Part B - Languages

OpenMP - Prefix Scan

Introduce the prefix scan pattern
Describe the different algorithms for implementing a prefix scan
Present the code for a simple filter application

Introduction | Parallel Algorithms | Filter Application | Exercises

Searching, lexical analysis, sorting, string comparison and stream compaction applications implement patterns that entail loop-carried dependencies. In these patterns, a particular iteration depends on the result of previous iterations. Parallel algorithms that implement these pattern cannot avoid these loop-carried dependencies. The scan pattern is a pattern that addresses this issue. This pattern is similar to the reduction pattern, but stores the intermediate results.

This chapter describes several solutions to the prefix scan problem, starting with the serial version and followed by parallel algorithms. The code is structured in preparation for an OpenMP solution with the final step left as an exercise for the reader.

INTRODUCTION

The scan pattern has two distinct solutions that depends on the initial condition of the problem at hand:

- inclusive scan - operates on all previous elements including the initial element
- exclusive scan - operates on all previous elements excluding the initial element

The source code for an inclusive prefix-sum scan is listed on the left. The code for an exclusive prefix-sum scan is listed on the right.

```
y[0] = x[0];
for (int i = 1; i < n; i++) {
    y[i] = x[i] + y[i - 1];
```

```
y[0] = 0;
for (int i = 1; i < n; i++)
    y[i] = x[i - 1] + y[i - 1];
```

Inclusive Scan Pattern

An inclusive scan takes an array of elements (listed on the left side) and returns an array of cumulative values (listed on the right side):

$$[a_0, \ a_1, \ a_2, \ ..., \ a_{n-1}]$$

$$[a_0, \ a_0 \ op \ a_1, \ a_0 \ op \ a_1 \ op \ a_2, \ ..., \ a_0 \ op \ a_1 \ op \ ... \ op \ a_{n-1}]$$

where **op** denotes the operation on a_i and a_{i+1}.

For example, an inclusive prefix-sum scan on the set of elements on the left yields the set of elements on the right:

```
[3,
 1,
 7,
 0,   ==> inclusive scan ==>
 4,
 1,
```

```
[3,
 1 + 3  = 4,
 7 + 4  = 11,
 0 + 11 = 11,
 4 + 11 = 15,
 1 + 15 = 16,
```

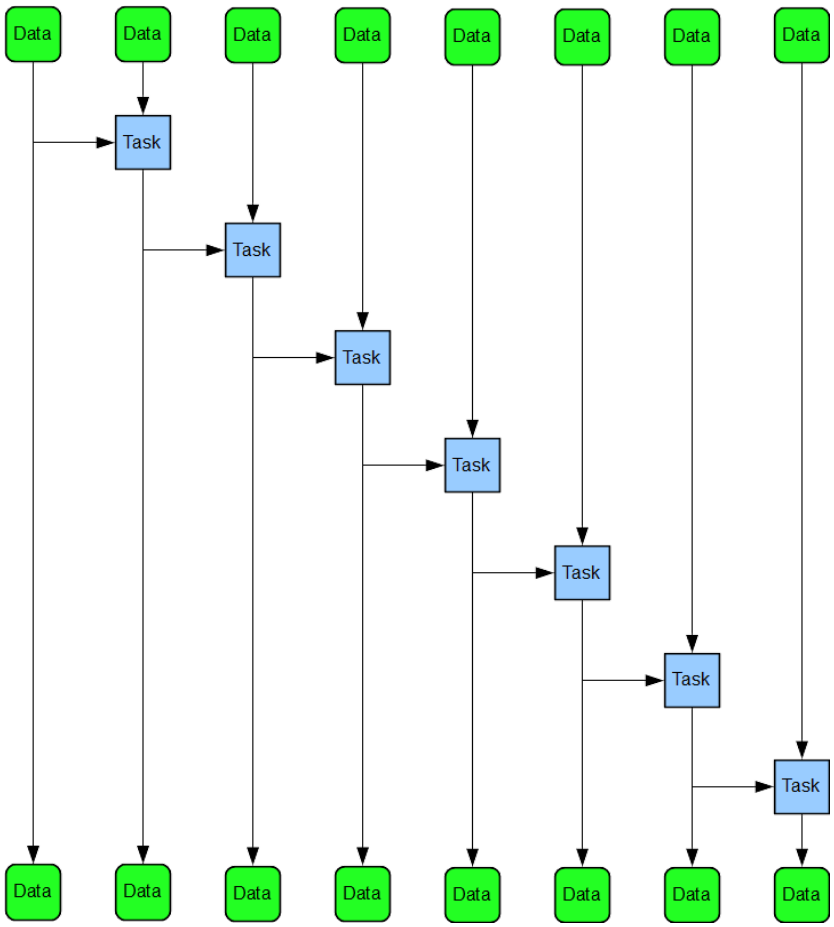
```
 6,                                  6 + 16 = 22,
 3]                                  3 + 22 = 25]
```

