CS260 – Algorithmic Engineering

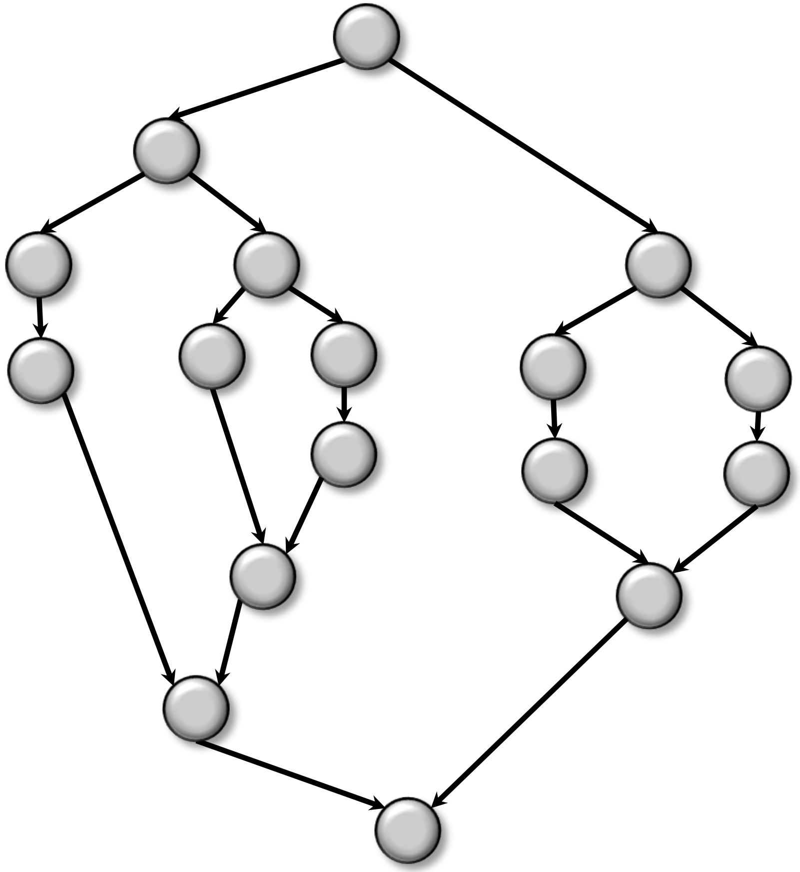
Yihan Sun

Parallel Algorithms and Implementations

* Some of the slides are from MIT 6.712, 6.886 and CMU 15-853.

Last Lecture

- **Scheduler:**
  - Help you map your parallel tasks to processors
- **Fork-join**
  - Fork: create several tasks that will be run in parallel
  - Join: after all forked threads finish, synchronize them
- **Work-span**
  - Work: total number of operations, sequential complexity
  - Span (depth): the longest chain in the dependence graph



Can be scheduled in time: $O\left(\frac{W}{p}+S\right)$ for work W, span S on p processors

