Computação de Alto Desempenho (HPC)

Conceitos Fundamentais de Paralelização

Fernando Roig

Escola de Inverno em Astronomia

Observatório Nacional – Julho 2026

O que é HPC?

- Uso de supercomputadores ou grandes clusters de máquinas interligadas para executar, de forma confiável, cálculos e processamento de dados em escala muito maior e mais rápida do que um computador comum.
- Apoia-se em modelagem de problemas, ciência de dados e engenharia de software.
- Ligada à paralelização de algoritmos: o ganho de desempenho decorre de dividir problemas grandes em muitas tarefas menores, que podem ser executadas simultaneamente em múltiplos núcleos de CPU, GPUs ou nós de um cluster.
- Exige algoritmos projetados para explorar paralelismo de dados e de tarefas de forma eficiente.

Objetivos da apresentação

- Entender por que utilizar programação paralela.
- Diferenciar memória compartilhada e memória distribuída.
- Conhecer o modelo básico do OpenMP.
- Conhecer o modelo básico do MPI.
- Identificar as situações em que cada abordagem é mais adequada.
- Observar exemplos curtos de código em C.

Escopo

O foco será conceitual e introdutório. Não serão abordados exemplos detalhados em outras linguagens nem operações avançadas.

Processos, threads e paralelismo

- Um programa paralelo utiliza mais de uma unidade de execução.
- Uma **thread** é uma linha de execução dentro de um processo.
- Um **processo** possui seu próprio espaço de memória.
- Threads de um mesmo processo normalmente compartilham memória.
- Processos diferentes normalmente possuem espaços de memória separados.

Dois paradigmas

- Memória compartilhada: comunicação por dados compartilhados.
- Memória distribuída: comunicação por troca explícita de mensagens.

Memória compartilhada e distribuída

Memória compartilhada

- Threads acessam uma memória comum.
- A comunicação ocorre implicitamente por variáveis compartilhadas.
- É necessário controlar o acesso concorrente.
- OpenMP é o principal exemplo.

Memória distribuída

- Cada processo possui seu próprio espaço de memória.
- A comunicação ocorre explicitamente por mensagens.
- O programa deve indicar o que enviar e receber.
- MPI é o principal exemplo.

Não determinismo

- Threads executam de forma aproximadamente autônoma.
- A ordem relativa de execução pode variar entre execuções.
- O mesmo programa, com a mesma entrada, pode produzir saídas em ordens diferentes.

Exemplo

Se duas threads imprimem simultaneamente:

```
Thread 0:  valor = 7          ou          Thread 1:  valor = 19
Thread 1:  valor = 19                     Thread 0:  valor = 7
```

A ordem diferente pode ser inofensiva, mas nem sempre é.

Condição de corrida

Considere duas threads executando:

```
sum += local_value;
```

Essa operação pode envolver várias etapas:

1. Ler o valor atual de `sum`.
2. Somar o valor local.
3. Escrever o resultado de volta em `sum`.

Problema

Se duas threads realizarem essas etapas simultaneamente, uma atualização pode sobrescrever a outra.

O resultado passa a depender de qual thread "vence a corrida".

Exclusão mútua e sincronização

- Uma seção crítica é uma região que deve ser executada por apenas uma thread por vez.
- Um lock ou mutex protege essa região.
- Uma barreira faz as threads aguardarem umas pelas outras.
- Locks e barreiras garantem correção, mas podem reduzir o paralelismo.

Regra prática

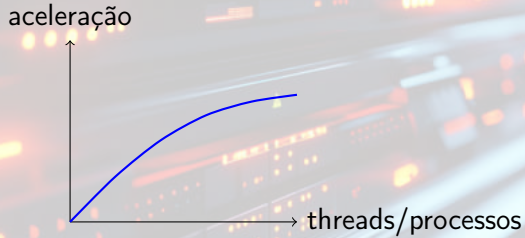
Regiões críticas devem ser pequenas e pouco frequentes.