The second operand in the right side panel (highlighted) is the value of the operation for the immediately preceding element. That is,

```
[3, 1, 7, 0, 4, 1, 6, 3] => inclusive scan =>   [3, 4, 11, 11, 15, 16, 22, 25]
```

Serial Version

A serial version of an inclusive scan is shown below:



A Serial Version of the Inclusive Scan Pattern

The following templated function performs a serial inclusive scan using the operation specified by **`combine`**:

```
template <typename T, typename C>
void incl_scan(
    const T* in,                // source data
    T* out,                     // output data
    int size,                   // size of data sets
    C combine,                  // combine operation
    T initial                   // initial value
    ) {

    for (int i = 0; i < size; i++) {
        initial = combine(initial, in[i]);
        out[i] = initial;
    }
}
```

The work of this algorithm is O(n) and the span is O(n).

Exclusive Scan Pattern

An exclusive scan takes the array of elements listed on the left side and returns the array of elements listed on the right side:

```
[a0,
 a1,
 a2,    ==> exclusive scan ==>
 a3,
 ...,
 an-1]
```

```
[I,
 a0,
 a0 op a1,
 a0 op a1 op a2,
 ...,
 a0 op a1 op ... op an-2]
```

where **I** is the identity element for the operation.

For example, an exclusive prefix-sum operation on the set of elements on the left yields the set of elements on the right:

```
[3,
 1,
 7,
 0,    ==> exclusive scan ==>
 4,
 1,
 6,
 3]
```

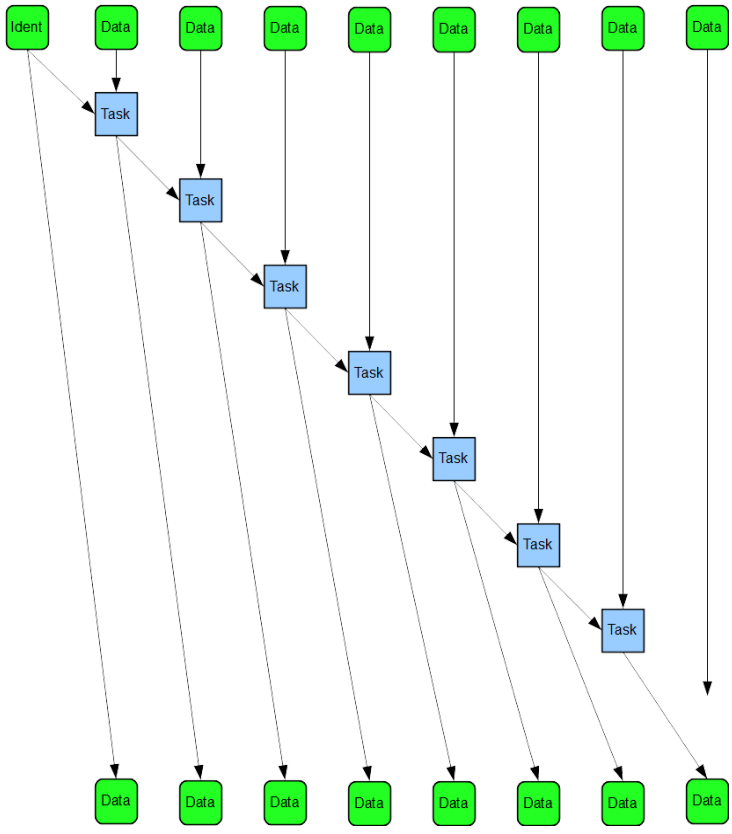
```
[0,
 3 + 0  = 3,
 1 + 3  = 4,
 7 + 4  = 11,
 0 + 11 = 11,
 4 + 11 = 15,
 1 + 15 = 16,
 6 + 16 = 22]
```

