Part B - Languages

OpenMP - Prefix Scan

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

Introduction | Parallel Algorithms | Exercises

The prefix scan pattern represents a template for solutions to problems with loop-carried dependencies; ones in which a particular iteration depends on the result of the previous iteration. These solutions have applications in searching, lexical analysis, sorting, string comparison and stream compaction. The parallel algorithms that implement this pattern must manage these dependencies. The pattern is similar to the reduction pattern, but, unlike that pattern, entails storing all the intermediate results.

This chapter introduces the prefix scan pattern and presents several algorithms that implement serial and parallel solutions. The serial algorithms covered here include a naive solution, a tiled three step solution and a balanced tree solution. These solutions serves as the basis for their parallel counterparts. In addition to those counterparts OpenMP supports a prefix scan directive. The SPMD and work sharing algorithms provide alternatives to this directive with more programmer control.

INTRODUCTION

The scan pattern covers two distinct loop-carried dendenciess. They differ on the problem's initial condition:

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

Consider the source code listed on the left. The loop-carried dependency is an inclusive prefix-sum iteration. The loop-carried dependency in the source code on the right is an exclusive prefix-sum iteration.

```
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

The inclusive scan takes an array of values (listed on the left side) and returns an array of cumulative values (listed on the right side), where **op** denotes the operation on a_i and a_{i+1}. :

```
[a0,
 a1,
 a2,
 ...,
 an-1]
```

```
[a0,
 a0 op a1,
 a0 op a1 op a2,
 ...,
 a0 op a1 op ... op an-1]
```

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

