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

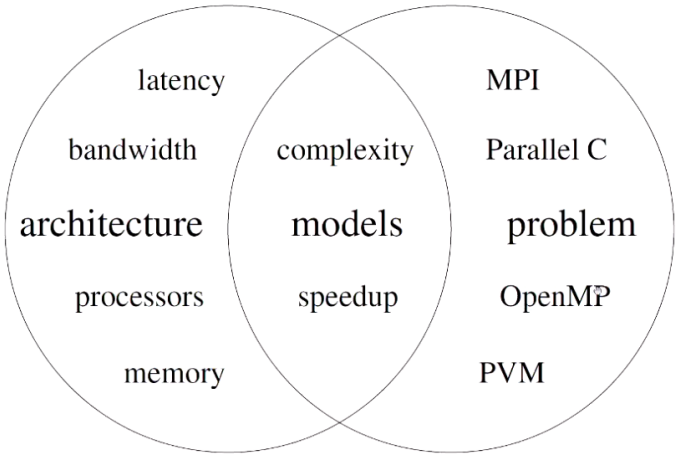


Figure: Overlapping aspects of parallel computing.

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
- $f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)) \Rightarrow f(n) = \Theta(g(n))$
- 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.
  - 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*: 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 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 Speedup$(p) = \Omega(p)$
  - Very obvious again...

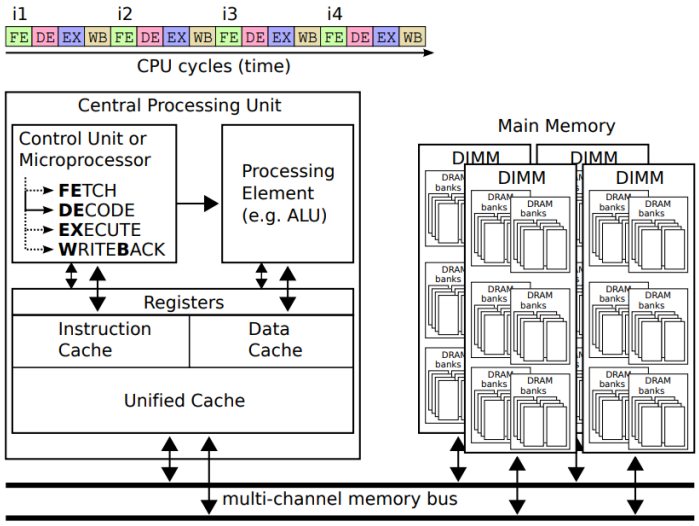
Algorithms

- 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 × 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 → 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

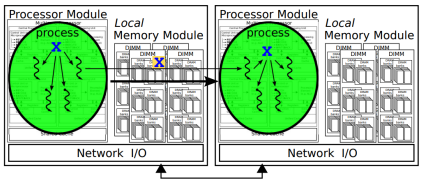
- Synchronization is more expensive if threads are on cores further away.
  - It's expensive in general.
- Instruction reordering: thread continues with other instructions while it waits on earlier instructions.
- Speculative execution: don't wait on instructions, just go for it and if it fails then unroll.
- Some programming patterns are more friendly to NUMA.

Message passing considerations

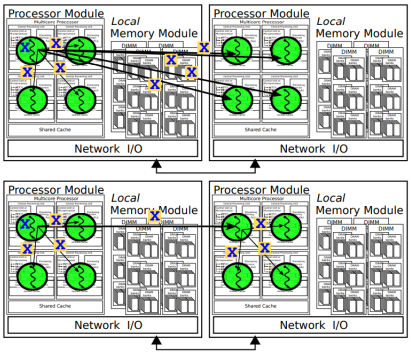
- Multi processing have to pass messages around because processes don't share address space.
- Hard to predict performance.

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.