1. Theoretical Foundation of HPC for Computational Science and Engineering

Shizheng Wen shiwen@student.ethz.ch

1 Introduction

Computing:

- serial computing: serial implementation of programs for single CPUs
- parallel computing: program is executed using multiple machines
  - A problem is broken into discrete parts that can be solved concurrently.
  - Each part is further broken down to a series of instructions.
  - Instructions from each part execute simultaneously on different machines (here: CPUs).

Why parallel computing:

- physical/parctial constraints for even fast serial computers
  - transmissin speeds - the speed of a serial computer is directly dependent upon how fast data case move through hardware.
  - limits to miniaturization - processor technology is allowing an increasing number of transitors to be place on a chip.
- econmics limitations - increasingly expensive to make a single processor fater. using a larger number of moderately fast commodity processors to achive the same performance is less expensive.
- energy limt - limits imposed by cooling needs for chips and supercomputers.

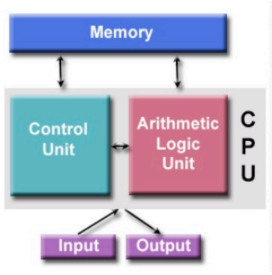
Firstly, people use the single-core processor. However, they found that power wall will happen when continuing to use single-core processor. After hitting the wall, people started to think multi-core processor. Multi-core processor can also help us conduct parallel programming. In this definition, a core means a computational unit.

Therefore, after that, all major processor vendors are producing multicore chips, giving rise to that every machine is already a parallel machine.

Parallel computing concepts and terminology:

- Terminology: units of measure
  - Flop: floating point operation, usually double precision unless noted.
  - Flop/s ("flops"): floating point opeartions per second.
  - Bytes: size of data (a double precision floating point number is 8 bytes).
- von Neumann Architecture
  - memory: RAM (radom access memory) stores both program instructions and data
    - program instructions are coded data which tells the computer to do something
    - data is simply information to be used by the program.
  - control units - fetch instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.

- arithmetic units - perform basic arithmetic operations
- input/output - is the interface to the human operator
-



- Cluster: A set of computers connected over a Local Area Network (LAN) that functions as a single computer with large multiprocessor.
- Multicores, multiprocessors:
  - Multiple processors per chip.
    - Processors are referred to as cores. 【所以是同一个概念】

    → Number of cores per chip is expected to double per year. 【moole's law】

Performance evalutions:

1. Computational cost (energy)
   - Total execution time on all processors:
     - *t(1)*: execution time (wall-clock) on a single proc.
     - $t_i(Nproc)$: execution time (wall-clock) on processor i out of *Nproc* processors.
     - Cost: Cp(Nproc)=Nproc*MAX$_i$(t_i(Nproc))
     - Cost-optimal algorithm: cost to solve the problem on Nproc processors is equal to cost to solve it on a single processor.

       Speedup factor:
2. Speedup factor:
   - How many times faster the problem is solved on *Nproc* processors than on a single processor:
   - Speedup (using j samples):
     -
     $$S(Nproc) = \frac{t(1)}{mean_j \max_i t_{ij}(Nproc)} \frac{N(Nproc)}{N(1)}$$
     - *N(1)* and *N(Nproc)* are the problem sizes.
     - The maximum speedup is *Nproc* (linear speedup).
     - Super linear speedup, i.e., *S(Nproc)* > *Nproc* is an artifact of using a sub-optimal sequential algorithm.
3. Scaling Reference

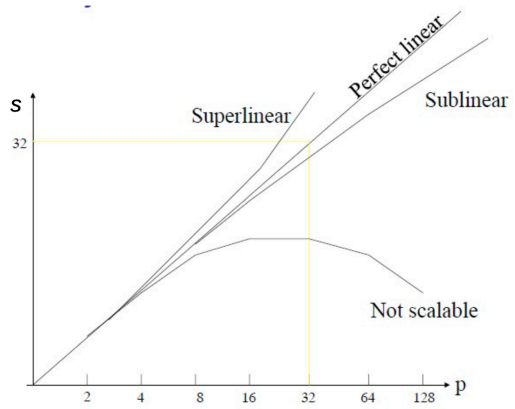
Strong scaling 和 weak scaling是高性能计算中我们做分析的一种手段，针对我们尝试解决的不同问题。 Youtube

- Strong scaling:
- run a problem faster (problem doesn't change)
  - Problem size does not change with the number of processors, so the amount of work per processor decreases with the increase of processors.
  - Weak scaling:
    - Run a larger problem (or more) with fixed time
    - Problem size increases proportionally with the number of processors, so the amount of work per processor remains constant with the increasing number of processors .
  - Strong scaling is harder to achieve because the communication overhead grows.

4. Parallel efficiency:
   - Fraction of the time the processors are being used for computation:

$$E(Nproc) = \frac{t(1)}{t_i(Nproc)Nproc} = \frac{S(Nproc)}{Nproc}$$

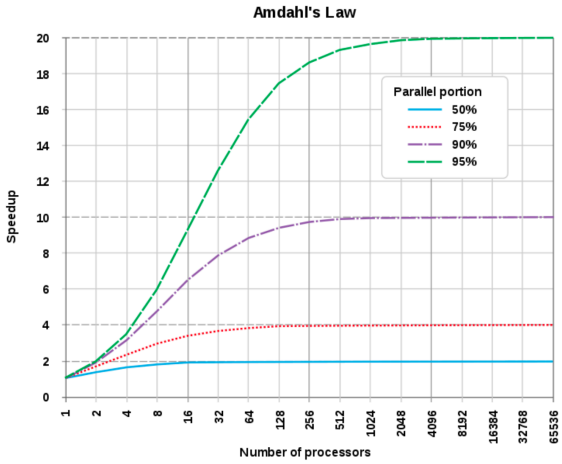
   - The efficiency is between 0 and 1 (or 0% and 100%) with an ideal program achieving 100% efficiency.
5. Scalability



6. Amdahl's law:
   - Upper bound for speedup of a given program (fixed-size problem, **strong scaling**): S_s means speed up factor defined in 2
   - *f*: serial factor, i.e., fraction of the program that cannot be parallelized (t_s = serial time):

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

   - Maximum speed with an infinite number of processors (N) is *1/f*. If e.g., 5% of the program is sequential, the speedup can never exceed 20!
   - Question: what value of *f* is required to obtain a speedup of 90 using 100 cores ?



7.
   - Compared with 2, Amdahl's also discussed the effect of portion of the parallel part in the code to the speed up.
   - Also remember that speed up doesn't mean that your code have high efficiency.
8. Gufstafson's law (serial part does not grow with problem size): don't understand!
   - Assume the cost(T) of parallel execution on N processors is $T = 1$.
   - Let s denote the serial part and p the parallel time, thus: $s + p = 1$.
   - The cost of the serial calculations is: $T' = s + N*p$
   - The scaling is then:

$$S=\frac{T'}{T}=\frac{s+Np}{s+p}=\frac{s+Np}{1}=s+Np=s+N(1-s)$$

Thus for large N:

$$S\to N(1-s) \quad \text{for} \quad N\to\infty$$

   - "Any sufficiently large problem scales well".

References

1 2

- cache (缓存):
  - 原始意义是指访问速度比一般随机存取存储器（RAM）快的一种高速存储器，通常它不像系统主存那样使用DRAM技术，而使用昂贵但较快速的SRAM技术。缓存的设置是所有现代计算机系统发挥高性能的重要因素之一。
  - 缓存是指可以进行高速数据交换的存储器，它先于内存与CPU交换数据，因此速率很快。
  - 缓存的工作原理是当CPU要读取一个数据时，首先从CPU缓存中查找，找到就立即读取并送给CPU处理；没有找到，就从速率相对较慢的内存中读取并送给CPU处理，同时把这个数据所在的数据块调入缓存中，可以使得以后对整块数据的读取都从缓存中进行，不必再调用内存。正是这样的读取机制使CPU读取缓存的命中率非常高（大多数CPU可达90%左右），也就是说CPU下一次要读取的数据90%都在CPU缓存中，只有大约10%需要从内存读取。这大大节省了CPU直接读取内存的时间，也使CPU读取数据时基本无需等待。总的来说，CPU读取数据的顺序是先缓存后内存。

- 缓存只是内存中少部分数据的复制品，所以CPU到缓存中寻找数据时，也会出现找不到的情况（因为这些数据没有从内存复制到缓存中去），这时CPU还是会到内存中去找数据，这样系统的速率就慢下来了，不过CPU会把这些数据复制到缓存中去，以便下一次不要再到内存中去取。随着时间的变化，被访问得最频繁的数据不是一成不变的，也就是说，刚才还不频繁的数据，此时已经需要被频繁的访问，刚才还是最频繁的数据，又不频繁了，所以说缓存中的数据要经常按照一定的算法来更换，这样才能保证缓存中的数据是被访问最频繁的。

- concurrency(并发性)；
- what is HPC? What is MPI?

---

2 High performance computing (HPC) principles, metrics, and models

Outline:

- principles of hpc
- the roofline model
- cache hierarchy

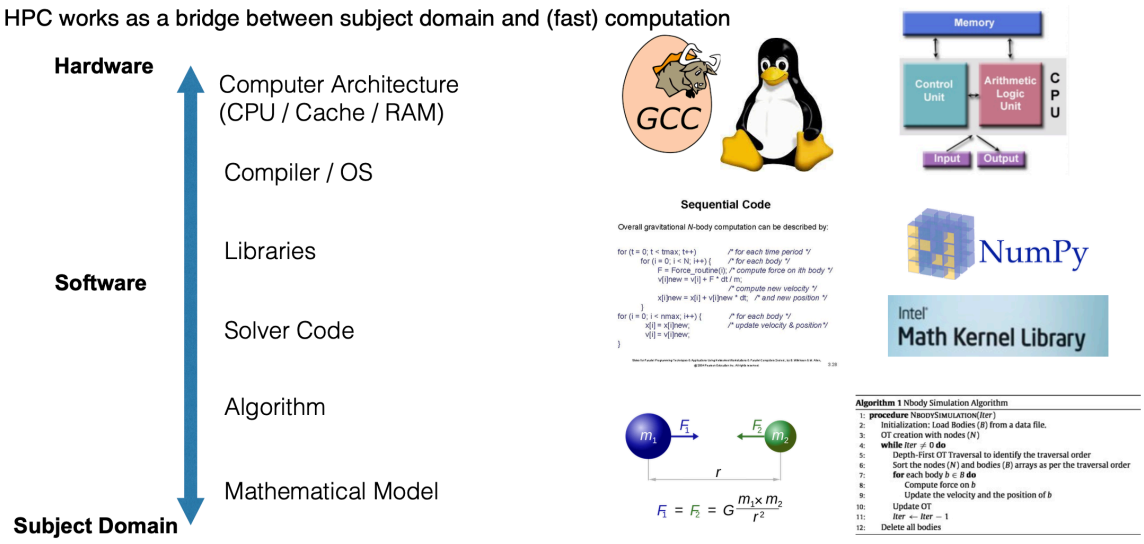
2.1 Principles of HPC

Definition:

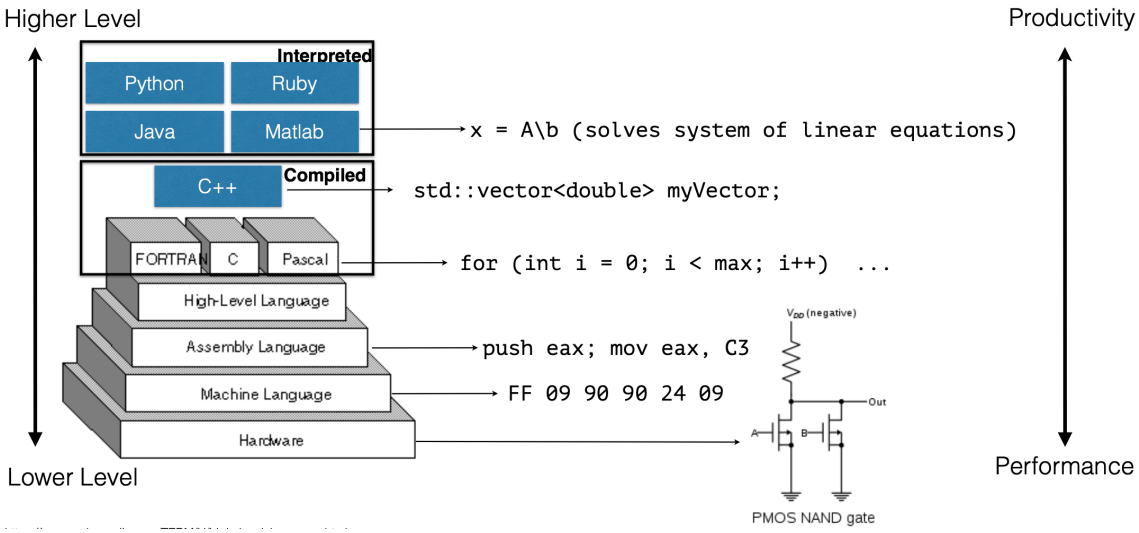
Wikipedia: HPC uses supercomputers and computer clusters to solve advanced computation problems.

HPC@UNIBAS: High Performance Computing or HPC refers to any computational activity requiring more than a single computer to execute a task, giving rise to parallel and distributed computing.

HPC is a multidisciplinary field



Computer Languages



HPC requires a quantitive approach

We need a precise ways to measure performance that allows us to:

- Compare the efficiency of different algorithms when solving a given problem
- Compare the efficiency of different implementations (code) of a given algorithm
- Determine whether an implementation uses the computational resources efficiently
- Compare the performance of the solver on different computer architectures

Architectural Metrics:

When an optimal algorithm is given, we can focus on processor metrics:

Performance = [MIPS, Million Instructions/Sec]

$$MIPS = \frac{InstructionCount}{ExecutionTime * 10^6}$$

Problem 1: Not all instructions are created equal! (some take longer than others)

Problem 2: Different CPU architectures may require more or less instructions to solve the same problem (faster or slower!)

Therefore, we redefined:

Performance = [MFLOP/s, Million Single-Precision Floating-Point Operations per second]

- Standard metric for performance in scientific computation
- Only focuses on *useful* instructions (add, subtract, multiply, divide)
- We know how many of them the algorithm requires

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

→ This algorithm requires $2n^3$ FLOP → An increase in FLOP/s will result in a speedup for this algorithm

- In this part, we defined the metrics for expressing the performance of parallel computing.

Arithmetic (Operational) Intensity

An algorithm's Arithmetic Intensity (**I**) is the ratio between its total work (**W**) to its memory operations (**Q**) [有个叫运算强度（Arithmetic intensity）的概念，即flops per byte，表示一个字节的数据上发生的运算量，只要这个运算量足够大，意味着传输一个字节可以执行足够多的计算量。需要注意的是，计算强度这个概念的定义是针对算法的，是对某一个算法的度量]

An algorithm's Arithmetic Intensity (**I**) is the ratio between its total work (**W**) to its memory operations (**Q**)

$$I = \frac{W}{Q} = \frac{\#FLOPs}{\#Bytes}$$

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

This algorithm requires $O(n^3)$ FLOP

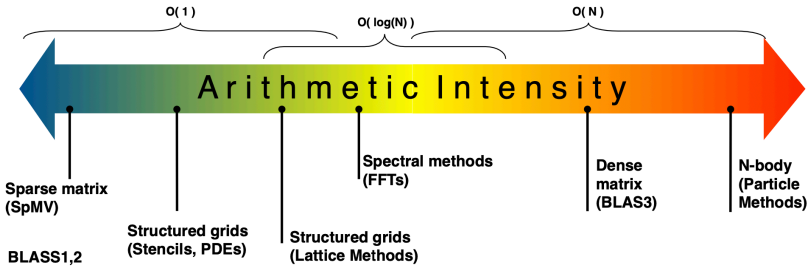
This algorithm requires $O(n^2)$ loads and stores (assuming all matrices fit in cache)

I = O(n) FLOPs / byte

The dense matrix multiplication algorithm requires more FLOPs than bytes as the problem size increases

7 Dwarfs

A dwarf is a pattern of (scientific) computation and communication. 7 dwarves first described in 2004, and expanded ever since (graphs / combinatorial logic / dynamic programming...). These patterns are well defined targets from algorithmic, software, and architecture standpoints.



2.2 Roofline Model

The roofline model allows us to:

- estimate how efficient is our code, based on a given computational platform (esp. multicore CPUs and GPUs).
- estimate how much more performance to obtain if we optimize it.
- compare different systems and evaluate which one is best for our code.

The model uses a *log-log* plot that relates:

- Arithmetic Intensity (I = [FLOP/bytes])
- Performance (FLOP/s)

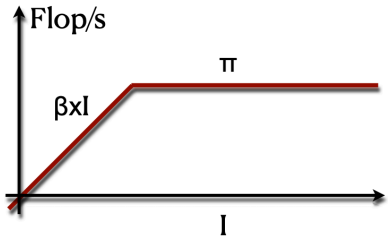
It is based on two main (real) assumptions:

- The system's CPU peak performance (π = [FLOP/s]) is limited

- The system's RAM memory bandwidth (β = [bytes/sec]) is limited.

Describes two ways in which an algorithm's performance may be limited by the system.

- Memory-Bound, given by the βxI line
- Compute-Bound, given by the π line

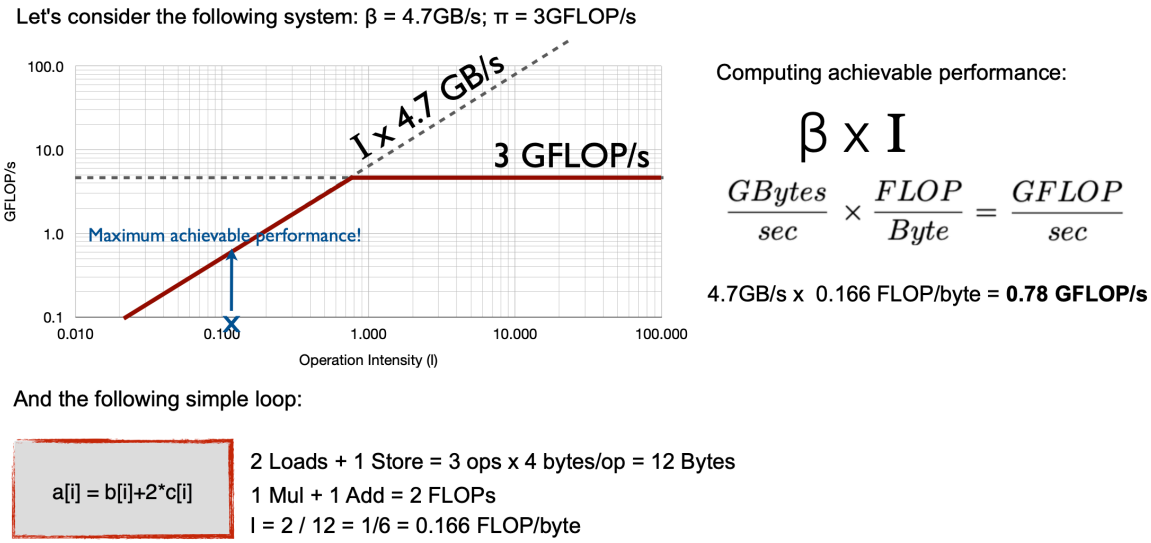


下面这个参考内容讲的很好：https://zhuanlan.zhihu.com/p/34204282
核心就是说：当我们在计算一个模型在一个平台上的理论性能时，我们不仅要关注平台的性能指标。还要关注模型（也就是算法）的性能指标，这两者之间的关系将决定我们的模型能否达到平台的理论性能上限。而Roof-line Model给出的是算法在平台的理论上线。

其实 Roof-line Model 说的是很简单的一件事：*模型在一个计算平台的限制下，到底能达到多快的浮点计算速度*。更具体的来说，Roof-line Model 解决的，是“*计算量为A且访存量为B的模型在算力为C且带宽为D的计算平台所能达到的理论性能上限E是多少*”这个问题。

那我们熟悉的情况来举例，对于一个机器学习模型，当我们使其在计算平台上跑时，我们可以估计该模型的性能能否全部发挥出来

Example: Memory-Bound

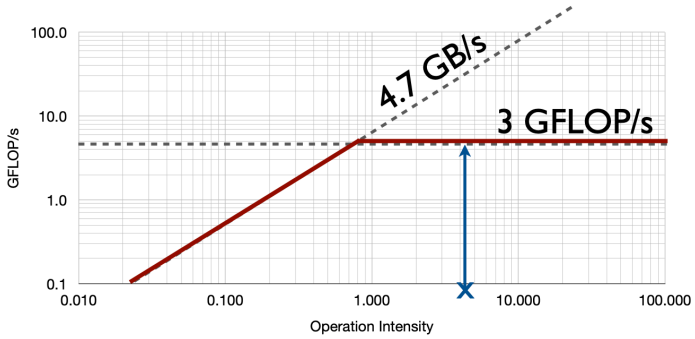


Example: Compute-Bound

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

I = O(n) FLOPs / byte

For a sufficiently large *n*...



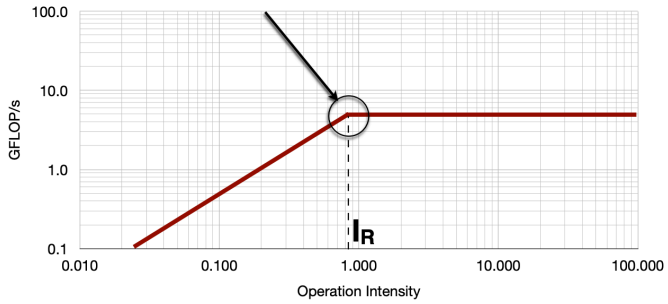
GFLOP/s =π

Determining Memory vs. Compute Bound: The Ridge Point

Next, we give some parameters to characterize the roofline model:
ridge point: the point at which the memory and computation bounds coincide. *characterizes the overall machine performance.*

- The more the ridge point is "to the left", the relatively easier it is to get peak performance

Ridge Point: the point at which the memory and computation bounds coincide.



I_R indicates the arithmetic intensity where the ridge point is located.

$$I_R = \frac{\pi}{\beta}$$

- If I < I_R then the algorithm is memory bound
- If I > I_R then the algorithm is compute bound

- Ridge point **characterizes the overall machine performance**

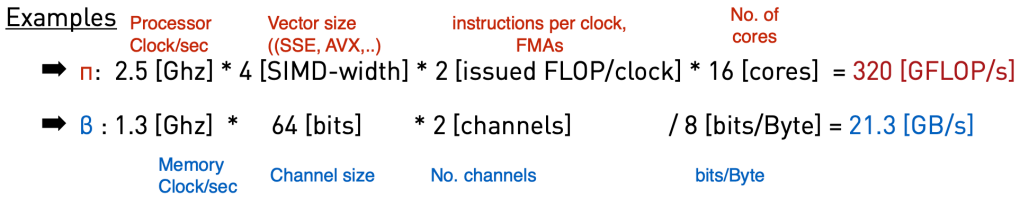
➡ The more the ridge point is "to the left", the relatively easier it is to get peak performance

➡ The more the ridge point is "to the right", the harder it is (optimizing for memory is harder)

Nominal performace:

How to estimate nominal π and β?

- Sometimes indicated in the manufacturer's page.
- From the hardware specification of the platform
- On Linux
  - type `cat /proc/cpuinfo`
  - And `sudo dmidecode --type memory`



似乎和处理器的时钟频率有关.

