CS.590.03.Fall.2015: Parallel Computing. Midterm Summary.

Ellie Zheng
10/17/2015

Contents:

I. System and programming models
  1. Parallelism. Shared memory. Data parallel (Vecterization)
  2. Potential problems: races. sync. dependencies.
  3. Performance
  4. GPU. Offload.

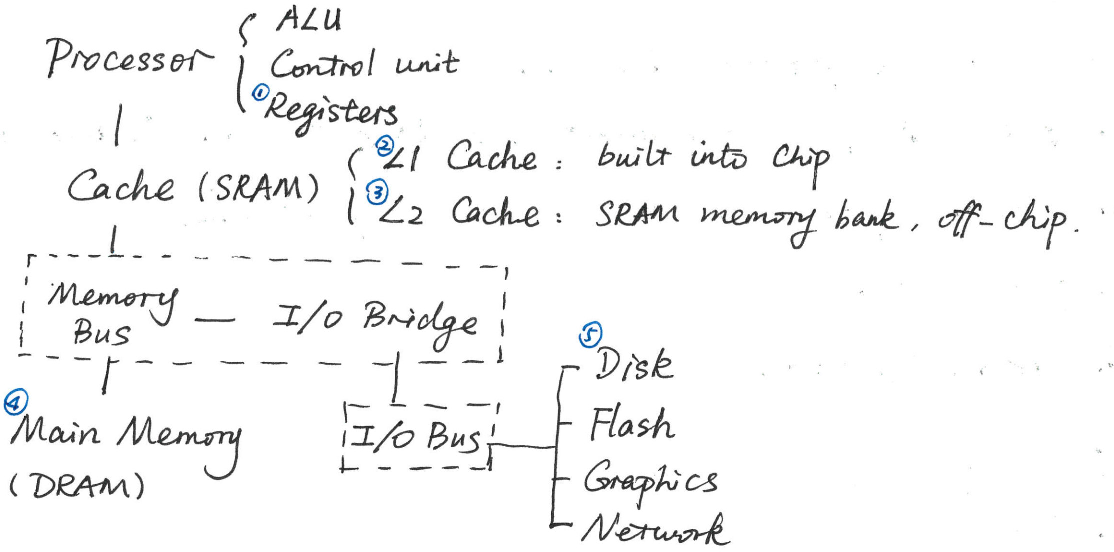
II. Patterns
  1. Map
  2. Collective: reduce. scan
  3. Data Reorganization: gather, scatter. pack. partitioning data
  4. Stencil: steneil. recurrence.
  5. Fork-join
  6. Pipeline

III. Models; Implementations
  1. Open-MP. Cilk-Plus. Threaded Building Blocks (TBB)
  2. CUDA

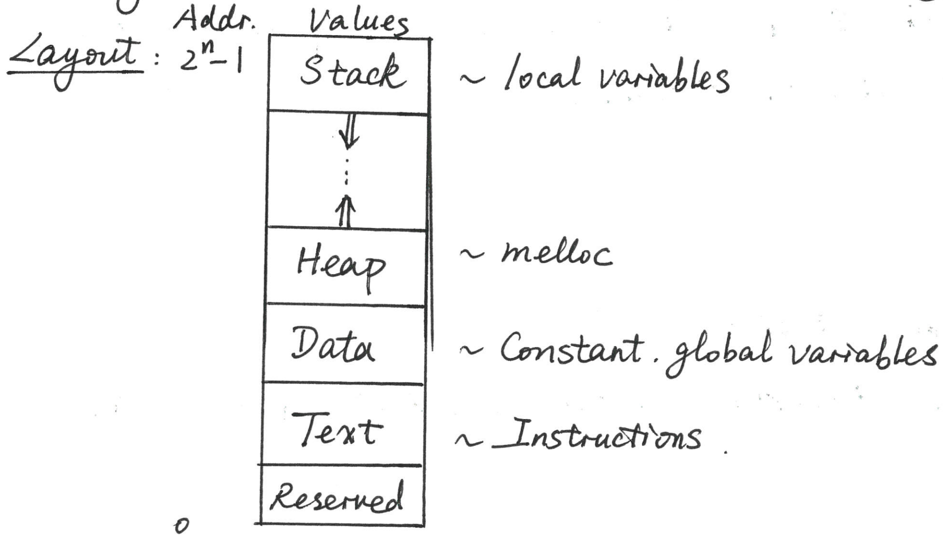
Part I: System.

0 Basics about the architecture.

i. System organization:

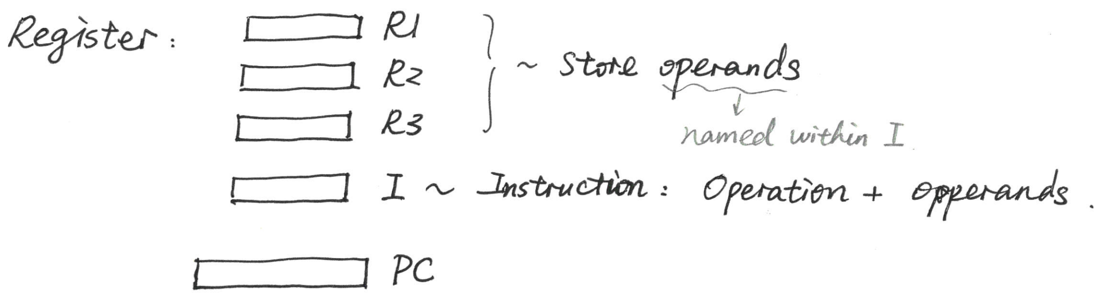


ii. Memory hierarchies: ① → ② → ③ → ④ → ⑤



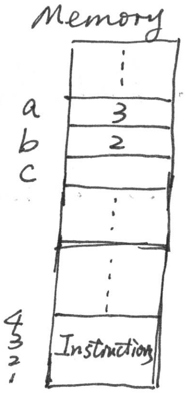
↓ variables are names for memory locations.

iii. Instruction Set Architecture (MIPS): LOAD/STORE ISA.



