

K-Means Clustering

11

The k-means algorithm [Mac67], also called Lloyd’s algorithm [Llo82], is a way of finding clusters in a dataset. It is popular because it is simple to implement. Parallelizing it with the **map** (Chapter 4) and **reduce** (Section 5.1) patterns is straightforward. However, a little cleverness can reduce the number of synchronizations by manipulating the code so that the map and reduce patterns can be fused. The parallel k-means implementation involves a reduction over a “heavy” type. The Cilk Plus implementation illustrates the mechanics of defining reducer hyperobjects for such reductions, when the predefined reducers do not suffice. The TBB implementation shows how to use thread-local storage in TBB.

11.1 ALGORITHM

The standard algorithm for k-means [Mac67, Llo82] starts by creating initial clusters and then iteratively updating them. Each update performs the two steps shown in [Figure 11.1](#):

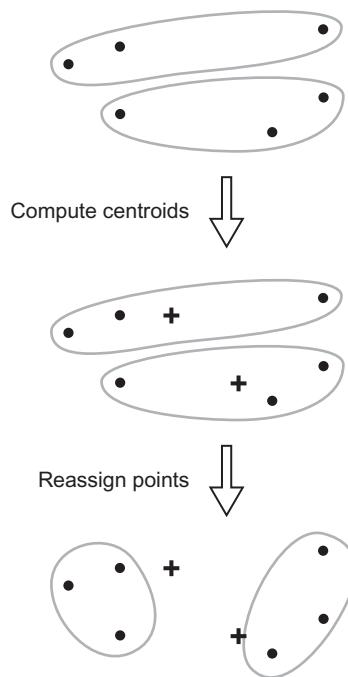
1. Compute the centroid of each cluster.
2. Reassign points to be in the cluster with the closest centroid.

In some situations, a cluster becomes empty. One common repair, which is used in the example, is to assign the point farthest from its current cluster to the empty cluster.

Each of the two steps can be parallelized along two dimensions, across points or across clusters. However, usually the number of points is large enough to provide sufficient **parallel slack** (Section 2.5.6), and sometimes the clusters are few, so only the point dimension is used in the example.

Here are the choices for parallelizing each step in more detail. The centroid of a cluster is the sum of points in it divided by the number of points. Each centroid can be computed separately, and each sum can be computed using parallel reduction. However, if only a few centroids are being computed, computing them independently has several problems:

- The number of clusters might be too few to provide enough parallel slack.
- The partitioning of points into clusters might be grossly imbalanced.
- If the points of a cluster are not kept compact in memory, there may be memory bandwidth issues, because a fetch of a point will bring in its **cache line** with unrelated points.

**FIGURE 11.1**

One iteration of the k-means algorithm with two clusters and six points. Each iteration of the algorithm computes the centroid (+) of each cluster and then reassigns each point to the cluster with the closest centroid.

The example provided here will use only the reduction parallelism approach. Typically, the number of points is much larger than the number of processors and thus provides sufficient parallel slack.

The second step likewise has two possibilities for parallelism. Each point's new cluster can be computed independently, and the distances from it to each centroid can be computed separately. We assume there are enough points to provide parallel slack, and the number of centroids might be small, so the example uses only parallelism across points.

There is another efficiency issue to discuss. The algorithm alternates two patterns over points: reduce to compute centroids and map to reassign points. As discussed in Section 5.2, it is often beneficial to fuse these operations so that only one sweep of memory is required instead of two. Computing a centroid actually involves two steps: computing the sum of its points and then dividing that sum by the size of the cluster. So the iterated steps expand to:

- *Sum*: Compute the sum of the points in each cluster. This is a reduction.
- *Divide*: Divide each cluster sum by the number of points in that cluster.
- *Reassign*: Reassign points to the cluster with the closest centroid. This is a map.

