Introduction

Background and Motivation

- Parallel computing is a part of HPC.
  - HPC also includes everything else that makes the computation fast.
  - No point parallelizing without increasing performance.
  - You might want to optimize for the architecture.
  - Sometimes overhead outweighs benefits from parallelization.
- Focusing on parallel algorithms.
  - Different version of parallel algorithms suits different architecture or models.
- Many application yo.
- People made super computers throughout the 1900s
- Super computers rely on carefully designed interconnects.
- Cloud computers are just AWS instances.
- Many aspects

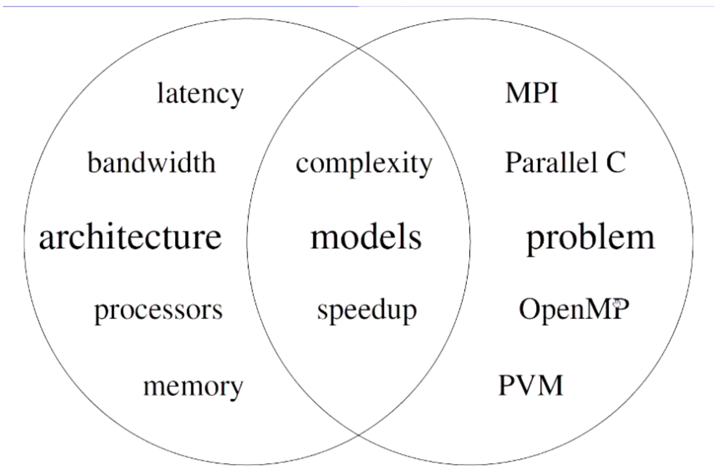


Figure: Overlapping aspects of parallel computing.

Complexity

- $f(n) = O(g(n)) \Rightarrow f$ grows no faster than g
- $f(n) = \Omega(g(n)) \Rightarrow f$ grows no slower than g
- $f(n) = o(g(n)) \Rightarrow f$ grows slower than g
- $f(n) = \omega(g(n)) \Rightarrow f$ grows faster than g
- $f(n) = \Omega(g(n)) \land f(n) = O(g(n)) \Rightarrow f(n) = \Theta(g(n))$
- Strictly speaking we should really use $\in$ instead of $=$
- Some common name for complexities:
  - Constant
  - Logarithmic
  - Polylog: $(\log(n))^c$
  - Linearithmic: $n \log n$
  - Quadratic: n^2
  - Polynomial or geometric
  - Exponential
  - Factorial
- Log factor are often ignored.

Model

- RAM model: *random access machine*
  - Common model when we talk about sequential time complexity.
- Multiplying the number of computers by a constant factor doesn't change the complexity.
  - Solution: allow p, the number of processors to increase with problem size and hence reduces the complexity.

PRAM

- Parallel Random Access Machine
- p number of RAM processors, each have private memory and share a large shared memory, all memory access takes the same amount of time.
- Does things synchronously, AKA in lock steps.
- PRAM pseudo code looks like regular pseudo code but there's this
  for $i \leftarrow 0$ **to** $n - 1$ **do in parallel**
  processor i **does** thingy

Many different PRAM model

- EREW: exclusive read, exclusive write
- CREW: concurrent read, exclusive write
- CRCW: concurrent read, concurrent write
  - Concurrent write have different types
    - COMMON: Error when two processor tries to write to the same location with different value.
    - ARBITRARY: Pick a arbitrary processor if many processor writes the same time.
    - PRIORITY: Processor with lowest ID writes.
    - COMBINING: Runs a function whenever multiple processors tries to write at the same time.
      - Too powerful.
- ERCW: exclusive read, concurrent write (never used)

Power of model: expresses the set of all problems that can be solved within a certain complexity.

- A is more powerful that B if A can solve a larger set of problems within any complexities.
- A is equally powerful as B if they can solve the same set problems within any complexities.
- Partial ordering.
- COMMON, ARBITRARY, PRIORITY and COMBINING are in increasing order of power.
- Any CRCW PRIORITY PRAM can be simulated by a EREW PRAM with a complexity increase of $\mathcal{O}(\log p)$

- *Parallel Computation Thesis*: any thing can be solved with a Turing Machine with polynomially bounded space can be solved in polynomially bounded space with unlimited processors.
  - Unbounded *word sizes* are not useful, so we limit word counts to $\mathcal{O}(\log p)$
- *Nick's Class* (NC): Solvable in polylog time with ploy number of processors.
- Widely believed that $\mathbf{NP} \neq \boldsymbol{P}$