The second operand in the right side panel (highlighted) is the value of the operation for the immediately preceding element. That is,

```
[3, 1, 7, 0, 4, 1, 6, 3] => exclusive scan =>   [0, 3, 4, 11, 11, 15, 16, 22]
```

Serial Version

A serial version of an exclusive scan is shown below:



A Serial Exclusive Scan Pattern

The following templated function performs a serial exclusive scan using the operation specified by **combine**:

```
template <typename T, typename C>
void excl_scan(
    const T* in,                // source data
    T* out,                     // output data
    int size,                   // size of data sets
    C combine,                  // combine operation
    T initial                   // initial value
    ) {

    if (size > 0) {
        for (int i = 0; i < size - 1; i++) {
            out[i] = initial;
            initial = combine(initial, in[i]);
        }
        out[size - 1] = initial;
    }
}
```

The work of this algorithm is O(n) and the span is O(n).

Serial Version Example

The following serial code is a prototype for investigating different parallel solutions to the prefix scan problem. Parallel solutions for the inclusive and exclusive scan functions are listed in the following section. Each scan function uses the operation specified by **combine**:

```
// Prefix Scan Example
// Serial Version
// prefix_scan.serial.cpp
// after McCool etal. (2012)

#include <iostream>

template <typename T, typename C>
void incl_scan(
    const T* in,                // source data
    T* out,                     // output data
    int size,                   // size of data sets
    C combine,                  // combine operation
    T initial                   // initial value
    ) {

    for (int i = 0; i < size; i++) {
        initial = combine(initial, in[i]);
        out[i] = initial;
    }
}

template <typename T, typename C>
void excl_scan(
    const T* in,                // source data
    T* out,                     // output data
    int size,                   // size of data sets
    C combine,                  // combine operation
    T initial                   // initial value
    ) {

    if (size > 0) {
        for (int i = 0; i < size - 1; i++) {
            out[i] = initial;
            initial = combine(initial, in[i]);
        }
        out[size - 1] = initial;
    }
```

```
}

template <typename T, typename C, typename S>
void scan(
    const T* in,   // source data
    T* out,        // output data
    int size,      // size of source, output data sets
    C combine,     // combine expression
    S scan_fn,     // scan function (exclusive or inclusive)
    T initial      // initial value
    )
{
    scan_fn(in, out, size, combine, T(0));
}

int main() {
    const int N = 9;
    int in[N] = { 3, 1, 7, 0, 1, 4, 5, 9, 2 };
    int out[N];
    // combine operation
    auto combine = [](int a, int b) { return a + b; };

    scan(in, out, N, combine, incl_scan<int,
        decltype(combine)>, (int)0);

    for (int i = 0; i < N; i++) {
        std::cout << out[i] << ' ';                    3 4 11 11 12
        if (i % 5 == 4) std::cout << std::endl;        16 21 30 32
    }
    std::cout << std::endl;

    scan(in, out, N, combine, excl_scan<int,
        decltype(combine)>, (int)0);

    for (int i = 0; i < N; i++) {
        std::cout << out[i] << ' ';                    0 3 4 11 11
        if (i % 5 == 4) std::cout << std::endl;        12 16 21 30
    }
    std::cout << std::endl;
}
```

The **`decltype(combine)`** template argument passes the type of the **`combine`** operation to the **`scan`** template.

