical and practical approach to building intelligent agents using Large Language Models (LLMs). Bridging the gap between classical AI theory and modern frameworks, we map fundamental concepts defined by Russell and Norvig to implementations in LangChain and LangGraph. We explore architectures ranging from simple reflex agents to multi-agent systems with state persistence, demonstrating how to orchestrate complex workflows in production environments.

Resumo

Este capítulo apresenta uma abordagem teórico-prática para a construção de agentes inteligentes utilizando Grandes Modelos de Linguagem (LLMs). Estabelecendo uma ponte entre a teoria clássica de IA e frameworks modernos, mapeamos conceitos fundamentais definidos por Russell e Norvig para implementações em LangChain e LangGraph. Exploramos arquiteturas que variam de agentes reativos simples a sistemas multiagentes com persistência de estado, demonstrando como orquestrar fluxos de trabalho complexos em ambientes de produção.

1.1. Introdução e Motivações

A engenharia de sistemas modernos atravessa uma mudança de paradigma impulsionada pela integração massiva de Grandes Modelos de Linguagem (LLMs). Estes modelos deixaram de ser apenas ferramentas de processamento de texto para se tornarem componentes centrais em arquiteturas complexas de software. Nesse cenário, frameworks de orquestração como LangChain[1] e LangGraph[2] emergem como a infraestrutura essencial,

[1]https://docs.langchain.com/

[2]https://docs.langchain.com/langgraph/

fornecendo as fundações arquiteturais para construir e implantar sistemas inteligentes [1]. A adoção de agentes autônomos neste contexto permite que os desenvolvedores aproveitem essa abundância de capacidade cognitiva, transformando LLMs em entidades ativas capazes de realizar planejamento dinâmico e coordenar fluxos de trabalho complexos, superando as limitações de scripts estáticos tradicionais [5].

A teoria de agentes inteligentes, formalizada extensivamente na literatura clássica de Inteligência Artificial, define um agente como uma entidade que percebe seu ambiente através de sensores e age sobre ele através de atuadores [4]. Durante décadas, a implementação destes agentes exigiu a codificação manual de regras ou o treinamento de modelos específicos para tarefas restritas. No entanto, o advento dos LLMs alterou fundamentalmente este paradigma. Na arquitetura de agentes modernos, consolida-se o entendimento de que o LLM atua como o *controlador central* ou "cérebro" do agente, possuindo capacidades de raciocínio generalista e conhecimento paramétrico prévio para orquestrar os demais módulos do sistema [5].

Apesar da disponibilidade de ferramentas poderosas, existe uma desconexão frequente entre os fundamentos teóricos da IA e a prática de desenvolvimento de software. Desenvolvedores muitas vezes criam "chains" (cadeias de execução) sem compreender que estão, na verdade, implementando agentes reativos simples, ou utilizam grafos de estado sem perceber a correlação com agentes baseados em modelo.

Este capítulo visa preencher essa lacuna, consolidando o conteúdo teórico-prático do minicurso. Não trataremos apenas da sintaxe das ferramentas, mas de como elas materializam conceitos clássicos de racionalidade, planejamento e sistemas multiagentes. Ao longo das próximas seções, demonstraremos como a função matemática abstrata de um agente se traduz em código Python moderno, permitindo a construção de *chatbots* que não apenas conversam, mas executam tarefas complexas de forma robusta e auditável.

1.2. Fundamentos Teóricos e Mapeamento Moderno

Para construir sistemas robustos com LangChain e LangGraph, é imperativo revisitar a definição formal de um agente. Segundo Russell e Norvig [4], o comportamento de um agente é descrito pela **função do agente** (f), que mapeia uma sequência completa de percepções (P^*) para uma ação (A):

$$f : P^* \to A \tag{1}$$

No contexto de desenvolvimento moderno com LLMs, podemos realizar um mapeamento direto destes componentes teóricos para artefatos de software:

- **Sensores (P):** Correspondem às interfaces de entrada do sistema. Em uma aplicação baseada em LangChain, a percepção não é apenas o *prompt* do usuário, mas inclui o histórico da conversa (`chat_history`), o contexto injetado via *Retrieval-Augmented Generation* (RAG) e os resultados de execuções anteriores de ferramentas.

- **Função do Agente (f):** É implementada pelo próprio LLM (como GPT-4, Gemini

ou Llama 3). O modelo atua como o motor de raciocínio, processando a sequência de percepções para determinar a próxima etapa.

- **Atuadores (*A*):** Tradicionalmente vistos como mecanismos físicos em robótica, no desenvolvimento de software eles se manifestam através do conceito de *Tool Calling*. Ferramentas são funções Python, chamadas de API ou consultas a bancos de dados que permitem ao agente alterar o estado do ambiente, expandindo seu espaço de ação para além da geração de texto [5, 1].

1.2.1. Racionalidade versus Onisciência

Um ponto crucial para a implementação de agentes eficazes é a distinção entre racionalidade e onisciência. Um agente onisciente saberia o resultado real de suas ações, o que é impossível na realidade [4]. Um agente racional, por sua vez, escolhe a ação que maximiza o desempenho esperado, dado o seu conhecimento atual.

Os LLMs não são oniscientes; eles sofrem de alucinações e possuem conhecimento desatualizado (o *cutoff* de treinamento) [5], o que exige mecanismos de grounding externo para garantir a factibilidade das respostas. Portanto, ao projetar um sistema com LangGraph, devemos focar na **racionalidade**: desenhar o fluxo para que o agente reconheça a insuficiência de suas informações internas e utilize ferramentas (atuadores) para buscar dados externos antes de responder, maximizando assim a utilidade da resposta.

1.2.2. Taxonomia de Arquiteturas: De Cadeias a Grafos

A evolução das ferramentas de orquestração reflete a hierarquia de complexidade dos agentes. A seguir, correlacionamos os tipos de agentes clássicos com os padrões de projeto utilizados neste minicurso.

1.2.2.1. Agentes Reativos Simples e Chains

Os agentes reativos simples selecionam ações com base apenas na percepção atual, ignorando o histórico de percepções [4]. Eles operam sob regras de condição-ação diretas. No ecossistema LangChain, isso equivale a uma **Chain** básica (como a `RunnableSequence`). O sistema recebe uma entrada, processa-a através de um *prompt template* e um LLM, e gera uma saída. Não há persistência de estado complexo nem loops de decisão. É um fluxo linear e funcional, ideal para tarefas atômicas como classificação de sentimento ou tradução, mas insuficiente para diálogos longos e complexos.

1.2.2.2. Agentes Baseados em Modelo e Grafos de Estado

Para operar em ambientes parcialmente observáveis e complexos, o agente precisa manter um **estado interno** que depende do histórico de percepções e reflete como o mundo evolui independentemente do agente. Aqui reside a necessidade do **LangGraph**. Diferente de uma *chain* linear, o LangGraph introduz o conceito de **State** (um objeto de estado compartilhado e persistente). Isso permite a construção de grafos cíclicos onde o agente pode:

1. Raciocinar (“Thought”);
2. Agir (“Action”);
3. Observar o resultado (“Observation”);
4. Atualizar seu estado interno;
5. Decidir se precisa agir novamente ou responder ao usuário.

Esta estrutura cíclica implementa o padrão **ReAct** [6], que materializa o ciclo de monitoramento e correção descrito na literatura de planejamento clássico. Essa dissonância entre as abordagens é ilustrada na Figura 1.1.

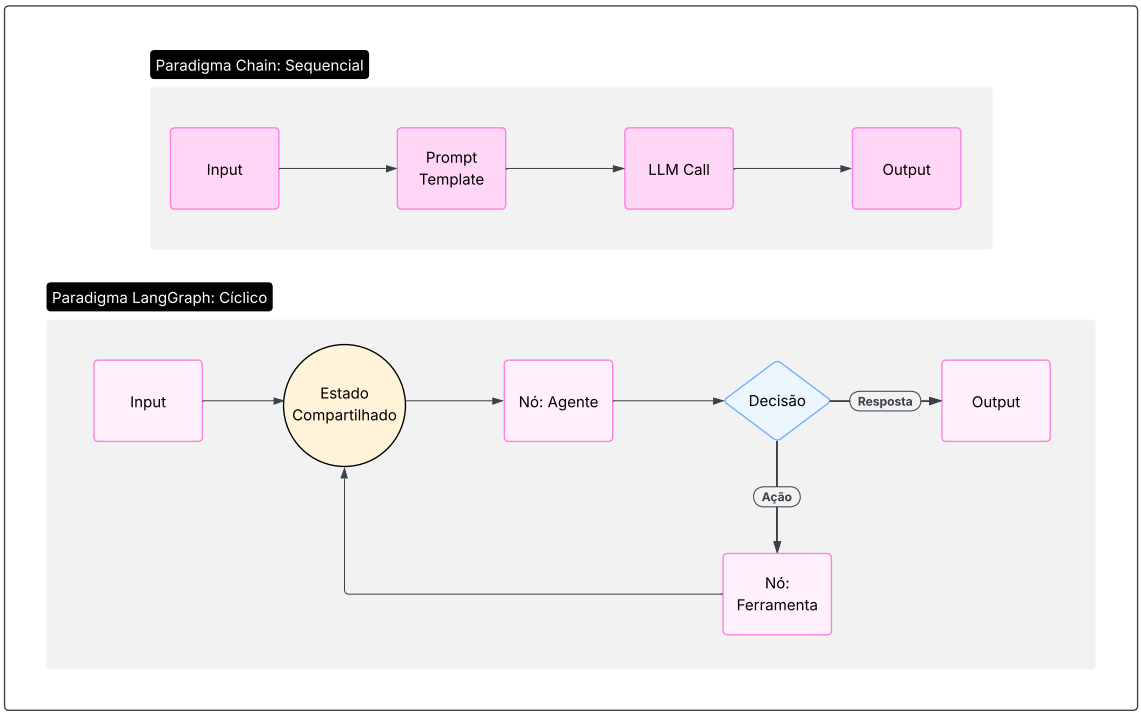

Figura 1.1: Comparação entre Agente Reativo Simples (Chain) e Agente Baseado em Modelo (Grafo)

1.2.3. O Problema da Persistência e Redutores

Um dos desafios centrais na IA clássica é o problema da representação e persistência: como descrever o que muda no mundo sem ter que listar tudo o que permaneceu inalterado? Na linguagem de planejamento PDDL (*Planning Domain Description Language*), isso é resolvido especificando apenas os efeitos de adição e remoção de uma ação [2].

O LangGraph adota uma abordagem análoga para o gerenciamento de memória através de **Redutores** (*Reducers*). Ao definirmos o estado do agente, utilizamos operações (como `operator.add`) que especificam como novas informações (ex: uma nova mensagem) devem ser mescladas ao estado existente. O framework garante que o histórico anterior persista, aplicando apenas as “deltas” (mudanças) necessárias. Isso garante a

integridade do contexto conversacional e permite funcionalidades avançadas como *Time Travel* (reverter o estado do agente para um ponto anterior) e *Human-in-the-loop* (intervenção humana antes da execução de uma ação crítica).

1.3. Implementação de Agentes com LangChain e LangGraph

Tendo estabelecido a fundamentação teórica de que um agente é uma função que mapeia um histórico de percepções para ações em um ambiente incerto, voltamo-nos agora para a implementação computacional destes conceitos. A biblioteca **LangChain** fornece as primitivas de abstração para interagir com Grandes Modelos de Linguagem (LLMs), enquanto o **LangGraph** oferece a infraestrutura de orquestração necessária para gerenciar estado e fluxos cíclicos de controle.

Esta seção disseca os componentes fundamentais para a construção de um agente, partindo da interface básica de chat até a definição de grafos de estado persistentes.

1.3.1. Componentes Atômicos: Modelos e Mensagens

Na arquitetura de agentes modernos, o LLM atua como o "cérebro" ou, mais formalmente, como o mecanismo de raciocínio. O LangChain abstrai as diferenças entre provedores (OpenAI, Google Gemini, Anthropic) através da classe `ChatModel`. Diferentemente de modelos de completação de texto antigos (que recebiam uma única *string*), os `ChatModels` operam sobre listas de **Mensagens**, que correspondem diretamente às **percepções** (P) na teoria de agentes.