Definitions (need to remember)

- $w(n) = t(n) \times p(n)$ where $w(n)$ is the work / cost, $t(n)$ is the time and $p(n)$ is the number of processors.
  - Optimal processor allocation means: $t(n) \times p(n) = \Theta(T(n))$ where $T(n)$ is the time taking by a sequential algorithm.
    - Equivalent to $t(n) \times p(n) = O(T(n))$ because $t(n) \times p(n) = \Omega(T(n))$ always.
  - $\text{Speedup}(n) = \frac{T(n)}{t(n)}$
    - Speedup optimal = processor optimal.
  - Optimal: processor optimal AND $t(n) = \mathcal{O}(\log^k n)$
    - Processor optimal and polylog in time.
  - Efficient: Assume $T(n) = \Omega(n)$ $w(n) = \mathcal{O}(T(n) \log^\alpha n)$ AND polylog in time
    - Optimal but polylog increase in work.
- *size*: $\text{Size}(n)$ is the total number of operations it does.
- *efficiency*: $\eta(n)$ speedup per processor
  - $\eta(n) = \frac{T(n)}{w(n)} = \frac{\text{Speedup}(n)}{p(n)}$
- You can decrease p and increase t by a factor of $O\left(\frac{p_1}{p_2}\right)$, $w(n)$ doesn't increase its complexity.
  - Can't do it the other way around.

Brent's Principle (important)

- If something can be done with size x and t time with infinite processors, then it can be done in $t + \frac{x-t}{p}$ time with p processors

Amdahl's Law

- Maximum speedup: if f is the fraction of time that can't be parallelized, then $\text{Speedup}(p) \to \frac{1}{f}$ as $p \to \infty$
  - Honestly very obvious.

Gustafson's Law

- s is fraction time of serial part, r is fraction time of parallel part, then $\text{Speedup}(p) = \Omega(p)$
  - Very obvious again…

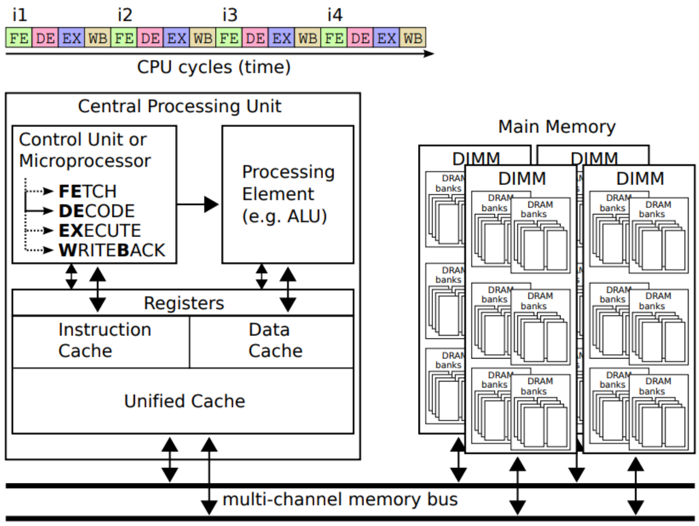
Algorithms

- sum
- logical or
- Maximum
  - n^2 processors all compare all elements and set is_max array to false if element isn't maximum.
  - Only processor with element being max write it to the returning memory address.
- Maximumn^2
  - $\mathcal{O}(\log \log n)$
  - n processor on n elements.
  - Is efficient
  - Make elements into a square, find maximum on each row recursively.
  - Find maximum of maximum of the rows using maximum.
  - $\mathcal{O}(\log \log n)$ levels of recursion, each level takes $\mathcal{O}(1)$ times
- Element Uniqueness
  - Have an array size of MAX_INT.
  - Write processor ID to the array with the element.
  - Check if processor ID is indeed there, if not there's another element there.
- Replication

- $O(\log n)$
- Replication optimal
  - $p = \frac{n}{\log(n)}$ and copy at the end.
- Broadcast
  - Just replicate
- Simulate PRIORITY with COMMON n^2
  - Minimum version of Maximum
- Simulate PRIORITY with EREW
  - All processor wants to write
  - Sort array A of tuples (address, processorID) using Cole's Merge Sort.
  - For each processor k, if $A[k]$.address $\neq A[k-1]$.address then $A[k]$.processorID is the smallest ID that wants to write to that address.

Architecture

- Fetch Decode Execute WriteBack



- Bus is a wire and everyone can see everything on that wire.