We would like to optimize this using **fusion**. Alas, the reduction and map are in the wrong order to fuse directly, and the divide step separates them. These obstacles are overcome by observing that the k-means algorithm loops over these steps. The first *Sum* step can be peeled from the loop, which permits

rotating the loop, so the algorithm looks like:

```
Sum
do {
    Divide
    Reassign
    Sum
} while( any point moved to another cluster );
```

Now the map pattern for step *Reassign* can be fused with the reduction pattern for step *Sum*. The transformed loop executes *Sum* one more time than the original loop executes it, which adds slight overhead. The savings from fusing greatly outweigh this overhead.

11.2 K-MEANS WITH CILK PLUS

[Listing 11.1](#) shows the top-level Cilk Plus code. Take a look at the overall structure. The input is an array `points` of n points and a value k specifying the number of desired clusters. The routine fills array `centroid[0:k]` with the centroids of the clusters, and fills array `cluster_id[0:n]` with the index of each point's centroid in `centroid[0:k]`. The routine assumes that `distance2(p,q)` returns the square of the Euclidean between points p and q .

The `do-while` loop spanning lines 14 to 34 uses the rotated step structure discussed in the previous section to iterate the steps of *Divide*, *Reassign*, and *Sum*. The `cilk_for` on lines 23 to 33 does the fused map/reduce for the *Reassign* and *Sum* steps. The *Reassign* step is expressed using an array notation operation `__sec_reduce_min_index(x)`, which returns the index of the minimum value in array section x . In the code, x is the result of mapping `distance2` over the section `centroid[0:k]` and scalar value `points[i]`. The mapping of the scalar value treats it as a section of k copies of the value. The array notation here is largely for notational convenience, not **vectorization**. The data is in “**array of structures**” form, so it is unlikely to be profitably vectorized unless the hardware has **gather** instructions. See [Listing 11.7](#) (page 287) for equivalent serial code.

The variable `sum` is a **hyperobject**. In a serial version of the code, it would be an array of `sum_and_count` objects. The j th object is used to record information about the j th cluster, specifically the sum of its points and the number of points. [Listing 11.2](#) shows the definition of `sum_and_count`.

The definition makes fairly minimal assumptions about what a point is. All it assumes is the following:

- `point()` constructs an identity element for point addition.
- $q += p$ sets point q to the sum of points q and p .
- p/n returns a point q such that if n copies of q are added together the sum is p .

The last method, `operator+=`, is not necessary for a sequential k-means algorithm. It is there so the parallel implementation can merge the information in two `sum_and_count` objects.

11.2.1 Hyperobjects

The code uses two hyperobjects:

- `change` counts changes in cluster assignments.
- `sum` accumulates the sum points in each cluster.

```

1 void compute_k_means( size_t n, const point points[], size_t k, cluster_id id[],
2                      point centroid[] ) {
3
4     // Create initial clusters and compute their sums
5     elementwise_reducer<sum_and_count> sum(k);
6     sum.clear();
7     cilk_for( size_t i=0; i<n; ++i ) {
8         id[i] = i % k;
9         // Pealed “Sum step”
10        sum[id[i]].tally(points[i]);
11    }
12
13    // Loop until clusters do not change
14    cilk::reducer_opadd<size_t> change;
15    do {
16        // Repair any empty clusters
17        repair_empty_clusters( n, points, id, k, centroid, sum );
18
19        // “Divide step”: Compute centroids from sums
20        centroid[0:k] = sum.get_array()[0:k].mean();
21
22        sum.clear();
23        change.set_value(0);
24        cilk_for( size_t i=0; i<n; ++i ) {
25            // “Reassign step”: Find index of centroid closest to points[i]
26            cluster_id j = __sec_reduce_min_ind(distance2(centroid[0:k],points[i]));
27            if( j!=id[i] ) {
28                // A different centroid is closer now.
29                id[i] = j;
30                ++change;
31            }
32            // “Sum step”
33            sum[j].tally(points[i]);
34        }
35    } while( change.get_value()!=0 );
36 }
```

LISTING 11.1

K-means clustering in Cilk Plus.

Both `change` and `sum` must be hyperobjects because they are updated by concurrent iterations of the `cilk_for` loop. Object `change` is an instance of the Cilk Plus template class `reducer_opadd` defined in `<cilk/reducer_opadd.h>` and performs the obvious addition reduction.