For example, $a+b \to c$

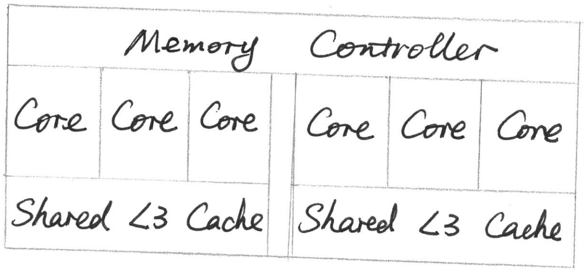
PC	Instruction	R1	R2	R3	C
1	Load a, R1	3	/	/	/
2	Load b, R2	3	2	/	/
3	Add R1, R2, R3	3	2	5	/
4	Store R3, C	3	2	5	5



Another type of ISA is called 2-address ISA.

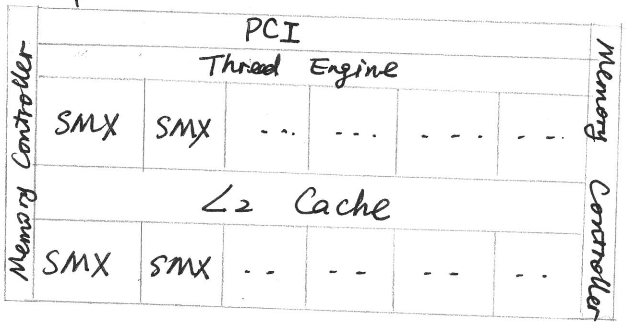
iv. Shared memory multipossessors

(1) Intel i7:



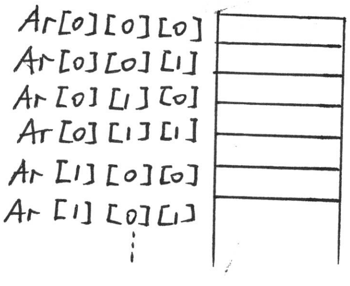
+ SIMD

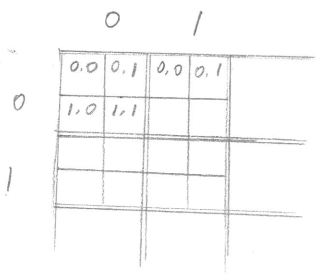
(2) NVIDIA Kepler



Each SMX has 192 cores, 2048 threads.

v. Memory layout of N-dimensional arrays: ROW-MAJOR MAPPING.



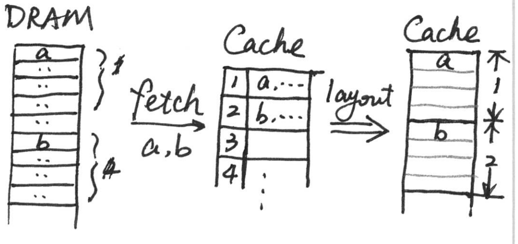


vi. Cache organization:

① Cache fetch data in Chunks called "blocks"

② Each block fits into a "block frame"

③ Block frames map into memory of address



I. Parellelism, Programming models

1. Basic Concepts

i. Parellel processing = concurrency + parellel hardware

ii. Parellel programming model:

- Shared memory
- Data parellel
- Message passing

* Note that parellism, threads, ... can be based on either hardware level or software/programming leve They should be treated seperately and clearly.

Note that it's from the programers' point of view, not how system actually works.

iii. Hardware has mixed types of parallelism, and can run most of the above programming models. Hardware threads can generate software threads.

2. Shared Memory (↔ Message passing)

i. Basic idea: ① Processors read from and write to the same variables in shared memory.

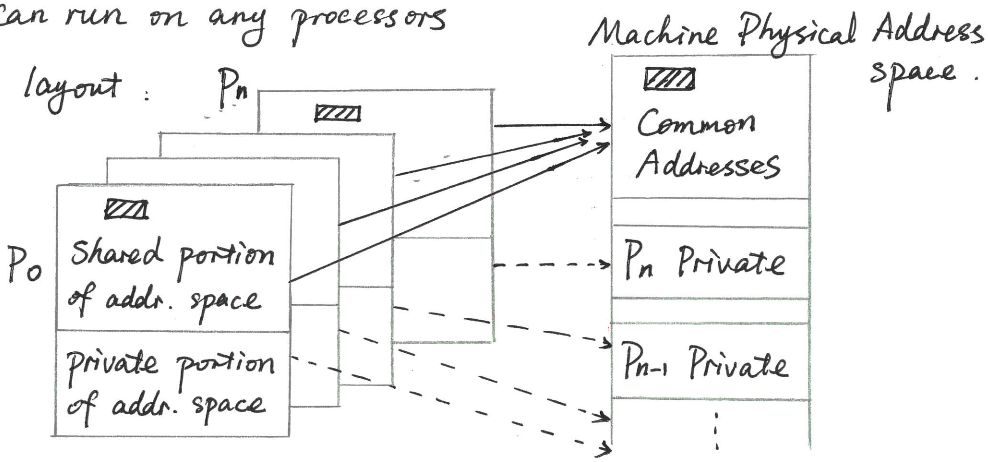
② Variables can be shared and private.

ii. Hardware Support:

① Processors connect to one shared memory

② Kernel can run on any processors

③ Memory layout:



④ More supports on specific issues in the following sections.

iii. Issues related to concurrency.

① Data Race $\xrightarrow{\text{Sol.}}$ Atomic sequence of instructions: <u>mutex</u>

↓ non-deterministic

↓ sequence of operations appears to execute to completion without any intervening operations

- LOCK (counter → lock);
  Conter → value = ... ;
  UNLOCK (counter → lock);
- while (test & set (counter → lock));
  counter → value = ... ;
  counter → lock = 0 ;

② Fine-grain locking v.s. a highly contended locking

↓ e.g., a node in a graph

↓ e.g.. a whole graph.

③ Deadlock : when code requires locking of multiple mutexes at once.

- avoid mutexes by replicating the resources.
  e.g., pipeline.
- hold only one lock at a time.
- acquire locks on multiple mutexes in the same order.

T_1	T_2
lock(x)	
	lock(y)
lock(y)?	
	lock(x)?

iv. Global synchronization : Barrier

- wait for all tasks to enter, then allow all to continue.
- is another type of sync other than lock.

3. Data Parallel (↔ task parallel)

i. Basic idea: ① Perform same operation (instruction) on different elements simutanously.