- Pipeline: let's do all of them at the same time for the next 4 instructions
  - Need to predict the next 4 instructions sometimes.
- Superpipeline: Do all of them for the next 8 (or more) instructions.
- Superscalar: Multiple pipeline in parallel

- Word size: 64 bits, 32 bits etc, various aspects:
  - Integer size
  - Float size
  - Instruction size
  - Address resolution (mostly bytes)
- Single instruction multiple data SIMD
  - Make word size more complicated

- Coprocessor
  - Used to means stuff directly connected to the CPU like a floating point processor.
  - Now can means FPGA or GPU.

- Multicore processor are just single core duplicated but they all have one extra single shared cache.

- Classification of parallel architectures
  - SISD regular single core.
  - SIMD regular modern single core.
  - MIMD regular multicore.
  - MISD doesn't exist.
- SIMD vs MIMD
  - Effectively SIMD vs non-SIMD
  - Most processor have multicore and SIMD on each core.
    - So a balance between the two.
  - SIMD cores are larger so less of them fit on a die.
  - SIMD is faster at vector operations.
  - SIMD is not useful all the time so sometimes the SIMD part sit idle.
  - SIMD is harder to program.

- Shared memory: All memory can be accessed by all processors.
  - All memory access truly equal time: symmetric multi-processor.
    - Only can have so many cores when the bus is only so fast.
    - Making more buses doesn't help cause space also slows things down.
    - Sometimes can be done with switching interconnect network.
  - Some processor access some memory faster.
    - More complex network.
  - Distributed shared memory: each processor have its own memory but interconnect network exist so you can read other people's memory.
    - *non-uniform memory access* NUMA
    - Static interconnect network: each node connect to some neighbors.
      - *degree*: just like degree in graphs.
      - *diameter*: just like in graphs.
      - *cost* $=$ degree $\times$ diameter
- Distributed memory: Each processor have its own memory. Each process live on one processor.

- Blade contains Processor / Package / Socket which contains Core which contains ALU.

- Implicit vs explicit: explicit $\to$ decision made by programmer
  - Parallelism: Can I write a sequential algorithm.
  - Decomposition: Can I pretend threads processes doesn't exist.
  - Mapping: Can I pretend all cores are the same.
  - Communication.

- Single Program Multiple Data: one exe
- Multiple Program Multiple Data: multiple exe

Other HPC considerations
- Cache friendliness
- Processor-specific code
- Compiler optimization.
  - Compiler from CPU maker are usually better.
  - So Intel compiler is better than both clang and gcc.

Memory interleaving
- Memory module takes a while to recharge, so we interleave a page on different memory module.

Automatic Vectorization

- Sometimes compilers automatically insert SIMD instructions in place of loops.
  - Depends on the availabilities of a lot of things, including the OS.
- Manual SIMD:

```
multiply_and_add(const float* a, const float* b, const float* c, float* d) {

  for(int i=0; i<8; i++) {
    d[i] = a[i] * b[i];
    d[i] = d[i] + c[i];
  }
}

__m256 multiply_and_add(__m256 a, __m256 b, __m256 c) {
  return _mm256_fmadd_ps(a, b, c);
}
```

- AVX have to be aligned: i.e. 256 bits SIMD have to be 256 bits aligned - address is multiple of 256 bits.

Multithreading

- Synchronization is more expensive if threads are on cores further away.
  - It's expensive in general.

- Instruction reordering: thread continues with other instructions while it waits on earlier instructions.
- Speculative execution: don't wait on instructions, just go for it and if it fails then unroll.

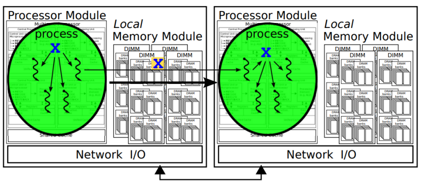
- Some programming patterns are more friendly to NUMA.

Message passing considerations

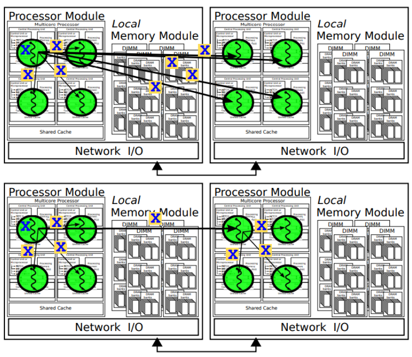
- Multi processing have to pass messages around because processes don't share address space.
- Hard to predict performance.

Wants good communication patterns

- For multithread multiprocess:



- For single thread multiprocess:



OpenMP

- Abstracts single process, multithreaded program execution on a single machine.

