Part B - Languages

OpenMP - Map Reduce

Describe the two most common parallel patterns
Implement the map and reduce algorithms using OpenMP directives

Map | Reduce | Exercises

The simplest and most common pattern in parallel programming is the map pattern. The map pattern is often combined with another pattern - the reduce pattern - for a compound solution. Hadoop technology is based on the Map-Reduce concept. Some writers associate the success of Google's search engine with this compound pattern.

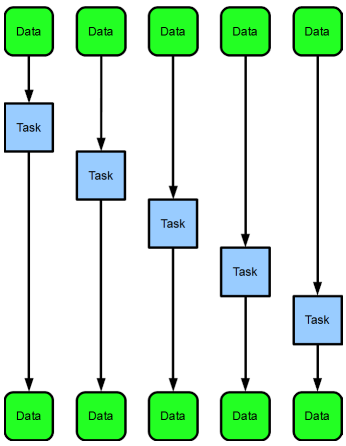
This chapter describes several algorithms that implement the map and reduce patterns. In each case, the serial version of the code is presented, followed by common parallel OpenMP implementations.

THE MAP PATTERN

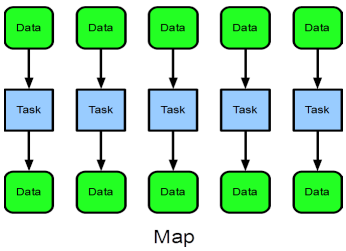
Introduction

The Map pattern models the same task executing on different subsets of a set of data elements. The execution of the task on one subset is independent of the execution of the same task on any other subset. Since the tasks are independent of one another, they can execute in parallel without further analysis.

Serial versions of map algorithms have growth rates of O(n). Parallel versions reduce the elapsed times from O(n) to O(n/p) in direct proportion to the number of processors (p) available. The depth of each figure below represents the elapsed time.



A Serial Version of the Map Pattern



A Parallel Version of the Map Pattern

Elemental Function

The task executed on each element of a collection is called the pattern's *elemental function*. This elemental function operates on each data element separately. The data elements are completely unrelated. Due to their independence the tasks do not communicate with one another.

The data that an elemental function accesses are classified into two distinct groups

- *uniform* - data that are the same across all instances of the elemental function
- *varying* - data that may differ from one instance to another of the elemental function

The map pattern is deterministic as long as the solution does not depend on the order of execution of the elemental function.

Assumptions

The map pattern assumes that the elemental function

- has no side effects
- performs no random writes to memory

The elemental function may read data in memory other than its assigned data.

Example - saxpy

The **saxpy** operation represents the scaling of a vector by a constant factor, followed by the addition of a second vector to the scaled result and finally followed by storing the final result in the second vector. This operation is the conventional prototype for implementing the map pattern and takes its name from the corresponding function in the BLAS (Basic Linear Algebra Subprograms) specification.

The **s** in the operation's name identifies a single precision evaluation. The **a** denotes alpha - the constant factor. **a** is the uniform variable. The **x** refers to the original input vector. The **p** denotes the plus operation. The **y** refers to the augmenting input vector. The elements of both vectors are varying.

The **saxpy** operation is represented mathematically by the following statement

```
y[i] := a * x[i] + y[i]
```

This exhibits a flow dependence, but this flow dependence is not carried from one element to another: that is, it is not loop-carried.

Serial Version

The serial version of **saxpy** is listed in the first function on the left. The result of execution is shown on the right:

```
// Map Pattern
// saxpy.cpp

#include <iostream>
constexpr int N = 1000;

void saxpy(float a, const float* x, float* y, int n) {
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}

 int main() {
    // allocate on the heap
    float* x = new float[N];
    float* y = new float[N];
    float a{ 2.0f };

    // initialize
    for (int i = 0; i < N; i++)
        x[i] = y[i] = 1.0f;

    saxpy(a, x, y, N);

    // sum y[i]
    float s = 0.0f;
    for (int i = 0; i < N; i++)
        s += y[i];

    std::cout << "Sum y[i] = " << s << std::endl;             Sum y[i] = 3000
```

```
    // deallocate
    delete [] x;
    delete [] y;
}
```

OpenMP Implementations

OpenMP implementation options for the map pattern include:

- SPMD - the original technique
- Work Sharing - Multi-threading with internal scheduling of threads
- SIMD - vectorization
- Work Sharing with Vectorization - Multi-threading with internal scheduling of threads and vectorization

SPMD Implementation

The SPMD implementation distributes the work across a team of threads under explicit programmer control. We allocate the data elements to each thread in a team and the threads execute uniformly.

```
void saxpy(float a, const float* x, float* y, int n) {
    #pragma omp parallel // start a parallel region
    {
        int tid = omp_get_thread_num();
        int nt = omp_get_num_threads();
        for (int i = tid; i < n; i += nt)
            y[i] = a * x[i] + y[i];
    } // end of the parallel region
}
```

The control settings `i = tid` and `i += nt` in the `for` clause explicitly allocate the data elements to their respective threads.

Work-Sharing Implementation

The work-sharing implementation allocates the data elements across the team of threads according to a *scheduling option*. We have less control, can accept the default scheduling option or specify the option ourselves. The `for` construct implements work sharing.

Work Sharing - 09 Part 1 Module 5

```
void saxpy(float a, const float* x, float* y, int n) {
    #pragma omp parallel // start a parallel region
    {
        #pragma omp for // accept the default scheduling option
        for(int i = 0; i < n; i++)
            y[i] = a * x[i] + y[i];
    } // end of the parallel region
}
```

We can combine the separate constructs into a compound `parallel for` construct in the same directive.

The optional scheduling clause on a `for` construct specifies how we would like to partition the `n` elements into chunks. Each chunk is executed on a single thread.

For example, to schedule the iterations at compile time into chunks of 4 per thread, we write:

```
void saxpy(float a, const float* x, float* y, int n) {
    #pragma omp parallel for schedule[static, 4]
    for(int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
```

```
}
```

OpenMP supports five scheduling strategies for work sharing:

- **`schedule[static, chunk_size]`** - designed for iterations of more or less equal length - **`chunk_size`** defaults to n/p - chunks are assigned to threads in a round-robin fashion.
- **`schedule[dynamic, chunk_size]`** - designed for iterations of variable length - **`chunk_size`** defaults to 1 - each thread executes a group and then requests the next group
- **`schedule[guided, chunk_size]`** - variable length iterations with shrinking groups as unassigned iterations diminish - **`chunk_size`** defaults to 1
- **`schedule[auto]`** - leaves the scheduling to the compiler or to the OpenMP runtime
- **`schedule[runtime]`** - determined at runtime by an environment variable setting - useful for testing

We can specify the scheduling at environment, program or directive levels. OpenMP selects the schedule in priority order from specific to general (1 is highest priority):

1. compiler directive clause **`schedule(static|dynamic|guided|auto[, chunk_size])`**
2. runtime library call **`omp_set_schedule(omp_sched_t kind, int chunk_size)`**
   `kind=omp_sched_static|omp_sched_dynamic|omp_sched_guided|omp_sched_auto`
3. environment variable **`OMP_SCHEDULE=static|dynamic|guided|auto[,chunk_size]`**

Chunking the collection into sub-collections of prescribed chunk size is also called *tiling*.

OpenMP SIMD Implementation

The SIMD implementation distributes the data across vector lanes in a vactor register. The **`simd`** construct implements SIMD processing without multi-threading.

```
void saxpy(float a, const float* x, float* y, int n) {
    #pragma omp simd
    for(int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

OpenMP Work-Sharing with SIMD Implementation

The work-sharing SIMD implementation distributes the tasks across a team of threads and across the vector lanes for each thread. The **`parallel for simd`** construct implements this option.

```
void saxpy(float a, const float* x, float* y, int n) {
    #pragma omp parallel for simd
    for(int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

Visual Studio Solution

You can find a Visual Studio solution with each of these projects here.

THE REDUCE PATTERN

Introduction

The reduce pattern is the second most popular pattern in parallel programming. This pattern applies the same function to two data elements to produce a single result. Applied successively to an entire collection of data elements, this pattern produces a single data value for the collection.

The following mathematical formula expresses the combination operation:

```
f(a, b) = a op b
```

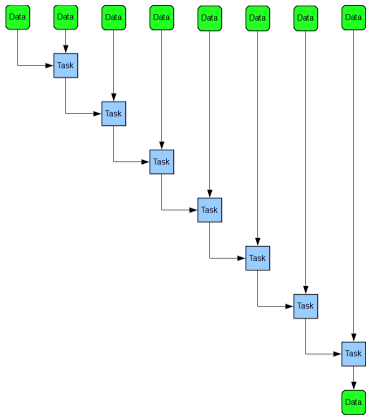
f denotes the function that performs the operation **op** on data elements **a** and **b**.