As mensagens são objetos tipados que estruturam a comunicação:

- **SystemMessage:** Define a *persona*, instruções base e restrições do agente. Corresponde ao "conhecimento prévio" ou "design" do agente na teoria de Russell e Norvig.
- **HumanMessage:** Representa o *input* do usuário ou estímulo externo.
- **AIMessage:** Representa a resposta ou ação gerada pelo modelo.
- **ToolMessage:** (Crucial para agentes) Contém o resultado da execução de uma ferramenta, funcionando como uma nova percepção derivada de uma ação do agente.

Um exemplo básico de invocação de um modelo, que seria equivalente a um agente reflexo sem memória, pode ser implementado conforme a Listagem 1.1.

Este código implementa a função $f(P) \to A$, onde P é a lista `messages` e A é a `response`. Contudo, este sistema é **sem estado** (*stateless*). Cada invocação é independente, incapaz de manter um diálogo coerente ou realizar tarefas multi-etapas. Para evoluir para um agente completo, necessitamos de uma arquitetura que suporte persistência e ciclos.

```python
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

model = ChatOpenAI(model="gpt-4o")

messages = [
    SystemMessage(content="Você é um assistente especializado em física teórica."),
    HumanMessage(content="Explique a segunda lei da termodinâmica.")
]

# Execução da função do agente (f:  P* -> A)
response = model.invoke(messages)
print(response.content)
```

Listing 1.1: Invocação básica de um Modelo de Chat

1.3.2. De Cadeias (Chains) a Grafos de Estado

Tradicionalmente, o LangChain popularizou o conceito de **Chains** (Cadeias) — sequências lineares de operações (Input $\rightarrow$ Passo 1 $\rightarrow$ Passo 2 $\rightarrow$ Output). Embora úteis para pipelines de dados determinísticos, as cadeias são insuficientes para agentes autônomos.

Um agente autônomo operando no mundo real frequentemente necessita de loops de controle: ele deve tentar uma ação, observar o resultado, corrigir se necessário e tentar novamente. Isso exige uma topologia de grafo, onde nós representam etapas de computação e arestas representam transições de controle, permitindo ciclos.

O **LangGraph** formaliza essa necessidade através da classe `StateGraph`. Um grafo no LangGraph é definido por três elementos principais:

1. **State (Estado):** Um esquema de dados compartilhado que persiste ao longo de toda a execução do grafo.
2. **Nodes (Nós):** Funções Python que recebem o estado atual, processam algo (ex: chamam o LLM), e retornam uma atualização para o estado.
3. **Edges (Arestas):** Lógica de controle que determina qual nó executar a seguir, podendo ser condicional (baseado na saída do LLM).

1.3.3. O Schema de Estado e Redutores

A gestão do estado é o diferencial crítico do LangGraph. Ao contrário de simplesmente sobrescrever variáveis, o framework utiliza o conceito de **Redutores** (inspirado em conceitos de programação funcional e sistemas como Redux).

Quando um nó retorna um valor, esse valor não substitui o estado global; ele é "reduzido" (mesclado) ao estado atual. Para listas de mensagens, a operação de redução mais comum é a **adição** (`operator.add`), garantindo que o histórico da conversa seja preservado incrementalmente. A Listagem 1.2 apresenta a definição canônica de um Estado para um agente conversacional.

```python
from typing import Annotated, TypedDict
from langgraph.graph.message import add_messages

# Definição do Schema do Estado
class AgentState(TypedDict):
    # 'messages' é a chave principal.
    # A anotação 'Annotated[list, add_messages]' instrui o grafo a:
    # 1. Permitir que nós retornem apenas a NOVA mensagem.
    # 2. Usar a função 'add_messages' para anexar essa nova mensagem
    # à lista existente, em vez de substituir a lista inteira.
    messages: Annotated[list, add_messages]
```

Listing 1.2: Definição de Estado com Redutores

1.3.4. Construindo o Grafo do Agente

Com o estado definido, a construção do agente segue um padrão declarativo. O grafo deve ser inicializado, os nós adicionados e as arestas conectadas. Um padrão comum é o grafo de nó único que cicla sobre si mesmo ou termina, dependendo da decisão do modelo. A Listagem 1.3 mostra a implementação de um agente que processa mensagens e mantém memória de curto prazo.

```python
from langgraph.graph import StateGraph, START, END

# 1. Definir a função do nó (o comportamento do agente)
def chatbot_node(state: AgentState):
    # O nó recebe o estado atual (histórico completo)
    # E invoca o modelo
    response = model.invoke(state["messages"])
    # Retorna apenas a atualização (a nova mensagem AI)
    return {"messages": [response]}

# 2. Inicializar o Grafo com o Schema de Estado
builder = StateGraph(AgentState)

# 3. Adicionar o nó ao grafo
builder.add_node("chatbot", chatbot_node)

# 4. Definir o fluxo de controle (Arestas)
# O fluxo inicia no ponto de entrada 'START' e vai para o nó 'chatbot'
builder.add_edge(START, "chatbot")
# Após responder, o agente encerra a execução
builder.add_edge("chatbot", END)

# 5. Compilar o grafo
# A compilação transforma a definição em um 'Runnable' executável
agent_graph = builder.compile()
```

Listing 1.3: Construção de Grafo com Memória

Este código materializa o conceito de **Agente Reativo Simples com Estado**. A função `compile()` cria uma estrutura otimizada que gerencia a passagem de mensagens. Quando executamos este grafo, o LangGraph cria automaticamente um *snapshot* do estado inicial, executa o nó `chatbot`, aplica o redutor `add_messages` para inserir a resposta do LLM no histórico e, finalmente, transita para o estado `END`.

Esta arquitetura é a fundação sobre a qual agentes mais complexos — capazes de usar ferramentas e tomar decisões de roteamento — são construídos. A introdução de **Arestas Condicionais** é o próximo passo lógico para permitir que o agente decida *se* deve parar ou *se* deve executar uma ação (ferramenta), implementando o verdadeiro paradigma de autonomia.

1.4. Agentes Autônomos: Do Reflexo à Razão e Ação

A arquitetura de Agente Reflexivo Simples com Estado, discutida anteriormente, permite a manutenção de um histórico de conversação, mas carece de uma característica fundamental para a verdadeira autonomia: a capacidade de intervir no ambiente para alterar seu estado ou recuperar informações externas. Na taxonomia de Russell e Norvig [4], agentes racionais operam através de sensores para perceber o ambiente e atuadores para agir sobre ele. No contexto de *Large Language Models* (LLMs), os "atuadores" são materializados através do mecanismo de *Tool Calling* (chamada de ferramentas).

A transição de um sistema conversacional passivo para um agente ativo é fundamentada teórica e arquiteturalmente no paradigma **ReAct** (*Reasoning and Acting*). Yao et al. [6] demonstram que a capacidade de raciocínio de um LLM é significativamente ampliada quando o modelo é instruído a gerar traços de raciocínio (*thoughts*) intercalados com ações específicas (*actions*) e a observação de seus resultados (*observations*). Diferente da abordagem *Chain-of-Thought* (CoT) isolada, que opera apenas no espaço latente do modelo, o ReAct permite a atualização dinâmica do conhecimento do agente através da interação com ferramentas. A Figura 1.2 detalha este fluxo de execução: o modelo raciocina (Reasoning), decide atuar (Action), recebe o retorno da ferramenta (Observation) e repete o ciclo se necessário.

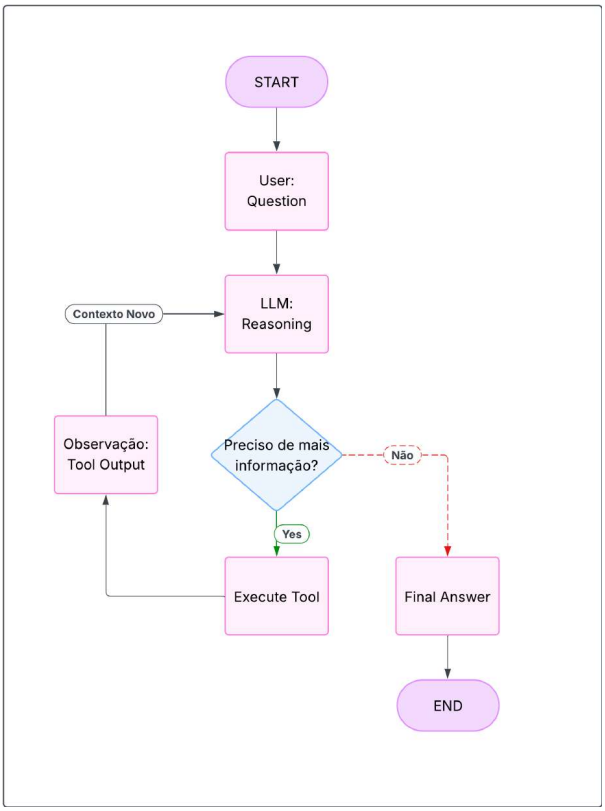

Figura 1.2: Fluxo de execução do paradigma ReAct: Raciocínio, Ação e Observação

1.4.1. Implementação do Ciclo Cognitivo no LangGraph

Para implementar essa autonomia no LangGraph, a topologia do grafo deve evoluir de uma sequência linear para um ciclo condicional. O agente deve decidir, a cada passo, se possui informações suficientes para responder ao usuário (retornando ao estado `END`) ou se necessita invocar uma ferramenta (transitando para um nó de ferramentas).

A materialização deste comportamento exige a definição de ferramentas como funções Python tipadas e a vinculação destas ao modelo de linguagem. A Listagem 1.4 apresenta a configuração essencial para um agente ReAct.

A materialização deste comportamento autônomo no LangGraph repousa sobre pilares técnicos que visam mitigar a natureza estocástica dos LLMs. O primeiro pilar é o **Vínculo de Ferramentas** (`bind_tools`). Ao injetar as assinaturas das funções diretamente no esquema da API do modelo (em vez de apenas descritas no prompt textual), reduz-se significativamente a alucinação na geração de argumentos.

Concomitantemente, a configuração da **Temperatura Determinística** (*temperature=0*) é mandatória para agentes que interagem com sistemas externos. Segundo Russell e Norvig [4], um agente racional deve selecionar a ação que maximiza sua medida de desempenho; em termos de engenharia, isso implica que, dado um mesmo estado e input, o agente deve selecionar consistentemente a mesma ferramenta.

```python
from langchain.tools import tool
from langgraph.prebuilt import ToolNode, tools_condition

# Definição de ferramentas com docstrings rigorosas
@tool
def search_database(query: str):
    """Realiza busca no banco de dados vetorial para informações contextuais."""
    # Implementação da busca...
    pass

# Vinculação das ferramentas ao modelo
# O método .bind_tools() injeta as assinaturas das funções
# no esquema da API do modelo, permitindo que o LLM
# estruture a saída JSON correta para execução.
llm_with_tools = llm.bind_tools([search_database])

def agent_node(state: AgentState):
    # O modelo decide se gera texto ou uma chamada de ferramenta
    response = llm_with_tools.invoke(state["messages"])
    return {"messages": [response]}

# Construção do Grafo ReAct
workflow = StateGraph(AgentState)
workflow.add_node("agent", agent_node)
workflow.add_node("tools", ToolNode([search_database]))

workflow.add_edge(START, "agent")

# A função tools_condition verifica se a última mensagem contém 'tool_calls'.
# Se sim, roteia para o nó "tools"; caso contrário, encerra em END.
workflow.add_conditional_edges("agent", tools_condition)
workflow.add_edge("tools", "agent")
```

Listing 1.4: Agente ReAct com Ferramentas

1.5. Sistemas Multiagentes: Colaboração e Orquestração

À medida que a complexidade das tarefas aumenta, a capacidade de um único agente (monolítico) de manter o contexto e raciocinar sobre múltiplos domínios diminui. O paradigma de Sistemas Multiagentes (SMA) aborda essa limitação decompondo problemas complexos em subproblemas tratáveis por unidades especializadas.