Custo

Se muitas threads passam a esperar, o programa pode ficar praticamente serial.

Desempenho e escalabilidade

- A aceleração raramente cresce linearmente com o número de threads ou processos.
- Parte do programa pode permanecer serial.
- Sincronização e comunicação introduzem custos.
- O acesso à memória pode se tornar um gargalo.
- Em algum ponto, adicionar recursos deixa de produzir ganho.



OpenMP: visão geral

- OpenMP é uma API (Application Programming Interface) para programação paralela em memória compartilhada.
- É baseada principalmente em diretivas inseridas no código-fonte.
- O programa cria uma equipe de threads.
- Trechos específicos são executados em paralelo.
- É especialmente útil para laços e regiões de código independentes.

Diretiva fundamental

```
#pragma omp parallel
```

Primeiro programa OpenMP

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Thread %d\n", id);
    }

    return 0;
}
```

- Todas as threads executam o bloco associado à diretiva.
- `omp_get_thread_num()` identifica a thread; sempre começa em 0.
- A ordem das mensagens não é garantida.

Paralelização de laços

A diretiva mais comum para laços é:

```
#pragma omp parallel for
```

- O compilador distribui iterações entre as threads.
- Cada iteração deve ser independente das demais.
- A escolha do laço adequado é essencial para obter desempenho.

Exemplo conceitual

```
1       #pragma omp parallel for {
2       for (i = 0; i < N; i++)
3           process(data[i]);
4       }
5
```

Cuidado

Dependências entre iterações podem tornar a paralelização incorreta.

Escopo das variáveis

`shared` Uma cópia é compartilhada pelas threads.

`private` Cada thread possui sua própria cópia.

`default(none)` Obriga a declarar explicitamente o escopo.

`num_threads` Solicita um número de threads.

Boa prática

Utilizar `default(none)` ajuda a revelar dependências e evitar erros de escopo.

Pergunta fundamental

Quais variáveis são realmente compartilhadas? Quais podem ser privadas?

critical e reduction

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    int sum = 0;
    #pragma omp parallel num_threads(4)
    {
        int local_sum = 0;
        for (int i = 1; i <= 1000; i++) {
            local_sum += i;
        }
        #pragma omp critical
        {
            sum += local_sum;
        }
    }
}
```

- Protege uma região arbitrária.
- Pode se tornar gargalo.

critical e reduction

Uma redução combina resultados parciais produzidos pelas threads.

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    omp_set_num_threads(4);
    int sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 1; i <= 1000; i++) {
        sum += i;
    }

    printf("sum = %d\n", sum);
    return 0;
}
```

- Cada thread trabalha com uma cópia parcial.
- O OpenMP combina as cópias ao final.
- É preferível a proteger cada soma individual com `critical`.

Balanceamento e particionamento

- O trabalho deve ser distribuído de forma aproximadamente uniforme.
- Threads ociosas representam recursos desperdiçados.
- Dados podem ser particionados por:
  - blocos de um vetor;
  - linhas de uma matriz;
  - regiões de uma imagem;
  - subconjuntos de partículas.
- A política de escalonamento deve considerar o custo das iterações.

Ideia central

Paralelismo não é apenas criar threads: é distribuir trabalho útil.

Escalonamento de laços (`schedule`)

Política	Funcionamento	Característica
`static,chunk`	Iterações distribuídas previamente	Baixo overhead; pode desbalancear
`dynamic,chunk`	Threads solicitam novos blocos	Melhor balanceamento; maior overhead
`guided,chunk`	Blocos inicialmente maiores e progressivamente menores	Compromisso entre overhead e balanceamento

Analogia com a distribuição do baralho.

Exemplo

