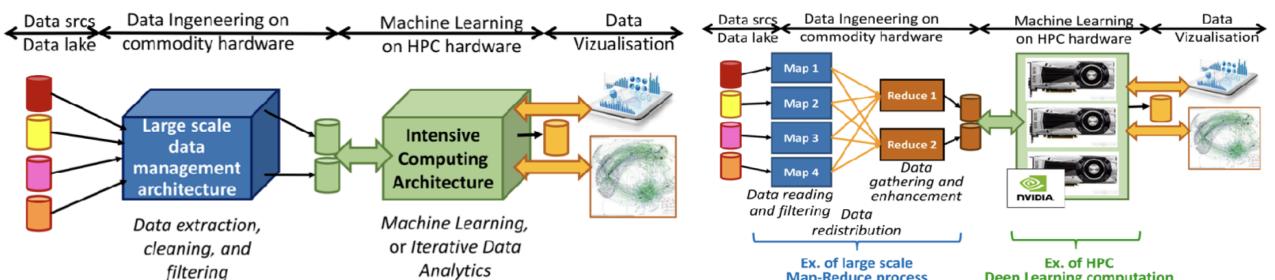
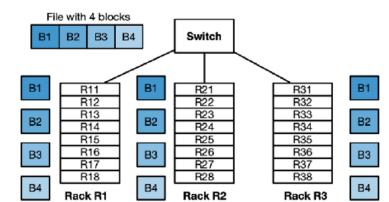


SUPERCOMP

AULA 1:

- Solução de alto desempenho:
 - algoritmos eficientes
 - implementação eficiente
 - paralelismo
- Escalabilidade
 - habilidade de um sistema lidar com o aumento da carga de processamento sem apresentar uma degradação significante em seu desempenho
 - vertical
 - scale-up
 - fazer upgrade na infra existente (+ memória, por exemplo)
 - horizontal
 - scale-out
 - adicionar novas máquinas ao parque computacional
 - distribuir os dados e o trabalho de processamento em diversas máquinas
 - podemos fazer uso de um cluster de máquinas
 - dados são armazenados em um sistema de arquivos distribuído (ex: HDFS)
 - cada arquivo é dividido em blocos de tamanho fixos
 - cada bloco é replicado em diversos nós do cluster
- Lei de Amdahl
 - numa aplicação existe sempre uma parte que não pode ser paralelizada
 - seja S a parte do trabalho sequencial, 1-S é a parte suscetível de ser paralelizada
 - mesmo que a parte paralela seja perfeitamente escalável, o aumento do desempenho (speedup) está limitado pela parte sequencial
 - Speedup = $1 / (S + ((1-S) / \text{numero_de_processadores}))$
- Big Data
 - faz referência ao grande volume, variedade e velocidade de dados que demandam formas inovadoras e rentáveis de processamento da informação, para melhor percepção e tomada de decisão

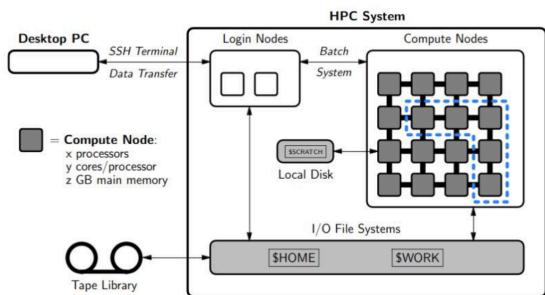


- paralelismo importa, mas nem sempre é a única solução

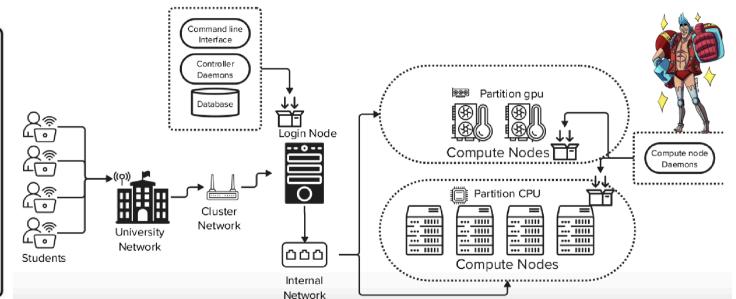
AULA 2:

- HPC
 - o que é?
 - ambiente com um grande poder computacional
 - que tipo de problema pode resolver?

- comportamento da turbulência
- biotecnologia
- qual o mercado?
- no mundo
 - top 500
 - santos dumont
 - frontier
- Arquitetura HPC

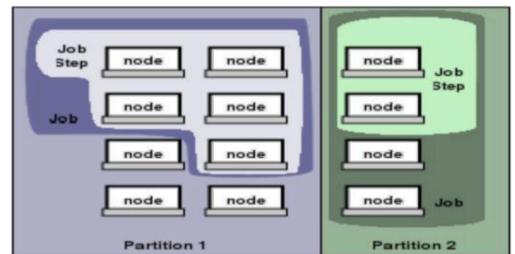


- Ambiente HPC - Cluster Franky



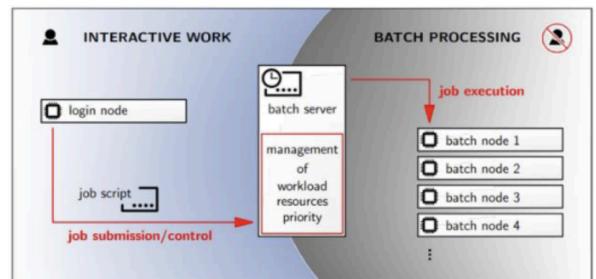
- SLURM

- Simple Linux Utility for Resource Management
- sistema de gerenciamento de filas e recursos para clusters Linux
- amplamente adotado devido a sua eficiência e simplicidade
- terminologia
 - Computing Node: computer used for the execution of programs
 - Partition: group of nodes into logical sets
 - Job: allocation of resources assigned to a user for some time
 - Step: sets of (possible parallel) tasks with a job

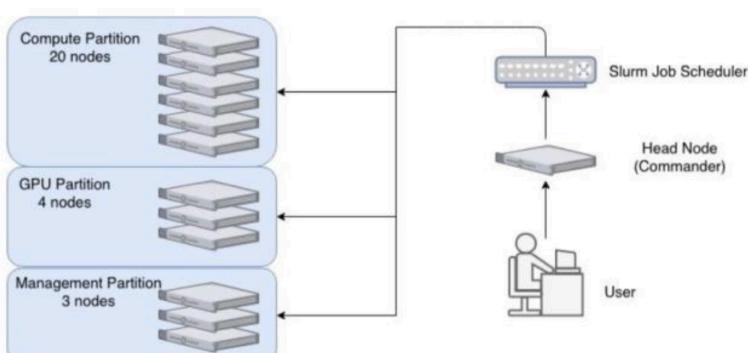


- Jobs

- interativo:
 - runs in terminal (just like using a local machine)
 - can interact with the job while running
- batch
 - submit to server and runs by itself, until finished or error
 - cannot interact with the job while running

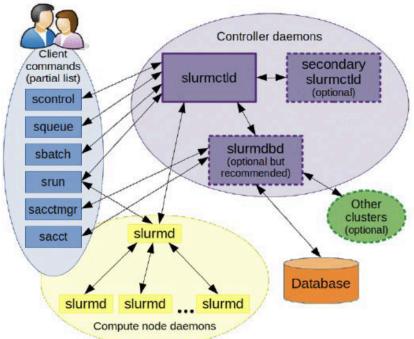


- Partitions (filas)

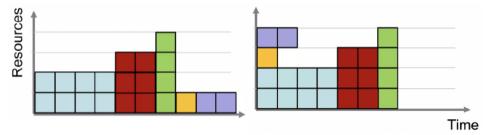


AULA 3:

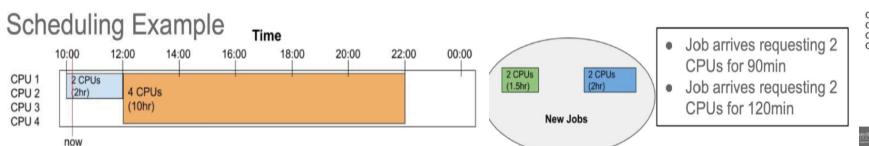
- A linguagem e o hardware importam
- Componentes do SLURM =====>
- Como o SLURM prioriza?
 - Age: length of time a job has been waiting in the queue, eligible to be scheduled
 - Fairshare: difference between the portion of the computing resource that has been promised and the amount of resources that has been consumed
 - Job size: number of nodes or CPUs a job is allocated
 - Partition: factor associated with each node partition
 - Default - FIFO scheduling
 - Backfill scheduling
 - job priority
 - time limit (important)



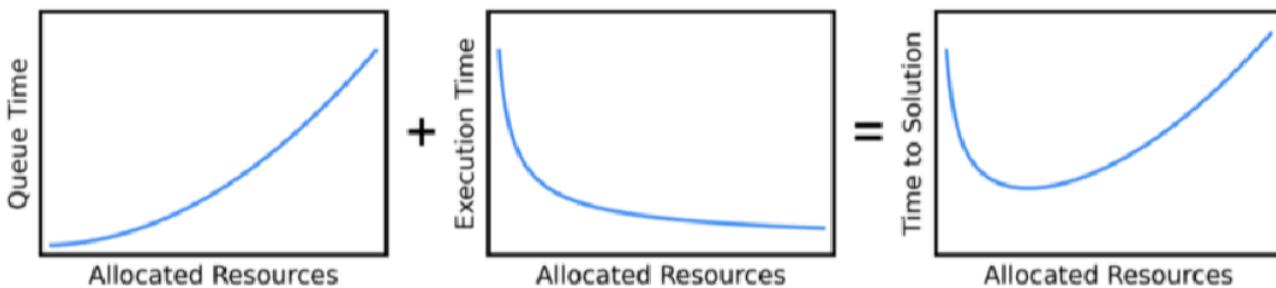
FIFO e Backfill



Scheduling Example



- Quanto tempo dura o job?

