

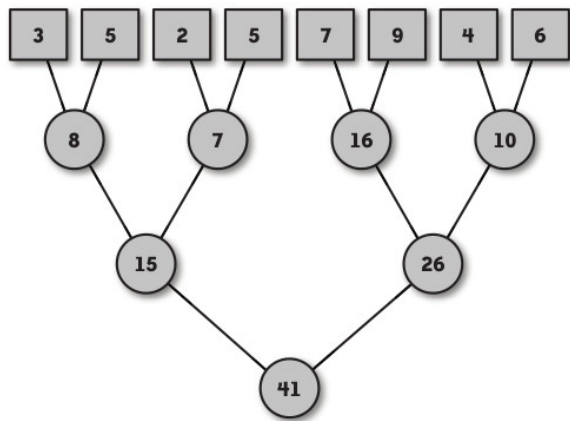
## 0.1 Parallel Algorithms

### 0.1.1 Parallel Sum

The problem is to compute the sum of all values within a given array.

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

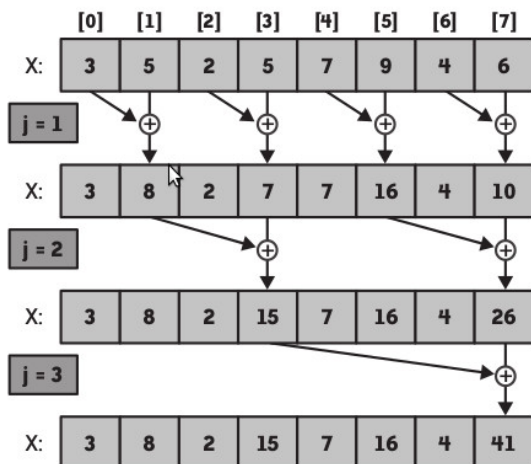
The operation that is performed to accumulate elements of the array is not limited to addition. It may be any associative and commutative operation: multiplication, maximum, logical and, etc. Here is an idea:



It may be translated into the following parallel code

```
for j = 1 .. log_2(n) {
    for k = 0 .. n step 2^j in parallel {
        X[k] = X[k - 2^(j-1)] + X[k]
    }
}
```

Here is an example:



Problems: we will have to implement barriers to make sure inner (parallel) iteration is fully completed before going into the next outer iteration. Also the number of parallel adds in the inner loop may need to be controlled.

Note:

- the above algorithm may be useful in CUDA and some other libraries.
- algorithm uses the original array to store intermediate results. If this is not desirable extra memory of size  $n/2$  is required.

### 0.1.2 Parallel Sum - practical

Many parallel algorithm are based on appropriate data decomposition. For parallel sum the idea is quite simple: preprocess array - determine chunks that may be added in parallel (the number of chunks is the desired number of threads). That add the partial sums (serial code will probably be enough for the last part).

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4

int N = 1000000; // number of elements in array X
int *X;
int chunk_sums[NUM_THREADS]; // global storage for partial results

void *Summation (void *pArg)
{
    int tid = *((int *) pArg);
    int local_sum = 0;
    int start, end;

    start = (N/NUM_THREADS) * tid;
    end = (N/NUM_THREADS) * (tid+1);

    if (tid == (NUM_THREADS-1)) end = N; // the last may do more work

    for (int i = start; i < end; ++i) {
        local_sum += X[i];
    }
    chunk_sums[tid] = local_sum;
}

int main(int argc, char* argv[])
{
    int sum = 0;
    pthread_t thread_ids[NUM_THREADS];

    // initialize array
    X = new int[N];
```

```

for ( int i = 0; i < N; ++i ) {
    X[i] = 1;
}

int* args[NUM_THREADS];
for ( int j = 0; j < NUM_THREADS; ++j ) {
    args[j] = new int(j);
}

for ( int j = 0; j < NUM_THREADS; ++j ) {
    pthread_create(&thread_ids[j], NULL, Summation, (void *)args[j]);
}

for ( int j = 0; j < NUM_THREADS; ++j ) {
    pthread_join(thread_ids[j], NULL);
    sum += chunk_sums[j];
}

for ( int j = 0; j < NUM_THREADS; ++j ) {
    delete args[j];
}

delete [] X;