The other hyperobject `sum` requires more work to implement, because it requires a twist slightly beyond the capabilities of the predefined hyperobjects. If k were a compile-time constant, then

```

1 struct sum_and_count {
2     sum_and_count() : sum(), count(0) {}
3     point sum;
4     size_t count;
5     void clear() {
6         sum = point();
7         count = 0;
8     }
9     void tally( const point& p ) {
10        sum += p;
11        ++count;
12    }
13    point mean() const {
14        return sum/count;
15    }
16    void operator+=( const sum_and_count& other ) {
17        sum += other.sum;
18        count += other.count;
19    }
20 };

```

LISTING 11.2

Type `sum_and_count` for computing mean of points in a cluster.

`reducer_opadd` could be used. We could define a fixed-length vector type, `sum_and_count_vec`, to hold an array of k `sum_and_count` objects, and define `operator+=` on it to do elementwise `operator+=`. Then variable `sum` could be declared as a `reducer_opdd<sum_and_count_vec>` and the program would work.

But k is a runtime value. The construction of the reducer needs to remember k and use it when constructing thread-local views. The solution is to instantiate the predefined template `cilk::reducer` with a **monoid** that remembers k . Listing 11.3 shows a complete implementation.

There are three parts to the reducer:

- A View that implements a view of the hyperobject
- A Monoid that defines operations views
- An elementwise_reducer wrapper around the mechanics that provides a nice public interface

Class `View` has an array of k `sum_and_count` objects. It has responsibility for construction and destruction of the array. Because k is the same for all views, the code makes it part of the `Monoid`, not the `View`.

Class `Monoid` defines operations on views. Its base class defines defaults for the signatures that `cilk::reducer` expects of a monoid. For example, the `Monoid` uses the defaults for allocating/deallocating memory for a view and destroying a view. The `Monoid` overrides the default for constructing

```

1 template<typename T>
2 class elementwise_reducer {
3     struct View {
4         T* array;
5         View( size_t k ) : array( new T[k] ) {}
6         ~View() {delete[] array;}
7     };
8
9     struct Monoid: cilk::monoid_base<View> {
10         const size_t k;
11         void identity(View* p) const {new(p) View(k);}
12         void reduce(View* left, View* right) const {
13             left->array[0:k] += right->array[0:k];
14         }
15         Monoid( size_t k_ ) : k(k_) {}
16     };
17     cilk::reducer<Monoid> impl;
18 public:
19     elementwise_reducer( size_t k ) : impl(Monoid(k), k) {}
20     void clear() {impl.view().array[0:impl.monoid().k].clear();}
21     T* get_array() {return impl.view().array;}
22     operator sum_and_count*() {return get_array();}
23 };

```

LISTING 11.3

Defining a hyperobject for summing an array elementwise in Cilk Plus. The names of each struct can be changed. The names of methods `identity` and `reduce` must not be changed, because they are used internally by `cilk::reducer<Monoid>`.

a view initialized to an identity value, because it needs to pass k to the view constructor. The `Monoid` also specifies how to reduce two views. As `cilk::reducer` requires, our method puts the reduction into the left view. Note that k comes from the monoid, not the view. The array notation there provides brevity.

Class `elementwise_reducer` has the actual reducer, as member `elementwise_reducer::impl`. The public methods provide a nice public interface. For example, method `clear`, which applies method `clear()` elementwise to each element in the view, hides the use of the two key methods on `impl`:

- `view()` returns a reference to the current view.
- `monoid()` returns a reference to the monoid.

The constructor for `impl` takes two arguments. The first is the `Monoid`. The second is the argument with which to construct the leftmost view. In the example, this argument is k , so that the reducer will construct the leftmost view as `View(k)`.

11.3 K-MEANS WITH TBB

The k-means algorithm has the same basic structure in TBB as it does in Cilk Plus. The primary differences are:

- The array notation statements are written out as serial loops. Since the array notation was used for brevity, not for vectorization, the loss is only of notational convenience, not performance.
- The hyperobjects are replaced by thread-local storage. Merging of local views into a global view is done with an explicit loop.
- **Tiling** of iterations in the parallel loops is explicit, so that thread-local lookup can be done once per tile.