Teoricamente, a transição para SMA no contexto de LLMs resgata conceitos de Inteligência Artificial Distribuída (DAI). No LangGraph, isso é implementado através de arquiteturas de grafo onde o **Estado** atua como um "Quadro Negro" (*Blackboard*), um padrão clássico de engenharia de software onde múltiplos especialistas leem e escrevem em uma memória compartilhada[1].

1.5.1. Paralelização e o Padrão Map-Reduce

Enquanto o ciclo ReAct introduz a autonomia sequencial, certas classes de problemas exigem a execução simultânea de sub-tarefas independentes para otimização da latência total do sistema. No contexto de Agentes Cognitivos, a paralelização permite que um LLM decomponha uma consulta complexa em múltiplos tópicos de investigação, delegue-os a instâncias de agentes (operários) e agregue os resultados subsequentes.

A justificativa matemática para esta arquitetura reside na redução do tempo total de execução. Em um sistema sequencial,$T_{seq} = \sum_{i=1}^{n} t_i$. Em uma topologia paralela ideal,

$T_{par} \approx \max(t_i) + \delta_{\text{agregação}}$, uma lógica visualizada na topologia da Figura 1.3.

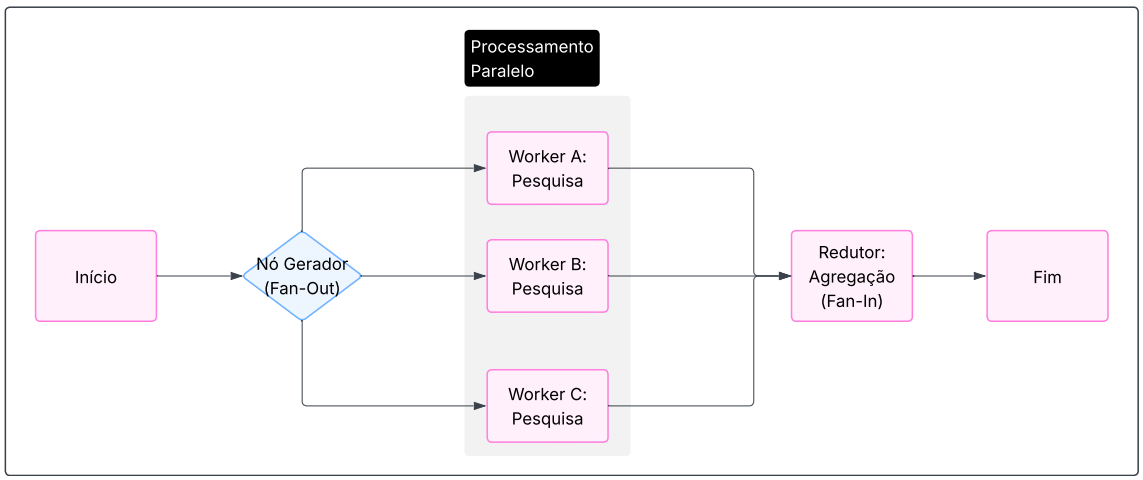

Figura 1.3: Topologia Map-Reduce para paralelização de tarefas em Agentes

Para evitar condições de corrida na escrita do estado paralelo, utiliza-se redutores especiais. A Listagem 1.5 demonstra o uso de redutores para garantir que escritas paralelas sejam fundidas corretamente.

```python
import operator
from typing import Annotated, TypedDict

class ResearchState(TypedDict):
    # O redutor operator.add garante que escritas paralelas
    # sejam fundidas em uma lista única, preservando todos os resultados.
    results: Annotated[list[str], operator.add]
```

Listing 1.5: Estado com Redutor para Paralelização

1.5.2. Arquiteturas Hierárquicas: Padrão Supervisor

A arquitetura de **Supervisor** introduz uma hierarquia onde um agente orquestrador delega tarefas a subagentes especialistas, mantendo a visão global do processo, um padrão observado em sistemas como MetaGPT e ChatDev [5]. Diferente de um roteamento estático (baseado em regras *if-else* rígidas), o roteamento em sistemas baseados em LLMs é semântico e dinâmico.Conforme ilustrado na Figura 1.4, o nó central (Supervisor) não executa o trabalho, mas atua como um roteador semântico, encaminhando a solicitação para o especialista mais adequado (Matemático ou Informações).

Para implementar este padrão de forma determinística, utiliza-se a técnica de *Structured Output* (Saída Estruturada). A Listagem 1.6 demonstra a definição do esquema de decisão e o nó supervisor.

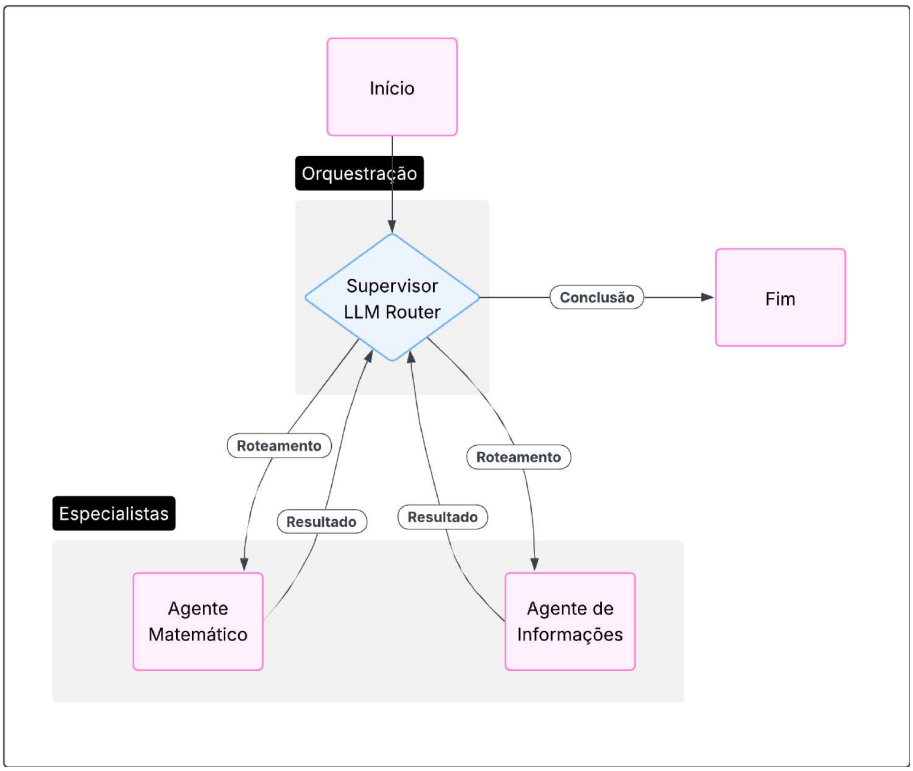

Figura 1.4: Arquitetura Hierárquica: O Supervisor atua como orquestrador central

```python
from typing import Literal
from pydantic import BaseModel

class RouteResponse(BaseModel):
    next: Literal["matematico", "informacoes", "finalizar"]

def supervisor_node(state: Estado):
    # O método .with_structured_output() acopla o esquema Pydantic
    # garantindo que o retorno seja um objeto Python validado
    chain = prompt | llm.with_structured_output(RouteResponse)
    decision = chain.invoke(state)

    # A decisão é persistida no estado
    return {"proximo_agente": decision.next}
```

Listing 1.6: Nó Supervisor com Saída Estruturada

1.6. Persistência de Estado e Memória de Longo Prazo

Um desafio fundamental na engenharia de agentes sobre LLMs é a natureza *stateless* (sem estado) dos modelos de fundação: a função $f(prompt) \rightarrow texto$ não retém memória de invocações anteriores. Contudo, aplicações reais exigem continuidade, permitindo que o usuário retome conversas, faça referências a interações passadas e mantenha transações de longa duração.

No framework LangGraph, a persistência não é apenas um registro de logs, mas

um mecanismo ativo de **Checkpointing**. O *checkpointer* intercepta o fluxo de execução a cada transição de nó, serializando o objeto `State` e armazenando-o em uma base durável (como SQLite ou PostgreSQL). Isso permite não apenas a memória conversacional, mas também funcionalidades avançadas como "Time Travel" (reverter o estado para um ponto anterior) e "Human-in-the-loop" (pausar para aprovação humana e retomar).

A implementação da persistência exige a configuração de um gerenciador de memória no momento da compilação do grafo e a utilização de identificadores de sessão (`thread_id`) durante a invocação, conforme a Listagem 1.7.

```
from langgraph.checkpoint.memory import MemorySaver

# Inicialização do Checkpointer
# Em produção, substitui-se MemorySaver (RAM) por AsyncSqliteSaver
memory = MemorySaver()

# O parâmetro 'checkpointer' habilita o versionamento do estado
app = workflow.compile(checkpointer=memory)

# Configuração da Sessão
# O 'thread_id' atua como chave primária para recuperação do contexto
config = {"configurable": {"thread_id": "sessao_usuario_123"}}

# Invocação
# O grafo carregará automaticamente as mensagens prévias associadas ao
    ID
response = app.invoke(inputs, config=config)
```

Listing 1.7: Compilação com Persistência

Tecnicamente, o *checkpointer* resolve o problema da **persistência do planejamento** descrito por Ghallab et al. [2]. Ele assegura que o plano parcial construído pelo agente (o conjunto de mensagens e resultados de ferramentas acumulados) sobreviva a interrupções do sistema ou à latência entre interações do usuário.

1.7. Estudo de Caso: Agente de E-commerce

Consolidando os conceitos apresentados, esta seção demonstra um Assistente de E-commerce que executa alterações de estado via banco de dados SQL. A inclusão da camada de persistência transcende a mera funcionalidade de histórico, habilitando capacidades cognitivas e operacionais avançadas no sistema:

- **Conversação Multi-Turno:** O agente mantém acesso a fatos e entidades citados em interações anteriores, simulando uma memória episódica de curto prazo.
- **Human-in-the-loop:** O sistema pode interromper sua execução antes de ações sensíveis e aguardar aprovação humana.
- **Depuração Temporal:** Permite aos engenheiros inspecionar estados passados da conversa e "rebobinar" a execução para testar caminhos alternativos de raciocínio (*Time Travel*).

A Figura 1.5 detalha a arquitetura completa deste estudo de caso, demonstrando a interação entre os agentes especialistas e o banco de dados.

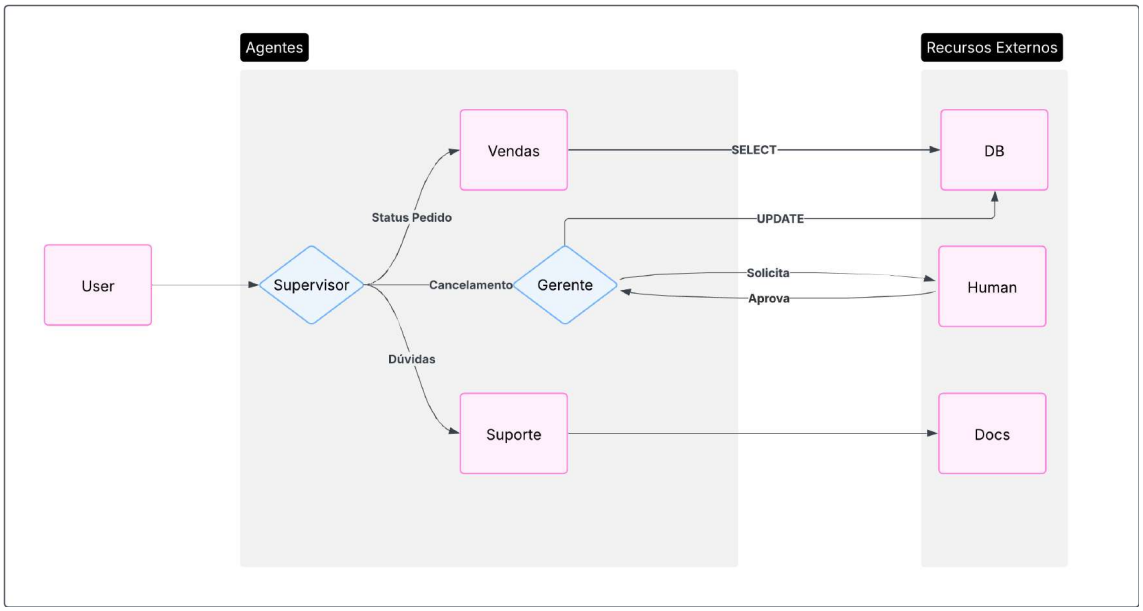

Figura 1.5: Arquitetura do Agente de E-commerce com persistência e ferramentas SQL

A integração com banco de dados SQL via *Tool Calling* exige rigor na definição das ferramentas. O LLM deve ser capaz de inferir corretamente os parâmetros da consulta. A Listagem 1.8 ilustra como uma ferramenta de consulta SQL é exposta ao agente.