PARALLEL ALGORITHMS

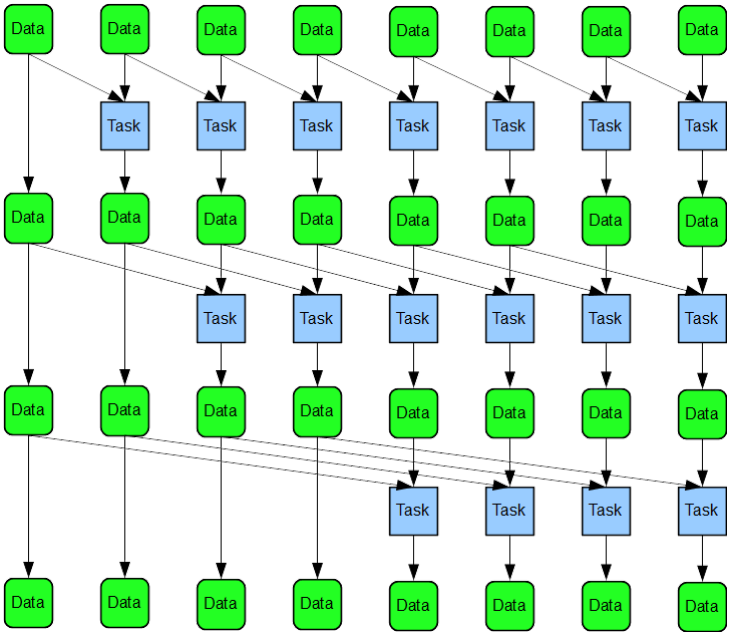
Assumptions

Parallel implementations of the scan pattern assume that

- the data elements can be combined in any order
- an identity element exists - guaranteeing meaning if the number of data elements is zero

A Naive Parallel Solution

Hillis and Steele (1986) published an early solution to the prefix-scan problem. The following pattern illustrates the operations in this algorithm.



Naive Solution to the Parallel Scan Problem (Hillis-Steele, 1986)

Data is updated row by row. Consider a row in this graph. For some columns in that row, the data element is used to augment a data element in another column and is also augmented by data from another column. To avoid pre-mature over-writing of data, we introduce double-buffering.

The following functions implement serial versions of this pattern.

```
template <typename T, typename C>
void incl_scan(
    const T* in, // source data
    T* out,      // output data
    int size,    // size of source, output data sets
    C combine,   // combine expression
    T initial    // initial value
    )
{
    T* buffer = new T[2 * size];
    int pout = 0, pin = 1;
    for (int i = 0; i < size; i++)
        buffer[pout * size + i] = in[i];
    for (int stride = 1; stride < size; stride *= 2) {
        pout = 1 - pout;
        pin  = 1 - pout;
        for (int i = 0; i < size; i++) {
            if (i >= stride)
                buffer[pout * size + i] = combine(buffer[pin * size + i],
                 buffer[pin * size + i - stride]);
            else
                buffer[pout * size + i] = buffer[pin * size + i];
        }
    }
    for (int i = 0; i < size; i++)
        out[i] = buffer[pout * size + i];
    delete[] buffer;
}

template <typename T, typename C>
void excl_scan(
    const T* in, // source data
    T* out,      // output data
    int size,    // size of source, output data sets
    C combine,   // combine expression
    T initial    // initial value
    )
{
```

```
    if (size > 0) {
        T* buffer = new T[2 * size];
        int pout = 0, pin = 1;
        for (int i = 0; i < size; i++)
            buffer[pout * size + i] = i > 0 ? in[i - 1] : initial;
        for (int stride = 1; stride < size; stride *= 2) {
            pout = 1 - pout;
            pin = 1 - pout;
            for (int i = 0; i < size; i++) {
                if (i >= stride)
                    buffer[pout * size + i] = combine(buffer[pin * size + i],
                     buffer[pin * size + i - stride]);
                else
                    buffer[pout * size + i] = buffer[pin * size + i];
            }
        }
        for (int i = 0; i < size; i++)
            out[i] = buffer[pout * size + i];
        delete[] buffer;
    }
}
```