printf("The sum of array elements is %d\n", sum);
return 0;
}

```

### 0.1.3 Prefix Scan

The problem is to calculate an array of partial sums of a given array:

Original vector	3	5	2	5	7	9	4	6
Inclusive prefix scan	3	8	10	15	22	31	35	41
Exclusive prefix scan	0	3	8	10	15	22	31	35

Corresponding serial code:

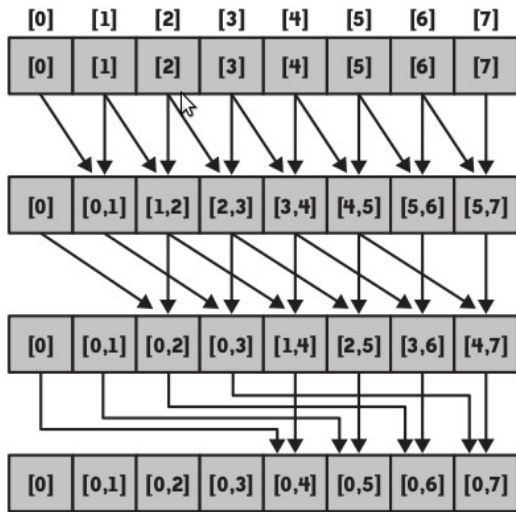
```

prefixScan[0] = a[0];
for (int i = 1; i < N; ++i) {
    prefixScan[i] = prefixScan[i-1] + a[i];
}

```

Note how each iteration depends on the result of the previous! In the current form the code is not obviously parallelizable. Moreover - loop unrolling is also difficult.

Here is a very nice idea  $[i, j]$  denotes a partial sum for range indices  $i \dots j$  with endpoints included:



Corresponding code

```
for j = 1 to log_2(n) {
  for ( k = 2^(j-1) to n-1 parallel {
    X[k] = X[k - 2^(j-1)] + X[k]
  }
}
```

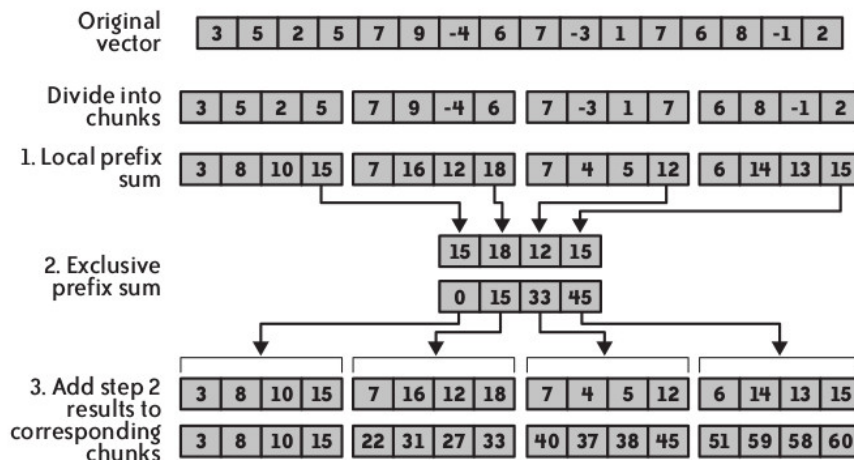
Note: prefix scan may be used with other operation, such operation has to be associative, but does not have to be commutative, since the order of the **scan** is fix by the problem statement. As a result – we have to respect it, that's why we use

$X[k] = X[k - 2^{(j-1)}] + X[k]$

in the code above.

Problems: we will have to implement barriers to make sure inner (parallel) iteration is fully completed before going into the next outer iteration. Also the number of parallel adds in the inner loop may need to be controlled.

## 0.1.4 Prefix Scan - practical



### 0.1.5 Selection problem

This is also known as *median* and  $k^{th}$  *largest*.

The problem is to find  $k^{th}$  largest in a unordered collection (array). Obvious solution to sort and use index operation will be  $O(n \log n)$  algorithm.

Here is a linear time algorithm:

1. If the size of the data set to be used is less than some constant size,  $Q$ , sort the data and return the  $k^{th}$  element; otherwise, subdivide the data set into chunks of size  $Q$  and whatever is left over.
2. Sort each chunk and find the median of each.
3. Recursively call the selection routine to find the median of the medians found in the previous step.
4. Partition the data set into three subsequences: those whose elements are less than the median of medians, those that are equal to the median of medians, and those that are greater than the median of medians.
5. Determine which subsequence contains the  $k^{th}$  element, from the sizes of the three subsequences, and recursively call the selection routine on that subsequence. If the  $k^{th}$  element is not in the subsequence of smaller or larger items, it must be in the subsequence equal to the median of medians, so just return the median of medians value.

Notes:

- $Q=5$  is reasonable
- in step 4) just count the number of element in the 3 subsequences. Do not allocate, create a second array of the length of original and mark element 0,1,2 (or whatever): '-1' - less, '0' - equal, '1' - greater
- in step 5) when you know which subsequences is required, allocated it. Use marks to decide which element to copy.