Neste cenário, a persistência é vital. Se um usuário pergunta "Onde está meu pedido?" (acionando o agente de Vendas) e, em seguida, diz "Cancele-o" (acionando o agente de Gerência), o sistema deve manter o contexto de *qual* pedido está sendo discutido durante a transição entre os especialistas.

```python
import sqlite3
from langchain.tools import tool

@tool
def consultar_pedidos(cliente: str) -> list:
    """
    Consulta o status dos pedidos de um cliente específico no banco de dados.
    Retorna uma lista de tuplas contendo produto, status e valor.
    """
    conn = sqlite3.connect('ecommerce_v2.db')
    cursor = conn.cursor()
    cursor.execute(
        "SELECT produto, status, valor FROM pedidos WHERE cliente = ?",
        (cliente,)
    )
    return cursor.fetchall()

# A docstring é consumida pelo LLM para entender a semântica da ferramenta.
# A tipagem (cliente: str) define o esquema JSON esperado na chamada.
```

Listing 1.8: Ferramenta SQL com Tipagem Rigorosa

1.8. Observabilidade e Avaliação Sistemática

A transição de um protótipo em *notebook* para um sistema de produção impõe um desafio metodológico: a natureza estocástica dos *Large Language Models*. Diferente de software determinístico, onde a função $f(x) = y$ é constante, em agentes generativos $P(y|x)$ é uma distribuição de probabilidade. Isso torna testes unitários tradicionais baseados em asserções exatas (`assert result == expected`) insuficientes.

Para mitigar a incerteza e garantir a confiabilidade do agente, a engenharia deve adotar práticas de **Observabilidade** (Tracing) e **Avaliação Baseada em Modelo** (*LLM-as-a-Judge*).

1.8.1. Tracing de Cadeias e Grafos

A observabilidade em sistemas baseados em grafos (como LangGraph) não se resume ao registro de *logs* textuais. É necessário capturar a **Árvore de Execução** completa, detalhando a latência, o consumo de tokens e, crucialmente, os estados intermediários em cada transição de nó, conforme ilustrado na Figura 1.6.

Ferramentas de rastreamento, como o LangSmith, permitem decompor a falha na resposta final em seus componentes causais primários, indo além da análise superficial do texto gerado. Através da inspeção do *trace*, é possível discernir se o erro originou-se de uma **Recuperação Falha**, onde os documentos relevantes não foram encontrados no banco vetorial; de um **Raciocínio Incorreto**, onde o modelo alucinou conclusões falsas mesmo possuindo os dados corretos no contexto; ou de uma **Falha na Ferramenta**, decorrente de exceções de *runtime* ou erros de sintaxe em consultas SQL e APIs externas.

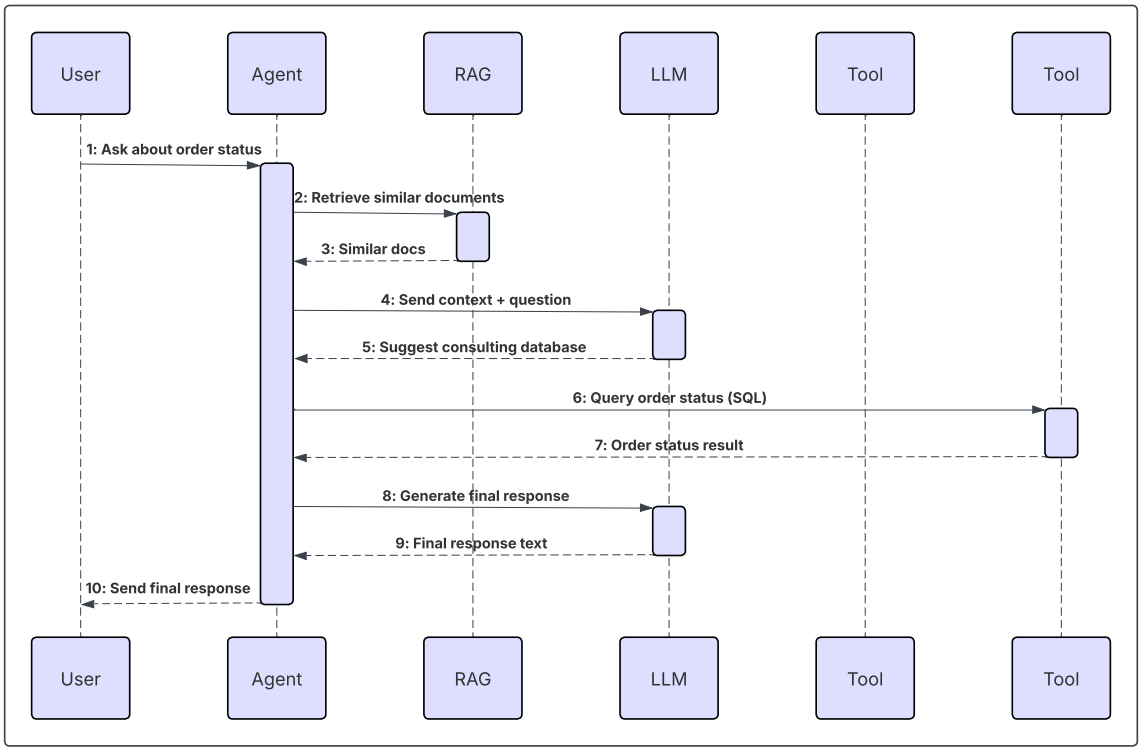

Figura 1.6: Visualização de trace de execução com decomposição temporal

1.8.2. Avaliação Automatizada (LLM-as-a-Judge)

A avaliação escalável de agentes conversacionais utiliza um LLM mais robusto (o "Juiz") para avaliar as saídas do agente em desenvolvimento (o "Aluno").Esta técnica, conhecida como avaliação subjetiva baseada em modelo, tem sido adotada para mitigar os alto custos da avaliação humana [5].

A implementação de um avaliador no ecossistema LangChain envolve a criação de um *dataset* de referência (perguntas e respostas ideais) e a execução de uma cadeia de avaliação. A Listagem 1.9 demonstra como definir um critério de correção factual.

A justificativa técnica para o uso de `labeled_criteria` em vez de comparação de strings (como distância de Levenshtein) reside na capacidade do LLM de entender equivalência semântica.

```python
from langsmith import Client
from langchain.evaluation import load_evaluator

# Definição do critério de avaliação
# O avaliador "labeled_criteria"compara a previsão com a referência
evaluator = load_evaluator("labeled_criteria", criteria="correctness")

def avaliar_agente(run, example):
    # Extrai a resposta do agente e a resposta esperada do dataset
    predicao = run.outputs["messages"][-1].content
    referencia = example.outputs["resposta_esperada"]

    # O LLM "Juiz"analisa semanticamente se a predição atende à
       referência
    score = evaluator.evaluate_strings(
        prediction=predicao,
        reference=referencia,
        input=example.inputs["pergunta"]
    )
    return {"key": "correctness", "score": score["score"]}

# A execução deste script sobre um dataset gera métricas quantitativas
# (ex:  85% de acurácia) permitindo comparar diferentes versões do
   grafo
```

Listing 1.9: Avaliador de Correção Factual

1.9. Arquitetura de Implantação

A etapa final do ciclo de vida do agente é a exposição de sua lógica como um serviço consumível por interfaces de usuário (Frontend Web, Mobile ou WhatsApp). A arquitetura recomendada segue o padrão de **Microsserviços Assíncronos** alinhando-se ao paradigma emergente de Agent-as-a-Service (AaaS), onde agentes encapsulam capacidades cognitivas expostas via APIs padronizadas [1].

O objeto compilado do LangGraph (`app = graph.compile()`) adere à interface `Runnable`, o que facilita sua integração com servidores web assíncronos como FastAPI. O framework **LangServe** automatiza a criação de endpoints RESTful padronizados, suportando não apenas requisições síncronas, mas também *streaming* de tokens, essencial para a experiência do usuário em chatbots (efeito de digitação). O fluxo completo de dados nesta arquitetura de microsserviços é apresentado na Figura 1.7.

A Listagem 1.10 ilustra a conversão do grafo desenvolvido nas seções anteriores em uma API de produção.

Esta arquitetura de microsserviços permite que o agente seja escalado horizontalmente, com múltiplas instâncias do servidor processando requisições em paralelo, garantindo alta disponibilidade e baixa latência mesmo sob carga elevada.

