Problems Prediction

<1> Operational Intensity & Roofline Model

① # FlOP : floating point operation, double (unless noted)

② Arithmetic (Operational Intensity) $I = \frac{W}{Q} = \frac{\#\text{FlOPs}}{\#\text{Bytes}}$

针对算法的，对一个 algorithm's metric，算法的本征量，与计算平台无关 [float = 4 byte (32 bits), double = 8 byte]

W: amount of work / i.e. floating point operations required

Q: memory transfer / i.e. access from DRAM to lowest cache level

W: 直接计算访问 + - * / 的数目 → 配上循环

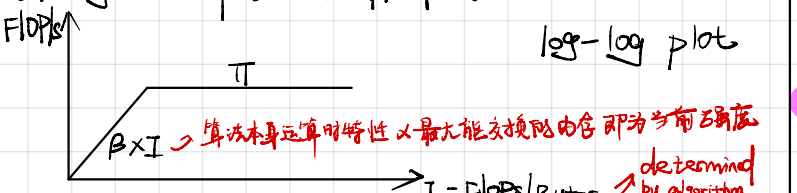
Q: Depends on cache size
- No cache (reading directly from memory): 每一次数据访问都被 counted
- perfect cache: 数据只用读取+写入一次

每一次循环都只读取相应行列 → realistic cache

⇒ 若忘写入算不算访问 ⇒ hw01

★计算时注意数据类型，结果自用 OI = O(N) ⇒ tutorial 02 slides

③ Roofline model: estimate how efficient is our code based on a given computational platform



① CPU peak performance of this platform: π = FlOP/s (compare bound)

② RAM memory bandwidth (β = bytes/sec) (memory bound)

↓ ? How to estimate π and β based on the hardware specification?

π = clock speed × arithmetic units × Vec × cores

vectorization: AVX512 bit → 8 double / 16 float; AVX256 bit → 4 double / 8 float

β = clock speed × 6.5 bytes/cycle → 直接用也行

<2> Amdahl's law

- Metrics for parallel computing
  - → $t(1)$: execute time on a single core
  - → $t_j(N)$: execute time on processor out of N cores

Speed up (j samples): $S(N_{proc}) = \frac{t(1)}{\text{mean}_j \max_i t_{ij}(N_{proc})} \times \frac{N(Proc)}{N(1)}$

$N(Proc)$, $N(1)$ → problem size → linear speedup N_{proc}; super linear speedup $S > N_{proc}$

- Strong scaling & Weak scaling
  - ↳ 是我们对并行计算性能分析的一种手段
  - → Strong scaling: 问题尺度不变，增加并行单元，观察性能变化，用 Speed up 来度量
  - → weak scaling: 问题尺度随着并行单元数等比变化，理想上来说，时间不变。用 $\frac{t(1)}{t(N_{proc})}$ 度量效率来表征

- Amdahl's law

$$S_s(N) = \frac{t_s}{f t_s + (1-f)\frac{t_s}{N}} = \frac{N}{1+(N-1)f}$$

⇒ 理想化 strong scaling t_s/N

⇒ problem-size fixed

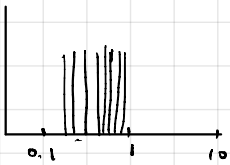
⇒ partial {parallel / serial}

Problem: Operational Intensity & Roofline model

hw01 : 把读入和写出分开来算，注意，若 C[n×n]，在读入时可能进行了 n^2 次，写出时则可能只有 n 次。

hw01: Be careful when calculating π → should use consider vectorization. and vectorization → float/double.

单位: W → flops Q → bytes



Mock Exam: @Draw Roofline model

①计算 0.1 时把 n 记得带上

② 给出的 iteration scheme 和最后代码不同，我们可以将一些常数的组合假设成常数，从而减少计算强度 ⇒ 在 Mock exam，我们没有将 constant 的载入算作一次 memory access。同理，要从具体代码设计出发，求出多少次 memory access.