Last Lecture

- Write C++ code in parallel

Pseudocode

```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

Code using Cilk

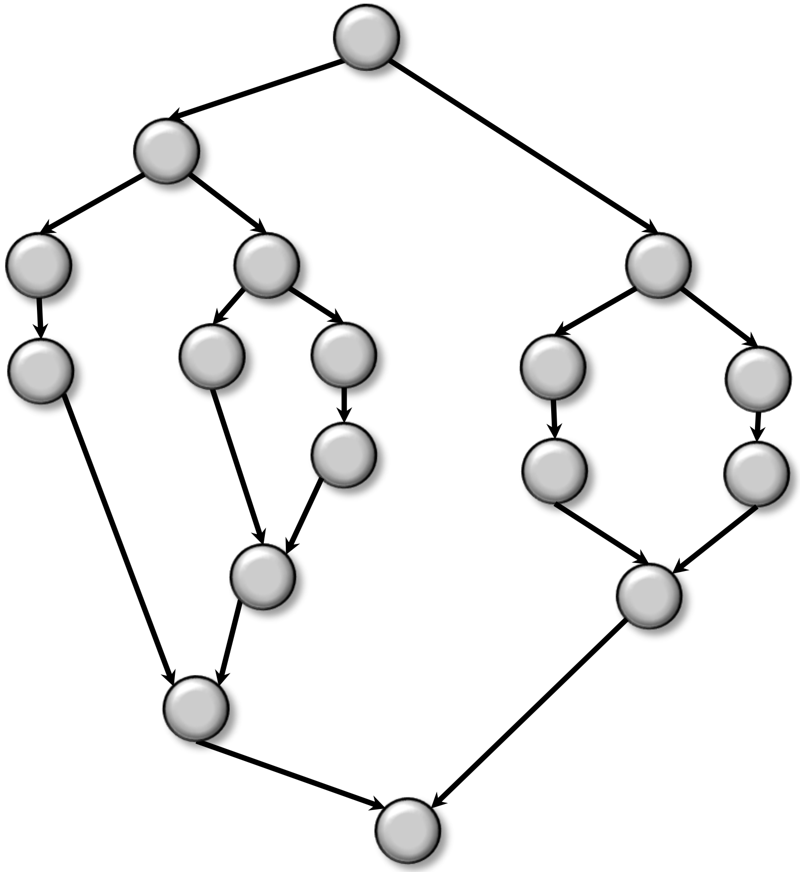
```
int reduce(int* A, int n) {
    if (n == 1) return A[0];
    int L, R;
    L = cilk_spawn reduce(A, n/2);
        R = reduce(A+n/2, n-n/2);
    cilk_sync;
    return L+R; }
```

Last Lecture

- **Reduce/scan algorithms**
    - Divide-and-conquer or blocking
- **Coarsening**
    - Avoid overhead of fork-join
    - Let each subtask large enough

Concurrency & Atomic primitives

Concurrency



- **When two threads access one memory location at the same time**
- **When it is possible for two threads to access the same memory location, we need to consider concurrency**
  - Usually we only care when at least one of them is a write
  - Race – will be introduced later in the course
- **Parallelism ≠ concurrency**
  - For the reduce/scan algorithm we just saw, no concurrency occurs (even no concurrent reads needed)

Concurrency

- **The most important principle to deal with concurrency is the correctness**
  - Does it still give expected output even when concurrency occurs?
- **The second to consider is the performance**
  - Usually leads to slowdown for your algorithm
  - The system needs to guarantee some correctness – results in much overhead

Concurrency

- Correctness is the first consideration!

A joke for you to understand this:
Alice: I can compute multiplication very fast.
Bob: Really? What is 843342×3424?
Alice: 20.
Bob: What? That’s not correct!
Alice: Wasn’t that fast?

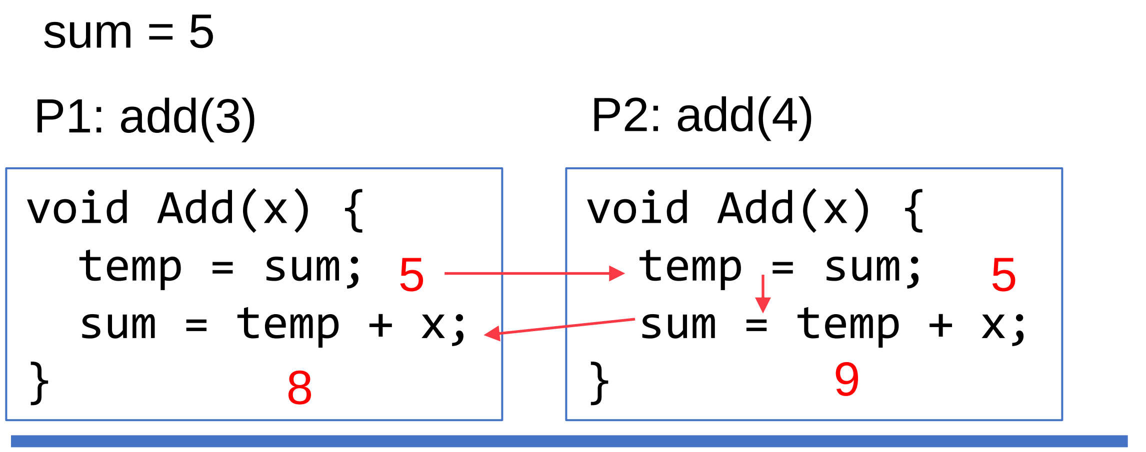
- Sometimes concurrency is inevitable
  - Solution 1: Locks – usually safe, but slow
  - Solution 2: Some atomic primitives
    - Supported by most systems
    - Needs careful design

Atomic primitives

- **Compare-and-swap (CAS)**
  - bool CAS(value* p, value vold, value vnew): compare the value stored in the pointer p with value $vold$, if they are equal, change p's value to vnew and return true. Otherwise do nothing and return false.
- **Test-and-set (TAS)**
  - bool TAS(bool* p): determine if the Boolean value stored at p is false, if so, set it to true and return true. Otherwise, return false.
- **Fetch-and-add (FAA)**
  - integer FAA(integer* p, integer x): add integer p's value by x, and return the old value
- **Priority-write:**
  - integer PW(integer* p, integer x): write x to p if and only if x is smaller than the current value in p

Use Atomic Primitives

- **Fetch-and-add (FAA): integer FAA(integer* p, integer x): add integer p's value by x, and return the old value**
  - Multiple threads want to add a value to a shared variable
  - Multiple threads want to get a global sequentialized order



sum = 5

P1: add(3)

```
void Add(x) {
  temp = sum;
  sum = temp + x;
}
```

P2: add(4)

```
void Add(x) {
  temp = sum;
  sum = temp + x;
}
```

sum = 8 (but should be 12)