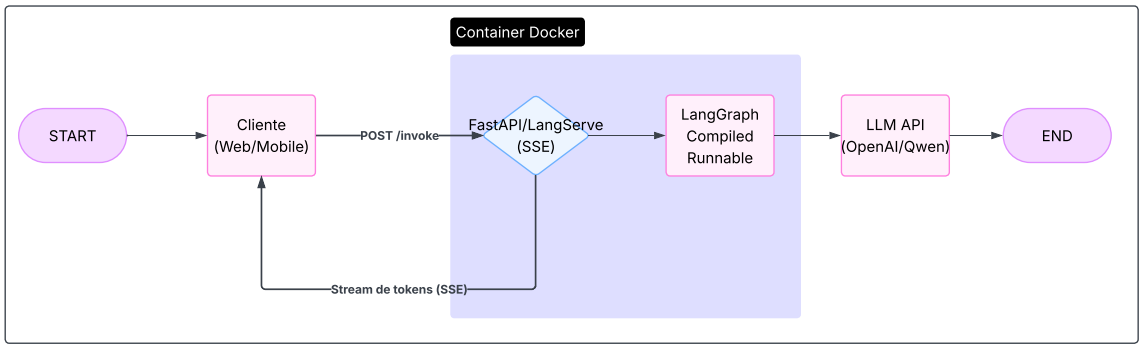

Figura 1.7: Diagrama de Implantação e fluxo de dados via Microsserviços

```python
from fastapi import FastAPI
from langserve import add_routes

app = FastAPI(
    title="Servidor do Agente de E-commerce",
    version="1.0",
    description="API para interação com Agente Multiagente via
        LangGraph"
)

# Criação automática de rotas:
# POST /agente/invoke - Execução síncrona
# POST /agente/stream - Streaming de tokens e atualizações de estado
# POST /agente/batch - Processamento em lote
add_routes(app, graph_app, path="/agente")

if __name__ == "__main__":
    import uvicorn
    # Execução do servidor em host local
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Listing 1.10: Exposição via LangServe

1.10. Considerações Finais

A evolução dos sistemas conversacionais, partindo de cadeias lineares simples para arquiteturas multiagente complexas baseadas em grafos, representa uma mudança de paradigma na Inteligência Artificial Aplicada: a transição da "Engenharia de Prompt"para a "Engenharia de Fluxo"(Flow Engineering). Neste capítulo, demonstramos que a construção de agentes robustos exige a integração de conceitos fundamentais da Ciência da Computação:

1. **Racionalidade e Autonomia:** Via ciclo ReAct.
2. **Sistemas Distribuídos e Coordenação:** Modelados via arquiteturas de Supervisor e Paralelismo
3. **Persistência:** Via Checkpointers e Banco de Dados

Desafios persistentes incluem **Latência e Custo**, onde arquiteturas multiagente multiplicam chamadas ao LLM, e **Segurança e Alinhamento**, garantindo que agentes autônomos não executem ações destrutivas.

Apesar dos avanços significativos na orquestração de agentes, pesquisadores e engenheiros enfrentam desafios persistentes para a adoção massiva dessa tecnologia. O primeiro obstáculo reside no binômio Latência e Custo; arquiteturas multiagente multiplicam exponencialmente o número de chamadas ao LLM para uma única resolução. O uso de modelos menores e especializados (Small Language Models), ajustados via Fine-Tuning para tarefas específicas (como geração de SQL), apresenta-se como uma solução viável para a sustentabilidade econômica. Simultaneamente, impõe-se o desafio da Segurança e Alinhamento [5]. Garantir que um agente autônomo com acesso a ferramentas de escrita não execute ações destrutivas ou indesejadas requer o desenvolvimento de guardrails determinísticos e conformidade ética rigorosa [3] que operem externamente ao espaço latente do modelo, assegurando que a autonomia do agente permaneça circunscrita às regras de negócio definidas.

Conclui-se, portanto, que o desenvolvimento de Agentes de IA é uma disciplina interdisciplinar, exigindo rigor tanto no design algorítmico quanto na modelagem do conhecimento e na arquitetura de software.

Agradecimentos

O presente trabalho foi financiado parcialmente pela SSP/PI, com apoio da Fundação Cultural e de Fomento à Pesquisa, Ensino, Extensão e Inovação (FADEX). Agradecemos também à equipe técnica da SSP e Trinhub, pelo apoio e disponibilidade constante em conversar com os pesquisadores do Departamento de Computação da UFPI no desenvolver das habilidades da equipe nas tecnologias adotadas nesse minicurso.

Referências

[1] Hana Derouiche, Haithem Mezni, and Zaki Brahmi. Agentic ai frameworks: Architectures, protocols, and design challenges. *arXiv preprint arXiv:2508.10146*, 2025.

[2] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 1998.

[3] Satyadhar Joshi. Advancing innovation in financial stability: A comprehensive review of ai agent frameworks, challenges and applications. *World Journal of Advanced Engineering Technology and Sciences*, 14(2):117–126, 2025.

[4] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 4th edition, 2020.

[5] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024.

[6] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Wen, and Mike Lewis. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Capítulo 2

Perfilamento e Visualização da Escalabilidade de Aplicações Paralelas com o Parallel Scalability Suite

Felipe H. Santos-da-Silva, João B. Fernandes, Anderson B. N. da Silva, Ítalo A. S. Assis e Samuel Xavier-de-Souza

Abstract

Supercomputers play an important role in scientific and technological advances; however, many parallel applications fail to fully exploit their computational capacity. Traditional development tools for high-performance computing often focus on improving performance in specific configurations, without providing adequate support for analyzing how programs behave across different computing environments. In this context, the Parallel Architectures for Signal Processing Laboratory at UFRN developed the Parallel Scalability (PaScal) Suite, a toolset designed to analyze the scalability of parallel applications. The PaScal Suite enables the evaluation of scalability trends and the prediction of program behavior under different hardware configurations and workloads. The suite integrates the PaScal Analyzer and PaScal Viewer, which support execution, data collection, and visualization of performance metrics, helping developers identify bottlenecks and critical regions in parallel programs. This tutorial presents fundamental concepts of scalability and performance evaluation, along with practical activities demonstrating how the tool can be used for profiling and analyzing parallel applications in high-performance computing environments.

Resumo

Supercomputadores são fundamentais para o avanço científico e tecnológico, porém muitas aplicações paralelas não conseguem explorar plenamente sua capacidade de processamento. Ferramentas tradicionais de apoio ao desenvolvimento para sistemas de alto desempenho costumam focar na aceleração em configurações específicas, sem oferecer suporte adequado para analisar o comportamento do programa em diferentes ambientes computacionais. Nesse contexto, o Laboratório de Arquiteturas Paralelas para Processamento de Sinais da UFRN desenvolveu o Parallel Scalability (PaScal) Suite, um conjunto

de ferramentas voltado à análise da escalabilidade de aplicações paralelas. O PaScal Suite permite avaliar tendências de escalabilidade e prever o comportamento de programas em diferentes configurações de hardware e cargas de trabalho. O conjunto integra as ferramentas PaScal Analyzer e PaScal Viewer, que auxiliam na execução, coleta e visualização de métricas de desempenho, facilitando a identificação de gargalos e pontos críticos em aplicações paralelas. Neste minicurso, são apresentados os conceitos fundamentais de escalabilidade e métodos de avaliação de desempenho, além de atividades práticas que demonstram o uso da ferramenta no perfilamento e análise de aplicações paralelas em ambientes de computação de alto desempenho.

2.1. Análise de Desempenho e Escalabilidade

A avaliação de programas em Computação de Alto Desempenho (HPC) é composta por múltiplas métricas. Embora o objetivo primário seja frequentemente a melhoria do **desempenho absoluto**, ou seja, a redução do tempo de execução ou o aumento da vazão de tarefas, essa métrica isolada não é suficiente para caracterizar a qualidade de uma solução paralela. É fundamental analisar também a **escalabilidade** e a **eficiência**, que são medidas que descrevem como o desempenho se comporta mediante a variação dos recursos computacionais ou do tamanho do problema.

Portanto, enquanto o desempenho absoluto mede a velocidade em uma configuração específica, a escalabilidade mede a capacidade do sistema de sustentar o crescimento desse desempenho.

Para ilustrar a relação entre esses conceitos, considere a seguinte analogia: imagine uma cozinha de restaurante com uma equipe responsável por lavar pratos.

O **desempenho** pode ser observado na rapidez com que a tarefa é executada no estado atual. Se apenas uma pessoa está lavando os pratos, o desempenho é medido pela quantidade de pratos lavados por minuto. Se essa pessoa trabalha rapidamente, temos uma alta produtividade sequencial.

Já a **escalabilidade** refere-se ao comportamento dessa produtividade quando a equipe é expandida. Imagine que a demanda aumente e mais pessoas sejam contratadas. Se a equipe conseguir dividir o trabalho de forma equilibrada, onde cada novo membro adiciona uma capacidade de trabalho proporcional sem atrapalhar os demais, dizemos que o sistema possui boa escalabilidade.

Por outro lado, se a adição de novos membros fizer com que eles esbarrem uns nos outros, disputem espaço na pia ou aguardem por ferramentas compartilhadas, a eficiência do grupo cairá. No contexto computacional, isso representa o *overhead* de paralelização: o tempo gasto com comunicação e sincronização entre fluxos de execução que não contribui diretamente para a solução do problema, limitando o ganho de desempenho esperado.

2.1.1. Speedup

Um dos métodos fundamentais para quantificar o ganho de desempenho em processamento paralelo é o ***speedup***. Esta métrica estabelece a relação entre o tempo de execução da versão sequencial de um algoritmo e o tempo de sua execução paralela. Sendo $T_{\text{sequencial}}$ o tempo necessário para executar a tarefa em um único núcleo e T_{paralelo} o tempo

utilizando P núcleos, o *speedup* (S) é definido pela razão:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}} \tag{1}$$

Teoricamente, o cenário ideal é denominado ***speedup* linear**, onde $S = P$. Nesse caso ideal, se o trabalho for dividido perfeitamente e sem custos adicionais, o programa paralelo seria exatamente P vezes mais rápido que o sequencial.

No entanto, na prática, obter essa aceleração linear é raro. Fatores intrínsecos à paralelização, como a sobrecarga (*overhead*) de comunicação, a sincronização entre *threads* e seções críticas que exigem mecanismos de exclusão mútua, tendem a degradar o desempenho. Esses elementos limitam o ganho, resultando frequentemente em um *speedup* sublinear ($S < P$).

2.1.2. Eficiência e Escalabilidade

A **eficiência** é a métrica que avalia o custo-benefício da paralelização, quantificando o quão efetivamente os recursos computacionais (como núcleos) são aproveitados para gerar o *speedup*. Matematicamente, ela é definida como a razão entre a aceleração obtida (S) e o número de processadores utilizados (P):

$$E = \frac{S}{P}. \tag{2}$$

Uma eficiência de 100% (ou 1) indicaria o cenário ideal onde o *speedup* é linear, significando que não há desperdício de recursos. Contudo, o valor de E tende a diminuir conforme P aumenta.

Retomando a analogia dos pratos (Seção 2.1), isso ocorre quando a adição de pessoas na cozinha gera conflitos por espaço. Em sistemas computacionais, aumentar apenas os recursos sem aumentar o tamanho do problema frequentemente prejudica a eficiência. Máquinas massivas, como supercomputadores, exigem problemas de grande escala para justificar o uso de seus milhares de núcleos e manter níveis elevados de eficiência.

A **escalabilidade**, portanto, é essencial para avaliar se um algoritmo consegue sustentar sua eficiência à medida que os recursos aumentam[1]. A análise de escalabilidade é mais complexa que a de desempenho pontual, pois exige a execução e coleta de dados de múltiplas configurações. Além disso, há uma escassez de ferramentas que forneçam artefatos visuais nativos e específicos para essa análise, muitas vezes entregando dados excessivamente detalhados que dificultam a interpretação rápida das tendências de escala.

A Figura 2.1 ilustra a discrepância entre o potencial teórico e a realidade prática. Ela compara o desempenho de pico teórico com os resultados dos benchmarks *HPL* e *HPGC* para os 10 principais supercomputadores (2018-2022). Observa-se que, enquanto o *HPL* (verde) se aproxima mais do pico teórico (laranja), o *HPGC* (azul), que representa problemas mais complexos e irregulares, apresenta resultados consideravelmente inferiores. Isso evidencia a dificuldade de manter a eficiência em problemas que exigem comunicação intensiva, mesmo em máquinas de ponta.

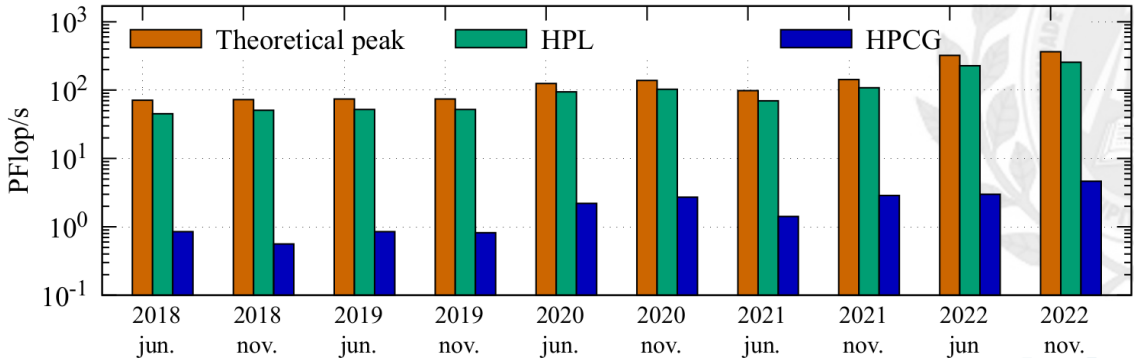

Figura 2.1. Comparação do desempenho médio dos 10 principais supercomputadores da lista Top500 (2018-2022) em termos de pico teórico, HPL e HPGC. O desempenho teórico de pico é consistentemente maior que os resultados medidos pelos benchmarks HPL e HPGC

2.1.3. Desafios na Escalabilidade de Sistemas Modernos

A escalabilidade paralela tem se tornado uma preocupação crítica devido à crescente demanda por desempenho. Com a onipresença de arquiteturas *multicore*, a eficiência no uso de múltiplos núcleos tornou-se determinante para lidar com volumes de dados massivos. Em áreas como inteligência artificial, *big data* e simulações científicas, a capacidade de escalar recursos afeta diretamente a produtividade e o custo-benefício. No entanto, observa-se ainda uma escassez de ferramentas dedicadas especificamente à análise e otimização da escalabilidade, o que eleva a complexidade desse processo.

As limitações para atingir essa escalabilidade ideal originam-se, principalmente, da comunicação, sincronização e desbalanceamento de carga. A comunicação excessiva entre núcleos gera um *overhead* significativo, degradando a eficiência global. Simultaneamente, a necessidade de sincronização frequente obriga *threads* a entrarem em períodos de inatividade (espera), subutilizando o hardware disponível.

Além disso, mesmo quando comunicação e sincronização são minimizadas, o desbalanceamento de carga pode tornar-se o principal responsável pela queda no *speedup*. Esse fenômeno ocorre quando diferentes unidades de processamento recebem quantidades desiguais de trabalho. O resultado é que algumas unidades de processamento finalizam suas tarefas prematuramente e permanecem ociosos aguardando os demais, reduzindo o *speedup* geral. Identificar e mitigar esse ciclo de ineficiências, através de técnicas como escalonamento de trabalho e ocultação de latência, é o desafio central na otimização de aplicações paralelas.