Reduction Operations

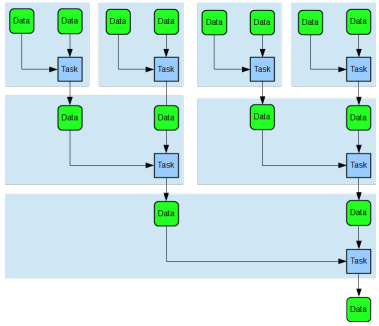
The reduction operations that OpenMP supports include (initialization constants are noted in parentheses):

- **+** - addition - (0)
- ***** - multiplication - (1)
- **-** - subtraction - (0)
- **&** - bit-wise and - (~0)
- **|** - bit-wise or - (0)
- **^** - bit-wise exclusive or - (0)
- **&&** - logical and - (1)
- **||** - logical or - (0)
- **max** - maximum - least representable number
- **min** - minimum - most representable number

The symbolic representations of the reduce pattern's serial and parallel implementations are shown below:



A Serial Version of the Reduce Pattern



A Parallel Version of the Reduce Pattern

Each implementation applies the specified function to each element of the data set in order from first to last, successively repeating on the accumulated result. Note that the parallel implementation uses a staging algorithm to reduce the elapsed time.

Assumptions

The reduce pattern assumes that

- the data elements can be combined in any order (associative law holds)
- an identity element exists - guarantees meaning if the number of data elements is zero

Since the arrangement of the data elements is open for any reduction algorithm, the reduction operator must be one that exhibits associativity. The associativity property enables reordering of the data elements. Operations on floating-point data are only approximately associative.

A reordering of the data elements may improve reduction efficiency with vector processors. Regrouping requires that the reduction operator exhibits not only associativity but also commutativity.

The two mathematical properties that are important in implementing the reduce pattern are:

- the operation's associativity - examples: + * && || ^ max min matmul quaternion multiply
- the operation's commutativity - examples: + * && (yes) - / saturation (no)

Loop-Carried Data Dependencies

Sources: Barlas, G. (2015), Jordan, H.F., Alaghband, G. (2003).

The reduction pattern involves loop-carried data dependencies, which may be managed using the **`reduction`** clause. The three code snippets below show how to use the work sharing construct **`for`** with the **`reduction`** clause to manage loop-carried dependencies. The source code containing the loop-carried dependency is listed on the left. The work sharing solution using the **`reduction`** clause is listed on the right.

- A loop-carried flow dependency caused by the presence of a reduction variable. The **`for`** construct partitions the loop and localizes the loop-carried dependency into separate dependencies within each thread. Each thread executes its loop-carried flow dependency serially. The team of threads executes in parallel and assembles the result subsequently.

```
float sum = 0.0f;
for (int i = 0; i < n; i++)
    sum = sum + f(i);
```

```
float sum = 0.0f;
#pragma omp for reduction(+:sum)
for (int i = 0; i < n; i++)
    sum = sum + f(i);
```

- A loop-carried output dependency where a unique value results from the iteration. The **`for`** construct partitions the loop and localizes the loop-carried dependency into separate dependencies within each thread. Each thread executes its loop-carried flow dependency serially. The team of threads executes in parallel and assembles the result subsequently.

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

```
w = x[0];
#pragma omp for reduction(max:w)
for (int i = 1; i < n; i++)
    if (w < x[i])
        w = x[i];
}
```

- Two loop-carried flow dependencies, where one is caused by consumption of a value produced in a previous iteration. One of the dependencies can be parallelized, the other not. The parallelizable dependency is separated from the non-parallelizable one. This refactoring is called *loop fission*.

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

```
for (int i = 1; i < n; i++)
    x[i] = x[i] + x[i - 1];
w = y[0];
#pragma omp parallel for reduction(+:w)
for (int i = 1; i < n; i++)
    w = w + y[i];
```

OpenMP Algorithms

Common OpenMP algorithms that implement the Reduce pattern include:

- reduction clause implementations
- fixed-tile-size using the SPMD programming model
- stride-based tiling with and without vectorization

Tiling divides the data into chunks. Each chunk is called a *tile* or a *block*. The algorithm operates on each chunk separately.

