CENTRO UNIVERSITÁRIO UNIEURO
ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

Ana Tereza Guimarães Pereira
Guilherme Henrique Almeida da Silva

PROCESSAMENTO DE IMAGENS COM MPI

Projeto final

Brasília - DF
26/05/2025

1. **Introdução**

   Este projeto tem como objetivo principal realizar o processamento eficiente de imagens geográficas ou científicas no formato `.tif` com quatro canais (RGBA), convertendo-as para imagens em escala de cinza (Grayscale), utilizando as extensões `.jpeg` ou `.png`. A solução implementa paralelismo com MPI (Message Passing Interface) para acelerar o processamento em ambientes com múltiplos núcleos ou máquinas, buscando otimização de desempenho e economia de recursos computacionais.

2. **Desafios da Solução**

   2.1. Conversão de arquivo .tif RGBA para .jpeg ou .png Gray

   O formato .tif com quatro canais representa imagens com componentes Red, Green, Blue e Alpha (transparência). Para convertê-la para uma imagem de duas dimensões em escala de cinza, é necessário aplicar uma transformação que reduza os canais RGB em um único canal de intensidade.

   Solução adotada:

   A. O código utiliza a biblioteca Rasterio para abrir imagens .tif com múltiplos canais (neste caso, até 4).
   B. A função segmentar_imagem() converte os dados lidos para o formato RGB (dados[:3]), descartando o canal Alpha se houver.
   C. Essa imagem RGB é depois convertida em tensor do TensorFlow e processada por um modelo de segmentação semântica, que retorna uma máscara segmentada (matriz 2D).
   D. A saída (mascara_segmentacao) é uma imagem em escala de cinza (com classes segmentadas) que é salva em formato .png com cv2.imwrite()

   2.2. Divisão de trabalho MPI

   O uso da biblioteca MPI permite distribuir o carregamento e processamento da imagem entre vários processos. Esse modelo de paralelismo é especialmente útil para imagens de grande dimensão, otimizando tempo e uso da CPU.

   Solução adotada:

   A. O código inicializa o MPI com mpi4py, e cada processo recebe um rank.
   B. A imagem é dividida em blocos horizontais por linha (np.array_split).
   C. Cada processo MPI (com base no seu rank) é responsável por processar apenas os blocos que lhe pertencem.
   D. O processo mestre (rank 0) coordena o processo: processa seus próprios blocos e coleta os resultados dos outros processos com comm.recv().

2.3. Nível de Complexidade

A solução exige domínio sobre:

- Processamento de imagens multidimensionais
- Programação paralela com MPI
- Sincronização de processos
- Manipulação eficiente de arquivos grandes

A combinação dessas áreas torna o projeto desafiador, demandando planejamento e conhecimento técnico sólido.

Solução adotada:

A. O código lida com vários níveis de complexidade, como:
B. Leitura de arquivos grandes com múltiplos canais.
C. Divisão eficiente da imagem entre processos MPI.
D. Integração com um modelo de machine learning da web (TensorFlow Hub).
E. Tratamento de erros na leitura com try/except.
F. Coleta e ordenação dos blocos processados para reconstrução da imagem final.

2.4. Otimização do uso de memória RAM

Durante o processamento de grandes imagens, é fundamental evitar o carregamento de toda a imagem na memória de uma só vez, pois isso pode causar estouro de memória (Out of Memory).

Solução adotada:

A. A imagem não é carregada completamente na memória.
B. Em vez disso, são usadas janelas de leitura (rasterio.windows.Window) para ler apenas o bloco correspondente às linhas atribuídas ao processo.
C. Isso permite um processamento em partes, ideal para imagens grandes.
D. Além disso, o uso de np.array_split() com blocos menores (8 vezes o número de processos) evita sobrecarga de RAM mesmo em sistemas com menos recursos.

2.5. Entendimento do problema de segmentação

Em algumas aplicações, além da conversão para escala de cinza, é necessário segmentar áreas de interesse na imagem. Isso pode ser feito com:

- Espaço de cor HSV: separa cor, saturação e intensidade, útil para destacar regiões específicas.