Problem: Amdahl's law

HW1 : 当考虑 communication cost 0.001(n+1)

$$S_{cn} = \frac{T(1)}{T(n)} = \frac{t_s}{f t_s + (1-f)\frac{t_s}{N} + 0.001(n+1)t_s}$$

communication is related to serial computation

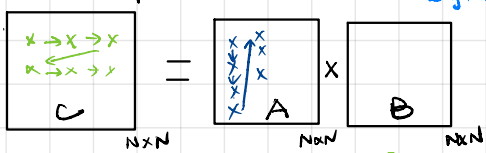
<3> Cache

CPU | register | cache | 内存 RAM

1. Cache Optimization → Linear Algebra Operations

effects of storing and using data in different ways on the efficiency of the code. 见 HW1 HW3 → 我们在FD中 advance() 更新时可 row-wise 更新，访问

① Hadmard product of 2 matrix



↓ 前提，我们数据是 row-wise 存在一维 vector 中

for i → N{ for j → N }

row-wise: C[i·N+j] = A[i·N+j] + B[i·N+j]

column-wise: C[j·N+i] =

↳ 也可 index 不变，循环角标换顺序

- 两种不同形式的数据访问。第一种更快，因为数据在 cache 存储是 row-wise，这一次访问和下次访问都可以在 cache 中找到，不涉及 memory ⇒ cache 的频繁交换
- 常用 std::swap(,), avoiding coping memory

Dwarfs

<1> Diffusion Equation and Finite Difference

Logic of code

- Define a struct Diagnostics → Store $\hat{C}$ and t (总浓度)
  → 之后画全局扩散曲线图用
- Define a Diffusion2D class / construct
  → 生成向量 dimension → consider ghost information

member function ↓

Diffusion2D { 初始化函数，与类同名，初始参数值给 private

↓

advance() { 依据迭代格式进行迭代，often loop，2D 就是双 loop Euler + Finite Difference [can be parallel by openMP → collapse(2) and MPI local_N]

↓

Compute diagnostic() { 计算总浓度，写一个循环即可 [涉及 reduction 操作] 通常也会判断 write_diagnostics ⇒ do not compute ghost node

↓

initialization() { 根据 initial values 初始化 [也可涉及到循环，用 openMP collapse(2)]

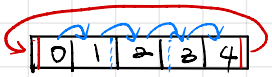
↓

Compute histogram { 画出浓度分布图。⇒ OpenMP 里需要用到 max and min reduce 从而得到 global max and global min. Then, using MPI_Allreduce

• In MPI

achieved in the advance()

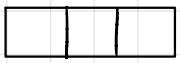
1. We need to exchange the ghost information. After that, we can use Diffusion2D.advance()
2. main() 函数中实现 MPI_Init(), rank, size.
   By the way D L N 这些系统参数在 compile 时指定。
3. For local_N, achieve it in the Initialization function (来初始化)
   ↓ number of grid points of this process
4. 初始化函数最后一步有 initialize(), 会根据 rank 不同进行 initialization, initialize 是加了 ghost 之后的值



信息传递时会出现经典的 cyclic communication, 有 send recev or 奇偶分开 → 见 MPI (II) more on blocking point-to-point communication

advance()

Send-receive : Send of a message to one destination receive a message from another process.



U[1] → Send previous rank
U[0] → receive previous rank

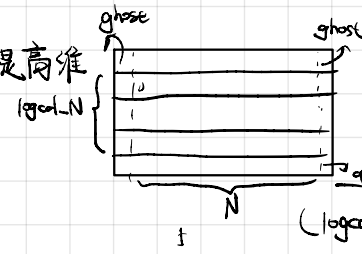
⇓ 见 mock exam coding MPI

U[N] → Send
U[N+1] → receive

HW5 Solution

↳ 处理两种边界 { ① 固定边界，均用的是 MPI_PROC_NULL 给 pre_rank 和 next_rank 赋值
② 周期性边界 { next = (rank+1) % size;
prev = (rank+size-1) % size

如果是高维



index → C[i*(N+2)+j], N

⇒ 作业 HW5 没有交换两侧 ghost，没什么问题，因为两侧固定边界 (local_N+2)(N+2)

5. Compute_diagnostic()

See HW5 (2D) / mock exam (1D)

- We can use MPI_IN_PLACE reduction

6. Compute_histogram()

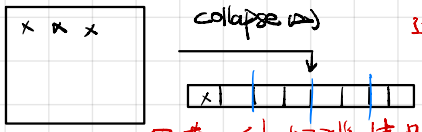
- Define variable global_max globle_min
- use MPI_Allreduce() for message transmission between all ranks → See HW5