Reduction Clause Algorithms

The most straightforward implementation of the reduce pattern in OpenMP involves the **`reduction`** clause and requires explicit identification of the reduction operation.

The client application that calls the implementations described in this sub-section is:

```
// Reduction Pattern - Reduction Clause
// reduce.cpp

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

constexpr int N = 1 << 26;

int main() {
    int* a = new int[N]; // data set
    for (int i = 0; i < N; ++i) a[i] = 1;

    int sum = reduce(0, a, N); // perform reduction

    std::cout << "sum is " << sum << std::endl;        sum is 67108864
    delete[] a;
}
```

sum receives the result of the reduction. The test data is stored in **a**.

The serial implementation of **reduce** for a summation on an array of **int**s is

```
// Reduction Pattern - Serial Algorithm for ints
// reduce.h

int reduce(
    int identity,  // identity value for an empty set
    const int* a,  // points to the data set
    int n          // number of elements in the data set
) {

    int accum = identity;
    for (int i = 0; i < n; i++)
        accum += a[i];
    return accum;
}
```

The templated implementation of **reduce** for an array of any type **T** is

```
// Reduction Pattern - Serial Algorithm for Type T
// reduce.h

template <typename T>
T reduce(
    T identity,  // identity value for an empty set
    const T* a,  // points to the data set
    int n        // number of elements in the data set
) {

    T accum = identity;
    for (int i = 0; i < n; i++)
        accum = combine(accum, a[i]);
    return accum;
}
```

Vectorization

The **reduce** function below implements vectorization:

```
// Reduction Pattern - Vectorization
// reduce.h

#include <omp.h>

template <typename T>
T reduce(
    T identity,  // identity value for an empty set
    const T* a,  // points to the data set
    int n        // number of elements in the data set
) {

    T accum = identity;
    #pragma omp simd reduction(+:accum)
    for (int i = 0; i < n; i++)
        accum += a[i];
    return accum;
}
```

Work-Sharing

The **reduce** function below implements a specialized version of working sharing:

```
// Reduction Pattern - Work Sharing
// reduce.h

#include <omp.h>

template <typename T>
T reduce(
    T identity,  // identity value for an empty set
    const T* a,  // points to the data set
    int n        // number of elements in the data set
) {

    T accum = identity;
    #pragma omp parallel for reduction(+:accum)
    for (int i = 0; i < n; i++)
        accum += a[i];
    return accum;
}
```

Work-Sharing and Vectorization

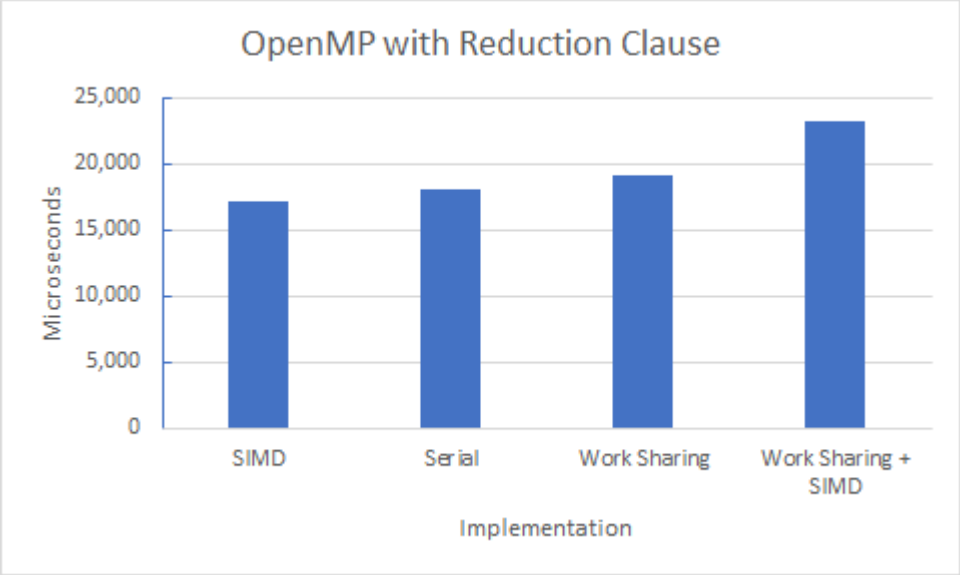
The **reduce** function below adds vectorization:

```
// Reduction Pattern - Work Sharing and Vectorization
// reduce.h

#include <omp.h>

template <typename T>
T reduce(
    T identity,  // identity value for an empty set
    const T* a,  // points to the data set
    int n        // number of elements in the data set
) {

    T accum = identity;
    #pragma omp parallel for simd reduction(+:accum)
```