```
Shared variable sum
void Add(x) {
  sum = sum + x;
}
```

✗

```
Shared variable sum
void Add(x) {
  FAA(&sum, x);
}
```

```
Shared variable count
int get_id {
  return FAA(&count, 1);
}
```

Use Atomic Primitives

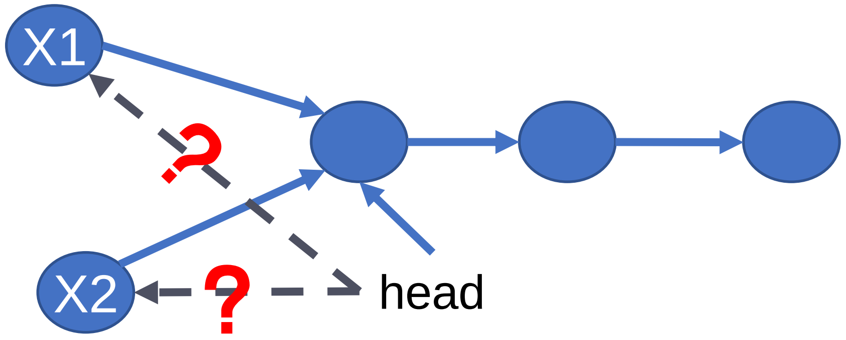
```
struct node {
  value_type value;
  node* next; };
shared variable node* head;
```

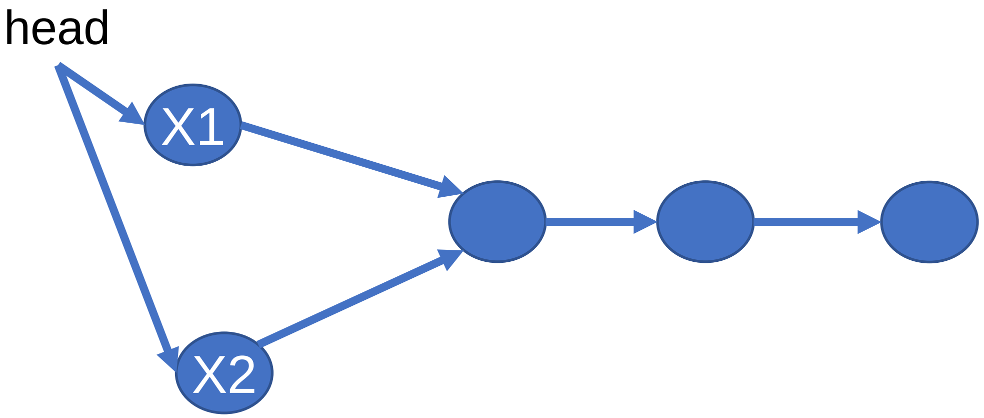
- **Compare-and-swap:**
  - Multiple threads wants to add to the head of a linked-list

```
void insert(node* x) {
   node* old_head = head;
   x->next = old_head;
   while (!CAS(&head, old_head, x)) {
     node* old_head = head;
     x->next = old_head;   }
}
```

```
void insert(node* x) {
   x->next = head;
   head = x;
}
```

❌





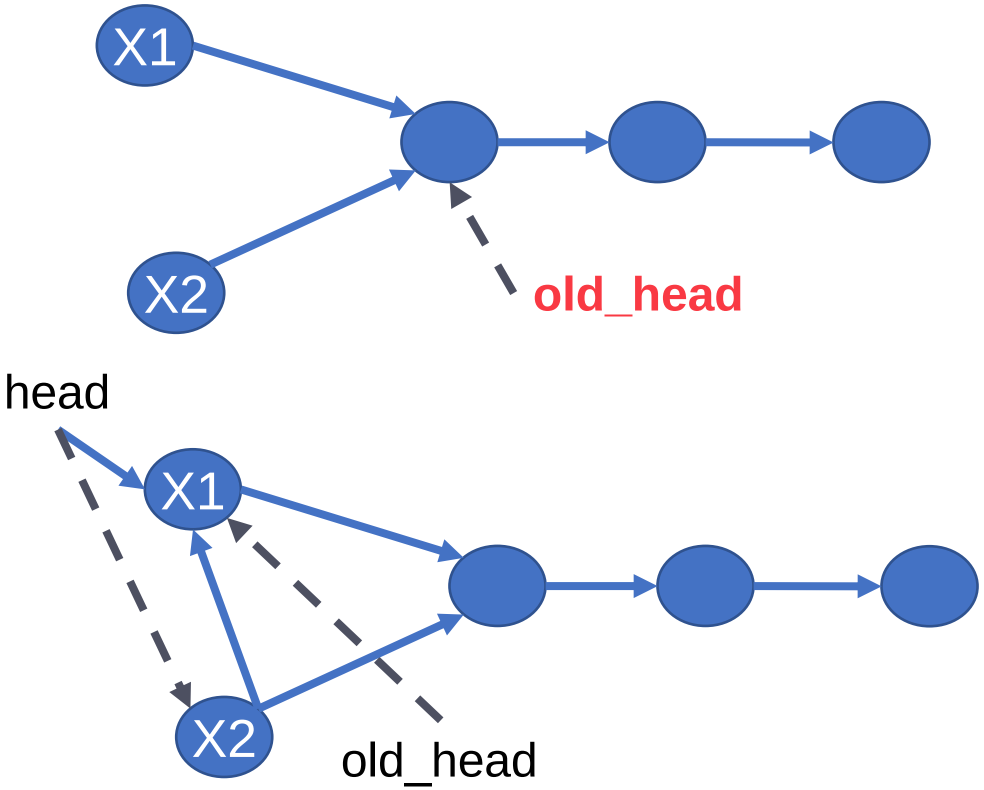
Use Atomic Primitives