2.1.4. Lei de Amdahl

Em 1967, Gene Amdahl fez uma observação que ficaria conhecida como a **Lei de Amdahl** [2]. A lei estabelece um limite teórico no *speedup* que pode ser alcançado com a paralelização, considerando que uma fração do código de um programa é "inerentemente sequencial" e, portanto, não pode ser paralelizada. De acordo com Amdahl, mesmo que uma grande parte do código seja paralelizada, o tempo de execução total será limitado pela parte que permanece sequencial.

Suponhamos que 90% de um programa possa ser paralelizado e que o tempo de execução em um único processador seja $T_{\text{sequencial}} = 20$ segundos. Usando a Lei de Amdahl, podemos calcular o tempo de execução paralelizado da seguinte forma:

$$T_{\text{paralelo}} = 0,9 \times \frac{T_{\text{sequencial}}}{P} + 0,1 \times T_{\text{sequencial}} = \frac{18}{P} + 2, \tag{3}$$

onde P representa o número de processadores. E o *speedup* será:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}} = \frac{20}{\frac{18}{P} + 2}. \tag{4}$$

Conforme P aumenta, o *speedup* se aproxima de um valor máximo. Neste exemplo, mesmo com 1000 processadores, o *speedup* não excede 10. Mais genericamente, se uma fração s do programa for intrinsecamente sequencial, o limite máximo de *speedup* será dado por $S \leq \frac{1}{s}$.

2.1.5. Lei de Gustafson

Embora a **Lei de Amdahl** seja amplamente utilizada para modelar o speedup em sistemas paralelos, ela pressupõe que o tamanho do problema permanece constante à medida que o número de processadores aumenta (escalabilidade forte). Para abordar essa limitação, John Gustafson propôs, em 1988, uma alternativa chamada **Lei de Gustafson**, que considera o aumento do tamanho do problema conforme mais processadores são adicionados.

A Lei de Gustafson parte da premissa de que o tempo de execução se mantém constante enquanto aumentamos o tamanho do problema proporcionalmente ao número de processadores P. Normalizando o tempo de execução paralelo para 1, o trabalho total realizado seria a soma da parte serial s e da parte paralela, que agora é multiplicada por P (já que há mais processadores trabalhando no mesmo tempo). Assim, o *speedup* escalado (S_{scaled}) é calculado como:

$$S_{scaled} = s + P \times (1 - s). \tag{5}$$

Rearranjando os termos algebricamente, chegamos à forma mais comum da equação apresentada por Gustafson [3]:

$$S_{scaled} = P - (P - 1) \times s, \tag{6}$$

onde P é o número de processadores (frequentemente denotado como N na literatura original de Gustafson) e s é a fração sequencial do programa.

Suponhamos que $s = 0,1$, ou seja, 10% do código é sequencial. Para $P = 100$ processadores, o *speedup* S pode ser calculado como:

$$S = 100 - (100 - 1) \times 0,1 = 100 - 99 \times 0,1 = 100 - 9,9 = 90,1. \tag{7}$$

Esse resultado mostra que o *speedup* é de $90,1$, significativamente maior do que o limite previsto pela Lei de Amdahl.

2.1.6. Tipos de Escalabilidade

Em 2.1.2 fomos apresentados rapidamente ao conceito de escalabilidade. Definindo de maneira mais precisa: suponha que um programa paralelo seja executado com um número fixo de *threads* e um problema de tamanho fixo, e que tenha uma eficiência inicial E. Agora, se aumentarmos o número de *threads* e também ajustarmos o tamanho do problema de maneira adequada, podemos verificar se a eficiência E é mantida.

Para ilustrar matematicamente a escalabilidade, considere um modelo hipotético simplificado onde o tempo de execução é linearmente proporcional ao tamanho do problema n (ou seja, $T_{\text{sequencial}} = n$) e o tempo paralelo possui um custo fixo de sobrecarga igual a 1 unidade de tempo (ou seja, $T_{\text{paralelo}} = \frac{n}{P} + 1$).

Neste cenário hipotético, a eficiência E seria descrita por:

$$E = \frac{S}{P} = \frac{T_{\text{sequencial}}}{P \times T_{\text{paralelo}}} = \frac{n}{P(\frac{n}{P}+1)} = \frac{n}{n+P}. \tag{8}$$

Essa equação demonstra que, para manter E constante à medida que P aumenta, n também precisa aumentar. Para determinar se o programa é escalável, aumentamos o número de processadores por um fator k e verificamos o fator x pelo qual o tamanho do problema deve aumentar. O número de processadores se torna kP e o tamanho do problema se torna xn. A equação que devemos resolver para x é:

$$E = \frac{n}{n+P} = \frac{xn}{xn+kP}. \tag{9}$$

Quando $x = k$, a eficiência E permanece inalterada. Podemos simplificar a equação para:

$$\frac{kn}{kn+kP} = \frac{kn}{k(n+P)} = \frac{n}{n+P}. \tag{10}$$

Ou seja, se aumentarmos o tamanho do problema n na mesma proporção que aumentarmos o número de *threads* P, a eficiência se mantém constante (escalabilidade fraca).

2.1.7. Análise de Eficiência e Escalabilidade Paralela

Um programa é dito **fortemente escalável** quando consegue manter a eficiência fixa mesmo sem aumentar o tamanho do problema. Isso significa que, ao adicionar mais processadores, o programa mantém uma alta eficiência sem precisar aumentar proporcionalmente a carga de trabalho.

Por outro lado, um programa é considerado **fracamente escalável** quando consegue manter a eficiência fixa à medida que o tamanho do problema aumenta na mesma proporção que o número de processadores. Isso quer dizer que, embora mais recursos

sejam adicionados, o problema precisa crescer proporcionalmente para que a eficiência não decresça.

2.1.8. Avaliando a Escalabilidade com Tabelas

Uma forma de avaliar a escalabilidade é utilizar tabelas que correlacionam o número de núcleos com o tamanho do problema. Considere agora uma aplicação real cuja complexidade de tempo sequencial seja quadrática ($T_{\text{sequencial}} = n^2$) e a execução paralela sofra com uma sobrecarga logarítmica de comunicação ($T_{\text{paralelo}} = \frac{n^2}{P} + \log_2 P$).

Os tempos de execução para diversas configurações são apresentados na Tabela 2.1.

Núcleos (P)	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
1	100.00	400.00	1600.00	6400.00	25600.00	102400.00
2	51.00	201.00	803.00	3201.00	12801.00	51201.00
4	27.00	102.00	402.00	1602.00	6402.00	25602.00
8	15.50	53.00	203.00	803.00	3203.00	12803.00
16	10.25	29.00	104.00	404.00	1604.00	6404.00
32	8.13	17.50	55.00	205.00	805.00	3205.00
64	7.56	12.25	31.00	106.00	406.00	1606.00
128	7.78	10.13	19.50	57.00	207.00	807.00

Tabela 2.1. Tempos de execução (em segundos) variando núcleos (P) e tamanho (n).

A equação que representa o tempo de execução sequencial é $T_{\text{sequencial}} = n^2$, onde n é o tamanho do problema. Já o tempo de execução paralelo é $T_{\text{paralelo}} = \frac{n^2}{P} + \log_2 P$, onde P representa o número de núcleos utilizados.

Uma vez que os tempos de execução foram obtidos, como descrito na Tabela 2.1, podemos agora calcular o *speedup* para cada combinação. O *speedup* é definido como:

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}}. \tag{11}$$

Segue a tabela que representa os valores para *speedup*

Após o cálculo do *speedup*, calculamos a eficiência E:

$$E = \frac{S}{P}. \tag{12}$$

A tabela abaixo representa o cálculo de eficiência para cada combinação possível de configuração.

Após a obtenção da Tabela 2.3, podemos avaliar a escalabilidade do programa. Observa-se que, para tamanhos de problema menores, como $1x$, a eficiência apresenta uma queda acentuada à medida que o número de núcleos aumenta. Esse comportamento vai contra o conceito de escalabilidade forte.

Núcleos (P)	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.96	1.99	2.00	2.00	2.00	2.00
4	3.70	3.92	3.98	4.00	4.00	4.00
8	6.45	7.35	7.80	7.97	7.99	8.00
16	9.76	13.79	15.38	15.84	15.96	16.00
32	12.31	22.86	29.09	31.22	31.80	31.96
64	13.22	32.65	51.61	60.48	63.05	63.76
128	12.85	39.51	82.05	112.28	123.67	126.89

Tabela 2.2. Speedups obtidos para diferentes números de núcleos e tamanhos de problema.

Núcleos (P)	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.98	1.00	1.00	1.00	1.00	1.00
4	0.93	0.98	1.00	1.00	1.00	1.00
8	0.81	0.92	0.98	1.00	1.00	1.00
16	0.61	0.86	0.96	0.99	1.00	1.00
32	0.38	0.71	0.91	0.98	0.99	1.00
64	0.21	0.51	0.81	0.94	0.98	1.00
128	0.10	0.31	0.64	0.88	0.97	0.99

Tabela 2.3. Eficiências obtidas para diferentes números de núcleos e tamanhos de problema.

No entanto, ao analisar a escalabilidade de forma linear, isto é, aumentando o tamanho do problema proporcionalmente ao número de núcleos, percebe-se que a eficiência se mantém mais estável, próxima de 1. Isso indica que o programa consegue distribuir melhor a carga de trabalho em cenários com maior demanda de processamento, mostrando uma capacidade de escalabilidade sob essas condições.

Portanto, podemos concluir que o programa exibe escalabilidade fraca para problemas de menor dimensão, mas aproxima-se de uma escalabilidade forte quando o tamanho do problema cresce na mesma proporção que o número de núcleos.

A análise por tabelas ainda é viável com uma pequena contagem de núcleos e taxas crescentes quadraticamente, agora para análises mais detalhadas, o uso de tabelas começa a ser um problema, a Tabela 2.4 ilustra uma parte da Tabela 2.3 agora com taxas lineares.

À medida que o detalhamento nas análises cresce, maiores ficam as tabelas. Vale ressaltar que estas tabelas não são geradas automaticamente por ferramentas de perfilamento tradicionais; essas ferramentas geralmente fornecem apenas os dados para uma única célula da tabela por vez. O desenvolvedor que se preocupa com a escalabilidade é deixado sozinho para rodar, coletar e de alguma forma visualizar os pontos críticos e gargalos da aplicação.

Embora tabelas sejam frequentemente utilizadas para essa tarefa, elas podem ocul-

Núcleos (P)	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
3	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
5	0.87	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
6	0.81	0.95	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
7	0.84	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
8	0.81	0.92	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
9	0.84	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
10	0.75	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
11	0.72	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
12	0.72	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
13	0.75	0.95	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
14	0.63	0.95	0.97	0.99	1.00	1.00	1.00	1.00	1.00	1.00
15	0.63	0.94	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
16	0.61	0.86	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00
17	0.57	0.85	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
18	0.57	0.94	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
19	0.83	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
20	0.85	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
21	0.83	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
22	0.50	0.81	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00
23	0.43	0.79	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00
24	0.48	0.78	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00
25	0.46	0.78	0.94	0.99	1.00	1.00	1.00	1.00	1.00	1.00
26	0.71	0.88	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00
27	0.44	0.76	0.93	0.97	1.00	1.00	1.00	1.00	1.00	1.00
28	0.43	0.75	0.92	0.98	0.99	1.00	1.00	1.00	1.00	1.00
29	0.42	0.74	0.86	0.95	0.98	1.00	1.00	1.00	1.00	1.00
30	0.43	0.72	0.92	0.98	1.00	1.00	1.00	1.00	1.00	1.00
31	0.72	0.81	0.91	0.98	0.99	1.00	1.00	1.00	1.00	1.00
32	0.38	0.71	0.85	0.94	0.96	0.97	0.98	0.99	1.00	1.00

Tabela 2.4. Eficiências obtidas para diferentes números de núcleos e tamanhos de problema.

tar tendências importantes de escalabilidade. Além disso, gráficos de linha ou barras não são adequados para visualizar esse tipo de dado, já que muitas vezes disfarçam as tendências. No caso de perfis voltados para escalabilidade, seria necessário criar uma tabela para cada região de código de interesse, o que, manualmente, se torna ainda mais demorado e, com ferramentas de perfilamento, pode ser ainda mais trabalhoso.

2.2. Introdução ao PaScal Suite

O *Parallel Scalability Suite* (PaScal Suite) é um conjunto de ferramentas desenvolvido para avaliar tendências de escalabilidade em programas paralelos. O pacote simplifica a execução, medição e comparação de múltiplos cenários de teste, permitindo a análise de ambientes com diferentes quantidades de processadores e cargas de trabalho.

A ferramenta fornece elementos visuais que auxiliam o usuário a compreender o comportamento do programa e a reconhecer gargalos que exigem otimização. O PaScal Suite é composto por dois módulos principais que serão detalhados a seguir: o *PaScal Analyzer* (Seção 2.2.1), responsável pela coleta de dados, e o *PaScal Viewer*, responsável pela visualização gráfica das métricas.

2.2.1. PaScal Analyzer

O *Parallel Scalability Analyzer* [4], ou **PaScal Analyzer**, é uma ferramenta projetada para perfilar aplicações, permitindo a medição e a comparação de execuções com diferentes configurações alternativas. Seu objetivo é facilitar o entendimento da capacidade de escalabilidade de uma aplicação paralela, observando como ela utiliza os recursos computacionais disponíveis. Além disso, a ferramenta se destaca por seu baixo nível de intrusão nos programas analisados. Após sua execução, o *Analyzer* organiza os dados coletados para uma futura análise visual. Esses dados podem incluir tempo de execução, potência e outros medidores de desempenho; contudo, neste trabalho, vamos nos concentrar **no** tempo de execução.

No *Analyzer*, a instrumentação permite que os desenvolvedores observem como diferentes partes da aplicação utilizam os recursos. A instrumentação automática permite a execução sem modificar o código, enquanto a manual envolve a adição explícita de chamadas para marcar regiões de interesse.

O Código 2.1 ilustra a instrumentação manual com o *Analyzer*. Para isso, é necessário incluir a biblioteca `pascalops.h` no cabeçalho do código e delimitar a zona perfilada com `pascal_start(id)` e `pascal_stop(id)`.