② It emphasizes the distributed nature of data, not the processing.
(task parallelism)

③ Not conflict with task parallel;
rather, it exploits another level of parellelism within one thred

④ Scalar execution ⟶ vector execution

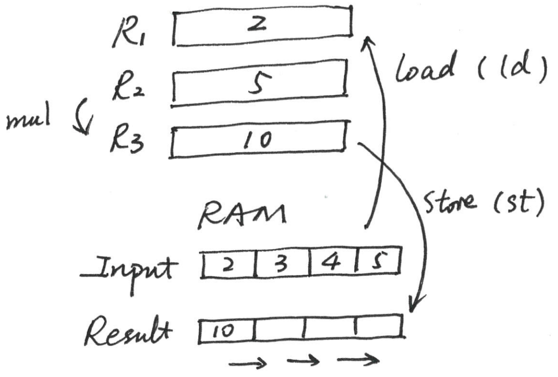
↓ { one instruction
one set of data in reg.

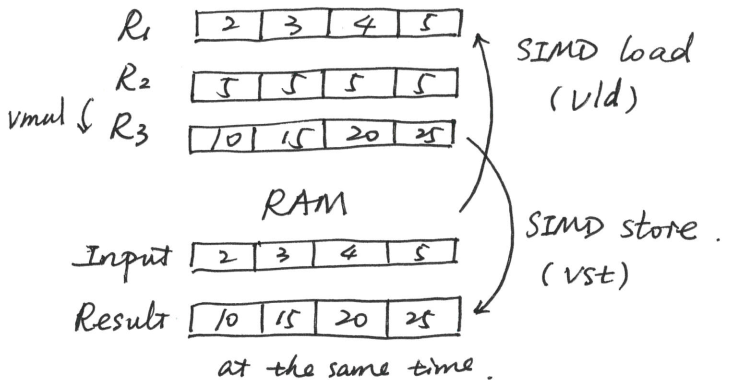
↓ { one instruction
~~mulp~~ multiple sets of data in reg.

ii. SIMD (single instruction multiple data) / Vectorization

↳ might need mutating loops.

- scalar execution :
- Vector / SIMD execution





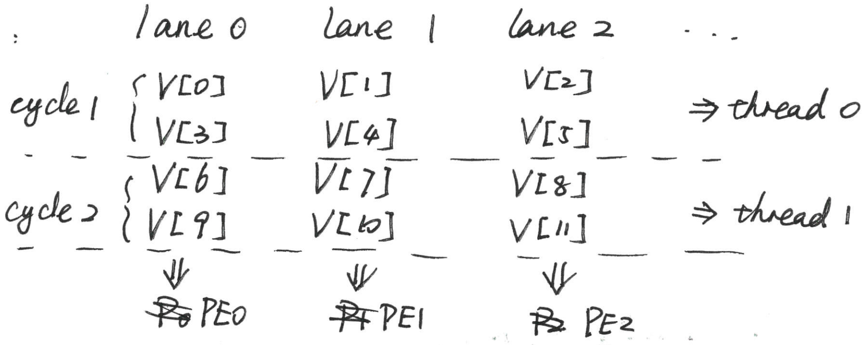
- Conditional execution : add vector flag registers (single-bit mask per ele)

```
e.g., for ( i=0; i<N; i++ ) {
          if ( A[i] != B[i] ) { A[i] = - B[i] ; }
      }
```

- Multithreaded execution :

```
while ( ! done )
    fetch thread i instruction
    execute instruction
    i = i+1 mod # HW threads.
```

- Multiple data lanes : (require data reorg.)



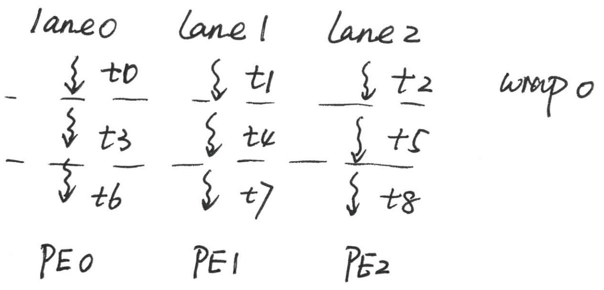
	lane 0	lane 1	lane 2	. . .
cycle 1	V[0] V[3]	V[1] V[4]	V[2] V[5]	⇒ thread 0
cycle 2	V[6] V[9]	V[7] V[10]	V[8] V[11]	⇒ thread 1
	⇓ ~~P0~~ PE0	⇓ ~~P1~~ PE1	⇓ ~~P2~~ PE2	

⇓ processing element (core)
each of them has multiple threads

- Hardware Support : (GPU is better)
  - * Interleaved threading .
  - * SIMD execution .

iii. SIMT (Single instruction multiple threads)

- Threads are grouped into wraps.
- # threads per ~~g~~ wrap = # lanes
- Hardware support :
  - * Massive threads per core
  - * wrap schedulers .



lane 0	lane 1	lane 2	
t0	t1	t2	wrap 0
t3	t4	t5	
t6	t7	t8	
PE0	PE1	PE2	

- possible issues : Divergence.
  - * Control divergence
    - ∘ Conditional branches.
      e.g., if (block others) ; execute ;
      else block.
  - * Memory address divergence
    - ∘ delay of memory access, while others are waiting.
      e.g., when try to access to different cache blocks.

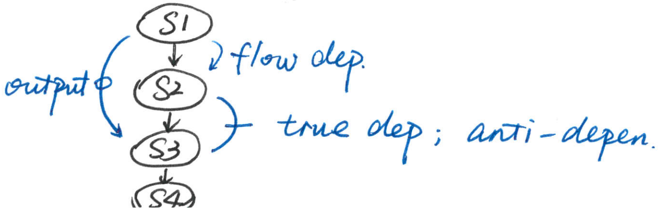
II. Dependencies.

1. Independent v.s. Dependent :
   whether the order of their execution affects the computational outcome.
2. Comparisons of three types of dependencies.

	True / Flow ~	Anti-dependence	Output dependence
Defination	S2 reads a value writen by S1	S2 writes a value read by S1	S2 writes a value written by S1
Notation	S1: X = □ ⋮ S2: □ = X δ	S1: □ = X ⋮ S2: X = □ δ^{-1}	S1: X = □ ⋮ S2: X = □ δ^{o}
Can remove?	NO	use different variable names. Rearrange / eliminate statements.	