```
struct node {
  value_type value;
  node* next; };
shared variable node* head;
```

- **Compare-and-swap:**
  - Multiple threads wants to add to the head of a linked-list

```
void insert(node* x) {
   node* old_head = head;
   x->next = old_head;
   while (!CAS(&head, old_head, x)) {
     node* old_head = head;
     x->next = old_head;   }
}
```

```
void insert(node* x) {
   x->next = head;
   head = x;
}
```



Concurrency – rule of thumb

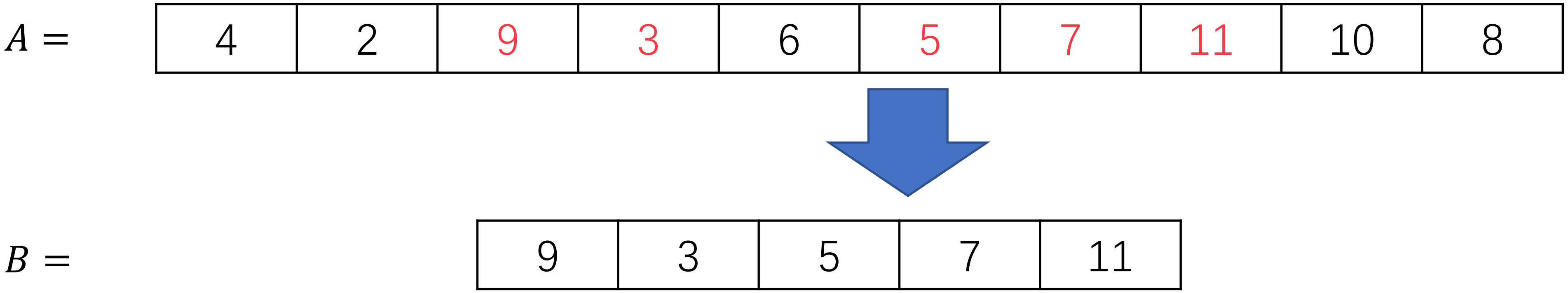
- Do not use concurrency, algorithmically
- If you have to (with the guarantee of correctness)
    - Do not use concurrent writes
    - If you have to (with the guarantee of correctness)
        - Do not use locks, use atomic primitives (still, with the guarantee of correctness)

Filtering/packing

Parallel filtering / packing

- Given an array A of elements and a predicate function f, output an array B with elements in A that satisfy f

$$f(x) = \begin{cases} true & if\ x\ is\ odd \\ false & if\ x\ is\ even \end{cases}$$



$A =$

4	2	9	3	6	5	7	11	10	8

$B =$

9	3	5	7	11

Parallel filtering / packing

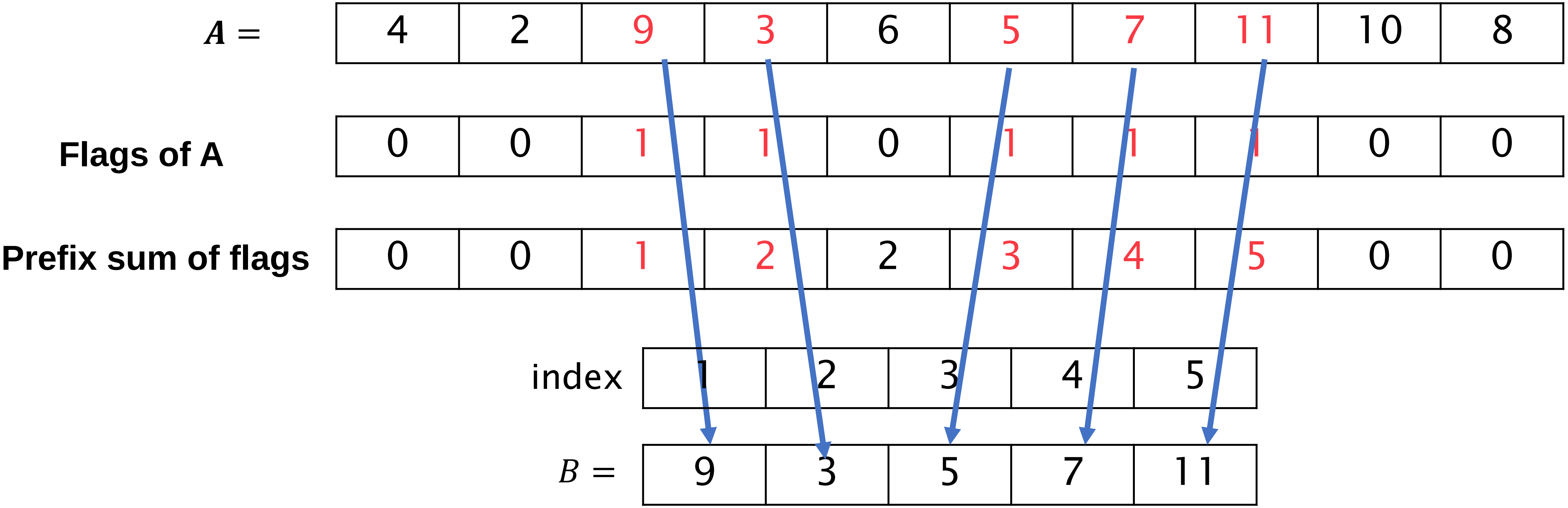
- **How can we know the length of B in parallel?**
  - Count the number of red elements – parallel reduce
  - $O(n)$ work and $O(\log n)$ depth

$A =$	4	2	9	3	6	5	7	11	10	8
	0	0	1	1	0	1	1	1	0	0

Parallel filtering / packing

- **How can we know where should 9 go?**
  - 9 is the first red element, 3 is the second, …

