CS381-16 PARALLEL & DISTRIBUTED COMPUTING

PROFESSOR JUN LI

SPRING 2022

February 03, 2022

A process can run on one single core.

You can't improve performance by creating more instances of a process

Modify/write new code so you can leverage more processors.

Parallel programming → Performance programming

Programmer must divide application so that

- each processor has roughly equal at same time
- overhead of scheduling & coordination does not fritter away the potential performance benefits of parallelism.

ex:

- one task runs faster than other & one CPU waits for other completion then the CPU is also losing performance

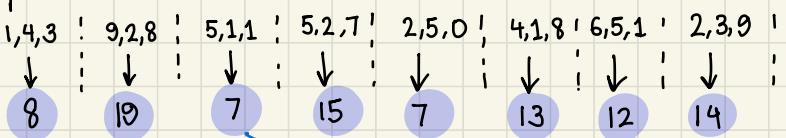
A general conversion from a serial program to a parallel program does not always perform well. → it is sometimes better to devise a new algorithm altogether.

Example: serial to parallel

each of the cores runs the code, taking n/p values

each core will have different values of my_sum

Ex: p = 8 cores, n = 24



Values of my_sum across cores

main core to do final my_sum

this step can be added to the code

runs the "if"

each core is programmed to send to main/"master" core w/ a if/else

runs the "else"

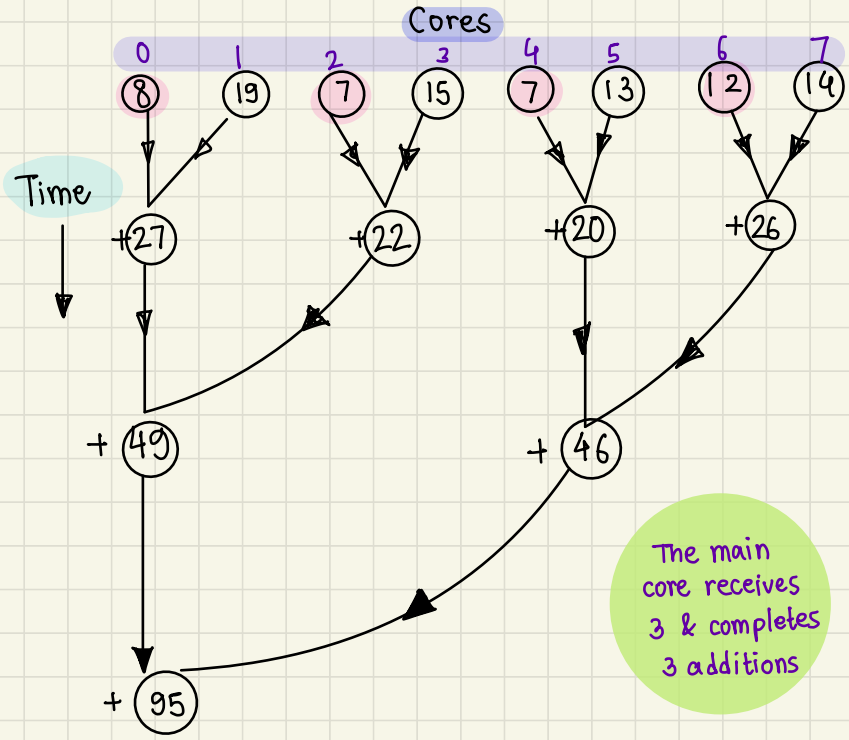
we can use a master core to add the final result

The main core does 7 receives & 7 additions

This is very similar to the idea of divide & conquer.

Can we do better?

The master code does not do all the work - the computation is shared among the other cores. - the cores are paired to add results.



The main core receives 3 & completes 3 additions

- We reduced from 8 numbers to 4 numbers
- we split the 7 additions to different cores. Each core has a small number of additions.
- we might not split evenly but we can still reduce the number of additions

- If you divide the size of the problem by 2, you get $\log_2 x$

Improvement by a factor of 100

The effect/difference is more dramatic w/ a larger number of cores.

How to write parallel programs?

There are two ways:

① Task parallelism - partition different tasks among cores

② Data parallelism - divide input data among cores

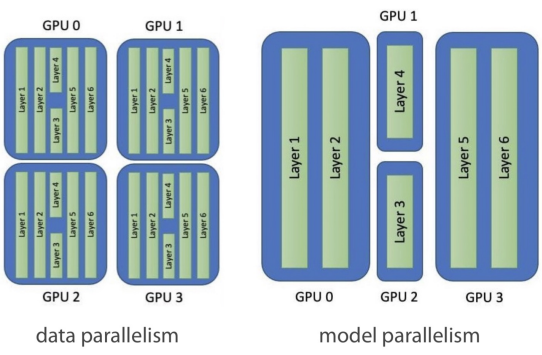
Ex: 15 Q's 300 Exams & 3 TA's

- Data parallelism
  - TA#1: 100 exams
  - TA#2: 100 exams
  - TA#3: 100 exams
- Task parallelism:
  - TA#1: Q 1-5 in all exams
  - TA#2: Q 11-15 in all exams
  - TA#3: Q 6-10 in all exams

Parallelism in GPU → take the slide Data parallelism & Model parallelism

Division of work – data parallelism

```
1 my_sum = 0;
2 my_first_i = . . . ;
3 my_last_i = . . . ;
4 for (my_i = my_first_i; my_i < my_last_i; my_i++) {
5     my_x = Compute_next_value( . . .);
6     my_sum += my_x;
7 }
```



data parallelism model parallelism

Division of work – task parallelism

```
1 if (I'm the master core) {
2     sum = my_X;
3     for each core other than myself {
4         receive value from core;
5         sum += value;
6     }
7 } else {
8     send my_x to the master;
9 }
```

Tasks
1) Receiving
2) Addition

Cores need to coordinate their work

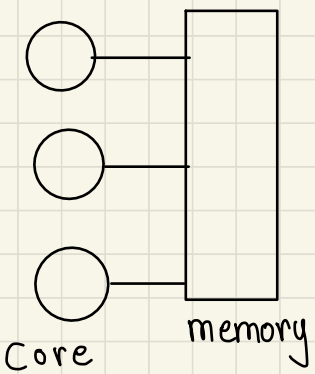
① Communication
② Load balancing
③ Synchronization

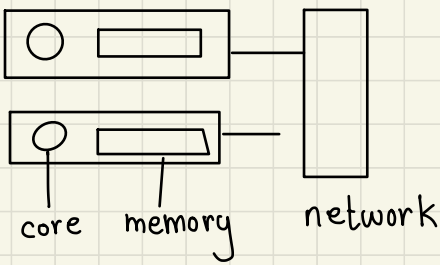
Types of Parallel Systems

February 10, 2022

① Shared memory
- the cores can share access to the computer's memory
- coordinate the cores by having them examine & update shared memory locations

② Distributed Memory
- each core has its own private memory
- cores must communicate explicitly over a network





- **Concurrent computing** – a program is one in which multiple tasks can be in progress at any instant.
- **Parallel computing** – a program is one in which multiple tasks cooperate closely to solve a problem
- **Distributed computing** – a program may need to cooperate with other programs to solve a problem.

- Concurrent Computing: a program is one in which multiple tasks can be in progress at any instant

- Distributed computing
  ex: internet
- Parallel computing
  - more narrow & closely related tasks
  ex: multiple cores on a computer with multiple threads

MPI is the model for distributed memory system

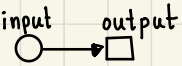
Concluding Remarks

- The laws of physics have brought us to the doorstep of multicore technology.
- Serial programs typically don't benefit from multiple cores.
- Automatic parallel program generation from serial program code isn't the most efficient approach to get high performance from multicore computers.

- Learning to write parallel programs involves learning how to coordinate the cores.
- Parallel programs are usually very complex and therefore, require sound program techniques and development.

Parallel Hardware & Parallel Software

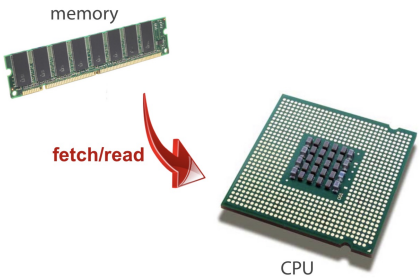
Some background

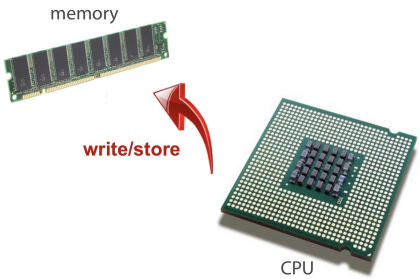


- computer runs one program at a time
  - the architecture behind it is the Von Neumann Architecture
    - CPU & Memory connected by bus
    - CPU has
      - ALU: calculations
      - Control registers: decoding the instructions & instruct the CPU for corresponding functionality
    - Registers store input & output data
    - Load instructions can move data from & between CPU registers & main memo
- Main memory is a collection of locations, each of which is capable of storing both instructions & data in von neumann architecture
- CPU: (see before)
  - Register: very fast storage
  - Program counter: find instruction for next step
  - Bus: hardware & wires connecting CPU & memory
- Von Neumann Bottleneck: the bottleneck can become the bus
- A operating system process: An instance of a computer program being executed.

  Components of a process:
  - The executable machine language program
  - A block of memory
  - Descriptors of resources OS allocated to the process
  - Security information
  - Information of the state of process

    ex: kernel state
- Multitasking: running multiple processes virtually at the same time. Time on the CPU is split into time slices used by processes. One process gets one time slice, & once it's up it waits for another time slice. If a process needs to wait for a resource, it will block & the OS moves to another process.





An operating system "process"

An instance of a computer program that is being executed.

Components of a process:

- The executable machine language program.
- A block of memory.
- Descriptors of resources the OS has allocated to the process.
- Security information.
- Information about the state of the process.

Multitasking

Gives the illusion that a single processor system is running multiple programs simultaneously.

Each process takes turns running. (time slice)

- After its time is up, it waits until it has a turn again.

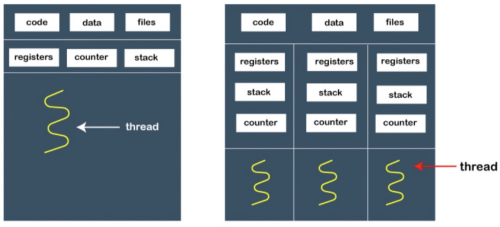
If a process needs to wait for a resource, it will stop executing and the operating system can run another process. (block)

Threading

Threads are contained within processes.

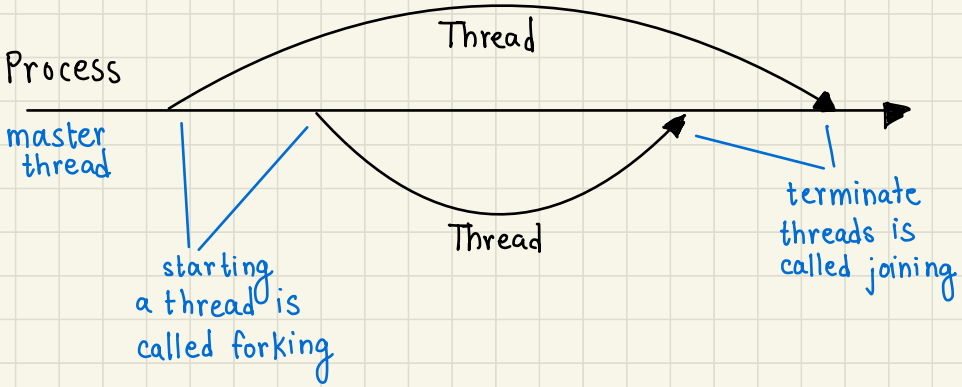
They allow programmers to divide their programs into (more or less) independent tasks.

The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.



Single-threaded process Multi-threaded process

- Threading : They allow programmers to divide their programs into (more or less) independent tasks.
  When one thread blocks, another can run.
  Threads are contained within processes.



- Caching : A collection of memory locations that can be accessed in less time than some other memory locations.
  A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.
- Principle of locality : Accessing one location is followed by an access of a nearby location
  Spatial locality : accessing a nearby location
  Temporal location: accessing in the near future

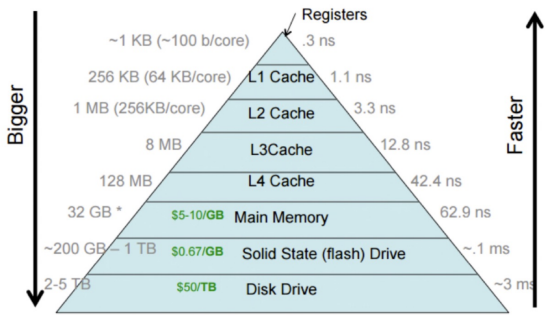
Von Neumann Model - Modifications to it

Basics of caching

A collection of memory locations that can be accessed in less time than some other memory locations.

A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.

Memory hierarchy



Different levels of cache between local & main memory

The bigger the cache, the slower the access time

In general a **cache** is a collection of memory locations that can be accessed in less time than some other memory locations. In our setting, when we talk about caches we'll usually mean a **CPU cache**, which is a collection of memory locations that the CPU can access more quickly than it can access main memory. A CPU cache can either be located on the same chip as the CPU or it can be located on a separate chip that can be accessed much faster than an ordinary memory chip.

Once we have a cache, an obvious problem is deciding which data and instructions should be stored in the cache. The universally used principle is based on the idea that programs tend to use data and instructions that are physically close to recently used data and instructions. After executing an instruction, programs typically execute the next instruction; branching tends to be relatively rare. Similarly, after a program has accessed one memory location, it often accesses a memory location that is physically nearby. An extreme example of this is in the use of arrays. Consider the loop

```
float z[1000];
. . .
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

Accessing one location is followed by an access of a nearby location.

Spatial locality

accessing a nearby location

Temporal locality

accessing in the near future

Principle of locality

```
1 float z[1000];
2 …
3 sum = 0.0;
4 for (i = 0; i < 1000; i++)
5    sum += z[i];
```

The principle that an access of one location is followed by an access of a nearby location is often called **locality**. After accessing one memory location (instruction or data), a program will typically access a nearby location (**spatial** locality) in the near future (**temporal** locality).

In order to exploit the principle of locality, the system uses an effectively *wider* interconnect to access data and instructions. That is, a memory access will effectively operate on blocks of data and instructions instead of individual instructions and individual data items. These blocks are called **cache blocks** or **cache lines**.

Level of Cache

L1 - fastest & smallest
L2
L3 - largest & slowest

In computer networking, latency is **an expression of how much time it takes for a data packet to travel from one designated point to another**. Ideally, latency will be as close to zero as possible.

Cache can provide higher access latency
Cache saves a copy of data from main memory

Cache hit

- if we have cache we look through levels of cache for the data.
- all values in cache are also in memory
- you find the value

Cache miss → example: we want to fetch x in cache but it's only in main memory

- value not available in L1, L2, L3 & eventually we go back to main memory to load the data
- if we always have cache miss, it is not helpful

However, the value of cache may be inconsistent with main memory.
There are different policies to handle this.

1. Write-through cache: update the data in memory at the time it is written to cache. This is more reliable as you have less chance to lose data.

2. Write-back cache: mark data in cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.
   - this can provide higher performance but we might use write through is not always available, & you have less chance to lose data.

When the CPU writes data to a cache, the value in the cache and the value in main memory are different or **inconsistent**. There are two basic approaches to dealing with the inconsistency. In **write-through** caches, the line is written to main memory when it is written to the cache. In **write-back** caches, the data isn't written immediately. Rather, the updated data in the cache is marked *dirty*, and when the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

Cache Mappings

1. Full associative: a new line can be placed at any location in the cache
2. Direct mapped: each cache has a unique location that it is mapped to However, this way the excess space might not be offerable even if accessible as it's uniquely mapped.
3. n-way set associative: each cache line can be placed in n-different locations of cache.

When more than one line in memory can be mapped to several different locations in cache so we need to be able to decide which line should be replaced or evicted.

When more than one line in memory can be mapped to several different locations in a cache (fully associative and n-way set associative), we also need to be able to decide which line should be replaced or **evicted**. In our preceding example, if, for example, line 0 is in location 0 and line 2 is in location 1, where would we store line 4? The most commonly used scheme is called **least recently used**. As the name

example of caching methods:

	Cache Location		
Memory Index	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Cache & Programs ?

If we load column by column instead of row by row in the cache, we still have space in the cache line.

If we visit by columns we have a better cache rate

Caches and programs

```
double A[MAX][MAX], x[MAX], y[MAX];
/* Initialize A and x, assign y = 0 */
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j< MAX; j++)
        y[i] *= A[i][j] * x[j];
/* Assign y = 0 */
/* Second pair of loops */
for (j = 0; < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j] * x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

example, C stores two-dimensional arrays in "row-major" order. That is, although we think of a two-dimensional array as a rectangular block, memory is effectively a huge one-dimensional array. So in row-major storage, we store row 0 first, then row 1, and so on. In the following two code segments, we would expect the first pair of nested loops to have much better performance than the second, since it's accessing the data in the two-dimensional array in contiguous blocks.

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

first pair is faster than next pair since there are less cache misses since it accesses the loaded cache more often, there are less misses.

more ahead →

Solution:

Suppose the matrix has order 8 or 64 elements; the cache line size is still 4; the cache can store 4 lines; and the cache is direct-mapped. Then the following table shows how A is stored in cache lines.

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[0][4]	A[0][5]	A[0][6]	A[0][7]
2	A[1][0]	A[1][1]	A[1][2]	A[1][3]
3	A[1][4]	A[1][5]	A[1][6]	A[1][7]
4	A[2][0]	A[2][1]	A[2][2]	A[2][3]
5	A[2][4]	A[2][5]	A[2][6]	A[2][7]
6	A[3][0]	A[3][1]	A[3][2]	A[3][3]
7	A[3][4]	A[3][5]	A[3][6]	A[3][7]
8	A[4][0]	A[4][1]	A[4][2]	A[4][3]
9	A[4][4]	A[4][5]	A[4][6]	A[4][7]
10	A[5][0]	A[5][1]	A[5][2]	A[5][3]
11	A[5][4]	A[5][5]	A[5][6]	A[5][7]
12	A[6][0]	A[6][1]	A[6][2]	A[6][3]
13	A[6][4]	A[6][5]	A[6][6]	A[6][7]
14	A[7][0]	A[7][1]	A[7][2]	A[7][3]
15	A[7][4]	A[7][5]	A[7][6]	A[7][7]

Assuming that no lines of A are in the cache when the first pair of loops begins, we see that there will be two misses for each row of A. Also, after the first two rows have been read, the cache will be full, and each miss will evict a line. As in the example in the text, an evicted line won't need to be read again. So we see that the total number of misses for the first pair of loops is 16, or

$$\frac{\text{number of elements in A}}{\text{cache line size}}.$$

More generally, then, for the first pair of loops, the number of misses is only affected by the size of A, not the size of the cache.