3. Directed Acyclic Graphs (DAG) : capture data flow parallelism.
   - Node : operation to be performed.
   - Arc : path for data flow ⇒ TRUE dependency.

Example :
S1 : a = 1
S2 : b = a
S3 : a = b+1
S4 : c = a



III. Performance.

1. Evaluation of performance: // resources for solution

i. latency: total time to complete a task.

ii. throughout: rate at which a series of tasks can be completed.

iii. power consumption

2. Performance metrics

i. Speedup: $S_p = \frac{T_1}{T_p}$ → execution time on a single processor (sequential time) / → - - - - - - - p-processor system.

ii. Efficiency: $E_p = \frac{S_p}{p}$

iii. Cost: $C_p = T_p \cdot p$

iv. Scalability: Ability of parallel algorithm to achieve performance gains proportional to:

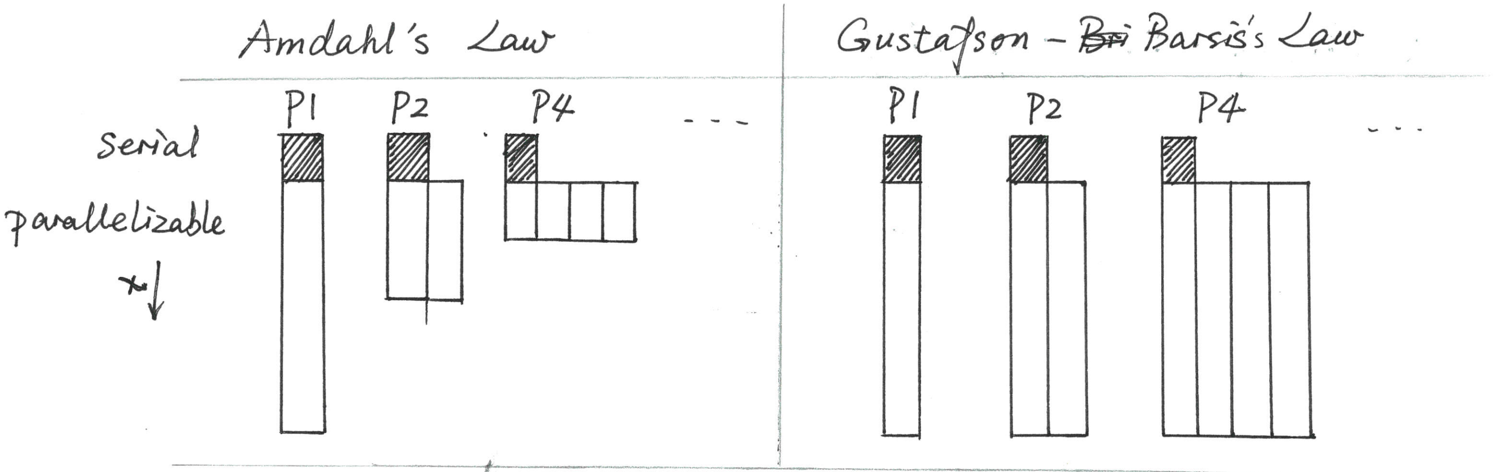
- the number of processors.
- the size of the problem.

Defination of scalability can be in terms to

Amdahl's Law
⇑
constant problem
constant time ⇒ Gustafson
constant error

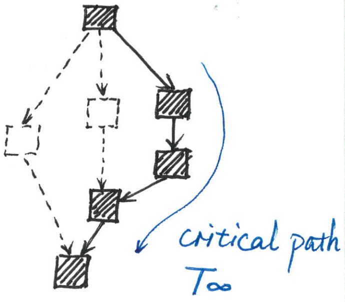
3. Two speedup models:

	Amdahl's Law	Gustafson-Barsis Law
when apply?	constant problem size	constant time (problem size increases when p↑)
scalability	Strong scaling	Weak scaling
Speedup	$S_p = \frac{1}{f + (1-f)/p}$ where $T_p = fT_1 + (1-f)\frac{T_1}{p}$ $\Rightarrow S_\infty = \frac{1}{f}$ ⇒ sequential fraction	$S_p = 1 + (p-1)f_{par} = 1 + (p-1)(1-f)$ where $f_{par} = \frac{W_{par}}{W_{seq} + W_{par}} = 1-f$
feature	• Too optimistic • perfect efficiency is hard to achieve	• can maintain or increase parallel efficiency as problem scales.



4. **Work-Span Model** to estimate computation time.

i. Extreme times for $P=1$ and $P=\infty$



- Work : T_1 ↓ Serial execution
- Span : T_∞ ↓ time along critical path (path through DAG that takes the longest time)

ii. Lower / Upper bound on greedy scheduling.

- Lower bound : $T_P \geq \mathrm{Max}\left(\frac{T_1}{P}, T_\infty\right)$
- Upper bound : $T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty$

$\Downarrow$

Brent's Lemma : (time for critical path | addition time for other tasks not on the critical path

iii. Simplified running time estimation

- When $T_\infty \ll T_1$, $T_P = \frac{T_1}{P} + T_\infty$
  - * increasing work hurts T_P proportionally.
  - * Span impacts scalability.
- When $\frac{T_1}{T_\infty} \gg P$ (parallel slack), $T_P = \frac{T_1}{P}$
  - * linear speedup.
  - * over decomposition.

iv. Standard work-span model considers only computation, not communication or memory

5. Asymptotic Complexity to compare algorithms.

i. Time complexity
ii. Space complexity
→ describe how { execution time / memory requirement } grow with input size.

Notations: $O(f(N))$: upper bound
$\Omega(f(N))$: lower bound
$\Theta(f(N))$: both upper and lower bound.

6. Factors that influence the performance of parallel apps.

i. Sequential performance.

ii. Critical paths

- problem : * long chain of dependence spread across processors.
- solution : * eliminate long chain
  * removing works from critical path.

iii. Bottlenecks

- problem : * one processor holds things up.
  * sending / collecting data between processors when p is large.
  * show up when scaling.
- solution : * more efficient communication
  * more layers for master slave.

iv. Algorithmic overhead.