```
Filter(A, n, B, f) {
  new array flag[n], ps[n];
  para_for (i = 1 to n) {
    flag[i] = f(A[i]); }
  ps = scan(flag, n);
  parallel_for(i=1 to n) {
    if (ps[i]!=ps[i-1])
      B[ps[i]] = A[i];
  } }
```

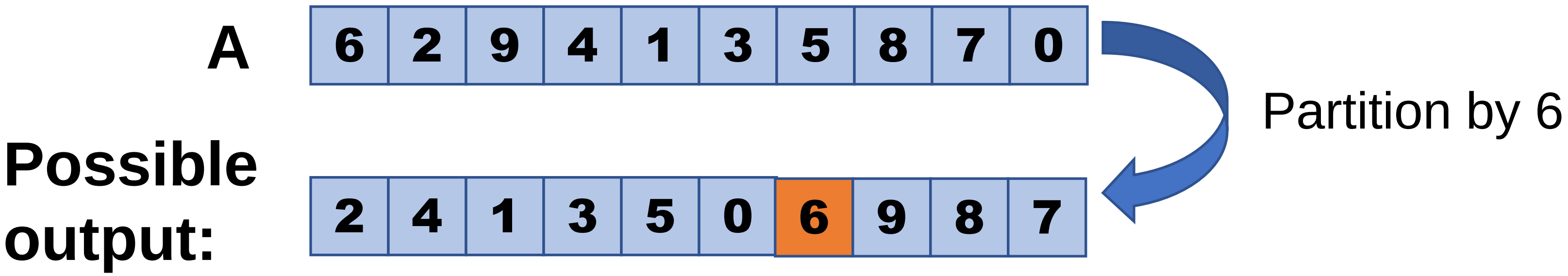


$A =$	4	2	9	3	6	5	7	11	10	8
Flags of A	0	0	1	1	0	1	1	1	0	0
Prefix sum of flags	0	0	1	2	2	3	4	5	0	0

index	1	2	3	4	5
$B =$	9	3	5	7	11

Application of filter: partition in quicksort

- For an array A, move elements in A smaller than k to the left and those larger than k to the right

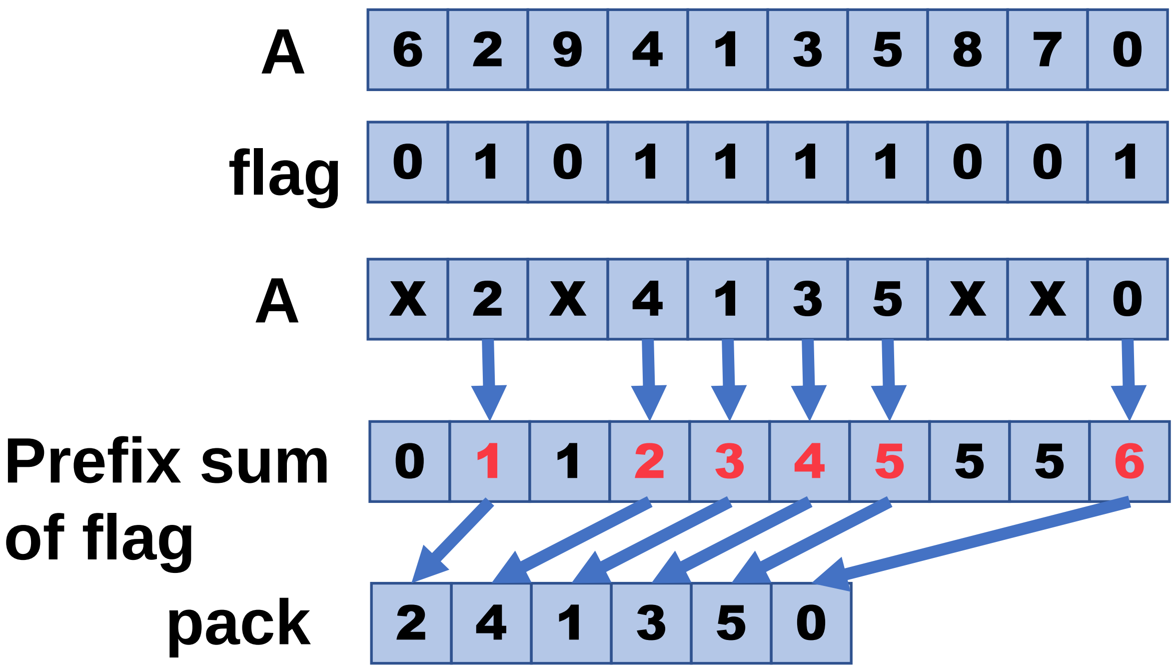


- The dividing criteria generally can be any predictor

Using filter for partition

(Looking at the left part as an example)

using 6 as a pivot



A	6	2	9	4	1	3	5	8	7	0
flag	0	1	0	1	1	1	1	0	0	1
A	X	2	X	4	1	3	5	X	X	0
Prefix sum of flag	0	1	1	2	3	4	5	5	5	6
pack	2	4	1	3	5	0				