```
#pragma omp parallel for schedule(dynamic, 10)
```

Desafios do OpenMP

- Condições de corrida.
- Uso incorreto de variáveis compartilhadas.
- Dependências entre iterações.
- Excesso de regiões críticas.
- Sobrecarga de criação, sincronização e escalonamento.
- Gargalos de acesso à memória.
- Dificuldade de depurar resultados não determinísticos.

Ferramentas úteis

Compiladores com suporte a OpenMP e ferramentas de análise de concorrência podem ajudar na detecção de erros.

Práticas recomendadas em OpenMP

1. Começar com uma região paralela pequena.
2. Declarar explicitamente o escopo das variáveis.
3. Preferir `reduction` quando a operação permitir.
4. Minimizar regiões `critical`.
5. Verificar dependências entre iterações.
6. Medir o desempenho com diferentes números de threads.
7. Comparar políticas de escalonamento.

Exemplos de aplicação de OpenMP

- Simulações físicas: a dinâmica molecular e o problema de N corpos são exemplos de aplicação que podem ser altamente paralelizados com OpenMP. Em cada passo de tempo, cada thread pode calcular as interações entre um subconjunto de átomos da molécula ou corpos do sistema.
- Processamento de imagens: as operações de processamento de imagens, como filtragem, convolução e transformações de Fourier, podem ser facilmente paralelizadas em OpenMP. Por exemplo, um filtro de média pode ser aplicado a uma imagem dividindo-a em várias regiões e processando cada região em um thread separado.
- Aprendizado de máquina: as etapas de treinamento e inferência de muitos modelos de aprendizado de máquina podem ser paralelizadas com OpenMP. Por exemplo, o treinamento de redes neurais pode ser acelerado dividindo o conjunto de dados em lotes e processando cada lote em um thread separado.

Compilação e execução com OpenMP

Compilação com GCC

```
gcc -fopenmp -O2 -o programa programa.c
```

Execução

```
export OMP_NUM_THREADS=4
./programa
```

- `-fopenmp` habilita o suporte à API.
- `-O2` solicita otimizações do compilador.
- O número de threads pode ser alterado sem recompilar o programa.

Transição para MPI

OpenMP

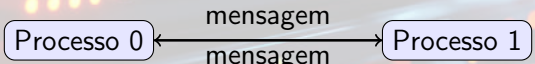
- Threads.
- Memória compartilhada.
- Comunicação implícita por variáveis.
- Geralmente dentro de um nó.

MPI

- Processos.
- Memória distribuída.
- Comunicação explícita por mensagens.
- Pode utilizar vários nós.

MPI: modelo de passagem de mensagens

- MPI é uma especificação para comunicação entre processos.
- Cada processo possui seu próprio espaço de endereçamento.
- Processos trocam dados por meio de mensagens.
- O programa deve indicar origem, destino e conteúdo da comunicação.



Ranks, SPMD e comunicadores

- Normalmente, todos os processos executam o mesmo programa.
- Esse modelo é chamado de SPMD: *Single Program, Multiple Data*.
- Cada processo possui um identificador chamado **rank**.
- Os ranks normalmente variam de 0 a $P - 1$.
- Uma tag serve para identificar uma mensagem.
- Um comunicador define o grupo de processos que participa do intercâmbio de mensagens.

Comunicador mais comum

`MPI_COMM_WORLD`

(analogia dos Correios)

Inicialização de um programa MPI

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int rank, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    printf("Processo %d de %d\n", rank, nprocs);

    MPI_Finalize();
    return 0;
}
```

- `MPI_Init`: inicializa o ambiente MPI.
- `MPI_Comm_rank`: obtém o rank.
- `MPI_Comm_size`: obtém o número de processos.
- `MPI_Finalize`: encerra o ambiente MPI.

Comunicação ponto a ponto

```c
if (rank == 0) {
    int value = 42;

    MPI_Send(&value, 1, MPI_INT,
             1, 0, MPI_COMM_WORLD);
}
else if (rank == 1) {
    int value;

    MPI_Recv(&value, 1, MPI_INT,
             0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);

    printf("Recebido: %d\n", value);
}
```

- O processo 0 envia uma mensagem.
- O processo 1 recebe a mensagem.
- A mensagem possui dados, quantidade, tipo, destino/origem, tag e comunicador.