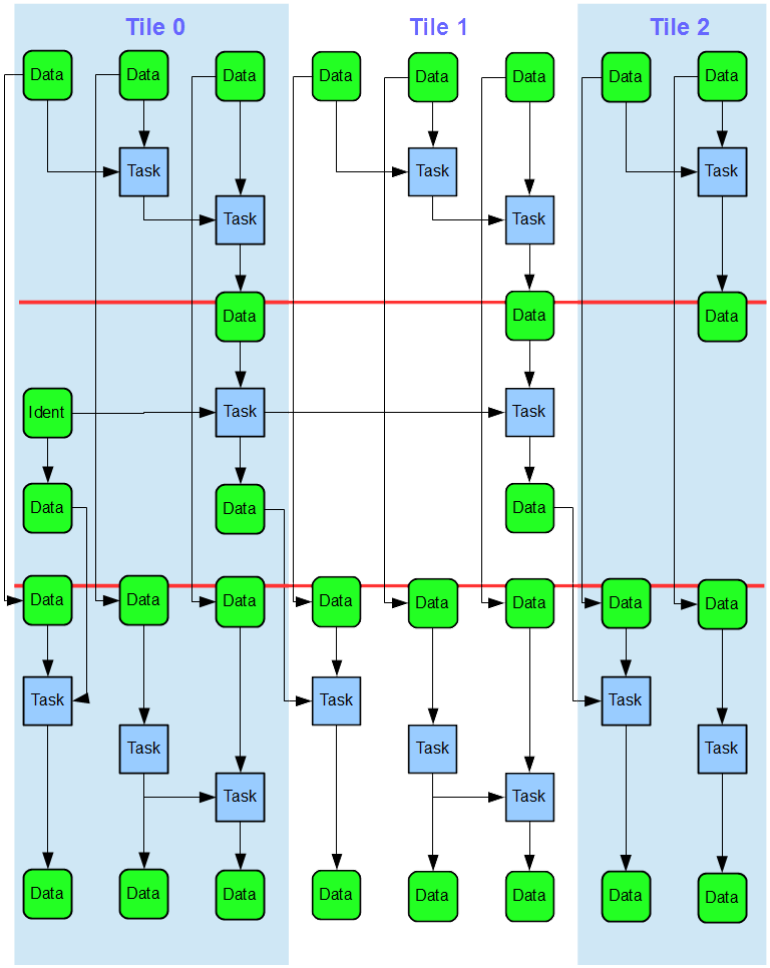
The reason that we call this solution naive is that its work has O(n log n) complexity. Since the serial version has O(n) complexity, this solution is clearly work-inefficient; it requires more work than the serial pattern requires.

Tiled Three-Step Solution

A tiled solution is one alternative to the naive algorithm. In this alternative, each processor operates on a single tile, with the processors working in parallel. Kkeeping the serial operations within each tile avoids the work-inefficient solution.

The tiled algorithm involves three distinct steps:

1. each thread independently reduces the values for its tile saving the result for the tile to the corresponding last element of the global array for use in the next step
2. a single thread performs an exclusive in place scan on the global array of reduced results for all tiles together, overwriting the elements of the global array
3. each thread independently performs the selected scan on the original input values for its tile using the exclusive scan term as its initial value



A Tiled Three-Step Solution to the Parallel Scan Pattern (McCool, et al., 2012)

Compared to the serial solution above (**prefix_scan.serial.cpp**), this parallel solution requires a reduction template with corresponding changes to the **scan** argument list. The following code is in preparation for an SPMD implementation:

```
template <typename T, typename C>
T reduce(
    const T* in, // points to the data set
    int n,       // number of elements in the data set
    C combine,   // combine operation
    T initial    // initial value
    ) {

    for (int i = 0; i < n; i++)
        initial = combine(initial, in[i]);
    return initial;
}

// incl_scan function definition

// excl_scan function definition

template <typename T, typename R, typename C, typename S>
void scan(
    const T* in,   // source data
    T* out,        // output data
    int size,      // size of source, output data sets
    R reduce,      // reduction expression
    C combine,     // combine expression
    S scan_fn,     // scan function (exclusive or inclusive)
    T initial      // initial value
    )
{
    const int tile_size = 2;
    if (size > 0) {
```