```
[3,
 1,
 7,
```

```
[3,
 1 + 3  = 4,
 7 + 4  = 11,
```

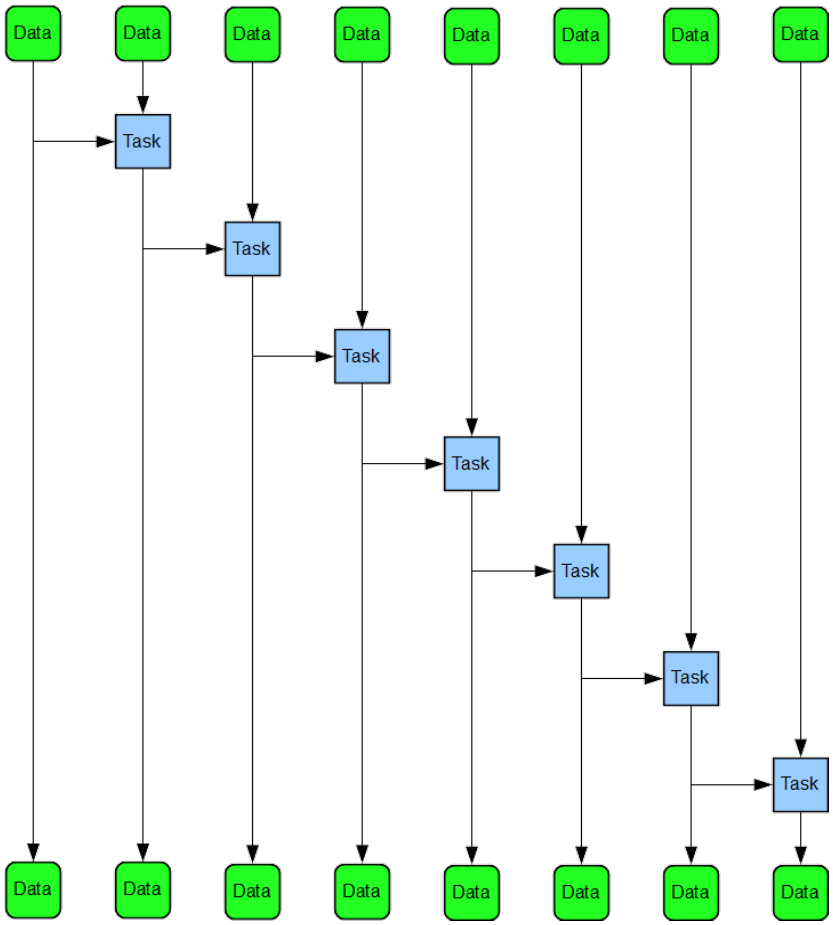
```
0,    ==> inclusive scan ==>          0 + 11 = 11,
4,                                    4 + 11 = 15,
1,                                    1 + 15 = 16,
6,                                    6 + 16 = 22,
3]                                    3 + 22 = 25]
```

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

The figure shown below represents a serial implementation of an inclusive scan. Note how the intermediate values remain in memory. Each task applies the selected combination operation to its operands:



A Serial Version of Inclusive Scan

The following templated function performs a serial inclusive scan using the combination 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 two 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

The exclusive scan takes an array of elements as listed on the left side and returns the array of elements listed on the right side:

$$[a_0, a_1, a_2, a_3, \ldots, a_{n-1}] \Longrightarrow \text{exclusive scan} \Longrightarrow [I, a_0, a_0 \text{ op } a_1, a_0 \text{ op } a_1 \text{ op } a_2, \ldots, a_0 \text{ op } a_1 \text{ op } \ldots \text{ op } a_{n-2}]$$

where `I` denotes the identity element for the combination operation.

For example, an exclusive prefix-sum operates on the set of values on the left to yield the set of values on the right:

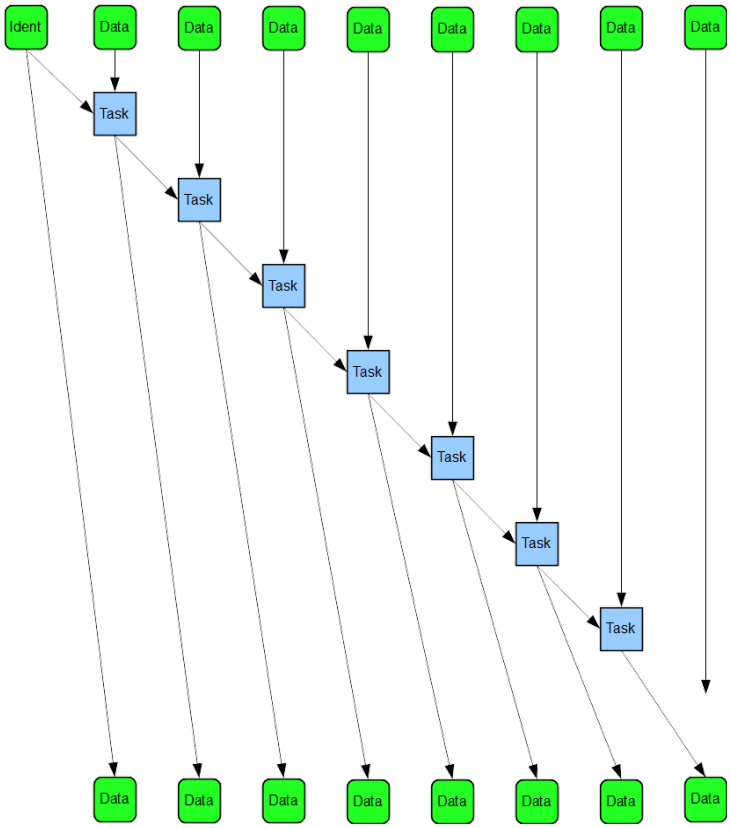
```
[3,                                  [0,
 1,                                   3 + 0  = 3,
 7,                                   1 + 3  = 4,
 0,   ==> exclusive scan ==>          7 + 4  = 11,
 4,                                   0 + 11 = 11,
 1,                                   4 + 11 = 15,
 6,                                   1 + 15 = 16,
 3]                                   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. The first element of the result, which is the identity element is not the result of any combination:



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 Algorithms

The client code for the direct algorithms is shown on the left. The results for no command line argument are shown on the right:

```
// Prefix Scan
// scan.cpp
// 2020.10.12
```

```
// Chris Szalwinski

#include <iostream>
#include "scan.h"

int main(int argc, char** argv) {
    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;
    }
    std::cout << "Simple Serial Prefix Scan" << std::endl;

    // initial values for testing
    const int N = 9;
    const int in_[N]{ 3, 1, 7, 0, 1, 4, 5, 9, 2 };

    // command line arguments: 0 for testing, 1 for large arrays
    int n, nt{ 1 };
    if (argc == 1) {
        n = N;
    }
    else {
        n = 1 << std::atoi(argv[1]);
        if (n < N) n = N;
    }
    int* in = new int[n];
    int* out = new int[n];

    // initialize
    for (int i = 0; i < N; i++)
        in[i] = in_[i];
    for (int i = N; i < n; i++)
        in[i] = 1;

    // combination operation
    auto add = [](int a, int b) { return a + b; };

    // Inclusive Prefix Scan
    //
    scan(in, out, n, add, incl_scan<int, decltype(add)>, (int)0);
    for (int i = 0; i < N; i++)
        std::cout << out[i] << ' ';                                    3 4 11 11 12
    std::cout << out[n - 1] << std::endl;                              16 21 30 32

    // Exclusive Prefix Scan
    //
    scan(in, out, n, add, excl_scan<int, decltype(add)>, (int)0);
    for (int i = 0; i < N; i++)
        std::cout << out[i] << ' ';                                    0 3 4 11 11
    std::cout << out[n - 1] << std::endl;                              12 16 21 30

    delete[] in;
    delete[] out;
}
```

The fifth argument to the **scan** function selects between exclusive and inclusive scans. The fourth argument identifies the combination operation. The results shown correspond to a run without a command line argument. The command line argument, if any, provides the exponent in calculating the number of data elements (2^exponent) for more realistic assessments.

The **decltype(combine)** template argument identifies the type of the combination operation.

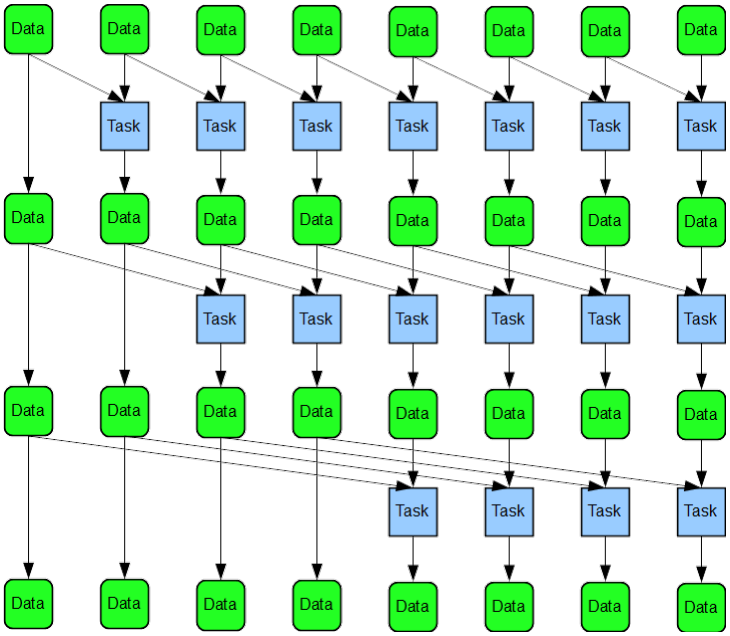
Simple Serial Algorithm

The following source code implements a simple serial algorithm for inclusive and exclusive scans using any combination operation. The **combine** function identifies the operation to be applied to the operand pair.

```
// Prefix Scan Example - Simple Serial
// scan.h
// 2020.10.12
// Chris Szalwinski
// 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
) {

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

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

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));
}
```

The Naive Algorithm

Hillis and Steele (1986) published an early solution to the prefix-scan problem. The following figure shows the combination operations in this algorithm. The data is updated in stages. Each row of tasks constitues a stage. The stride between the data operands doubles in moving from one stage to the next.