- Abstracts: Multi-socket, multi-core, threads, thread synchronization, memory hierarchy, SIMD, NUMA.
- Everything OMP does are hints.

- *internal control variable*: ICV: OMP_NUM_THREADS, OMP_THREAD_LIMIT, OMP_PLACES, OMP_MAX_ACTIVE_LEVELS.
- Can also be set with functions in `#include "omp.h"`

Execution Model
- There's an implicit parallel region on the outside.
- There's by default an implicit barrier at the end of each parallel region.
  - `no-wait` removes the implicit barrier
- If a parallel region is encountered, then the threads split and a new team is created.
- A lot of parallel region nested can create a lot of thread very quickly.
  - Can limit nesting by `OMP_MAX_ACTIVE_LEVELS`.

Memories: global, stack, heap
- Threads have their own stack but share global and heap.

Directives
- The `#pragma omp` thingy.
- Allows specifying parallelism and still allow the base language to be the same.
  - Theoretically, simply remove the directives and program will just run like a sequential program.
- Syntax:

```
#pragma omp <directive name> [[,]<clause> [[,]<clause> [...]]]
<statement / block>
```

- Multiple directives can be applied to one following block
- Some directives are *stand alone*, they don't have structured block following them.

Synchronization
- *thread team* is a group of threads.
- `barrier` will block threads in a team that reach it early.
- `flush` will enforce consistency between different thread's view of memory.
- `critical` ensures a critical region where only one thread can be in it at a time.
- `atomic` is faster than critical but only for simple operations.

- `simd` make use of SIMD instructions.
- *places*: specify how processing units on the architecture are partitioned.

- Thread encounters a parallel directive -> split itself into the number of threads.
- `#pragma omp parallel`
  - create some number of threads and do its thing.
  - clauses:
    - `num_threads(int)` overrides ICV, limited by OMP_THREAD_LIMIT
    - `private(list of variables)` each thread will have own memory allocated to private variable.
      - Default for variables on stack.
    - `shared(list of variables)` all thread share the same variables, same piece of memory.
      - OpenMP will add locks.
    - `threadprivate(list of variables)` variable stay with the thread if all threadprivate directives are identical.
  - Can combine with `for`, `loop`, `sections` and `workshare`.

- `#pragma omp for`
  - clauses:
    - `schedule([modifier[, modifier]:]kind[, chunk_size])`
      - kind:
        - `static`: divided into chunk_size (default $\frac{\text{iterations}}{\text{num threads}}$) and distributed round-robin over the threads.
        - `dynamic`: chunks of chunk_size (default 1) distributed to threads as they complete them.
        - `guided`: like `dynamic` but varying `chunk_size`, large chunks at the start and small chunks at the end.
        - `auto`: default.
        - `runtime`: determined by `sched-var` ICV.
      - modifier:
        - `monotonic`: chunks are given in increasing logical iteration
        - `nonmonotonic`: default, allows *work stealing*: I finished early, I will now take your work.
        - `simd`: try to make the loop into SIMD constructs.
    - `collapse(n)`: n nested loops are combined into one large logical loop.
    - `ordered[(n)]`: There are operations in the loop that must be executed in their logical order.
    - `reduction([reduction-modifier, ] reduction-modifier:list)`: a list of variable that will be used in a reduction operation.
      - Allowed operations: +, -, *, &, ^, &&, ||, max, min
      - Example: `#pragma omp parallel for reduction(+:x)`, `x` is the result, `+` is the operation.
        - `x` starts as a private variable initialized to the identity value.
        - global `x` will be assigned to the sum of all `x`s at the end.
- `#pragma omp loop`
  - Work for any loop, not just for.
  - Main diff to for is `bind`
- `#pragma omp sections`
  - Have `#pragma omp section` inside.
  - Each `#pragma omp section` gets executed by one thread.
  - clauses:
    - `private(list of variables)`: each thread will have its own version of the variable.
    - `firstprivate(list of variables)`: same as private but memory is initialized to the global version.
    - `lastprivate(list of variables)`: copy the private variables to the global version for the "lexically last" private variables.
      - A variable can be firstprivate and lastprivate at the same time.
- `#pragma omp single`
  - Only do it in a single thread in the team, used inside `#pragma omp parallel`
    - `private(list of variables)`: each thread will have its own version of the variable.
    - `firstprivate(list of variables)`: same as private but memory is initialized to the global version.
- `#pragma omp workshare`
  - Here's a bunch of independent statements / blocks, figure out how to parallelize it.
- `#pragma omp atomic`
  - critical for read, write, update (`x += 1`), compare (`if (expr < x) x = expr;`).
- `#pragma omp critical [(name) [[,] hint(hint-expression)]]`
  - clauses:
    - `(name)`: two critical region with the same name can't happen at the same time.
      - All no name critical region are treated as having the same name.