Comunicação bloqueante e não bloqueante

Comunicação bloqueante

A chamada pode esperar até que a operação esteja suficientemente concluída para retornar.

Comunicação não bloqueante

A chamada inicia a operação e permite que o processo continue executando outro trabalho. É necessário verificar posteriormente a conclusão.

Cuidado

Comunicações mal ordenadas podem causar deadlock: cada processo fica esperando uma operação que o outro ainda não iniciou (analogia da ligação telefônica)

Operações coletivas

Operações coletivas envolvem um grupo de processos.

- `Broadcast`: um processo distribui o mesmo dado aos demais.
- `Reduce`: combina valores em um processo destino.
- `Scatter`: distribui partes de um conjunto de dados entre os processos.
- `Gather`: coleta partes de um conjunto de dados entre os processos.
- `Barrier`: sincroniza a chegada das mensagens dos processos.

Será apresentado apenas um exemplo de redução.

MPI_Reduce

```c
int local_value = rank + 1;
int total_value = 0;

MPI_Reduce(&local_value,
           &total_value,
           1,
           MPI_INT,
           MPI_SUM,
           0,
           MPI_COMM_WORLD);

if (rank == 0) {
    printf("Total = %d\n", total_value);
}
```

- Cada processo possui um valor local.
- O MPI combina os valores com uma operação, neste caso, soma.
- O resultado é armazenado no processo raiz, o rank 0.

Sincronização e correção em MPI

Em MPI, os processos normalmente **não compartilham diretamente a mesma memória**. Os principais problemas de correção são:

- mensagens enviadas e recebidas em ordem incompatível;
- tags ou fontes incorretas;
- buffers reutilizados antes da conclusão da comunicação;
- deadlocks;
- dependências entre etapas executadas em ordens diferentes.

Mensagem importante

A comunicação explícita facilita a separação dos dados, mas exige que o programador controle corretamente a coordenação entre processos.

Desempenho da comunicação MPI

- Comunicação possui latência e custo de transferência.
- Muitas mensagens pequenas podem ser mais caras que poucas mensagens maiores.
- O agrupamento de mensagens pode reduzir o overhead.
- O balanceamento de carga evita processos ociosos.
- Operações coletivas devem ser usadas de forma consciente.

Regra prática

Reduzir a quantidade de comunicação, sem comprometer a correção do algoritmo.

Desafios do MPI

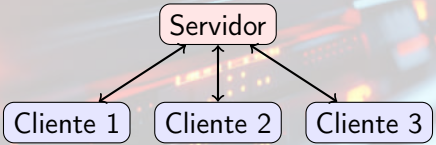
- Maior complexidade de programação.
- Necessidade de particionar os dados explicitamente.
- Controle de origem, destino, tags e buffers.
- Possibilidade de deadlock.
- Depuração mais difícil.
- Balanceamento de carga entre processos.
- Custo de comunicação entre nós.

Práticas recomendadas em MPI

1. Dividir o problema em subproblemas bem definidos.
2. Manter clara a responsabilidade de cada processo.
3. Evitar dependências desnecessárias entre processos.
4. Minimizar mensagens pequenas e frequentes.
5. Validar origem, destino, tag e tamanho dos buffers.
6. Medir separadamente computação e comunicação.
7. Testar com diferentes números de processos.

Exemplos de aplicação de MPI

- Simulação de dinâmica molecular, para calcular as interações entre as moléculas em um sistema.
- Problema de N corpos.
- Análise de grandes conjuntos de dados genéticos e para simular a dinâmica de sistemas biológicos.
- Na área financeira, cálculo de preços de opções e outras análises de risco.
- Em geral, qualquer problema que use o paradigma servidor-cliente



Compilação e execução com MPI

Compilação

```
mpicc -O2 -o programa programa.c
```

Execução

```
mpirun -np 4 ./programa
```

- `mpicc` utiliza um compilador C configurado para MPI.
- `-np 4` solicita quatro processos.
- A configuração exata pode variar conforme o ambiente computacional.