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

Consider the second row in this graph. For columns not subject to further combination operations the data elements remain unchanged. For the columns contributing to the combination operation the values of data elements are separated by the stride for this second stage. To avoid pre-mature over-writing of data, this algorithm requires double-buffering. The highlighted source code shows the cyclic switching between buffers.

The following templated functions implement this algorithm.

```
// Prefix Scan - Hillis and Steele
// scan.h
// 2020.10.12
// Chris Szalwinski

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;
    }
}

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));
}
```

We call this solution naive because its work has O(n log n) complexity. Since the serial version has only O(n) complexity, this solution is work-inefficient; that is, it requires more work than does the simple serial algorithm.

Three-Step Tiled Algorithm

The three-step tiled algorithm is an alternative to the naive algorithm that incorporates a reduction algorithm. Compared to the client code above, this alternative requires a reduction template with corresponding changes to the **scan** argument list. The modified client code is:

```
// Prefix Scan
// scan.cpp
// 2020.10.12
// Chris Szalwinski

#include <iostream>
#include "scan.h"
```

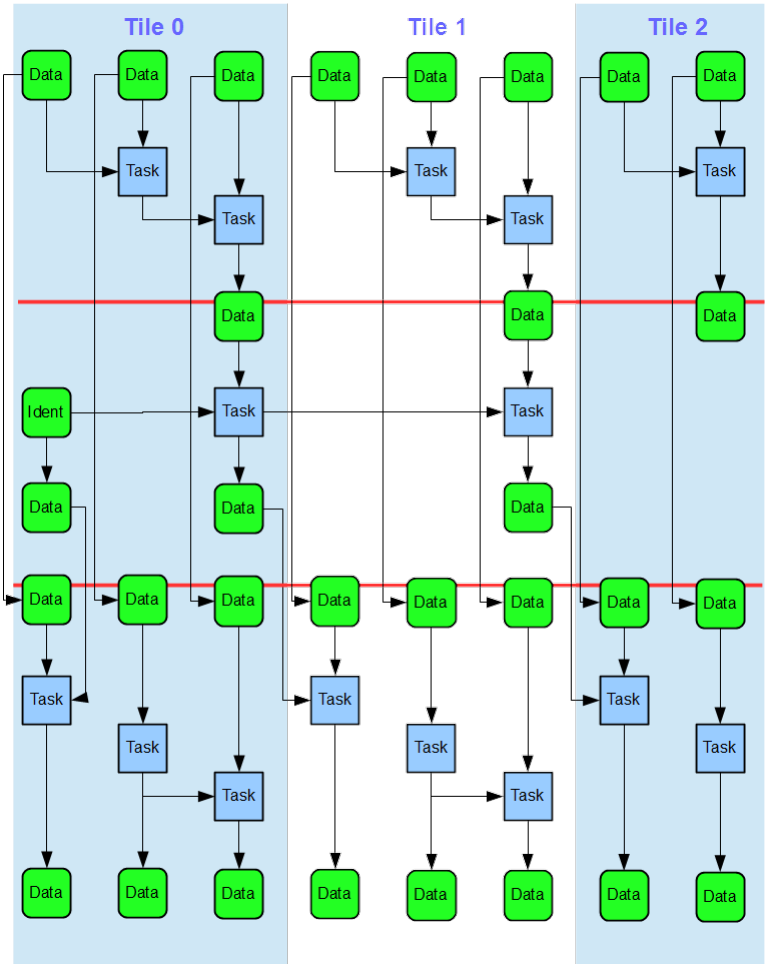
```
int main(int argc, char** argv) {
    if (argc > 2) {
        std::cerr << argv[0] << ": invalid number of arguments\n";
        std::cerr << "Usage: " << argv[0] << "\n";
        std::cerr << "Usage: " << argv[0] << "  power_of_2\n";
        return 1;
    }
    std::cout << "Simple Serial Prefix Scan" << std::endl;

    // initial values for testing
    const int N = 9;
    const int in_[N]{ 3, 1, 7, 0, 1, 4, 5, 9, 2 };

    // command line arguments: 0 for testing, 1 for large arrays
    int n, nt{ 1 };
    if (argc == 1) {
        n = N;
    }
    else {
        n = 1 << std::atoi(argv[1]);
        if (n < N) n = N;
    }
    int* in = new int[n];
    int* out = new int[n];

    // initialize
    for (int i = 0; i < N; i++)
        in[i] = in_[i];
    for (int i = N; i < n; i++)
        in[i] = 1;

    // combination operation
    auto add = [](int a, int b) { return a + b; };

    // Inclusive Prefix Scan
    //
    scan(in, out, n, reduce<int, decltype(add)>,
        add, excl_scan<int, decltype(add)>, (int)0);
    for (int i = 0; i < N; i++)
        std::cout << out[i] << ' ';
    std::cout << out[n - 1] << std::endl;

    // Exclusive Prefix Scan
    //
    scan(in, out, n, reduce<int, decltype(add)>,
        add, excl_scan<int, decltype(add)>, (int)0);
    for (int i = 0; i < N; i++)
        std::cout << out[i] << ' ';
    std::cout << out[n - 1] << std::endl;

    delete[] in;
    delete[] out;
}
```