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

Visual Studio Solution

You can find a Visual Studio solution using the Intel 19.1 compiler with O2 optimization on each of these algorithms here. The timing statistics for the different options are shown in the figure below.



OpenMP Implementations of the Reduce Pattern with Reduction Clause

Algorithms without the Reduction Clause

The client application that calls the implementations described in this sub-section is:

```
// Reduction Pattern
// reduce.cpp

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

constexpr int N = 1 << 26;

int main() {
    int* a = new int[N]; // data set
    for (int i = 0; i < N; ++i) a[i] = 1;
    auto add = [](const int& a, const int& b) {
        return a + b;    // reduction operation is +
    };

    int sum = reduce(add, 0, a, N); // perform reduction

    std::cout << "sum is " << sum << std::endl;              sum is 67108864
    delete[] a;
}
```

sum receives the result of the reduction, the lambda expression **add** defines the reduction operation, and the test data is stored in **a**.

Serial Version

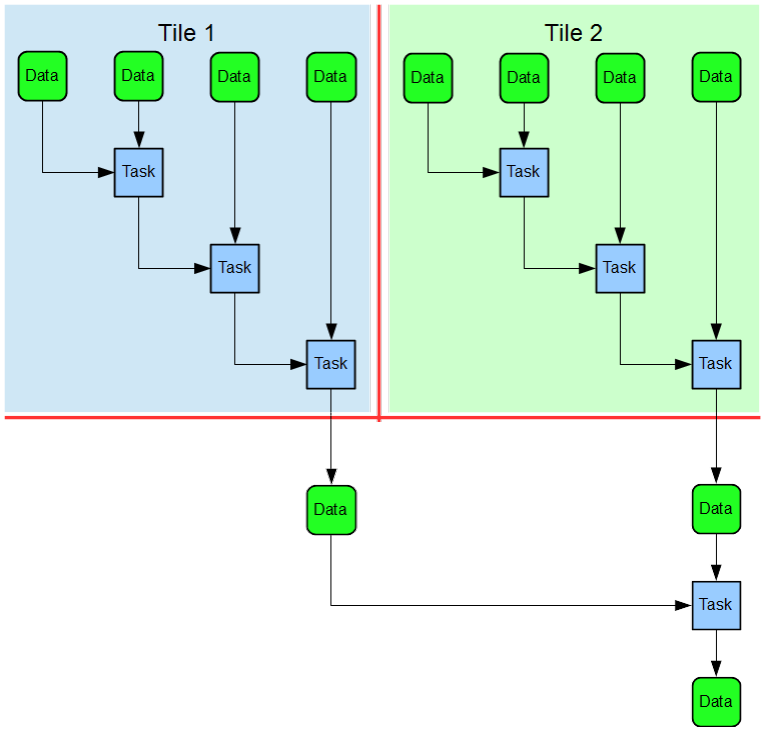
The templated implementation of **reduce** for an array of type **T** and a reduction operation of type **C** is

```
// Reduction Pattern - Serial Implementation
// reduce.h

template <typename T, typename C>
T reduce(
    C combine,   // function that performs the operation
    T identity,  // identity value for an empty set
    const T* a,  // points to the data set
    int n        // number of elements in the data set
) {

    T accum = identity;
    for (int i = 0; i < n; i++)
        accum = combine(accum, a[i]);
    return accum;
}
```

The template parameter (**C**) receives the type of the reduction or combination operation.

Fixed Tile Size with SPMD

The fixed-tile-size algorithm is a single-stage multi-threaded algorithm, illustrated in the figure below for a team of 2 threads. The number of tiles corresponds to the number of threads made available by the operating system. The size of each tile is determined from the number of elements and the number of tiles. Each thread performs a serial reduction on the data allocated to its tile. An array of elements of size equal to the number of tiles holds the results from each serial reduction. Outside the parallel region, the algorithm combines these partial results to obtain the final reduction result.