```
Partition(A, n, k, B) {
  new array flag[n], ps[n];
  parallel_for (i = 1 to n) {
    flag[i] = (A[i]<k); }
  ps = scan(flag, n);
  parallel_for(i=1 to n) {
    if (ps[i]!=ps[i-1])
      B[ps[i]] = A[i];
  } }
```

Can we avoid using too much extra space?

Implementation trick: delayed sequence

Delayed sequence

- **A sequence is a function, so it does not need to be stored**
  - It maps an index (subscript) to a value
  - Save some space!

Delayed sequence

- A sequence is a function, so it does not need to be stored
  - Save some space

```
int reduce(int* A, int n) {
    if (n == 1) return A[0];
    int L, R;
    L                                   ;

    ci
    r

int main() {
  cin >> n;
  parallel_for (int i = 0; i < n; i++)
    A[i] = i;
  cout << reduce(A, n) << endl;
```

Running time:
about 0.19s for n=10^9,
with coarsening

```
inline int get_val(int i) {return i;}
int reduce(int start, int n, function f)
{
    if (r
    int L
    L = c                                  , f);
        R                                  f);
    cilk_sync;
    return L+R; }

int main() {
  cin >> n;
  cout << reduce(0, n, get_val) << endl;
```

Running time:
about 0.16s for n=10^9,
with coarsening

Partition without the flag array

Old version

```
Partition(A, n, k, B) {
  new array flag[n], ps[n];
  parallel_for (i = 1 to n) {
    flag[i] = (A[i]<k); }
  ps = scan(flag, n);
  parallel_for(i=1 to n) {
    if (ps[i]!=ps[i-1])
      B[ps[i]] = A[i];
  }
```

New version

```
Partition(A, n, k, B) {
  new array ps[n];
  ps = scan(0, n,
      [&](int i) {return (A[i]<k);});
  parallel_for(i=1 to n) {
    if (ps[i]!=ps[i-1])
      B[ps[i]] = A[i];
  }
```

Equivalent to having an array:
`flag[i] = (A[i]<k);`
But without explicitly storing it

(We can also get rid of the ps[] array, but it makes the program a bit more complicated)

Implementation trick: nested/granular/blocked parallel for-loops

Nested parallel for-loops

- **Usually only need to parallelize the outmost one**
- **Make each parallel task large enough**

Parallel `i` loop

```
cilk_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Running time: 3.18s

Parallel `j` loop

```
for (int i = 0; i < n; ++i)
  for (int k = 0
    cilk_for (in
      C[i][j] +=
```

Running time: 531.71s

Parallel `i` and `j`

```
cilk_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    cilk_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Running time: 10.64s

Rule of Thumb

Parallelize outer loops rather than inner loops

Granular-for

- If some condition holds, run the for loop in parallel
  - Usually determining if the size of the for-loop is larger than a threshold
- Otherwise, run it sequentially

- E.g., for a for-loop with size smaller than 2000, run it sequentially, otherwise run it in parallel

```
#define granular_for(_i, _start, _end, _cond, _body) {   \
  if (_cond) {                                            \
    {parallel_for(size_t _i=_start; _i < _end; _i++) {    \
        _body                                             \
    }}                                                    \
  } else {                                                \
  {for (size_t _i=_start; _i < _end; _i++) {              \
      _body                                               \
    }}                                                    \
  }                                                       \
}
```

```
granular_for (i, 0, n, (n>2000), {A[i]=i});
```

Blocked-for

- For a for-loop, combine each _bsize of them as one task, and run them in parallel

- Also to avoid the case when each task is too small
    - Your scheduler can help do this in some sense, but it doesn't know much about your loop body
- E.g., put each 500 loop-body into one task

```
#define nblocks(_n,_bsize) (1 + ((_n)-1)/(_bsize))

#define blocked_for(_i, _s, _e, _bsize, _body)  {          \
    intT _ss = _s;                                          \
    intT _ee = _e;                                          \
    intT _n = _ee-_ss;                                      \
    intT _l = nblocks(_n,_bsize);                           \
    parallel_for (intT _i = 0; _i < _l; _i++) {             \
      intT _s = _ss + _i * (_bsize);                        \
      intT _e = min(_s + (_bsize), _ee);                    \
      for (intT _j = s; _j < e; j++)  {                     \
          _body                                             \
      }                                                     \
    }                                                       \
 }
```

# of blocks

From start of the block to the end of the block