The calls to the **scan** function include the address of a **reduce** templated function as an additional argument.

In the tiled three-step algorithm, the data is partitioned into tiles, each of which can work independently of any other tile. Restricting the combination operations to each tile avoids the work-inefficiency of the naive solution. This algorithm consists of 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 (McCool, et al., 2012)

The following code is a mock SPMD implementation for a single thread:

```
// Prefix Scan - Tiled Three-Step
// scan.h
// 2020.10.12
// Chris Szalwinski

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;
}

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 R, typename C, typename S>
int 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
)
{
    int mtiles = 8;
    if (size > 0) {

        // allocate memory for maximum number of tiles
        T* stage1Results = new T[mtiles];
        T* stage2Results = new T[mtiles];

        {
            // determine tile size
            int last_tile = mtiles - 1;
            int tile_size = (size - 1) / mtiles + 1;
            int last_tile_size = size - tile_size * last_tile;

            // step 1 - reduce each tile separately
            for (int itile = 0; itile < mtiles; ++itile)
                stage1Results[itile] = reduce(in + tile_size * itile,
                 itile == last_tile ? last_tile_size : tile_size, combine, T(0));

            // step 2 - perform exclusive scan on stage1Results
            // store exclusive scan results in stage2Results[]
            excl_scan(stage1Results, stage2Results, mtiles, combine, T(0));

            // step 3 - scan each tile separately using stage2Results[]
            for (int itile = 0; itile < mtiles; ++itile)
                scan_fn(in + tile_size * itile, out + tile_size * itile,
```