In OpenMP

1. In advance() ① provide a cache-friendly implementation (i) row-wise() (ii) read and store (iii) use std::swap() (iv) use C to update C_temp, then swap
   ② #pragma omp parallel for collapse(2)

2. Compute diagnostic
   ① #pragma omp parallel for collapse(2) reduction(+ : amount)

3. Compute_histogram HW2 Julia sets
   ① 用 reduction，但是 min max 见 HW3
   ② Accumulate equispaced bins
   Original code: for(i→N){for(j→N) bin= histbin++}



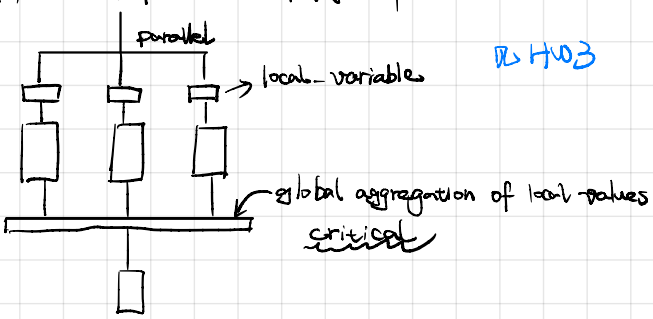
这种 shared memory ++ 很容易发生 race condition.

思考一种极端情况，两个 thread 同时执行这个 ++ 操作，但由于操作不是原子的，且两个接近同时发生，最后加 2 次，实际只有 1 次。

! 总结，多 thread 可能会对共享变量同时做操作 ! !

↓ 2 solutions

① 对 hist[bin]++ 做 atomic：也就是很快完成
② 在进入 #pragma omp parallel 之后创建私有局部变量，之后再用 critical 执行合并



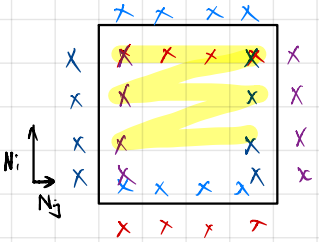
见 HW3

<补充，CSR format> HW6

Q1 : CSR format

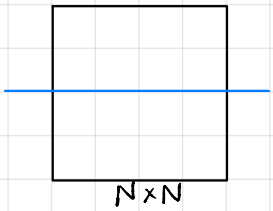
1. index for K, J is from 0

$$U_{0,0}^{n+1} = U_{0,0}^{n} + \frac{\Delta t}{h}(U_{0,1} + U_{0,N_j-1} + U_{1,0} + U_{N_i-1,0} - 4U_{0,0}^{n})$$



<2> Linear algebra operation with A splitting MPI

- 核心问题是 local index coordinate 与 global index coordinate



int myN = N / size

⇒ for rank 0 → local = global

for rank 1 ⇒ local 模拟 + myN×rank

(i_local + myN×rank, j) = (i, j)

之后关于它们操作全以此为依据，include

<1> 与 (i,j) 有关的 initialization

↓ 接着，当我们分 rank 计算完短 vector 时，为了不影响向针对这个 vector 之后的操作，我们可以用

↓ MPI_Allgather 再拼回来 (就作为例如 norm 操作)

↓ 每经过一次 MA 操作分一次，再拼一次进行操作

See Mock exam

HW6 给出更加复杂的过程，进行的不仅仅是 row，而是 block 的操作，下面给出一点分析

$u^{n+1} = u^n + Au^n$

3	4	5
rank 0	1	2

A

u

- 由于整个过程不涉及对向量所有或分未解，则我们不用 gather，直接全分块走到底。
将 u^n 也根据所对应 rank 分块，A 根据所对应 rank 离散化，整个程序一条路走到底。最后写入的时候再拼回原来的 u^n.

<3> Brownian Motion Based on OpenMP

HW2

① Calculate time based on OpenMP

double GetWtime() { return omp_get_wtime(); }

② 会涉到 Histogram 计算，注意用之前提到2种策略

② #include <omp.h>

<4> Dimensionality reduction with PCA

HW3 Q2

这里PCA通过 covariance method 得到，其本质就是 X^TX 再做 Eigen decomposition. Actually, we can do SVD based on X. to help us achieve PCA.