For the second pair of nested loops, let's also assume that no lines of A are in the cache when the loops begin. When we're working with column 0 ($j = 0$), each time we multiply `A[i][0]*x[0]` we'll need to first load a new cache line, since the previously read lines will only contain elements from rows $< i$. So executing the loop with $j = 0$, will result in 8 misses. After reading in the four lines containing `A[0][0]`, `A[1][0]`, `A[2][0]`, and `A[3][0]`, respectively, subsequent reads of elements of column 0 will evict these four lines. So after completing the multiplications involving column 0 of A, no elements in the first 4 rows of A will be stored in the cache. So every read of an element of A in rows 0–3 of column 1 will result in a miss. In fact, we see that every multiplication will result in a miss and there will be 64 misses.

Observe, however, in the second pair of loops if we have an 8 line cache, then the multiplications involving columns 1–3 of A won't result in misses, and we won't have additional misses until we start the multiplications by elements in column 4. Once the lines containing elements of column 4 are loaded, there won't be any additional misses, and we see that with an 8 line cache, the total number of misses is reduced to 16.

Virtual Memory

We may not have memory for a large program in main memory.
Virtual memory functions as a cache for secondary storage.
It exploits the principle of spatial & temporal locality
It only keeps the active parts of running programs in main memory
Main memory serves as cache for execution of the program.

Virtual memory (1)

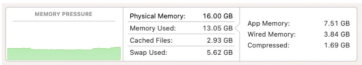
If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory.

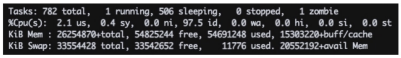
Virtual memory functions as a cache for secondary storage.

- It exploits the principle of spatial and temporal locality.
- It only keeps the active parts of running programs in main memory.

Virtual memory (2)

Swap space - those parts that are idle are kept in a block of secondary storage.

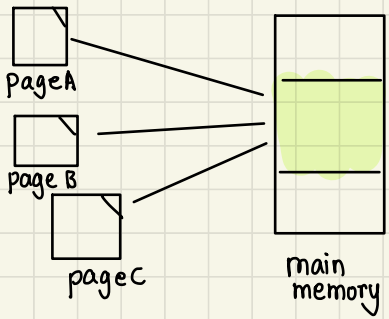




Pages – blocks of data and instructions.

- Usually these are relatively large
- Most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.

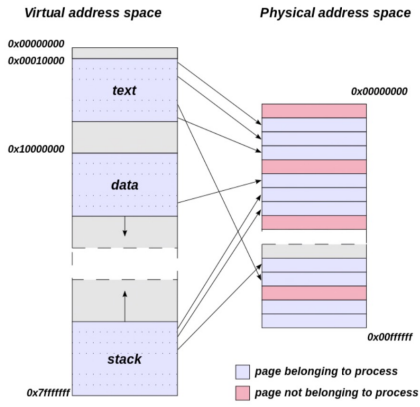
Swap space - those parts that are idle are kept in a block of secondary storage.
Pages - blocks of data & instructions.
Usually these are relatively large. Most systems have a fixed page size that currently ranges from 4-16 kilobytes.



Page table :
we have a virtual page number & the offset.
A virtual page number is mapped to a physical page number

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	···	13	12	11	10	···	1	0
1	0	···	1	1	0	0	···	1	1

Page table (2)



Translation-lookaside buffer (TLB)

TLB is a cache for the page table.

Using a page table may increase the runtime.

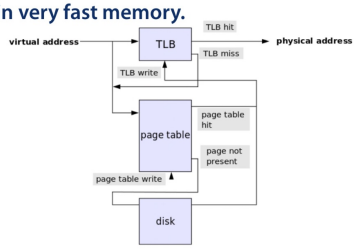
It caches a small number of entries (typically 16-512) from the page table in very fast memory.

Translation-lookaside buffer (TLB)

Using a page table has the potential to significantly increase each program's overall run-time.

A special address translation cache in the processor.

It caches a small number of entries (typically 16–512) from the page table in very fast memory.



Page fault – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.

TLB hit : the page number found

TLB miss: the page number not in TLB. A new entry is then created from main memory to the TLB.

Instruction Level Parallelism (ILP)

Attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions.

Pipelining - functional units are arranged in stages.

Multiple issue - multiple instructions can be simultaneously initiated.

Pipelining example (1)

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Add the floating point numbers
9.87×10^4 and 6.54×10^3

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Pipelined Addition.
Numbers in the table are subscripts of operands/results.

Assume each operation takes one nanosecond (10^{-9} seconds).

```
1 float x[1000], y[1000], z[1000];
2 for (i = 0; i < 1000; i++) {
3     z[i] = x[i] + y[i];
4 }
```

This for loop takes about 7000 nanoseconds.

One floating point addition still takes 7 nanoseconds.

But 1000 floating point additions now takes 1006 nanoseconds!

Divide the floating point adder into 7 separate pieces of hardware or functional units.

First unit fetches two operands, second unit compares exponents, etc.

Output of one functional unit is input to the next.

Multiple Issue - Part of Instruction Level Parallelism

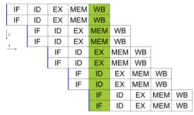
February 17, 2022

Multiple Issue (1)

Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

```
1 for (i = 0; i < 1000; i++)
2     z[i] = x[i] + y[i];
```

static multiple issue - functional units are scheduled at compile time.



dynamic multiple issue – functional units are scheduled at run-time.

superscalar

1. Static Multiple Issue
2. Dynamic multiple issue: superscalar

Speculation

- CPU can guess what instruction will be executed for the next few steps
- if it can guess correctly, we can improve performance.

In speculation, it might make a guess & make an independent similar instruction later.

```
z = x + y
if (z > 0)
    w = x
```

in speculation, the computer runs these instructions together simultaneously

Speculation (1)

In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.

In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.

```
1 z = x + y;
2 if (z > 0)
3       w = x;
4 else
5       w = y;
```

If the system speculates incorrectly, it must go back and recalculate w = y.

Hardware Multithreading

If speculation isn't a good opportunity.
Ex- one result is dependent on the other closely like Fibonacci

There aren't always good opportunities for simultaneous execution of different instructions.

Ex. Fibonacci number with dynamic programming

Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.

Ex., the current task has to wait for data to be loaded from memory.

Then we can do multithreading

1. Fine grained multithreading (FMT)
   - CPU knows how many threads
   - the processor switches between threads after each instruction, skipping threads that are stalled
   - sequential instructions have to wait after every instruction.

Pros: potential to avoid wasted machine time due to stalls.

Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

2. Course-grained Multithreading (CMT)
   - the processor switches threads only when threads are stalled
   - switching does not need to be instantaneous
   - con: there are switches in shorter stalls which can cause delays

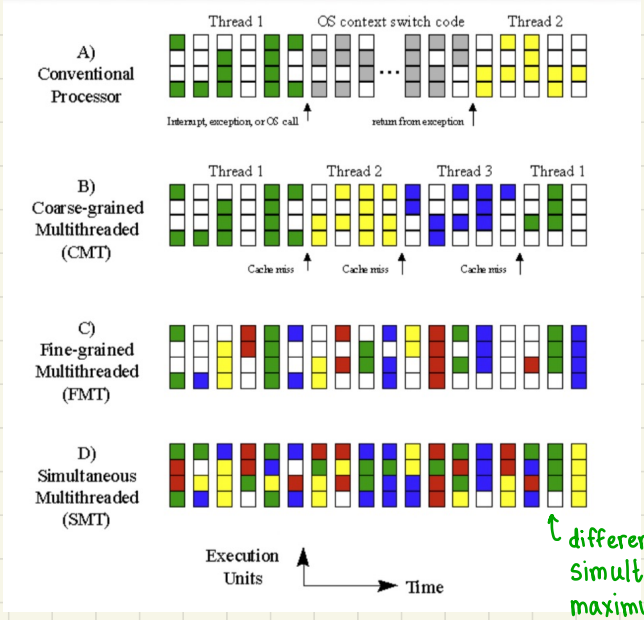
Pros: switching threads doesn't need to be nearly instantaneous.

Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays.

3. Simultaneous Multithreaded
   - a thread carries out multiple instructions & allocates based on available resource

Simultaneous multithreading (SMT) - a variation on fine-grained multithreading.

Allows multiple threads to make use of the multiple functional units.



↑ different threads working simultaneously to ensure maximum utilization of resources

A. Conventional Processor

- switches happen through exception, interrupt or OS call
- there is an OS context switch call

Parallel I

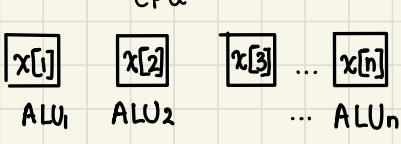
Flyn's Taxonomy

1. SISD
2. SIMD
3. MISD
4. MIMD

1. SIMD

- multiple data stream - data divided into multiple parts - data parallelism

ex




```
for(int i=0 ; i<n ; i++) {
    x[i] += y[i]
}
```

with multiple data units, we divide the data & make multiple rounds to carry out the task with different data points

- cons: All ALUs are required to execute the same instruction or remain idle
  Only effecient for large data parallel problems, not other types of more complex parallel programs

What if we don't have as many ALUs as data items?

Divide the work and process iteratively.

Ex. m = 4 ALUs and n = 15 data items.

Round3	ALU_1	ALU_2	ALU_3	ALU_4
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

SIMD drawbacks

All ALUs are required to execute the same instruction, or remain idle.

In classic design, they must also operate synchronously.

The ALUs have no instruction storage.

Efficient for large data parallel problems, but not other types of more complex parallel problems.

Vector processors (1)

Operate on arrays or vectors of data while conventional CPU's operate on individual data elements or scalars.

Vector registers.

Capable of storing a vector of operands and operating simultaneously on their contents.

Vectorized and pipelined functional units.

The same operation is applied to each element in the vector (or pairs of elements).

Vector instructions.

Operate on vectors rather than scalars.

Interleaved memory.

Multiple "banks" of memory, which can be accessed more or less independently.

Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.

Strided memory access and hardware scatter/gather.

The program accesses elements of a vector located at fixed intervals.

Vector processors - Pros

Fast.

Easy to use.

Vectorizing compilers are good at identifying code to exploit.

Compilers also can provide information about code that cannot be vectorized.

Helps the programmer re-evaluate code.

High memory bandwidth.

Uses every item in a cache line.

Vector processors - Cons

They don't handle irregular data structures as well as other parallel architectures.

A very finite limit to their ability to handle ever larger problems. (scalability)

Bandwidth is **the data transfer capacity of a computer network in bits per second (Bps)**. The term may also be used colloquially to indicate a person's capacity for tasks or deep thoughts at a point in time.

ex: Vector Processor:

operate on arrays or vectors of data while conventional CPU's operate on individual data elements or scalars

- Vectorized & pipelined functional units
  the same operation is applied to each element in vector (or pairs of elems
  The operation is performed on a vector rather than scalar

- Interleaved memory
  - multiple "banks" of memory which can be accessed independently
    As they're accessed independentl there is less delay
  - Strided memory access & hardware scatter gather
    - the program accesses elements of a vector at fixed intervals

Vector processors Pros:

1. Uses every item in a cache line: as when data is carried to cache, the entire array of data is copied

Cons

1. Hard to scale: the number of functional components is fixed. A finite ability to handle larger problems
2. Can't handle irregular data.

Ex: GPU

Convert graphical data to pixels

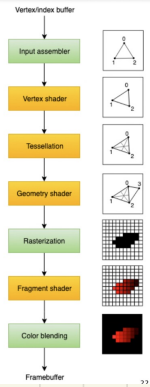
Graphics Processing Units (GPU)

Real time graphics application programming interfaces or API's use points, lines, and triangles to internally represent the surface of an object.

A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.

Several stages of this pipeline (called shader functions) are programmable.

Typically just a few lines of C code.



Shader functions are also implicitly parallel, since they can be applied to multiple elements in the graphics stream.

GPU's can often optimize performance by using SIMD parallelism.

The current generation of GPU's use SIMD parallelism.

Although they are not pure SIMD systems.

2. MIMD

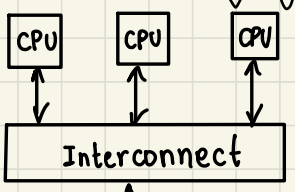
- processor can receive multiple streams of instructions & those instructions can receive multiple streams of data
- this is what we'll use in parallel computing

MIMD

Supports multiple simultaneous instruction streams operating on multiple data streams.

Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

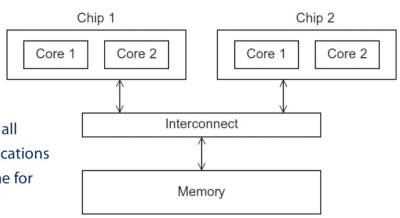
ex: Shared Memory System



ex: UMA multicore system
NUMA multicore system

UMA multicore system

shared Memory System



Time to access all the memory locations will be the same for all the cores.

Shared Memory System (1)

A collection of autonomous processors is connected to a memory system via an interconnection network.

Each processor can access each memory location.

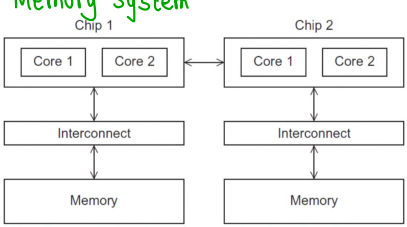
The processors usually communicate implicitly by accessing shared data structures.

Most widely available shared memory systems use one or more multicore processors.

(multiple CPU's or cores on a single chip)

NUMA multicore system

Shared Memory System



A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

- Distributed Memory System
  - Cluster
    - A collection of commodity systems
    - each node of a cluster indicates a unit

Distributed Memory System

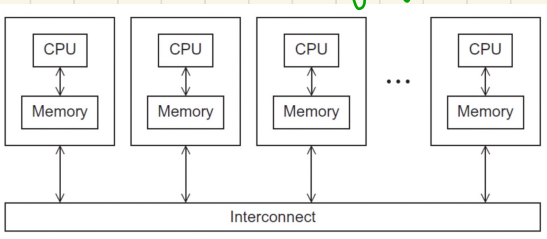
Clusters (most popular)

A collection of commodity systems.

Connected by a commodity interconnection network.

Nodes of a cluster are individual computations units joined by a communication network.

Distributed Memory System



Interconnection networks

Affects performance of both distributed and shared memory systems.

Two categories:

Shared memory interconnects

Distributed memory interconnects

Shared Memory Interconnects

Shared memory interconnects

Bus interconnect

A collection of parallel communication wires together with some hardware that controls access to the bus.

Communication wires are shared by the devices that are connected to it.

As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.

Switched interconnect

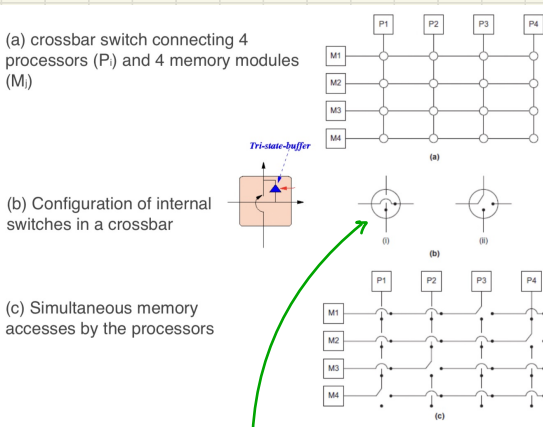
Uses switches to control the routing of data among the connected devices.

Crossbar

Allows simultaneous communication among different devices.

Faster than buses.

But the cost of the switches and links is relatively high. - eventually more expensive



internal of crossbar

- there are internal switches that can decide which memory visits which processor

Distributed Memory Interconnects

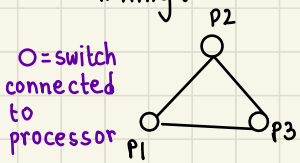
February 22, 2022

Two groups

① Direct interconnect: each switch is connected to a processor memory pair, & the switches are connected to each other.

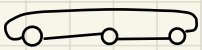
There can be different topology of switches

1. Ring:

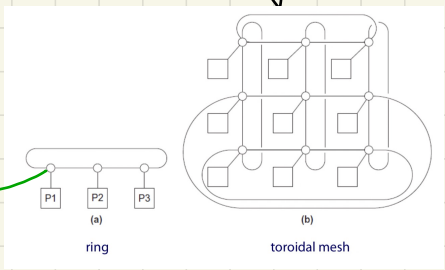


O = switch connected to processor

or

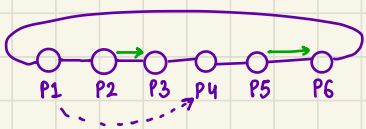


switch



There may be a case that there is contention in a link

Here, P2 cannot access P4 since there is no path available



2. Toroidal mesh: we have rings in higher dimensions.
   It will have more links & be more expensive
   Allows more concurrent communication at the same time.

② Indirect Interconnects
- each processor sends data to switching network
- switches may not be directly connected to a processor

Bisection Width: or "connectivity" / how many simultaneous connections between halves
- used to measure performance.
- a measure of number of "simultaneous connections" or connectivity

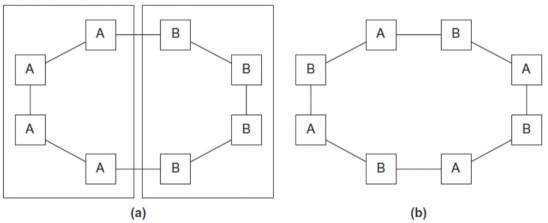
Bandwidth
- the rate at which a link can transmit data
- usually given in megabits or megabytes per second

Bisection bandwidth
- combination of throughput - it shows the bandwidth

A measure of network quality.

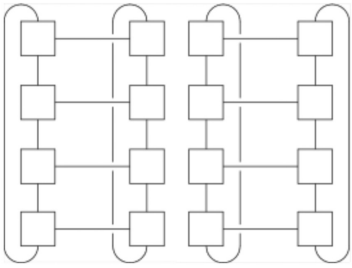
Instead of counting the number of links joining the halves, it sums the bandwidth of the links.

Two bisections of a ring



A bisection of a toroidal mesh

remove the minimum number of links needed to split the set of nodes into two equal halves



p : the number of processors

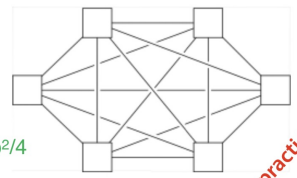
bisection width : $2\sqrt{p}$

Fully connected network

- each switch is directly connected to every other switch.
- bisection width = $p^2/4$ (total number of links)

Fully connected network

Each switch is directly connected to every other switch.

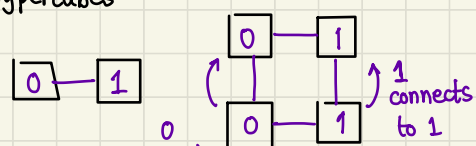


bisection width = $p^2/4$

impractical

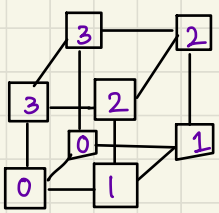
total link = p(p-1)/2

Hypercubes



2 switches

8 switches



32 switches

Number of switches increases 2^3

$p = 2^d$

bisection width = $\frac{p}{2} = 2^{d-1}$

Hypercube

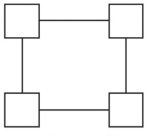
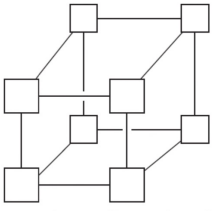
Highly connected direct interconnect.