Then, we found that the real measured performance have a discrepancy from nominal performance.

Expected discrepancy from nominal quantities:

- ✓FLOP/s: 90-100% of nominal performance
- x GByte/s : 50-70% of nominal performance

This may reveal

- Programming skills in extracting system performance
- best case scenario for more complex kernels

View the Slide, which offer many study case for explaining it.

---

3 Cache/Thread-Level Parallelism

Outline:

- Cache Hierarchy and Logic
- Cache Optimization
- Concurrency & Parallelism
- Threading Models and Libraries

3.1 Cache Hierachy and Logic

- 缓存初步了解
  - motivation：是为了解决冯诺依曼体系中存储单元和计算单元之间的缓慢访问【即：CPU的阻塞问题】
  - 其利用的基本原理是物理内存之间的临近存储
  - **局部性：**程序访问完一个存储区域往往会访问接下来的区域，这个原理称为局部性。在访问完一个内存区域（指令或者数据），程序会在不久的将来（时间局部性）访问邻近的区域（空间局部性）。
  - ！CPU Cache是由系统硬件来控制的，而编程人员并不能直接决定什么数据和什么指令也应该在Cache中。但是，了解空间局部性和时间局部性原理可以让我们对Cache有些许间接的控制。例如，C语言以“行主序”来存储二维数组。
- threads: 线程 process：进程
  - 进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

  - 线程（英语：thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

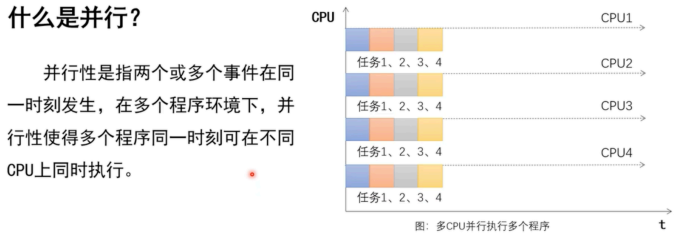
- concurrency（并发性）；parallelism
- registers：寄存器
- memeory latency 内存延迟 本质原因是处理器和内存的频率不同
- CPU频率：就是CPU的时钟频率，简单说是CPU运算时的工作频率(1秒内发生的同步脉冲数)的简称。单位是Hz。它决定计算机的运行速度，随着计算机的发展，主频由过去MHZ发展到了现在的GHZ(1G=1024M)。通常来讲，在同系列微处理器，主频越高就代表计算机的速度也越快，但对与不同类型的处理器，它就只能作为一个参数来作参考。
- 内存：RAM; ROM; Cache三者之间的关系
  - 通常所说的内存是指的随机存储器。只读存储器通常只能用来读取，通常是系统，如bios这些装在上面。cache位于内存和处理器之间。
- 1 byte = 8 bits [1]
  1KiB= 1,024 bytes [1]
  1MiB= 1,048,576 bytes [1]
  1GiB= 1,073,741,824 bytes [1]
  1TiB= 1,099,511,627,776 bytes [1]
- register（寄存器），cache（缓存）与内存（ram）之间的联系【参考】：按与CPU远近来分，离得最近的是寄存器，然后缓存，最后内存。寄存器是最贴近CPU的，而且CPU只与寄存器中进行存取。
  - 寄存器是CPU内部用来存放数据的一些小型存储区域，用来暂时存放参与运算的数据和运算结果。其实寄存器就是一种常用的时序逻辑电路，但这种时序逻辑电路只包含存储电路。寄存器的存储电路是由锁存器或触发器构成的，因为一个锁存器或触发器能存储1位二进制数，所以由N个锁存器或触发器可以构成N位寄存器。寄存器是中央处理器内的组成部分。寄存器是有限存储容量的高速存储部件，它们可用来暂存指令、数据和位址。
- HDD (hard disk drive): 机械硬盘
- 缓存在CPU上，是CPU的组成成分之一
- difference between concurrency and parallelism: https://blog.csdn.net/weixin_47513022/article/details/115656874

3.2 Lecture

1. Cache Hierarchy: a brief history
2. How does a Cache work
   42:15

4 Thread-Level Parallelism

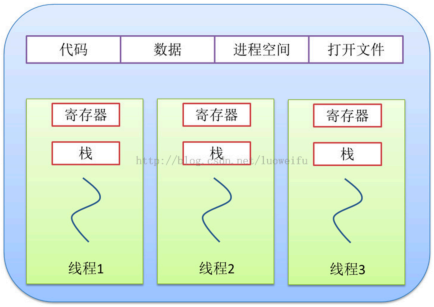
- concurrency & parallelism
- threading models and libraries
- Intro to OpenMP
- threads: 线程　process：进程
  - 进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。
  - 线程（英语：thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。
- concurrency（并发性）；parallelism



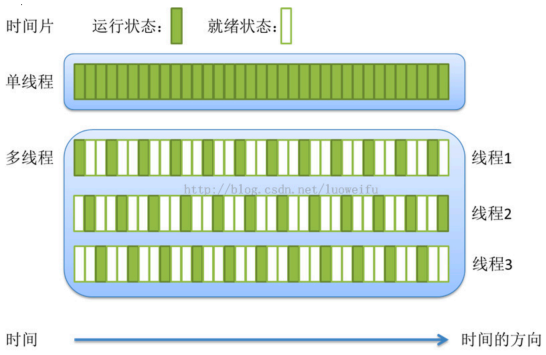
Introduction:

- 首先介绍了concurrency和parallelism的概念，接着进一步介绍实现parallelism的几种方式。其中包含非常重要的Thread-level parallelism。作者介绍了几种threading models和相应的库。最后着重介绍了OpenMP。这里主要是说明如何通过OpemMP来实现并行。
- 我觉得我们的核心是要从硬件和软件两个角度去认识并行
- API是什么
  - MPI, Pthreads, OpenMP and CUDA, four our the most widely used APIs for parallel programming.
- 进程和线程 参考内容
  - 进程是程序执行的最小单元，而线程是操作系统分配资源的最小单元
  - 一个进程由多个线程组成。线程是一个进程中代码的不同执行路线。
  - 当我们打开一个应用程序的时候，系统便会为这个应用程序划分相应的进程：**进程**是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位
  - 进程一般由程序、数据集合和进程控制块三部分组成。
    - 程序用于描述进程要完成的功能，是控制进程执行的指令集；
    - 数据集合是程序在执行时所需要的数据和工作区；
    - 程序控制块(Program Control Block，简称PCB)，包含进程的描述信息和控制信息，是进程存在的唯一标志。

- **线程**是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本单位（正因为其实处理器调度和分配的基本单元，因此我们才会想到合理分配线程，从而达到高效并行计算）。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间。
- 线程组成：
  - 线程ID
  - 当前指令指针(PC)
  - 寄存器和堆栈。
  - **而进程由内存空间(代码、数据、进程空间、打开的文件)和一个或多个线程组成。**



- 进程和线程之间的关系
-



2. HPC Libraries

1 OpenMP

1.1 Basics

Concurrency and Parallelism

Concurrency: The existence of two or more stream of instructions, whose execution order cannot be determined a priori.

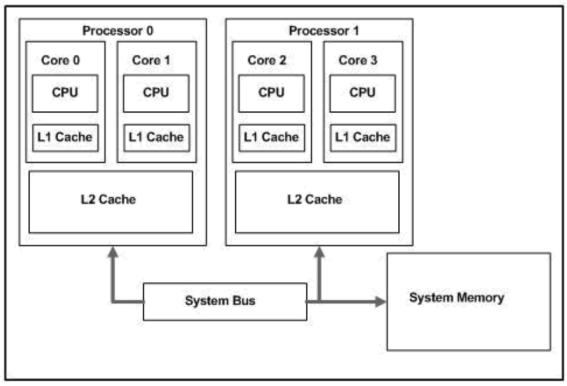
Parallelism: The existence of two or more stream of instructions executing simultaneously.

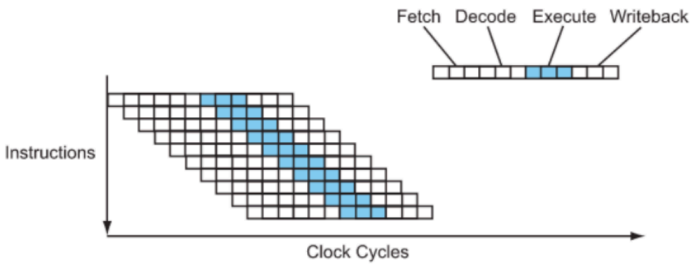
There can be concurrency without parallelism, but there cannot be parallelism without concurrency.

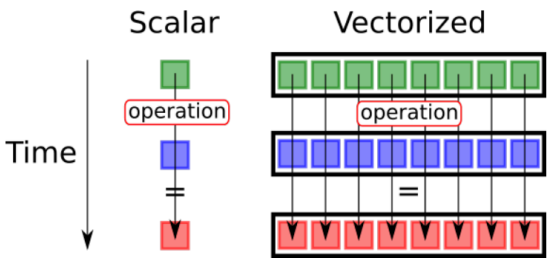
Support for Parallelism in Hardware

- Multiple physical cores (Thread-Level Parallelism)

- Pipelining (Instruction-Level Parallelism)
- Vectorization (Data-Level Parallelism)







Processes and Threads

Processes:

- Are OS structures (memory+code+threads)
- Operate on their own private virtual address space (64-bit addresses)
- Managed by the OS scheduler
- Contain one or more threads

Threads (Kernel-Level):

- Represents a CPU execution state (register values, program counter)
- Executes a stream of instructions (code) within a running process
- All threads associated to the same process share the same virtual address space
- Programmer can control creation/deletion of additional threads

1.2 Threading Libraries

POSIX Threads (Pthreads)

- Standardized C language threads programming interface http://pubs.opengroup.org/onlinepubs/9699919799/
- Header file: `#include <pthread.h>`
- Compilation: `gcc -pthread -o myprog myprog.c`

- Allows access to low-level features on shared memory platforms
- More lines of code compared to higher-level API's
- Not straightforward for incremental parallelism

C++11 Threads

- POSIX threads API is quite rigid and requires lots of additional code
- Another possibility to work with threads is available in C++ (2011 standard)
- Integrate well with C++ and its features
- Relatively easy to get started with
- Header file: `#include <thread>`
- Compilation: `g++ -std=c++11 -pthread -o my-dog my-dog.cpp`

Comparision between two Threads-oriented libraries:

C++11 Threads:

```
1 #include <thread>
2 #define NTHREADS 8
3
4 void do_work(const size_t tid)
5 {
6   // more local data
7   // do work
8 }
9
10 int main(int argc, char *argv[])
11 {
12   std::thread threads[NTHREADS];
13
14   // spawn threads
15   for (size_t t = 0; t < NTHREADS; ++t)
16     threads[i] = std::thread(do_work, t);
17   // join threads
18   for (size_t t = 0; t < NTHREADS; ++t)
19     threads[i].join();
20
21   return 0;
22 }
```

No pointers or C-style casts.

Code is easier to read.

A thread is an object (OOP)

POSIX Threads:

```
1 #include <pthread.h>
2 #define NTHREADS 8
3
4 void *do_work(void *t)
5 {
6   /* thread ID */
7   long tid = (long)t;
8   /* more local data */
9   /* do work */
10  pthread_exit((void *)t);
11 }
12
13 int main(int argc, char *argv[])
14 {
15   pthread_t threads[NTHREADS];
16   void *status;
17   long t;
18   /* spawn threads */
19   for (t = 0; t < NTHREADS; ++t)
20     pthread_create(&threads[t], NULL, do_work, (void *)t);
21   /* join threads */
22   for (t = 0; t < NTHREADS; ++t)
23     pthread_join(threads[t], &status);
24   pthread_exit(NULL);
25 }
```

Both POSIX threads and C++11 threads still impose quite large boilerplate code.

OpenMP: Open Multi-Processing

An **A**pplication **P**rogramming **I**nterface (API) to explicitly define mutli-threaded parallelism on shared memory architectures.

- **Comprised of 3 components:**
  - Pre-compiler directives (#pragma) (not present in POSIX or C++11)
  - Runtime library calls [Optional]
  - Environment variables (not present in POSIX or C++11) [Optional]
- **Header file:** `#include <omp.h>` **[Optional]**
- **Compile with:**
  - `g++ -fopenmp -o my-dog my-dog.cpp` (GCC)
  - `clange++ -fopenmp -o my-dog my-dog.cpp` (LLVM)
  - `icpc -qopenmp -o myprog myprog.cpp` (Intel)

Reason why we use OpenMP

Pros:

- Easy to use and allows to parallelize existing serial programs with minimum effort
- Parallelization can be done incrementally
- Widely used and supported
- Portability:
  - C/C++ and Fortran
  - Supports Unix/Linux platforms as well as Windows

Cons:

- Does not automatically check for race conditions, deadlocks
- Is not designed for distributed memory systems
- Nor is it designed for parallel I/O (user is responsible for synchronization)

Comparision between two Threads-oriented libraries:

Open MP:

```
1 #include <omp.h>
2 #define NTHREADS 8
3
4 void do_work(const size_t tid)
5 {
6    // more local data
7    // do work
8 }
9
10 int main(int argc, char *argv[])
11 {
12 #pragma omp parallel num_threads(NTHREADS) // spawn threads
13    {
14       // need omp.h header for this
15       const size_t t = omp_get_thread_num();
16       do_work(t);
17    } // join threads (implicit barrier)
18
19    return 0;
20 }
```

Even less boilerplate code

Distinction between sequential code and parallel regions is very clear

Implicit barrier at end of structured block (synchronization point)

C++11 Threads:

```
1 #include <thread>
2 #define NTHREADS 8
3
4 void do_work(const size_t tid)
5 {
6    // more local data
7    // do work
8 }
9
10 int main(int argc, char *argv[])
11 {
12    std::thread threads[NTHREADS];
13
14    // spawn threads
15    for (size_t t = 0; t < NTHREADS; ++t)
16       threads[i] = std::thread(do_work, t);
17    // join threads
18    for (size_t t = 0; t < NTHREADS; ++t)
19       threads[i].join();
20
21    return 0;
22 }
```

1.3 Syntax

Usage of OpenMP

OpenMP Syntax Format:

- Precompiler directives:
  - c/c++: `#pragma omp construct [clause [clause] ...]`
- since we use directives, no changes need to be made for a complier that doesn't support OpenMP!

Thread Count:

The number of threads in a parallel region is determined by the following factors, in order of precedence:

1. Evaluation of an `if` clause within the omp pragma

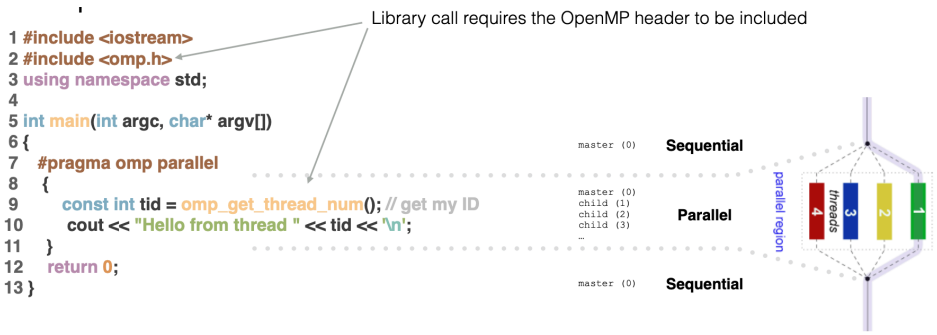
2. Setting the `num_threads` clause eg: `#pragma omp parallel num_threads(2)`
3. Use of the `omp_set_num_threads()` library function
4. Setting the `OMP_NUM_THREADS` environment variable
5. Implementation default — usually the number of CPUs available

Core Concepts: Fork/Join Parallel Regions

- Program executes in serial until it encounters a parallel directive
- The thread that encounters the parallel region creates the children (team of threads)
- The encountering thread becomes the master thread inside the region (thread_id = 0)
- The threads synchronize at a barrier at the end of the parallel region. The master thread then continues

```
#pragma omp parallel [clause llist] new-line
•       Structured block
```

Example:



- Note: If the structure block in a single line, then you may omit the curly braces.

Output:

```
1 Hello from thread Hello from thread Hello from thread Hello from thread 3 2 201
2 201
3
4
```

1.3.1 Work Sharing Constructs

Definition of Work Sharing Constructs

OpenMP gives us access to the following work sharing constructs:

- *Loop construct*: Specifies that the iterations of one or more associated loops will be executed in parallel by the threads in the team. Used very often!
- *Sections construct*: A non-iterative work sharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.

- *Single construct*: Specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread).

Remember: All work sharing constructs have an implicit barrier at the end (unless the *nowait* clause is used).

Example: Parallelizing a For-loop Manually

```
1 #include <omp.h>
2
3 int main(int argc, char* argv[])
4 {
5   const int N = 1000; // number of points in array
6   double A[N];      // array of data we want to work on in parallel
7
8   #pragma omp parallel
9   {
10    const int tid = omp_get_thread_num();
11    const int nthreads =  omp_get_num_threads();
12    int npoints = N / nthreads; // I am working on this many points
13    const int mystart = tid * npoints;  // my start in the array
14    if (tid == nthreads-1) // ensure to take care of all elements in A
15      npoints = N - mystart;
16
17    // iterate over array A in parallel
18    for (int i = 0; i < npoints; ++i)
19      A[mystart+i] = // do something here
20  }
21  return 0;
22 }
```

1. Get my ID and the number of threads in the team
2. Determine the number of points I need to work on (you will see this type of code often in MPI)
3. Iterate through my portion of array A

- Manually allocating the the number of points needed to operate in every thread is very cumbersome, but is very often in MPI. If we use `#pragma omp parallel`, OpenMP does the partitioning of work for us! (Less code, cleaner and more readable!)

Next, we will discuss these sharing construct one by one!

1.3.1.1 Loop Construct (Parallel for-loops)

Definition

The loop construct is the most important of the OpenMP work sharing constructs. Let's look a bit closer:

```
#pragma omp for [clause[[,]clause]...] new-line
for-loops
```

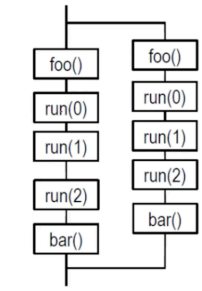
Clause can be:

- `private(list)`; `firstprivate(list)`; `lastprivate(list)`
- `schedule(kind [, chunk_size])`
- `collapse(n)`
- `reduction(reduction-identifier:list)`
- `nowait`
- Note: There are a few more clauses that we do not discuss in this lecture notes. You can find a full list in the OpenMP specification. https://www.openmp.org/specifications/

regular use

- `#pragma omp parallel`

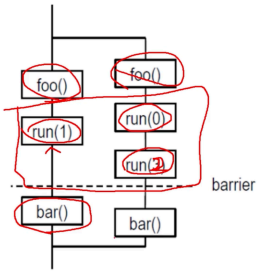
- 后面的structure block被并行执行，设置了多少个thread，等价上来说就是执行多少次
- 相当于并行的thread做了同一个事情，代码执行只有速度上的差异，没有执行量上的差异。
-




```
#pragma omp parallel \
 num_threads(2)
{
    foo();
    for (int i = 0; i < 3; i++)
        run(i);
    bar();
}
```

- `#pragma omp parallel for`
  - 后面的for循环被拆分成多少个子快，每个子块通过一个thread去并行。
  - 但是这里需要注意的是，如果各个子块thread之间有数据依赖的话，则很有可能会出错
  -

```
#pragma omp parallel \
 num_threads(2)
{
    foo();
    #pragma omp for
    for (int i = 0; i < 3; i++)
        run(i);
    bar();
}
```



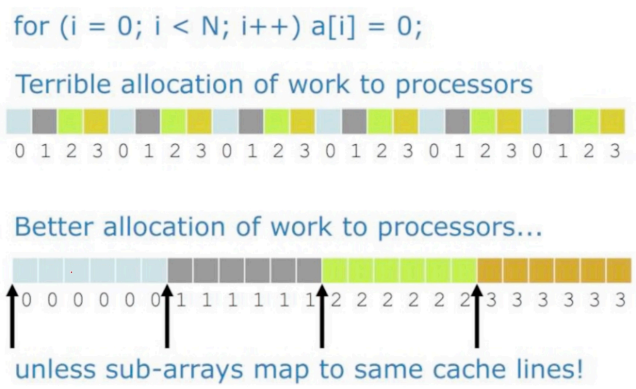
  - 并行潜力的实现主要体现在对循环的处理上.

Loop Scheduling

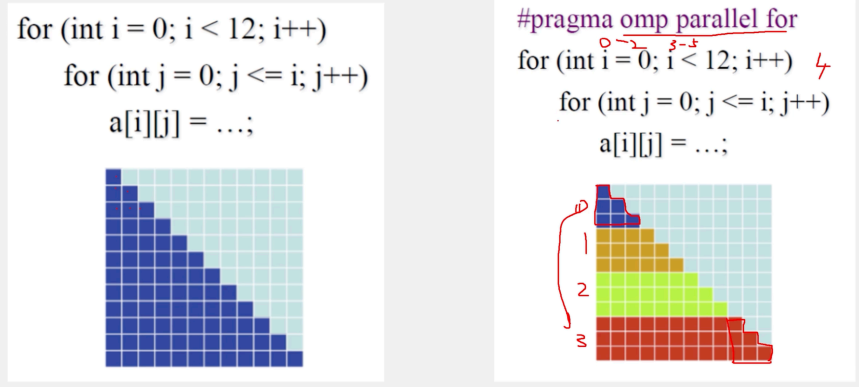
Motivation: There are different ways to distribute the work among threads in a team. We care about this to have more control over parallel granularity and load-balancing.