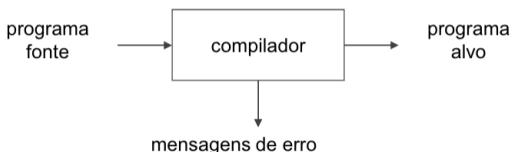


- uma boa estratégia é escolher o menor conjunto de recursos que proporcione uma aceleração razoável em relação ao baseline (sequencial)
- portanto, não é só pedir, mas pedir corretamente
- Como pedir? O que pedir?
 - por meio de configurações do arquivo .slurm
 - #!/bin/bash
 - #SBATCH --job-name=convolucao_buffado
 - #SBATCH --output=buffado_%j.txt
 - #SBATCH --ntasks=4
 - #SBATCH --cpus-per-task=4
 - #SBATCH --mem=1024
 - #SBATCH --time=00:10:00
 - #SBATCH --partition=espec
- Usando o SLURM
 - sinfo: exibe informações sobre os recursos e estados das filas e nós no cluster gerido pelo SLURM
 - srun: submeter jobs de forma interativa
 - sbatch: submeter jobs em forma de lote

- `squeue`: lista os jobs que estão em execução / na fila de espera para serem executados
- `sprio`: exibe as prioridades dos jobs na fila de espera, com base em fatores configurados no SLURM, (ex: política de agendamento)
- `sacct -j <job_id>`: mostra um resumo do job

AULA 4:

- O que pode causar tempos diferentes rodando o mesmo código?
 - partição dos dados desequilibrada
 - overhead do SLURM
 - carregamento dinâmico e inicialização de bibliotecas
 - contention de recursos
 - latência de comunicação
- Algoritmo
 - sequência finita de passos executáveis que resolve um problema
 - implementar um algoritmo: transformação de um algoritmo em um programa executável
 - complexidade computacional (classe de algoritmos)
 - estrutura de dados
- Implementação
 - medido em segundos
 - tecnologias usadas: linguagens de programação, bibliotecas
 - hardware usado: clock de CPU, RAM, cache, número de núcleos, etc
- dado um “bom” algoritmo, definir:
 - linguagem de programação adequada
 - paralelismo indicado
 - implementação paralela eficiente
- C++
 - derivado da linguagem C
 - Processo de compilação
 - compilador é um software que lê a especificação de um programa em uma linguagem-fonte e o traduz em um programa em uma linguagem-alvo



- `compilar: g++ file.cpp -o file`
- `compilar no mac: clang++ -std=c++11 -stdlib=libc++ -Wall hello.cpp -o hello`
- Pontos comuns com C

- Função principal: `int main(int argc, char *argv[]);`
- Comentários: `/* */`
- Tipos de dados: `int, float, double, char`
- Variações de dados: `unsigned, short, long`
- Qualificadores de variáveis: `const, static`
- Casting: `(int) (float) (char)`
- Operações: `+, -, *, /, %`
- Atribuição: `=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=`
- Incremento e decremento: `++ --`
- Operadores lógicos: `!, &&, ||`
- Operador condicional ternário : `(? :)`
- Operador vírgula: `(,)`
- Operadores Bitwise : `(&, |, ^, ~, <<, >>)`
- Construção Condicional : `if, else, switch`
- Loops: `while, do-while, for`
- Comparadores: `==, !=, >, <, >=, <=`
- Diretivas de pré-processamento (`#define, etc.`)
- Declaração de funções: `type func(...) { ... }`
- Vetores e Matrizes: `type name [elements][...];`
- Enumeradores: `enum`
- Organização de dados: `structs`
- Redefinidor de tipos: `typedef`

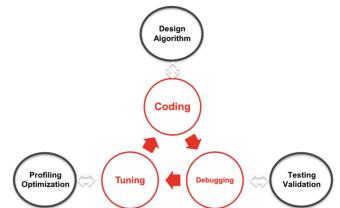
- Input/Output: cin (input >>); cout (output <<); cerr (error); clog (logging)
- Namespace: declara uma região de escopo para os identificadores
- Inicialização de variáveis: type identifier = initial_value;
- Inicialização de vetores/matrizes:
 - preencher um vetor sem definir seu tamanho com [] → o compilador assumirá automaticamente o tamanho para o vetor que corresponde ao número de valores incluídos entre as chaves {}
 - int foo [] = { 16, 2, 77, 40, 12071 };
- Tipos de dados novos
 - bool true / false
 - string

AULA 5:

- Arquivos
 - ifstream e ofstream
 - ./arquivo < entrada.txt > saida.txt
- srun ou sbatch?
 - sbatch precisa criar o .slurm; >> sbatch slurm_meu_job.slurm
 - srun não tem .slurm; srun --pty /bin/bash

AULA 6:

- Perguntas:
 - Qual o papel do nó de login no cluster franky?
 - servir como o ponto de entrada para o cluster
 - Como o SLURM contribui para o funcionamento do cluster franky?
 - gerencia a execução das tarefas submetidas pelos usuários
 - Para que serve a pasta SCRATCH em um Cluster como o Franky e o Santos Dumont?
 - serve como área de armazenamento temporário para arquivos e dados
 - Qual é a principal vantagem de utilizar um cluster em comparação com um único computador?
 - Maior escalabilidade e poder computacional
 - Em um cluster, o que é um "nó"?
 - Um computador dentro do cluster que processa tarefas atribuídas
 - O que é paralelismo em HPC?
 - Execução de múltiplas tarefas simultaneamente em vários processadores
 - Qual é a função do comando srun no SLURM?
 - Iniciar um job interativo nos nós alocados
 - Qual é a função do comando squeue no SLURM?
 - Mostrar o estado atual dos jobs na fila do cluster
 - O que faz a opção partition em um script SLURM?
 - Especifica a fila na qual o job será submetido
- Ciclo habitual de desenvolvimento =====>
- Medição de desempenho
 - otimizamos um algoritmo:
 - como medir quanto tempo cada função demora?
 - nossa função ficou mais rápida? se sim, quanto? se não, porquê?
 - como medir "quantidade de trabalho feito"?



- Profiling
 - análise de um programa durante sua execução, de modo a determinar seu consumo de memória e/ou tempo de execução
 - possível responder duas perguntas:
 - onde o programa consome mais recursos?
 - onde concentrar meus esforços de otimização
- Conclusões
 - entrada e saída custam caro
 - implementações diferentes do mesmo algoritmo podem ter desempenho diferentes
 - detalhes finos só são visíveis com o auxílio de ferramentas de profiling

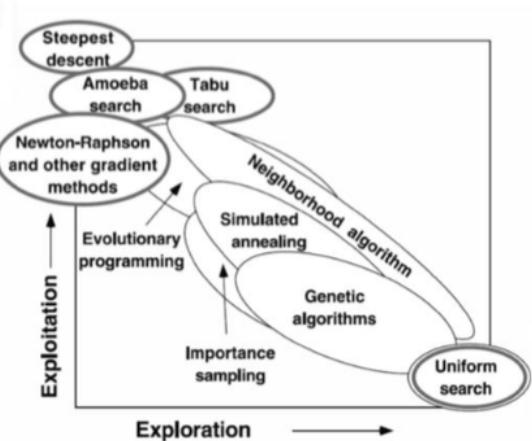
AULA 7:

- problemas difíceis aparecem em muitas áreas:
 - Pesquisa operacional (logística, produção, etc.)
 - Machine Learning
 - Finanças
 - Marketing
 - Planejamento Urbano – Mobilidade
- Resolução de Problemas - Otimização
 - Função objetivo: algo que queremos maximizar ou minimizar
 - Restrições: definem quais possíveis soluções são válidas
 - Muitas classes de problemas:
 - Programação Linear / Inteira
 - Programação não-linear
 - Otimização combinatória
 - objetivo: selecionar um objeto com melhor função objetivo dentre uma coleção finita
 - não tem derivada nem vizinhança
 - coleção não é densa
 - técnicas tradicionais de cálculo e otimização não funcionam, pois nosso problema é discreto
- Busca exaustiva – Força bruta
 - normalmente a primeira ideia para resolver problemas computacionais
 - mas se para uma mochila nós temos n possíveis itens, então temos 2^n possíveis combinações para serem testadas
 - para a mochila binária, a complexidade do algoritmo é $O(2^n)$, o que nos limita a executar esse approach apenas para pequenos valores de n

AULA 8:

Exploration vs. exploitation

- Busca global
 - avalia todas possibilidades de resposta, e sempre retorna a melhor
 - limitações:
 - e se não tiver tempo para esperar achar a melhor solução?
 - será possível encontrar uma aproximação em tempo razoável? como?