- problem : * parallel algorithm introduces additional operations
- solution : * Choose algorithmic variants that minimize overhead.
  * Use two-level algorithms.

v. Computational overhead.

vi. Load Imbalance

- problem: uneven distribution of work.
- solution: * overdecomposition (# tasks ≫ # workers)
  * work-stealing scheduler.

vii. Speculative Loss

- problem: do A and B in parallel. but B is ultimately not needed.

7. Performance Analysis and Tuning.

- goal: reduce wall clock execution time.
- how: find the Hot spot and Bottleneck elimination.

IV. GPU and Offload. (Heterogeneous Computing)

1. GPU

$\text{CPU} \longleftrightarrow \text{GPU}$

i. Memory management

- declare functions to GPU: __global__
- use pointers to transfer variables, through: cudaMemcpy.
- allocate / free memory on GPU: cudaMalloc / free.
- execute functions on GPU: through myfunc<< M,N >>(*variables)

ii. Parallelism

- use multiple blocks to run in parallel.
- use multiple threads for a block to introduce additional parallelism

2. Offload to MIC

$\text{CPU} \xleftrightarrow{\text{PCI}} \text{MIC}.$

i. Memory management

- declare functions and variables to MIC: __attribute__
- use in/out to transfer variables / pointers.
- allocate memory for pointers only; free ~~of~~ after offload by default.
- execute codes on MIC by: # pragma offload target (MIC)

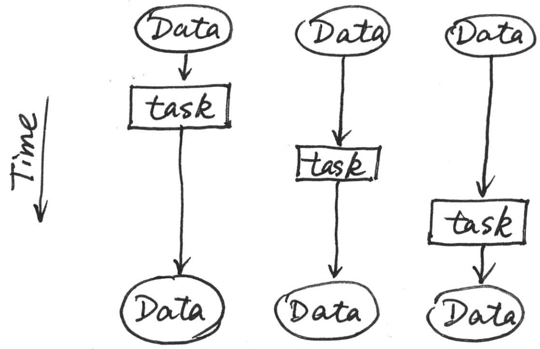
ii. Parallelism: within offload target (MIC), the parallelization is realized in the same way as in CPU.

Part II : Patterns

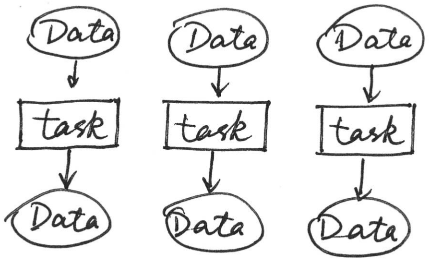
General note : patterns are universal themes and idioms that can be used in any system and algorithm. They can be serial or parallel. What we want to do here is trying to parallelize the patterns.

I. Map : **embarrassing parallism.**

1. Serial v.s. parallel pattern :



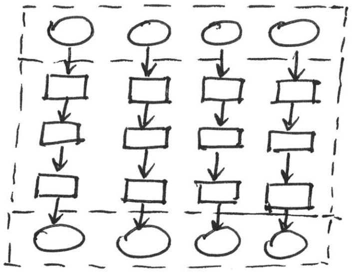
serial map



parallel map.

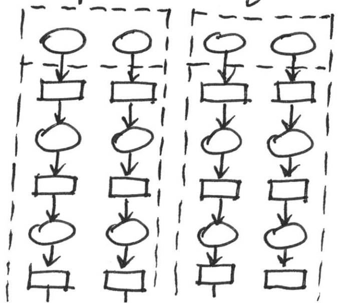
2. Application
   - Completely independent operations. e.g., vectorization.
   - Scaled Vector Addition (SAXPY)
3. Optimization

   i. Code Fusion : map of sequence v.s. sequence of map.



keep intermediate data in registers.

   ii. Cache Fusion : process sequences of maps in small tiles sequentially

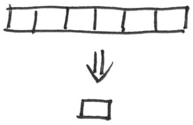


intermediate data might reside in the cache.

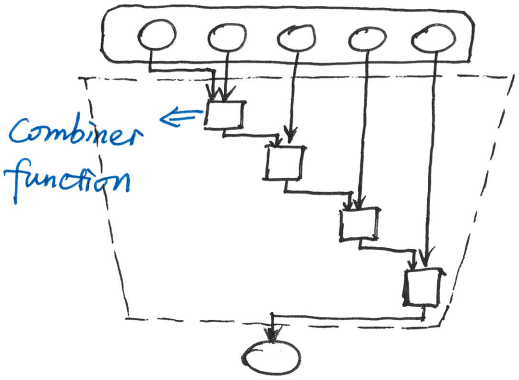
II. Collective.

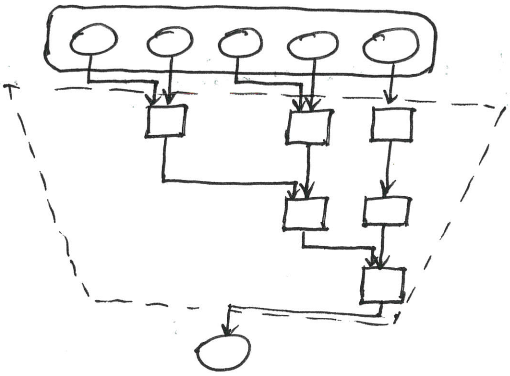
1. Reduce

i. Main feature :



ii. Serial v.s. parallel reduction :

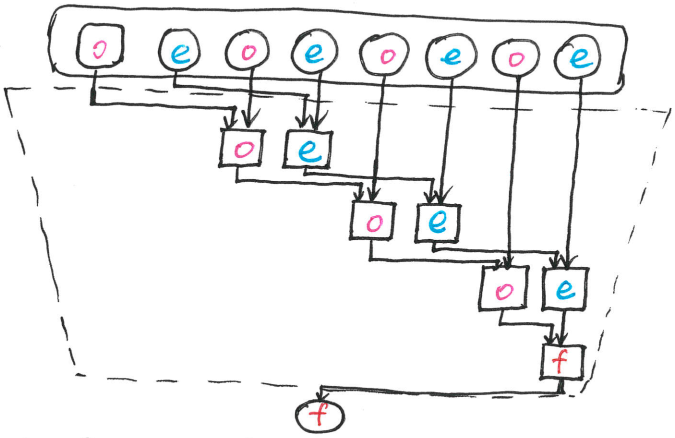




iii. Application

- addition
- multiplication
- max / min.