Usually, we discussed about loop scheduling in `for` loop.

```
#pragma omp for schedule (kind [, chunk_size]) new-line
for-loops
```



- 内存的访问是连续的，在读入cache中，不进行大量换入换出等额外开销。

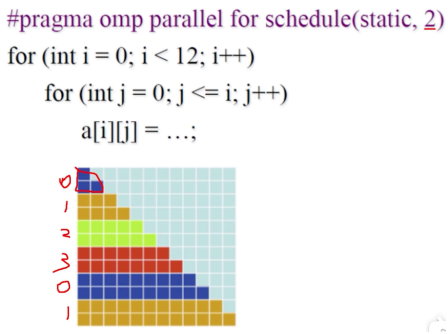
- 

Solutions: 循环调度(Loop Scheduling):

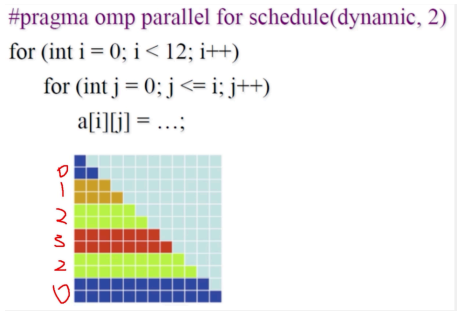
循环调度的种类：

- 静态调度：在循环执行之前，就已经将循环任务分配好 （就像我们的`#pragma omp parallel for`）
- 动态调度：在循环执行的过程中，边执行变分配

OpenMP提供了一个选项`schedule`，它能够将循环的分配给每个线程。当采用不同的参数的时候，我们会使用不同的调度方式。

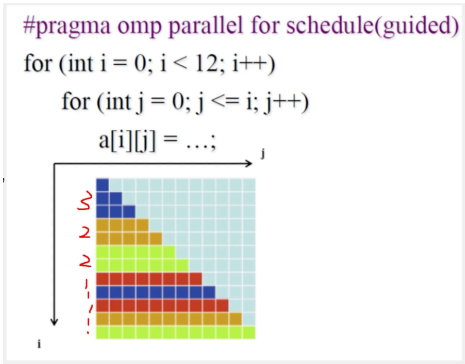
- `schedule(static[,chunk])`
  - 当采用static的参数时，chunk代表了每一块分块的大小
  - 他会采取轮转制度，按照线程的数量，获得分配的一块大小的内容。在循环开始前就已经确定。
  - 低开销，但是可能会造成分配不均
  - 

- `schedule(dynamic[,chunk])`
  - 当采用dynamic的参数时
  - chunk代表了每一块分块的大小
  - 每个线程执行完毕后，会自动获取下一块
  - 高开销，但是能减少分配不均衡的情况

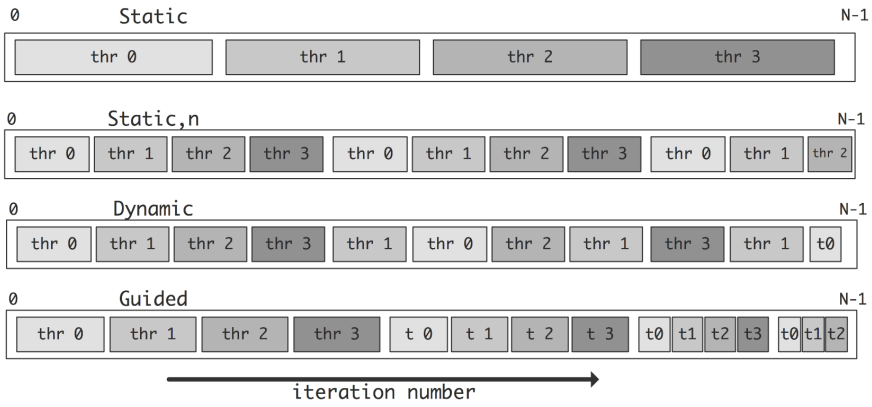


–

- `schedule(guided[,chunk])`
  - 当采用guide的参数时，会按照一定的规则分配块
  - 这是一种动态的分配
  - 每一块的分配的数量时在不断收缩的，但是最小不会小于chunk（默认是1）
  - 最初的块会被定义成：
    - 循环数量/线程数
  - 其余块的大小会被定义成：
    - 剩余循环数量/线程数
  - 计算效率比dynamic高一点点。且也有缺陷，例如这个循环从下往上执行，就会导致第一个thread承担非常大的负担。
  -



Loop scheduling kind (in a picture):



Nested Loops

Motivation: How does OpenMP parallelize nested loops:

```
#pragma omp parallel
{
  #pragma omp for
  for(int i=0;i<N;++i){
    for(int j=0;j<N;++j){
      A[i][j]= //do something here
    }
  }
}
```

- The `#pragma omp for` applies to the for-loop that immediately follows it
- Because the innermost loop and the outermost loop are independent, can we use more efficient way for parallel?

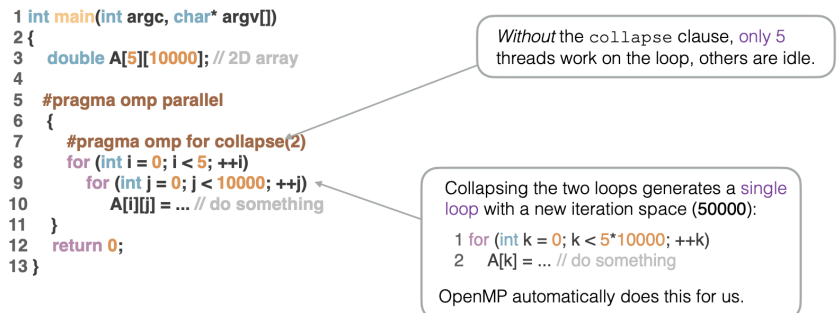
Solution: Loop Collapsing

Multiple loops associated to a loop construct can be collapsed (combined or fused) into a single loop with a larger iteration space if they are perfectly nested.

```
#pragma omp for collapse(n) new-line
for-loops
```

- `n` must be a constant positive integer expression.
- But be aware: If execution of any associated loop changes any of the values used to compute any of the iteration counts, then the behavior is unspecified!
- Benefits of loop collapsing: (1) Can achieve better utilization of available resources; (2) Increased performance due to better load-balancing.

Example 1:



Example 2:

```
1 int main(int argc, char* argv[])
2 {
3   const int N = 1000;
4   double x[N], y[N]; // Vectors
5   double A[N][N]; // Matrix
6   // initialize data...
7
8   // parallel matrix vector product
9   #pragma omp parallel
10   {
11    #pragma omp for collapse(2)
12     for (int i = 0; i < N; ++i)
13     {
14       y[i] = 0.0;
15       for (int j = 0; j < N; ++j)
16         y[i] += A[i][j] * x[j];
17     }
18   }
19   return 0;
20 }
```

Can we collapse these loops?

No. Only perfectly nested loops can be collapsed. *Perfectly nested*: The loop body of the outer loop can consist only of the inner loop.

Reduction

Definition: A reduction is a common operation where a specific operation is performed on all elements on some set of data (distributed on different threads). This can be done efficiently with the reduction clause:

```
#pragma omp for reduction(reduction-identifier: list) new-line
   for-loops
```

The reduction identifier can be one of the following built-in operations:

+, -, *, &, |, ^, &&, ||, min, max

User-defined operators can also be implemented.

Comma separated list of reduction variables

Example: Sum the elements in an array in parallel

```
1 int main(int argc, char* argv[])
2 {
3   double a[1000];
4   // initialize data...
5
6   double sum = 0.0; // sum the elements in here
7   #pragma omp parallel
8   {
9     #pragma omp for reduction(+: sum)
10     for (int i = 0; i < 1000; ++i)
11       sum += a[i];
12   }
13   return 0;
14 }
```

What you learned so far, can you explain the visibility and value of the variable `sum` in lines `6, 9, 11` and `12`?

Line 6: The variable is shared, everyone could see it (only the `master` thread can see it at this point). Its value is initialized to zero.

Line 9: OpenMP creates a private copy for each thread and initializes the variable to 0 (in the case of the '+' operator).

Line 11: Each thread sums up its array elements into its own (thread private) variable `sum`. Its value depends on the data the thread is working on.

Line 12: OpenMP sums up all thread local sum variables and stores it in the shared variable `sum`. Its value is the result of the reduction.

Detailed Introduction of Reduction:(归并)

reduction也是一种常见的选项，它为我们的parallel，for和sections提供一个归并的功能。如果for loop仅仅只设计到数据的访问和存储在一个vector里面，则往往不涉及到reduction，而如果存储在一个double里面，则通常要考虑归并。【在Eigen中，我们也常常遇到数据reduction的情况，在这种情况our motivation往往是将一个Vector/Matrix归并成一个single scalar】

使用方法如下：

```
#pragma omp ... Reduction(op:list)
```

他会提供一个私有的变量拷贝并且初始化该私有变量。

私有变量的初始化的值取决于选择的归并的操作符。

这些变量的拷贝会在本地线程进行更新。

在最后的出口中，所有的变量拷贝将会通过操作符所定义的规则进行合并的计算，计算成一个共享变量【reduction的核心部分】。

- 有点像`lastprivate`选项，但`lastprivate`只会最后用最后一个线程的私有变量对公有变量进行赋值。而reduction则是按照运算符讲所有线程中操作后再对公有变量进行运算和赋值（初始变量原有的值也会在操作中，这点不像`lastprivate`会选择直接覆盖。
-

Reduction(归并):

Rection提供的操作符几乎都是符合结合律的二元操作符

$$a_1 \oplus a_2 \oplus a_3 \oplus \ldots \oplus a_n$$

本地变量的初始值如下所示：

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

1.3.1.2 Section Construct

`sections` can be used to assign individual structured blocks to different threads in a team:

```
int main(int argc, char* argv[])
{
#pragma omp parallel
  {
   #pragma omp sections // nowait
    {
      #pragma omp section
       x_calculation(); // independent computation
      #pragma omp section
       y_calculation(); // independent computation
      #pragma omp section
       z_calculation(); // independent computation
    } // other threads in the team wait here.
      // Use '#pragma omp sections nowait' to let them continue without
      // synchronization
  }
  return 0;
}
```

Note: You may also use the `reduction` clause with `sections`

sections 和section:

我们希望不同的线程执行不一样的代码

- `section`
  - sections在封闭代码的指定部分中，由线程组进行分配任务。当然我们也可以用`tid==1, tid==0`来确定执行
  - 每个独立的section都需要在sections里面。
    - 每个section都是被一个线程执行的。
    - 不同的section可能执行不同的任务

- 如果一个线程够快，该线程可能执行多个section 【因为使用线程组分配的】

1.3.1.3 Single Construct

`single` can be used to assign a structured block to one single thread in a team (can be considered as a synchronization construct as well):

```
int main(int argc, char* argv[])
{
#pragma omp parallel
  {
    do_many_things();
    #pragma omp single // nowait
    {
      exchange_boundaries();
    } // other threads in the team wait here

    // IFF this function does not depend on memory locations that need to
    // be visible after exchange_boundaries() has been executed, then
    // adding the 'nowait' clause to the single construct is more
    // efficient!
    do_many_other_things();
  }
  return 0;
}
```

- `single` ：该选项是在并行块里使用的
  - 它告诉编译器接下来紧跟的下段代码将会由只一个线程执行。
  - 它可能在处理多段线程不安全代码时非常有用
  - 在不使用no wait选项时，在线程组中不执行single的线程们将会等待single的结束
- `master` ：该选项是在并行块中使用的
  - 编号为0的线程master进行执行
  - 它告诉编译器接下来的一段代码只有有主线程执行
  - 它不会出现等待现象 所谓是否会出现等待造成的影响就是共有数据是否会出现混乱访问。
  -

Example: *master* **and** *single nowait*

```
tid = omp_get_thread_num();
if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

=

```
#pragma omp master
{
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

≈

```
#pragma omp single nowait
{
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

1.3.1.4 Combined Construct

Motivation: There is only a single *for*-loop in the parallel region, do I need all this code?

Solution: No, combined constructs are shortcuts for specifying one construct immediately nested inside another construct. This applies to the `loop` and `sections` constructs we have seen so far.

Example:

```
1 #include <omp.h>
2
3 int main(int argc, char* argv[])
4 {
5   const int N = 1000;
6   double A[N];
7
8   #pragma omp parallel
9   {
10    #pragma omp for nowait
11      for (int i = 0; i < N; ++i)
12        A[i] = // do something here
13  }
14  return 0;
15 }
```

Identical

```
1 #include <omp.h>
2
3 int main(int argc, char* argv[])
4 {
5   const int N = 1000;
6   double A[N];
7
8     #pragma omp parallel for
9     for (int i = 0; i < N; ++i)
10      A[i] = // do something here
11  return 0;
12 }
```

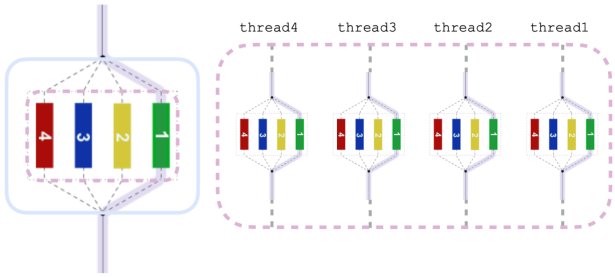
Note: *Implies* `nowait`, you can not use it as a clause here

Common beginners mistake is as follows:

```
1 #include <omp.h>
2
3 int main(int argc, char* argv[])
4 {
5   const int N = 1000;
6   double A[N];
7
8   #pragma omp parallel
9   {
10    #pragma omp parallel for
11    for (int i = 0; i < N; ++i)
12      A[i] = // do something here
13  }
14  return 0;
15 }
```

What is happening here?

Nested parallelism! Each thread in the outer region executes a parallel `for`-loop on its own.



1.3.1.5 Synchronization Constructs

在很多时候，需要线程之间团结协作完成某个任务，这就要求线程能够完成一致协调合作。OpenMP提供了多个操作来实现线程之间的同步和互斥。

同步构造 (synchronization constructs) 确保对线程组之间**共享的内存地址**的**一致访问**，用于**顺序约束**和共享数据的**访问保护**。OpenMP supports several synchronization constructs:

- `critical` (critical sections)
- `atomic`
- `barrier`
- `section` `master` (in fact, not a synchronization construction)
- `ordered` `flush` `lock` (not discussed in the lecture)

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

Race Conditions

Let's use an simple example to explain the race conditions

```
1 class MyCounter
2 {
3 public:
4   MyCounter() : count(0) {}
5
6   void increment() { count++; }
7   size_t get_count() const { return count; }
8
9 private:
10   size_t count;
11 };
```

```
1 int main(int argc, char* argv[])
2 {
3   MyCounter counter; // our counter class
4
5 #pragma omp parallel for
6   for (int i = 0; i < 100; ++i)
7     counter.increment();
8
9   cout << counter.get_count() << endl;
10   return 0;
11 }
```

Output of the correspond code:

Incrementing a `MyCounter` object 100 times:

Single thread:

```
$ ./count100
100
$ ./count100
100
$ ./count100
100
```

Sequential is correct!

4 threads:

```
$ ./count100
100
$ ./count100
86
$ ./count100
90
```

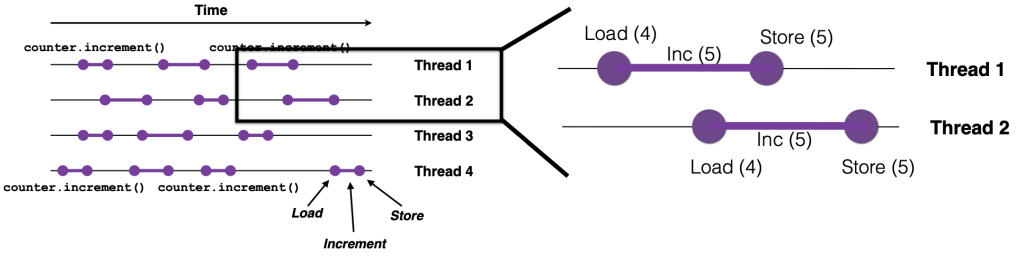
Concurrent is not!

From the above example, we can notice that the code for multi-threads is influenced by race conditions:

```
1 int main(int argc, char* argv[])
2 {
3   MyCounter counter; // our counter class
4
5 #pragma omp parallel for
6   for (int i = 0; i < 100; ++i)
7     counter.increment();
8
9   cout << counter.get_count() << endl;
10   return 0;
11 }
```

Why does this happen?

- The high-level *count++* is not **atomic,** but comprised of multiple instructions: (load, increment, store)
- The ordering of these instructions is not enforced among the multiple threads



- At one time, the shared memory is accessed by multiple threads. Furthermore, we will define the criticial sections as follows:

Critical Sections

```
1 int main(int argc, char* argv[])
2 {
3   MyCounter counter; // our counter class
4
5 #pragma omp parallel for
6   for (int i = 0; i < 100; ++i)
7     counter.increment();
8
9   cout << counter.get_count() << endl;
10   return 0;
11 }
```

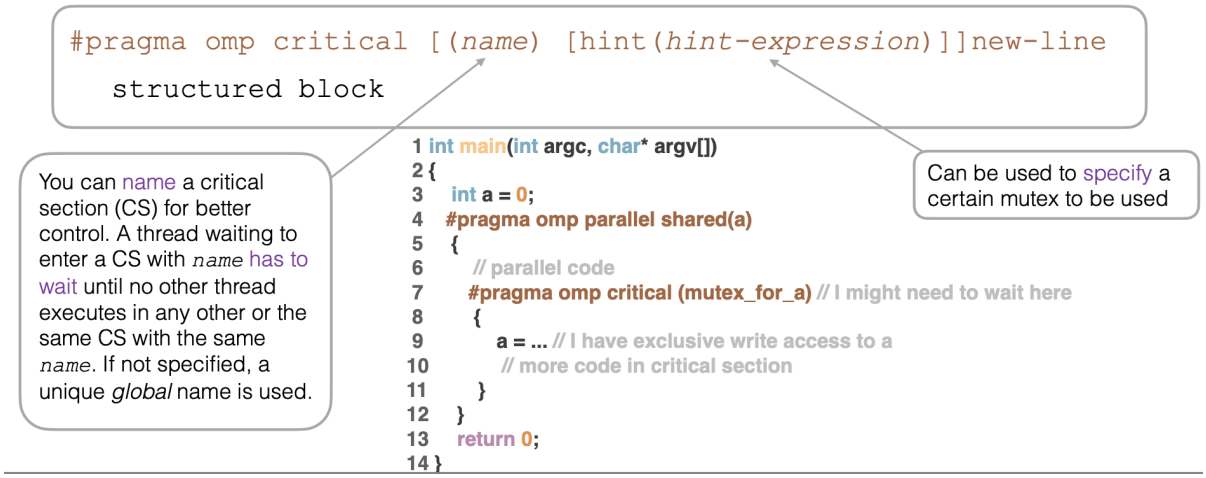
A critical section is part of a code where *multiple threads access a shared variable*

We need to limit threads to access critical sections one at a time: We use **Mutual Exclusion Mechanisms**. Next, we will introduce several commands to use multual exclusion.

Synchronization: critical

The critical construct provides mutual exclusion in critical sections:

- No two threads can simultaneously access the structured block that follows
- The critical section is executed sequentially (no parallelism!)

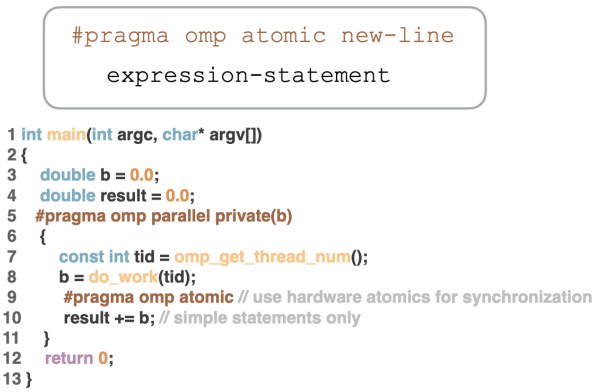


- `#pragma omp critical`
  - OpenMP提供了一个实现互斥的接口。critical选项
  - 使用方法 `#pragma omp critical`
    - 它告诉编译器在接下来的一段代码在同一个时间段将会只由一个线程进行
    - 如果一段代码在同一个时间段被多个线程进行，那么有可能在数据的写入或者读出时，会发生竞争。
- 好处：解决了竞争现象
- 坏处：使用critical会让程序执行减少并行化程度。必须要写代码的人手动判断哪些部分需要critical 【保护某些关键数据，一般来说就是修改我们的共享变量时候】

Synchronization: Atomics

The atomic construct is similar to critical:

- Can only be used for a simple expression (only one memory address can be updated)
- Main difference: uses hardware atomics (no locks!), thus more efficient but limited



- `#pragma omp atomic`

- 在特殊的情况下，除了使用critical选项控制临界区以外，我们还可以使用其他选项去保证内存的控制是原子的
- OpenMP提供了一个选项：atomic（原子）
- 它只在特殊的情况下使用：
  - 在自增或者自减的情况下使用
  - 在二元操作数的情况下使用
- 只会应用于一条指令 (This is fine for simple operations.)

```
#pragma omp atomic
counter +=5;
```

-

```
#pragma omp parallel for
{
    for ( I = 0; I < n; i++){
        #pragma omp critical
        x[index[i]] += workOne(i)
        y[index[i]] += workTwo(i)
    }
}
```

Critical 保护内容：
- workOne()的调用
- Index[i]的寻找
- x[index[i]] 与workOne(i)的加法运算
- 对x数组的赋值

最终对于x的更新来说，这是串行执行的

```
#pragma omp parallel for
{
    for ( I = 0; I < n; i++){
        #pragma omp atomic
        x[index[i]] += workOne(i)
        y[index[i]] += workTwo(i)
    }
}
```

atomic 保护内容：
- x[index[i]] 与workOne(i)的加法运算
- 对x数组的赋值

同样不会产生更新的错误
相对于critical来说，只有index的访问是不安全的，其他都是安全的。

`atomic` is fine for simple operations. But what if we have critical sections comprised of many high-level operations? `mutex.lock()` can be a solution.

Synchronization: Locks

Example: Master/Worker Model

```
std::queue<task> taskQueue;
taskQueue.fill(random);
MyLock mutex;

#pragma omp parallel
{
  while(true)
  {
    mutex.lock(); // Obtain lock
    if (workQueue.empty()) break; // End if no tasks left
    task = workQueue.front() // Obtain task
    workQueue.pop(); // Take it out of the queue
    mutex.unlock(); // Release lock

    performTask(task); // Do task
  }
}
```

This critical section is comprised of many operations that cannot be atomized.

Solution: use mutual exclusion locks (mutex)

Only one thread can enter the mutual exclusive region. Other threads will busy-wait at the *lock* operation until the thread inside runs the *unlock* operation.

Work is performed outside the critical region. Why?

Compiler Assistance for debugging shared memory

GCC (Clang) offer some help for debugging shared memory:

```
g++ -fopenmp -fsanitize=thread -g
```

```
1 int main(void)
2 {
3    int tid = -1;
4    #pragma omp parallel
5    {
6       const int mytid = omp_get_thread_num();
7       tid = mytid; // race condition
8    }
9    std::cout << tid << std::endl;
10    return 0;
11 }
```

```
==================
WARNING: ThreadSanitizer: data race (pid=887504)
  Read of size 4 at 0x7fff3318404c by main thread:
    #0 main ./race.cpp:7 (race+0x401265)

  Previous write of size 4 at 0x7fff3318404c by thread T1:
    #0 main._omp_fn.0 ...
    #1 gomp_thread_start ...

  Location is stack of main thread.