- Exploration X Exploitation
 - Exploration:
 - decisão não localmente ótima feita "de propósito"
 - visa adicionar variabilidade nas soluções geradas
 - Exploitation:
 - explorar alguma propriedade do problema
 - pode ser uma intuição que leve a bons resultados em curto prazo
 - Algoritmo Genético para a Mochila Binária
 - De volta para a Mochila
 - heurística/busca global: 100%
- Exploitation**
- como adicionar Exploration?
 - alternar decisões às vezes
 - escolha qualquer às vezes
 - Exploration
 - requer a capacidade de criar um programa que executa de maneira diferente a cada execução
 - Precisa:
 - criar fonte de aleatoriedade;
 - maneira de gerar sequências de números aleatórios
-
- ```

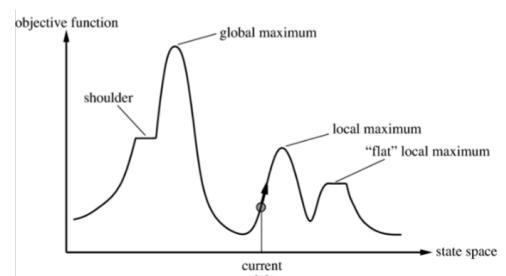
graph TD
 A[Create Initial Population] --> B[Fitness Measure]
 B --> C[Selection]
 C --> D[Crossover]
 D --> E[Mutation]
 E --> B
 B -- N --> C
 F{Solution is Reached} -- Y --> G[Stop]
 F -- N --> B

```

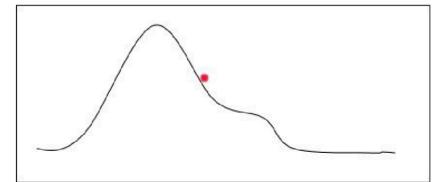
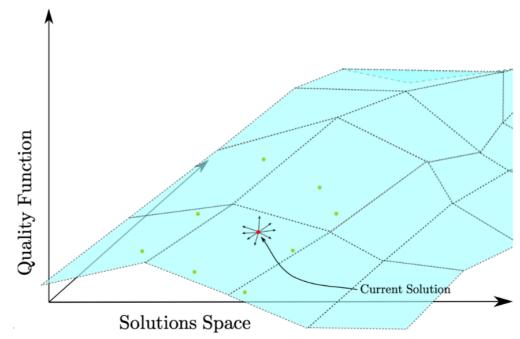
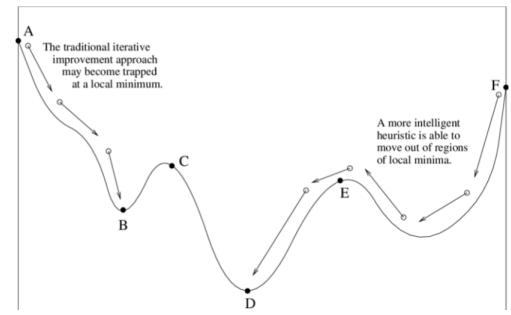
- Números aleatórios
  - gerador de números aleatórios é impossível de ser criado usando um computador
    - impossível predizer qual será o próximo número aleatório "de verdade";
    - um computador executa uma sequência de comandos conhecidos, baseando-se em dados guardados na memória: execução determinística
- Números (pseudo-)aleatórios
  - gerador de números pseudo-aleatório (pRNG): algoritmo determinístico que gera sequências de números que parecem aleatórias
    - Determinístico: produz sempre a mesma sequência
    - Sequências que parecem aleatórias: não conseguíamos distinguir uma sequência gerada por um pRNG e uma sequência aleatória de verdade
- Gerando números aleatórios
  - Sorteio de números aleatórios é dado por 2 elementos:
    - gerador: produz bits aleatórios a partir de um parâmetro seed. Cada seed gera uma sequência diferente de bits
    - distribuição de probabilidade: gera sequência de números a partir de um conjunto de números

## AULA 9:

- Problemas de otimização
  - em muitos problemas de otimização, o caminho para se atingir o objetivo é irrelevante, desde que a possamos conseguir uma solução para o problema em si



- Melhorias após solução aleatória
  - exemplo: minimização ----->
- Como podemos melhorar?
  - pra mochila, após gerarmos soluções iniciais aleatórias, podemos fazer duas ações:
    - encher mochila: verificar se algum objeto não selecionado cabe na mochila
    - trocar 2 objetos: verificar se é possível substituir objeto selecionado por outro de melhor valor que foi deixado de fora
  - (condições necessárias, mas não suficientes, para optimalidade)
- Busca Local
  - não precisa do caminho para obter a solução => buscar possíveis soluções por meio da aleatoriedade
  - obtém-se uma solução inicial aleatória, em tempo plausível
  - função que avalie a qualidade (fitness) dessa solução. Essa qualidade não está relacionada a trajetória da solução, apenas a solução em si
  - a partir dessa solução inicial, fazer busca na vizinhança de soluções (local), de modo a melhorar a qualidade da solução
  - Hill Climbing
    - algoritmo clássico para otimização, bastante eficiente na tarefa de encontrar máximos ou mínimos locais
    - iniciar em um ponto aleatório X e fazer sua avaliação
    - mover do nosso ponto X para um novo ponto vizinho X'
      - se X' for uma solução melhor que X, ficar nele e repetir o processo
      - caso contrário, voltar pra X e visitar outro vizinho ou interromper
  - Busca local com perturbação
    - ideia baseada em dois estágios:
      - gera uma solução inicial aleatória, e realiza busca local (hill climbing) [intensificação]
      - faz uma perturbação na melhor solução encontrada, para evitar máximo local [diversificação]
    - desafio: controlar intensificação X diversificação (critério de parada)
  - Desvantagens
    - depende da solução inicial
    - aleatorizada (o que nem sempre é um problema)
    - oferece garantia fraca (máximos/mínimos locais) de qualidade
    - estratégia de perturbação pode ajudar
  - Vantagens
    - rápida
    - resultados bons para N grande
    - não ficou bom? Rode mais vezes



## PALESTRA

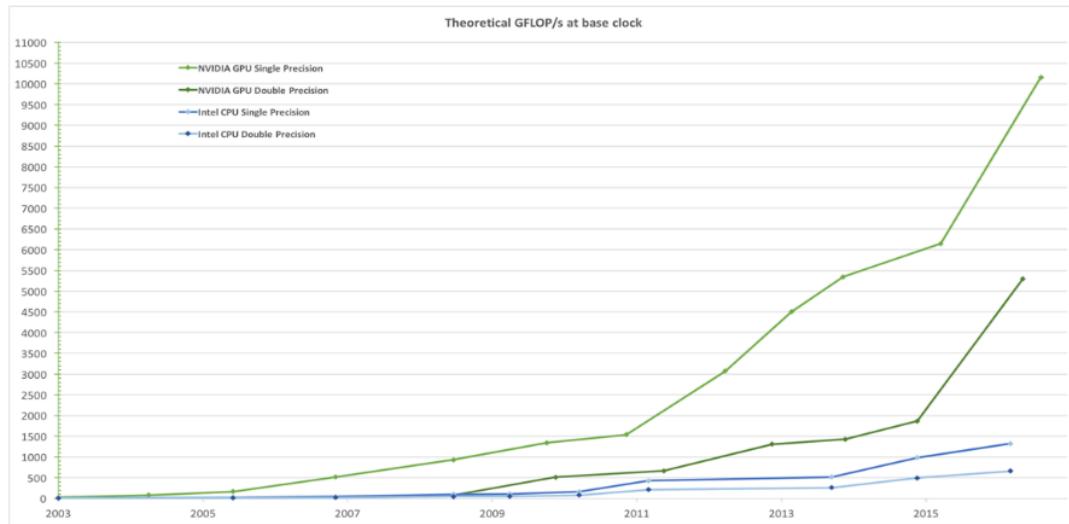
- Pedro Mário Cruz e Silva
  - Senior Solutions Architect
  - NVIDIA
    - GPU
    - CPU - arm
    - DPU
    - NIC
- AI for Science and Engineering

## AULA 10:

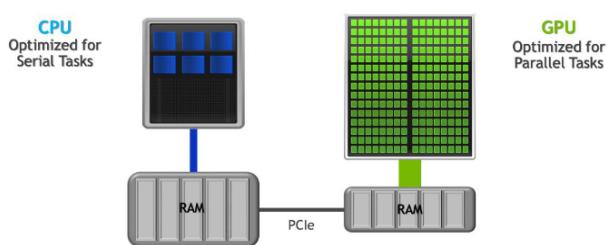
- Heurística
  - “Truque” usado para resolver um problema rapidamente
  - por velocidade, sacrificamos ao menos um entre: otimalidade, corretude, precisão, exatidão
  - boa heurística é suficiente pra obter resultados aproximados/ganhos de curto prazo
  - processo:
    - explorar alguma propriedade do problema
    - dividir em partes menores, que podem ser resolvidas rapidamente e combinar os resultados
  - Heurísticas para a mochila
    - pegar o mais caro primeiro
    - pegar o mais leve primeiro