```
#include "pascalops.h"
...
int main(){
    pascal_start(1);
    #pragma omp parallel
    { ... }
    pascal_stop(1);
    ...
}
```

Código 2.1. Exemplo de uso do PaScal Analyzer com OpenMP

A biblioteca `pascalops` fica visível para o *GCC* após a instalação do *Analyzer*. Após delimitar as zonas, compila-se com o argumento `-lmpascalops`:

```
g++ main.cpp -fopenmp -lmpascalops
```

Para realizar a instrumentação automática, executa-se o *pascalanalyzer* com a opção `-t aut`:

```
pascalanalyzer -t aut -c 8,4,2,1 \
               -i 100,200,400,800 \
               -o out.json <binario_executavel>
```

No exemplo acima, utilizamos `-t` para o tipo de instrumentação, `-c` para a quantidade de núcleos e `-i` para fornecer **as entradas** que serão usadas no programa. O

argumento `-o` especifica o arquivo de saída; caso omitido, o *Analyzer* salvará a saída com um nome padronizado composto pelo **nome do binário seguido de data e hora da execução** (ex: `binario_2025-10-20_14-30.json`).

2.2.2. PaScal Viewer

O *Parallel Scalability Viewer*[5] é uma aplicação web para visualizar métricas de desempenho. Ele aceita como entrada o arquivo JSON do *Analyzer*, fornecendo diagramas de cores similares a mapas de calor. O Viewer encontra-se disponível no domínio: `https://pascalsuite.imd.ufrn.br/`.

A Figura 2.2 apresenta a interface do *Viewer*. Nos diagramas apresentados, o eixo vertical (Y) representa o número de núcleos utilizados, enquanto o eixo horizontal (X) representa os tamanhos do problema (ou entradas).

O primeiro diagrama (superior esquerdo) mede a eficiência global ($E = S/P$). O segundo (superior direito) avalia a escalabilidade global. Vale notar que, diferentemente da eficiência que possui fórmula fechada clássica, a métrica de escalabilidade no PaScal é derivada comparando a manutenção da eficiência da configuração atual em relação à configuração base. Cores próximas ao azul indicam alta escalabilidade (eficiência mantida), enquanto tons de marrom apontam degradação.

Os diagramas inferiores medem a escalabilidade forte (esquerda, onde o tamanho do problema é fixo e núcleos variam) e a escalabilidade fraca (direita, onde ambos variam proporcionalmente).

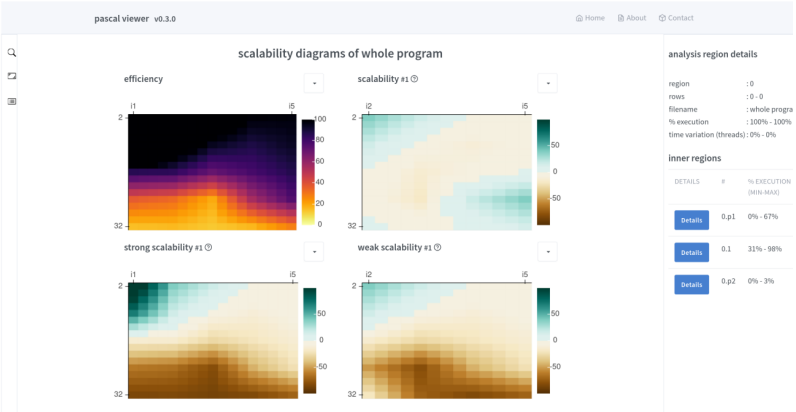

Figura 2.2. Captura de tela da interface da ferramenta web PaScal Viewer. O eixo Y indica contagem de núcleos e o eixo X as configurações de entrada.

2.3. Estudo de Caso e Aplicabilidade

Agora, avaliaremos a escalabilidade de um programa paralelo. O material utilizado nesta seção está disponível no repositório público do PaScal Suite: `https://gitlab.com/lappsufrn/pascal-suite-tutorial/-/tree/minicurso2024`.

Nosso objeto de estudo é o método de *Red-Black Gauss-Seidel*. O Código 2.2 apresenta a versão paralela utilizando *OpenMP*.

```
1 #include <omp.h>
```

```cpp
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <iostream>

#define MAX_ITER 1000

void initialize(double **A, int n) {
  // inicialize a matriz
}

double matrix_calculation(double **A, int n) {
  double tmp;
  double diff = 0;
  int i, j;

#ifdef _OPENMP
#pragma omp parallel private(tmp, i, j)
  {
#pragma omp for reduction(+ : diff)
    for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j) {
        if ((i + j) % 2 == 1) {
          tmp = A[i][j];
          A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j] +
              A[i][j + 1] +
                           A[i + 1][j]);
          diff += fabs(A[i][j] - tmp);
        }
      }
    }
#pragma omp single
    {
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                             A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    }
  }
#endif // _OPENMP
  return diff;
}
```

```
49
50 void solve_parallel(double **A, int n) {
51   int iters;
52   double diff = 0;
53   for (iters = 1; iters < MAX_ITER; ++iters) {
54     diff = matrix_calculation(A, n - 1);
55   }
56 }
57
58 int main(int argc, char *argv[]) {
59   int i;
60   double **A;
61   int n;
62
63   n = std::stoi(argv[1]);
64   A = new double *[n + 2];
65   for (i = 0; i < n + 2; i++) {
66     A[i] = new double[n + 2];
67   }
68
69   initialize(A, n);
70
71   solve_parallel(A, n - 1);
72 }
```

Código 2.2. Implementação do método Red-Black Gauss-Seidel

Configuramos um ambiente variando núcleos entre 1, 2, 4 e 8. Primeiro compilamos o código (note que para a instrumentação automática, não é estritamente necessário vincular a biblioteca `pascalops`, mas o faremos por consistência):

```
1     g++ -Wall -o RB -fopenmp RB.cpp
```

Executamos o *Analyzer*:

```
1     pascalanalyzer ./RB -c 8,4,2,1 \
2                    -i 1001,1501,2001,2501 \
3                    -r 5 -t aut -o RB.json
```

Definimos tamanhos de problema (1001 a 2501) crescendo de forma não linear (aproximadamente quadrática) para tentar acompanhar o aumento da capacidade de processamento dos núcleos (escalabilidade fraca). O argumento `-r` define 5 repetições para cada teste.

A Figura 2.3 apresenta o resultado no *Viewer* para o **primeiro conjunto de testes**. O diagrama de eficiência global demonstra queda à medida que o problema cresce. O gráfico de escalabilidade global revela baixa escalabilidade em todos os pontos.

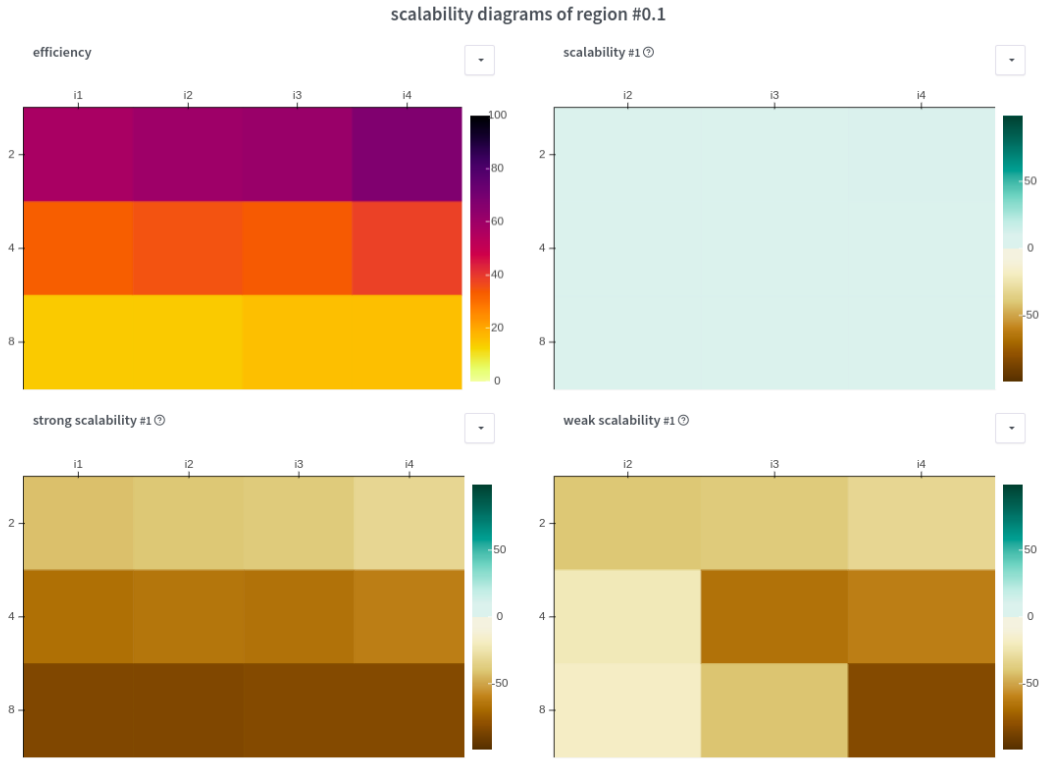

Figura 2.3. Diagramas do PaScal Viewer. A quantidade de núcleos varia em 1, 2, 4 e 8 (eixo Y) e o tamanho do problema varia nas configurações i1 (1001), i2 (1501), i3 (2001) e i4 (2501) no eixo X.

Para identificar gargalos, utilizamos a instrumentação manual. Incluímos `pascalops.h` e delimitamos as regiões. A Região 1 engloba a execução paralela total; a Região 2, o primeiro laço; e a Região 3, o segundo laço. O Código 2.3 ilustra essa instrumentação.

Recompilamos com `-lmpascalops` e executamos com `-t man`:

```
1    g++ -Wall -o RB -fopenmp RB.cpp -lmpascalops
2    pascalanalyzer ./RB -c 8,4,2,1 \
3                   -i 1001,1501,2001,2501 \
4                   -r 5 -t man -o RB.json
```

No *Viewer*, acessamos as sub-regiões (Figura 2.4). A interface indica que a Sub-Região 3 representa até 76% do tempo total.

A Figura 2.5 exibe a Sub-Região 2. Conclui-se que esse trecho é mais eficiente que a execução completa **(veja a Figura 2.3)**.

Já a Figura 2.6 apresenta a Sub-Região 3. Esse trecho apresenta baixa eficiência global, indicando ser o gargalo.

Avaliando o trecho de código delimitado por `pascal_start(3)` e `pascal_stop(3)`, identificamos que o laço aninhado responsável por calcular o valor de `diff` nos índices pares de nossa matriz, está em uma diretiva `omp single` que indica que, aquele tre-

cho deverá ser executado por somente uma thread. É possível que, se subtrairmos essa diretiva desnecessária desse trecho e adicionarmos uma diretiva `omp for` com a cláusula `reduction(+ : diff)`, vamos conseguir otimizar essa operação de redução e garantir que o laço alvo seja paralelizado.

Aplicando as modificações pensadas, a região 3 ficaria da seguinte forma:

```
pascal_start(3);
#pragma omp for reduction (+ : diff)
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                              A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    pascal_stop(3);
  }
```

Após isso, recompilamos o código e refizemos mais um conjunto de testes com o *Analyzer* agora para identificar os possíveis ganhos de nossa solução.

A Figura 2.7 mostra que, após nossa implementação, a Região 1 apresentou ganhos de eficiência em todos os pontos, embora ainda apresente baixa escalabilidade, abrindo portas para uma futura investigação mais profunda.

A Figura 2.8 mostra os gráficos de desempenho referentes à Sub-Região 3 após a identificação de um possível gargalo e a implementação de nossa solução. É possível observar que houve um ganho de eficiência em todos os pontos de nosso diagrama de eficiência global. Os valores também melhoram para os gráficos de escalabilidade forte e fraca, embora nosso código ainda não tenha atingido um nível de escalabilidade desejável.

2.4. Conclusão

Neste capítulo, discutimos a relação e as distinções entre desempenho e escalabilidade no contexto da computação paralela, apresentando o PaScal Suite como ferramenta de apoio. Através de um estudo de caso, identificamos e mitigamos um gargalo de desempenho, ilustrando como a visualização de dados pode guiar a otimização de código paralelo.

Referências

[1] Pacheco, P. S. (2011) *An introduction to parallel programming*. Morgan Kaufmann.

[2] Amdahl, G. M. (1967) *Validity of the single processor approach to achieving large scale computing capabilities*. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (pp. 483–485). Association for Computing Machinery.

[3] John L. Gustafson. 1988. Reevaluating Amdahl's law. Commun. ACM 31, 5 (May 1988), 532–533. https://doi.org/10.1145/42411.42415

[4] Silva, Vitor, Nóbrega-da-Silva, Anderson, Valderrama Sakuyama, C., Manneback, Pierre, and Xavier-de-Souza, Samuel (2022). "A Minimally Intrusive Approach for Automatic Assessment of Parallel Performance Scalability of Shared-Memory HPC Applications". *Electronics*, vol. 11, no. 5. DOI: 10.3390/electronics11050689.

[5] Nóbrega-da-Silva, Anderson, Cunha, Daniel, Silva, Vitor, Araújo Furtunato, Alex Fabiano, and Xavier-de-Souza, Samuel (2019). "PaScal Viewer: A Tool for the Visualization of Parallel Scalability Trends". In: *Handbook of Research on Emerging Developments and Applications of High Performance Computing*. pp. 250-264. ISBN: 978-981-13-6209-5. DOI: 10.1007/978-3-030-17872-7_15.

```
\\ outras bibliotecas
#include "pascalops.h"

double matrix_calculation(double **A, int n) {
double tmp;
double diff = 0;
int i, j;

  pascal_start(1);
#ifdef _OPENMP
#pragma omp parallel private(tmp, i, j)
  {
    pascal_start(2);

#pragma omp for reduction(+ : diff)
    for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j) {
        if ((i + j) % 2 == 1) {
          tmp = A[i][j];
          A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j] +
              A[i][j + 1] +
                          A[i + 1][j]);
          diff += fabs(A[i][j] - tmp);
        }
      }
    }

    pascal_stop(2);
pascal_start(3);
#pragma omp single
    {
      for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
          if ((i + j) % 2 == 0) {
            tmp = A[i][j];
            A[i][j] = 0.2 * (A[i][j] + A[i][j - 1] + A[i - 1][j]
                + A[i][j + 1] +
                            A[i + 1][j]);
            diff += fabs(A[i][j] - tmp);
          }
        }
      }
    }

    pascal_stop(3);
  }
  pascal_stop(1);
#endif // _OPENMP

  return diff;
}
```

Código 2.3. Instrumentação manual da região paralela 1 de nosso objeto de estudo.

Figura 2.4. Acesso às Sub-Regiões 2 e 3 após instrumentação manual.

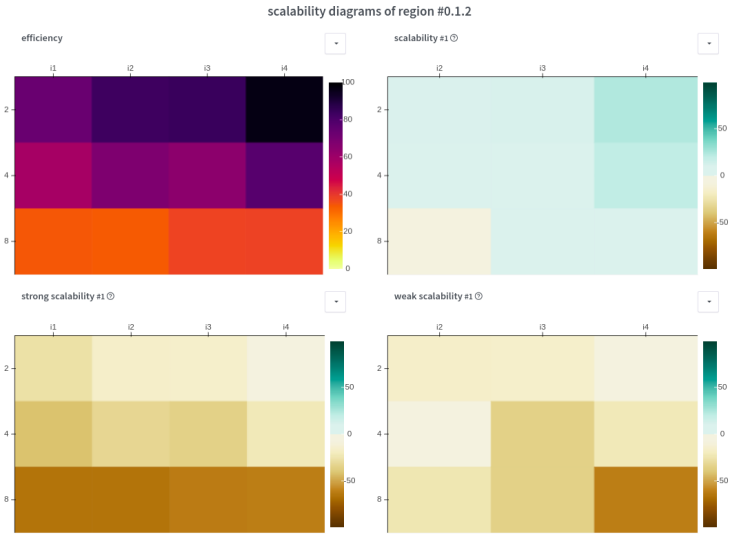

Figura 2.5. Diagramas de eficiência e escalabilidade da Sub-Região 2.

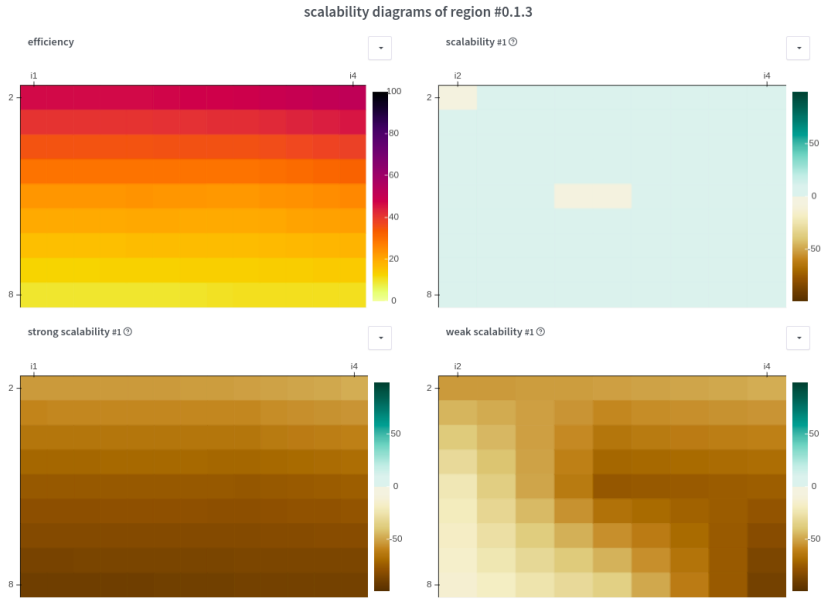

Figura 2.6. Diagramas da Sub-Região 3 no segundo conjunto de testes.

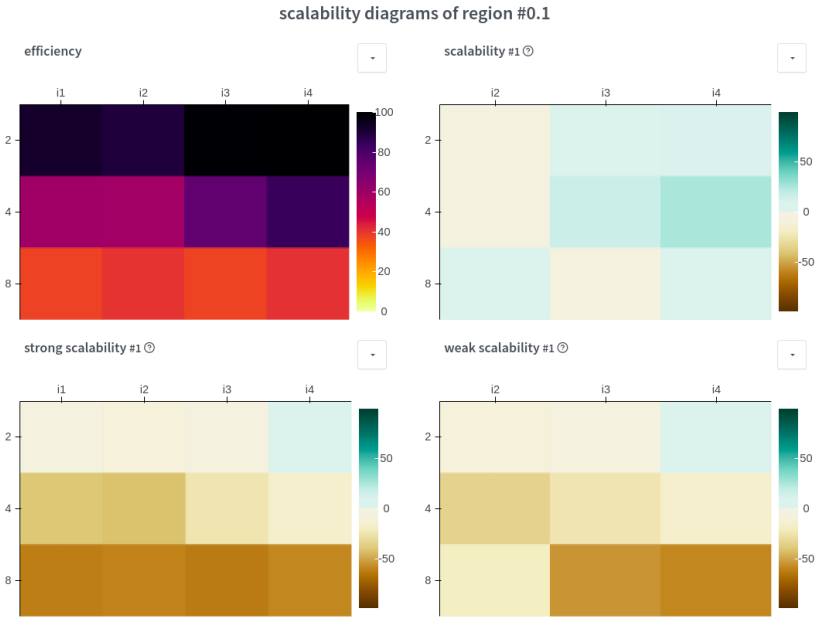

Figura 2.7. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Região 1 após nossa solução. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X

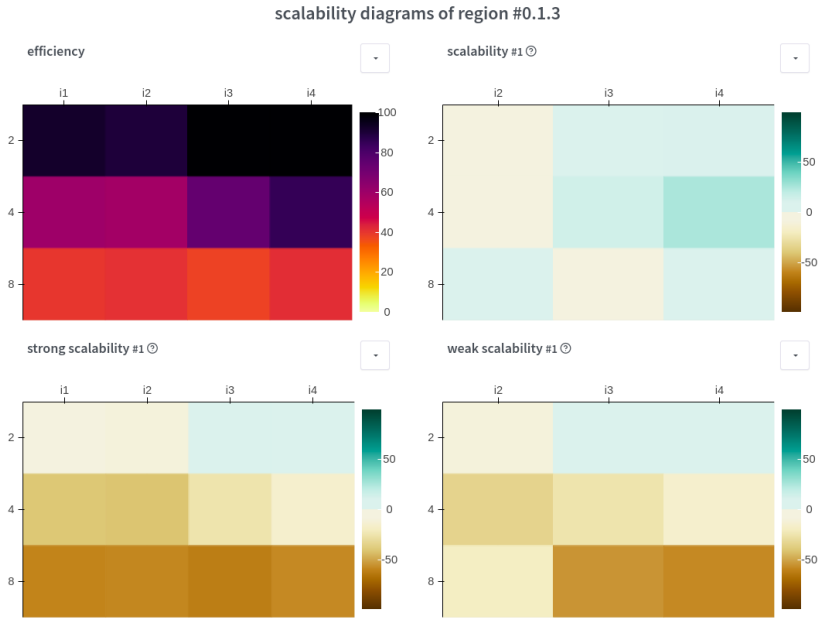

Figura 2.8. Captura de tela do PaScal Viewer com diagramas de eficiência e escalabilidade da Sub-Região 3 após nossa solução. A quantidade de núcleos varia em 2,4,8 no eixo Y enquanto o tamanho do problema varia em i1,i2,i3 e i4 no eixo X