  Location is global '<null>' at 0x000000000000 ([stack]+0x00000001e04c)

  Thread T1 (tid=887506, running) created by main thread at:
    #0 pthread_create ...
    #1 gomp_team_start ...
    #2 __libc_start_main <null> (libc.so.6+0x26ee2)

SUMMARY: ThreadSanitizer: data race ./race.cpp:7 in main
==================
1
ThreadSanitizer: reported 1 warnings
```

Synchronization: Master

The master construct is not a synchronization construct:

- (It belongs to this section in the OpenMP specification)
- Only the master thread of the team executes the structured block
- There is no implicit barrier

```
#pragma omp master new-line
   structured block
```

```
1 int main(int argc, char* argv[])
2 {
3   #pragma omp parallel
4   {
5      do_work();
6     #pragma omp master
7      {
8         update_irregularities();
9      }
10      // only correct if do_more_work() does not depend on memory locations
11      // that are updated in update_irregularities(). Otherwise a barrier is
12      // required: #pragma omp barrier (or use #pragma omp single)
13      do_more_work();
14   }
15   return 0;
16 }
```

Note: The `master` construct is similar to the `single` work sharing construct seen earlier

We have discussed this command in chapter *Single Construct*.

Synchronization: Barriers

The barrier is an explicit synchronization construct:

- Threads are only allowed to continue once all threads have arrived at the barrier (collective)
- Is a standalone directive (no associated code to execute)

```
#pragma omp barrier new-line
```

```
1 int main(int argc, char* argv[])
2 {
3   #pragma omp parallel shared(a, b)
4    {
5      const int tid = omp_get_thread_num();
6      a[tid] = big_calculation(tid);
7
8     #pragma omp barrier // must synchronize here
9
10     #pragma omp for nowait
11      for (int i = 0; i < 1000; ++i)
12        b[i] = another_big_calculation(i, a); // because this depends on a
13   }
14   return 0;
15 }
```

A `barrier` is required here because the next computation depends on the array `a`

Example: Synchronizing Threads

```
double results[threadCount];

#pragma omp parallel
{
  int myId = omp_get_thread_num(); // Identify Thread Id
  results[myId] = randomNumber(); // Store value in vector

   #pragma omp barrier

  for (int i = 0; i < threadCount; i++)
  cout << "My Id: " << myId << " Val: " << results[i] << endl
}
```

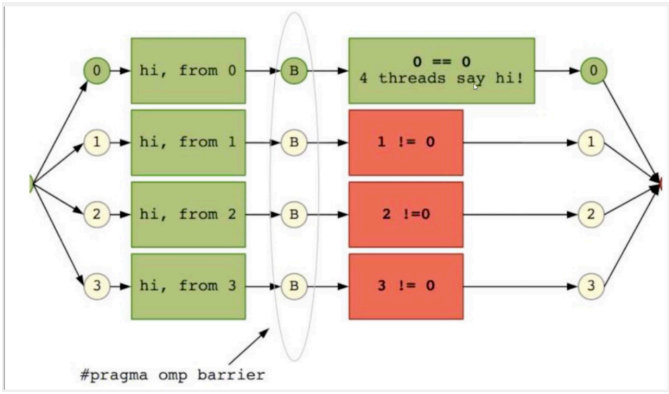
Threads will arrive at this point at different times, printing different results.

Solution: Use a thread-wide barrier

All threads will wait by the barrier until all of them have reached that point (and generated their results).
Now all threads will print the same results.

If different threads are not synchronized, threads will arrive at point at different times, printing different results, we can use following command for synchronization:

- `barrier` 实现不同thread之间的同步
  - 用与实现同步的一种手段。他会在代码的某个点，令线程停下直到所有的线程都到达该地方。
  - 使用的语法如下：
  - `#pragma omp barrier`
  - 许多情况下，它已经能够自动的插入到工作区结尾。比如说在for，single。但是它能够被nowait禁用。
  -



Synchronization: Implicit barrier's

Remember: The following constructs imply an implicit barrier at the end:

- `parallel`
- Work sharing constructs:
  - `for`
  - `sections`
  - `single`

Note: You can skip the implicit barrier of the work sharing constructs with the nowait clause (except if combined with parallel)

- `nowait`
  - 用于打断自动添加的barrier的类型
  - 如parallel中的for以及single
    - `#pragma omp for nowait`
    - `#pragma omp single nowait`

1.3.2 Data Environment

Introduction

Recall: We are working in a shared memory environment.

- In OpenMP, variables outside a parallel region are shared by default.
- Global/static variables are shared.
- Not everything is shared in OpenMP.
  - Loop index variables
  - Automatic variables in functions that are called inside the parallel region.

Three types of data-sharing attributes in OpenMP:

1. Predetermined (eg. Loop counters, `threadprivate` variables)
2. Explicitly determined (what you specify in a clause of a construt)
3. Implicitly determined (everything else)

Note: Data-sharing clauses can only be used in the lexical extent of the OpenMP directive.

Commonly used cluase:

- `default(shared | none)`
- `shared(list)`

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`

Default Variable 默认变量:

- `pragma omp ...default (shared | none)`
  - You can chang the default (shared, implicitly determined). The clause must only appear once
  - `none` means that every variable you reference in a parallel region must be explicitly determined (unless its attribute is predetermined)
-

Example:

```
1 int main(int argc, char* argv[])
2 {
3   int var;
4  #pragma omp parallel default(none) shared(var)
5   {
6     // the variable var is shared (explicitly determined)
7     // the above is the similar to:
8     //  #pragma omp parallel
9     // but now var is implicitly determined
10  }
11  return 0;
12 }
```

Use **`shared`**`(list)` to explicitly determine *a comma separated list* of variables as shared

Private variable 私有变量:

- `#pragma omp ... private(<variable list>)`
  - 变量作用域分为私有变量和公有变量。上面的命令能够直接告诉编译器去使得共享变量作为每个线程中的私有变量。 Creates a private copy of each variable in the list. The private copy has a different address (avoiding data races)
  - 如果j被定义为私有变量，那么在for循环里面，所有的线程都不能访问其他j（尽管j是共享变量）
  - 所有线程都不会使用到先前的定义。也就是私有变量在各个thread中会被初始化，和之前的值完全没有关系 The private copy is not initialized, no association with the global variable.
  - 所有线程都不能给共享的j赋值
  - 在循环的入口以及出口，都不会进行定义（就是前两条的总结）
  -

Example:

```
1 int main(int argc, char* argv[])
2 {
3   int sum = 0;
4   #pragma omp parallel for private(sum)
5   for (int i = 0; i < 1000; ++i)
6   {
7     // Threads have not initialized sum, its value is undefined. Inside
8     // this region, each thread observes a different address for the sum
9     // variable.
10    sum = sum + i;
11  }
12  // Because sum has been declared private in the parallel region, its value
13  // after the parallel region is the same as it was before entering the
14  // region.
15  return 0;
16 }
```

`sum` is *not* initialized to zero. Its value is not defined. You have to assign its initial value in the parallel region.

- `#pragma omp parallel for firstprivate(x)`
  - Special case of `private (list)`

- 告诉编译器私有变量在第一个循环会继承共享变量的值 The private copy is initialized to the value seen by the `master` thread. Its association with the global variable is its initialization.
- 其使用方法和private几乎一致。
- 其本质就是在第一个循环实现了数据的一个拷贝（从共有 → 私有）
-

Example:

```
1 int main(int argc, char* argv[])
2 {
3   int sum = 0;
4   #pragma omp parallel for firstprivate(sum)
5   for (int i = 0; i < 1000; ++i)
6   {
7     // sum has been initialized to sum = 0. Inside this region, each thread
8     // observes a different address for the sum variable.
9     sum = sum + i;
10  }
11  // Because sum has been declared firstprivate in the parallel region, its
12  // value after the parallel region is the same as it was before entering
13  // the region.
14  return 0;
15 }
```

sum is initialized to zero.

- `#pragma omp parallel for lastprivate(x)`
  - Special case of `private (list)`
  - The private copy is not initialized, as in `private(list)`
  - Its association with the global variable is its update after the parallel region (careful: data races) 告诉编译器私有变量会在最后一个循环出去的时候，用私有变量的值替换掉我们的共享变量的值。
  - 因为不同thread的速度不一样，到底是哪个thread进行赋值呢？答：当负责最后一个iteration的线程离开循环的时候，它会将该私有变量的值赋值给当前共享变量的值。
  -

Example Loop Construct:

```
1 int main(int argc, char* argv[])
2 {
3   int a = 0;
4   #pragma omp parallel for lastprivate(a)
5   for (int i = 0; i < 100; ++i)
6     a = i;
7   // after the parallel region: a = 99
8   return 0;
9 }
```

Example Sections Construct:

```
1 int main(int argc, char* argv[])
2 {
3   int a = 0;
4   #pragma omp parallel sections lastprivate(a)
5   {
6     #pragma omp section // first section
7       a = 1;
8     #pragma omp section // second section (and the last)
9       a = 2;
10  }
11  cout << a;
12  return 0;
13 }
```

1.4 Addition

This chapter will complement further topics about the use of OpenMP. Following topics are rarely discussed.

1.4.1 False Sharing

1.4.2 Library Routines

1.4.3 Environment Variables

1.5 Common Usage

1. `#pragma omp parallel num_threads(i) if (paralle: i>2)` : Explicitly specify to which directive the if applies. Useful for composite and combined constructs. If not specified, the if applies to all constructs to which an if clause can apply.

```
for (int i = 1; i <= 4; ++i){
cout<<"Number of threads ="<<i<<endl;
#pragma omp parallel num_threads(i) if (paralle: i>2)
  {
    const int tid=omp_get_thread_num();
    #pragma omp critical//the next line is a critical section!
    cout<<"Hello from the thread"<<tid<<'\n';
  }}
```

---

2 MPI

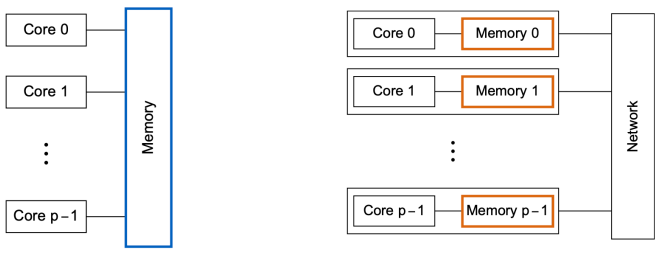
References:

- LLNL

2.1 Distributed systems

2.1.1 Hardware Model

Recall: Shared vs Distributed memory



Shared memory system: each core has access to a common memory

Distributed memory system: each core has access to a private memory

Shared vs Distributed memory parallelism

- Shared memory parallelism:
  - Enabled by multithreading*
  - Easy to access data from any threads
  - Might have race conditions
  - Must be aware of: false sharing, NUMA architecture

  - Looks simple, but there is a lot of implicit complexity

- Distributed memory parallelism:
  - Only possible through multiprocessing
  - Each process has its own private memory
  - Must explicitly exchange data across processes People prescribed
  - No race conditions No shared memory, no race conditions
  - No false sharing
  - Can scale beyond a single computer to solve much larger problems

2.1.2 Programming Model

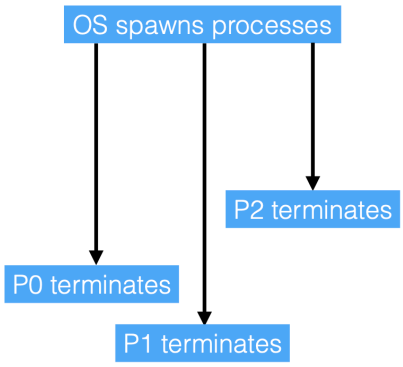
SPMD Programming Model:

Single Program, Multiple Data

- A programming model for distributed memory parallelism.
- Parallelism is achieved by creating multiple instances (processes) of the same program. 不像OpenMP是以产生多种线程来实现并行。
- Each process operates on its own set of data.
- Each process might follow a different execution path.

```
1 if (IAmThisProcess())
2     DoThis();
3 else if (IAmThatProcess())
4     DoThat();
5 ...
```

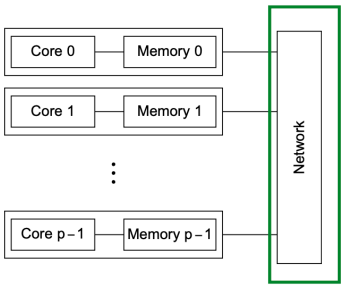
```
1 if (dataOnMyProcess() > 4)
2     DoThis();
3 else
4     DoThat();
5 ...
```

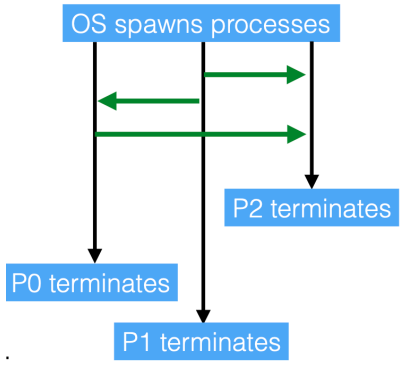


OpenMP就相当于OS开了一个程序（进程process），然后把这个程序执行过程中的一段段小的任务切割，分配到该process下的不同线程（thread）去处理。而MPI则是一个OS开了多个程序（多个进程），各自的算完之后再进行数据之间的交流和传播。

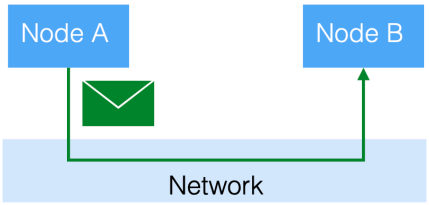
Communication in SPMD:

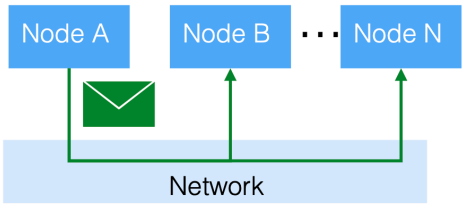
- In SPMD, Processes cannot communicate or synchronize implicitly through shared memory. We need an explicit communication mechanism, e.g: Message Passing



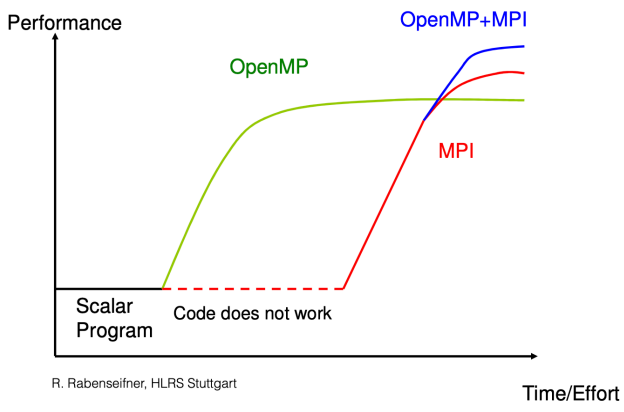


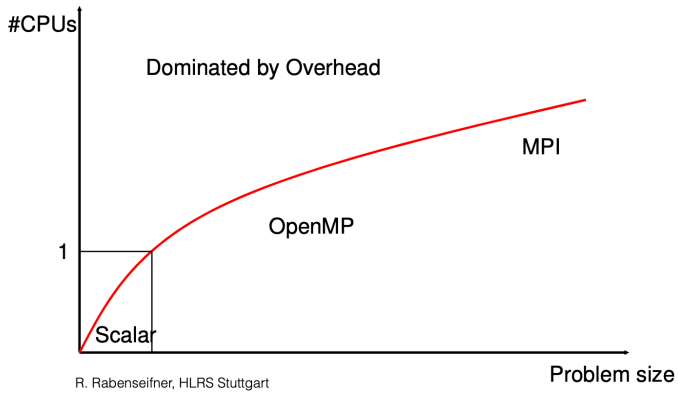
- Messages (communicated in the network) can be classified in terms of participants:
  - Point-to-point communication: a process A sends a message to another process B
  - Collective communication: More complex communication patterns involving multiple processes. In general, can be expressed in terms of Point-to-Point communications (we can write a loop to achieve collective communication by point-to-point communication)





Productivity Comparison





- For very large problem, we can consider to use the combination of OpenMP + MPI (Multiprocess, for every process, we can use multithreading)

2.1.3 MPI

MPI: Message Passing Interface:

- MPI defines a **standard API** for message passing in SPMD applications 个人理解来看就是个语法标准
- Implementations are C libraries
- There are several implementations:
  - MPICH https://www.mpich.org/

  - OpenMPI https://www.open-mpi.org/
  - MVAPICH (cluster oriented) http://mvapich.cse.ohio-state.edu/ DeinoMPI (windows) http://mpi.deino.net/

- MPI allows exploiting distributed memory systems
- MPI(Message Passing Interface)，由其字面意思也可些许看出，是一个信息传递接口。可以理解为是一种独立于语言的信息传递标准。而OpenMPI和MPICH等是对这种标准的具体实现。也就是说，OpenMPI和MPICH这类库是具体用代码实现浏MPI标准。因此我们需要安装OpenMPI或者MPICH去实现我们所学的MPI的信息传递标准。

History of the MPI standard

MPI-1 - 1992: Standardization of the message passing protocol into MPI
More than 60 people from 40 organizations involved
Point to point, blocking collective communications,
One-sided communication, I/O, creation of processes…

MPI-2 - 1998: FORTRAN and C++ wrappers (C++ is now deprecated)

MPI-3 - 2012: Non-blocking collective communications + One-Sided Communication

MPI-3.1 - 2015: Non-blocking collective I/O communications

MPI-4 - 2021: Persistent collectives / 64-byte sizes.

- The MPI standard API goals:
  - Work on distributed memory, shared memory and hybrid systems
  - Portable, efficient, easy to use

Compile and execute MPI programs

- MPI is simply a C library, we only need to link it to the main program
- MPI usually provides a wrapper mpic++ with compiler flags

OpenMPI: `mpic++ main.cpp -o main` `mpiexe -n 3 main` launch 3 processes able to communicate with MPI

Compile and execute MPI programs: Euler

You need to load the MPI module (after the compiler module if one is needed). On Euler, Open MPI is recommended; you can also try MVAPICH2 or Intel MPI.

`module load openmpi` `mpic++ main.cpp -o main`

Laugh a job with `bsub` The number of processes does not need to be passed to mpirun

`module load openmpi` `sub -n 4 mirin ./main`

Simple example with MPI

```
1 #include <mpi.h>
2
3 int main(int argc, char **argv)
4 {
5     MPI_Init(&argc, &argv);
6
7     printf("Hello HPCSE class!\n");
8
9     MPI_Finalize();
10    return 0;
11 }
```

include header

Can NOT call MPI routines!

Initialize MPI environment

Can call MPI routines

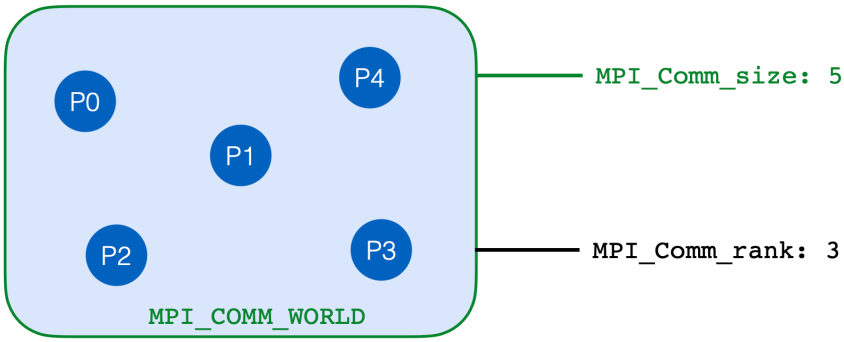
Finalize MPI environment

Can NOT call MPI routines!

```
> mpic++ hello.cpp -o hello
> mpiexec -n 2 ./hello
Hello HPCSE class!
Hello HPCSE class!
```

MPI Communicators:

- represents a grouping of processes and allows communication between them.
- Each process has a unique rank within a given communicator
- `MPI_COMM_WORLD` is the communicator for all processes; defined after `MPI_Init` is called
- All MPI routines involving communications require a communicator object of type `MPI_Comm`



MPI Rank and Size

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

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);

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

    printf("Hello from rank %d out of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Communicator of ALL processes

Get process ID

Get number of processes

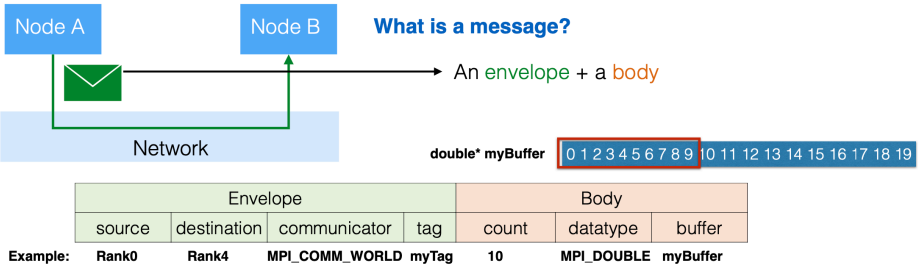
```
> mpiexec -n 3 ./hello
Hello from rank 1 out of 3
Hello from rank 0 out of 3
Hello from rank 2 out of 3
```

Asynchronous execution:
Order is non deterministic

- `rank` 是当前process所处的rank

2.2 Blocking point-to-point communication

Point-to-point communication



Source, Destination:
Rank IDs within the communicator
Tag:
User-defined message numeric label

Count: number of elements (not bytes)
Datatype: Element data type
Buffer: A pointer to the data itself

2.2.1 MPI_Send, MPI_Recv

常用命令:

1. `MPI_Init`
   - 任何MPI程序都应该首先调用该函数。 此函数不必深究，只需在MPI程序开始时调用即可（必须保证程序中第一个调用的MPI函数是这个函数）。

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

2. `MPI_Finalize`
3. - 任何MPI程序结束时，都需要调用该函数。

```c
MPI_Finalize() //C++
```

3. `MPI_COMM_RANK`

   - 该函数是获得当前进程的进程标识，如进程0在执行该函数时，可以获得返回值0。可以看出该函数接口有两个参数，前者为进程所在的通信域，后者为返回的进程号。通信域可以理解为给进程分组，比如有0-5这六个进程。可以通过定义通信域，来将比如[0,1,5]这三个进程分为一组，这样就可以针对该组进行“组”操作，比如规约之类的操作。这类概念会在HPCSE II。MPI_COMM_WORLD是MPI已经预定义好的通信域，是一个包含所有进程的通信域，目前只需要用该通信域即可。

     在调用该函数时，需要先定义一个整型变量如myid，不需要赋值。将该变量传入函数中，会将该进程号存入myid变量中并返回。

     比如，让进程0输出Hello，让进程1输出Hi就可以写成如下方式。

   -

     ```
     int MPI_Comm_Rank(MPI_Comm comm, int *rank)
     ```

   -

     ```
     #include "mpi.h"
     int main(int *argc,char* argv[])
     {
         int myid;
         MPI_Init(&argc,&argv);
         MPI_Comm_Rank(MPI_COMM_WORLD,&myid);
         if(myid==0)
         {
             printf("Hello!");
         }
         if(myid==1)
         {
             printf("Hi!");
         }
         MPI_Finalize();
     }
     ```

4. `MPI_COMM_SIZE`

   ```
   int MPI_Comm_Size(MPI_Comm, int *size)
   ```

   - 该函数是获取该通信域内的总进程数，如果通信域为MP_COMM_WORLD，即获取总进程数，使用方法和MPI_COMM_RANK相近。

5. `MPI_SEND`

   ```
   int MPI_Send(type* buf, int count, MPI_Datatype, int dest, int tag, MPI_Comm
   comm)
   ```

- 该函数为发送函数，用于进程间发送消息，如进程0计算得到的结果A，需要传给进程1，就需要调用该函数。
- 该函数参数过多，不过这些参数都很有必要存在。

  这些参数均为传入的参数，其中buf为你需要传递的数据的起始地址，比如你要传递一个数组A，长度是5，则buf为数组A的首地址。count即为长度，从首地址之后count个变量。datatype为变量类型，注意该位置的变量类型是MPI预定义的变量类型，比如需要传递的是C++的int型，则在此处需要传入的参数是MPI_INT，其余同理。dest为接收的进程号，即被传递信息进程的进程号。tag为信息标志，同为整型变量，发送和接收需要tag一致，这将可以区分同一目的地的不同消息。比如进程0给进程1分别发送了数据A和数据B，tag可分别定义成0和1，这样在进程1接收时同样设置tag0和1去接收，避免接收混乱。

6. `MPI_RECV`