## AULA 11:

- Desempenho em GFLOPS



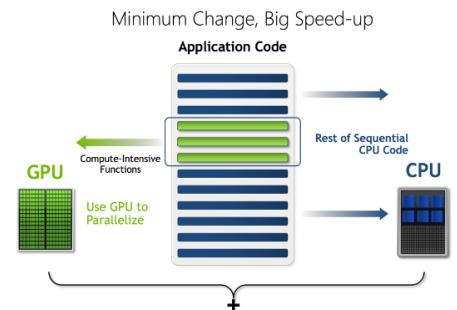
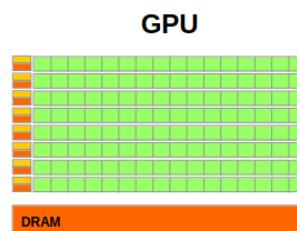
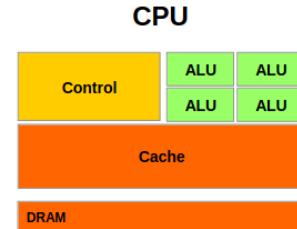
- CPU e GPUs



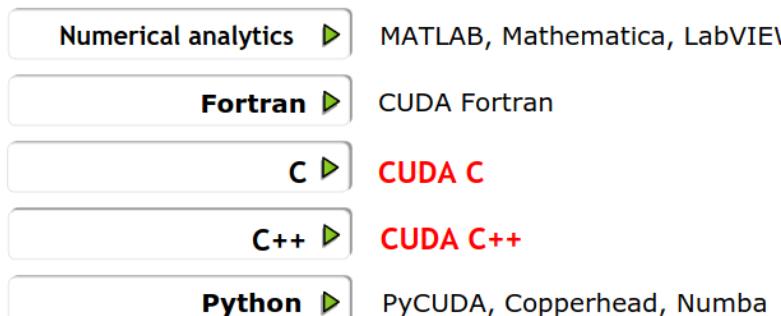
- Speed X Throughput



- CPU minimiza latência
  - ULA potente, minimiza latência das operações
  - cache grande:
    - acelera operações lentas de acesso à RAM
    - cache-misses são custosos
    - baixa performance / watt
- GPU maximiza throughput
  - ULA simples
    - eficiente energeticamente
    - alta taxa de transferência
  - cache pequeno
    - acesso contínuo a RAM
  - controle simples
  - número massivo de threads
- CPU X GPU
  - CPUs para partes sequenciais onde uma latência mínima é importante
    - CPUs podem ser 10X mais rápidas que GPUs para código sequencial
  - GPUs para partes paralelas onde a taxa de transferência (throughput) bate a latência menor
    - GPUs podem ser 10X mais rápidos que as CPUs para código paralelo

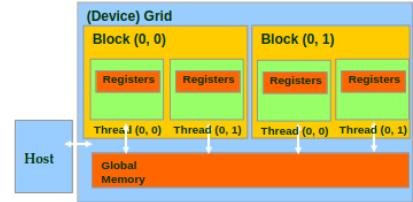


- Como usar a GPU:
  - Aplicações
    - bibliotecas: fácil de usar alto desempenho
      - Facilidade de uso: O uso de bibliotecas permite a aceleração da GPU sem conhecimento aprofundado da programação da GPU
      - Simplicidade: Muitas bibliotecas aceleradas por GPU seguem APIs padrão, permitindo aceleração com mudanças mínimas de código
      - Qualidade: As bibliotecas oferecem implementações de alta qualidade de funções encontradas em uma ampla variedade de aplicativos
    - diretivas de compilação: simples de usar portabilidade de código
    - linguagens de programação: maior desempenho, maior flexibilidade



- Programando para GPU
  - Compilador especial: nvcc
  - Endereçamento de memória separado
    - Dados precisam ser copiados de/para GPU
    - Isto leva tempo
  - Funções especiais (kernels) para rodar na GPU

- Memória em GPUs
  - código da GPU (device) pode:
    - cada thread ler e escrever nos registradores
    - ler e escrever na memória global
  - código da CPU (host) pode:
    - transferir dados de e para memória global
  - Fluxo dos programas
    - Parte 1: copia dados CPU → GPU
    - Parte 2: processa dados na GPU
    - Parte 3: copia resultados GPU → CPU
- Nvidia Thrust
  - Vantagens:
    - Simplifica transferências de memória
    - Duas operações customizáveis (reduce, transform)
    - Suporta OpenMP e CUDA
  - Desvantagens:
    - Limitado: menos recursos e desempenho que CUDA C
    - Só tem dois tipos de operações
    - Baseado em templates – difícil de debugar erros de compilação
  - Tipos de Dados
    - `thrust::device_vector<T>`
      - vetor genérico de dados na GPU
      - automaticamente alocado e desalocado
      - cópia é feita usando atribuição
    - `thrust::host_vector<T>`
      - vetor genérico de dados na CPU
      - pode ser substituído por containers da STL ou ponteiros “normais”
  - Iteradores
    - funcionam igual aos iteradores de `std::vector`
    - `v.begin()` // primeiro elemento
    - `v.end()` // último elemento
    - `v.begin() + 2` // `v[2]`
    - `i = v.begin() + 3; *i = 4; // v[3] = 4`
  - Redução
    - resume o vetor para um escalar
      - soma todos elementos
      - máximo/mínimo do vetor
      - contagens
      - suporta iteradores, o que torna a operação flexível
  - Transformação
    - operações elemento a elemento entre pares de vetores ou um só vetor.
      - aritmética ponto a ponto
      - permite criação de operações customizadas
      - suporta iteradores de entrada e saída
      - funciona também para operações locais (imagens)



```

thrust::host_vector<double> vec_cpu(10); // alocado na CPU
vec[0] = 20;
vec[1] = 30;

thrust::host_vector<double> vec_gpu (10); // alocado na GPU
vec_gpu = vec_cpu; // copia o conteúdo da CPU para GPU
thrust::device_vector<double> vec2_gpu (vec_cpu); // também transfere para GPU

```

```

thrust::device_vector<int> v(5, 0); // vetor de 5 posições zerado
// v = {0, 0, 0, 0, 0}
thrust::sequence(v.begin(), v.end()); // inicializa com 0, 1, 2, ...
// v = {0, 1, 2, 3, 4}
thrust::fill(v.begin(), v.begin() + 2, 13); // dois primeiros elementos = 3
// v = {13, 13, 2, 3, 4}

val = thrust::reduce(iter_comeco, iter_fim, inicial, op);
// iter_comeco: iterador para o começo dos dados
// iter_fim: iterador para o fim dos dados
// inicial: valor inicial
// op: operação a ser feita.

```

```

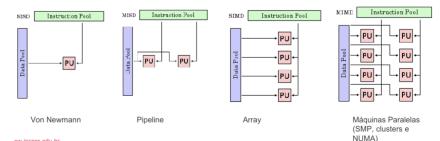
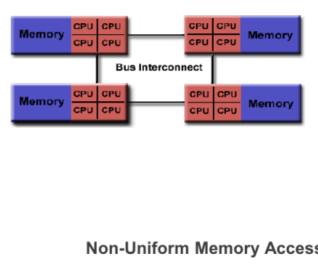
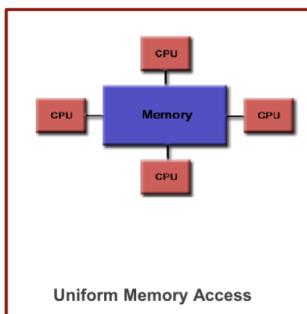
thrust::device_vector<double> V1(10, 0);
thrust::device_vector<double> V2(10, 0);
thrust::device_vector<double> V3(10, 0);
thrust::device_vector<double> V4(10, 0);
// inicializa V1 e V2 aqui

// soma V1 e V2
thrust::transform(V1.begin(), V1.end(), V2.begin(), V3.begin(), thrust::plus<double>());
// multiplica V1 por 0.5
thrust::transform(V1.begin(), V1.end(),
 thrust::constant_iterator<double>(0.5),
 V4.begin(), thrust::multiplies<double>());

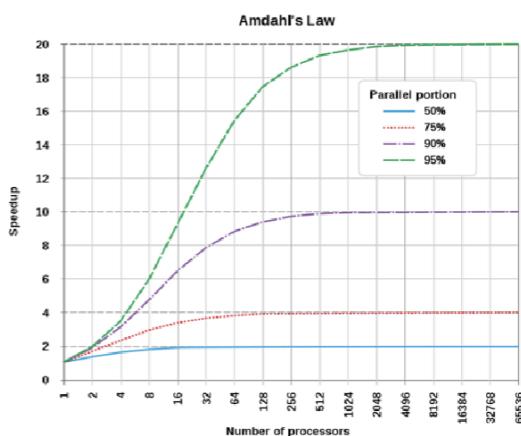
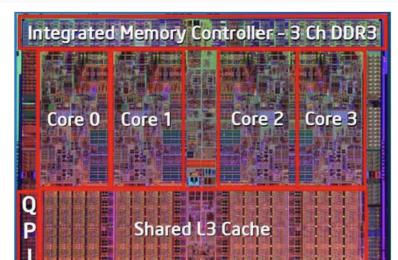
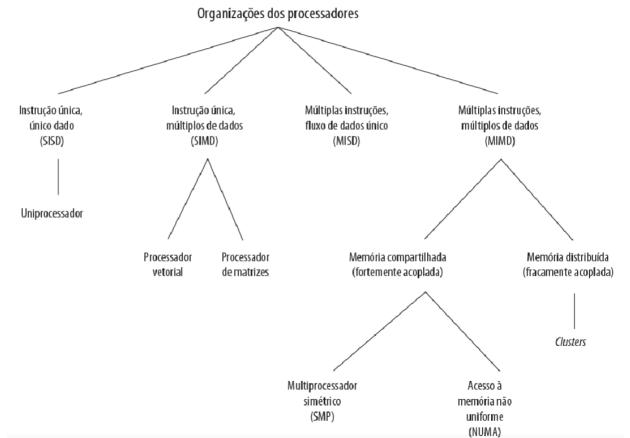
```