iv (a)



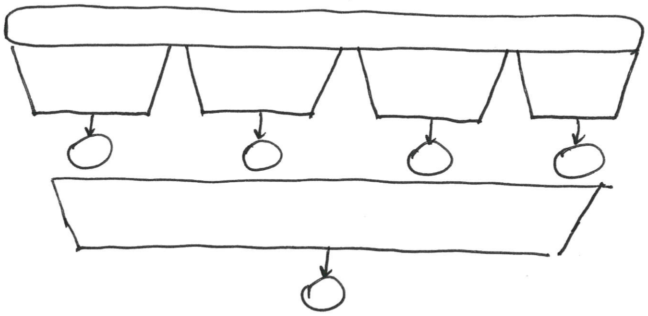
iv. Optimization → reordering.

(a) Vectorization

e.g., two-lane SIMD : combine odd & even elements seperately.
may need data reorg.

(b) Tiling.

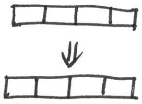
→ reduce the storage from O(n) to O(1) by operating each tile ~~batch~~ in serial or vectorization, rather than the tree ordering.



(c) Fuse reduction with maps.

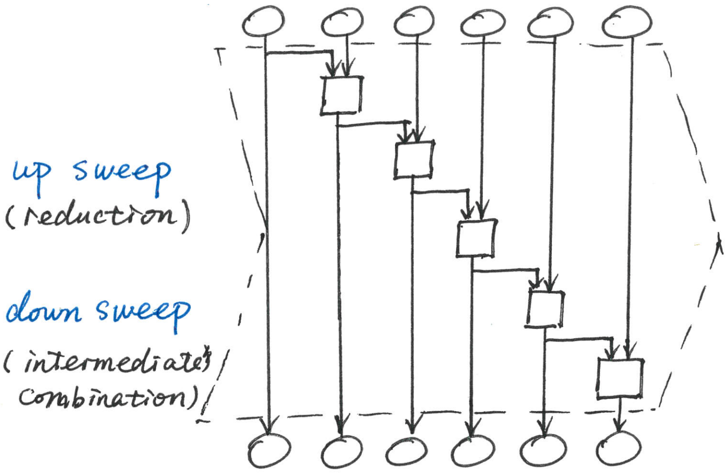
2. Scan

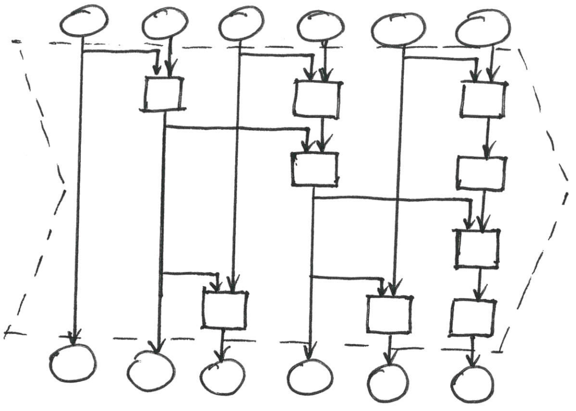
i. Main feature:



(prefix computation)

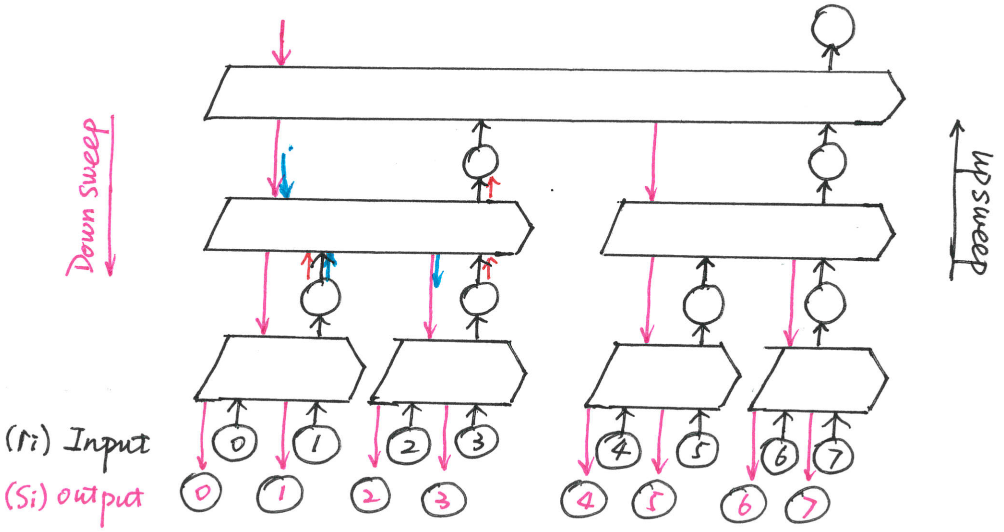
ii. Serial v.s. parallel scan

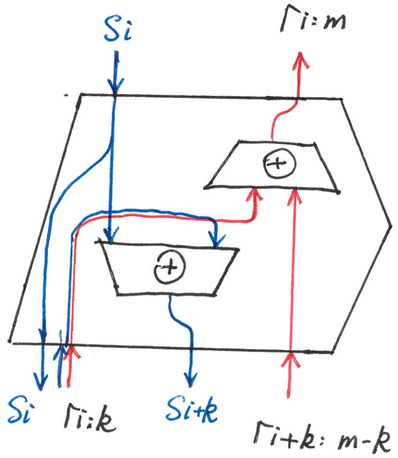




iii. Application: maximum.

iv. Implementation with Fork-Join:

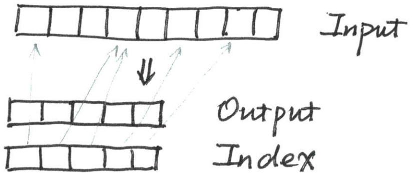




III. Data Reorganization.

1. Gather

i. Main feature:



n_index = n_output

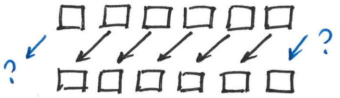
random read.