    - `hint(hint-expression)`:
        - `omp_sync_hint_uncontended`
        - `omp_sync_hint_contended`
        - `omp_sync_hint_speculative`: try to speculate.
        - `omp_sync_hint_nonspeculative`: don't try to speculate.
- `#pragma omp ordered`
    - Inside loops so that they're executed in their logical order.
- `#pragma omp barrier`
    - Explicit barrier.
- `#pragma omp flush`
    - Sync cache.
    - Be aware of code reordering.
- `#pragma omp task`
    - The *Task Model*: specify work without allocating work to threads.
    - Task is a unit of work.
    - Task have dependencies such as completion of other tasks.
    - Task may generate other tasks.
    - Uses many same clauses such as `private`, `shared` and `firstprivate`.
    - Task can have data affinity.
    - clauses:
        - `depend([depend-modifier,] dependence-type:locator-list)`.
        - `priority(int)`: hint of order of execution.
        - `affinity([aff-modifier :] locator-list)`
- `#pragma omp taskloop`
    - clauses:
        - `num_tasks([strict:]num-tasks)`: specify the number of tasks that will be generated.
        - `grainsize([strict:]grainsize)`: how many iteration per task.
- `#pragma omp taskwait`
    - Wait for all current child tasks to finish

Places

- OMP_PLACES: list of power units by their identifiers
    - `{0,1,2,3},{4,5,6,7}`: specify two places each with 4 processing units.
        - use `hwloc-ls` to find processing unit number.
    - `threads(8)`: 8 places on 8 hardware threads
    - `cores(4)`: 4 places on 4 cores.
    - `ll_caches(2)`: 2 places on 2 set of cores where all the cores in a set shares their last level cache.
    - `numa_domains(2)`: 2 places on 2 set of cores whose closes memory is the same or similar distance.
    - `sockets(2)`: 2 places on two sockets
    - OMP_PLACES partition power units into places. Which can then be referred to by `proc_bind(type)` clause in `parallel` directives.
- `proc_bind(type)`: overrides OMP_PROC_BIND, only in `parallel` directives.
    - `primary`: All threads created in the team are in the same place.
    - `close`: Threads are allocated to places in a round-robin fashion - first thread in place i, second thread in place i + 1, third thread in place i + 2
    - `spread`: Place thread in a way so that the distance between the power unit ID are as far as possible.

Memory
- Sending memory to other numa domains cost cache as well because the send operation needs to be done by a CPU which means cache.
- OpenMP memory classification:
  - `omp_default_mem_space`: DRAM
  - `omp_large_cap_mem_space`: SSD
  - `omp_const_mem_space`: optimized for read only
  - `omp_high_bw_mem_space`: high bandwidth
  - `omp_low_lat_mem_space`: low latency.
- Memory allocator have traits:
  - `sync_hint`: expected concurrency - `contended` (default), `uncontended`, `serialized`, `private`
  - `alignment`: default byte.
  - `access`: which thread can access the memory, `all` (default), `cgroup`, `pteam`, `thread`
  - `pool_size`: total amount of memory the allocator can allocate.
  - `fallback`: on error return null or exit, default is first try standard allocator and return null if fail.
  - `partition`: `environment` (default), `nearest`, `blocked`, `interleaved`. How is the allocated memory partitioned over the allocator's storage resource.

Prefix Sum
- Doesn't have to be sum, can also be any other associative operations (like prod, min, max).
- The only way to reduce depth is to increase size (hopefully only slightly).

Upper/Lower parallel prefix algorithm
- Divide array into two parts and compute their prefix sum.
- Add the sum of the first part to the second part.
- $\Theta(\log n)$ time complexity
- $\Theta(n \log n)$ work
- $\Theta(n \log n)$ size
- Half of the processors are idle all time except first iteration. (can probably be easily fixed)

Odd/Even parallel prefix algorithm
- Divide array into odd and even indices parts.
- Add odd indices to even indices.
- Compute prefix of even part recursively.
  - Now the even part contains the correct prefix.
- Compute the odd part in one parallel step.
- Same complexity as Upper/Lower, but 2 times slower.

Ladner and Fischer's parallel prefix algorithm.
- Optimal possible time.
- Split array into two parts and use odd even for the first part, upper lower for the second part.
  - Odd even for the first part is beneficial because the last element is available one step earlier.