Built inductively:

A one-dimensional hypercube is a fully-connected system with two processors.

A two-dimensional hypercube is built from two one-dimensional hypercubes by joining "corresponding" switches.

Similarly a three-dimensional hypercube is built from two two-dimensional hypercubes.

	d=1	d=2	d=3
	 one-dimensional	 two-dimensional	 three-dimensional
BW	1	2	4

$p = 2^d$ bisection width : $\frac{p}{2} = 2^{d-1}$

② Indirect Interconnects

- each processor sends data to switching network

Omega network

- total number of regular switches cross bar switches x 4

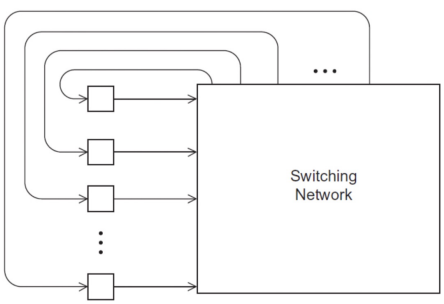
Indirect interconnects

Simple examples of indirect networks:

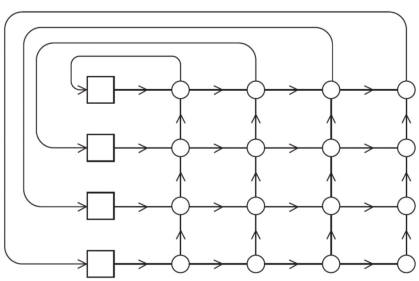
- Crossbar
- Omega network

Often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network.

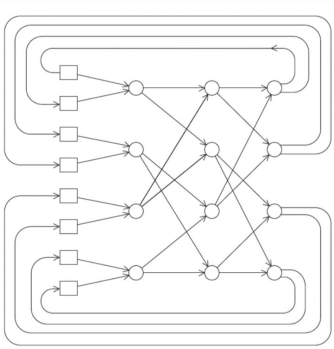
A generic indirect network



Crossbar interconnect for distributed memory

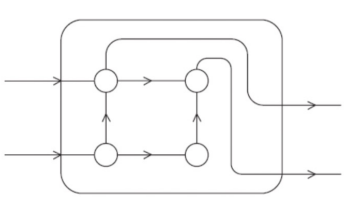


An omega network



$\frac{1}{2}p \log_2 p$ crossbar switches

A switch in an omega network



More definitions

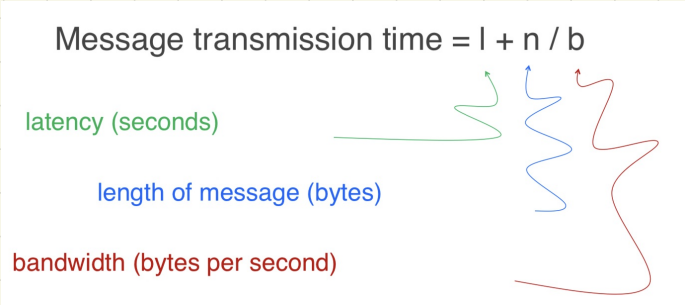
Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.

Latency

The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.

Bandwidth

The rate at which the destination receives data after it has started to receive the first byte.



Cache Coherence

As programmers have no control when cache gets updated, we have issue w/ parallel processing.

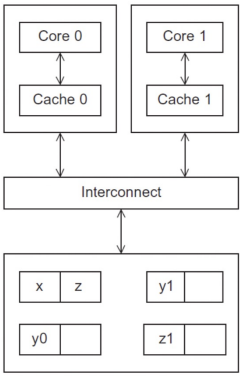
To enforce cache coherence, the cores share a bus.

When we have any write of data in the memory, it is placed on the bus so it can be seen by all the cores.

→ more on next page

Cache coherence

Programmers have no control over caches and when they get updated.



A shared memory system with two cores and two caches

Cache coherence

y0 privately owned by Core 0
y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

Time	Core 0	Core 1
0	y0 = x;	y1 = 3*x;
1	x = 7;	Statement(s) not involving x
2	Statement(s) not involving x	z1 = 4*x;

y0 eventually ends up = 2
y1 eventually ends up = 6
z1 = ???

Snooping Cache Coherence

The cores share a bus .

Any signal transmitted on the bus can be "seen" by all cores connected to the bus.

When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.

If core 1 is "snooping" the bus, it will see that x has been updated and it can mark its copy of x as invalid.

Directory Based Cache Coherence

Uses a data structure called a directory that stores the status of each cache line.

When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Snooping cache coherence

There are two main approaches to insuring cache coherence: **snooping cache coherence** and directory-based cache coherence. The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus. Thus, when core 0 updates the copy of x stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that x has been updated and it can mark its copy of x as invalid. This is more or less how snooping cache coherence works. The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the *cache line* containing x has been updated, not that x has been updated.

A couple of points should be made regarding snooping. First, it's not essential that the interconnect be a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and write-back caches. In principle, if the interconnect is shared—as with a bus—with write-through caches there's no need for additional traffic on the interconnect, since each core can simply "watch" for writes. With write-back caches, on the other hand, an extra communication *is* necessary, since updates to the cache don't get immediately sent to memory.

Directory-based cache coherence

Unfortunately, in large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated (but see Exercise 2.15). So snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade. For example, suppose we have a system with the basic distributed-memory architecture (Figure 2.4). However, the system provides a single address space for all the memories. So, for example, core 0 can access the variable x stored in core 1's memory, by simply executing a statement such as y = x.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory**. The directory stores the status of each cache line. Typically, this data structure is distributed; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. Thus, when a line is read into, say, core 0's cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Clearly there will be substantial additional storage required for the directory, but when a cache variable is updated, only the cores storing that variable need to be contacted.

Answer to 2.15

(a) Core 0: x = 5
Core 1: y = x
Core 1 does not have x in its cache.
Core 0 uses snooping cache coherence but it uses write back cache which means that extra communication is necessary since the cache does not get immediately sent to memory. If the cache for x = 5 is sent to memory by the time Core 1 executes y = x, then y receives the value 5 otherwise it receives an old value of x which was already in memory.

(b) If it uses a directory based protocol, so when x = 5 is read into Core 0's cache, the directory entry corresponding to that line is updated to show that Core 0 has a copy of the line. However when a cache variable is updated only the cores storing that variable is updated. As Core 1 does not already have the value in its cache. Similarly from before if x is not updated in main memory, Core 1 will have an old copy of x, else if it is updated Core 1 will receive the updated value for x.

(c) In the last two problems, there is guaranteed cache coherence only if both cores already have the variable in cache. It should be the case that a core which does not have a value in it's cache should have the updated value. We can do this by employing a critical section, so that a core can only get the value of x once it's been updated in main memory.

Parallel Software

① SPMD

- single program multiple data

SPMD – single program multiple data

A SPMD programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

```
if (I'm thread/process i)
   do this;
else
   do that;
```

From now on…

In shared memory programs:

- Start a single process and fork threads.
- Threads carry out tasks.

In distributed memory programs:

- Start multiple processes.
- Processes carry out tasks.

→ the burden is on software

Writing Parallel Programs

1. Divide the work among the processes/threads
   (a) so each process/thread gets roughly the same amount of work
   (b) and communication is minimized.

```
double x[n], y[n];
…
for (i = 0; i < n; i++)
      x[i] += y[i];
```

2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among processes/threads.

These last two problems are often interrelated.

Shared Memory

- Dynamic threads : real-time thread creation & deletion.
- Static threads
  - use case: network service : when we have a new incoming connection we create a new thread. We want to keep a pool of threads for less work & a lower overhead

Shared Memory

Dynamic threads

Master thread waits for work, forks new threads, and when threads are done, they terminate

Efficient use of resources, but thread creation and termination is time consuming.

Static threads

Pool of threads created and are allocated work, but do not terminate until cleanup.

Better performance, but potential waste of system resources.

Nondeterminism

```
...
printf("Thread %d > my_val = %d\n",
    my_rank, my_x) ;
...
```

Thread 1 > my_val = 19
Thread 0 > my_val = 7

Thread 0 > my_val = 7
Thread 1 > my_val = 19

```
my_val = Compute_val(my_rank) ;
x += my_val ;
```

Time	Core 0	Core 1
0	Finish assignment to my_val	In call to Compute_val
1	Load x = 0 into register	Finish assignment to my_val
2	Load my_val = 7 into register	Load x = 0 into register
3	Add my_val = 7 to x	Load my_val = 19 into register
4	Store x = 7	Add my_val to x
5	Start other work	Store x = 19

Race condition

Critical section

Mutually exclusive

Mutual exclusion lock (mutex, or simply lock)

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val ;
Unlock(&add_my_val_lock);
```

When we launch multiple threads, the OS has control over scheduling
Execution is non-deterministic in parallel programming. Threads can run at different times. x is a shared variable in the example

```
my_val = Compute_val( my_rank);
x += my_val;
```

In order to avoid race condition we have a critical section. We can make the code mutually exclusive. We can place mutual exclusion lock (mutex, or simply lock)

```
my_val = Compute_val (my_rank);
Lock (& add_my_val_lock)
x += my_val;
Unlock (& add_my_val_lock)
```

→

Busywait : we can use busywaiting to specify sequence in the non-deterministic model

busy-waiting

```
my_val = Compute_val ( my_rank ) ;
if ( my_rank == 1)
     while ( !ok_for_1 );  /* Busy-wait loop */
x += my_val ;  /* Critical section */
if ( my_rank == 0)
     ok_for_1 = true ;  /* Let thread 1 update x */
```

Distributed Memory: each core has its own corresponding memory
In order to exchange data, we need specific computation core.

- You usually run distributed memory in multiple servers. Since it is not possible to run multiple threads in multiple servers, we use multiple processes

Distributed Memory

Message-passing

```
char message [100] ;
. . .
my_rank = Get_rank ( ) ;
if ( my_rank == 1) {
     sprintf ( message , "Greetings from process 1" ) ;
     Send ( message , MSG_CHAR , 100 , 0 ) ;
} else if ( my_rank == 0) {
     Receive ( message , MSG_CHAR , 100 , 1 ) ;
     printf ( "Process 0 > Received: %s\n" , message ) ;
}
```

Message passing : passing messages from one core to another.

Input & Output : I/O operations lose performance

- we want to use conventions to not lose too much performance
- in most cases, only a single process/thread will be used for all output to stdout rather than debugging output. Debug output should always include the rank or id of the process/thread that's generating the output

Input and Output

In distributed memory programs, only process 0 will access *stdin*. In shared memory programs, only the master thread or thread 0 will access *stdin*.

In both distributed memory and shared memory programs all the processes/threads can access *stdout* and *stderr*.

However, because of the indeterminacy of the order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout* other than debugging output.

Debug output should always include the rank or id of the process/thread that's generating the output.

Only a single process/thread will attempt to access any single file other than *stdin*, *stdout*, or *stderr*. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.

Performance

Speedup

Number of cores = p

Serial run-time = T_{serial}

Parallel run-time = $T_{parallel}$

linear speedup

$$T_{parallel} = T_{serial} / p$$

$$S, \text{speedup} = \frac{T_{serial}}{T_{parallel}}$$

p = num of processors

$$E, \text{effecien cy of a parallel program} = \frac{S}{p} = \frac{T_{serial}}{p * T_{parallel}}$$

when we increase p, effeciency is not necessarily increasing.
As with more p, there is usually more overhead so there is not a linear increase.

Effeciency decreases with p increase when
- less data in each core - more core communication

more data will give you a higher effeciency.

$$T_{parallel} = T_{serial} / p + T_{overhead}$$

T_{serial}/p: can be optimized

$T_{overhead}$: cannot be optimized so it remains unchanged

ex: $E = \frac{20}{\frac{18/p + 2}{p}} = \frac{20}{18 + 2p}$ As p increases, effeciency decreases

as $p \to \infty$, $\frac{18}{p} \to 0$, so we have $\frac{20}{2} = 10$ speed up

you can only go 10x faster. You can't go 10x faster

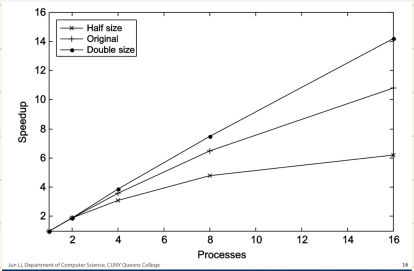
Speedups and efficiencies of a parallel program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

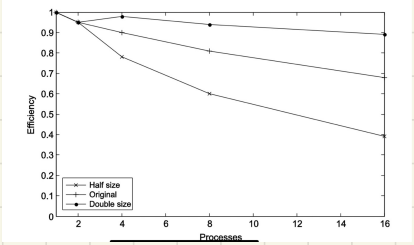
Speedups and efficiencies of parallel program on different problem sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Speed-up



Efficiency



Amdahl's Law

Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.

Scalability

If we can increase the problem size & it maintains effeciency E then the problem is scalable.

① Strongly scalable:
- keep problem size same
- increase threads & processes

② Weakly scalable
- increase problem size in proportion to thread/process increase

Scalability

Suppose that we increase the number of processes/threads, if we can find a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is scalable.

If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is strongly scalable.

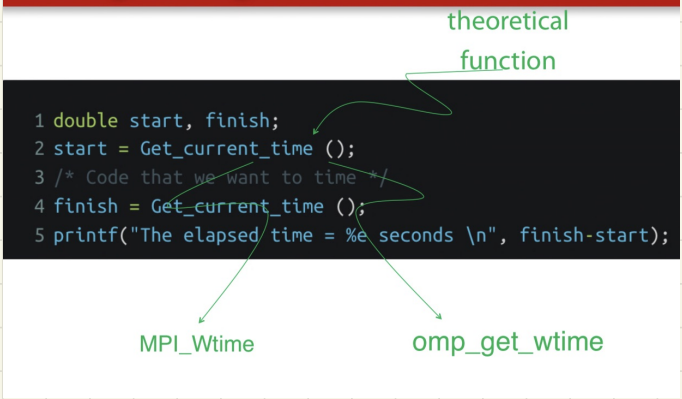
If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/ threads, the problem is weakly scalable.

Taking timing

- wall clock time
  - we may use it for tracking performance

we can have problem tracking time for distributed computing as each can have their own variable for tracking time

Taking Timings




```
1 private double start, finish;
2 start = Get_current_time ();
3 /* Code that we want to time */
4 finish = Get_current_time ();
5 printf ("The elapsed time = %e seconds \n", finish-start);
```

private is a part of the process

```
1 shared double global_elapsed;
2 private double my_start, my_finish, my_elapsed;
3 ...
4 /* Synchronize all processes/threads */
5 Barrier();
6 my_start = Get_current_time();
7
8 /* Code that we want to time */
9 ...
10
11 my_finish = Get_current_time();
12 my_elapsed = my_finish - my_start;
13
14 /* Find the max across all processes/threads */
15 global_elapsed = Global_max(my_elapsed);
16 if (my_rank == 0)
17     printf ("The elapsed time = %e seconds \n", global_elapsed);
```

global time elasped

Parallel Program Design

March 1, 2022

There is no one single way to have the best performance

Foster's methodology

1. Partitioning: divide the computation to be performed and the data operated on by the computation into small tasks.

   The focus here should be on identifying tasks that can be executed in parallel.

We are going to partition it into small tasks

Foster's methodology

2. Communication: determine what communication needs to be carried out among the tasks identified in the previous step.

ex: input & output between different cores

Foster's methodology

3. Agglomeration or aggregation: combine tasks and communications identified in the first step into larger tasks.

   For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

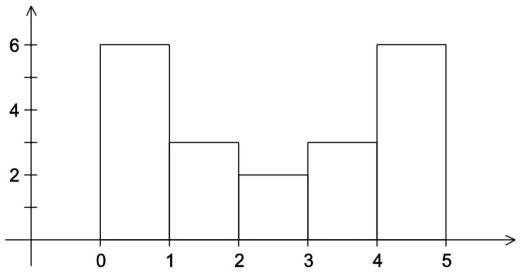
Foster's methodology

4. Mapping: assign the composite tasks identified in the previous step to processes/threads.

   This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

Example - histogram

1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



we can try to make a serialized program to a parallelized program

Serial program - input

1. The number of measurements: data_count
2. An array of data_count floats: data
3. The minimum value for the bin containing the smallest values: min_meas
4. The maximum value for the bin containing the largest values: max_meas → defines range of numbers
5. The number of bins: bin_count

Serial program - output

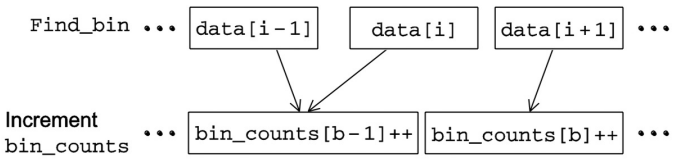
1. bin_maxes : an array of bin_count floats – array of floating point number which defines the max
2. bin_counts : an array of bin_count ints – height of bar corresponding to each bin