## AULA 12:

- Paralelismo
  - uso de múltiplos processadores ao mesmo tempo para resolver um problema
  - objetivo: aumento do desempenho, ou seja, a redução do tempo necessário pra resolver um problema
  - usar por 2 motivos: problemas mais complexos / maiores e clock dos processadores se aproximando dos limites ditados pela física
- Taxonomia de Flynn
  - forma de classificar computadores paralelos
  - baseia-se no fato de um computador executar uma sequência de instruções sobre uma sequência de dados
- Sistemas multi-core



### - Organização dos processadores



### Exemplo 1 – Supondo 8 cores

```
vector<double> dados;
vector<double> resultados;
for (int i = 0; i < dados.size(); i++) {
 resultados[i] = funcao_complexa(dados[i]);
}
```

### Exemplo 2 – Supondo 8 cores

```
vector<double> dados;
vector<double> resultados;
resultados[0] = 0;
for (int i = 1; i < dados.size(); i++) {
 resultados[i] = funcao_complexa(dados[i], resultados[i-1]);
}
```

Nenhum ganho! Depende da iteração anterior :(

## - Dependência

- uma iteração depende resultados calculados em iterações anteriores
- não existe nenhuma dependência em loop, por exemplo, diz-se que ele é ingenuamente paralelizável

## - Paralelismo

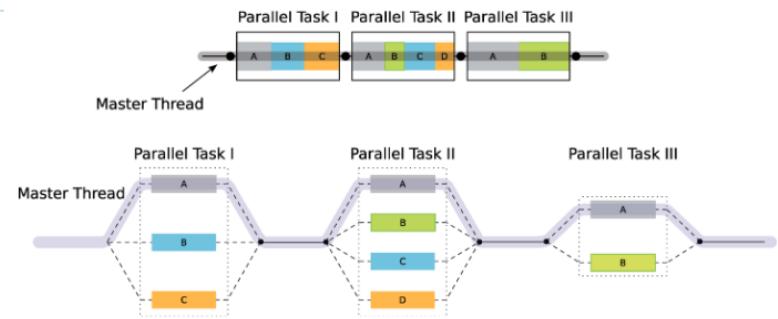
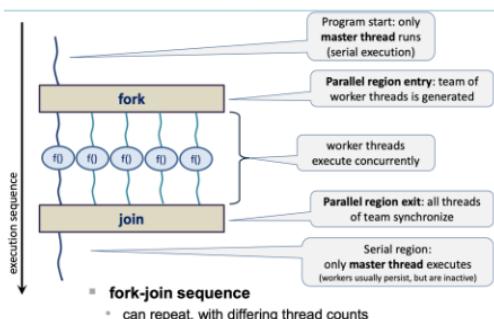
- Paralelismo de Dados: mesma operação (lenta) é executada para todos os elementos de um conjunto de dados (grande)
- Paralelismo de Tarefas: 2 ou mais tarefas independentes são executadas em paralelo, e se houver dependências, quebramos o problema em partes independentes e rodamos na ordem adequada

### - RESUMO:

- Parallelizar significa rodar código sem dependências simultaneamente
- Paralelismo de dados: mesma tarefas, dados diferentes
- Paralelismo de tarefas: heterogêneo
- Existem tarefas inherentemente sequenciais
- Ganhos são limitados a partes do programa

## - OpenMP

- conjunto de extensões para C/C++ e Fortran
- fornece construções que permitem parallelizar código em ambientes multi-core
- padroniza práticas SMP + SIMD + sistemas heterogêneos (GPU/FPGA)
- idealmente funciona com o mínimo de modificações no código sequencial



### - Sintaxe

```
// Arquivo interface da biblioteca OpenMP para C/C++
#include <omp.h>

// retorna o identificador da thread.
int omp_get_thread_num();

// indica o número de threads a executar na região paralela.
void omp_set_num_threads(int num_threads);

// retorna o número de threads que estão executando no momento.
int omp_get_num_threads();
```

| Name                                     | Result type | Purpose                                                                        |
|------------------------------------------|-------------|--------------------------------------------------------------------------------|
| omp_set_num_threads<br>(int num_threads) | none        | number of threads to be created for subsequent parallel region                 |
| omp_get_num_threads()                    | int         | number of threads in currently executing region                                |
| omp_get_max_threads()                    | int         | maximum number of threads that can be created for a subsequent parallel region |
| omp_get_thread_num()                     | int         | thread number of calling thread (zero based) in currently executing region     |
| omp_get_num_procs()                      | int         | number of processors available                                                 |
| omp_get_wtime()                          | double      | return wall clock time in seconds since some (fixed) time in the past          |
| omp_get_wtick()                          | double      | resolution of timer in seconds                                                 |

```
// Cria a região paralela. Define variáveis privadas e
// compartilhadas entre as threads.
#pragma omp parallel private(...) shared(...)
{ // Obrigatoriamente na linha de baixo.

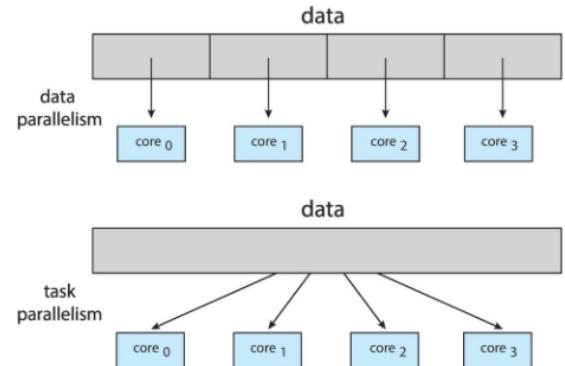
 // Apenas a thread mais rápida executa.
 #pragma omp single

}
```

## • Exemplo 3 – Supondo 8 cores

```
vector<double> dados;
vector<double> resultados1;
vector<double> resultados2;
resultados1[0] = resultados2[0] * 0;
for (int i = 1; i < dados.size(); i++) {
 resultados1[i] = funcao_complexa(dados[i], resultados1[i-1]);
 resultados2[i] = funcao_complexa2(dados[i], resultados2[i-1]);
}
```

Podemos calcular resultados1 e resultados2 paralelamente



Código sequencial

```
for(i = 0; i < N; i++)
 a[i] = a[i] + b[i];
```

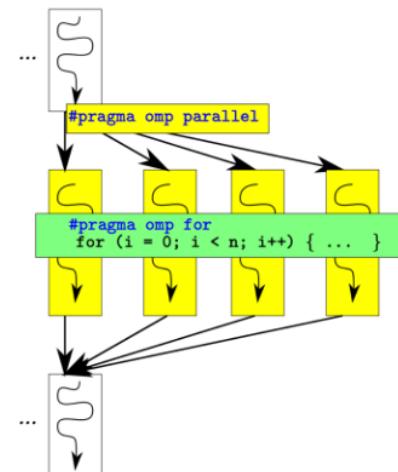
Região OpenMP parallel

```
#pragma omp parallel
{
 int id, i, Nthrds, istart, iend;
 id = omp_get_thread_num();
 Nthrds = omp_get_num_threads();
 istart = id * N / Nthrds;
 iend = (id+1) * N / Nthrds;
 if(id == Nthrds-1) iend = N;
 for(i = istart; i < iend; i++)
 a[i] = a[i] + b[i];
}
```

Região paralela OpenMP com uma construção de divisão de laço

```
#pragma omp parallel
#pragma omp for
for(i = 0; i < N; i++) a[i] = a[i] + b[i];
```

[www.inf.ufpr.edu.br](http://www.inf.ufpr.edu.br)



### - Variáveis

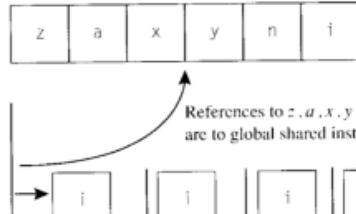
Global shared memory



Serial execution  
(master thread only)

All data references are to global shared instances

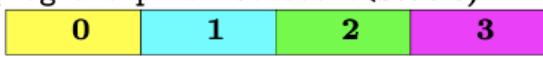
Global shared memory  
Parallel execution  
(multiple threads)



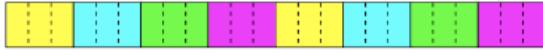
References to z, a, x, y, n are to global shared instances

The behavior of private variables in an OpenMP program.