Pointer jumping
- All processor replace next with next next, so you start going in 2^n steps for each iteration.

Sorting
- Merge sort parallelized is $O(n)$ because last merge is sequential.
- Quick sort parallelized is $O(n)$ because the first split is sequential.

Parallel merge

- $\mathcal{O}\left(\frac{n}{p} + \log p\right)$ or $\mathcal{O}(\log n)$ where $p = n$
- Two sorted list, assume all value are below n where n is the length of the resulting array.
- Count unique value for both of them.
- Write the sum of count for both array to result array with index X.
- Now the count is sorted.
- Compact the result array.
- Use prefix sum to space the resulting array evenly so that there are $\text{count} - 1$ null element after even element.
- Use distribution to fill out the rest of the array.

Compaction

- Move all non null element to the first part of the array.
- Use prefix sum to count the index of each empty element.
- Move each non empty element to its index.

Unique Counts

- Sorted array to (value, count) element.
- Find all places where the adjacent values are different.
- Use prefix sum to find their index.
- Reverse engineer their count with old indices.

Distribution

- Array with some null value, fill with the closest non null value to the left.
- Best complexity is achieved with simple broadcast.

1. Use prefix sum and unique count to figure out how many empty element are after each non empty one.
2. Do sequential distribute with each processor.
3. For the processors where their first element is null:
   1. Still need to fill
   2. Use info obtained previously at the very first step to calculate how long does this null sequence last.
   3. All processor involved in the null sequence, broadcast!

Rank sort

- Count the number of element smaller and number of element bigger, and just write this element to the array.
- Use n^2 processors. Can count the index in $\mathcal{O}(\log n)$ time.
- With a combine PRAM, it can be done in $\mathcal{O}(1)$ time.

Rank Merge

- Much simpler than Parallel merge, just use binary search to find the ranks.

Bitonic MergeSort

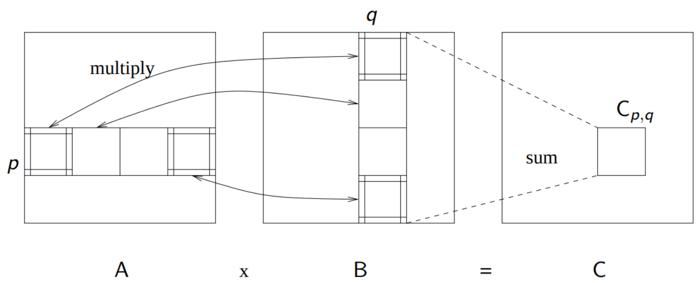
- Bitonic is a sequence is two monotonic sequences but one up and one down.
- You can find the pivot and turn this into a regular merge.
- Alternatively:
  - compare and maybe swap each two pair of element in two part of the array (none of them reversed.)
  - You end up getting two bitonic array.
  - Keep doing this and you sort it.

- Same time complexity.

You can use bitonic sort to do bitonic merge sort, by keep constructing bitonic lists and merging them with bitonic sort.

Matrix Multiplication

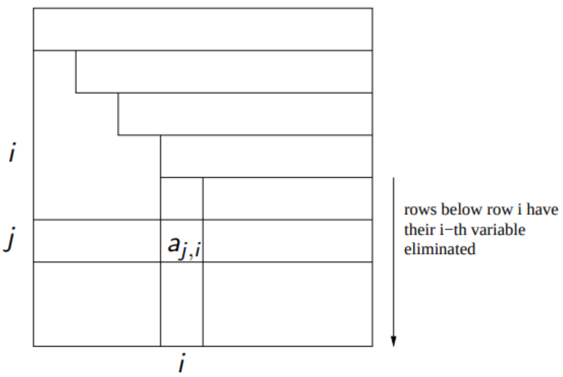
- Matrix multiplication doesn't have dependencies between them, so easy to parallelize with less than n^2 processors.
- Pretty trivial to parallelize in ideal conditions, so we will focus on practical side of matrix multiplication.
- For huge matrices, we can divide them into smaller one, multiply the smaller ones, and then sum the smaller ones.



- You can divide the matrices into 4 parts recursively, until the matrix is small enough to fit into the cache.

Gaussian elimination

- Common for matrix to be sparse, aka mostly zero.
- Gaussian elimination is for dense matrix.
- Solving system of linear equation by getting rid of variables one by one by rewriting them in terms of other variables.