```
for (b = 0; b < bincount; b++)
    bin_maxes[b] = min_meas + bin_width*(b+1);
```

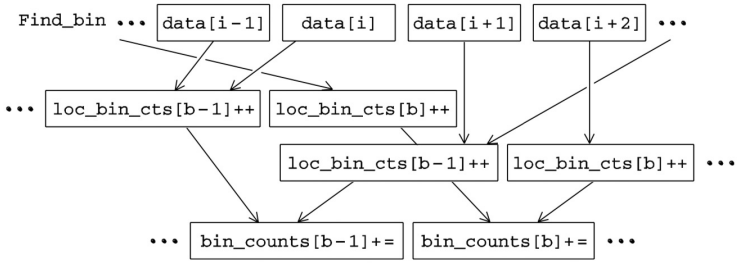
```
for (i = 0; i < datacount; i++) {
    bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);
    bin_counts[bin]++;
}
```

Now to convert it to a parallel program we use Forster's Terminology

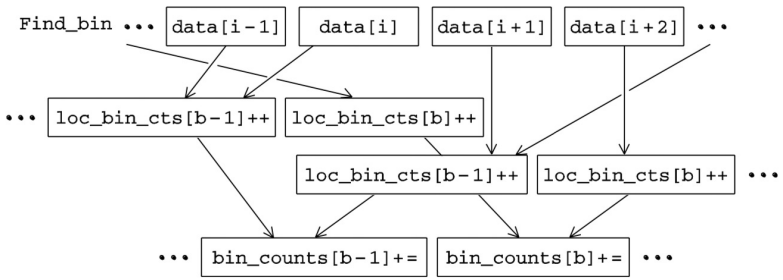
First two stages of Foster's Methodology



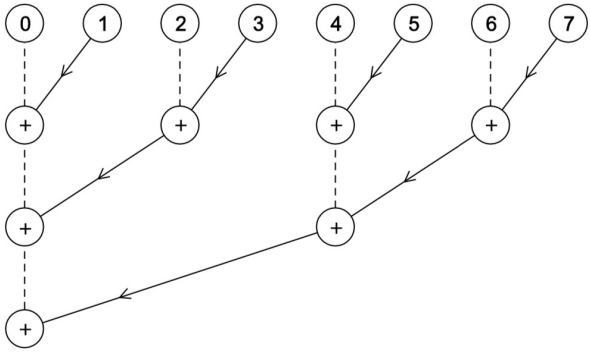
Alternative definition of tasks and communication



Alternative definition of tasks and communication



Adding the local arrays



Concluding Remarks (1)

Serial systems

The standard model of computer hardware has been the von Neumann architecture.

Parallel hardware

Flynn's taxonomy.

Parallel software

We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching.

SPMD programs.

Concluding Remarks (2)

Input and Output

We'll write programs in which one process or thread can access stdin, and all processes can access stdout and stderr.

However, because of nondeterminism, except for debug output we'll usually have a single process or thread accessing stdout.

Concluding Remarks (3)

Performance

Speedup

Efficiency

Amdahl's law

Scalability

Parallel Program Design

Foster's methodology

Parallel Programming with MPI

Roadmap

MPI Overview

Process Model & Language Bindings

Messages & Point-to-Point Communication

Nonblocking Communication

Collective Communication

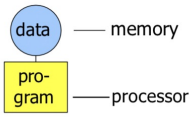
Error Handling

Shared Memory

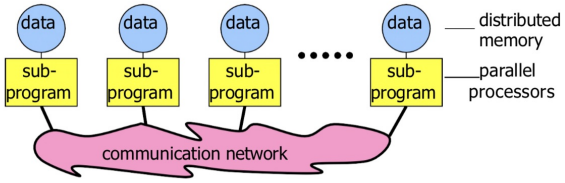
Parallel I/O

The Message-Passing Programming Paradigm

- Sequential Programming Paradigm

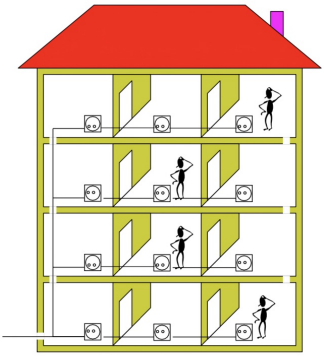


- Message-Passing Programming Paradigm



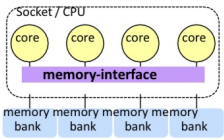
Analogy: Electric Installations in Parallel

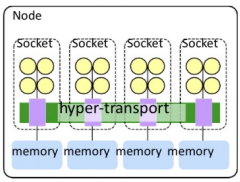
- MPI sub-program
  = work of one electrician on one floor
- data
  = the electric installation
- MPI communication
  = real communication to guarantee that the wires are coming at the same position through the floor



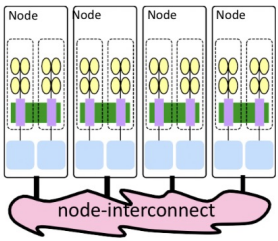
Parallel hardware architectures – other ways MPI is also used

shared memory





distributed memory



Socket/CPU

→ memory interface

UMA (uniform memory access) SMP (symmetric multi-processing) All cores connected to all memory banks with same speed

Node

→ hyper-transport

ccNUMA (cache-coherent non-uniform memory access)

Shared memory programming is possible

Performance problems:

- Threads should be **pinned** to the physical sockets
- **First-touch** strategy is needed to minimize remote memory access

Cluster

→ node-interconnect

NUMA (non-uniform memory access)

!! fast access only on its own memory **!!**

Many programming options:

- Shared memory / symmetric multi-processing inside of each node
- distributed memory parallelization on the node interconnect
- **Or simply one MPI process on each core**

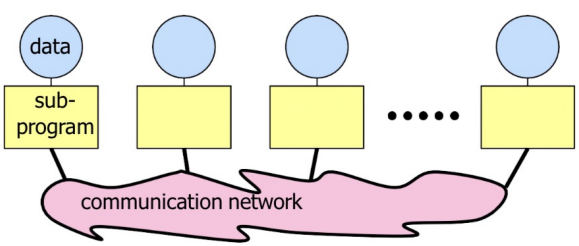
Shared memory programming with OpenMP

MPI works everywhere

The Message-Passing Programming Paradigm

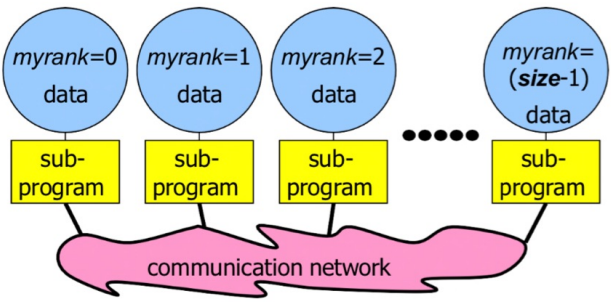
MPI Overview

- Each processor in a message passing program runs a ***sub-program:***
  - written in a conventional sequential language, e.g., C, Fortran, or Python
  - typically the same on each processor (SPMD),
  - the variables of each sub-program have
    - **the same name**
    - **but different locations (distributed memory) and different data!**
    - **i.e., all variables are private**
  - communicate via special send & receive routines (***message passing***)



Data and Work Distribution

- the value of ***myrank*** is returned by special library routine
- the system of ***size*** processes is started by special MPI initialization program (mpirun or mpiexec)
- all distribution decisions are based on ***myrank***
- i.e., which process works on which data




```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{  int n;   double result;
   int my_rank, num_procs;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

   if (my_rank == 0)
   { printf("Enter the number of elements (n): \n");
     scanf("%d",&n);
   }
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

   result = 1.0 * my_rank * n;
   printf("I am process %i out of %i handling the %ith part of n=%i elements,result=%f\n",
                  my_rank,   num_procs,      my_rank,      n,                result);

   if (my_rank != 0)
   { MPI_Send(&result,1,MPI_DOUBLE,0,99,MPI_COMM_WORLD);
   }
   else
   { int rank;
     printf("I'm proc 0: My own result is %f \n",result);
     for (rank=1; rank<num_procs; rank++)
     {
        MPI_Recv(&result,1,MPI_DOUBLE,rank,99,
       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("I'm proc 0: received result of
          process %i is %f \n", rank, result);
     }
   }
   MPI_Finalize();
}
```

Compiled, e.g., with: `mpicc first-example.c`
Started, e.g., with: `mpiexec -n 4 ./a.out`
Then, this code is running 4 times in parallel !

application-related data

MPI-related data

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

reading the application data *n* from stdin only by process 0

process 0 is sender, all other processes are receivers

broadcasting the content of variable *n* in process 0 into variables *n* in all other processes

doing some **application work** in each process

send to process 0

sending some results from all processes (except 0) to process 0

Process 0: receiving all these messages and, e.g., printing them

receiving the message from process *rank*

```
Enter the number of elements (n): 100

I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
I'm proc 0: My own result is 0.0
I'm proc 0: received result of process 1 is 100.0
I'm proc 0: received result of process 2 is 200.0
I'm proc 0: received result of process 3 is 300.0
```

Jun Li, Department of Computer Science, CUNY Queens College

General idea of how an MPI program works

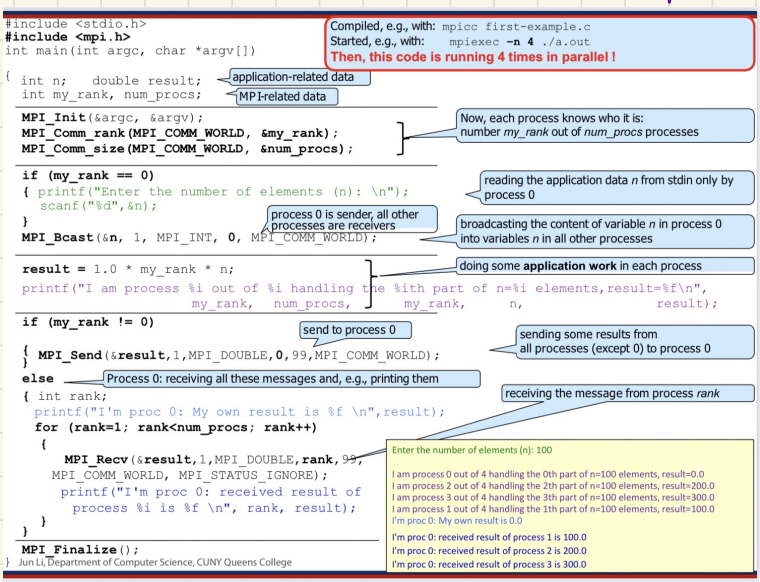
March 3, 2022

We're looking at the equivalent Python version

```
mpirun -n 4 echo "hello"
» hello
» hello
» hello
» hello
```

For different versions of MPI the command will be different. He prefers we use Python

MPI continued



Run mpiexec -n 4 python first-example.py

```python
from mpi4py import MPI

# application-related data
n = None
result = None

comm_world = MPI.COMM_WORLD
# MPI-related data
my_rank = comm_world.Get_rank() # or my_rank = MPI.COMM_WORLD.Get_rank()
num_procs = comm_world.Get_size() # or ditto ...

if (my_rank == 0):
    # reading the application data "n" from stdin only by process 0:
    n = int(input("Enter the number of elements (n): "))

# broadcasting the content of variable "n" in process 0
# into variables "n" in all other processes:
n = comm_world.bcast(n, root=0)

# doing some application work in each process, e.g.:
result = 1.0 * my_rank * n
print(f"I am process {my_rank} out of {num_procs} handling the {my_rank}ith part of n={n}
  elements, result={result}")

if (my_rank != 0):
    # sending some results from all processes (except 0) to process 0:
    comm_world.send(result, dest=0, tag=99)
else:
    # receiving all these messages and, e.g., printing them
    rank = None
    print(f"I'm proc 0: My own result is {result}")
    for rank in range(1,num_procs):
        result = comm_world.recv(source=rank, tag=99)
        print(f"I'm proc 0: received result of process {rank} is {result}")
```

Access

MPI: addresses are the ranks of the MPI processes (sub-programs)

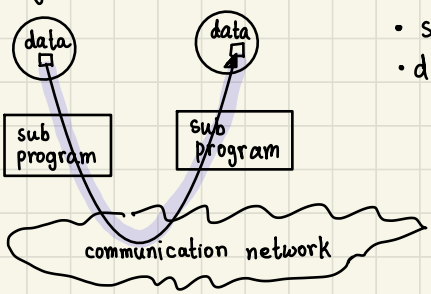
- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
  - mail box
  - phone line
  - fax machine
  - etc.
- MPI:
  - sub-program must be linked with an MPI library
  - sub-program must use include file of this MPI library
  - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool

- unique
- easy to remember
- follows a range of 0 to n-1

Access

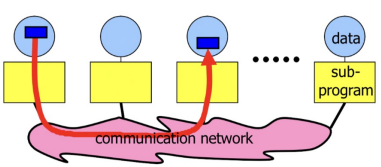
- A sub program needs to be connected to a message passing system
- MPI:
  - sub-program must be linked with an MPI library
  - sub-program must use include file of this MPI library
  - the total program (i.e. all sub-programs of the program) must be started with the MPI startup tool

Messages



- source data size - how much data will you send
- destination buffer size - how much space is available to receive data

Messages



- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
  - sending process
  - source location
  - source data type
  - source data size
  - receiving process
  - destination location
  - destination data type
  - destination buffer size

i.e., the ranks

basic or derived datatypes

Addressing

- Messages need to have addresses to be sent to.
- Addresses are similar to:
  - mail addresses
  - phone number
  - fax number
  - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)

Addressing

MPI: addresses are the ranks of the MPI processes (sub-programs)

- unique
- easy to remember
- follows a range of 0 to n-1

- Point to point communication - easiest way to send
  1. Synchronous send
  2. Asynchronous send through buffer

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
  - synchronous send
  - buffered = asynchronous send

Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.

Buffered = Asynchronous Sends

- Only know when the message has left.

Blocking operations

- Non-blocking operations
  - Non-blocking procedures are not the same as sequential subroutine calls
    -the operation may continue while the application executes the next statements
  - You need to specify to the program that the resources have been freed.

Blocking Operations

- Operations are activities, such as
  - sending (a message)
  - receiving (a message)
- Some operations may **block** until another process acts:
  - synchronous send operation **blocks until** receive is posted;
  - receive operation **blocks until** message was sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

Nonblocking Operations

Nonblocking operations consist of:

- A nonblocking procedure call: it returns immediately and allows the sub-program to perform other work
- At some later time the sub-program must ***test*** or ***wait*** for the completion of the nonblocking operation

- All nonblocking procedures must have a matching wait (or test) procedure. (Some system or application resources can be freed only when the nonblocking operation is completed.)
- A nonblocking procedure immediately followed by a matching wait is equivalent to a blocking procedure.
- Nonblocking procedures are not the same as sequential subroutine calls:
  - the operation may continue while the application executes the next statements!

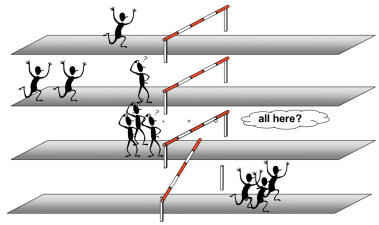
- Collective Communications
  - Several processes involved at once
  - Can be built out of point-to-point communication

Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms.
- Can be built out of point-to-point communications.

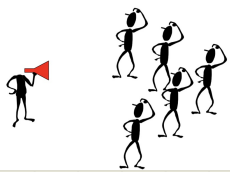
Barriers

- Synchronize processes.



Broadcast

- A one-to-many communication.



Reduction Operations

- Combine data from several processes to produce a single result.

- Reduction operation: combine data from several processes to produce a single result
- Barriers
  - useful if we want to synchronize different processes
  - we specify a barriers, we ask a process to wait at a certain point until all needed processes reach the barrier & then the process can start again

  Ex: this is useful when we may want to load data but not measure that as a part of runtime. we can set up barrier until all the data is loaded to all the processes.

Parallel File I/O

The I/O operations are hard to be parallelized. With more cores it takes more time.

With serial I/O in a parallel program, there is:

1) waste of resources
2) negative side effect for users

Parallel computation → need for parallel I/O
→ do parallel I/O

Process Model & Language Binding - how to setup for MPI program

MPI function format

- you don't want to use object-serialization for send & receive as it has a low performance. You want to use Python.

Initializing MPI : must be done at beginning

Exiting MPI :

- all communication terminated
- you should not expect too much work after this, you might only have return

Starting the MPI program

- mpirun -np number of processes ./executable
- mpiexec -n number of processes ./executable

Communicator MPI_COMM_WORLD

- all sub-programs combine here & communicate

Header files

C
- C / C++

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

Python
- Python

```
from mpi4py import MPI
```

MPI_Init() must be called before any other MPI routine
(only a few exceptions, e.g., MPI_Initialized)

C
- int MPI_Init(int *argc, char ***argv)

```
#include <mpi.h>
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  ....
```

MPI-2.0 and higher:
Also
MPI_Init(NULL, NULL);

Python
- # MPI.Init()

This call is not needed, because automatically called at the import of MPI at the begin of the program