```
block_for (i, 0, n, 500, {A[i]=i});
```

Implementation trick: dos and don'ts

Allocate large memory

- **Don’t (frequently, dynamically) allocate memory in parallel**
  - This has to go through the OS
  - New space cannot be allocated in parallel with other threads running

- **Allocate enough memory in advance**
  - When needed, distribute the memory to the threads

- **This means – using std::vector *can* slow down your parallel code (if you are not careful enough)**
  - When resizing it needs to allocate new space and delete the old one
  - If you want to use std::vector, reserve enough space before starting parallel running

Generating random numbers

- **Do not use the default random number generator**
  - Use system time
  - Involve synchronization – slows down parallel performance

- **Use a hash function instead**
  - Just write some random things as your hash function – it's a pseudo-random number generator anyway

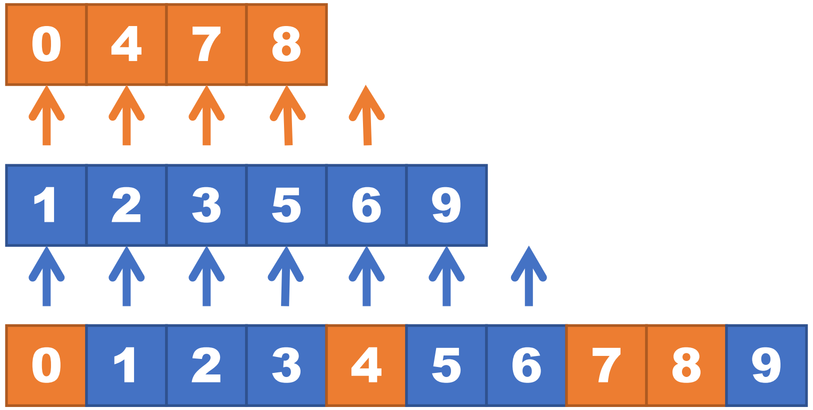
```
// a 32-bit hash function
 inline uint32_t random_hash(uint32_t a) {
   a = (a+0x7ed55d16) + (a<<12);
   a = (a^0xc761c23c) ^ (a>>19);
   a = (a+0x165667b1) + (a<<5);
   a = (a+0xd3a2646c) ^ (a<<9);
   a = (a+0xfd7046c5) + (a<<3);
   a = (a^0xb55a4f09) ^ (a>>16);
   return a;
 }