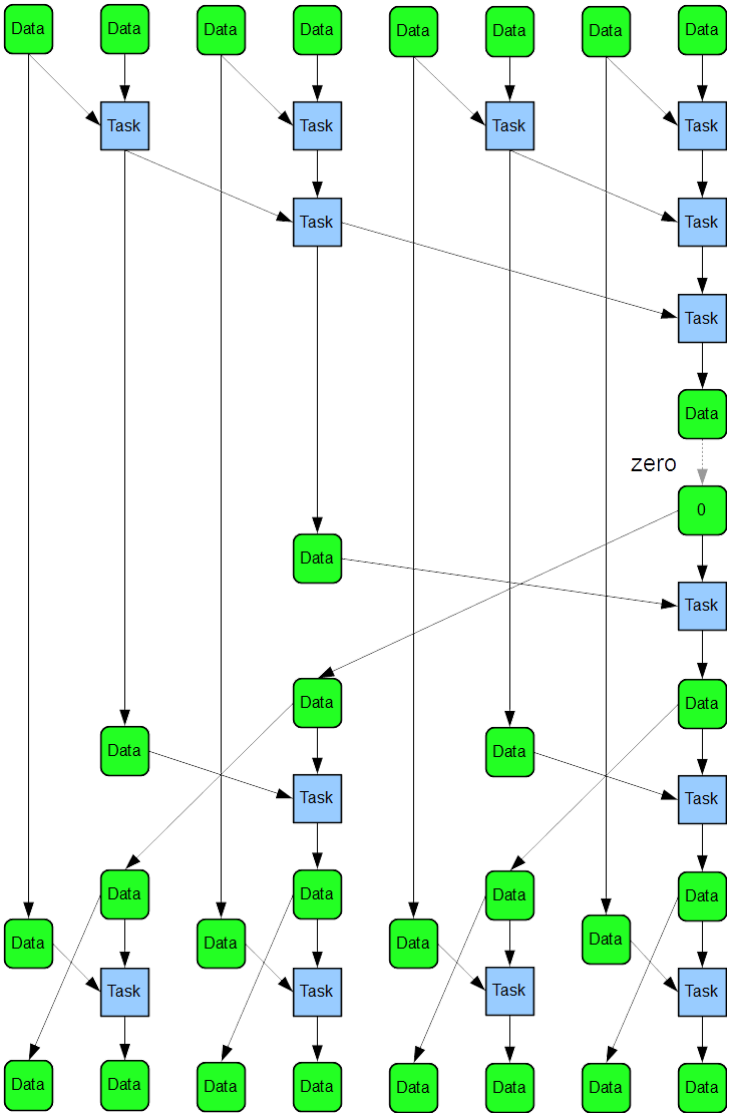
```
        // requested number of tiles
        int ntiles = (size - 1) / tile_size + 1;
        T* reduced = new T[ntiles];
        T* scanRes = new T[ntiles];
        int last_tile = ntiles - 1;
        int last_tile_size = size - last_tile * tile_size;

        // step 1 - reduce each tile separately
        for (int itile = 0; itile < ntiles; itile++)
            reduced[itile] = reduce(in + itile * tile_size,
            itile == last_tile ? last_tile_size : tile_size, combine, T(0));

        // step 2 - perform exclusive scan on all tiles using reduction outputs
        // store results in scanRes[]
        excl_scan(reduced, scanRes, ntiles, combine, T(0));

        // step 3 - scan each tile separately using scanRes[]
        for (int itile = 0; itile < ntiles; itile++)
            scan_fn(in + itile * tile_size, out + itile * tile_size,
             itile == last_tile ? last_tile_size : tile_size, combine,
             scanRes[itile]);
        delete [] reduced;
        delete [] scanRes;
    }
}

int main() {
    const int N = 9;
    int in[N] = { 3, 1, 7, 0, 1, 4, 5, 9, 2 };
    int out[N];
    // combine operation
    auto combine = [](int a, int b) { return a + b; };

    scan(in, out, N, reduce<int, decltype(combine)>, combine,
        incl_scan<int, decltype(combine)>, (int)0);

    for (int i = 0; i < N; i++) {
        std::cout << out[i] << ' ';
        if (i % 5 == 4) std::cout << std::endl;
    }
    std::cout << std::endl;

    scan(in, out, N, reduce<int, decltype(combine)>, combine,
        excl_scan<int, decltype(combine)>, (int)0);

    for (int i = 0; i < N; i++) {
        std::cout << out[i] << ' ';
        if (i % 5 == 4) std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

This algorithm has a maximum asymptotic speedup of $\Theta(N^{1/2})$. It requires twice as many invocations of the **`combine`** function as the serial scan (McCool, et al. 2012).

Balanced-Tree Solution

Guy Blelloch (1993) proposed a work-efficient solution based on a balanced-tree approach that consists of three sequential steps:

1. upsweep (partial reduction)
2. clearing of reduced value
3. a downsweep (complete the scan)



A Balanced-Tree Solution to the Scan Pattern (Blelloch, 1993)

A balanced-tree algorithm in preparation for a work-sharing implementation is:

```
template <typename T, typename C>
void incl_scan(
    const T* in, // source data
    T* out,      // output data
    int size,    // size of source, output data sets
    C combine,   // combine expression
    T initial    // initial value
    )
{
    // initialize
    for (int i = 0; i < size; i++)
        out[i] = in[i];

    // upsweep (reduction)
    for (int stride = 1; stride < size; stride <<= 1) {
        for (int i = 0; i < size; i += 2 * stride)
            out[2 * stride + i - 1] = combine(out[2 * stride + i - 1],
               out[stride + i - 1]);
    }

    // clear last element
    T last = out[size - 1];
    out[size - 1] = T(0);

    // downsweep
    for (int stride = size / 2; stride > 0; stride >>= 1) {
```

```
        for (int i = 0; i < size; i += 2 * stride) {
            T temp = out[stride + i - 1];
            out[stride + i - 1] = out[2 * stride + i - 1];
            out[2 * stride + i - 1] = combine(temp, out[2 * stride + i - 1]);
        }
    }

    // shift left for inclusive scan and add last
    for (int i = 0; i < size - 1; i++)
        out[i] = out[i + 1];
    out[size - 1] = last;
}

template <typename T, typename C>
void excl_scan(
    const T* in, // source data
    T* out,      // output data
    int size,    // size of source, output data sets
    C combine,   // combine expression
    T initial    // initial value
    )
{
    // initialize
    for (int i = 0; i < size; i++)
        out[i] = in[i];

    // upsweep (reduction)
    for (int stride = 1; stride < size; stride <<= 1) {
        for (int i = 0; i < size; i += 2 * stride)
            out[2 * stride + i - 1] = combine(out[2 * stride + i - 1],
               out[stride + i - 1]);
    }

    // clear last element
    out[size - 1] = T(0);

    // downsweep
    for (int stride = size / 2; stride > 0; stride >>= 1) {
        for (int i = 0; i < size; i += 2 * stride) {
            T temp = out[stride + i - 1];
            out[stride + i - 1] = out[2 * stride + i - 1];
            out[2 * stride + i - 1] = combine(temp, out[2 * stride + i - 1]);
        }
    }
}
```

This algorithm takes O(ln N) time, which is asymptotically better than the tiled algorithm (McCool, et al. 2012).

Note that the increment on **`i`** is **`2 * stride`**. Since the elements for a given **`stride`** (**`stride + i - 1`** and **`2 * stride + i - 1`**) are independent of any other iteration for that stride, the inner iteration can be parallelized. The entries for each stride iteration are listed in the table below.

stride	i	stride + i - 1	2 * stride + i - 1
4	0	3	7
2	0	1	3
2	4	5	7
1	0	0	1
1	2	2	3
1	4	4	5
1	6	6	7

FILTER APPLICATION

A filter application is one that may benefit from a prefix-scan algorithm.

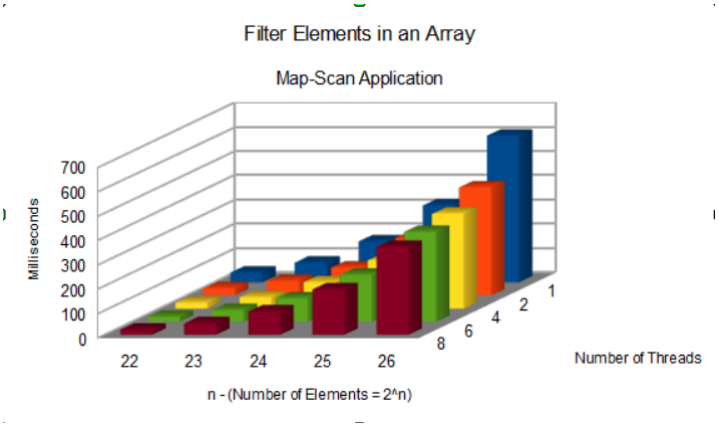
The following program creates a bit vector that identifies the elements that satisfy a filter condition, performs a prefix-scan on that vector and uses the resultant scan vector to move the qualifying elements from the input vector to their positions in the output vector.