```
from mpi4py import MPI
# MPI.Init() is not needed
....
```

MPI Function Format

In C and Python: case sensitive

C
- C / C++: error = MPI_Xxxxxx(parameter, ...);
  MPI_Xxxxxx(parameter, ...);

Python
- Python: result_value_or_object = input_mpi_object.mpi_action(parameter, ...)

direct communication of numPy arrays (like in C)	Or with object-serialization:
comm_world = **MPI.COMM_WORLD**	
comm_world.**S**end((snd_buf, ...), ...)	comm_world.**s**end(snd_buf, ...)
comm_world.**R**ecv((rcv_buf, ...), ...)	rcv_buf = comm_world.**r**ecv(...)

Mixed cases:

- MPI procedures in C — MPI_Xxx_mixed
- MPI type declaration — MPI_Xxx_mixed
- MPI constant — MPI_XXX_UPPER

Exiting MPI

C
- C/C++: int MPI_Finalize()

Python
- Python: # MPI.Finalize()

This call is not needed, because automatically called at the end of the program

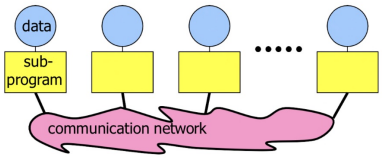
- **Must** be called last by all processes.
- User must ensure the completion of all pending communications (locally) before calling finalize
- After MPI_Finalize:
  - Further MPI-calls are forbidden
  - Especially re-initialization with MPI_Init is forbidden
  - **May** abort the calling process if its rank in MPI_COMM_WORLD is ≠0

Handles

- MPI object
-

Starting the MPI Program

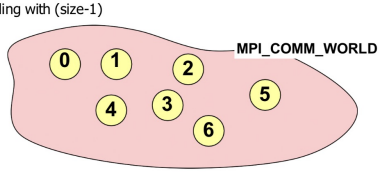
- Start mechanism is implementation dependent
- mpirun –np ***number_of_processes*** ./***executable*** (most implementations)
- mpiexec –n ***number_of_processes*** ./***executable*** (with MPI-2 and later)



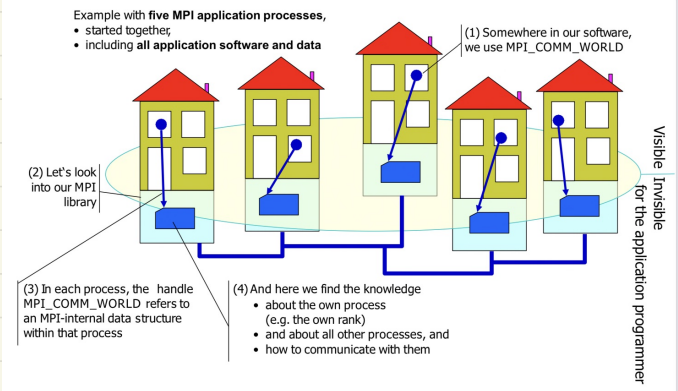
- The parallel MPI processes exist at least after MPI_Init was called.

Communicator MPI_COMM_WORLD

- All processes (= sub-programs) of one MPI program are combined in the **communicator MPI_COMM_WORLD**.
- MPI_COMM_WORLD is a predefined **handle** in
  - mpi.h
  - Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



Handles refer to internal MPI data structures



Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - **predefined constants** in C include file **mpi.h** or **MPI module of mpi4py**
    - Example: MPI_COMM_WORLD or MPI.COMM_WORLD
    - Can be used in initialization expressions or assignments.
    - They are link-time constants, i.e., need not to be compile-time constants.
  - **values returned** by some MPI routines, to be stored in variables, that are defined as
    - C: special MPI typedefs, e.g., MPI_Comm sub_comm;
    - Python: Type of object defined by the creating function, e.g., sub_comm = MPI.COMM_WORLD.Split(…)
- Handles refer to internal MPI data structures

C
Python

CS381-16 March 8, 2022

Overview of MPI

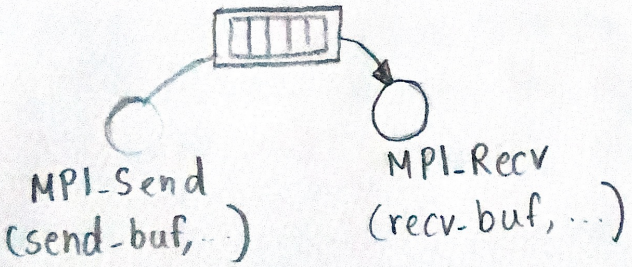
Handles
- identify MPI objects
- handles refer to internal MPI data structures

Uses of MPI
- Rank: address for different processes

Python
```
rank = comm.Get_rank()
```

Size
Python `size = comm.Get_size()`

Messages & Point-to-Point Communication



There is a question of which protocol should be used
Protocols are
1. Buffered protocol
2. Synchronous protocol
3. Direct protocol
4. Automatic decision

Message format

MPI basic data types
- in Python all data types can be used eg. MPI.FLOAT

Point to Point Communication

Sending a Message

- In C we shouldn't specify the buffer type in the signature since it could be any type

Receiving a Message

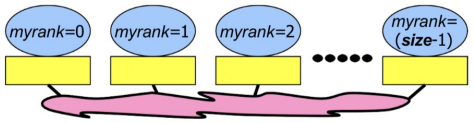
- Format in send & receive is pretty similar

Rank

- The rank identifies different processes.
- The rank is the basis for any work and data distribution.

C
- C/C++: int MPI_Comm_rank(MPI_Comm comm, int *rank)

Python
- Python: *rank* = comm.Get_rank()



Size

- How many processes are contained within a communicator?

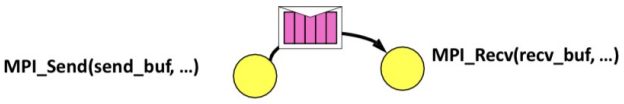
C
- C/C++: int MPI_Comm_size(MPI_Comm comm, int *size)

Python
- Python: *size* = comm.Get_size()

Fortran & C: Interface definitions
- On these slides & in the MPI standard
- You have to write the corresponding procedure **calls**
- E.g., in C:
  MPI_Comm_size (MPI_COMM_WORLD, &size);

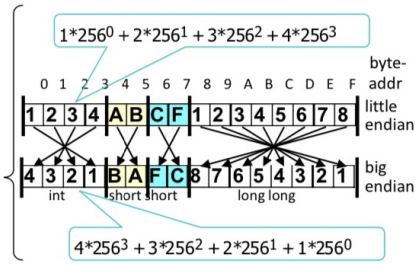
Python: Mix of usage of the interface and typed argument list
- See MPI for Python (mpi4py.github.io), and
  MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io), and
  The API reference (mpi4py.github.io/apiref/index.html)

Major decisions: performance and functionality



Important questions / decisions

- How to address the destination process?
- Which message content?
  - How to handle strided data?
  - Data conversion in inhomogeneous cluster



- Which protocol?
  - When must the **send** be completed?
    - **As soon as possible → buffered protocol → low latency ☺ / bad bandwidth ☹**
    - **Only after the receive is called → synchronous protocol → sending process blocked by receiver → high latency ☹ / good bandwidth ☺**
    - **Application guarantees that corresponding receive is already called → direct protocol → sending process is not blocked / good bandwidth ☺**
    - **Automatic decision → may be blocked / low latency ☺ / good bandwidth ☺**

Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic datatype.
  - Derived datatypes
- Derived datatypes can be built up from basic or derived datatypes.
- Datatype handles are used to describe the type of the data in the memory.

Example: message with 5 integers

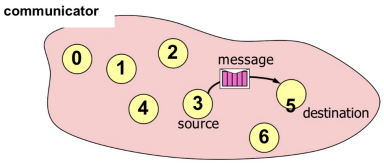
2345	654	96574	-12	7676

Python: Python: messages can be stored in

- Objects → using send(...), recv(...), ... mpi4py routines (Lower-case methods) → slow object serialization
- Buffers as numPy arrays → using Send(...), Recv(...), ... → fast communication (Upper-case methods)

Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
- Processes are identified by their ranks in the communicator.



Receiving a Message

C

- C/C++: int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Python

- Python: comm.Recv(buf, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)
  obj = comm.recv(buf=None, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)
  buf is only a temporary buffer, deprecated since version 3.0.0

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in status.
- On can pass MPI_STATUS_IGNORE instead of a status argument.
- Output arguments are printed blue-cursive.
- **Message matching rule:** receives only if comm, source, and tag match.
- Python: Send requires that the matching receive is a Recv / ditto for send and recv

count, datatype is **not** part of this matching rule

Sending a Message

C

- C/C++: int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Python

- Python: comm.Send(buf, int dest, int tag=0)
  comm.send(obj, int dest, int tag=0)

- buf is the starting point of the message with count elements, each described with datatype.
- dest is the rank of the destination process within the communicator comm.
- tag is an additional nonnegative integer piggyback information, additionally transferred with the message.
- The tag can be used by the program to distinguish different types of messages.
- Python: – buf must implement the Python buffer protocol, e.g., numPy arrays
  - buf can be buf or (buf, datatype) or (buf, count, datatype)
  - with C datatypes in Python syntax, e.g., MPI.INT, MPI.FLOAT, ...
  – obj is any Python object that can be serialized with the pickle method

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Type matching:

```
float sndbuf[n];                          float rcvbuf[n];
MPI_Send(sndbuf, n, MPI_FLOAT,...)        MPI_Recv(rcvbuf, n, MPI_FLOAT,...)
```

  1. Send-buffer's C/Python type must match with the send datatype handle Send
  2. datatype handle must match with the receive datatype handle
  3. Receive datatype handle must match with receive-buffer's C/Python type

- Tags must match → typical usage: **different tags for different data**

```
#define TAG_velocity 111                  #define TAG_velocity 111
MPI_Send( velocity_sndbuf, ... TAG_velocity, ...)   MPI_Recv( velocity_rcvbuf, ... TAG_velocity, ...)
```

  → The velocity message will never be received in, e.g., a temperature array

- Receiver's buffer must be large enough.

Wildcarding

- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's *status* parameter.

- With info assertions **New in MPI-4.0**
  - "mpi_assert_no_any_source" = "true" and/or
  - "mpi_assert_no_any_tag" = "true"

  stored on the communicator using MPI_Comm_set_info(),
  - an MPI application can tell the MPI library that it will never use MPI_ANY_SOURCE and/or MPI_ANY_TAG on this communicator

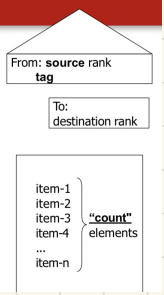
  → may enable lower latencies.
- Other assertions:
  - "mpi_assert_exact_length" = "true" → receive buffer must have exact length
  - "mpi_assert_allow_overtaking" = "true" → message order need not to be preserved

Communication Envelope

- Envelope information is returned from MPI_RECV in *status*.

C
- C/C++: MPI_Status status;
  status.MPI_SOURCE
  status.MPI_TAG
  status.MPI_ERROR

Python
- Python: status.Get_source()
  status.Get_tag(),
  status.Get_error()…



Receive Message Count

C
- C/C++: int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

Python
- Python: *count* = status.Get_count(Datatype datatype=BYTE)

Caution:

```
buf = np.zeros((100,), dtype=np.double)
comm.Send((buf, 5, MPI.DOUBLE), ....)
comm.Recv((buf, 100, MPI.DOUBLE), ...., status)
count = status.Get_count(MPI.DOUBLE) # → 5
count = status.Get_count() # → 40
```

Communication Modes

- Send communication modes:
  - synchronous send → MPI_**S**SEND
  - buffered [asynchronous] send → MPI_**B**SEND
  - standard send → MPI_**SEND**
  - Ready send → MPI_**R**SEND
- Receiving all modes → MPI_**RECV**

Communication Modes — Definitions

Sender mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Standard send **MPI_SEND**	Either synchronous or buffered	uses an internal buffer
Ready send **MPI_RSEND**	May be started **only** if the matching receive is already posted!	highly dangerous!
Receive **MPI_RECV**	Completes when a message has arrived	same routine for all communication modes

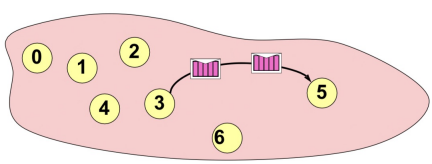
Rules for the communication modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode

  → all risks of synchronous send
- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting —> idle time
  - high latency / best bandwidth
- Buffered send (**MPI_BSEND**)
  - low latency / bad bandwidth
- Ready send (**MPI_RSEND**)
  - use **never**, except you have a *200% guarantee* that Recv is already called in the current version and all future versions of your code,
  - may be the fastest

Message Order Preservation

- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other.**
- This is true even for non-synchronous sends.



- If both receives match both messages, then the order is preserved.

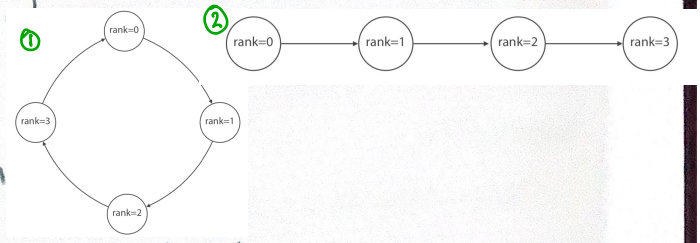
March 10, 2022

MPI: Non-blocking Communication

Blocking: you call a function & you stop there until the function stops executing

Example of why you need non-blocking communication

1. Ring Communication:
   1. Cyclic boundary condition
   2. Non-cyclic boundary condition



Blocking Routines → Risk of deadlocks & serialization.

1. For cyclic boundary:

```
MPI_Send(..., right_rank, ...)
MPI_Recv(..., left_rank, ...)
```

worst case: send function chooses to be synchronous.
If the MPI library chooses the synchronous protocol
i.e. MPI_send waits until MPI_Recv → There is deadlock.

- If you choose the buffer send you can place it so that there is asynchronous protocol

2. For non-cyclic boundary:


```
if (my_rank < size-1)
    MPI_Send(..., right, ...)
if (my_rank > 0)
    MPI_Recv(..., left, ...)
```

different from slide, this is correct

If the MPI library chooses the synchronous protocol then there is serialization ↯ → messages delivered sequentially

Blocking Routines -> Risk of Deadlocks & Serializations

For cyclic boundary:

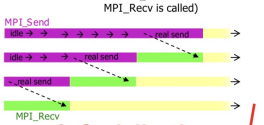
```
MPI_Send(..., right_rank, ...)
MPI_Recv(   ..., left_rank,  ...)
```

→ **Deadlock**

(If the MPI library chooses the synchronous protocol, i.e. MPI_Send waits until MPI_Recv is called)

For non-cyclic boundary:

```
if (myrank < size-1)
  MPI_Send(..., left, ...);
if (myrank > 0)
  MPI_Recv(    ..., right,  ...);
```

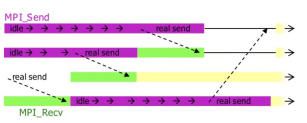


→ **Serialization**

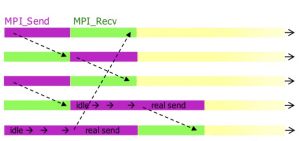
(If the MPI library chooses the synchronous protocol)

Cyclic communication – other bad ideas

```
if (myrank < size-1)
  { MPI_Send(..., left, ...);
  MPI_Recv(    ..., right,  ...);
}else {
  MPI_Recv(    ..., right,  ...);
  MPI_Send(..., left, ...);
}
```




```
if (myrank%2 == 0)
  { MPI_Send(..., left, ...);
  MPI_Recv(    ..., right,  ...);
}else {
  MPI_Recv(    ..., right,  ...);
  MPI_Send(..., left, ...);
}
```



→ **Serialization**

(If the MPI library chooses the synchronous protocol)

Non-Blocking Communications

Separate communication into **three phases**:

- Initiate nonblocking communication
  - returns immediately
  - routine name starting with MPI_**I**...

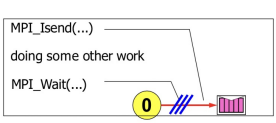
  → it is local,
  i.e., it returns independently of any other process' activity

"I" stands for
- Immediately (=local) and
- Incomplete (=nonblocking[1])

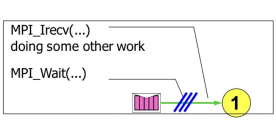
- Do some work (perhaps involving other communications?)
- Wait for nonblocking communication to **complete**, i.e.,
  - the send buffer is read out, or
  - the receive buffer is filled in

Non-Blocking Examples

- Nonblocking **send**



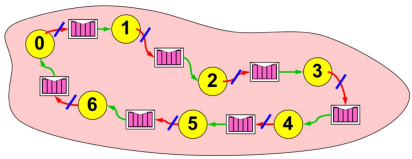
- Nonblocking **receive**



/// = waiting until operation locally completed

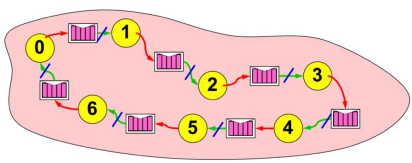
Non-Blocking Send

- Initiate nonblocking send
  → in the ring example: Initiate nonblocking send to the right neighbor
- Do some work:
  → in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for nonblocking send to complete

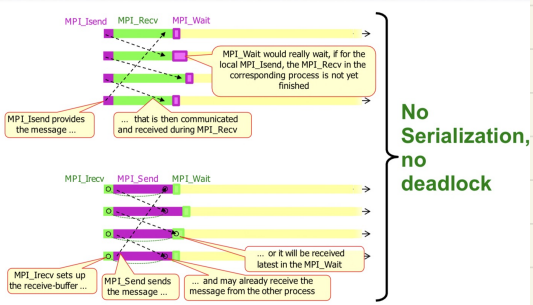


Non-Blocking Receive

- Initiate nonblocking receive
  → in the ring example: Initiate nonblocking receive from left neighbor
- Do some work:
  → in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for nonblocking receive to complete



Timelines for both solutions



Request Handles

Request handles

- are used for nonblocking communication
- **must** be stored in local variables
  - in C/C++: MPI_Request
  - in Python: automatically

C Python

the value
- **is generated** by a nonblocking communication routine
- **is used** (and freed) in the MPI_WAIT routine

Nonblocking Synchronous Send

C

- C/C++: MPI_Issend(&buf, count, datatype, dest, tag, comm, [OUT] &*request_handle*);

  MPI_Wait([INOUT] &request_handle, &*status*);

Python

- Python: *request* = comm_world.Issend(…) / status = MPI.Status(); request.Wait(*status*)

- buf must not be modified between Issend and Wait (in all progr. languages)
  (In MPI-2.1, this restriction was stronger: "should not access", see MPI-2.1, page 52, lines 5-6)
- "Issend + Wait directly after Issend" is equivalent to blocking call (Ssend)
- Nothing returned in status (because send operations have no status)

Nonblocking Receive

C

- C/C++: MPI_Irecv (*buf*, count, datatype, source, tag, comm, [OUT] &*request_handle*);

  MPI_Wait([INOUT] &request_handle, &*status*);

Python

- Python: *request* = comm_world.Irecv(…) / status = MPI.Status(); request.Wait(*status*)

- buf must not be used between Irecv and Wait (in all progr. languages)
- Message status is returned in Wait

Blocking and Non-Blocking

- Send and receive can be blocking or nonblocking.
- A blocking send can be used with a nonblocking receive, and vice-versa.
- Nonblocking sends can use any mode
  - standard – MPI_ISEND
  - synchronous – MPI_ISSEND
  - buffered – MPI_IBSEND
  - ready – MPI_IRSEND
- Synchronous mode affects completion, i.e. MPI_Wait / MPI_Test, not initiation, i.e., MPI_I….

Completion

C

- C/C++:
  MPI_Wait(&request_handle, &*status*);
  MPI_Test(&request_handle, &*flag*, &*status*);

Python

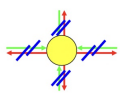
- Python:
  status = MPI.Status(); request_handle.Wait(*status*)
  status = MPI.Status(); *flag* = request_handle.Test(*status*)

- one must
  - WAIT or
  - loop with TEST until request is completed,
    i.e., flag == non-zero (C) or True (Python)

Multiple Non-Blocking Communications

You have several request handles:

- Wait or test for completion of **one** message
  - **MPI_Waitany / MPI_Testany**
- Wait or test for completion of **all** messages
  - **MPI_Waitall / MPI_Testall *)**
- Wait or test for completion of **at least one** messages
  - **MPI_Waitsome / MPI_Testsome *)**



*) Each status contains an additional error field.
This field is only used if MPI_ERR_IN_STATUS is returned (also valid for send operations).

Other MPI features: Send-Receive in one routine

- MPI_Sendrecv & MPI_Sendrecv_replace
  - Combines the triple "MPI_Irecv + Send + Wait" into one routine

New in MPI-4.0

- Nonblocking MPI_Isendrecv & MPI_Isendrecv_replace
  - Whereas blocking MPI_Sendrecv was used to prevent
    - **serializations and**
    - **deadlocks,**
  - the nonblocking MPI_Isendrecv can be used, e.g., to parallelize the existing communication calls in multiple directions
    → e.g., to minimize idle times if only some neighbors are delayed

Cyclic communication - other bad ideas

For odd-even one:

the last three have to wait for each other so there is serialization

Non-blocking Communication

There are three phases

Examples:

- Non-blocking send MPI_Isend
- Non-blocking receive

We can take advantage of the non-blocking communication by carrying out other processes meanwhile.

Non-blocking send

All the messages can be sent at once by coupling non-blocking send & non-blocking receive

All the sends don't have to wait for the other sends to complete

Non-blocking Receive

All cores are initialized for receive

As it is non-blocking receive & it is paired w/ send

In the foreground it is doing a send, in the background it is doing a receive

Timeline for both solutions

① MPI-Wait would really wait if for the local MPI_Isend, the MPI-Recv in the corresponding process is not yet finished
→ In non-blocking send & blocking receive.

② MPI non-blocking receive & blocking send

In these two solutions we have
- no serialization
- no deadlock

Non-blocking synchronous send

- A wait directly after a non-blocking send is blocking

Non-blocking receive

Blocking & Non-blocking
- Send & receive can be blocking or non-blocking
- A blocking send can be used w/a non-blocking receive & vice-versa
- Synchronous mode affects completion

Completion

One must
- WAIT or
- loop with TEST until request is completed
  i.e. flag == non-zero (C) or True (Python)

Multiple Non-blocking Communications

Other MPI features : Send-Receive in one routine
MPI_Sendrecv : uses separate buffers for send & receive
MPI_Sendrecv-replace : uses the same buffer for send & receive

Performance Options

In terms of performance, there is no specific fastest method by MPI as it might be different based on the environment.

Use cases for non-blocking operations

- To prevent serializations & deadlocks
- Real overlapping of
  - Several communications
  - communication & computation

Collective Communication

Performance options

Which is the fastest neighbor communication?

- MPI_Irecv + MPI_Send
- MPI_Irecv + MPI_Isend
- MPI_Isend + MPI_Recv
- MPI_Isend + MPI_Irecv
- MPI_Sendrecv

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming but efficiency of MPI application-programming is not portable!

Use cases for nonblocking operations

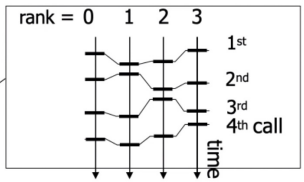
- To prevent serializations and deadlocks
  (as if overlapping of communication with other communication)
- Real overlapping of
  - several communications
  - communication and computation

Internally: tree based algorithm

March 15, 2022

Characteristics of Collective Communication

- Collective action over a communicator.
- All process of the communicator must communicate, i.e., must call the collective routine.
- On a given communicator, the n-th collective call must match on all processes of the communicator.



- In MPI-1.0 – MPI-2.2, all collective operations are blocking. Nonblocking versions since MPI-3.0.
- No tags.

very important

- For each message, the amount of data sent must exactly match the amount of data specified by the receiver

  → It is forbidden to provide receive buffer count arguments that are too long (and also too short, of course)

  Exception with Python (mpi4py): if a buffer argument represents #processes of messages (e.g. snd_buf in comm.Scatter) and the argument count is to be derived from the buffer argument (i.e. is not explicitly defined in the argument list), then this count argument is derived from the inferred number of elements of the buffer divided by the size of the communicator.

e.g., when passing snd_buf, or (snd_buf, datatype).
For buffer options such as BufSpec, BufSpecV, …, see e.g. mpi4py/typing.pyi at master · mpi4py/mpi4py (github.com)

Barrier Synchronization

C

- C/C++: int MPI_Barrier(MPI_Comm comm)

Python

- Python: comm.Barrier() or comm.barrier()

- MPI_Barrier is normally never needed:
  - all synchronization is done automatically by the data communication:
    - **a process cannot continue before it has the data that it needs.**
  - if used for debugging:
    - **please guarantee, that it is removed in production.**
  - for profiling: to separate time measurement of
    - **Load imbalance of computation [MPI_Wtime(); MPI_Barrier(); MPI_Wtime()]**
    - **communication epochs [MPI_Wtime(); MPI_Allreduce(); …; MPI_Wtime()]**

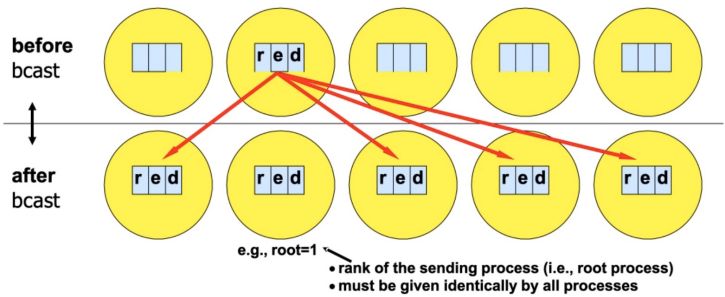
Broadcast

C

- C/C++: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

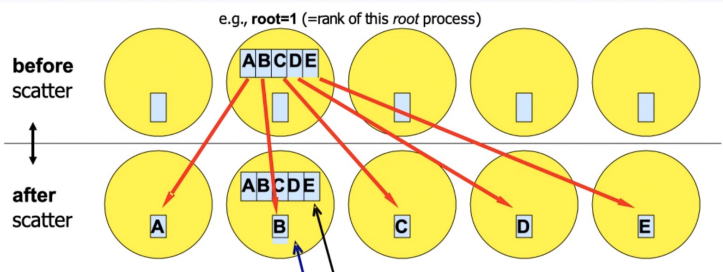
Python

- Python: comm.Bcast(buf, int root=0) or comm.bcast(obj, int root=0)



Example:
MPI_Bcast(buf, 3, MPI_CHAR, 1, MPI_COMM_WORLD);

Scatter



C

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

Python

comm.Scatter(sendbuf *or* None, recvbuf, int root=0)
recvobj = comm.scatter(sendobj *or* None, int root=0)

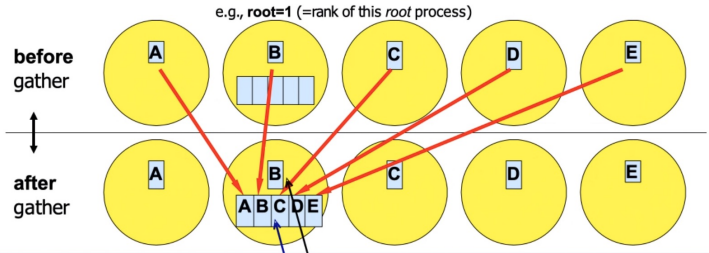
See, e.g., Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

sendcount describes only one message

Example: MPI_Scatter(sbuf, 1, MPI_CHAR, *rbuf*, 1, MPI_CHAR, 1, MPI_COMM_WORLD);

Completely ignored at all processes except *root*

Gather



C

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

Python