ii. Serial v.s. parallel gather

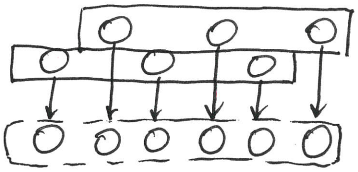
```
void gather (
        size_t  n_input;
        size_t  n_index; // = n_output;
        Data    input[ ];
        Data    output[ ],
        Idx     index[ ]
){                                  (map) parallel_for to parallelize it.
    for ( size_t i=0; i< n_index or n_output; ++i){
        ~~size_t assert ( 0<= index[i]~~
        size_t  index_value = index[i];
        assert ( 0<= index_value && index_value < n_input );
        output[i] = input[index_value]; // perform random read.
    }
}
```

iii. Special cases of gather.

- Shifts
  - * requires handling boundary condition
  - * can be done via vectorization.

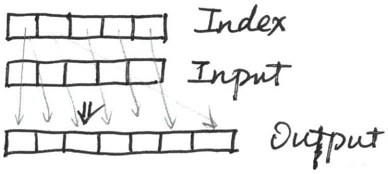


- Zip (interleave)
  - * Application : convert between SOA and AOS
    combine two data sets.



- Unzip ⇔ zip.

2. Scatter



n_index = n_input

random write.

i. Main feature.

ii. Serial v.s. parallel scatter

```
void scatter (
        size_t  n_input; // n_index
        size_t  n_output;
        ----
){
```

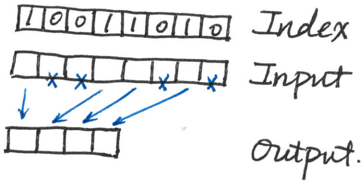
```
    for ( size_t i=0; i< n_index or n_input ; ++i ) {
        size_t index_value = index[i];
        assert ( 0<= index_value && index_value < n_output);
        ~~output[i] = input~~
        output[index_value] = input[i]; // perform random write.
    }
}
```

iii. Possible ~~solu~~ problem: collisions / race conditions.

- Atomic scatter: no rule determines which will be retained
  ⇒ ~~determin~~ non-deterministic.
- Permutation scatter:
  - * collisions are illegal → check in advance.
  - * turn scatter into gather.
  - * e.g., matrix / image transpose.
- Merge scatter:
  - * use addition as the merge operator
  - * require both associative and commutative operators
- Priority scatter
  - * which one is retained is based on a priority assigned according to position.
  - * e.g., 3D graphics rendering.

3. Pack

i. Main feature:

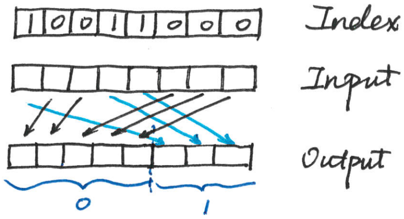


ii. Algorithm:

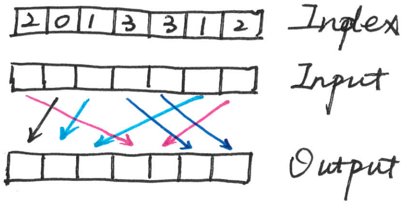
- Booleans (true / false) → integer 0's and 1's.
- Scan of this array with addition.
- write values to output array based on offsets.

iii. Generalization of pack :

- Split



- Unsplit : inverse of split.
- Bin :
  - * $ Index supports more categories
  - * e.g., Radix sort.



iv. Fusing map and pack

- map : check pairs for collision
  pack : store actual collisions.
- good when most of the elements of a map are discareled.

4. Partitioning Data

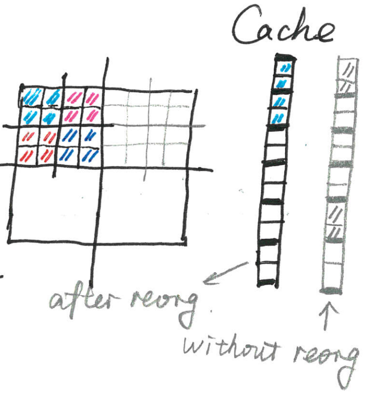
i. Partitions : non-overlapping, equal sized.

ii. Segmentation : non-overlapping, non uniform.

} can be 1D, 2D, 3D

e.g., store sparse matrices.

iii. Layouts of partitioning : making tiles contiguous to enhance cache utilization.



- 4-D block

		1		2	
		1	2	1	2
1	1	0	1	2	3
	2	4	5	6	7
2	1	8	9	10	11
	2	12	13	14	15
D	T				

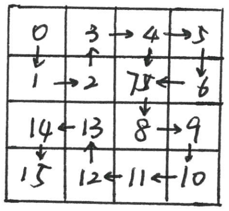
Ar[D][D][T][T]

- Morton order

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Ar[i][j] ⇒ Ar[f_m(i,j)]

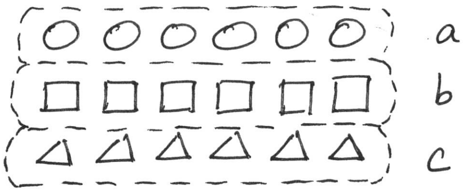
- Hilbert order



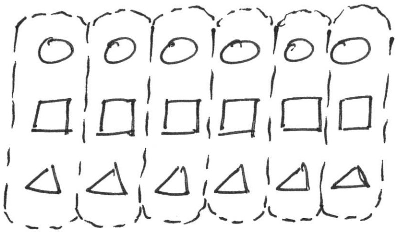
Ar[f_h(i,j)]

↘ each element is adjacent to its neighbor.

iv. Array of Structures (AoS) v.s. Structures of Arrays (SoA)



- better for vectorization
- myarray. a [i]

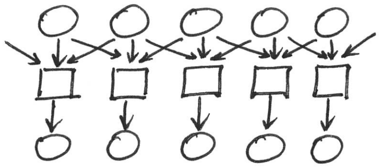


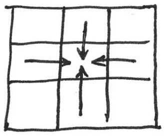
- better for randomly accessing data
- myarray [i] . a