[Listing 11.4](#) shows the declarations for a type `tls_type` that will hold thread-local views of the `sum_and_count` from [Listing 11.2](#) (page 283). The thread-local storage is implemented using an instance of template `tbb::enumerable_thread_specific`, which implements a collection of thread-local views. The expression `tls.local()` returns the thread-local view for the calling thread. If such a view does not yet exist, the method creates one.

The way a new view is created depends upon how the `enumerable_thread_specific` was constructed. There are three ways, as shown in the following fragment:

```
enumerable_thread_specific<T> a;
enumerable_thread_specific<T> b(x); // x assumed to have type T
enumerable_thread_specific<T> c(f); // f assumed to be a functor
```

Local views for `a` will be default-constructed. Local views for `b` will be copy-constructed from exemplar `x`, which must be of type `T`. Local views for `c` will be constructed using `T(f())`.

Our example uses the last way, where `T` is a `view`. The constructor for `view` expects `k` as an argument. In the final code, `k` is a local variable in the surrounding context. So the code will use a lambda expression as `f` when declaring an instance of `tls_type`, like this:

```
tls_type tls([&]{return k;});

1 class view {
2     view( const view& v );           // Deny copy construction
3     void operator=( const view& v ); // Deny assignment
4 public:
5     sum_and_count* array;
6     size_t change;
7     view( size_t k ) : array(new sum_and_count[k]), change(0) {}
8     ~view() {delete[] array;}
9 };
10
11 typedef tbb::enumerable_thread_specific<view> tls_type;
```

LISTING 11.4

Declaring a type `tls_type` for thread-local views in TBB.

A thread can access all of the views by using STL conventions, because a `enumerable_thread_specific` acts like a STL container of views.

For example, Listing 11.5 accumulates the sum of `change` in all views into a global view and resets the local values. Our k-means example will rely on that code and the similar code in Listing 11.6 that accumulates sums of points. For compilers not supporting C++11 `auto` declarations (Section D.1), replace the `auto` with `tls_type::iterator`.

Using thread local storage for reductions has two limitations compared to the Cilk Plus reducers:

- The reduction operation must be **commutative** as well as **associative**. Reducer hyperobjects merely require associativity.
- Using a serial loop for reducing the local views can become a scalability bottleneck, since its span is inherently $\Omega(P)$.

The latter limitation can be addressed by using TBB's `parallel_reduce` template to reduce the local views if the span becomes an issue. For our example, it is not worth the trouble as long as there are many more points than hardware threads, because then the reduction of local views is a small contributor to the total running time.

```

1 void reduce_local_counts_to_global_count( tls_type& tls, view& global ) {
2     global.change = 0;
3     for( auto i=tls.begin(); i!=tls.end(); ++i ) {
4         view& v = *i;
5         global.change += v.change;
6         v.change = 0;
7     }
8 }
```

LISTING 11.5

Walking local views to detect changes. The variable `tls` is a `tbb::enumerable_thread_specific<view>`, by way of the typedef in Listing 11.4.

```

1 void reduce_local_sums_to_global_sum( size_t k, tls_type& tls, view& global ) {
2     for( auto i=tls.begin(); i!=tls.end(); ++i ) {
3         view& v = *i;
4         for( size_t j=0; j<k; ++j ) {
5             global.array[j] += v.array[j];
6             v.array[j].clear();
7         }
8     }
9 }
```

LISTING 11.6

Walking local views to accumulate a global sum. Each local view is cleared in preparation for the next iteration of the k-means algorithm.