```
int MPI_Recv(type* buf, int count, MPI_Datatype, int source, int tag, MPI_Comm
comm, MPI_Status *status)
```

- 该函数为MPI的接收函数，需要和MPI_SEND成对出现。
- 参数和MPI_SEND大体相同，不同的是source这一参数，这一参数标明从哪个进程接收消息。最后多一个用于返回状态信息的参数status。

  在C和C++中，status的变量类型为MPI_Status，分别有三个域，可以通过status.MPI_SOURCE，status.MPI_TAG和status.MPI_ERROR的方式调用这三个信息。这三个信息分别返回的值是所收到数据发送源的进程号，该消息的tag值和接收操作的错误代码。

  在Fortran中，status的变量类型为长度是MPI_STATUS_SIZE的整形数组。通过status(MPI_SOURCE)，status(MPI_TAG)和status(MPI_ERROR)来调用。

  SEND和RECV需要成对出现，若两进程需要相互发送消息时，对调用的顺序也有要求，不然可能会出现死锁或内存溢出等比较严重的问题，具体在之后的对等模式这一章中详细介绍。
- The count variable is the maximum size that can be received, message can be smaller
- The size of the actual message can be retrieved with MPI_Probe or from the status

- An `MPI_Recv` matches a message sent by `MPI_Send` only if comm, tag, source and dest match.

MPI Datatypes

MPI_Datatype	C type
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG_INT	long long int
MPI_LONG_LONG	long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- MPI does not perform type conversion (e.g. `float` to `int`)
- MPI performs representation conversion (e.g. little endian to big endian) when possible
- MPI allows you to define custom datatypes (covered in HPCSE II)

Examples: sending an integer

```
> mpiexec -n 4 ./message_int
Rank 1 received the following integer: 1
Rank 2 received the following integer: 4
Rank 3 received the following integer: 9
```

```
1 #include <mpi.h>
2
3 int main(int argc, char **argv)
4 {
5     int rank, size;
6     MPI_Init(&argc, &argv);
7     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8     MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10    int tag = 42;
11    int message;
12
13    if (rank == 0) {
14        for (int i = 1; i < size; ++i) {
15            message = i * i;
16            MPI_Send(&message, 1, MPI_INT, i, tag, MPI_COMM_WORLD);
17        }
18    }
19    else {
20        MPI_Recv(&message, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
21        printf("Rank %d received the following integer: %d\n", rank, message);
22    }
23
24    MPI_Finalize();
25    return 0;
26 }
```

send one integer to all ranks != 0

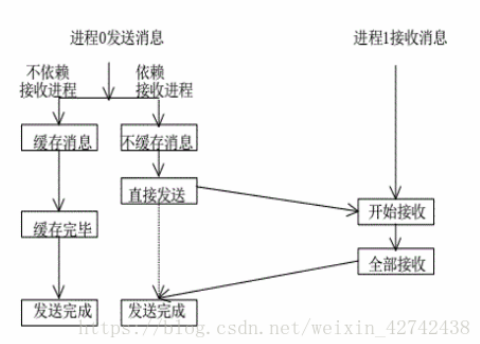
receive one integer from rank 0

2.2.2 Send modes

MPI_Send: Communication Modes:

MPI四种通信模式及其函数：Ref. 1 Ref. 2

1. 标准通信模式：`MPI_SEND`
   (a) “standard send”: Uses buffered send for small messages. Uses synchronous send for large messages. Prefer non-buffered sends to avoid unnecessary copies.
   (b) 标准通信模式下，是否对发送的数据进行缓存是由MPI自身决定的，而不是由并行程序员来控制。
   如果MPI不缓存将要发送的数据：对于阻塞通信，只有当相应的接收调用被执行后，并且发送数据完全到达接收缓冲区后，发送操作才算完成。对于非阻塞通信，发送操作虽然没有完成，但是发送调用可以正确返回，程序可以接下来执行其它的操作。[对于非阻塞通信，发送操作甚至可能没有完全放倒缓存区，就直接返回了]
   如果MPI缓存将要发出的数据：发送操作不管接收操作是否执行，都可以进行 而，且发送操作可以正确返回而不要求接收操作收到发送的数据。
   (c)



2. 缓存通信模式：`MPI_BSEND`
   (a) “buffered send”: Stores the message in a buffer and returns immediately. See MPI_Buffer_attach to allocate buffer for MPI
   (b) 缓存通信模式下，由用户直接对通信缓冲区进行申请、使用和释放。对通信缓冲区的合理与正确使用是由程序设计人员自己保证的。

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

缓存通信模式不管接收操作是否启动，发送操作都可以执行。
采用缓存通信模式时，消息发送能否进行及能否正确返回不依赖于接收进程，完全依赖于是否有足够的通信缓冲区可用。对于非阻塞发送，正确退出并不意味者缓冲区可以被其它的操作任意使用，对于阻塞发送返回后其缓冲区是可以重用的。

`int MPI_Buffer_attach(void *buffer, int size) //用于申请缓存`
`int MPI_Buffer_detach(void **buffer, int *size) //用于释放缓存`

释放缓存是阻塞调用，它一直等到使用该缓存的消息发送完成后才返回，这一调用返回后用户可以重新使用该缓冲区或者将这一缓冲区释放。

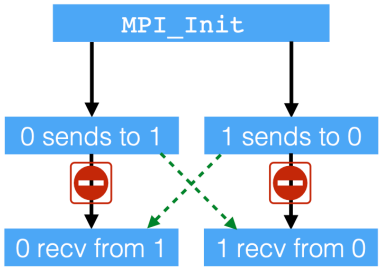
3. 同步通信模式：`MPI_SSEND`
   (a) “Synchronous send”: Will return only once a matching recv has been posted and started to receive the message.
   (b) 同步通信模式的开始不依赖于接收进程相应的接收操作是否已经启动，但是同步发送却必须等到相应的接收进程开始后才可以正确返回。因此，同步发送返回后，意味着发送缓冲区中的数据已经全部被系统缓冲区缓存，并且已经开始发送。这样当同步发送返回后，发送缓冲区可以被释放或重新使用。
4. 就绪通信模式：`MPI_RSEND`
   (a) "Ready send": Can only be called if a matching recv is already posted (user responsibility)
   (b) 在就绪通信模式中，只有当接收进程的接收操作已经启动时，才可以在发送进程启动发送操作。否则，当发送操作启动而相应的接收还没有启动时，发送操作将出错。对于非阻塞发送操作的正确返回，并不意味着发送已完成，但对于阻塞发送的正确返回，则发送缓冲区可以重复使用。

四种通信模式的区别都在消息发送端，而消息接收端的操作都是MPI_RECV。

2.2.3 Watch out for deadlocks!

Example: data exchange

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int tag1 = 42, tag2 = 43;
    int sendMessage, recvMessage;

    if (rank == 0) {
        sendMessage = 7;
        MPI_Send(&sendMessage, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
        MPI_Recv(&recvMessage, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else if (rank == 1) {
        sendMessage = 14;
        MPI_Send(&sendMessage, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD);
        MPI_Recv(&recvMessage, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    printf("Rank %d received the following integer: %d\n", rank, recvMessage);

    MPI_Finalize();
    return 0;
}
```

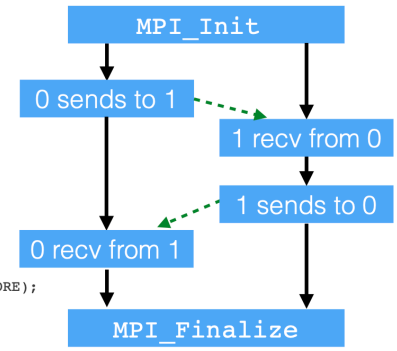


Deadlock

- Actually, whether use buffer or not of `MPI_Send` is determined by the computer, but it prefer synchronous . Therefore, `MPI_Send` will give rise to dead lock.

⇓ One possible deadlock fix: change ordering

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int tag1 = 42, tag2 = 43;
    int sendMessage, recvMessage;

    if (rank == 0) {
        sendMessage = 7;
        MPI_Send(&sendMessage, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
        MPI_Recv(&recvMessage, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else if (rank == 1) {
        sendMessage = 14;
        MPI_Recv(&recvMessage, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&sendMessage, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD);
    }

    printf("Rank %d received the following integer: %d\n", rank, recvMessage);

    MPI_Finalize();
    return 0;
}
```



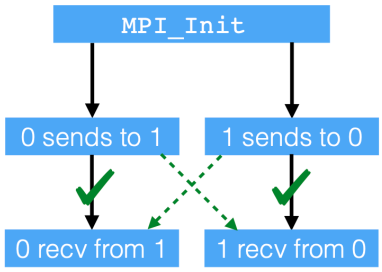

```
> mpiexec -n 2 ./message_exchange_fixed
Rank 0 received the following integer: 14
Rank 1 received the following integer: 7
```

⇓ Another possible deadlock fix: buffer send

```c
int rank, size, bufsize;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int tag1 = 42, tag2 = 43;
int sendMessage, recvMessage;

MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &bufsize);
bufsize += MPI_BSEND_OVERHEAD;

char *buffer = new char[bufsize];
MPI_Buffer_attach(buffer, bufsize);

if (rank == 0) {
    sendMessage = 7;
    MPI_Bsend(&sendMessage, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
    MPI_Recv(&recvMessage, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if (rank == 1) {
    sendMessage = 14;
    MPI_Bsend(&sendMessage, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD);
    MPI_Recv(&recvMessage, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

printf("Rank %d received the following integer: %d\n", rank, recvMessage);

MPI_Buffer_detach(buffer, &bufsize);
delete[] buffer;

MPI_Finalize();
```

memory allocation

release memory



Buffered send can return before the message is received

```
> mpiexec -n 2 ./message_exchange_fixed
Rank 0 received the following integer: 14
Rank 1 received the following integer: 7
```

Example: computing π

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

Serial code:

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    long nsteps = 100000000;
    double sum = 0;

    for (long i = 0; i < nsteps; ++i)
        sum += (1.0 - 2.0 * (i % 2)) / (2.0 * i + 1.0);

    printf("pi is around %g\n", sum * 4.0);

    return 0;
}
```

How to parallelise this with MPI?

- Each rank perform its own chunk of the sum
- All ranks need to communicate and sum up its result to one rank only
- This rank should report the result

```
> ./pi
pi is around 3.14159
```

⇓ We can split the loop into chunks and perform computing

```
4 …
5     MPI_Init(&argc, &argv);
6
7     int size, rank;
8     const int tag = 42;
9     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10    MPI_Comm_size(MPI_COMM_WORLD, &size);
11
12    long nsteps = 100000000;
13    double sum = 0;
14
15    long chunk = (nsteps + size - 1)  / size;
16    long start = chunk * rank;
17    long end   = std::min(chunk * (rank + 1), nsteps);
18
19    for (long i = start; i < end; ++i)
20        sum += (1.0 - 2.0 * (i % 2)) / (2.0 * i + 1.0);
21
22    if (rank == 0) {
23        double tot_sum = sum;
24        double other;
25        for (int i = 1; i < size; ++i) {
26            MPI_Recv(&other, 1, MPI_DOUBLE, i, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
27            tot_sum += other;
28        }
29        printf("pi is around %g\n", tot_sum * 4.0);
30    } else {
31        MPI_Send(&sum, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
32    }
33
34    MPI_Finalize();
35 …
```

```
> mpiexec -n 4 ./pi_mpi
pi is around 3.14159
```

split the loop into chunks

Reduction takes O(size) steps!
Bad if we have many ranks…

collect and add sums from other ranks

send partial sum to rank 0

- HOWEVER, the disadvantages of this task is that Reduction taks O(size) steps! And we know that communction is very pricy. Therefore, we will introduce the block collective communications for better performance.

2.3 Blocking collective communication

2.3.1 Common operations

Some very common collective operations have been implemented in MPI. Blocking: the operation is completed for the process once the function returns.

用blocking的时候要注意，所有rank都要包含这些代码行。由于是blocking，因此只有当所有的rank都运行到这个代码行时，collective communication才可以启动!

These operations include:

- Reduce
- AllReduce
- Broadcast
- Gather
- AllGather
- Scatter
- Barrier
- AllToAll (“complete exchange”)
- Scan
- ExScan
- Reduce_Scatter

All to One
One to All
All to All
Other

They involve all ranks of a same communicator and therefore must be called by all the ranks of the concerned communicator

Blocking: the operation is completed **for the process** once the function returns

2.3.2 MPI usage

Reduction

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
int root, MPI_Comm comm);
```

- **`sendbuf`**: input data to reduce
- **`recvbuf`**: output result (only for root)
- **`count`**: number of elements (per process) to reduce
- **`datatype`**: type of element to reduce
- **`op`**: operation to operate (see next slide)
- **`root`**: The rank id to which the result is output
- **`comm`**: communicator, must contain participating ranks

This is a collective operation, meaning that **ALL** processes in the communicator must call it

Data can be reduced in place to save memory, in which case **`recvbuf`** is input-output and **`sendbuf`** should be set to **`MPI_IN_PLACE`**

MPI implements a few basic operations:

MPI_Op	Operation
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

You can define a custom operation using **`MPI_OP_CREATE`**

The operation must be associative

The operation may be not commutative, but it's better if it is because allows MPI to reorder ops for performance

⇓ Back to example: computing π

```cpp
#include <mpi.h>

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

    int size, rank;
    const int tag = 42;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    long nsteps = 100000000;
    double sum = 0, totSum = 0.;

    long chunk = (nsteps + size - 1)  / size;
    long start = chunk * rank;
    long end   = std::min(chunk * (rank + 1), nsteps);

    for (long i = start; i < end; ++i)
        sum += (1.0 - 2.0 * (i % 2)) / (2.0 * i + 1.0);

    MPI_Reduce(&sum, &totSum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
        printf("pi is around %g\n", totSum * 4.0);

    MPI_Finalize();

    return 0;
}
```

```
> mpiexec -n 4 ./pi_mpi
pi is around 3.14159
```

reduction is called by all processes and send the result to rank 0

only rank 0 has the result

- The code is much more concise.

```cpp
for (long i = start; i < end; ++i)
    sum += (1.0 - 2.0 * (i % 2)) / (2.0 * i + 1.0);

MPI_Reduce(rank ? &sum : MPI_IN_PLACE, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0)
    printf("pi is around %g\n", sum * 4.0);
```

In-place reduction

- In-place reduction for saving memory.

⇓ MAX_LOC and MIN_LOC operations

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

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    srand48(42 + rank * 42); // some unique seed per rank
    double myval = drand48();

    printf("rank %d has value %g\n", rank, myval);

    struct DoubleInt {
        double val;
        int key;
    } sendBuf = {myval, rank}, recvBuf;

    MPI_Reduce(&sendBuf, &recvBuf, 1, MPI_DOUBLE_INT, MPI_MINLOC, 0, MPI_COMM_WORLD);

    if (rank == 0)
        printf("The smallest value is %g from rank %d\n", recvBuf.val, recvBuf.key);

    MPI_Finalize();
    return 0;
}
```

MPI defines more data types of the form
MPI_X_INT
X = FLOAT, DOUBLE, INT...

This allows to use the **MPI_MINLOC** and **MPI_MAXLOC** operations

```
> mpiexec -n 4 ./minloc
rank 3 has value 0.465616
rank 0 has value 0.744525
rank 2 has value 0.891919
rank 1 has value 0.318222
The smallest value is 0.318222 from rank 1
```

- We can notice that the the send argument is a struct.

Barrier

```c
int MPI_Barrier(MPI_Comm comm);
```

A barrier synchronizes all processes in a communicator. The function returns after all processes have entered the call.

Again: all threads in the communicator MUST call the barrier
Otherwise this creates a deadlock

Even ranks wait for odd ranks forever

```c
#include <mpi.h>

void doWork();
void doWorkNeedingSynchronization();

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    doWork();

    if (rank % 2 == 0) {
        MPI_Barrier(MPI_COMM_WORLD);
        doWorkNeedingSynchronization();
    }

    MPI_Finalize();
    return 0;
}
```

- Every process in this communicator should use `MPI_Barrier`. Otherwise, it will cause dead lock.

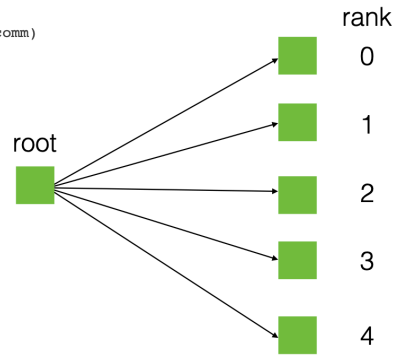
Broadcast

```c
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

A broadcast sends data from the root rank to all ranks in the communicator.

A naive implementation:

```
1 void bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
2 {
3     int rank, size;
4     MPI_Comm_rank(comm, &rank);
5     MPI_Comm_size(comm, &size);
6
7     if (rank == root) {
8         for (int i = 0; i < size; ++i) {
9             if (i == rank) continue;
10            MPI_Send(buffer, count, datatype, i, 42, comm);
11        }
12    }
13    else {
14        MPI_Recv(buffer, count, datatype, root, 42, comm, MPI_STATUS_IGNORE);
15    }
16 }
```



MPI makes use of a tree structure to obtain better performance

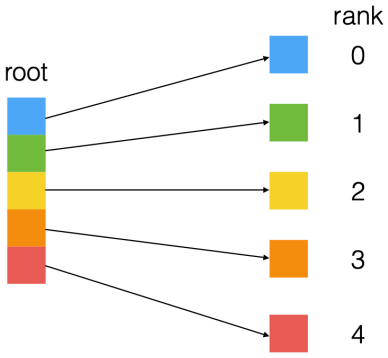
Scatter

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Send data in separate chunks from the root rank to all ranks in the communicator.

sendbuf, **sendcount**, **sendtype**: Only significant to root

sendcount: Number of element sent to each process



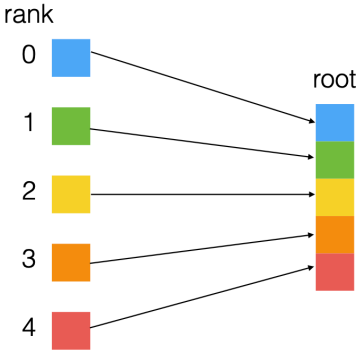
Gather

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Collect chunks of data from all ranks in the communicator to the root rank (inverse of scatter)

recvbuf, **recvcount**, **recvtype**: Only significant to root

recvcount: Number of element for a single receive

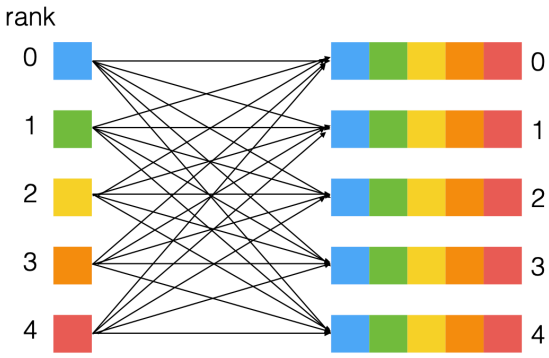


AllGather

```
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
```

Same as Gather but all ranks get the result.

recvcount: Number of element for a single receive

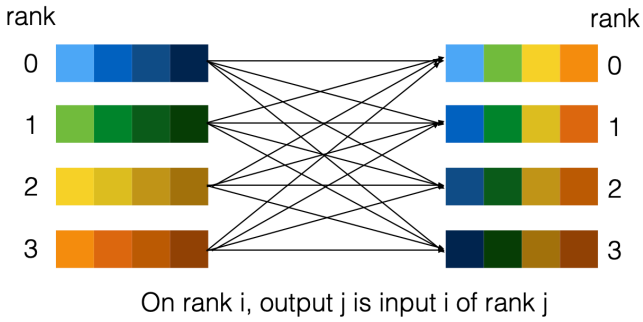


Alltoall

```
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount,MPI_Datatype recvtype, MPI_Comm comm);
```

Shuffle the data between ranks: acts like a transpose.

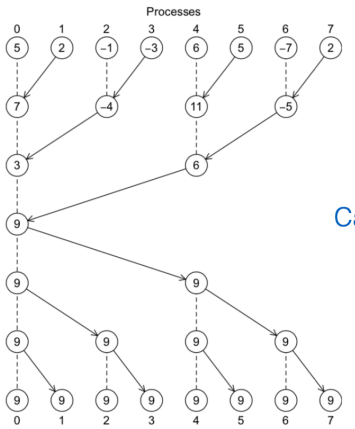
Can be seen as an extension of AllGather, except that each rank has different data



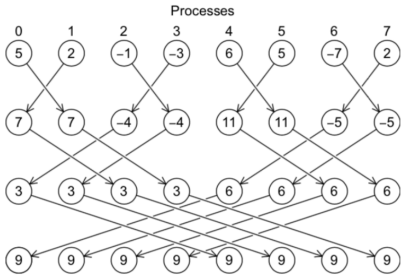
AllReduce

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Reduce the data, result is broadcasted to all ranks. We use it when computing histogram in Finite Difference.



Can be more clever!



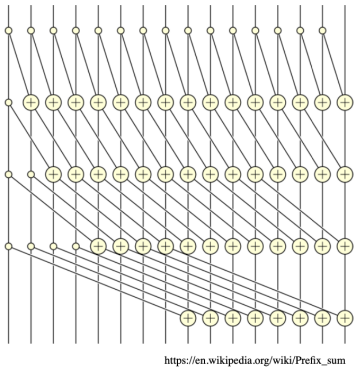
Scan

```
int MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Perform an inclusive prefix reduction

Example for sum: $y_i = \sum_{j=0}^{i} x_j$

rank	0	1	2	3	4	5
input	2	1	4	5	3	1
output	2	3	7	12	15	16



https://en.wikipedia.org/wiki/Prefix_sum

ExScan

```
int MPI_Exscan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm);
```

Perform an exclusive prefix reduction.

Example for sum: $y_i = \sum_{j=0}^{i-1} x_j$

Same as scan, excepts that we do not count the local data

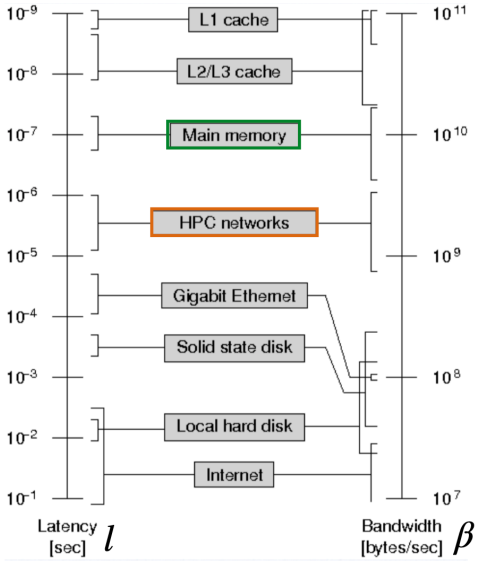
rank	0	1	2	3	4	5
input	2	1	4	5	3	1
output	0	2	3	7	12	15

2.4 More on blocking point-to-point communication

2.4.1 MPI_Probe, MPI_Sendrecv

Communication Cost: Bandwidth and Latency

Time to transmit message of size S : $T = l + S/\beta$



- Implications:
  - For small messages, the fixed cost of **latency** (l) is the dominant factor in overhead.
  - For large messages, **bandwidth** (β) cost becomes more important.
- Network communication is much slower than main memory access
- Conclusion: We need to work as hard (and even harder) to reduce the cost of network overheads as we did to reduce in-memory data motion.