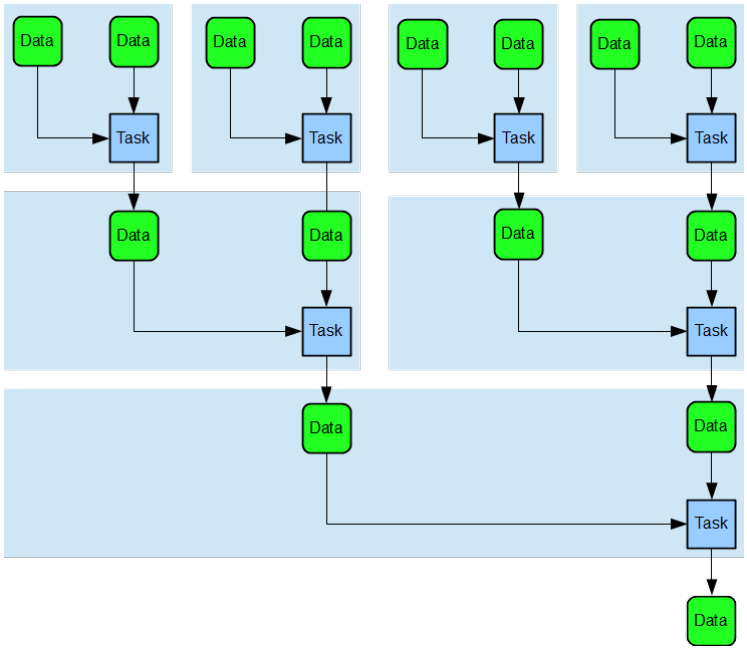
Reduce Using Single-Stage Tiling with Fixed-Tile-Size SPMD

The **reduce** function listed below implements this algorithm. Note that the last tile may have less data elements than the other tiles:

```
// Reduction Pattern - Fixed Tile Size SPMD
// reduce.h

#include <omp.h>

template <typename T, typename C>
T reduce_tile(
    C combine,   // function that performs the operation
    T identity,  // identity value for an empty set
    const T* a,                    // points to the data set
    int n                            // number of elements in the data set
) {

    T accum = identity;
    for (int i = 0; i < n; i++)
        accum = combine(accum, a[i]);
        return accum;
}

template <typename T, typename C>
T reduce(
    C combine,   // function that performs the operation
    T identity,  // identity value for an empty set
    const T* in, // points to the data set
    int n        // number of elements in the data set
) {

    // estimate the number of tiles
    int mtiles = omp_get_max_threads();
    // allocate space for thread reduction vlaues
    T* partial = new T[mtiles];
    omp_set_num_threads(mtiles); // request one thread per tile

    #pragma omp parallel
    {
        int itile = omp_get_thread_num();
        int ntiles = omp_get_num_threads();
        if (itile == 0) mtiles = ntiles; // actual number of tiles
        int tile_size = (n - 1) / ntiles + 1; // actual tile size
        int last_tile = ntiles - 1;
        int last_tile_size = n - last_tile * tile_size;
        partial[itile] = reduce_tile(combine, identity,
            in + itile * tile_size, // starting address
            itile == last_tile ? last_tile_size : tile_size);
    }

    // accumulate partial reduction values
    T accum = reduce_tile(combine, identity, partial, mtiles);

    delete[] partial;
    return accum;
}
```

Multi-Stage Tiling with Strides

The multi-stage tiling algorithm is illustrated in the figure below. At each stage, each tile spans a different number of data elements. The amount data processed is halved from the previous stage while the difference between the data elements that are combined is doubled. This difference is called the *stride* of the specific stage and is a power of 2. Input data that has been processed is no longer required, and intermediate results that are needed for further processing overwrite the previously stored data.



Reduce Using Multi-Stage Tiling with Strides

The first stage on 8 data values produces the first set of 4 intermediate results. The second stage reduces these 4 intermediate results to 2 intermediate results. The third stage reduces these 2 intermediate results to the final result. Synchronization is necessary between adjacent stages to ensure that the data for the next stage has been determined and stored.

Synchronization of threads is an expensive operation.