OpenMP versus MPI

Aspecto	**OpenMP**	**MPI**
Unidade de execução	Thread	Processo
Memória	Compartilhada	Distribuída
Comunicação	Variáveis compartilhadas	Mensagens
Escopo típico	Um nó	Um ou vários nós
Complexidade	Menor	Maior
Escalabilidade	Limitada pelo nó	Pode alcançar clusters

Quando usar cada abordagem?

OpenMP é uma boa escolha quando

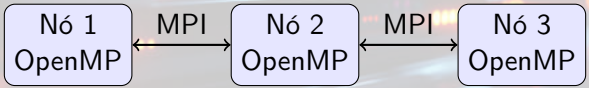
- O programa já possui laços independentes.
- Os dados estão em uma memória compartilhada.
- O objetivo é paralelizar rapidamente um código C.
- O problema cabe confortavelmente em um nó.

MPI é uma boa escolha quando

- O problema precisa utilizar vários nós.
- Os dados podem ser particionados.
- A comunicação entre subproblemas é controlável.
- A escalabilidade distribuída é importante.

Programação híbrida

- Um cluster pode conter vários nós.
- Cada nó possui vários núcleos.
- MPI pode distribuir o trabalho entre nós.
- OpenMP pode utilizar os núcleos dentro de cada nó.



Observação

A programação híbrida aumenta a flexibilidade, mas também a complexidade.

Fluxo típico de desenvolvimento com OpenMP

1. Implementar e validar a versão serial.
2. Identificar os trechos com maior custo computacional.
3. Procurar laços com iterações independentes.
4. Adicionar uma diretiva OpenMP.
5. Verificar escopo das variáveis.
6. Testar diferentes números de threads.
7. Comparar correção e desempenho.

Princípio

Primeiro garantir a correção; depois otimizar o desempenho.

Fluxo típico de desenvolvimento com MPI

1. Implementar e validar a versão serial.
2. Dividir o problema em subproblemas.
3. Definir os dados locais de cada processo.
4. Identificar as mensagens necessárias.
5. Implementar inicialização e finalização MPI.
6. Testar com poucos processos.
7. Medir comunicação, balanceamento e escalabilidade.

Princípio

A decomposição dos dados costuma ser mais importante que a própria chamada de comunicação.

Resumo dos conceitos fundamentais

- Paralelismo utiliza múltiplas unidades de execução.
- OpenMP usa threads e memória compartilhada.
- MPI usa processos e troca explícita de mensagens.
- Condições de corrida exigem controle de acesso.
- `reduction` é uma ferramenta importante no OpenMP.
- Ranks, mensagens e comunicadores são fundamentais no MPI.
- Comunicação e sincronização têm custo.
- O melhor desempenho depende da decomposição do problema.

Mensagem final

OpenMP

Comece pelo laço: identifique iterações independentes e controle cuidadosamente o escopo das variáveis.

MPI

Comece pelos dados: divida o problema e defina claramente como os processos trocarão informações.

Paralelizar é distribuir trabalho sem perder a correção.

Dica para os Pythonisos

Tanto MPI quanto OpenMP podem ser usados em Python.

- MPI possui uma biblioteca nativa em Python: `mpi4py`
- OpenMP não pussui suporte nativo, mas pode ser utilizado através de ferramentas intermediárias:
  - Cython: `cython.parallel`
  - Fortran/C + `f2py/cffi`
  - Numba: `@njit(parallel=True)`
  - Algumas bibliotecas de NumPy e SciPy já vem otimizadas com OpenMP.

Referências

- OpenMP
  `https://www.openmp.org/`
- Referência rápida do OpenMP:
  `https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf`
- Open MPI
  `https://www.open-mpi.org/doc/`
- MPI Forum.
  `https://www.mpi-forum.org/`

Obrigado!