```
// Map Scan Filter Solution
// filter.h

#include <omp.h>

template <typename T, typename F, typename C>
int map_scan(
    const T* in,   // source data
    T* out,        // output data
    int size,      // size of source, output data sets
    F filter,      // filter expression
    C combine      // combine expression
    )
{
    // initialize
    int* index = new int[size];
    #pragma omp parallel for
    for (int i = 0; i < size; i++)
        index[i] = filter(in[i]);

    // upsweep (reduction)
    for (int stride = 1; stride < size; stride <<= 1) {
        #pragma omp parallel for
        for (int i = 0; i < size; i += 2 * stride)
            index[2 * stride + i - 1] = combine(index[2 * stride + i - 1],
             index[stride + i - 1]);
    }

    // clear last element
    T last = index[size - 1];
    index[size - 1] = T(0);

    // downsweep
    for (int stride = size / 2; stride > 0; stride >>= 1) {
        #pragma omp parallel for
        for (int i = 0; i < size; i += 2 * stride) {
            T temp = index[stride + i - 1];
            index[stride + i - 1] = index[2 * stride + i - 1];
            index[2 * stride + i - 1] = combine(temp,
             index[2 * stride + i - 1]);
        }
    }

    // copy in[i] to out[index[i + 1] - 1] if index[i + 1] != index[i]
    #pragma omp parallel for
    for (int i = 0; i < size - 1; i++)
        if (index[i] < index[i + 1])
            out[index[i + 1] - 1] = in[i];
    if (index[size - 1] < last)
        out[last - 1] = in[size - 1];

    delete[] index;
    return last;
}
```

```
// Filter using a Prefix Scan
// filter.cpp

#include <iostream>
#include <cstdlib>
#include <chrono>
#include "filter.h"
using namespace std::chrono;
```

```
// report system time
//
void reportTime(const char* msg, steady_clock::duration span) {
    auto ms = duration_cast<milliseconds>(span);
    std::cout << msg << " - took - " <<
        ms.count() << " milliseconds" << std::endl;
}

int main(int argc, char** argv) {
    // command line arguments - none for testing, 1 for larger runs
    if (argc > 2) {
        std::cerr << argv[0] << ": invalid number of arguments\n";
        std::cerr << "Usage: " << argv[0] << "\n";
        std::cerr << "Usage: " << argv[0] << "  power_of_2\n";
        return 1;
    }

    // setup input array for processing and filter
    //
    // default values for testing
    const int N = 8;
    const int in_[N] = { 3, 1, 7, 0, 1, 4, 5, 9 };

    int n, nn = N, noutput;
    if (argc == 1) {
        n = N;
    }
    else if (argc == 2) {
        n = 1 << std::atoi(argv[1]);
        if (n < N) n = N;
    }

    int* input  = new int[n];
    int* output = new int[n];
    int  key    = 4;

    // initialize
    for (int i = 0; i < N; i++)
        input[i] = in_[i];
    #pragma omp parallel for
    for (int i = N; i < n; i++)
        input[i] = i % key;
    auto add    = [](int a, int b) { return a + b; };
    auto filter = [&](int a) { return (a % key) != 0; };

    // filter the input array and place results in output
    //
    steady_clock::time_point ts, te;
    ts = steady_clock::now();
    noutput = map_scan(input, output, n, filter, add);
    te = steady_clock::now();

    // output up to the first N elements of the filtered array
    //
    if (noutput < N)
        nn = noutput - 1;
    for (int i = 0; i < nn; i++)
        std::cout << output[i] << ' ';
    std::cout << output[noutput - 1] << std::endl;

    reportTime("Map-Scan", te - ts);
}
```

The timing statistics from executions of this application for different powers of 2 and different numbers of threads are presented in the figure below.



Filtering Elements of an Array using a Balanced Tree Map-Scan Algorithm

EXERCISES

- Barlas, G. (2015). Multicore and GPU Programming - An Integrated Approach. Morgan-Kaufmann. 978-0-12-417137-4. pp.179-192.
- Jordan, H. F., Alaghband, G. (2003). Fundamentals of Parallel Processing. Prentice-Hall. 978-0-13-901158-7. pp. 269-273.
- Mattson, T. (2013). Tutorial Overview | Introduction to OpenMP
- Dan Grossman's 2010 Lecture on Parallel Prefix and Parallel Sorting