#pragma omp for schedule(static)



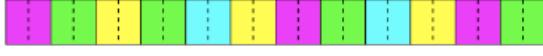
#pragma omp for schedule(static,3)



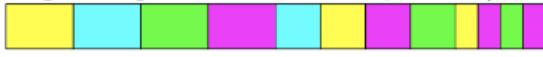
#pragma omp for schedule(dynamic)



#pragma omp for schedule(dynamic,2)



#pragma omp for schedule(guided)



#pragma omp for schedule(guided,2)



### - O conceito de redução

```
for(i=1; i<=n; i++){
 sum = sum + a[i];
}
```

#pragma omp parallel for reduction(+:sum)

```
{
 for(i=1; i<=n; i++){
 sum = sum + a[i];
 }
}
```

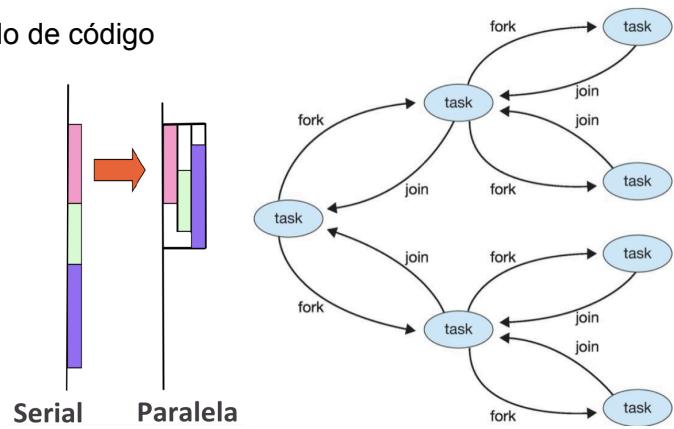
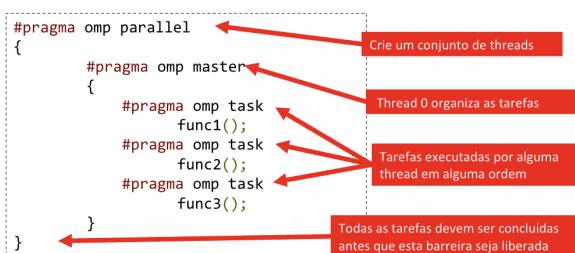
## AULA 13:

### - OpenMP

#### - Sections

- divide o trabalho de forma não iterativa em seções separadas, onde cada seção é executada por uma “thread” do grupo. Representa a implementação de paralelismo funcional
  - define a seção do código sequencial onde será definida as seções independentes, através da section;
  - cada section é executada por uma thread do grupo;
  - existe um ponto de sincronização implícita no final da section

- se existirem mais threads do que seções, o OpenMP decidirá, quais threads executarão os blocos de section, e quais, não executarão
- O que são tasks
  - definida em um bloco estruturado de código
  - podem ser aninhadas: uma task pode gerar novas tasks
  - cada thread pode ser alocada para rodar uma tarefa
  - não existe ordenação no início das tarefas
  - tarefas são unidades de trabalho independentes
- Tarefas em OpenMP



- Taskwait

```

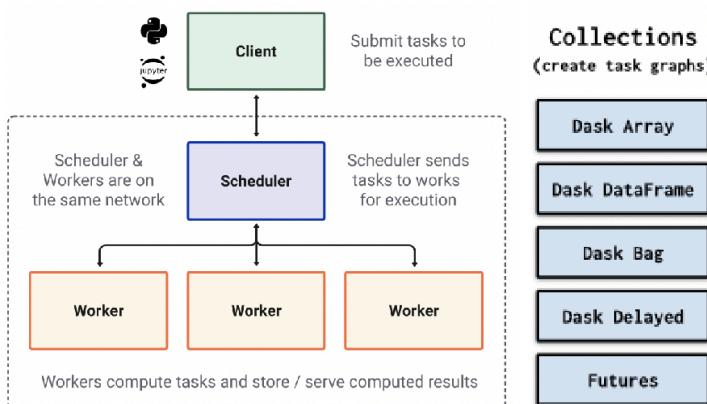
#pragma omp parallel
{
 #pragma omp single
 {
 #pragma omp task fred();
 #pragma omp task daisy();
 #pragma taskwait
 #pragma omp task billy();
 }
}

```

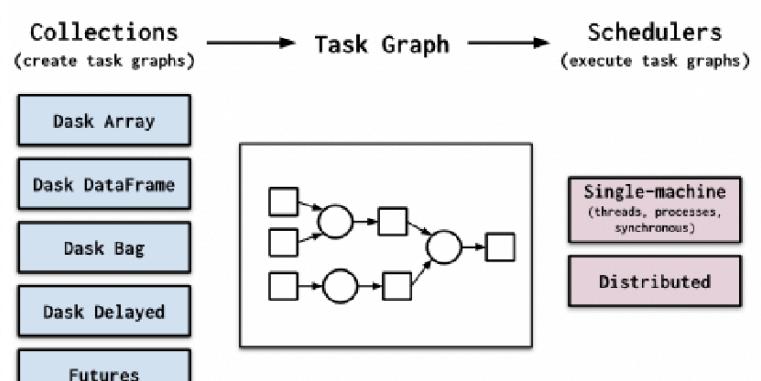
fred() and daisy() must complete before billy() starts

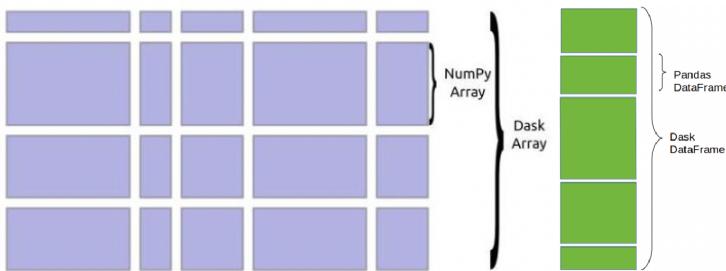
## AULA 15:

- Meios de implementar paralelismo:
  - GPU, OpenMP, MPI
  - indo para o alto nível: PySpark, Dask, Joblib, Numba
- Dask
  - Manipulação Eficiente de Grandes Conjuntos de Dados
  - Abstração de Paralelismo
  - Compatibilidade com Ecossistema Python
  - Escalabilidade
  - Aplicabilidade em Ciência de Dados e Análise
  - Preparação para o Mundo Profissional
  - Variedade de Tarefas Suportadas
  - Aprendizado Contínuo em Computação Paralela
  - Aplicações em diversos nichos: geoespacial, finanças, astrofísica, microbiologia, ciência ambiental
- Paralelismo com Dask



## - Componentes do Dask

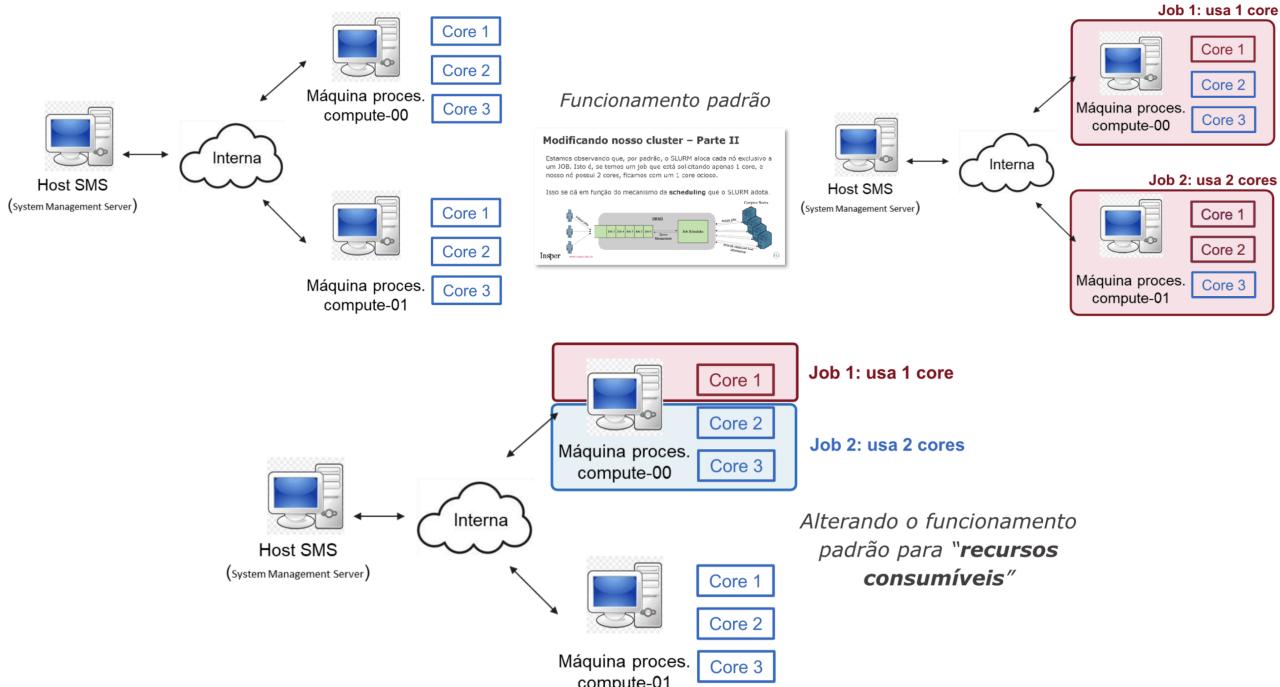
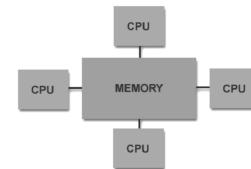