Covariance Matrix: $C = \frac{1}{N-1} X^T X$ (Here, data is a matrix)

<1> Transpose Data

<2> compute mean (every dimension)

<3> Compute std. $\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$ (HW3 Q2 中是 N-1)

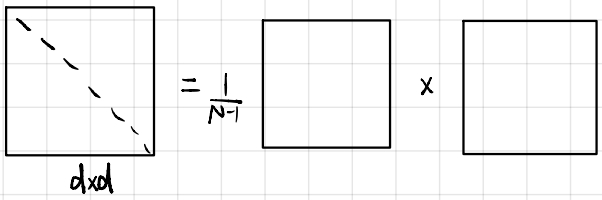
<4> Standardization / Normalization of Data

(Transormation of the data to zero mean unit variance)

$\frac{\text{Data}[i] - \text{mean}}{\sigma}$

※ 注意数据分 dimension 则每个 dimension 依次求

<5> 当我们 Standardization 矩阵后，便可以对其求 covariance Matrix.

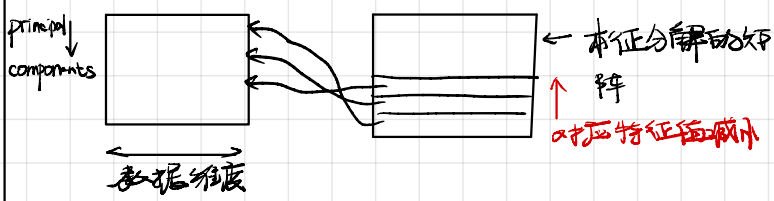


↑ Due to its symmetry，整个过程是3个 loop 嵌套。

<6> Compute the eigenvalues and eigen vectors

这里使用了 LAPACK.

<7> Extract the principal components & eigen values



<5> Monte Carlo

motivation: numerical Quadrature for high dimension problem

Using Composite Trapezodial Rule, we found

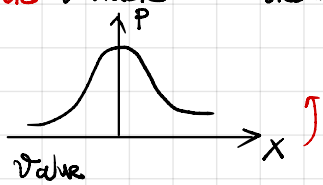
$\varepsilon = O(M^{-2/d})$ ↑ $M = n^d$ ⇒ # point evaluation

With higher dimension, the order of convergence decrease

↳ Curse of dimensionality. possible solution → Monte Carlo Integration

↓ introduce Probability

- Stochastic variable → described by PDF



关心的变量(随机)处于各个值的可能性大小

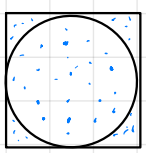
↓ Expected Value

即在这样一个随机可能处的范围下，我们对其期望是多少

↳ variance，可能随机的结果相对于我们期望之间是否集中

↓ Sampling: uniform distribution $\sim U(0,1)$

$x = a + (b-a)u$



→ $A = \int_{-1}^{1}\int_{-1}^{1} f(x,y)\,dx\,dy$ $f(x) = \int_0^1$

$O(M^{-1/2})$ → 误差相对采样点变化

↓ 到解 If we want to solve $\int_a^b f(x)dx$

Assuming stochastic variable $X \sim U(a,b)$ $p(x) = \frac{1}{b-a}$

$E[f(X)] = \frac{\int_a^b f(x)dx}{b-a}$ ⇒ $(b-a)E[f(x)] = \int_a^b f(x)dx$

↳ convert Integral into expected value

Assuming a variable $Y = f(X)$

生成N个 [a,b] 上随机数，代入 f(x) 得到生成的N个随机Y，则离散情况是 Expected = $\frac{\sum f(x_i)}{n}$ ⇒ $\int_a^b f(x)dx = \frac{b-a}{n}\sum f(x_i)$

HW4 (2) → based on MPI

t_1 = MPI_Wtime()

MPI

common code:

```
mpic++ -O3 xxx.cpp -o xxx
mpiexec -n 4 ./xxx
#include <mpi.h>
MPI_Init(&argc, &argv)
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Finalize();
```

```
#include <cassert>
...
assert(N % size == 0);
```

→ group communication

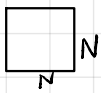
① Gather 这种操作的结果是 recvbuf 被拼成了一个很长的结果集，顺序以rank的顺序为主