Problem: Unkown size message:

What if we don't know the size of the message on the receive side in advance? In previous chapter, the message is previously known by us.

- potential solution: using two message, one for size first, then another for content. *problem: we pay twice the latency!*

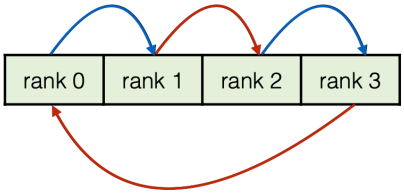
```
1  std::vector<double> message;
2
3  if (rank == 0) {
4      message.resize(rand() % 2048, 42.0);
5      int size = message.size();
6      MPI_Send(&size, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
7      MPI_Send(message.data(), message.size(), MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
8  }
9  else if (rank == 1) {
10     int size;
11     MPI_Recv(&size, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
12     message.resize(size);
13     MPI_Recv(message.data(), message.size(), MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
14     printf("Rank %d received %d doubles\n", rank, size);
15 }
```

- another solution `MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);`
  - obtain the status of a matching message as if it was obtained from `MPI_Recv`. (Only had the latency overhead!)

```cpp
std::vector<double> message;

if (rank == 0) {
    message.resize(rand() % 2048, 42.0);
    MPI_Send(message.data(), message.size(), MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
}
else if (rank == 1) {
    int size;
    MPI_Status status;
    MPI_Probe(0, tag, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_DOUBLE, &size);
    message.resize(size);
    MPI_Recv(message.data(), message.size(), MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Rank %d received %d doubles\n", rank, size);
}
```

Problem: Cyclic communication:

when you have a cyclic communication, you will be in the deadlock at most cases.




```cpp
std::vector<double> msg(n, (double) rank), lmsg(n);

int next = (rank        + 1) % size;
int prev = (rank + size - 1) % size;

if (rank % 2 == 0) {
    MPI_Send(msg.data(), msg.size(), MPI_DOUBLE, next, tag, MPI_COMM_WORLD);
    MPI_Recv(lmsg.data(), lmsg.size(), MPI_DOUBLE, prev, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else {
    MPI_Recv(lmsg.data(), lmsg.size(), MPI_DOUBLE, prev, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(msg.data(), msg.size(), MPI_DOUBLE, next, tag, MPI_COMM_WORLD);
}
```

- Potential solution: 1. Send from even to odd 2. send from odd to even. *problem: we pay twice the latency!*
- Another solution `MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status);`:
  - Automatically takes care of deadlocks for you, this is just a convenience tool.
  -

```cpp
std::vector<double> msg(n, (double) rank), lmsg(n);

int next = (rank        + 1) % size;
int prev = (rank + size - 1) % size;

MPI_Sendrecv(msg.data(), msg.size(), MPI_DOUBLE, next, tag,
             lmsg.data(), lmsg.size(), MPI_DOUBLE, prev, tag,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

  - This tool is very useful in finite difference.

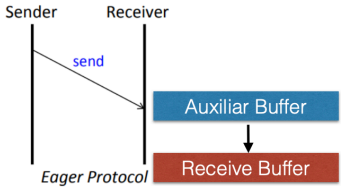
2.4.2 Eager vs RendezVous Protocols

Message passing protocols:

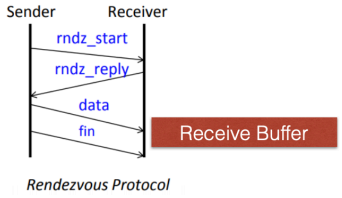
Internal implementation of MPI might use different protocols, e.g:

- Eager (an asynchronous protocol)
  - Sender assumes that the receiver can store the message

  - 不需要相互确认message envelop。直接发送
  -



- Rendezvous (a synchronous protocol)
  - the sender makes no assumption about the receiver
  - Requires handshaking between sender and receiver:
    - handshaking between message envelope
    - Send data
  -



Often: *eager protocol* for small message; *RendezVous protocol* for larger message

Eager protocol

- Sender assumes that the receiver can store the message.
- It is the responsibility of the receiver to buffer the message upon its arrival, especially if the receive operation has not been posted.
- This assumption may be based upon the implementation's guarantee of a certain amount of available buffer space on the receive process.
- Generally used for smaller message sizes (up to Kbytes). Message size may be limited by the number of MPI tasks as well.

Advantages: Reduced latency; Simpler code.

Disadvantages: Not scalable: Significant buffering may be required; Memory exhaustion; Consumes CPU (push data directly into the NIC)

RendezVous protocol

- The sender makes no assumption about the receiver.
- Requires handshaking between sender and receiver:
  1. Sender sends message envelope to receiver.
  2. Envelope received and stored by receiver.
  3. When buffer space is available, receiver replies to sender that requested data can be sent.

4. Sender receives reply from receiver and then sends data.
5. Destination process receives data.

Advantages: Scalable compared to eager protocol; Better memory management; No CPU usage (NIC reads from memory)

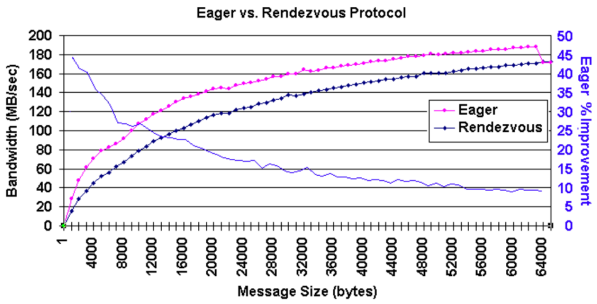
Disadvantages: Inherent synchronization (more latency); More difficult to implement

Summary

Internal MPI implementations often use: Eager protocol for small messages. RendezVous protocol for larger messages.

- The threshold can be modified by the user.
- Always try to post receive calls as soon as possible.

Measurements with IBM MPI code



2.5 Non-Blocking point-to-point communication

Review: Send an integer Our Motivation

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int tag = 42;
    int message;

    if (rank == 0) {
        message = 7;
        MPI_Send(&message, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        MPI_Recv(&message, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Rank %d received the following integer: %d\n", rank, message);
    }

    MPI_Finalize();
    return 0;
}
```

send one integer to rank 1
Might Idle until it is received

Blocking recv: Idle until the message is received

Could do useful work here instead of waiting

- Blocking communication may waste computational resources Can we improve performance?

2.5.1 MPI_Isend, MPI_Irecv

Blocking versus Non-blocking point-to-point:

- Blocking point-to-point:
  - The data buffer can be used directly after the function returns

  - The communication is not necessarily completed (e.g. MPI_Bsend, eager protocol) However, communication will start, and executing these operations will take some time

- Non-Blocking point-to-point:
  - The call returns immediately, communication might start after.
  - The data buffer can not be used right after the function returns.
  - creates a `MPI_Request` object
  - The completion of the operation can be checked with `MPI_Test`.
  - Can wait for completion with `MPI_Wait`. [see the request management in the follow]

  Note: a non-blocking send can match a blocking receive (and vice-versa)

Point-to-point: `MPI_Isend` and `MPI_Irecv`

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);
```

- The matching rule within a communicator is the same as for blocking versions: source, dest and tag must match
- The non-blocking calls create a `MPI_Request` object that helps keeping track and manage

  the action progress. The action is completed after the request completed.

The buffer must not be modified until the request is completed (waited for).

Non-Blocking Send modes:

类似于blocking的communication mode

- `int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
  - Returns immediately
  - Must be called only after a matching receive is posted
  - Can reuse the buffer once the request has completed
- `int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
  - Returns immediately
  - The request is completed once a matching receive started
  - Can reuse the buffer once the request has completed
- `int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`

- Returns immediately
- The request is completed once the buffer has been copied to the MPI internal buffer
- See `MPI_Buffer_attach` to allocate buffer for MPI

2.5.2 Request management

Request management:

MPI provides an API (application program interface) to test and wait for a request object:

- `int MPI_Wait(MPI_Request *request, MPI_Status *status);` Returns only after the request has completed and mark the request to "inactive".
- `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);` Returns flag = 1 if the operation described by request has completed, 0 otherwise. If the operation completed, set the request to "inactive".

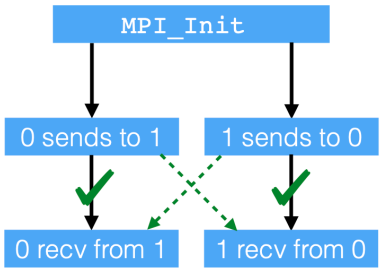
Request management: Wait for multiple requests:

- `int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status *array_of_statuses);`
  - Returns only after all the requests have completed and mark them to "inactive"
- `int MPI_Waitany(int count, MPI_Request array_of_requests[ ], int *index, MPI_Status *status);`
  - Returns only after any active request has completed, and mark the request to "inactive".
  - Output `index` is the corresponding index in the input array
- `int MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, int array_of_indices[], MPI_Status array_of_statuses[]);`
  - Returns only after at least one active request has completed. Returns the array or corresponding indices of size outcount. Mark the request(s) to "inactive".

Non-Blocking point-to-point:

example which solves the problem of deadlock in previous chapter:

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int tag1 = 42, tag2 = 43;
    int sendMessage, recvMessage;
    MPI_Request sendReq;
    MPI_Status status;

    if (rank == 0) {
        sendMessage = 7;
        MPI_Isend(&sendMessage, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD, &sendReq);
        MPI_Recv(&recvMessage, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else if (rank == 1){
        sendMessage = 14;
        MPI_Isend(&sendMessage, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, &sendReq);
        MPI_Recv(&recvMessage, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Wait(&sendReq, &status);

    printf("Rank %d received the following integer: %d\n", rank, recvMessage);

    MPI_Finalize();
    return 0;
}
```



Non blocking send returns immediately

Send is completed for sure
The buffer can be used now

- Question: at `MPI_Wait()`, 是通讯单元等待还是整个程序停下来等待

Watch out for memory management:

Example:

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int tag = 42;
    MPI_Request sendReqs[2];
    MPI_Status statuses[2];

    if (rank == 0) {
        int message = 7;
        MPI_Isend(&message, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &sendReqs[0]);

        message = 14;
        MPI_Isend(&message, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &sendReqs[1]);

        MPI_Waitall(2, sendReqs, statuses);
    }
    else if (rank == 1) {
        int r1, r2;
        MPI_Recv(&r1, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&r2, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Rank %d received the following integers: %d %d\n", rank, r1, r2);
    }

    MPI_Finalize();
    return 0;
}
```

This code has undefined behaviour: can you find the bug?

Send is not necessarily completed, can not reuse the buffer!
The first send might send 14 instead of 7

Possible fixes:

- Duplicate the memory buffer (define two variables)
- `MPI_Wait` after first send

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int tag = 42;
    MPI_Request sendReqs[2];
    MPI_Status statuses[2];

    if (rank == 0) {
        int message1 = 7, message2 = 14;
        MPI_Isend(&message1, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &sendReqs[0]);
        MPI_Isend(&message2, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &sendReqs[1]);

        MPI_Waitall(2, sendReqs, statuses);
    }
    else if (rank == 1) {
        int r1, r2;
        MPI_Recv(&r1, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&r2, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Rank %d received the following integers: %d %d\n", rank, r1, r2);
    }

    MPI_Finalize();
    return 0;
}
```

Possible fixes:

- Duplicate the memory buffer
- `MPI_Wait` after first send

```
> mpiexec -n 2 ./isend_bug_fixed
Rank 1 received the following integers: 7 14
```

Request management: Test multiple requests:

Similiar with `MPI_Wait()` mutiple requests

- `int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_of_statuses[]);`
  - Test if all the requests have completed and mark them to “inactive”.
- `int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status);`
  - Test if any active request has completed, and mark the request to “inactive”. Output index is the corresponding index in the input array
- `int MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount, **int** array_of_indices[], MPI_Status array_of_statuses[]);`
  - Test if at least one active request has completed. Returns the array or corresponding indices of size outcount. Mark the request(s) to “inactive”.

Example: Manual reduction

```c
double sumNaive(double val, int root, MPI_Comm comm)
{
    int rank, size, tag = 42;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    double sum = 0;

    if (rank == root) {
        double other;
        sum = val;
        for (int i = 0; i < size; ++i) {
            if (i == root) continue;
            MPI_Recv(&other, 1, MPI_DOUBLE, i, tag, comm, MPI_STATUS_IGNORE);
            sum += other;
        }
    }
    else
        MPI_Ssend(&val, 1, MPI_DOUBLE, root, tag, comm);

    return sum;
}
```

Apart from the linear number of steps: What is also bad about this code in terms of performance?

root can NOT receive a message rom rank i before receiving the message from rank i-1 because of the blocking receive

Example: manual reduction: non blocking version

```cpp
double sumNaiveNonBlocking(double val, int root, MPI_Comm comm)
{
    int rank, size, tag = 42;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    double sum = 0;

    if (rank == root) {
        std::vector<MPI_Request> requests(size, MPI_REQUEST_NULL);
        std::vector<MPI_Status> statuses(size);
        std::vector<double> others(size, 0.0);
        others[root] = val;

        for (int i = 0; i < size; ++i) {
            if (i == root) continue;
            MPI_Irecv(&others[i], 1, MPI_DOUBLE, i, tag, comm, &requests[i]);
        }

        MPI_Waitall(requests.size(), requests.data(), statuses.data());

        for (auto other : others) sum += other;
    }
    else
        MPI_Ssend(&val, 1, MPI_DOUBLE, root, tag, comm);

    return sum;
}
```

Now the non root ranks can return before everyone is done

root has to wait for everyone before doing the sum, can we do better?

Example: manual reduction: non blocking version 2

```cpp
double sumNaiveNonBlocking2(double val, int root, MPI_Comm comm)
{
    int rank, size, tag = 42;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    double sum = 0;

    if (rank == root) {
        std::vector<MPI_Request> requests(size, MPI_REQUEST_NULL);
        std::vector<double> others(size, 0.0);
        sum = val;

        for (int i = 0; i < size; ++i) {
            if (i == root) continue;
            MPI_Irecv(&others[i], 1, MPI_DOUBLE, i, tag, comm, &requests[i]);
        }

        for (int i = 0; i < size - 1; ++i) {
            int index;
            MPI_Waitany(requests.size(), requests.data(), &index, MPI_STATUS_IGNORE);
            sum += others[index];
        }
    }
    else
        MPI_Send(&val, 1, MPI_DOUBLE, root, tag, comm);

    return sum;
}
```

Now the root performs the sum as soon as it received ANY message

Example: MPI_Test

```cpp
double message;

if (rank == 0) {
    message = 11.00;
    MPI_Send(&message, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
}
else if (rank == 1) {
    MPI_Request request;
    MPI_Irecv(&message, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &request);
    int messageArrived = 0;

    while (!messageArrived) {
        doUsefulWork();
        MPI_Test(&request, &messageArrived, MPI_STATUS_IGNORE);
    }

    printf("The break is at %g\n", message);
}
```

Do useful work if message did not arrive

Here the message is guaranteed to be received

Use `MPI_Test` when you can do useful work instead of waiting

2.5.3 Example: diffusion in 1D with Finite differences

Application: the 1D diffusion equation

Problem Description:

Diffusion Equation: $\frac{\partial f}{\partial t} = D\Delta f$

Discretization with finite differences in space (1D):

$$\frac{\partial f(x_i)}{\partial t} \approx \frac{D}{h^2}\left[f(x_{i-1}) - 2f(x_i) + f(x_{i+1})\right]$$

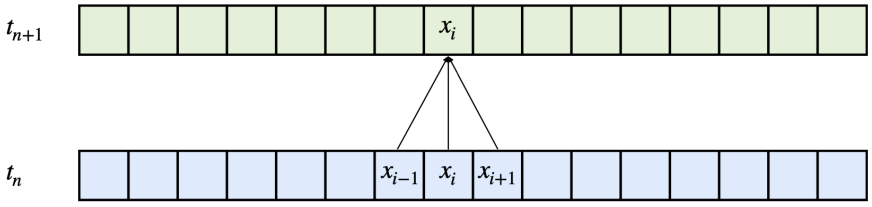
Discretization in time, forward Euler:

$$f(x_i, t_{n+1}) \approx f(x_i, t_n) + \frac{Ddt}{h^2}\left[f(x_{i-1}, t_n) - 2f(x_i, t_n) + f(x_{i+1}, t_n)\right]$$

We assume periodic boundary conditions for simplicity

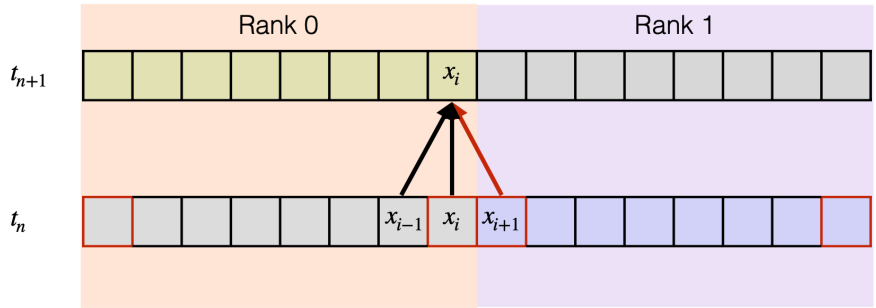
$$f(x_i, t_{n+1}) \approx f(x_i, t_n) + \frac{Ddt}{h^2}\left[f(x_{i-1}, t_n) - 2f(x_i, t_n) + f(x_{i+1}, t_n)\right]$$

It is a stencil computation:



⇓ 1D diffusion equation on distributed memory

Each rank needs to store only part of the data:



Need to communicate ghost cells at every time step

- We have ghost cells, but ghost cells are only used to update the boundary element.

⇓ Realize it in code:

1D diffusion equation with blocking point-to-point

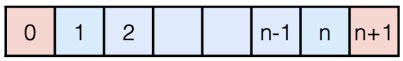
```cpp
std::vector<double> rho(n+2), rhoPrev(n+2);

for (int step = 0; step < nsteps; ++step) {

    if (rank % 2 == 0) {
        MPI_Send (&rhoPrev[n]  , 1, MPI_DOUBLE, next, tag, MPI_COMM_WORLD);
        MPI_Recv (&rhoPrev[0]  , 1, MPI_DOUBLE, prev, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send (&rhoPrev[1]  , 1, MPI_DOUBLE, prev, tag, MPI_COMM_WORLD);
        MPI_Recv (&rhoPrev[n+1], 1, MPI_DOUBLE, next, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else {
        MPI_Recv (&rhoPrev[0]  , 1, MPI_DOUBLE, prev, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send (&rhoPrev[n]  , 1, MPI_DOUBLE, next, tag, MPI_COMM_WORLD);
        MPI_Recv (&rhoPrev[n+1], 1, MPI_DOUBLE, next, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send (&rhoPrev[1]  , 1, MPI_DOUBLE, prev, tag, MPI_COMM_WORLD);
    }

    for (int i = 1; i < n + 1; ++i)
        rho[i] = rhoPrev[i] + alpha * (rhoPrev[i-1] - 2 * rhoPrev[i] + rhoPrev[i+1]);
    std::swap(rho, rhoPrev);
}
```

n local cells + 2 ghost cells

exchange ghost cells

compute next iteration

Overhead because of communication



Interior cells are independent from ghost cells, we could overlap communication and computation

- Ghost cells are only used for communication of interior cells. No overlap and computation for ghost cells.
- The idea of this code is: firstly do the ghost cells communication; secondly do the computation of interior cells.

1D diffusion equation: non blocking point-to-point

```cpp
std::vector<double> rho(n+2), rhoPrev(n+2);

for (int step = 0; step < nsteps; ++step) {

    MPI_Irecv(&rhoPrev[0]  , 1, MPI_DOUBLE, prev, tag1, MPI_COMM_WORLD, &req[0]);
    MPI_Irecv(&rhoPrev[n+1], 1, MPI_DOUBLE, next, tag2, MPI_COMM_WORLD, &req[1]);

    MPI_Isend(&rhoPrev[1], 1, MPI_DOUBLE, prev, tag2, MPI_COMM_WORLD, &req[2]);
    MPI_Isend(&rhoPrev[n], 1, MPI_DOUBLE, next, tag1, MPI_COMM_WORLD, &req[3]);

    auto applyStencil = [&](int i) {
        rho[i] = rhoPrev[i] + alpha * (rhoPrev[i-1] - 2 * rhoPrev[i] + rhoPrev[i+1]);
    };

    for (int i = 2; i < n; ++i)
        applyStencil(i);

    MPI_Waitall(4, req, stats);

    applyStencil(1);
    applyStencil(n);

    std::swap(rho, rhoPrev);
}
```

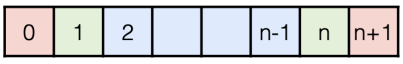
launch communication

compute next iteration for interior

wait for ghost cells and compute next iteration for boundaries

Overlap of computation and communication

More on this topic in HPCSE II!



2.6 Non-Blocking collective communication

Blocking versus Non-blocking collectives:

Blocking collective:

- The data buffer can be used after the function returns
- The process has completed his participation to the collective operation after the function returns
- The collective operation is not necessarily completed for everyone (partial barrier)

Non-Blocking collective:

- The data buffer can not be used right after the function returns.
- The call returns directly but creates a `MPI_Request` object.

- `MPI_Wait` and `MPI_Test` can be used to wait or check the completion

A non-blocking collective can **NOT** match a blocking collective! [this is different from point-to-point].

Non-blocking collective prototypes:

```
int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root,
                MPI_Comm comm, MPI_Request *request);

int MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                   MPI_Request *request);

int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm, MPI_Request *request)
...
```

Same names as blocking version with a "I" in front of the operation

Extra output argument of type `MPI_Request` at the end

2.6.1 Examples

Example: non blocking collective

```
1 double val = (double) (rank * rank);
2 double sumVals = 0;
3
4 MPI_Allreduce(&val, &sumVals, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD); <- This takes some time
5
6 doUnrelatedWork();                                                      <- Independent from reduction
7
8 if (rank == 0)
9     printf("total sum is %g\n", sumVals);
```

- `doUnrelatedWork()` can overlap with `MPI_Allreduce`
- As long as the code is independent from reduction.

```
 1 double val = (double) (rank * rank);
 2 double sumVals = 0;
 3
 4 MPI_Request request;
 5 MPI_Iallreduce(&val, &sumVals, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD, &request);
 6
 7 doUnrelatedWork();
 8
 9 MPI_Wait(&request, MPI_STATUS_IGNORE);   <- Need to wait for the result
10
11 if (rank == 0)
12     printf("total sum is %g\n", sumVals);
```

- `doUnrelatedWork()` overlaps with the reduction

Application: 1D diffusion equation statistics

Suppose we want to report the maximum density at each iteration. Assume we need it only at the next iteration.

```
1  for (int step = 0; step < nsteps; ++step) {
2
3      /* exchange ghost cells */
4      ...
5
6      localMax = getMax(rhoPrev.size(), rhoPrev.data());
7      MPI_Reduce(&localMax, &globalMax, 1, MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD);
8
9      /* compute next iteration in bulk, receive ghost cells, compute boundaries */
10     ...
11
12     if (rank == root) printf("%g\n", globalMax);
13 }
```

The reduction and the update are independent: -> We can overlap both of them:

```cpp
for (int step = 0; step < nsteps; ++step) {

    /* exchange ghost cells */
    ...

    localMax = getMax(rhoPrev.size(), rhoPrev.data());
    MPI_Ireduce(&localMax, &globalMax, 1, MPI_DOUBLE, MPI_MAX, root, MPI_COMM_WORLD, &reduceReq);

    /* compute next iteration in bulk, receive ghost cells, compute boundaries */
    ...

    MPI_Wait(&reduceReq, MPI_STATUS_IGNORE);
    if (rank == root) printf("%g\n", globalMax);
}
```

-> The reduction overhead is hidden.