- componentes
  - Dask Array
  - Dask DataFrame
  - Dask Bag (usados para paralelizar cálculos simples em dados não estruturados, como textos, logs, JSONs, objetos python)
  - Dask Delayed
  - Futures

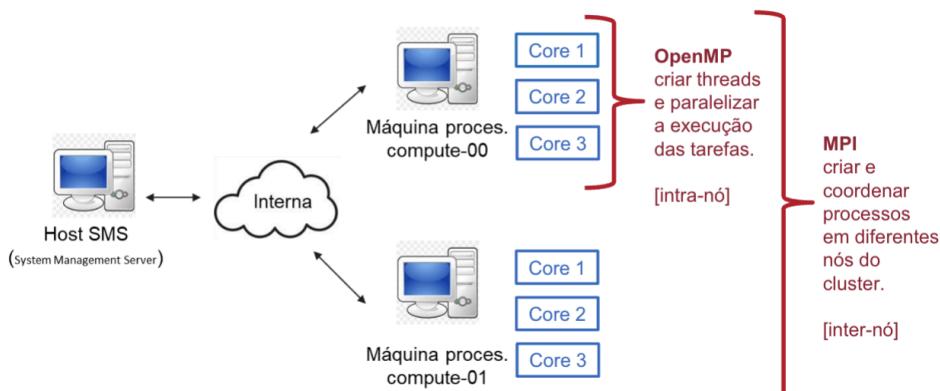
## AULA 17:

- Até onde vai OpenMP?
  - focado em ambientes de memória compartilhada
  - “paralelismo local”
  - estou “usando ao máximo” a máquina que tenho
- E o nosso cluster?



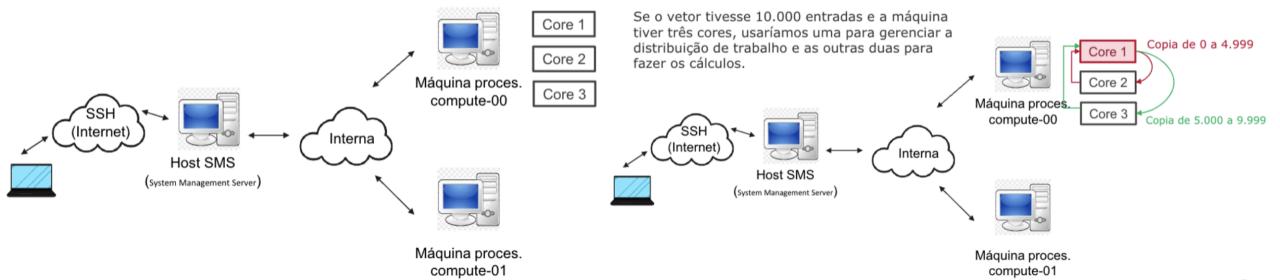
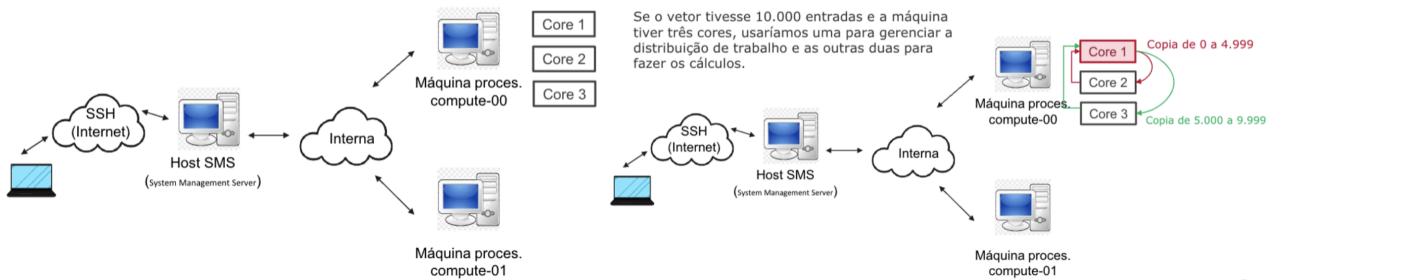
- OpenMP: paralelismo local, com memória compartilhada
- Cluster: paralelismo de tarefas, em memória distribuída
- Podemos combinar ambos? Sim, MPI.

- Combinando OpenMP com MPI



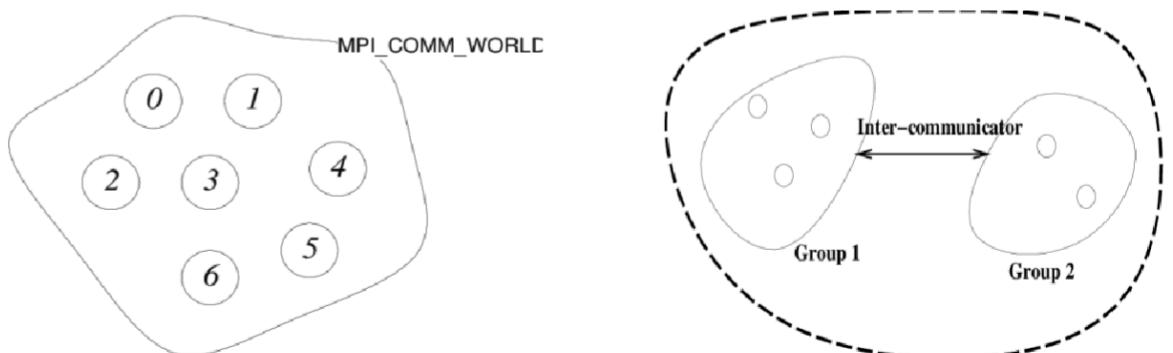
- MPI: Message Passing Interface

- é um padrão da indústria, não uma linguagem/implementação
- assume que não há compartilhamento de memória
- o programador deve:
  - dividir os dados
  - trabalhar com interações são bilaterais (send / receive)
  - reduzir comunicação pra otimizar desempenho
- Posso usar MPI localmente? (em ambiente de memória compartilhada)
  - Exemplo (local)



- Conceitos

- Rank: todo processo tem uma identificação atribuída pelo sistema quando o processo é iniciado. Começa no 0 até n-1 processos.
- Group: conjunto ordenado de N processos. Todo e qualquer grupo é associado a um "communicator" e, inicialmente, todos os processos são membros de um grupo com um "communicator" já pré-estabelecido (MPI\_COMM\_WORLD).
- Communicator: define um grupo, que poderão se comunicar entre si (contexto). MPI usa combinação de grupo e contexto para garantir comunicação segura e evitar problemas no envio de mensagens entre os processos.



- Código | Execução

- Include
  - `#include<mpi.h>`
- Compilação
  - `mpic++ programa.cpp -o programa`
- Execução (básica)
  - `mpirun -np <num processos> ./programa`

Na execução:

- `-np` especifica quantidade de processos a serem criados; cada um executa uma cópia do executável (SPMD)
- Há limites para `-np`: quantidade de slots disponíveis na arquitetura onde o executável será executado. `Slots` representam quantidade de processadores físicos disponíveis.
- `--use-hwthread-cpus` permite usar também os processadores lógicos
- `--oversubscribe` permite mais de um processo por `slot`
- `--hostfile <hostfilename>`, especifica nós que podem ser usados para atribuir os processos gerados (mapeamento de processos em processadores).
  - O parâmetro `hostfilename` é um arquivo texto com endereços ou nomes dos nós (hosts/máquinas). Um hostname por linha.
  - Os nós podem ser especificados sem o `--hostfile` (usar `--host=host-H`)

## <argumentos do programa>

- Tipos de dados

| Tipo do MPI            | Tipo do C                     |
|------------------------|-------------------------------|
| MPI_CHAR               | char                          |
| MPI_SHORT              | short int                     |
| MPI_INT                | int                           |
| MPI_LONG               | long int                      |
| MPI_LONG_LONG_INT      | long long int                 |
| MPI_UNSIGNED_CHAR      | unsigned char                 |
| MPI_UNSIGNED_SHORT     | unsigned short int            |
| MPI_UNSIGNED           | unsigned int                  |
| MPI_UNSIGNED_LONG      | unsigned long int             |
| MPI_UNSIGNED_LONG_LONG | unsigned long long int        |
| MPI_FLOAT              | float                         |
| MPI_DOUBLE             | double                        |
| MPI_LONG_DOUBLE        | long double                   |
| MPI_WCHAR              | wide char                     |
| MPI_PACKED             | special data type for packing |
| MPI_BYTE               | single byte value             |

- Funções básicas

- int MPI\_Init(int \*argc, char \*\*\*argv)
- int MPI\_Finalize()
- int MPI\_Comm\_size(MPI\_Comm comm, int \*size)
- int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)
- int MPI\_Send(void bufferE, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)
- int MPI\_Recv(void bufferR, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status status).