parallel_for (i = 0 to n) random[i]=random_hash(i);
```

Generate n random integers in parallel

Parallel merging

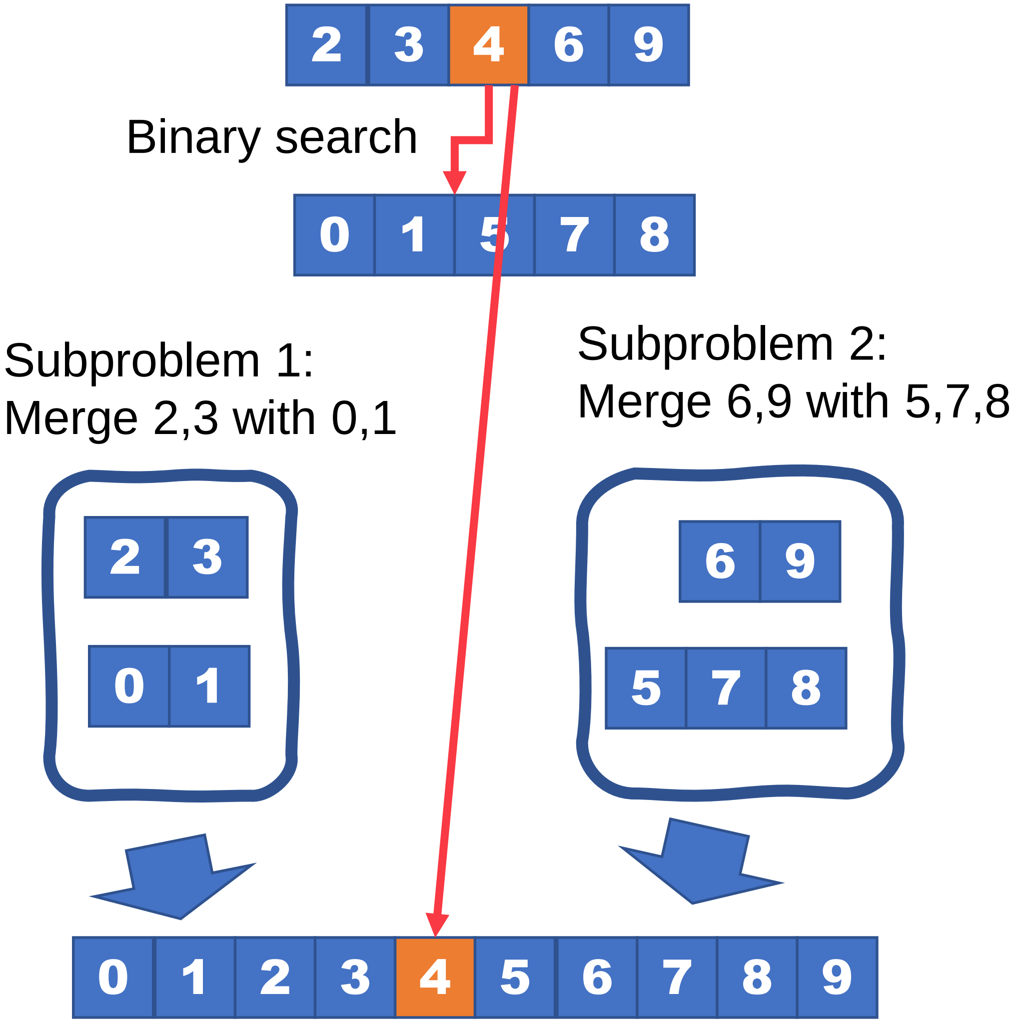
Parallel merging

- Given two sorted arrays, merge them into one sorted array
- Sequentially, use two moving pointers



A parallel merge algorithm

- **Find the median m of one array**
- **Binary search it in the other array**
- **Put m in the correct slot**
- **Recursively, in parallel do:**
    - Merge the left two sub-arrays into the left half of the output
    - Merge the right ones into the right half of the output

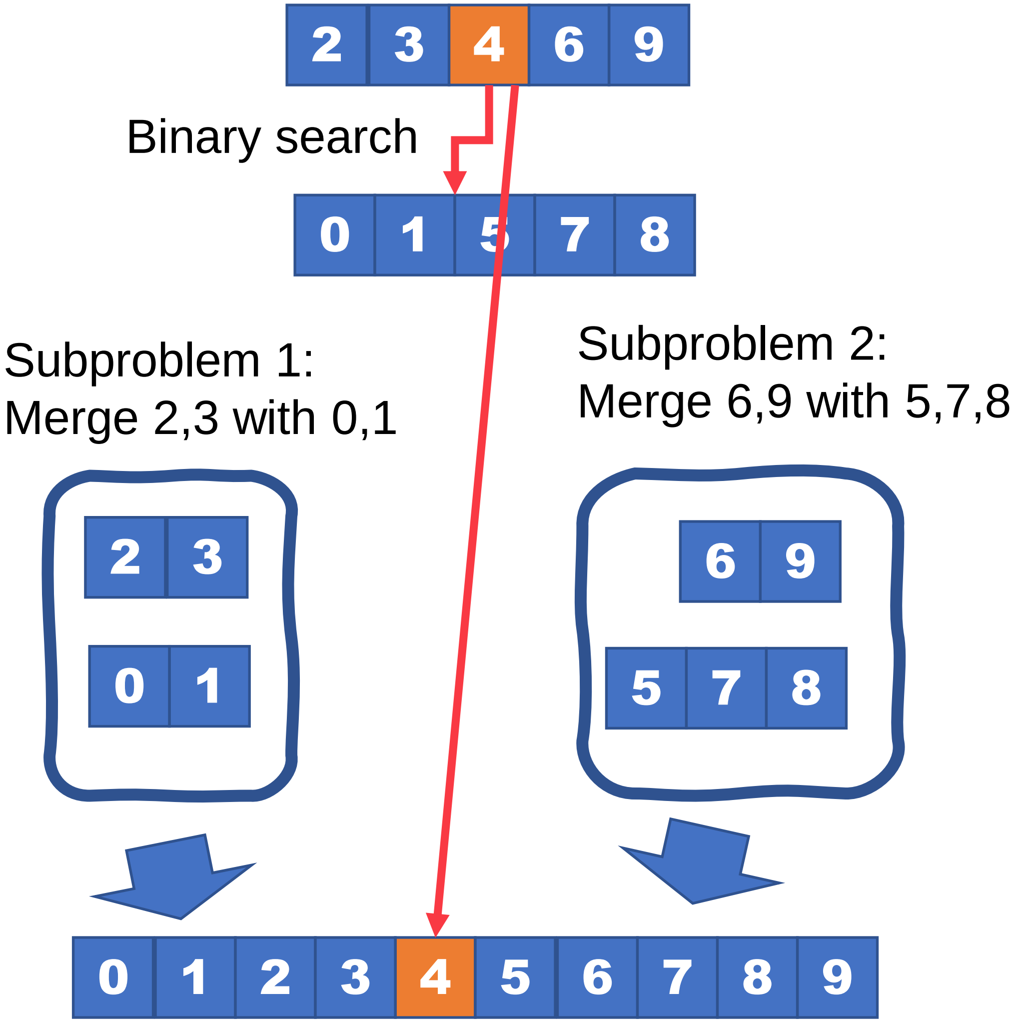


Subproblem 1:
Merge 2,3 with 0,1

Subproblem 2:
Merge 6,9 with 5,7,8

A parallel merge algorithm

```
//merge array A of length n1 and array B of length
n2 into array C.
Merge(A’, n1, B’, n2, C) {
  if (A’ is empty or B’ is empty) base_case;
  m = n1/2;
  m2 = binary_search(B’, A’[m]);
  C[m+m2+1] = A’[m];
  in parallel:
    merge(A’, m, B’, m2, C);
    merge(A’+m+1, n1-m-1, B’+m2+1, n2-m2-1, C+m+m2);
  return C;
}
```



Binary search

Subproblem 1:
Merge 2,3 with 0,1

Subproblem 2:
Merge 6,9 with 5,7,8