Another feature of TBB used in the example that deserves comment is the explicitly tiled form of `tbb::parallel_for`. The following pattern will be used to iterate over all n points in parallel:

```
tbb::parallel_for(
    blocked_range<size_t>(0,n),
    [...]( tbb::blocked_range<size_t> r ) {
        view& v = tls.local();
        for( size_t i=r.begin(); i!=r.end(); ++i ) {
            ...process point i...
        }
    }
);
```

The first argument to `parallel_for` specifies a range over which to iterate. The `parallel_for` splits that range into subranges and applies the functor argument to each subrange. In the pattern, the functor looks up its thread-local view and then processes each point in the subrange. The loop could be written more concisely as:

```
tbb::parallel_for( 0, n,
    [...]( size_t i ) {
        view& v = tls.local();
        ...process point i...
    }
);
```

but at the cost of executing `tls.local()` for each point. The explicitly tiled form of `parallel_for` should be used when there is a profitable opportunity to optimize the inner loop.

TBB does not have reduction operators like `__sec_reduce_min_ind` in Cilk Plus, so the code will use the auxiliary routine `reduce_min_ind` in Listing 11.7. Given that and the previous discussion, the rest of k-means is straightforward to write. Listing 11.8 shows the TBB code.

```
1 int reduce_min_ind( const point centroid[], size_t k, point value ) {
2     int min = -1;
3     float mind = std::numeric_limits<float>::max();
4     for( int j=0; j<k; ++j ) {
5         float d = distance2(centroid[j],value);
6         if( d<mind ) {
7             mind = d;
8             min = j;
9         }
10    }
11    return min;
12 }
```

LISTING 11.7

Routine for finding index of centroid closest to a given point. This routine is a serial equivalent of the `__sec_reduce_min_ind` expression on line 25 in Listing 11.1.

```

1 void compute_k_means( size_t n, const point points[], size_t k, cluster_id id[],
2   point centroid[] ) {
3
4   tls_type tls([&]{return k;});
5   view global(k);
6
7   // Create initial clusters and compute their sums
8   tbb::parallel_for(
9     tbb::blocked_range<size_t>(0,n),
10    [=,&tls,&global]( tbb::blocked_range<size_t> r ) {
11      view& v = tls.local();
12      for( size_t i=r.begin(); i!=r.end(); ++i ) {
13        id[i] = i % k;
14        // Peeled "Sum step"
15        v.array[id[i]].tally(points[i]);
16      }
17    );
18
19   // Loop until ids do not change
20   size_t change;
21   do {
22     // Reduce local sums to global sum
23     reduce_local_sums_to_global_sum( k, tls, global );
24
25     // Repair any empty clusters
26     repair_empty_clusters( n, points, id, k, centroid, global.array );
27
28     // "Divide step": Compute centroids from global sums
29     for( size_t j=0; j<k; ++j ) {
30       centroid[j] = global.array[j].mean();
31       global.array[j].clear();
32     }
33
34     // Compute new clusters and their local sums
35     tbb::parallel_for(
36       tbb::blocked_range<size_t>(0,n),
37       [=,&tls,&global]( tbb::blocked_range<size_t> r ) {
38         view& v = tls.local();
39         for( size_t i=r.begin(); i!=r.end(); ++i ) {
40           // "Reassign step": Find index of centroid closest to points[i]
41           cluster_id j = reduce_min_ind(centroid, k , points[i]);
42           if( j!=id[i] ) {
43             id[i] = j;
44             ++v.change;
45           }
46           // "Sum step"

```

```
47             v.array[j].tally(points[i]);
48         }
49     }
50 );
51
52     // Reduce local counts to global count
53     reduce_local_counts_to_global_count( tls, global );
54 } while( global.change!=0 );
55 }
```

LISTING 11.8

K-Means clustering in TBB.

11.4 SUMMARY

The k-means algorithm is a serial loop that iterates two fundamentally parallel steps: computing centroids and reassigning points, until it converges. By peeling part of the first iteration, the two parallel steps can become a single parallel sweep: map fused with reduction. The reduction type is somewhat heavy in the sense that it is bigger and takes more time to copy than a scalar type. The Cilk Plus implementation shows how to define custom reducer hyperobjects for performing reductions when a suitable built-in reducer is not available. The TBB implementation uses thread-local storage for the reduction, which works when the reduction operation is both commutative and associative.