```
comm.Gather(sendbuf, recvbuf or None, int root=0)
recvobj = comm.gather(sendobj, int root=0)
```

See, e.g., Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

```
CALL MPI_Gather(sbuf, 1, MPI_CHARACTER, rbuf, 1, MPI_CHARACTER, 1, MPI_COMM_WORLD);
```

Completely ignored at all processes except *root*

recvcount describes only one message

Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- $d_0 \circ d_1 \circ d_2 \circ d_3 \circ \ldots \circ d_{s-2} \circ d_{s-1}$
  - d_i = data in process rank i
    - **single variable, or**
    - **vector**
  - $\circ$ = associative operation
  - Example:
    - **global sum or product**
    - **global maximum or minimum**
    - **global user-defined operation**
- floating point rounding may depend on usage of associative law:
  - $[(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ [\ldots \circ (d_{s-2} \circ d_{s-1})]$
  - $((((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \ldots) \circ d_{s-2}) \circ d_{s-1})$
  - May be even worse through partial sums in each process:
    $\sigma_{i=0}^{n-1} x_i \rightarrow [[[(\sigma_{i=0}^{n/s-1} x_i \circ \sigma_{i=n/s}^{2n/s-1} x_i) \circ (\ldots \circ \ldots)] \circ [\ldots \circ (\ldots \circ \ldots)]]]$

E.g., with $n=10^8$ rounding errors may modify last 3 or 4 digits!

Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in *resultbuf.*

C

- C/C++: root=0;
  MPI_Reduce(&inbuf, &*resultbuf*, 1, MPI_INT,
  MPI_SUM, root, MPI_COMM_WORLD);

Python

- Python: comm_world = MPI.COMM_WORLD
  snd_buf = np.array(value, dtype=np.intc)
  resultbuf = np.empty((), dtype=np.intc)
  comm_world.Reduce(snd_buf, *resultbuf*, op=MPI.SUM)
- The result is only placed in *resultbuf* at the root process.

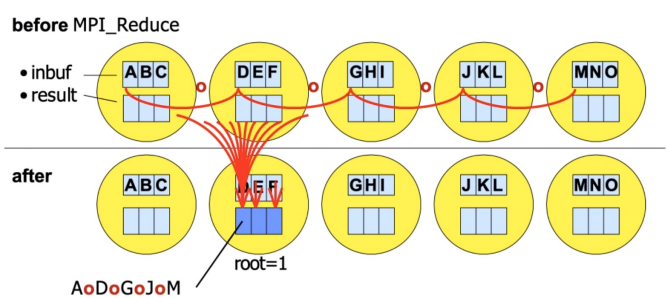
op=MPI.SUM and root=0 are defaults

The buffer can be added up & this way we can use reduce function for sum

Predefined Reduction Operation Handles

Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

MPI_Reduce



Reduce can be applied to arrays & vectors.

User-Defined Reduction Operations

- Operator handles
  - predefined
    - see table above
  - user-defined
- User-defined operation ▫:
  - associative
  - user-defined function must perform the operation vector_A ▫ vector_B
  - syntax of the user-defined function → MPI standard
- Registering a user-defined reduction function:

C

  - C/C++: MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *op)

Python

  - Python: op = MPI.Op.Create(func, commute=True *or* False)

- COMMUTE tells the MPI library whether FUNC is commutative.

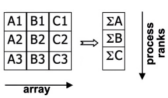
As a programmer you must ensure the function follows associative property.
It has two inputs

Variants of Reduction Operations

- MPI_Allreduce
  - no root,
  - returns the result in all processes

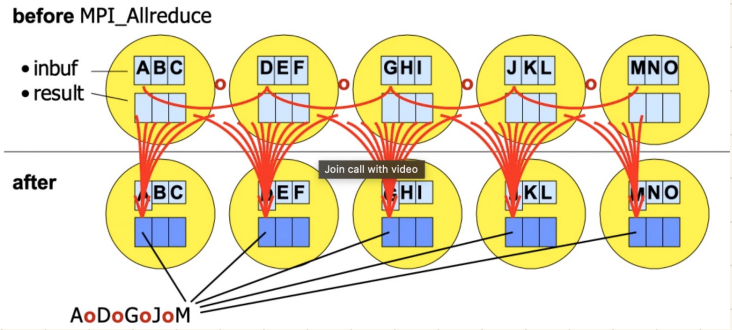
New in MPI-2.2

- MPI_Reduce_scatter_block and MPI_Reduce_scatter
  - result vector of the reduction operation is scattered to the processes into the real result buffers

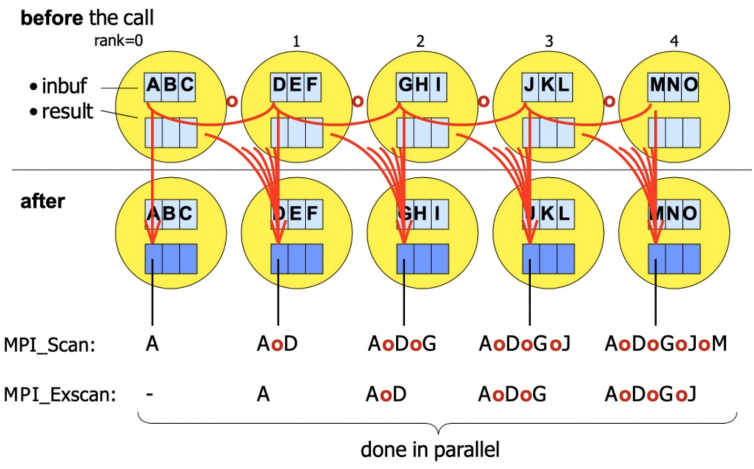


- MPI_Scan
  - prefix reduction
  - result at process with rank i := reduction of inbuf-values from rank 0 to rank i
- MPI_Exscan
  - result at process with rank i := reduction of inbuf-values from rank 0 to rank i-1

MPI_Allreduce

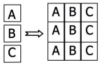


MPI_Scan and MPI_Exscan

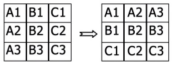


Other Collective Communication Routines

- MPI_Allgather → similar to MPI_Gather, but all processes receive the result vector



- MPI_Alltoall → each process sends messages to all processes



- MPI_.....v (Gatherv, Scatterv, Allgatherv, Alltoallv, Alltoallw)
  - → Each message has a different count and displacement
  - → array of counts and array of displs (Alltoallw: also array of types)
  - → interface does **not scale** to thousands of MPI processes!
  - → Recommendation: One should try to use data structures with same communication size on all ranks.

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I..... **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, ...

- **Nonblocking** collective operations do **not match** with **blocking** collective operations
  With point-to-point message passing, such matching is allowed
- Collective initiation and completion are separated
- **MPI_I...** calls are **local** (i.e., not synchronizing), whereas the **corresponding MPI_Wait** collectively **synchronizes** in same way as corresponding blocking collective procedure
- May have multiple outstanding collective communications on same communicator
- Ordered initialization on each communicator

you can have communication with overlapping computations.

MPI Datatype

MPI Datatypes

- In previous slides:
  - A messages was a contiguous sequence of elements of basic types:
  - buf, count, datatype_handle
- New goals in this part:
  - Transfer of any data in memory in one message
    - **Strided data (portions of data with holes between the portions)**
    - **Various basic datatypes within one message**

  - No multiple messages → **no multiple latencies** → requires initialization & terminal so additional overhead.
  - No copying of data into contiguous scratch arrays
    → **no waste of memory bandwidth**
- Method: **Datatype handles**
  - Memory layout of send / receive buffer
  - Basic types / **derived types**:
    - **vectors**
    - **subarrays**
    - **structs**
    - **others**

Message passing:
- **Goal and reality may differ !!!**

Parallel file I/O:
- Derived datatypes are **important** to express I/O patterns

In order to squeeze different datatypes in one message we need a user defined datatype.

Data Layout & the Describing Datatype Handle

March 17, 2022

- the memory is a multiple of 8, we may need to keep a hole before adding anything else in memory.
- in MPI to use a custom datatype you need a create_struct & commit to use it

```
MPI_Type_Create_struct( 2 , array_of_blocklengths, array_of_displacements , &buff_datatype)
                                                                          , array_of_types
```

- 2 → num types of data
- array_of_blocklengths → sum of size of arrays
- array_of_displacements → where does it start in memory → 0
- array_of_types → array that holds all the types

Derived Datatypes - Type Maps

- we need to describe the offset
- it is logically a pointer to a list of entries

Data Layout and the Describing Datatype Handle

```
struct buff_layout
  { int    i_val[3];
    double d_val[5];
  } buffer;
```

Compiler

```
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=0;
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=...;
MPI_Type_create_struct(2, array_of_blocklengths,
    array_of_displacements, array_of_types,
              &buff_datatype);
MPI_Type_commit(&buff_datatype);
```

```
MPI_Send(&buffer, 1, buff_datatype, ...)
```

&buffer = the start address of the data

the datatype handle describes the data layout

Jun Li, Department of Computer Science, CUNY Queens College

Derived Datatypes — Type Maps

- A derived datatype is logically a pointer to a list of entries:
  - *basic datatype* at *displacement*

Example:

derived datatype handle

basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

A derived datatype describes the memory layout of, e.g., structures, common blocks, subarrays, some variables in the memory

Contiguous Data

After you create a datatype you must commit it

After you finish using the derived datatype - you have to free up the space.

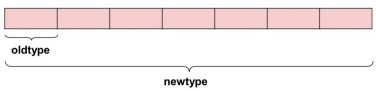
Vector Datatype

Example:

★ why is there a gap in the middle?
why is the stride 16?

Contiguous Data

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype



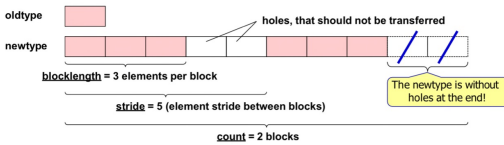
C

- C/C++: int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

Python

- Python: newtype = oldtype.Create_contiguous(int count)

Vector DataType



C

- C/C++: int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Python

- Python: newtype = oldtype.Create_vector(int count, int blocklength, int stride)

Committing and Freeing a Datatype

- Before a datatype handle is used in message passing communication, **it needs to be committed with MPI_TYPE_COMMIT.**
- This need be done only once (by each MPI process). (Using more than once ☹ corresponds to additional no-operations.)

C

- C/C++: int MPI_Type_commit(MPI_Datatype *datatype);

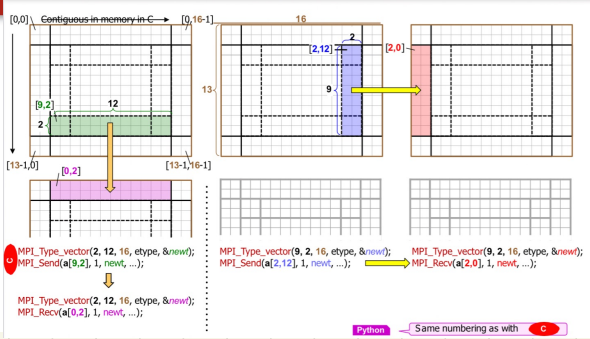
Python

- Python: datatype.Commit()

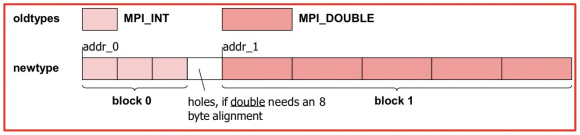
IN-OUT argument (although handle is not modified)

- If usage is over, one may call MPI_TYPE_FREE() to free a datatype and its internal resources.

Example with MPI_Type_vector



Struct Datatype



C

- C/C++: int MPI_Type_create_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

Python

- Python: newtype = MPI.Datatype.Create_struct(array_of_blocklengths, array_of_displacements, array_of_types)

count = 2
array_of_blocklengths = (3, 5)
array_of_displacements = (0, addr_1 – addr_0) 1)
array_of_types = (MPI_INT, MPI_DOUBLE)

1) Via MPI_Get_address and MPI_Aint_diff, see following slides

How to compute the displacement (1)

- array_of_displacements[i] := address(block_i) – address(block_0)

Retrieve an absolute address:

C

– C/C++: int MPI_Get_address(void* location, MPI_Aint *address)

Python

– Python: address = MPI.Get_address(location)

How to compute the displacement (2)

New in MPI-3.1

Relative displacement:= absolute address 1 – absolute address 2

C

– C/C++: MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)

Python

– Python: int MPI.Aint_diff(addr1, addr2)

Python's int allows 64 bit

New in MPI-3.1

New absolute address := existing absolute address + relative displacement:

C

– C/C++: MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)

Python

– Python: int MPI.Aint_add(base, disp)

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI_Aint and result in unexpected values on some platforms. The MPI_AINT_ADD and MPI_AINT_DIFF functions can be used to safely perform address arithmetic with MPI_Aint displacements. (*End of advice to users.*)

Example for array_of_displacements[i] := address(block_i) – address(block_0)

C

```
struct buff
   {     int i[3];
         double d[5];
   } snd_buf;
MPI_Aint iaddr0, iaddr1, disp;
MPI_Get_address( &snd_buf.i[0], &iaddr0);  // the address value &snd_buf.i[0] is stored into variable iaddr0
MPI_Get_address(&snd_buf.d[0], &iaddr1);
disp = MPI_Aint_diff(iaddr1, iaddr0);                    // MPI-3.0 & former: disp = iaddr1–iaddr0
```

New in MPI-3.1

Python