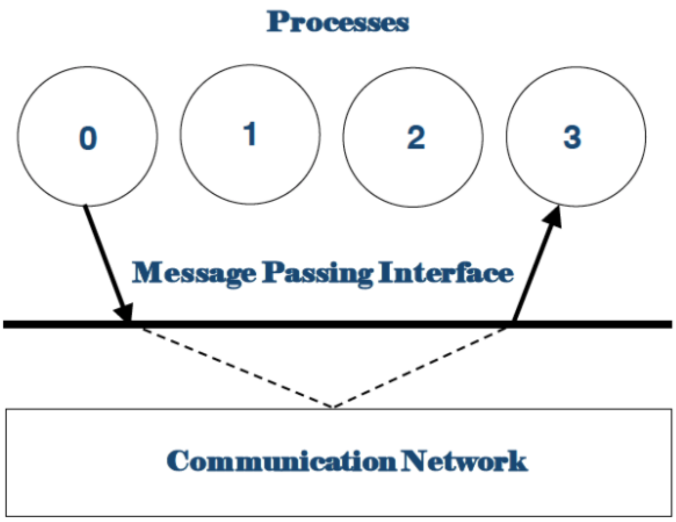


- If a coefficient of a variables is close to zero, then we run into numerical problems.
  - We can swap this row with the rows below to fix this. This is called *partial pivoting*
    - Partial because the columns aren't being swapped.
    - Can be done in $\mathcal{O}(\log_2 n)$
    - We will ignore partial pivoting to simplify our problem. (make it more theoretical)
- The eliminate step can be parallelized. Resulting in time complexity of $\mathcal{O}(n^2)$ with $p = n$
  - We can more better utilize processor if there are less than n processor by using *cyclic-striped partitioning*.

processor 1
processor 2
processor 3
processor 1
processor 2
processor 3
processor 1
processor 2

MPI

- Pass messages between processes.



- Provide consistent interface to have portable code on different architectures.
- Can be sync or async
  - sync: returns after message being read by receiver process.
  - async: returns asap and have some other thread or the kernel do the sending.
- MPI is a language.

Communicator

- Communicators: a set of processes.
  - `MPI_COMM_WORLD`: all processes.
- Rank: 0 based index of your processes given a communicator.
- Size: size of the communicator.

Functions

- `int MPI_Init(int *argc, char ***argv)`: first MPI call, initialize the MPI execution environment, takes away the MPI arguments by modifying `argc` and `argv`.
- `int MPI_Finalize()`: last MPI call.
- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,`
  `MPI_Status *status)`
  - For send and receive to succeed, rank must be valid, communicator must be same, tags must be same (or say idk what tag), message datatype must be compatible.
- `int MPI_Bcast(void *buf, int count, MPI_Datatype, int root, MPI_Comm comm)`
- `int MPI_Scatter(void *buf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - One node send different data to other nodes.
- `int MPI_Gather(void *buf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - Opposite of scatter.
- `int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
- `int MPI_Alltoall(void * sendbuf, int sendcount, MPI_Datatype sendtype, void * recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm )`
- `int MPI_Alltoallv(void * sendbuf, int * sendcounts, int * sdispls, MPI_Datatype sendtype, void * recvbuf, int * recvcounts, int * rdispls, MPI_Datatype recvtype, MPI_Comm comm)`
- `int MPI_Reduce(void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
  - `MPI_Op` can be: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, `MPI_LAND`, `MPI_BAND`, `MPI_LOR`, `MPI_BOR`, `MPI_LXOR`, `MPI_BXOR`.
- `int MPI_Barrier(MPI_Comm comm)`
  - Mostly used to share OS resource are not controlled by MPI.
- `MPI_Ssend`: sync
  - `MPI_Send` can be sync or async
  - Sync send can deadlock if the receiver is waiting for a message with a different tag.
- `MPI_Bsend`: Async
  - Many other type of `MPI_Send`
  - Async send can be out of order.
  - Can check status to check tag.
- `MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)`
- `int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)`
  - Create cartesian topology
- `int MPI_Dims_create(int nnodes, int ndims, int *dims)`
  - Will fill in the zeros in `dims`, will try to make dimensions as close to each other as possible.
- `int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)`
  - Map coordinates to rank.
- `int MPI_Cart_coords(MPI_Comm comm, int rank,, int maxdims, int *coords)`
  - Map rank to coordinates.
- `int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)`

- Source is who will send to me
- Dest is who will I send to

- `int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - Take the result of my source and replace my data which will be send to my dest.
  - Can use with `MPI_Cart_shift` but not necessary.