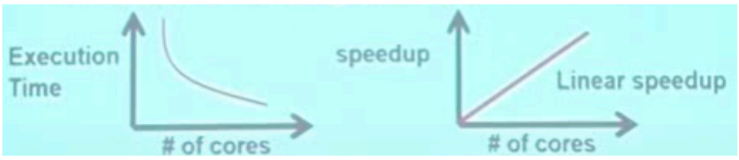
2.7 Performance metrics

Review motivation: How to measure efficiency? 我们提出了以下两种度量方式

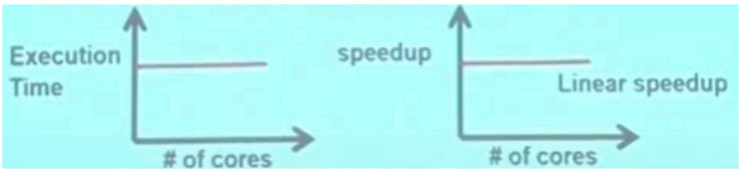
- Strong Scaling - Fix problem size N and increase number of processors P.
- Weak Scaling - Fix problem size per processor.

三、strong scaling & weak scaling

1.strong scaling: 使问题规模保持不变，增加处理器数量，用于找到解该问题最合适的处理器数量。即所用时间尽可能短而又不产生太大的开销。绘制如下图形：



2.weak scaling: 让问题规模（计算量）随处理器数量增加而增加。理想情况：



strong scaling的结果较难达到，因为随着处理器数量的增加通信开销成比例上升；而weak scaling的结果较容易达到。

2.7.1 Strong and weak scaling efficiencies

Performance metrics: strong scaling

"How much do we gain by adding more nodes for the same problem?"

Speedup: $S(n) = T(1)/T(n)$

Strong scaling efficiency: $\eta_S(n) = S(n)/n$

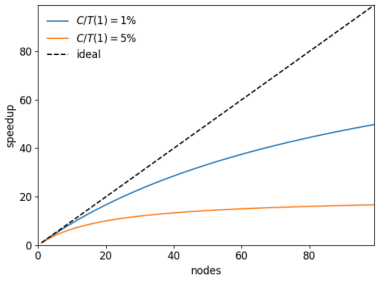
Amdahl's law: sequential part has disastrous effects on the strong scaling

The same applies with communication overhead

For n nodes:

$$T(n) = \frac{T(1)}{n} + C$$

$$S(n) = \frac{T(1)}{\frac{T(1)}{n} + C} = \frac{n}{1 + n\frac{C}{T(1)}}$$



Performance metrics: weak scaling

Increase the number of nodes **and** the problem size, such that the **work per node is constant**

Example:
"The amount of work is linear with problem size" (e.g. diffusion in 1D, forward FD)

⇒ multiplying the number of nodes by 2 requires to multiply the problem size by 2

Example:
"the amount of work is quadratic with problem size" (e.g. naive N body interactions)

⇒ multiplying the number of nodes by 2 requires to multiply the problem size by 2

Weak scaling efficiency: $\eta_W(n) = T(1)/T(n)$

理想的efficiency应该是1。因为在这个过程中，我们解决的问题尺度也成倍增加。

Weak scaling: communication overhead

MPI might introduce communication overhead This is reflected in the weak scaling efficiency

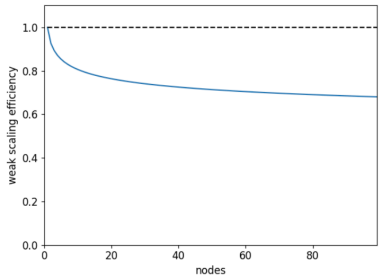
Example: all to all communications

```
1 for (int i = 0; i < nsteps; ++i)
2 {
3     val = doWork(val);
4     MPI_Allreduce(&val, &sumVals, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
5 }
```

constant time t_W

time $l + \beta \log n$

$$\eta_W(n) = \frac{T(1)}{T(n)} = \frac{t_W}{t_W + l + \beta \log n}$$



Weak scaling: improving with non-blocking calls

Example: all to all communications overlapped with computation

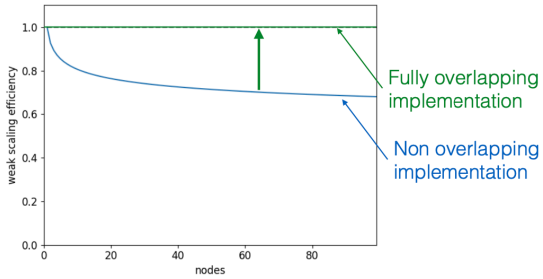
```
1  MPI_Request request = MPI_REQUEST_NULL;
2  double valBuf;
3
4  for (int i = 0; i < nsteps; ++i)
5  {
6      val = doWork(val);
7      valBuf = val;
8      MPI_Wait(&request, MPI_STATUS_IGNORE);
9      MPI_Iallreduce(&valBuf, &sumVals, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD, &request);
10 }
11 MPI_Wait(&request, MPI_STATUS_IGNORE);
```

constant time t_W

time $l + \beta \log n$

$$\eta_W(n) = \frac{T(1)}{T(n)} = \frac{t_W}{\max(t_W, l + \beta \log n)}$$

Ideal efficiency if $l + \beta \log n \leq t_W$



Performance metrics: Timing MPI programs

The API provides a timing function: `double MPI_Wtime();` You can profile part of your program as follows:

```
1 MPI_Barrier(MPI_COMM_WORLD);
2
3 double tstart = MPI_Wtime();
4
5 functionToProfile(MPI_COMM_WORLD);
6
7 double elapsedTime = MPI_Wtime() - tstart;
```

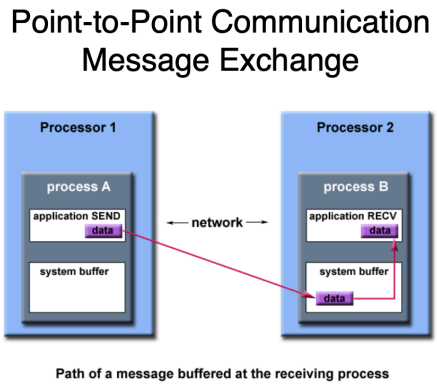
Wait for everyone to start

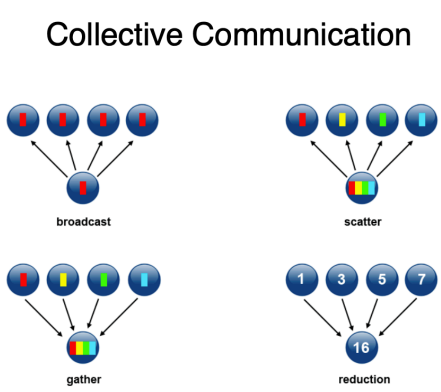
Depending on the problem, a barrier here is also necessary.

This gives a timing per rank. You can reduce it to get the max, mean, load imbalance...

2.8 Communication Topologies and Groups

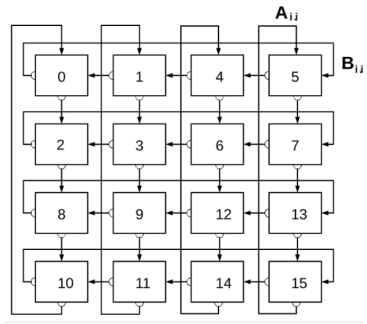
MPI Review

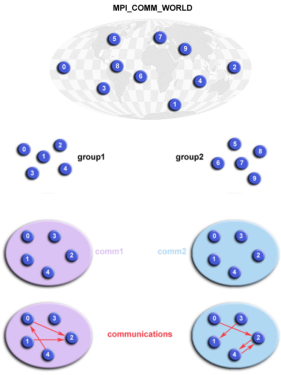




MPI Communicators:

So far we have always handled communication among all ranks as part of the same communicator: `MPI_COMM_WORLD`. Suppose, however, we need to communicate among subsets (rank or column-wise):

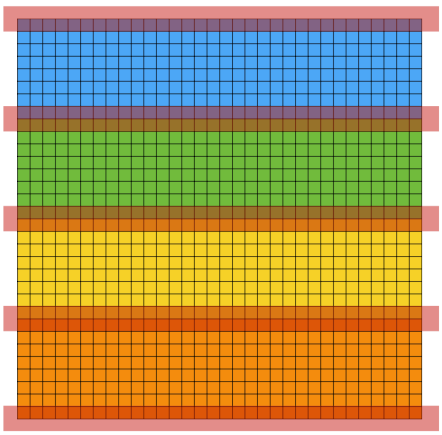


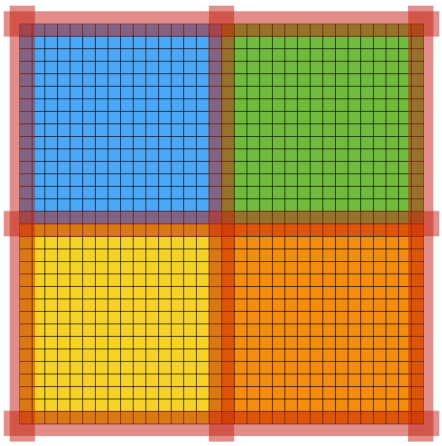


- We want to split the ranks into groups and build a new communicator for each group.
- We can then do communication operations within a group instead of within all rank.

Example: Domain decomposition

2D diffusion equation: how to decompose the domain?

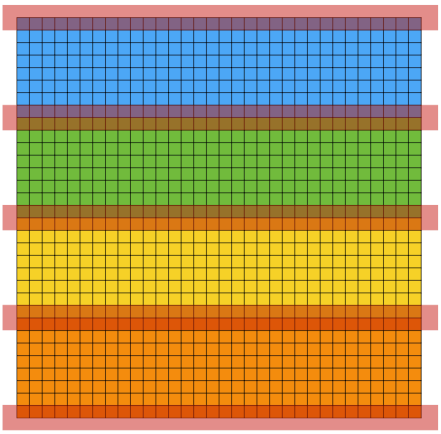




We want to minimize communication (C) - computation (W) ratio $\rho = C/W$

⇓ Domain decomposition, Row-wise approach

N x N grid, P ranks



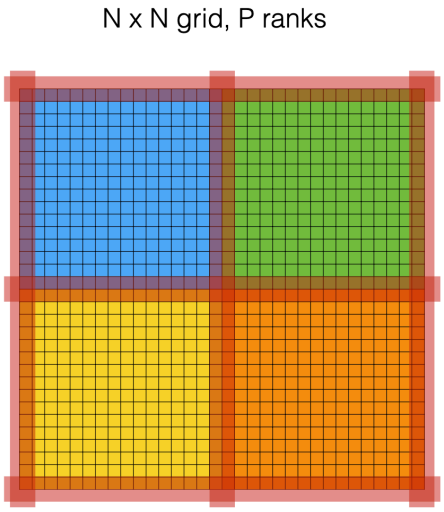
For **each** rank:

Communication $C = 2N$

Computation $W = N\frac{N}{P}$

Ratio $\rho = \frac{C}{W} = \frac{2P}{N}$

⇓ Domain decomposition, grid approach



For **each** rank:

Communication $C = 4\frac{N}{\sqrt{P}}$

Computation $W = \frac{N^2}{P}$

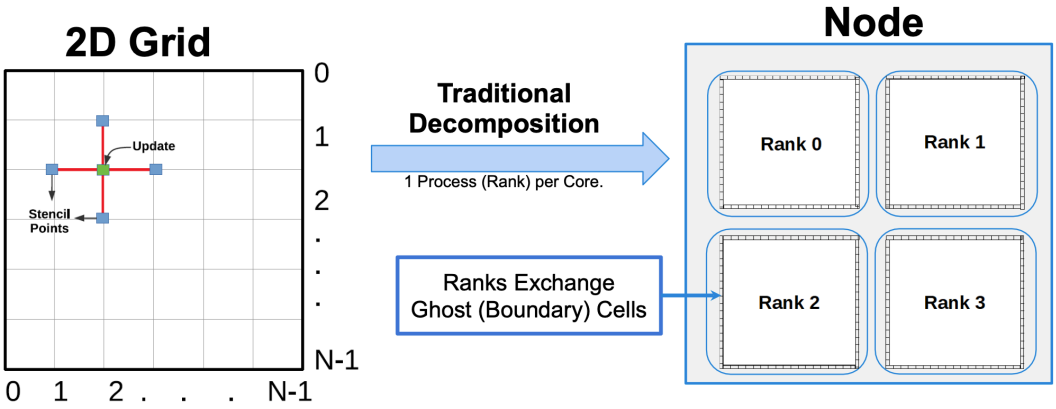
Ratio $\rho = \frac{C}{W} = \frac{4\sqrt{P}}{N}$

-> Better than "row-wise" approach

$$\frac{4\sqrt{P}}{N} \leq \frac{2P}{N} \quad for \quad P \geq 4$$

Example: Structured Grids

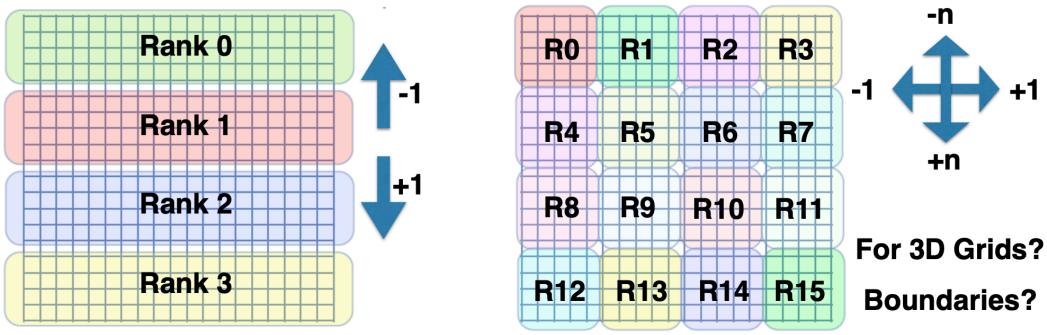
Structured Grid Stencil Solver: Iteratively approaches a solution.



Connectivity of MPI Ranks:

Determining the ranks of my neighbors:

- Easy in a one-dimensional layout
- Harder in two and more dimensions
- Even harder on irregular meshes



MPI provides an easy way to find neighbors in cartesian grids: *Topologies*

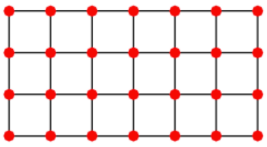
MPI Topologies:

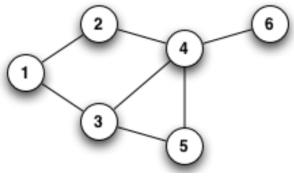
A connectivity graph-building utility for MPI Ranks.

- A (virtual) topology describes the “connectivity” of MPI processes in a communicator.
- There may be no relation between the physical network and the process topology.

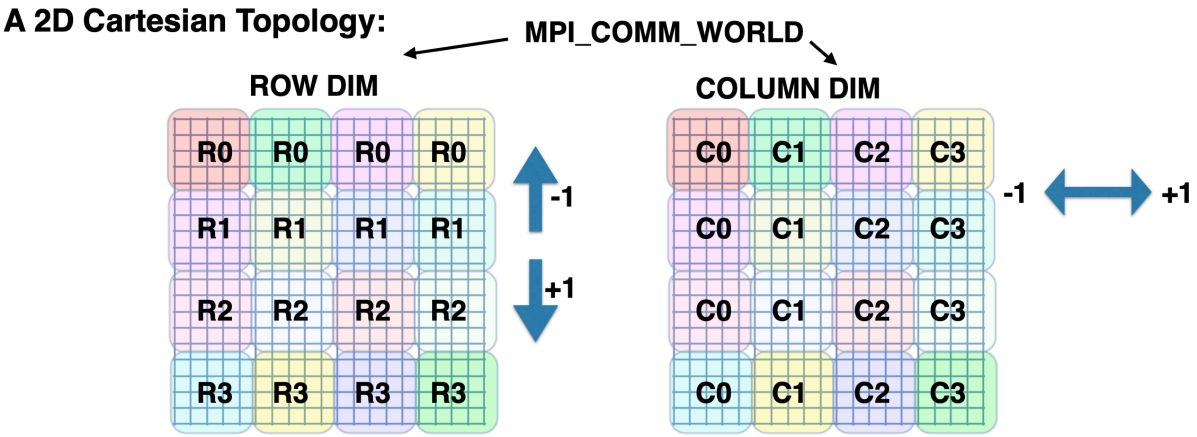
Two Types:

- **Cartesian topology:** each process is “connected” to its neighbors in a virtual grid.
- **Graph Topology:** any arbitrary connection graph.

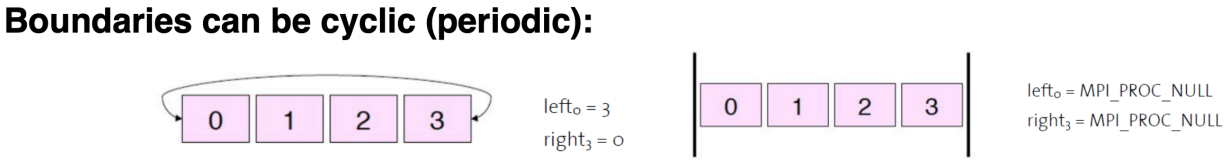




Cartesian Topology



Two different boundary conditions: periodic and dirichit boundary condition



Creating an MPI Cartesian Topology:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder,
MPI_Comm *comm_cart);
```

- `comm_old` : The source communicator.
- `ndims` : Number of dimensions
- `dims` : Integer array specifying the number of processes in each dimension
- `periods` : Integer aray of boolean values indicating whether the grid is periodic in that dimension
- `reorder` : Boolean flag indicating whether the processes may be reordered

- `comm_cart` : A new cartesian grid communicator.

⇓ To help creating a grid with a fair node distribution in each dimension, use:

```
int MPI_Dims_create(int nnodes, int ndims, int *dims);
```

dims before call	function call	dims on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- First, we will use MPI dimension to creat the `dims` , then substitute it to the `MPI_Cart_create`

⇓ Creating a 3D-Grid Topology

Example: 3D Cartesian Grid

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

 int size;
 MPI_Comm_size(&size);

 int nums[3] = {0,0,0};
 int periodic[3] = {false, false, false};

 MPI_Dims_create(size, 3, nums); // split the nodes automatically
 printf("Grid: (%d, %d, %d)\n", nums[0], nums[1], nums[2]);

 MPI_Comm cart_comm; // now everyone creates a a cartesian topology
 MPI_Cart_create(MPI_COMM_WORLD, 3, nums, periodic, true, &cart_comm);
 MPI_Comm_free(&cart_comm);
 MPI_Finalize();
}
```

- `MPI_Comm_free(&cart_comm);` :Marks the communicator object for deallocation ??what is the meaning of this line?

Finding Neighbors:

Let use MPI topologies to find the ranks nearest neighbors.

```
int MPI_Cart_shift(MPI_Comm comm, int dir, int disp, int *source, int *dest);
//Rationale: Gives the ranks shifted in the dimension given by direction by a certain
displacement, where the sign of displacement indicates the direction.
int left, right, bottom, top, front, back, newrank;
MPI_Comm_rank(cart_comm,&newrank);
MPI_Cart_shift(cart_comm, 0, 1, &left, &right);
MPI_Cart_shift(cart_comm, 1, 1, &bottom, &top);
MPI_Cart_shift(cart_comm, 2, 1, &front, &back);
printf("MyRank: %d -> NewRank: %d\n", rank, newrank);
printf("Left:  %d, Right:  %d\n", left,  right);
printf("Top:   %d, Bottom: %d\n", top,   bottom);
printf("Front: %d, Back:   %d\n", front, back);
```

- First, tell the machine how many ranks you want to use.
- Then, generate the cartesian topology for rank.
- At each rank, we can use `MPI_Cart_shift` to know its neigubors' rank number.
- Communicate information, and do the specific task.

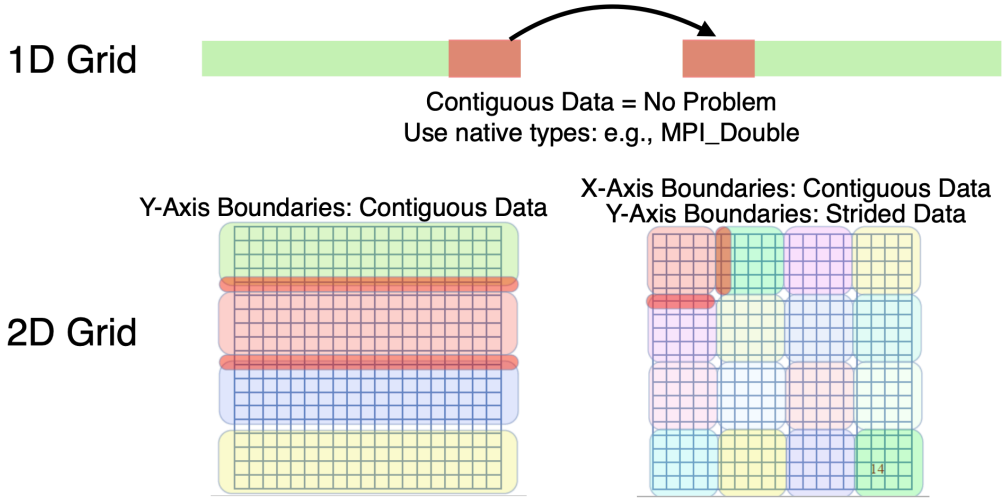
Useful Cartesian Functions:

- `int MPI_Cartdim_get(MPI_Comm comm, int *ndims)`
  - Rationale: Get number of dimensions.
- `int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *prds, int *coords)`
  - Rationale: Retrieves information about the cartesian topology associated with a communicator. The arrays are allocated with maxdims dimensions. dims and prds are the numbers used when creating the topology. coords are the dimensions of the current rank.
- `int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)`
  - Rationale: Get the rank of a given coordinate.
- `int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)`
  - Rationale: Get the coordinates of a given rank.

2.9 MPI Vector Datatype

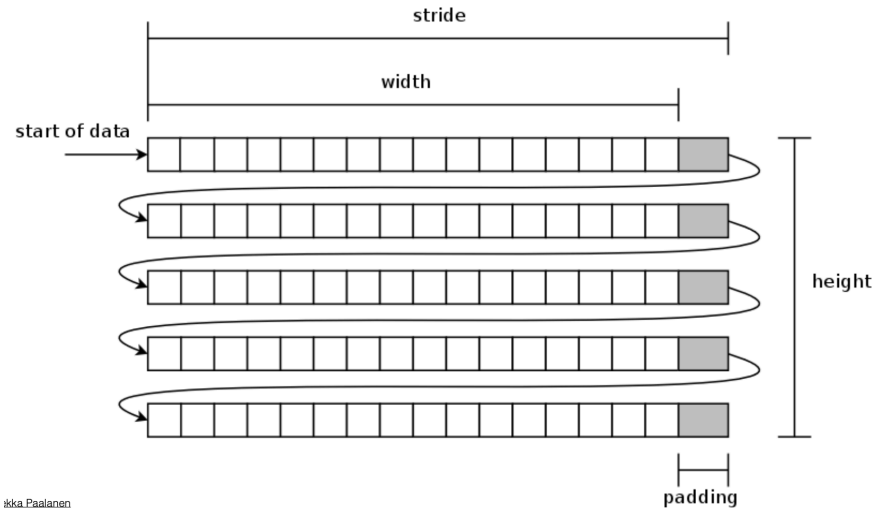
Grids: How do we exchange boundaries?