```
np_dtype = np.dtype([('i', np.intc, 3), ('d', np.double, 4)])
snd_buf = np.empty((),dtype=np_dtype)
addr0 = MPI.Get_address(snd_buf['i'])
addr1 = MPI.Get_address(snd_buf['d'])
disp = MPI.Aint_diff(addr1, addr0)
```

How to calculate displacement (1+2) → formulas here

- difference between the current type of data & the first type of data

In practice,

in Python there is no native struct

We can calculate difference

Scope & Performance Options

- There is no guarantee of performance

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the recv-buffer back into the strided application array
- Using derived datatype handles
- And which of the communication routines should be used?

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming
but
efficiency of MPI application-programming is not portable!

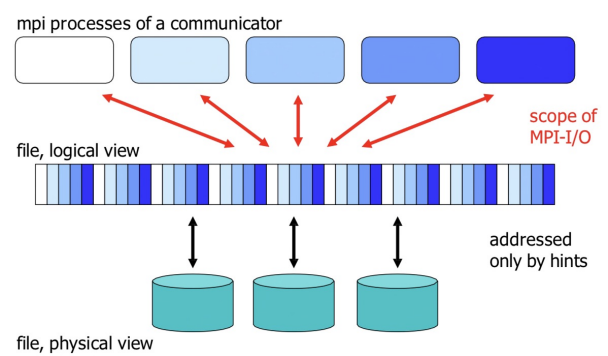
Parallel I/O

MPI-I/O Features

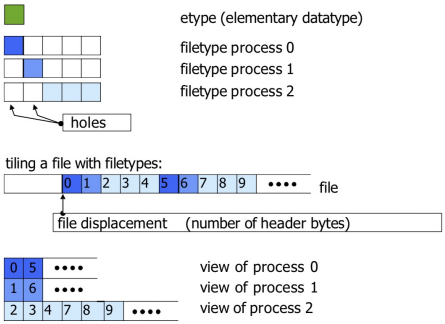
MPI-I/O Features

- Provides a high-level interface to support
  - data file partitioning among processes
  - transfer global data between memory and files (collective I/O)
  - asynchronous transfers
  - strided access
- MPI derived datatypes used to specify common data access patterns for maximum flexibility and expressiveness

Logical View / Physical View



Definitions



file - an ordered collection of typed data items

etypes
- is the unit of data access and positioning / offsets
- can be any basic or derived datatype
  (with non-negative, monotonically non-decreasing, non-absolute displacem.)
- generally contiguous, but need not be
- typically same at all processes

filetypes
- the basis for partitioning a file among processes
- defines a template for accessing the file
- different at each process
- the etype or derived from etype (displacements: non-negative, monoton. non-decreasing, non-abs., multiples of etype extent)

view
- each process has its own view, defined by: a displacement, an etype, and a filetype.
- The filetype is repeated, starting at **displacement**

offset - position relative to current view, in units of etype

Opening an MPI filename

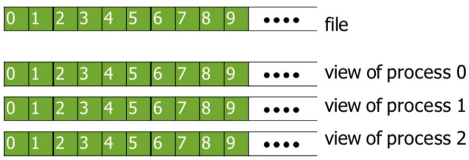
- Same filename could correspond to the same data

Default View

```
MPI_File_open(comm, filename, amode, info, fh)
```

- Default:
  - displacement = 0
  - etype = MPI_BYTE
  - filetype = MPI_BYTE

  } each process has access to the whole file



- Sequence of MPI_BYTE matches with any datatype
- Binary I/O (no ASCII text I/O)

Closing and Deleting a File

- Close: collective

```
MPI_File_close(fh)
```

- Delete:
  - automatically by MPI_FILE_CLOSE if **amode=MPI_DELETE_ON_CLOSE | ...** was specified in MPI_FILE_OPEN
  - deleting a file that is not currently opened:

```
MPI_File_delete(filename, info)
```

[*same implementation-dependent rules as in MPI_FILE_OPEN*]

Access Modes

- same value of **amode** on all processes in **MPI_File_open**
- Bit vector OR of integer constants
  - MPI_MODE_RDONLY - read only
  - MPI_MODE_RDWR - reading and writing
  - MPI_MODE_WRONLY - write only
  - MPI_MODE_CREATE - create if file doesn't exist
  - MPI_MODE_EXCL - error creating a file that exists
  - MPI_MODE_DELETE_ON_CLOSE - delete on close
  - MPI_MODE_UNIQUE_OPEN - file not opened concurrently
  - MPI_MODE_SEQUENTIAL - file only accessed sequentially: mandatory for sequential stream files (pipes, tapes, ...)
  - MPI_MODE_APPEND - all file pointers set to end of file [*caution: reset to zero by any subsequent MPI_FILE_SET_VIEW*]

Set/Get File View

- Set view
  - changes the process's view of the data
  - local and shared file pointers are reset to zero
  - collective operation
  - etype and filetype must be committed
  - datarep argument is a string that specifies the format in which data is written to a file: "native", "internal", "external32", or user-defined
  - same etype extent and same datarep on all processes
- Get view
  - returns the process's view of the data

```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info)
MPI_File_get_view(fh, disp, etype, filetype, datarep)
```

File Views

- Provides a visible and accessible set of data from an open file
- A separate view of the file is seen by each process through triple := (displacement, etype, filetype)
- User can change a view during the execution of the program - but collective operation
- A linear byte stream, represented by the triple (0, MPI_BYTE, MPI_BYTE), is the default view

Data Representation

- Data is represented the same way in the underlying structure
- can be user defined

- "native"
  - data stored in file identical to memory
  - on homogeneous systems no loss in precision or I/O performance due to type conversions
  - on heterogeneous systems loss of interoperability
  - no guarantee that MPI files accessible from C/Fortran
- "internal"
  - data stored in implementation specific format
  - can be used with homogeneous or heterogeneous environments
  - implementation will perform type conversions if necessary
  - no guarantee that MPI files accessible from C/Fortran

- "external32"
  - follows standardized representation (IEEE)
  - all input/output operations are converted from/to the "external32" representation
  - files can be exported/imported between different MPI environments
  - due to type conversions from (to) native to (from) "external32" data precision and I/O performance may be lost
  - "internal" may be implemented as equal to "external32"
  - can be read/written also by non-MPI programs
- user-defined

No information about the default,
i.e., datarep without MPI_File_set_view() is not defined

All Data Access Routines

positioning	synchronism	coordination		
		noncollective	collective	split collective
explicit offsets	blocking	READ_AT	READ_AT_ALL	READ_AT_ALL_BEGIN
		WRITE_AT	WRITE_AT_ALL	READ_AT_ALL_END
	nonblocking	IREAD_AT	IREAD_AT_ALL	WRITE_AT_ALL_BEGIN
		IWRITE_AT	IWRITE_AT_ALL	WRITE_AT_ALL_END
individual file pointers	blocking	READ	READ_ALL	READ_ALL_BEGIN
		WRITE	WRITE_ALL	READ_ALL_END
	nonblocking	IREAD	IREAD_ALL	WRITE_ALL_BEGIN
		IWRITE	IWRITE_ALL	WRITE_ALL_END
shared file pointer	blocking	READ_SHARED	READ_**ORDERED**	READ_**ORDERED**_BEGIN
		WRITE_SHARED	WRITE_**ORDERED**	READ_**ORDERED**_END
	nonblocking	IREAD_SHARED	N/A	WRITE_**ORDERED**_BEGIN
		IWRITE_SHARED		WRITE_**ORDERED**_END

Read e.g. **MPI_FILE**_READ_AT

New in MPI-3.1

Writing with Explicit Offsets

MPI_File_write_at(fh, offset, buf, count, datatype, *status*)

- writes **`count`** elements of **`datatype`** from memory **`buf`** to the file
- starting **`offset`** * units of **`etype`** from begin of view
- the elements are stored into the locations of the current view
- the sequence of basic datatypes of **`datatype`** (= signature of **`datatype`**) must match contiguous copies of the **`etype`** of the current view

Reading with Explicit Offsets

e.g. MPI_File_read_at(fh, offset, *buf*, count, datatype, *status*)

- attempts to read **`count`** elements of **`datatype`**
- starting **`offset`** * units of **`etype`** from begin of view (= **`displacement`**)
- the sequence of basic datatypes of **`datatype`** (= signature of **`datatype`**) must match contiguous copies of the **`etype`** of the current view
- EOF can be detected by noting that the amount of data read is less than **`count`**
  - i.e. EOF is no error!
  - use MPI_Get_count(status, datatype, *recv_count*)

Individual File Pointer, I.

e.g. MPI_File_read(fh, *buf*, count, datatype, *status*)

- same as *"Explicit Offsets"*, except:
- the offset is the current value of the **individual file pointer** of the calling process
- the individual file pointer is updated by

$$\text{new\_fp} = \text{old\_fp} + \frac{\text{elements(datatype)}}{\text{elements(etype)}} * \text{count}$$

i.e. it points to the next etype after the last one that will be accessed (*if EOF is reached, then recv_count is used, see previous slide*)

Individual File Pointer, II.

MPI_File_seek(fh, offset, whence)

- set individual file pointer fp:
  - set fp to offset – if whence=MPI_SEEK_SET
  - advance fp by offset – if whence=MPI_SEEK_CUR
  - set fp to EOF+offset – if whence=MPI_SEEK_END

MPI_File_get_position(fh, *offset*)

MPI_File_get_byte_offset(fh, offset, disp)

- to inquire offset
- to convert offset into byte displacement
  [*e.g. for* `disp` *argument in a new view*]

Shared File Pointer

- we want to give different files access to the same file without needing to maintain data consistency

- same view at all processes mandatory!
- the offset is the current, *global* value of the **shared file pointer** of `fh`
- multiple calls [*e.g. by different processes*] **behave as** if the calls were **serialized**
- non-collective, e.g.

```
MPI_File_read_shared(fh, buf, count, datatype, status)
```

- collective calls are *serialized* in the **order** of the processes' ranks, e.g.:

```
MPI_File_read_ordered(fh, buf, count, datatype, status)
```

```
MPI_File_seek_shared(fh, offset, whence)
MPI_File_get_position_shared(fh,offset)
MPI_File_get_byte_offset(fh, offset, disp)
```

- same rules as with individual file pointers

Jun Li, Department of Computer Science, CUNY Queens College

Nonblocking Data Access

```
e.g. MPI_File_iread(fh, buf, count, datatype, request)
     MPI_Wait(request, status)
     MPI_Test(request, flag, status)
```

- analogous to MPI-1 nonblocking

Application Scenery, I.

- Scenery A:
  - Task: Each process has to read the whole file
  - Solution: **MPI_File_read_all**
    = collective with individual file pointers,
    with same view (displacement+etype+filetype)
    on all processes
    [*internally: striped-reading by several process, only once from disk, then distributing with bcast*]
- Scenery B:
  - Task: The file contains a list of tasks,
    each task requires different compute time
  - Solution: **MPI_File_read_shared**
    =non-collective with a shared file pointer
    (same view is necessary for shared file p.)

- Scenery C:
  - Task: The file contains a list of tasks,
    each task requires **the same** compute time
  - Solution: **MPI_File_read_ordered**
    = **collective** with a **shared** file pointer
    (same view is necessary for shared file p.)
  - or: **MPI_File_read_all**
    = **collective** with **individual** file pointers,
    different views: *filetype* with
    MPI_Type_create_subarray(1, nproc,
    1, myrank, ..., datatype_of_task, *filetype*)
    [*internally: both may be implemented the same and equally with following scenery D*]

- Scenery D:
  - Task: The file contains a matrix,
    block partitioning,
    each process should get a block
  - Solution: generate different filetypes with
    MPI_Type_create_darray or **..._subarray**, the view on each process represents the block that should be read by this process,
    MPI_File_read_at_all with offset=0
    (= collective with explicit offsets)
    reads the whole matrix collectively
    [*internally: striped-reading of contiguous blocks by several process, then distributed with "alltoall"*]

Scenery – Nonblocking or Split Collective

- Scenery E:
  - Task: Each process has to read the whole file
  - Solution:
    - MPI_File_iread_all or MPI_File_read_all_begin
      = collective with individual file pointers,
      with same view (displacement+etype+filetype)
      on all processes
      [*internally: starting asynchronous striped-reading by several process*]
    - then computing some other initialization,
    - MPI_Wait or MPI_File_read_all_end.
      [*internally: waiting until striped-reading finished, then distributing the data with bcast*]

Error handling → "assembler for parallel computing"

Once there is error code, the behavior becomes unpredictable

Most important aspects:

- The communication should be reliable (same rule as for processor and memory)
- If the MPI program is erroneous → no warranties:
  - by default: abort, if error detected by MPI library otherwise, unpredictable behavior
    i.e., error handler MPI_ERRORS_ARE_FATAL is the default
  - C/C++: MPI_Comm_set_errhandler (comm, MPI_ERRORS_RETURN);

    Python: comm.Set_errhandler(MPI.ERRORS_RETURN)
    Newly added in MPI-4.0
  - directly after MPI_INIT with both comm = MPI_COMM_WORLD and MPI_COMM_SELF, then
    - **ierror returned by each MPI routine (except MPI window and MPI file routines)**
    - **undefined state after an erroneous MPI call has occurred (only MPI_Abort(…) should be still callable)**
  - Exception: MPI-I/O has default MPI_ERRORS_RETURN
    - Default can be changed through MPI_FILE_NULL:
    - MPI_File_set_errhandler (MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL)

      Python: MPI.FILE_NULL.Set_errhandler(MPI.ERRORS_ARE_FATAL)

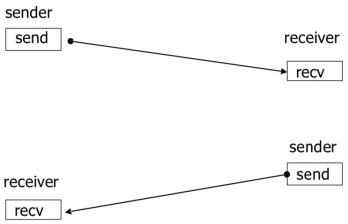
      — MPI_ERRORS_ARE_FATAL aborts the process and all connected processes
      — MPI_ERRORS_ABORT aborts only all processes of the related communicator
      New in MPI-4.0

Shared Memory

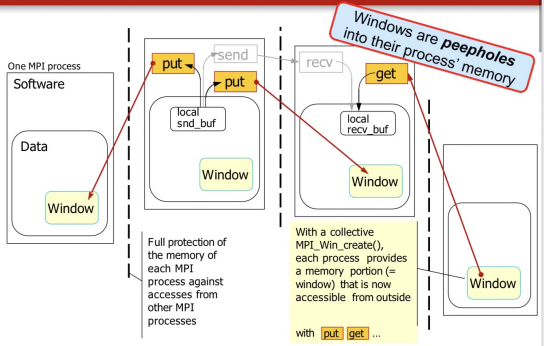
In general, MPI is a distributed memory system but it can take advantage of shared memory model

Cooperative Communication

- MPI-1 supports cooperative or 2-sided communication
- Both sender and receiver processes must participate in the communication

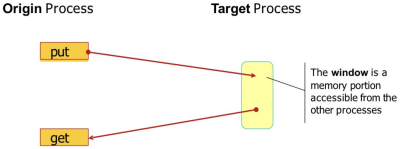


Typically, all processes are both, origin and target processes



One-sided Communication

- Communication parameters for both the sender and receiver are specified by one process (origin)
- User must impose correct ordering of memory accesses



One-sided Operations

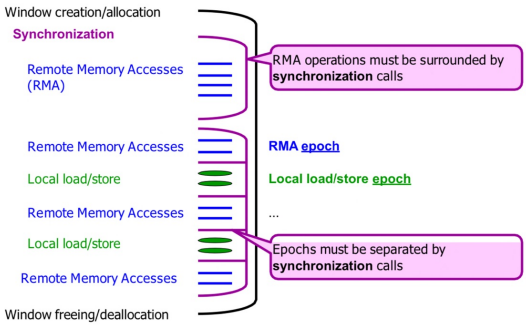
Three major sets of routines:

- Window creation or allocation
  - Each process in a group of processes **(defined by a communicator)**
  - defines a chunk of own memory – named ***window***,
  - which can be afterwards accessed by all other processes of the group.
- **R**emote **M**emory **A**ccess (RMA, nonblocking) routines
  - Access to remote windows:
    - **put, get, accumulate, …**
- Synchronization
  - The RMA routines are nonblocking and
  - must be surrounded by synchronization routines,
  - which guarantee
    - **that the RMA is locally and remotely finished**
    - **and that all necessary cache operation are implicitly done.**

Sequence of one-sided operations

For different IMA they are surrounded by different synchronization to make sure corresponding operations can be included.

Sequence of One-sided Operations



MPI_Put

- Performs an operation equivalent to a **send** by the **origin process** and a matching **receive** by the target process
- The origin process specifies the arguments for both origin and target
- **Nonblocking call** → finished by subsequent synchronization call
  → don't modify the origin (=send) buffer until next syncron

Where is the recv_buf in the target process ?

- The target buffer is at address
  target_addr = win_base$_{target_process}$
  + target_disp$_{origin_process}$ * disp_unit$_{target_process}$

As provided in MPI_Win_create or _all at the target proces

Like **send_buf, count, datatype** in MPI_Send

```
MPI_Put( origin_address, origin_count, origin_datatype,
         target_rank, target_disp_origin_process,
         target_count, target_datatype, win)
```

Like **dest** in MPI_Send

Like **count, datatype** in an MPI_Recv at the target process

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

Window Creation

- Specifies the region in memory (already allocated) that can be accessed by remote processes
- **Collective** call over all processes in the intracommunicator
- Returns an opaque object of type `MPI_Win` which can be used to perform the remote memory access (RMA) operations

A normal buffer argument — byte size, MPI_Aint

```
MPI_Win_create( win_base_addr_target, win_size_target,
                disp_unit_target, info, comm, win)
```

byte size, int

A window handle represents:
- all about the communicator
- and its processes,
- the location of the windows in all processes,
- the disp_units in all processes

MPI_Put

C

- C/C++: int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_co MPI_Datatype target_datatype, MPI_Win win)
  int MPI_Put_c(const void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, MPI_Count target_count, MPI_Datatype target_datatype, MPI_Win wi

Large count version, new in MPI-4.0

Python

- Python: win.Put((origin_buf, origin_count, origin_datatype), target_rank, (target_disp, target_count, target_datatype))

Window Creation

If you want to share a window, they need to be in same communicator

Window Creation with MPI_Win_create

C

- C/C++: int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
  int MPI_Win_create_c(void *base, MPI_Aint size, MPI_Aint disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)

Large count version, new in MPI-4.0

Python

- Python: win = MPI.Win.Create(memory, disp_unit, info, comm)

e.g., a numpy array

All Memory Allocation with modern C-Pointer

C

```
float *buf;  MPI_Win win; int max_length; max_length = ...;
MPI_Win_allocate( (MPI_Aint)(max_length*sizeof(float)), sizeof(float),
                  MPI_INFO_NULL, MPI_COMM_WORLD,  &buf,  &win);
// the window elements are buf[0] .. buf[max_length-1]
```

Python

```
np_dtype = np.single  # = C type float → MPI.FLOAT
max_length = ...
win = MPI.Win.Allocate(np_dtype(0).itemsize*max_length, np_dtype(0).itemsize, MPI.INFO_NULL,
                       MPI.COMM_WORLD)
buf = np.frombuffer(win, dtype=np_dtype)
# the window elements are buf[0] .. buf[max_length-1]
#buf = np.reshape(buf,())  # in case of max_length==1 and using buf as a normal variable instead of a 1-dim array
```

MPI_Get

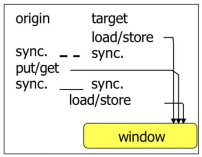
- Similar to the put operation, except that data is transferred from the target memory to the origin process
- To complete the transfer a synchronization call must be made on the window involved
- The local buffer should not be accessed until the synchronization call is completed