IV. Stencil & Recurrence

1. Stencil

i. Main feature : map + neighboring input elements



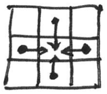


ii. input ≠ output

- same parallelism implementation as ~~in~~ map.
- other requirements : define offsets ; add boundary conditions

iii. input ⇐ output : in-place update

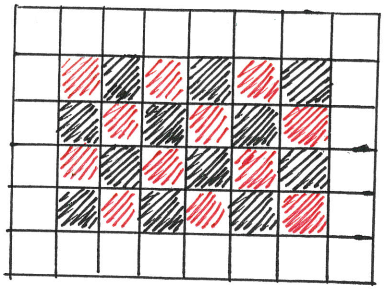
- problem formation :
  - * update alternatives
  - * towards convergency.
  - * iterative codes
- Applications :



  - * solve PDE : 2D-Jacobi iteration (4-point stencil)
  - * fluid dynamics
  - * Conway's game of life.

- parallelism : { Successive Over Relaxation (SOR)
  | Jacobi Iteration.

(a) Red / Black SOR



* write to red~~d~~ cells, read from black

* write to black cells, read from red

faster convergency than Jacobi iteration by "overpredicting" new solution.

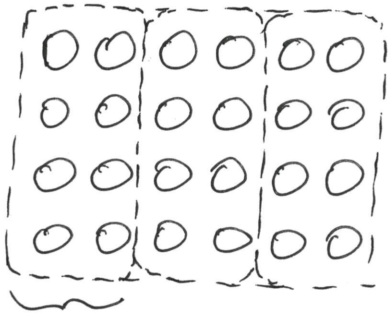
(b) 2D Jacobi iteration

* update each element to the average of N,S,E,W neighbors.

* parallelizable : by partitioning into blocks and using halos.

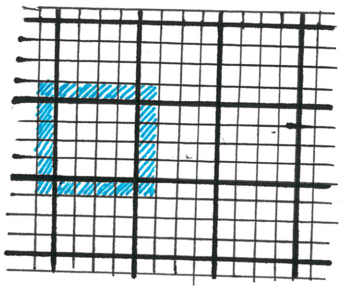
iv. Cache optimization : Strip-mining

- Basic ideas : * Assign "strips" (multiple columns) to each core
  - * Strip's size = { width : m * sizeof(cacheLine)
    | depth : height of array.
- Advantages : * avoid redundant memory access (than assigning rows to cores)
  - * avoid false sharing (than assigning random ~~colu~~ # of columns to cores)



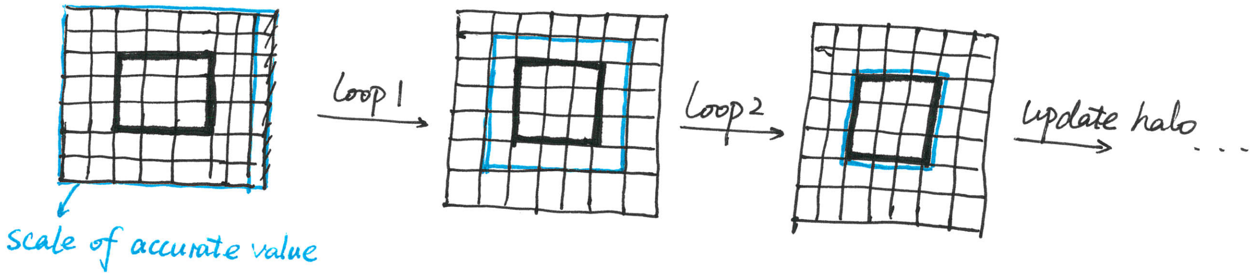
v. Communication optimization

- ghost cells



  - * a copy of ghost cells is kept in each thread's local ~~computations~~ memory
  - * Update ghost cells after each loop. (swap)
  - * Fine-grained sharing can lead to increased communication cost.

- Halo :
  - * Set of ghost cells.
  - * larger halo (deep halo) : ① can update (swap) ghost cells after many iterations.
    ② trade-off : more memory & computations
  - * Latency hiding : compute interior of stencil while waiting for ghost cell updates.



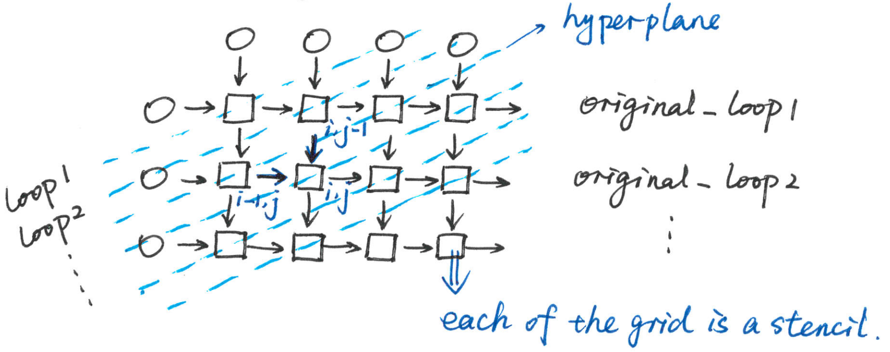
2. Recurrance

i. Main feature : stencil + dependencies between loops.

e.g., $a[i][j] = f(a[i-1][j], a[i][j-1]\}, b[i][j])$

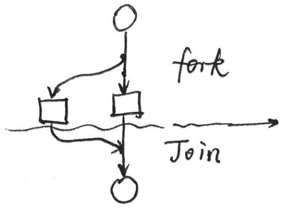
ii. Parallelism : separating hyperplane (wave front)

- which cut through grid of intermediate results.
- operations on the same hyperplane are executed simutanously.
- computation proceeds ⊥ hyperplane.
- Angle of hyperplane : make sure each iteration has constant number of tasks.



V. Fork-Join

1. Main feature :



```
if ( size< □)
    // parallel base case
else
    // divide and conquer
    Fork; // create tasks
    Join  // wait for child task completion
```

2. Applications :

   i. Recursively implementation of different algorithms and patterns

3. Optimization :

   i. Choosing base cases

      - create parallel slack by over decomposing the problem.
      - not go too deep to avoid too many scheduling overhead.