Contiguous vs. Strides Data: 主要问题体现在boundary数据在内存中并不是连续存放的。这样你在指定指针的时候基本上很难去进行数据访问交换。



Non-Contiguous Data Strides

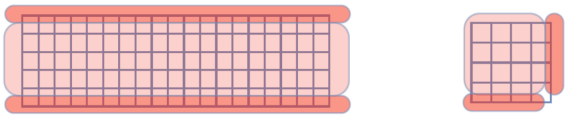
A common problem for communication in multi-dimensional grids. (Worse for 3D!)



- 对照上面那个矩阵，知道第一行和第二行的地址之间不是连续的，有很多的padding。

Strided/Vector Datatypes

Boundary elements in a multidimensional array (or matrix) can be described as strided vectors



```
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- *Rationale*: Build an MPI datatype for a vector array of blocklength contiguous entries that are spaced at a given stride. This specifies the distance between blocks

```
int MPI_Type_create_hvector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- *Rationale*: MPI_Type_vector but now the stride is given in bytes

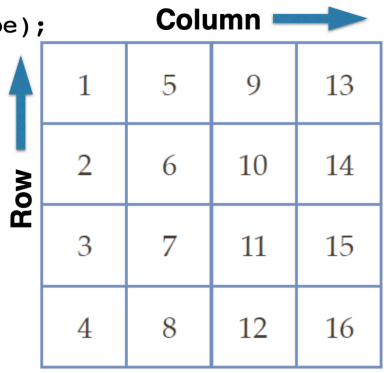
```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- `Rationale` : build an MPI datatype for a contiguous array

Example: Exchange Boundaries of a Square Domain

Let's create datatypes for the elements of a 4x4 matrix:

```
int NCols = 4;
int NRows = 4;
double* a[NCols * NRows];

MPI_Datatype rowType, colType;
MPI_Type_contiguous(NCols, MPI_DOUBLE, &colType);
MPI_Type_vector (Nrows, 1, NCols, MPI_DOUBLE, &rowType);
MPI_Type_commit(&rowType);
MPI_Type_commit(&colType);

// Perform Work and Communication
do_some_work();
MPI_Sendrecv(NCols, colType, ...);
MPI_Sendrecv(NRows, rowType, ...);

MPI_Type_free(&row);
MPI_Type_free(&col);
```



Multidimensional Arrays

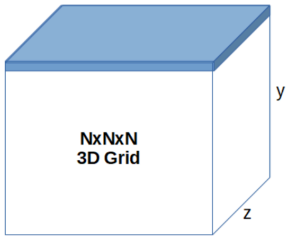
For multidimensional problems, `MPI_Type_create_subarray` is a more general solution.

```
int MPI_Type_create_subarray(int ndims, int sizes[], int subsizes[], int starts[], int order,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

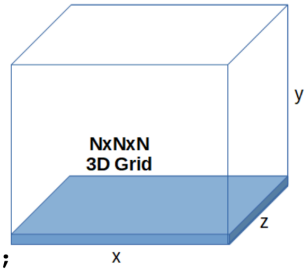
- *Rationale*: Builds an MPI datatype for a subarray of a larger array:
  - ndims: number of dimensions
  - sizes: extent of the full array in each dimension
  - subsizes: extent of the subarray in each dimension
  - starts: starting index of the subarray
  - order: array storage order, can be either of MPI_ORDER_C or MPI_ORDER_FORTRAN

MPI_Type_create_subarray(sizes, subsizes, starts, ...)

```
int sizes    = { N, N, N }
int subsizes = { N, 1, N }
int starts   = { 0, 0, 0 }
```



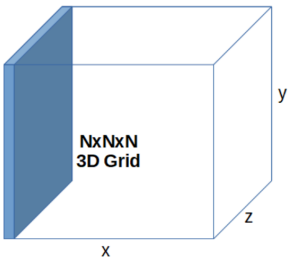

```
int sizes    = { N, N,   N }
int subsizes = { N, 1,   N }
int starts   = { 0, N-1, 0 }
```



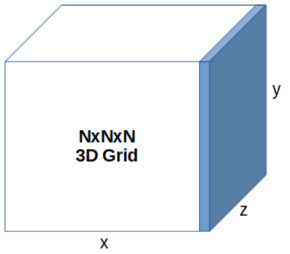

```
// Perform Work and Communication
MPI_Sendrecv(1, topFaceType,    topNeighbor, ...);
MPI_Sendrecv(1, bottomFaceType, bottomNeighbor, ...);
```

MPI_Type_create_subarray(sizes, subsizes, starts)

```
int sizes    = { N, N, N }
int subsizes = { 1, N, N }
int starts   = { 0, 0, 0 }
```

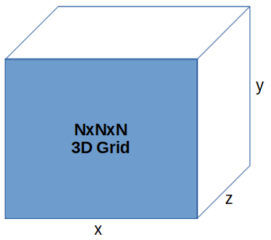



```
int sizes    = { N,   N, N }
int subsizes = { 1,   N, N }
int starts   = { N-1, 0, 0 }
```

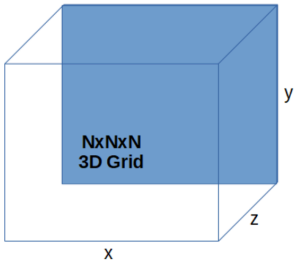


MPI_Type_create_subarray(sizes, subsizes, starts)

```
int sizes    = { N, N, N }
int subsizes = { N, N, 1 }
int starts   = { 0, 0, 0 }
```

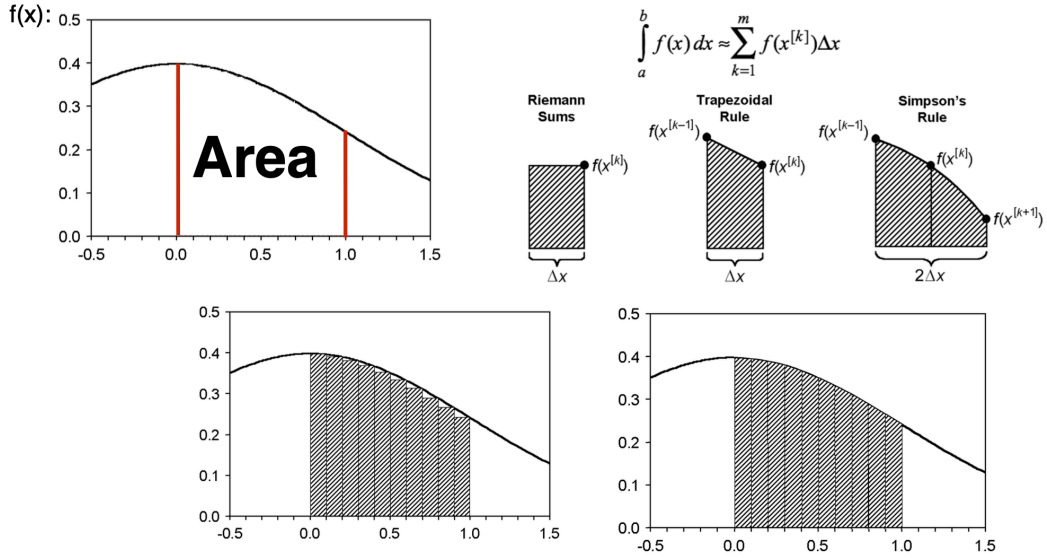



```
int sizes    = { N, N, N   }
int subsizes = { N, N, 1   }
int starts   = { 0, 0, N-1 }
```

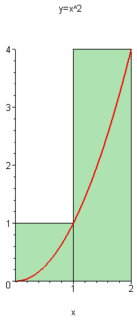


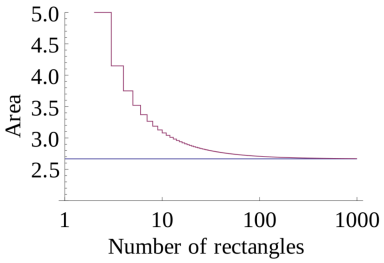
2.10 MPI Structure Datatype

Example: Numerical Integration



Riemann Sums and Riemann Sums code (Serial)






```c
int main(int argc, char** argv) {
 double a = read(); // lower bound of integration
 double b = read(); // upper bound of integration
 int    n = read(); // number of subintervals

 double area = 0.0;
 double dx = (b-a)/n;

 for (int i = 0; i < n; i++) // Integration
 {
  double x = a + i*dx;
  area += f(x)*dx;
 }

 printf("Area: %f\n", area);
}
```

Distributed Riemann Sums Code (MPI)

```c
int main(int argc, char** argv) {
 MPI_Init(&argc,&argv);
 int rankCount, myRank;

 MPI_Comm_size(MPI_COMM_WORLD,&rankCount);
 MPI_Comm_rank(MPI_COMM_WORLD,&myRank);

 double a, b;
  int n;

  if (myRank == 0)
  {
   a = read(); // lower bound of integration
   b = read(); // upper bound of integration
   n = read(); // number of subintervals
  }

  // Need to distribute these values to other ranks
```

```c
int myStart = myRank     * (n/rankCount);
int myEnd   = (myRank+1) * (n/rankCount);

double myArea = 0.0;
double dx = (b-a)/n;  // Delta X = b-a / number of intervals

for (int i = start; i < end; i++) // Integration
{
 double x = a + i*dx;
 myArea += f(x)*dx;
}

double TotalArea = 0.0;
 MPI_Reduce(&area, &TotalArea, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (myRank == 0) printf("Area: %f\n", TotalArea);
}
```

Distributing Parameters

1. Broadcast

```
if (myRank == 0)
 {
  a = read(); // lower bound of integration
  b = read(); // upper bound of integration
  n = read(); // number of subintervals
 }

// Use broadcasts to distribute parameters

 MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
 MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
 MPI_Bcast(&n, 1, MPI_INT,    0, MPI_COMM_WORLD);
```

(a) Inefficient: uses 3 broadcasts. Latency-intensive.

2. Byte-wise Comm

```
// Defining a struct for the parameters
struct parms {
 double a; // lower bound of integration
 double b; // upper bound of integration
 int n;    // number of subintervals
};

parms p;
if (myRank == 0)
 {
  p.a = read(); // lower bound of integration
  p.b = read(); // upper bound of integration
  p.n = read(); // number of subintervals
 }

// Broadcast the parameters as array of bytes
// Not portable on heterogeneous machines
MPI_Bcast(&p, sizeof(parms), MPI_BYTE, 0, MPI_COMM_WORLD);
```

Not Portable: Requires all ranks to have same architecture

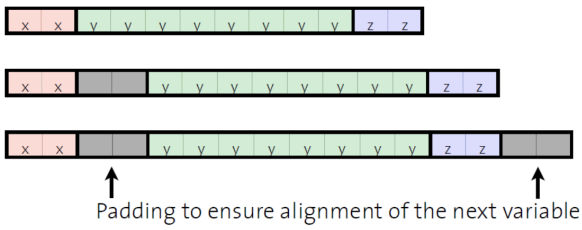
Alignment and Padding

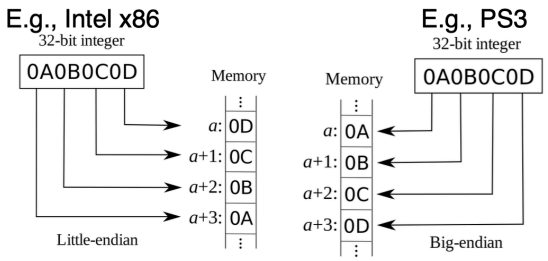
It is not safe to assume how the compiler/architecture lay data in memory.

```
struct parms {
  short  x;
  double y;
  short  z;
};
```

```
MPI_Aint offsets[3] = {0,
                       sizeof(short),
                       sizeof(short)+sizeof(double)};
```

It may be wrong due to unforeseen alignment/padding and endianness. Consider the following data layouts for the structure above:





MPI_Pack / MPI_Unpack

Functions for packing non-contiguous data into contiguous buffers.

- `int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outcount, int *position, MPI_Comm comm)`
- Rationale: packs the data given as input into the outbuf buffer starting at a given position. Outcount indicates the buffer size and position gets updated to point to the first free byte after packing in the data.

- `int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)`
  - Rationale: unpacks data from the buffer starting at given position into the buffer outbuf. position is updated to point to the location after the last byte read.
- `int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)`
  - Rationale: returns in size an upper bound for the number of bytes needed to pack incount values of type datatype. This can be used to determine the required buffer size.

Distributing Parameters (Pack/Unpack)

```
int size_double, size_int;
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD,&size_double);
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD,&size_int);
int buffer_size = 2*size_double+size_int;
char* buffer = new char[buffer_size];
// pack the values into the buffer on the master
if (myRank == 0)
{
 int pos=0;
 MPI_Pack(&a, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
 MPI_Pack(&b, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
 MPI_Pack(&n, 1, MPI_INT,    buffer, buffer_size, &pos, MPI_COMM_WORLD);
}

MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);


int pos=0;
MPI_Unpack(buffer, buffer_size, &pos, &a, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &nsteps, 1, MPI_INT, MPI_COMM_WORLD);
```

✓ Portable
✓ Single Broadcast
✘ Data Duplication

Distributing Parameters Recap

We have analyzed 3 alternatives:

- Multiple Broadcasts: requires excessive latency.
- Bytewise struct passing: non-portable.
- `MPI_Pack/Unpack:` works, but requires extra-copying.

⇓HOWEVER, We need another approach with high efficiency.

What we need: An efficient way to communicate structures or non-contiguous data.

Solution: Describe your data layout to MPI and use it as an MPI datatype.

```
struct parms {
  double a;
  double b;
  int nsteps;
};
```

type	count	offset
MPI_DOUBLE	2	0
MPI_INT	1	16

MPI Struct Type

General Approach: `MPI_Type_create_struct`, describes any data layout.

- `int MPI_Type_create_struct(int count, int blocklengths[], MPI_Aint offsets[], MPI_Datatype types[], MPI_Datatype *newtype)`

- Rationale: Builds a new MPI data type for a general data structure given by types, counts (blocklengths) and their offsets relative to the start of the data structure

- `int MPI_Type_commit(MPI_Datatype *datatype)`
  - Rationale: Commits the data type: finished building it. It can now be used.
- `int MPI_Type_free(MPI_Datatype *datatype)`
  - Rationale: Frees the data type, releasing any allocated memory.

Distributing Parameters (with MPI Structs)

```
// Defining a struct for the parameters
struct parms {
 double a; // lower bound of integration
 double b; // upper bound of integration
 int n;    // number of subintervals
};

MPI_Datatype parms_t;
int          blocklens[2] = {2,1};
MPI_Aint     offsets[2]   = {0,2*sizeof(double)};
MPI_Datatype types[2]     = {MPI_DOUBLE, MPI_INT};

MPI_Type_create_struct(2, blocklens, offsets, types,&parms_t);
MPI_Type_commit(&parms_t);
```

```
parms p;
if (myRank == 0)
 {
  p.a = read(); // lower bound of integration
  p.b = read(); // upper bound of integration
  p.n = read(); // number of subintervals
 }

// broadcast Parameters now using our custom type
MPI_Bcast(&p, 1, parms_t, 0, MPI_COMM_WORLD);

// and now free the type
MPI_Type_free(&parms_t);
```

✓ Single Broadcast
✓ No Data Duplication
? Portable

Safe usage of MPI_Type_create_struct

Solution: `int MPI_Get_address(void *location, MPI_Aint *address)`

- Rationale: Converts a pointer to the correct (MPI Internal) offset representation

```
MPI_Get_address(&p, &p_lb); // start of the struct is the lower bound
MPI_Get_address(&p.a, &p_a); // address of the first double
MPI_Get_address(&p.nsteps, &p_nsteps); // address of the integer
MPI_Get_address(&p+1, &p_ub); // start of the next struct is the upper bound

int blocklens[] = {0, 2, 1, 0};

MPI_Datatype types[] = {MPI_LB, MPI_DOUBLE, MPI_INT, MPI_UB};
MPI_Aint   offsets[] = {0, p_a-p_lb, p_nsteps-p_lb, p_ub-p_lb};
MPI_Datatype parms_t;
MPI_Type_create_struct(4, blocklens, offsets, types,&parms_t);
MPI_Type_commit(&parms_t);
```

```
parms p;
if (myRank == 0)
{
 p.a = read();
 p.b = read();
 p.n = read();
}
```

✓ Portable

Example: Sending a linked list

We can use MPI_Type_create_struct to send the contents of linked lists:

Sender

```
std::list<int> data;
for (int i=0; i<10; ++i) data.push_back(i);

std::vector<MPI_Datatype> types(10,MPI_INT);
std::vector<int>          blocklens(10,1);
std::vector<MPI_Aint>     offsets;
// How do we resolve the offsets?
```

Receiver

```
std::vector<int> data(10);
MPI_Status status;
MPI_Recv(&data[0], 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
for (int i=0; i < data.size(); i++) printf("%d\n", data[i]);
```

To send the linked list:

1. View the whole memory as base struct which using absolute addresses as offsets.
2. Pass MPI_BOTTOM as the buffer pointer in communication to indicate absolute addressing.

Sender

```
for (int& x : data) {
MPI_Aint address;
MPI_Get_address(&x, &address); // use absolute addresses
offsets.push_back(address);
}

MPI_Datatype list_type;
MPI_Type_create_struct(10, &(blocklens[0]), &offsets[0], &types[0] ,&list_type);
MPI_Type_commit(&list_type);

MPI_Send(MPI_BOTTOM, 1, list_type, 0, 42, MPI_COMM_WORLD);

MPI_Type_free(&list_type);
```

Cheatsheet

1 Cheatsheet for Examination

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

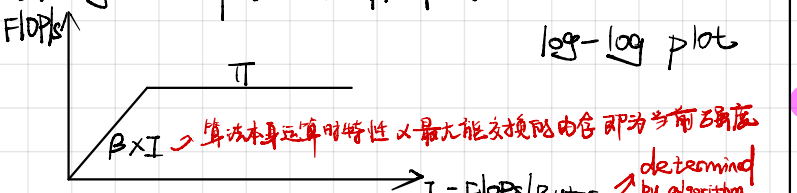
Q: Depends on cache size
- No cache (reading directly from memory): 每一次数据访问都被 counted
- perfect cache: 数据只用读取+写入一次

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

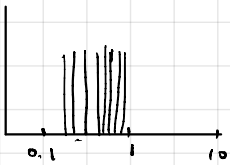
⇒ partial {parallel / serial}

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

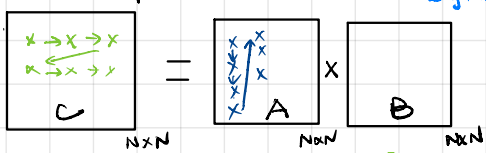
<3> Cache

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)]

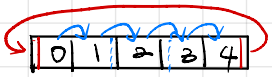
↓

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()

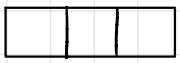
1. We need to exchange the ghost information. After that, we can use Diffusion2D.advance()
2. main() 函数中实现 MPI_Init(), rank, size.
   By the way D L N 这些系统参数在 compile 时指定。
3. For local_N, achieve it in the Initialization function (来初始化)
   ↓ number of grid points of this process
4. 初始化函数最后一步有 initialize(),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

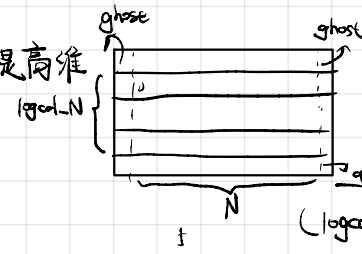
⇓ 见 mock exam coding MPI

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()

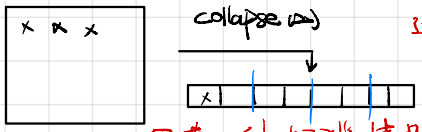
- Define variable global_max globle_min
- use MPI_Allreduce() for message transmission between all ranks → See HW5

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++}



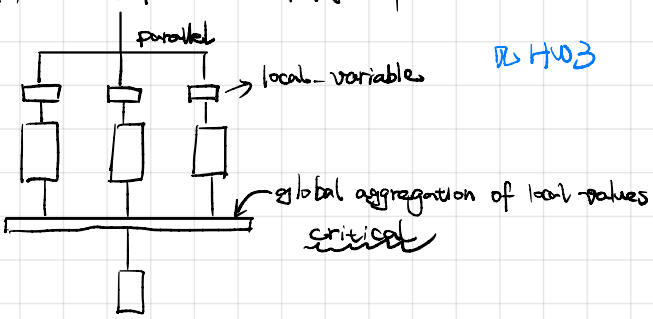
这种 shared memory ++ 很容易发生 race condition.

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



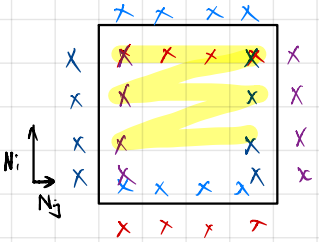
见 HW3

<补充，CSR format> HW6

Q1 : CSR format

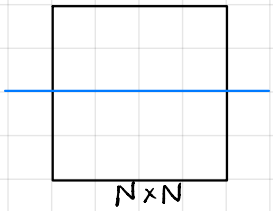
1. index for K, J is from 0

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



<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)

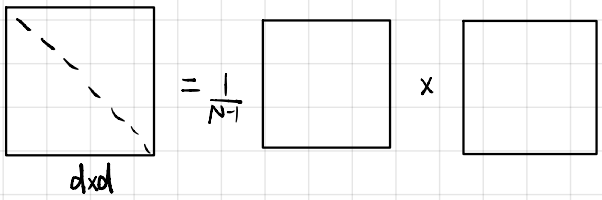
<4> Standardization / Normalization of Data

(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.

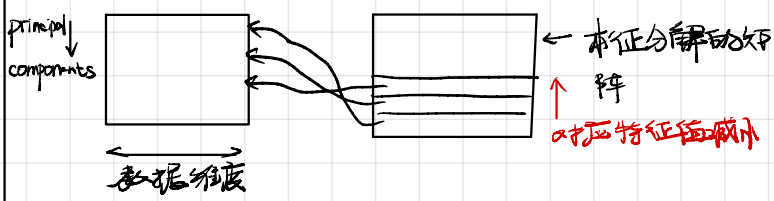


↑ Due to its symmetry，整个过程是3个 loop 嵌套。

<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

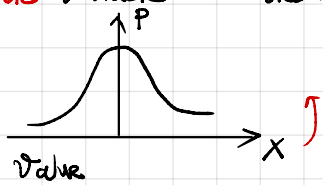
$\varepsilon = O(M^{-2/d})$ ↑ $M = n^d$ ⇒ # point evaluation

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

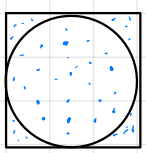
↓ Expected Value

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

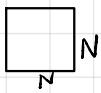
① Gather 这种操作的结果是 recvbuf 被拼成了一个很长的结果集，顺序以rank的顺序为主

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