```
MPI_Get( origin_address, origin_count, origin_datatype,
         target_rank, target_disp, target_count,
         target_datatype, win)
```

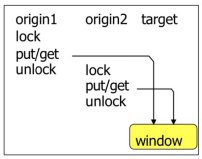
Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

Synchronization Calls (1)

- Active target communication
  - communication paradigm similar to message passing model
  - target process participates only in the synchronization
  - fence or post-start-complete-wait



- Passive target communication
  - communication paradigm closer to shared memory model
  - only the origin process is involved in the communication
  - lock/unlock



MPI_Accumulate

- Accumulates the contents of the origin buffer to the target area specified using the predefined operation `op`
- User-defined operations cannot be used
- Accumulate is **elementwise atomic**: many accumulates can be done by many origins to one target -> [*may be expensive*]

```
MPI_Accumulate(origin_address, origin_count,
               origin_datatype, target_rank, target_disp,
               target_count, target_datatype, op, win)
```

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

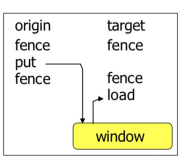
Synchronization Calls (2)

- Active target communication
  - MPI_Win_fence (like a barrier)
  - MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait/test
- Passive target communication
  - MPI_Win_lock, MPI_Win_unlock,
  - New in MPI-3.0 — MPI_Win_lock_all, MPI_Win_unlock_all,
  - New in MPI-3.0 — MPI_Win_flush(_all), MPI_Win_flush_local(_all), MPI_Win_sync

MPI_Win_fence

- Synchronizes RMA operations on specified window
- Collective over the window
- **Like a barrier**
- Used for active target communication
- Should be used before and after calls to put, get, and accumulate
- The `assert` argument is used to provide optimization hints to the implementation,
  - enables the optimization of internal cache operations
  - Integer 0 = no assertions
  - Several assertions with ***bitwise or*** operation

  E.g., in C: `MPI_MODE_NOSTORE | MPI_MODE_... | MPI_MODE_...`




```
MPI_Win_fence(assert, win)
```

Synchronization Calls (1)

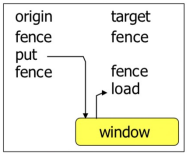
- Active Target Communication
  - We have both sources of communication
  -

Time goes down in the code

- Passive Target
  - Two origin & one target
  - Target does not need to do anything
  - Lock/unlock guarantees completion & returns the lock

MPI_Win_fence

- Synchronizes RMA operations on specified window
- Collective over the window
- **Like a barrier**
- Used for active target communication
- Should be used before and after calls to put, get, and accumulate
- The `assert` argument is used to provide optimization hints to the implementation,
  - enables the optimization of internal cache operations
  - Integer 0 = no assertions
  - Several assertions with ***bitwise or*** operation
    E.g., in C: `MPI_MODE_NOSTORE | MPI_MODE_... | MPI_MODE_...`



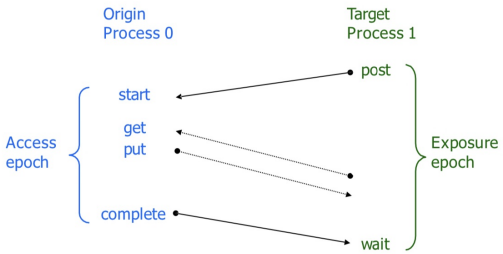

```
MPI_Win_fence(assert, win)
```

Start/Complete & Post/Wait, I

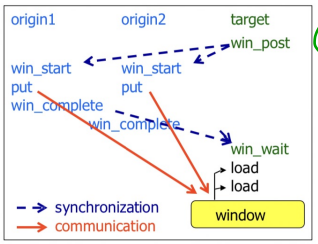
The more info you give to MPI the more weight it can assign

Start/Complete and Post/Wait, I.

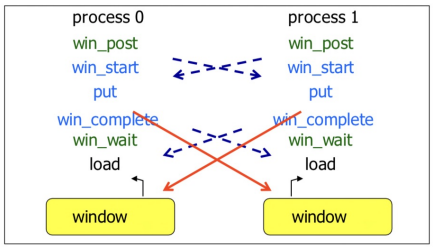
- Used for active target communication to restrict synchronization to a minimum



- RMA (put, get, accumulate) are finished
  - locally after win_complete
  - at the target after win_wait
- local buffer must not be reused before RMA call locally finished
- communication partners must be known
- no atomicity for overlapping "puts"
- assertions may improve efficiency
  --> give all information you have



- symmetric communication possible, only win_start and win_wait may block

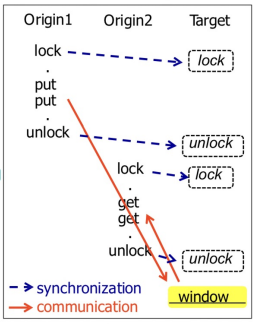


- Here, all processes are in the role of **target** *and* **origin**, i.e.
  - **expose** a window *and*
  - **access** windows per RMA

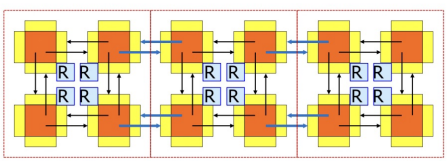
Lock/Unlock

- Does not guarantee a sequence
- agent may be necessary on systems without (virtual) shared memory
- Portable programs can use lock calls to windows in memory allocated **only** by `MPI_Alloc_mem`, `MPI_Win_allocate`, or

  `MPI_Win_attach` or `MPI_Win_allocate_shared` (New in MPI-4.0)
- RMA completed after `MPI_Unlock` at both origin and target

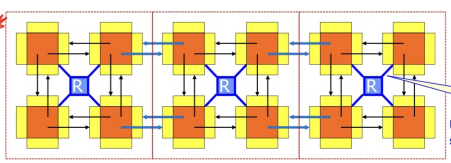


Programming opportunities with MPI shared memory: 1) Reducing memory space for replicated data



R = Replicated data in each MPI process

Example:
Cluster of 3 SMP nodes **without** using MPI shared memory methods



R = Shared memory → replicated data only once within each SMP node

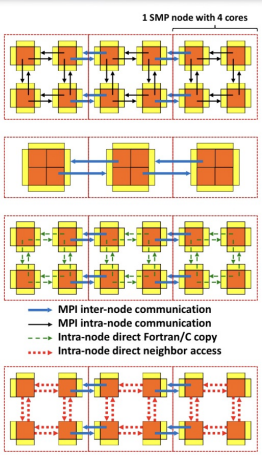
Direct loads & stores, no library calls

Using MPI shared memory methods

MPI shared memory can be used to **significantly reduce the memory needs** for **replicated data.**

Programming opportunities with MPI shared memory: 2) Hybrid shared/cluster programming models

- MPI on each core (not hybrid)
  - Halos between all cores
  - MPI uses internally shared memory and cluster communication protocols
- MPI+OpenMP
  - Multi-threaded MPI processes
  - Halos communica. only between MPI processes
- new! MPI cluster communication + MPI shared memory communication
  - Same as "MPI on each core", but
  - within the shared memory nodes, halo communication through direct copying with C or Fortran statements
- new! MPI cluster comm. + MPI shared memory access
  - Similar to "MPI+OpenMP", but
  - shared memory programming through work-sharing between the MPI processes within each SMP node



MPI – Summary

- Parallel MPI process model
- Message passing
  - blocking
    → several modes (**standard, buffered, synchronous**, ready)
  - nonblocking
    → to allow message passing from all processes in parallel
    → to avoid deadlocks and serializations
  - derived datatypes
    → to transfer any combination of data in one message
- Collective communications → a major opportunity for optimization
- One-sided communication and shared memory → functionality & perform.
- Parallel file I/O → important option on large systems / part of NetCDF, HDF5, ..

MPI targets portable and efficient message-passing programming but efficiency of MPI application-programming is not portable!

MPI-Summary

Next class review

He'll go through all previous assignment

Midterms Q's like the assignments - mostly assignment questions

- Simple coding
- Explain concepts
- About 5 questions

Any notes on textbook

- No electronic devices
- Short answer Q's - similar to previous assignments
- Some Q's modified from assignments

Midterm Review

March 24, 2022

Assignment 1

CS381-16 Assignment 2

Answer to 2.1

a. Floating point addition time = 5 + 2 * 2 = 9 ns
b. Unpipelined FP addition = 9 * 1000 = 9000 ns
c. Pipelined FP addition = fetch delay + other remaining operators store
   = 2 * 1000 + 5 + 2 = 2007 ns
d. When there is a level 1 cache miss, the fetch is done from L2 cache so the time to fetch increases from 2 nanoseconds to 5 nanoseconds. When there is a level 2 cache miss, there is a fetch from main memory, the time increases to 50 nanoseconds so there is significantly more delay.

(c) the gap between two consecutive instructions are 2 ns

(d)

Answer to 2.3

The first pair of nested loops will have 4 misses one for each beginning of the row A[0][0], A[1][0], A[2][0], A[3][0] as there would be miss before each new line is loaded in cache.
In the second pair, there would also be 4 misses, one for the first run of the nested for loop for A[0][0], A[1][0], A[2][0], A[3][0]. It seems that a larger matrix and a larger cache does improve the performance of the second loop only if the cache is large enough to hold all the elements of the matrix. Otherwise if there is a smaller cache, the first loop has a better performance than the second loop.

Answer to 2.4

Logical address space = 2 ^ 32 bytes
Total number of pages = 2^20
Size of each page = 2^12 bytes

Solution:

Suppose the matrix has order 8 or 64 elements; the cache line size is still 4; the cache can store 4 lines; and the cache is direct-mapped. Then the following table shows how A is stored in cache lines.

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[0][4]	A[0][5]	A[0][6]	A[0][7]
2	A[1][0]	A[1][1]	A[1][2]	A[1][3]
3	A[1][4]	A[1][5]	A[1][6]	A[1][7]
4	A[2][0]	A[2][1]	A[2][2]	A[2][3]
5	A[2][4]	A[2][5]	A[2][6]	A[2][7]
6	A[3][0]	A[3][1]	A[3][2]	A[3][3]
7	A[3][4]	A[3][5]	A[3][6]	A[3][7]
8	A[4][0]	A[4][1]	A[4][2]	A[4][3]
9	A[4][4]	A[4][5]	A[4][6]	A[4][7]
10	A[5][0]	A[5][1]	A[5][2]	A[5][3]
11	A[5][4]	A[5][5]	A[5][6]	A[5][7]
12	A[6][0]	A[6][1]	A[6][2]	A[6][3]
13	A[6][4]	A[6][5]	A[6][6]	A[6][7]
14	A[7][0]	A[7][1]	A[7][2]	A[7][3]
15	A[7][4]	A[7][5]	A[7][6]	A[7][7]

Assuming that no lines of A are in the cache when the first pair of loops begins, we see that there will be two misses for each row of A. Also, after the first two rows have been read, the cache will be full, and each miss will evict a line. As in the example in the text, an evicted line won't need to be read again. So we see that the total number of misses for the first pair of loops is 16, or

$$\frac{\text{number of elements in A}}{\text{cache line size}}.$$

More generally, then, for the first pair of loops, the number of misses is only affected by the size of A, not the size of the cache.

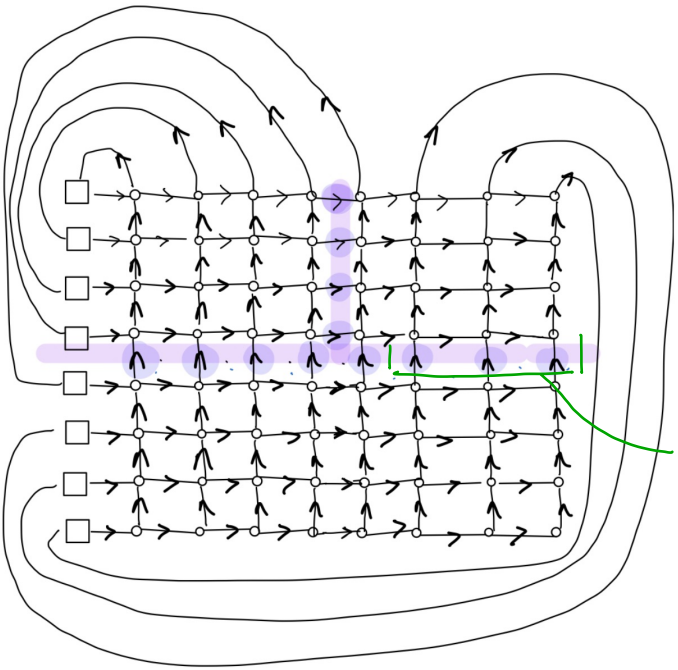
For the second pair of nested loops, let's also assume that no lines of A are in the cache when the loops begin. When we're working with column 0 ($j = 0$), each time we multiply `A[i][0]*x[0]` we'll need to first load a new cache line, since the previously read lines will only contain elements from rows $< i$. So executing the loop with $j = 0$, will result in 8 misses. After reading in the four lines containing `A[0][0]`, `A[1][0]`, `A[2][0]`, and `A[3][0]`, respectively, subsequent reads of elements of column 0 will evict these four lines. So after completing the multiplications involving column 0 of A, no elements in the first 4 rows of A will be stored in the cache. So every read of an element of A in rows 0–3 of column 1 will result in a miss. In fact, we see that every multiplication will result in a miss and there will be 64 misses.

Observe, however, in the second pair of loops if we have an 8 line cache, then the multiplications involving columns 1–3 of A won't result in misses, and we won't have additional misses until we start the multiplications by elements in column 4. Once the lines containing elements of column 4 are loaded, there won't be any additional misses, and we see that with an 8 line cache, the total number of misses is reduced to 16.

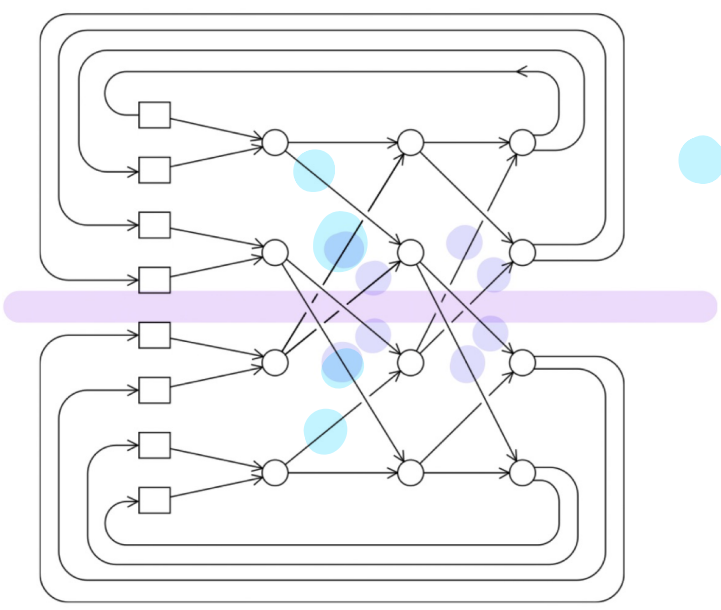
Answer to 2.14

As removing 12 unidirectional links makes it so that two parts are not connected, there need to be 6 links removed which is <= 8

Bisection Width = 6



did not need to get rid of this section



= in class

It does not matter if processors in one half can't communicate with others in it's half. As long as you divide it in half, it works

Figure 2: Omega network for distributed memory

As we need to remove 8 links to completely separate the two parts

The bisection width is therefore = 8/3 = 4

Answer to 2.15

(a) Core 0: x = 5

Core 1: y = x

Core 1 does not have x in its cache.

Core 0 uses snooping cache coherence but it uses write back cache which means that extra communication is necessary since the cache does not get immediately sent to memory. If the cache for x = 5 is sent to memory by the time Core 1 executes y = x, then y receives the value 5 otherwise it receives an old value of x which was already in memory.

(b) If it uses a directory based protocol, so when x = 5 is read into Core 0's cache, the directory entry corresponding to that line is updated to show that Core 0 has a copy of the line. However when a cache variable is updated only the cores storing that variable is updated. As Core 1 does not already have the value in its cache. Similarly from before if x is not updated in main memory, Core 1 will have an old copy of x, else if it is updated Core 1 will receive the updated value for x.

(c) In the last two problems, there is guaranteed cache coherence only if both cores already have the variable in cache. It should be the case that a core which does not have a value in it's cache should have the updated value. We can do this by employing a critical section, so that a core can only get the value of x once it's been updated in main memory.

2.15. **a.** Suppose a shared-memory system uses snooping cache coherence and write-back caches. Also suppose that core 0 has the variable `x` in its cache, and it executes the assignment `x = 5`. Finally suppose that core 1 doesn't have `x` in its cache, and after core 0's update to `x`, core 1 tries to execute `y = x`. What value will be assigned to `y`? Why?

b. Suppose that the shared-memory system in the previous part uses a directory-based protocol. What value will be assigned to `y`? Why?

c. Can you suggest how any problems you found in the first two parts might be solved?

Solution:

a. Since x is not in core 1's cache, the invalidation that core 0 sends won't have any effect on its cache. Furthermore, since the system uses write-back cache, when core 1 loads the line containing x, it may load a line containing the old value of x. So the assignment y = x might assign a value x had before the assignment x = 5 was executed.

b. In a directory-based system, when core 0 executes the assignment, it will invalidate the line containing x in main memory by notifying the directory. A problem may happen when core 1 loads the line containing x before the directory has finished invalidating the line.

c. The programmer could explicitly synchronize the two cores: core 1 won't attempt to use x until core 0 has notified it that the update has been completed. There are several alternatives that could be used. For example, among the synchronization methods discussed in Chapter 2, either semaphores or busy-waiting could be used.

CS381-16 Assignment 4

Answer to 2.23

In this aggregation, each core would have one bin to fully increment all values of that bin. Which means that there would have to be as many cores as there are bins, which could be many cores which reduces performance significantly. Also this way, as we call Find_bin to return b for each bin, we cannot make sure to divide the work between the cores roughly equally as the number of elements in bins are different, as one bin might have much more data than other bins.

2.23. In our application of Foster's methodology to the construction of a histogram, we essentially identified aggregate tasks with elements of `data`. An apparent alternative would be to identify aggregate tasks with elements of `bin_counts`, so an aggregate task would consist of all increments of `bin_counts[b]` and consequently all calls to `Find_bin` that return `b`. Explain why this aggregation might be a problem.

Solution: One problem with this approach is that we don't know which elements of `data` belong to which bins. For example, if process/thread q is responsible for all increments to `bin_counts[i]`, we have no way of insuring that process/thread q is assigned the elements of `data` that are assigned to `bin_counts[i]`. So when the processes/threads determine the bins to which the elements of `data` are assigned, every time a process/thread r other than q gets an element that belongs to bin i, there will have to be communication between processes/threads q and r. In the case of shared memory this will require some kind of mutual exclusion lock for each element of `bin_counts`, and this could result in a serious deterioration in performance as the processes/threads compete for access to element of `bin_counts`. In the case of distributed memory, process/thread r will have to send a message to process/thread q, and if the elements of `data` have been poorly distributed, the program could require communications for most of the elements of `data`.