The **reduce** function below implements the serial version of this algorithm:

```
// Reduction Pattern - Varying Tile Sizes - Serial
// reduce.h

#include <omp.h>

template <typename T, typename C>
T reduce(
    C combine,   // function that performs the operation
    T identity,  // identity value for an empty set
    T* in,       // points to the data set
    int n        // number of elements in the data set
) {

    for (int stride = 1; stride < n; stride *= 2) {
        for (int i = 0; i < n; i += 2 * stride)
            in[2 * stride + i - 1] = combine(in[2 * stride + i - 1],
             in[stride + i - 1]);
    }
    return in[n - 1];
}
```

This tree reduction algorithm provides better precision than the serial algorithm described in the previous sub-section since intermediate results tend to be the same size if original inputs are the same size. We can improve precision further by using larger precision accumulators.

The **reduce** function below implements a work sharing version of this algorithm:

```
// Reduction Pattern - Varying Tile Sizes - Work Sharing
```

```
// reduce.h

#include <omp.h>

template <typename T, typename C>
T reduce(
    C combine,   // function that performs the operation
    T identity,  // identity value for an empty set
    T* in,       // points to the data set
    int n        // number of elements in the data set
) {

    for (int stride = 1; stride < n; stride *= 2) {
        #pragma omp parallel for
        for (int i = 0; i < n; i += 2 * stride)
            in[2 * stride + i - 1] = combine(in[2 * stride + i - 1],
             in[stride + i - 1]);
    }
    return in[n - 1];
}
```

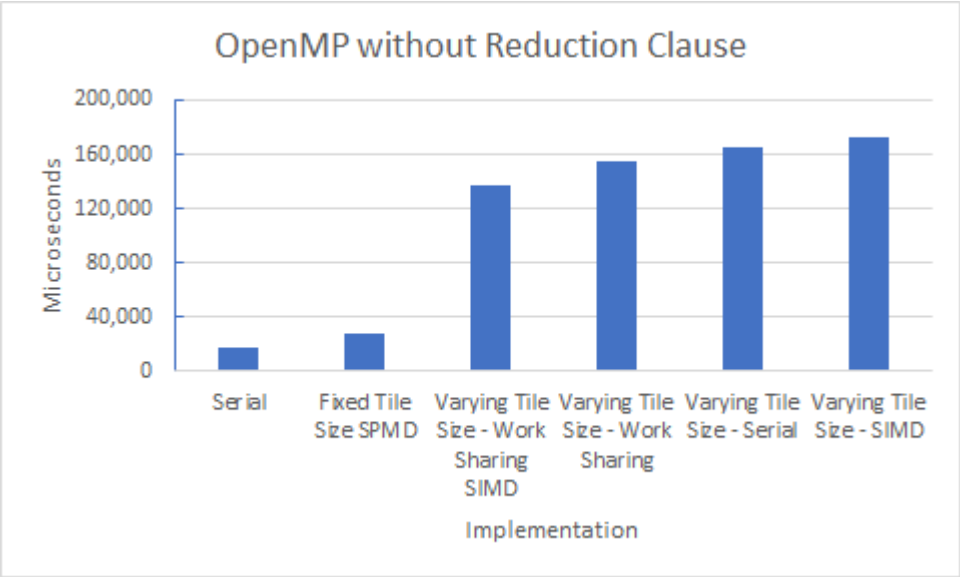
Note that the outermost loop does not meet the requirements for loop-level parallelization, since the change in stride is not iteration invariant. On the other hand, the inner loop complies. There is no loop-carried dependency on the inner loop.

The **reduce** function below adds vectorization:

```
// Reduction Pattern - Varying Tile Sizes - Work Sharing and Vectorization
// reduce.h

#include <omp.h>

template <typename T, typename C>
T reduce(
    C combine,   // function that performs the operation
    T identity,  // identity value for an empty set
    T* in,       // points to the data set
    int n        // number of elements in the data set
) {

    for (int stride = 1; stride < n; stride *= 2) {
        #pragma omp parallel for simd
        for (int i = 0; i < n; i += 2 * stride)
            in[2 * stride + i - 1] = combine(in[2 * stride + i - 1],
             in[stride + i - 1]);
    }
    return in[n - 1];
}
```

Visual Studio Solution

You can find a Visual Studio solution using the Intel 19.1 compiler with O2 optimization on each of these algorithms here. The timing statistics for the different options are shown in the figure below.



OpenMP Implementations of the Reduce Pattern without Reduction Clause

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