## MPI : Exemplo

Mensagem=dado(3 parâmetros de informações) + envelope(3 parâmetros de informações)

Ex.: CALL MPI\_SEND(sndbuf, count, datatype, dest, tag, comm, mp ierr)

CALL MPI\_RECV(recvbuf, count, datatype, source, tag, comm, status, mp ierr)

```
#include <iostream>
#include <cstring>
#include <mpi.h>

int main(int argc, char **argv) {
 char message[20];
 int i, rank, size, type = 99;
 MPI_Status status;

 // Inicializando o MPI
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size); // Obtém o número total de processos
 MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Obtém o rank do processo atual

 if (rank == 0) {
 // Processo com rank 0 envia "Hello, world" para os outros processos
 strcpy(message, "Hello, world");
 for (i = 1; i < size; i++) {
 MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
 }
 } else {
 // Outros processos recebem a mensagem do processo com rank 0
 MPI_Recv(message, 13, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
 std::cout << "Message from node " << rank << ":" << message << std::endl;
 }

 // Finalizando o MPI
 MPI_Finalize();
 return 0;
}
```

## AULA 18:

- Formas de comunicação

- Bloqueante vs Não bloqueante
- aspectos de hardware influenciam fortemente:
  - há buffers associados ao send ou ao receive?
  - há um hardware específico pra realizar a comunicação em paralelo à CPU?

- Forma mais comum de comunicação entre 2 processos no MPI é através de Sends e Receives, implementados pelas funções MPI\_Send e MPI\_Recv.
- Em teoria, são bloqueantes
- A maior parte das implementações do padrão MPI, provê um buffer para que o send não seja bloqueante
- Ao chamar MPI\_Send, a mensagem é copiada em um buffer e a execução do programa continua. Aí quando o destinatário chama MPI\_Recv, ele lê a mensagem do buffer.
- Em implementações onde não há o esse buffer, deve-se usar MPI\_Bsend, onde o buffer deve ser explicitamente alocado

#### - Broadcast e Reduce

- Mandar uma mensagem para todos os processos: MPI\_Bcast
- Receber mensagens dos nós filhos: MPI\_Reduce

#### - Broadcast

`MPI_Bcast(buffer, count, datatype, root, comm)`

**buffer**

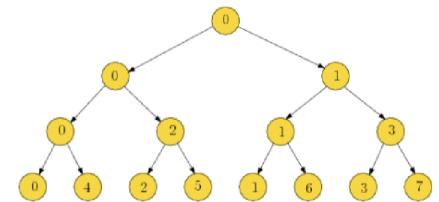
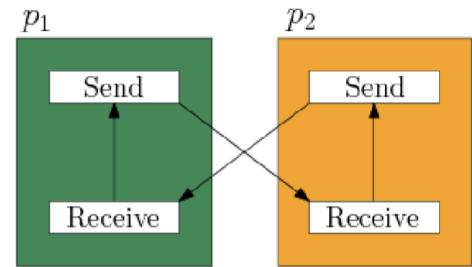
data to be distributed

**count**

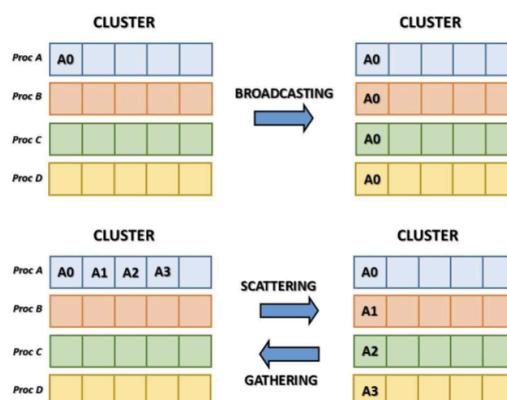
number of entries in buffer

**datatype**

data type of buffer



#### - Broadcast vs. Scatter / Gather



#### - Scatter / Gather

- Root manda uma quantidade igual de dados para todos os outros processos

`MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount,`

`recvtype, root, comm)`

**sendbuf**

send buffer (data to be scattered)

**sendcount**

number of elements sent to each process

**sendtype**

data type of send buffer elements

**recvbuf**

receive buffer

**recvcount**

number of elements to receive at each process

**recvtype**

data type of receive buffer elements

**root**

rank of sending process

**comm**

communicator

- Enviando um vetor: "Send / Recv" Vs "Scatter"

```
#include <iostream>
#include <mpi.h>

int main(int argc, char *argv[]) {
 MPI_Init(&argc, &argv);

 int rank, size;
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 const int ARRAY_SIZE = 100;
 int array[ARRAY_SIZE];

 if (rank == 0) {
 for (int i = 0; i < ARRAY_SIZE; i++) {
 array[i] = i;
 }
 }

 int chunk_size = ARRAY_SIZE / size;
 int recv_array[chunk_size];

 MPI_Scatter(array, chunk_size, MPI_INT, recv_array, chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

 // Imprime o vetor recebido
 std::cout << "Nº " << rank << " recebeu: ";
 for (int i = 0; i < chunk_size; i++) {
 std::cout << recv_array[i] << " ";
 }
 std::cout << std::endl;

 MPI_Finalize();
 return 0;
}

#include <iostream>
#include <mpi.h>

int main(int argc, char *argv[]) {
 MPI_Init(&argc, &argv);

 int rank, size;
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 const int ARRAY_SIZE = 100;
 int array[ARRAY_SIZE];

 if (rank == 0) {
 for (int i = 0; i < ARRAY_SIZE; i++) {
 array[i] = i;
 }
 }

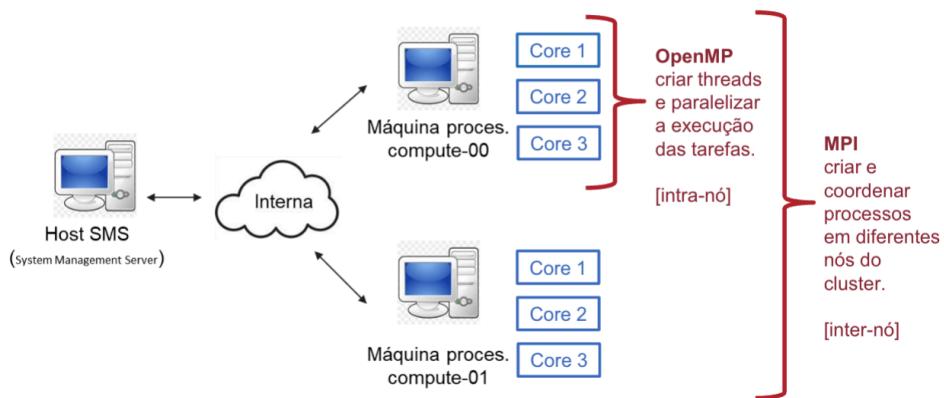
 int chunk_size = ARRAY_SIZE / size;
 int recv_array[chunk_size];

 if (rank == 0) {
 for (int i = 1; i < size; i++) {
 MPI_Send(&array[i * chunk_size], chunk_size, MPI_INT, i, 0, MPI_COMM_WORLD);
 }
 } else {
 MPI_Recv(recv_array, chunk_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 // Imprime o vetor recebido
 std::cout << "Nº " << rank << " recebeu: ";
 for (int i = 0; i < chunk_size; i++) {
 std::cout << recv_array[i] << " ";
 }
 std::cout << std::endl;
 }

 MPI_Finalize();
 return 0;
}
```

## AULA 19:

- Combinando OpenMP com MPI



- Exemplo: multiplicação de elementos

- Tarefa: calcular o quadrado de cada elemento em um array bidimensional
- MPI: Divide o array entre diferentes processos
- OpenMP: Paraleliza o cálculo dentro de cada processo

```
#include <iostream>
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {
 MPI_Init(&argc, &argv);

 int rank, size;
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 const int N = 10; // Dimensões do array bidimensional
 int data[N][N];

 // Inicialização do array pelo processo 0
 if (rank == 0) {
 for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
 data[i][j] = i + j;
 }
 }
 }

 // Dividir o array entre os processos
 int chunk_size = N / size;
 int local_data[data][N];
 MPI_Scatter(data, chunk_size * N, MPI_INT, local_data, chunk_size * N, MPI_INT, 0, MPI_COMM_WORLD);

 // Paralelização com OpenMP
#pragma omp parallel for collapse(2)
 for (int i = 0; i < chunk_size; i++) {
 for (int j = 0; j < N; j++) {
 local_data[i][j] *= local_data[i][j]; // Calcula o quadrado do elemento
 }
 }

 // Reunir os resultados no processo 0
 MPI_Gather(local_data, chunk_size * N, MPI_INT, data, chunk_size * N, MPI_INT, 0, MPI_COMM_WORLD);

 // Processo 0 imprime os resultados
 if (rank == 0) {
 for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
 std::cout << data[i][j] << " ";
 }
 std::cout << std::endl;
 }
 }

 MPI_Finalize();
 return 0;
}
```