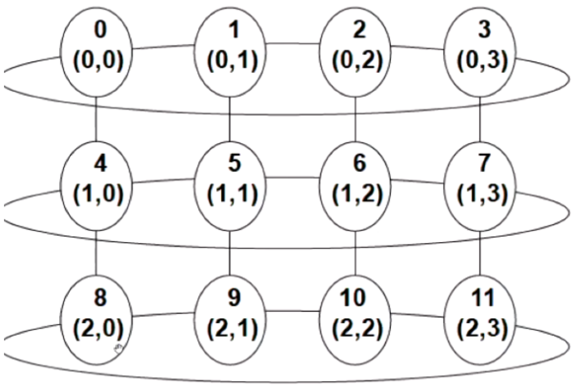
- `MPI_PROC_NULL`: send or receive from and to `MPI_PROC_NULL` does nothing so we don't have to move forward.

Multithreading

- 4 levels options of support:
  - 0, `MPI_THREAD_SINGLE`: only one thread will execute.
  - 1, `MPI_THREAD_FUNNELED`: only one thread will make MPI calls.
  - 2, `MPI_THREAD_SERIALIZED`: calls well never be concurrent.
  - 3, `MPI_THREAD_MULTIPLE`: No restrictions.
- Call `MPI_Init_thread(int *argc, char ***argv, int required, int *provided)` instead of `MPI_Init` to declare you need multithreading.

Topologies

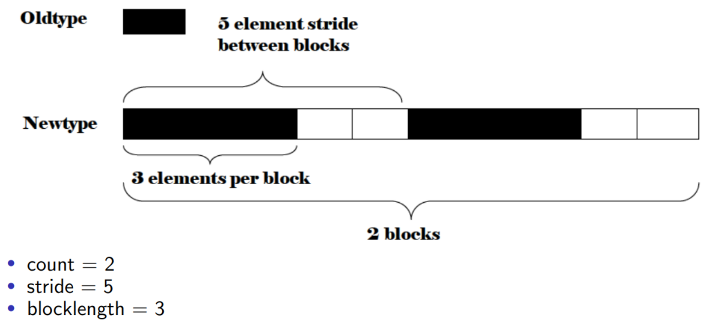
- Examples: ring.
- MPI have function that allows you to refer to "node below me" which will calculate the rank for you.
- Cylinder:



- Cartesian: a grid that might or might not be cyclic with each dimensions.
- Graph topologies: explicitly list neighbours of each node

Derived Types

- Can define new MPI_Datatype.
- Can be used in place of any MPI_Datatype can.
- Types are defined with other list of types and displacements (in bytes).
- `int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`: just an array of the same type.
- Vector Datatype:



  - Good for sub blocks of a matrix.
  - `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`
- `int MPI_Type_create_struct(int count, const int array_of_blocklengths[], const MPI_Aint array_of_displacements[], const MPI_Datatype array_of_types[], MPI_Datatype *newtype)`
  - Can represent any struct.
- After create datatype, must call `int MPI_Type_commit(MPI_Datatype *datatype)`
  - Can use uncommited datatype to build other data type.
- `int MPI_Type_free(MPI_Datatype *datatype)`
  - Free a committed type's memory.
- Count $\times$ Fundamental datatype count must match for send and receive.

GPU

- GPGPU: General purpose GPU
- Many CUDA core / Streaming processor make up of Streaming multiprocessor.
- GPU have:
  - Threads: smallest execution entity, each have their own id.
  - Block / warp made up of threads that execute in a single multiprocessor in sync. Every thread runs the same line of code at the same time.
    - Blocks are SIMD.
  - Grid: a bunch of blocks that execute a kernel function (kinda like a regular function).
    - Blocks are running the same code but not in lock step.
    - Blocks are independent to each other, can go in any order.
- GPU have memory that's high throughput.
  - Global memory: main memory 100ms
    - IO for grid
  - Shared memory 128kB: shared memory 5ns about the speed of L1 cache.
    - per block
    - Can use for collaboration within a block.
  - Register and local memory: fastest, around 10x faster than shared memory, not as fast as registers on CPU.
    - Per thread
    - Store stack vars.
- Compared with CPU
  - CPU wants to run one thread very fast: Sophisticated control, powerful ALU, Large cache.
  - GPU wants high throughput: simple control, small caches, many efficient ALU.
- CUDA: Compute Unified Device Architecture.
  - Extends C/C++, can run on both GPU and CPU.
  - Abstract from the hardware, same code run on many GPU.

- Auto thread management.
- Vendor-lock to Nvidia.
- Hard to debug, don’t have printf in early versions.