```
int SequentialSelect(int *S, int size, int k)
{
    if (size <= Q) return BaseCaseKth(S, size, k);
    int cNum = size/Q + 1; // number of chunks
    int *Medians = new int[cNum];

    for all chunks {
        Medians[j] = SortSelect5( S, offset, 3 ); // find medians (3rd) of subsequences
    }

    int M = SequentialSelect(Medians, cNum, (cNum+1)/2);

    int leg[3] = {0,0,0}; // counts for Less, Equal, Greater
    int *markS = new int[size]; // marks
    markS[0] = -1, markS[1] = 0, markS[2] = 1
}
```

```

CountAndMark(S, markS, size, M, leg);

if (leg[0] >= k) { // median is among "Less"
    int *sPack = new int[leg[0]];
    ArrayPack(S, sPack, size, markS, 0);
    return SequentialSelect(sPack, leg[0], k);
}
else if ((leg[0] + leg[1]) >= k) { // median is among "Equal", so problem solved
    return M;
}
else { // median is among "Greater"
    int *sPack = new int[leg[2]];
    ArrayPack(S, sPack, size, markS, 2);
    return SequentialSelect(sPack, leg[2], k-(leg[0]+leg[1]));
}
}

void CountAndMark(int S[], int Marks[], int num, int median, int leg[3])
{
    for (int i = 0; i < num; i++) {
        if (S[i] < median) {Marks[i] = 0; leg[0]++;} //less than
        else if (S[i] > median) {Marks[i] = 2; leg[2]++;} // greater than
        else {Marks[i] = 1; leg[1]++;} // equal to
    }
}

void ArrayPack(int S[], int sPack[], int num, int Marks[], int scanSym)
{
    int j = 0;
    for (int i = 0; i < num; i++)
        if (Marks[i] == scanSym) sPack[j++] = S[i];
}

```

Discussion:

- reminder in cs330 we used Partition function to find  $k^{th}$  element.

```

unsigned partition(int* a, unsigned begin, unsigned end) {
    unsigned i = begin, last = end-1;
    int pivot = a[last];
    for (unsigned j=begin; j<last; ++j) {
        if ( a[j]<=pivot ) {
            std::swap( a[j], a[i] );
            ++i;
        }
    }
    std::swap( a[i], a[last] );
}

```

```

return i;
}

```

- Worst case run-time of pivot-based  $k^{th}$  element was  $O(n^2)$
- Average case run-time of pivot-based  $k^{th}$  element is  $O(n)$
- Worst case run-time of median-of-medians-based  $k^{th}$  element is  $O(n)$ . Explanation: the worst case of cs330 solution is when all pivots split corresponding arrays into very uneven chunks - worst case 0 and  $n - 1$  sizes. In proposed algorithm pivot is chosen as a median of medians, therefore it guarantees that at least  $n/5$  elements will be smaller than the median of medians (i.e. belong to the "small" chunk).

### 0.1.6 Parallel Selection problem

There are 3 places that may be parallelized

- sorting 5-element chunks
- partitioning (as it was noted just count and mark)
- allocating array for the next round - packing

The first 2 tasks may be accomplished using "bag of jobs" paradigm:

```

Collection bag[n]; // bag of jobs
Semaphore sem(k); // k - number of simultaneous threads

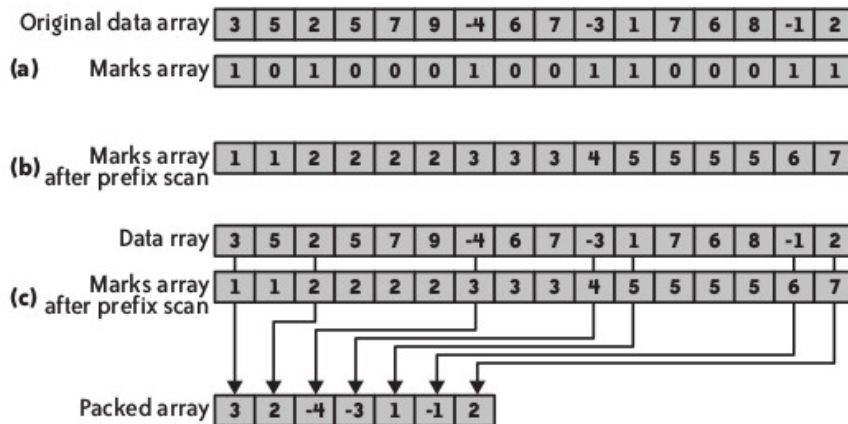
for ( i=0; i<n; ++i ) {
    sem.wait();
    thread( bag[i] );
    sem.signal();
}

```

The last task is more interesting. You have an array of marks (-1,0,1) and some elements (either marked as -1, or 1) should be copied into another array.

At first it looks that the operation is sequential.

But using a nice trick - prefix scan, we can parallelize the operation:



Since in selection the marks are -1,0,1, you may need to modify prefix scan to treat -1 as 0 to select 1's only (to select -1's only invert all values).

### 0.1.7 Quicksort

Programming assignment.

### 0.1.8 DFS