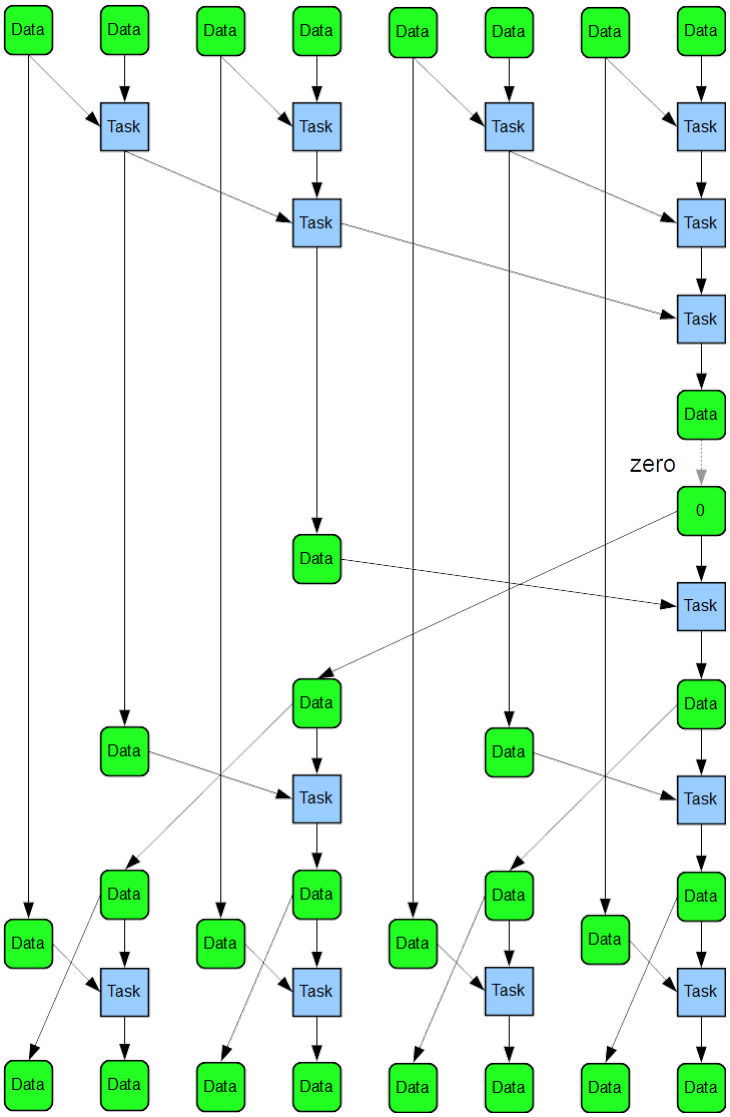
```
                    itile == last_tile ? last_tile_size : tile_size, combine,
                    stage2Results[itile]);
            }

            // deallocate memory
            delete[] stage1Results;
            delete[] stage2Results;
        }

        return mtiles;
    }
```

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 (a partial reduction)
2. clearing of the reduced value
3. a downsweep (completes the scan)



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

The client code for this algorithm is the same as that for the simple serial and naive algorithms.

A source code implementation of the balanced-tree algorithm itself 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]);
            }
        }
    }
```

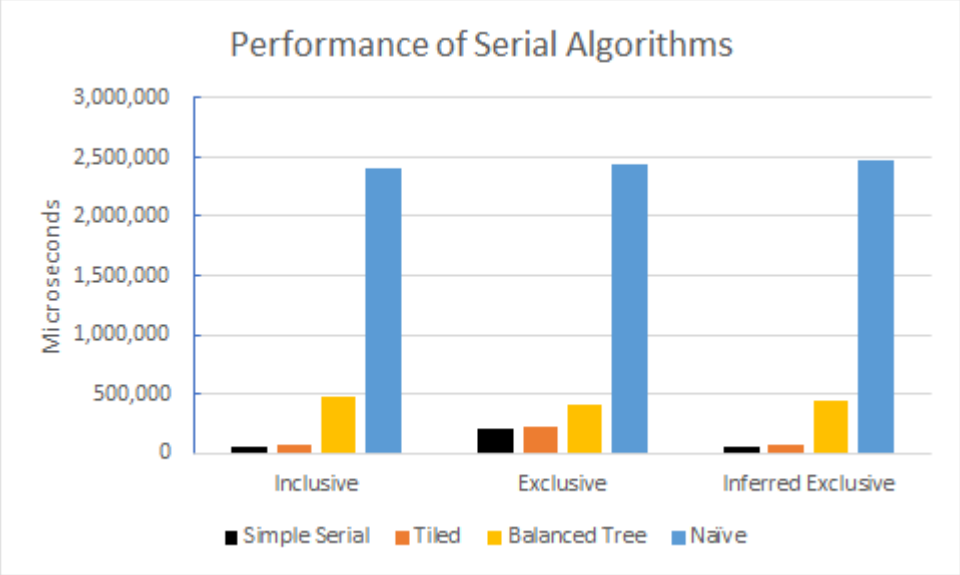
This algorithm takes O(ln N) time, which is asymptotically better than the tiled algorithm (McCool, et al. 2012).

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

Comparison

The figure below compares the performance of these different algorithms in serial mode. Note that the balanced tree solutions are consistently high, the naive solutions are highest, and the inferred exclusive solutions aremuch improved over the direct exclusive solutions. The tiled solutions are closest to the simple serial solutions, which consistently show the best performance.



Comparison of Solutions to the Scan Pattern

These results suggest that the tiled algorithm stands the best chance for improvement over the simple serial algorithm. The naive and balanced tree algorithms include partial reductions based on strides that increase across stages, which appears to be a relatively expensive technique.

PARALLEL ALGORITHMS

The parallel algorithms for inclusive and exclusive scans follow as upgrades to the serial algorithms using the basic OpenMP directives. In addition to these, OpenMP provides a dedicated directive specialized for scan problems.

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

Upgrades to the Serial Solutions

Three-Step Tiled Algorithm

The upgrades to the **scan** function of the code for the three-step tiled algorithm are highlighted below:

```
template <typename T, typename R, typename C, typename S>
int 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
)
{
    int mtiles = 1;
    if (size > 0) {
        // allocate memory for maximum number of tiles
        mtiles = omp_get_max_threads();
        T* stage1Results = new T[mtiles];
        T* stage2Results = new T[mtiles];
        omp_set_num_threads(mtiles);

        #pragma omp parallel
        {
            // step 1 - reduce each tile separately
            int itile = omp_get_thread_num();
            int ntiles = omp_get_num_threads();
            if (itile == 0) mtiles = ntiles;
            int last_tile = ntiles - 1;
            int tile_size = (size - 1) / ntiles + 1;
            int last_tile_size = size - tile_size * last_tile;
            tile_size = itile == ntiles - 1 ? last_tile_size : tile_size;
            stage1Results[itile] = reduce(in + itile * tile_size,
             itile == last_tile ? last_tile_size : tile_size, combine, T(0));
            #pragma omp barrier

            // step 2 - perform exclusive scan on stage1Results
            // store exclusive scan results in stage2Results[]
            #pragma omp single
            excl_scan(stage1Results, stage2Results, ntiles, combine, T(0));

            // step 3 - scan each tile separately using stage2Results[]
            scan_fn(in + itile * tile_size, out + itile * tile_size,
             itile == last_tile ? last_tile_size : tile_size, combine,
             stage2Results[itile]);
```

```
        }

        // deallocate memory
        delete[] stage1Results;
        delete[] stage2Results;
    }

    return mtiles;
}
```

Balanced Tree Algorithm

The upgrades to the **scan** function of the code for the three-step tiled algorithm are highlighted below:

```
template <typename T, typename C>
int 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
)
{
    int nt = 1;

    // initialize
    #pragma omp parallel for schedule(guided)
    for (int i = 0; i < size; i++)
        out[i] = in[i];

    // upsweep (reduction)
    for (int stride = 1; stride < size; stride <<= 1) {
        #pragma omp parallel for schedule(guided)
        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) {
        #pragma omp parallel
        {
            nt = omp_get_num_threads();
            #pragma omp for schedule(guided)
            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;

    return nt;
```

```
}

template <typename T, typename C>
int 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
)
{
    int nt = 1;

    // initialize
    #pragma omp parallel for schedule(guided)
    for (int i = 0; i < size; i++)
        out[i] = in[i];

    // upsweep (reduction)
    for (int stride = 1; stride < size; stride <<= 1) {
        #pragma omp parallel for schedule(guided)
        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) {
        #pragma omp parallel
        {
            nt = omp_get_num_threads();
            #pragma omp for schedule(guided)
            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]);
            }
        }
    }
    return nt;
}

template <typename T, typename C, typename S>
int 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
)
{
    return scan_fn(in, out, size, combine, T(0));
}
```

The Scan Directive

OpenMP offers support for prefix scan within its **simd** directive augmented by the **reduction** clause. The augmented directive takes the form