1. SPMD programming model P6
2. Point-to-Point communication P18
3. MPI_Send MPI_Recv P19 → 细节 P21-P23
4. MPI_Send communication modes P24 通常用 if 语句判断 rank 来辅助执行
   ↓
   Example: data exchange → dead lock P25-P28
5. MPI 常用手段 Split

Monte Carlo

```
n_local_size = double(N)/procs;
n_start = rank * n_local_size;
n_end = rank==(procs-1) ? N : n_start + n_local_size;
```

很巧妙



Diffusion

```
local_N = N/size;
if(rank == size-1) local_N += N % size;
用 C.resize((local_N+2)*(N+2), 0.0)
```

6. Blocking collective P31
   - Reduce · AllReduce · Broadcast · Gather · AllGather
   - Scatter · Barrier · AllToAll · Scan · ExScan
   - Reduce_Scatter

---

7. Communication Cost: Bandwidth and Latency P3
8. More on blocking point to point
   <1> unknown size message MPI_Probe P5
   <2> Cycle communication (often in diffusion) P6
   ↳ 分奇偶
   MPI_Sendrecv P7
9. Message passing protocols P8 - P12
   ↳ eager rendezvous
10. Non-Blocking point to point communication P14
    <1> MPI_Isend MPI_Irecv P15 { blocking non-blocking 比较 P16 / send modes P16
    ↓会创建 MPI_Request，用下面方法管理
    <2> Request management
    P17 { MPI_Wait → MPI_Test
    P18 { MPI_Waitall / MPI_Waitany / MPI_Waitsome
    test multiple request P22
    ↑ Example P23-P26
    <3> Example: 1D Finite Difference P27-P31
11. Non-Blocking collective communication P33-P38 (examples + 1D diffusion)
12. Performance metrics P40
    ① 提到 Communication overhead 影响 ② 提到 Timing MPI program

OpenMP

```
#include <omp.h>
g++ -fopenmp -o myprog myprog.cpp
```

Syntax Format:

```
#pragma omp construct [clause [clause]]
```

→ Structured block Unstructured block P35 Op1

- the threads synchronize at a barrier at the end of the parallel region, then master thread continues
- const int id = omp_get_thread_num(); 跟在parallel后
- #pragma omp critical 下面的代码每次只能一个thread执行，但具体是哪个随机. → P29 Op1

→ num_thread(i) if(parallel: i>2)

→ omp working sharing constructs

Op1 { Loop construct P34 / Section Construct P36 / Single Construct P37

Both have an implicit barrier at the end (unless nowait)

Thread count P24
不同于MPI，其并行数是在编译时指定.

→ omp_get_thread_num(); omp_get_num_threads(); P31

→ Loop scheduling (load balance) P38

```
#pragma omp parallel for schedule(static, 2)
#pragma omp parallel for schedule(dynamic, 2)
```

→ #pragma omp for collapse(2) P43
↓ perfectly loop

→ #pragma omp for reduction(+:sum) P46
↳ sum 会有 private copy, no race condition

→ #pragma omp parallel for combined construct P47

→ ☆ Data environments P49

- in OMP, data outside a parallel region are shared by default
- global/static variables are shared
  → default P51
  → private P52 → create a private copy of each variable in the list (avoid race) not initialized
  ↳ initialized { copy 之前 firstprivate P53 / update after lastprivate P54
  ↓ 有 race

Recall Data environment P1-P9

→ Race condition P12 { critical sections P13 P19 / Atomics P14 P20 / locks P15 / barriers P16

↓P18 又开始有系统介绍了 Synchronization，把 critical, atomics 又讲了一遍。

Synchronization constructs ensure consistent access to memory address that is shared among a team of threads

{ master P21 / barrier P22 → implicit barrier P23

{ false sharing P25 / Library routines / Environment variables

---

13. Communication Topologies and Groups
    ① Communicators P5 ② Example: domain decomposition P6 communication/computation ratio
    ③ Create MPI cartesian Topology P13 → finding neighbors P15 → 有用函数 P16
14. MPI Vector Datatype
    ① contiguous vs strided data P18 ② Strided datatype P20 → example P21
    ③ multidimensional arrays P23
15. MPI Struct Datatype
    ① riemann Suns 举例 P27-P28 → 引出问题 P28-P34
    ② 引用 MPI_Pack P35-P36 → 不满足，总结 P37
    ③ general approach MPI_Type_create_struct P38-P40 → 例子 P41-P43