```
int *visited; // notes when a node has been visited
int **adj;    // adj[] [] is adjacency matrix
int V;        // number of nodes in graph
```

```
void visit(int k)
{
    int i;
    visited[k] = 1;
    /*
        Do something to VISIT node k
    */
    for (i = 0; i < V; i++)
    {
        if (adj[k][i])
            if (!visited[i]) visit(i);
    }
}
```

```
void DFSearch()
{
    int k;
    for (k = 0; k < V; k++) visited[k] = 0;
    for (k = 0; k < V; k++)
        if (!visited[k]) visit(k);
}
```

Iterative version:

```
int *visited;
int **adj;
int V;
stack S; //stack of nodes (indices)

void DFSearch()
{
    int i, k;
    for (k = 0; k < V; k++) visited[k] = 0;
    for (k = V-1; k >= 0; k--) {
```



```

    push(S, k);
}

while (size(S) > 0) {
    k = pop(S);
    if (!visited[k]) {
        visited[k] = 1;
        /*
            Do something to VISIT node k
        */
        for (i = V-1; i >= 0; i--)
            if (adj[k][i]) push(S, i);
    }
} // end while
}

```

This code can be parallelize similar to Quicksort. The only difference is the present of array `visited` which is used a lot by `DFSearch`. Which means that if you use a single lock to guard access to it, it will become the bottleneck. Most threads will be siting idle waiting to get access to the array.

Solution: once possible solution is to use *readers-writers* design. Note that each thread will be both reader and writer depending on the position in code. This solution will probably still be slow – reader’s section is very small and the chance of multiple readers is low, which means that the behavior will be very similar to a single lock solution.

Solution: lock each element of `visited` with its own mutex. Fastest, but expensive. We may be dealing with hundreds of thousands of nodes.

Solutions in between:

- 2 locks – one for even, one for odd elements
- number of locks is equal to the number of threads – index  $i$  is locked by `mutex[i mod num_threads]`
- number of locks is twice the number of threads – index  $i$  is locked by `mutex[i mod (2 * num_threads)]`

Some code-related efficiency considerations:

This code’s critical section is too big:

```

j = k % NUM_LOCKS;
pthread_mutex_lock( &Vmutex[j] );
lVisited = visited[k];
if (!lVisited) {
    visited[k] = 1;
    /*
        Body of if statement
    */
}
pthread_mutex_unlock ( &Vmutex[j] );

```

This code has an error – another thread may read `visited[k]` while we are between critical sections:

```
j = k % NUM_LOCKS;
pthread_mutex_lock( &Vmutex[j] );
lVisited = visited[k];
pthread_mutex_unlock ( &Vmutex[j] );
if (!lVisited) {
    pthread_mutex_lock ( &Vmutex[j] );
    visited[k] = 1;
    pthread_mutex_unlock ( &Vmutex[j] );
    /*
        Body of if statement
    */
}
```

Correct, but difficult to read (and therefore maintain):

```
j = k % NUM_LOCKS;
pthread_mutex_lock( &Vmutex[j] );
lVisited = visited[k];
if (!lVisited) {
    visited[k] = 1;
    pthread_mutex_unlock ( &Vmutex[j] );
    /*
        Body of if statement
    */
} else {
    pthread_mutex_unlock ( &Vmutex[j] );
}
```

Using extra local variable:

```
j = k % NUM_LOCKS;
pthread_mutex_lock(&Vmutex[j]);
if (!visited[k]) {
    iWillVisit = 1;
    visited[k] = 1;
}
pthread_mutex_unlock(&Vmutex[j]);

if (iWillVisit) {
    /*
        Body of if statement
    */
    iWillVisit = 0; // prepare for the next iteration
}
```

Another use of local variable (I noticed that while implementing Quicksort):

```
int global_counter = 0;

void* thread( void* ) {
    /* stuff */
    pthread_mutex_lock( &global_counter_mutex );
    global_counter += (b-a);
    pthread_mutex_unlock( &global_counter_mutex );
    /* stuff */
    pthread_mutex_lock( &global_counter_mutex );
    --global_counter;
    pthread_mutex_unlock( &global_counter_mutex );
    /* stuff */
    pthread_mutex_lock( &global_counter_mutex );
    global_counter -= 8;
    pthread_mutex_unlock( &global_counter_mutex );
    /* stuff */
}
```

Can be rewritten (at least in some cases):

```
int global_counter = 0;

void* thread( void* ) {
    int local_counter_update = 0;
    /* stuff */
    local_counter_update += (b-a);
    /* stuff */
    --local_counter_update;
    /* stuff */
    local_counter_update -= 8;
    /* stuff */

    pthread_mutex_lock( &global_counter_mutex );
    global_counter += local_counter_update;
    pthread_mutex_unlock( &global_counter_mutex );
}
```