```
#pragma omp simd reduction(inscan, operation:variable)
```

The **reduction** clause includes the identifier **inscan** before the operation-variable pair. Within the scope of this **simd** directive, a separate directive specifies the type of the scan

```
#pragma omp scan [in|ex]clusive(variable)
```

where **[in|ex]clusive** identifies the type of scan and **variable** is the name of the accumulator.

For an inclusive sum on array **in[n]** with the result stored in array **out[n]**, we write:

```
int sum = 0;
#pragma omp simd reduction(inscan, +:sum)
for (int i = 0; i < n; ++i) {
    sum += in[i];
    #pragma omp scan inclusive(sum)
    out[i] = sum;
}
```

For an exclusive sum on array **in[n]** with the result stored in array **out[n]** we write:

```
int sum = 0;
#pragma omp simd reduction(inscan, +:sum)
for (int i = 0; i < n; ++i) {
    out[i] = sum;
    #pragma omp scan exclusive(sum)
    sum += in[i];
}
```

where here **sum** is the accumulator.

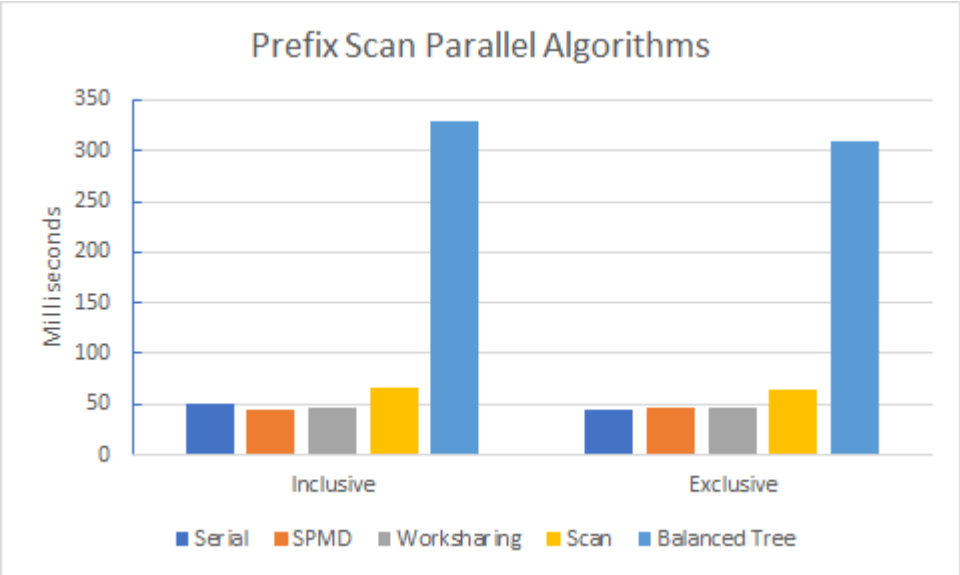
The upgrades to the **scan** function of the simple serial code are highlighted below:

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


    #pragma omp simd reduction(inscan, +:initial)
    for (int i = 0; i < size; i++) {
        initial += in[i];
        #pragma omp scan inclusive(initial)
        out[i] = initial;
    }
    return 1; // 1 thread
}

template <typename T>
int excl_scan(
    const T* in,                // source data
    T* out,                     // output data
```

```
    int size,                   // size of data sets
    T initial                   // initial value
) {

    if (size > 0) {
        #pragma omp simd reduction(inscan, +:initial)
        for (int i = 0; i < size - 1; i++) {
            out[i] = initial;
            #pragma omp scan exclusive(initial)
            initial += in[i];
        }
        out[size - 1] = initial;
    }
    return 1; // 1 thread
}

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

Comparison of Parallel Solutions

The figure below compares the performance of these different algorithms in parallel mode. Note that the balanced tree solutions are consistently high and the SIMD scan solutions are next highest. The tiled solutions are closest to the simple serial solutions, which consistently show the best performance.



Comparison of Parallel Solutions to the Scan Pattern

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