- Limiarização (Thresholding): converte a imagem em preto e branco com base em um valor de corte, útil para detecção de bordas ou regiões específicas.

  Solução adotada:

  A. Neste código, a segmentação não utiliza HSV ou limiarização manual, mas sim uma abordagem muito mais sofisticada com Deep Learning:
  B. Utiliza o modelo DeepLabV3+, carregado via tensorflow_hub.
  C. Esse modelo realiza segmentação semântica automática, reconhecendo e classificando regiões da imagem de forma muito mais precisa que métodos tradicionais como HSV ou thresholding.
  D. O resultado é uma máscara de classes, onde cada pixel recebe um rótulo predito

3. Ferramentas utilizadas

3.1. Linguagem

Python: escolhida pela ampla disponibilidade de bibliotecas de processamento de imagem, facilidade de uso e compatibilidade com bibliotecas MPI via mpi4py.

3.2. Bibliotecas

Biblioteca	**Finalidade**
rasterio	Leitura de imagens .tif, especialmente com dados georreferenciados
opencv (cv2)	Manipulação de imagens, conversão de formatos, limiarização
numpy	Operações matriciais de alto desempenho
os	Navegação e manipulação de arquivos e diretórios
time	Medição de tempo de execução
matplotlib.pyplot	Visualização e salvamento de imagens para depuração
mpi4py	Comunicação entre processos MPI em python

4. **Como funciona o programa?**

   O programa segue um fluxo de execução dividido entre os processos MPI, onde o rank 0 atua como coordenador (mestre) e os demais como trabalhadores (slaves).

   Fluxo de Execução:

   1. Inicialização com MPI

      O programa inicia com mpi4py, detectando o número de processos disponíveis e atribuindo um rank a cada um deles.

   2. Leitura da Imagem (Rank 0)
      - O rank 0 utiliza a biblioteca rasterio para abrir a imagem .tif.
      - Ele extrai as informações relevantes da imagem: largura, altura, número de bandas, dados dos canais (RGBA), etc.

   3. Distribuição dos Dados

      O rank 0 divide a imagem em partes (ex: faixas horizontais) e envia os blocos e metadados para os demais ranks via comm.send.

   4. Processamento Paralelo

      Cada rank trabalhador (rank > 0) recebe sua parte da imagem e:
      - Converte a parte recebida de RGBA para escala de cinza (Grayscale)
      - Salva essa parte como uma imagem .jpeg ou .png

   5. Pós-processamento (Rank 0)
      - O rank 0 também pode salvar sua parte da imagem.
      - Em seguida, utiliza OpenCV (cv2) para abrir a imagem já convertida e aplicar técnicas de segmentação como:
        1. Conversão para HSV e limiarização
        2. Limiarização adaptativa ou global

   6. Finalização

   - Todos os processos finalizam e liberam os recursos.
   - O rank 0 pode exibir ou salvar a imagem segmentada final em um diretório de resultados.

5. **Avaliação de Otimização**

   4.1. Tempo de Execução

   É medido com o módulo time antes e depois do processamento paralelo. A meta é que a versão paralela reduza significativamente o tempo em relação à versão sequencial.

Métrica usada:
Tempo total = fim - início (em segundos)

4.2. Eficiência

A eficiência é avaliada pela escala de desempenho com o número de processos:

- Speedup: Tsequencial/Tparalelo
- Eficiência: speedup/nº de processos

6. Resultados Esperados

- Redução de até 80% no tempo de processamento para imagens grandes (comparado com versão sequencial)
- Conversão precisa das imagens RGBA para Grayscale
- Segmentação clara (quando aplicada)
- Baixo uso de memória RAM (evitando crashes)
- Código modular e de fácil manutenção

7. Conclusão

Este projeto mostra a importância de unir processamento paralelo com eficiência computacional em tarefas de manipulação de imagens pesadas. A utilização de MPI via Python, aliada a bibliotecas especializadas como Rasterio e OpenCV, permite alcançar alta performance mesmo em máquinas com recursos limitados