Parallel algorithms

Jump to bottom

Amanieu d'Antras edited this page on Feb 6, 2015 · 3 revisions

Async++ provides several highly-optimized functions that allow you to efficiently perform operations in parallel. All of these functions take an optional scheduler as their first parameter, but keep in mind that part of the work will be done on the calling thread.

Parallel invoke

The `parallel_invoke` function takes a set of function objects which are all executed in parallel.

Example:

```
async::parallel_invoke([] {
    std::cout << "This is executed in parallel..." << std::endl;
}, [] {
    std::cout << "with this" << std::endl;
});
```

Parallel for

The `parallel_for` function takes a range and a function to execute for each element in the range. The range is split into chunks which are then processed in parallel.

Example:

```
async::parallel_for({0, 1, 2, 3, 4}, [](int x) {
        std::cout << x;
});
std::cout << std::endl;
```

Parallel reduce

The `parallel_reduce` function takes a range, an identity value and a reduction function. The reduction function is used to combine all elements of the range into a single element. As with `parallel_for`, the range is split into chunks and processed in parallel.

The `parallel_map_reduce` function is identical to `parallel_reduce` except that it accepts an extra function that is applied to every element before the reduction.

Example:

```
int r = async::parallel_reduce({1, 2, 3, 4}, 0, [](int x, int y) {
    return x + y;
});
std::cout << "The sum of {1, 2, 3, 4} is " << r << std::endl;
int r = async::parallel_map_reduce({1, 2, 3, 4}, 0, [](int x) {
    return x * 2;
}, [](int x, int y) {
    return x + y;
});
std::cout << "The sum of {1, 2, 3, 4} with each element doubled is " << r
          << std::endl;
```

Ranges

The parallel algorithms only accept *range objects*, which have `begin()` and `end()` member functions that return iterators. Most C++ containers, as well as initializer lists and fixed-length arrays can be used directly as ranges with no modifications. To handles other cases, Async++ provides two adapters to make working with ranges easier:

- The `irange()` function returns an `int_range<T>` object which represents a sequence of consecutive integers.
- The `make_range()` function takes a pair of iterators and returns a `range<T>` object which wraps the two iterators as a range.

Example:

```
async::parallel_for(async::irange(0, 5), [](int x) {
    std::cout << x;
});
std::cout << std::endl;
```

```
int* numbers = get_numbers();
size_t num_numbers = get_num_numbers();
async::parallel_for(async::make_range(numbers, numbers + num_numbers),
                    [](int x) {
    std::cout << x;
});
std::cout << std::endl;
```

Partitioners

Async++ controls how ranges are split using *partitioners*. Partitioners are simply ranges with an additional `split()` member function. This function should split the range into two halves, return a new partitioner containing one half and modify the current partitioner to only contain the other half. If `split()` return a partitioner with an empty range, this is interpreted to mean that all remaining elements should be processed in the current thread.

Async++ provides two partitioner implementations:

- `static_partitioner()` : This takes a range and an optional grain size, and splits the range into halves while the chunks are larger than the grain size. If a grain size is not specified then it default to the size of the range divided by 8 times the number of CPUs in the system.
- `auto_partitioner()` : This takes a range and splits it only if it is running in a different thread than it was previously. This works with the work-stealing thread pool scheduler by avoiding the creation of excessive chunks when other thread are busy with other work.

`parallel_for` , `parallel_reduce` and `parallel_map_reduce` all accept partitioners in the place of ranges to customize the behavior of the algorithm. This is done through the `to_partitioner` utility function which return the parameter if it is already a partitioner, but returns an `auto_partitioner` for ranges that aren't partitioners.

Example:

```
// Split the range into chunks of 8 elements or less
async::parallel_for(static_partitioner(async::irange(0, 1024), 8),
                    [](int x) {
        std::cout << x;
});
std::cout << std::endl;
```

▾ Pages 6