BAM!
"Simplify your Parallelism"

Stephan Dollberg

May 1, 2012

1 parallel constructs

bam::parallel constructs offer simplistic parallel replacements for standard library algorithms like STD::FOR_EACH.

All Parallel-Algorithms share a typical interface. They all take a range, a worker function and a grainsize parameter. Work is split into pieces and worked on by a limited count of threads which is determined on runtime. Task stealing is performed when a thread runs out of work. When no grainsize parameter is passed, the default value is 0, which means that implementation will choose a grainsize on runtime.

1.1 parallel_for

The interface looks like this:

```
template<typename ra_iter, typename worker_predicate>
void parallel_for(ra_iter begin, ra_iter end, worker_predicate worker, int
    grainsize = 0)
```

`bam::parallel_for` is a template which is the parallel replacement for a basic for-loop. Its use is pretty simple and straight forward, the following example should make it quite clear.

1.1.1 Example 1:

```
  typedef std::vector<int> container;
  typedef container::iterator iter;

  container v = {1, 2, 3, 4, 5, 6};

  auto worker = [] (iter begin, iter end) {
    for(auto it = begin; it != end; ++it) {
      *it = compute(*it);
    }
```

```
  };

  bam::parallel_for(std::begin(v), std::end(v), worker);
```

The snippet shows how to parallelize a simple compute operation which modifies each element in a vector. All that is needed is a worker lambda which takes a range then performs `compute` on every element in that range.

1.2 parallel_for_each

```
template<typename ra_iter, typename worker_predicate>
void parallel_for_each(ra_iter begin, ra_iter end, worker_predicate worker,
     int grainsize = 0)
```

`bam::parallel_for_each` is the replacement for `std::for_each`. It functions the very same as the original does and basically all you have to do is to replace the name. The following example shows a simple use case.

1.2.1 Example 1: Compute

```
  typedef std::vector<int> container;
  typedef container::iterator iter;

  container v = {1, 2, 3, 4, 5, 6};

  auto worker = [] (int& i) { i = compute(i); };
  bam::parallel_for_each(std::begin(v), std::end(v), worker);
```

Taking up the `bam::parallel_for` example and making it yet even easier, we basically just wrote a typically easy `std::for_each` function predicate and all we had to do was to replace the namespace identifier.

1.3 parallel_reduce

`bam::parallel_reduce` is doing simple data parallel tasks but unlike bam::parallel_for it does reduce operations and returns a value.

The interface is definied as followed:

```
template<
typename return_type,
typename ra_iter,
typename predicate,
typename join_predicate,
>
return_type parallel_reduce(ra_iter begin, ra_iter end, predicate worker,
    join_predicate join_worker, int grainsize = 0)
```

The interface is quite big but mighty, the following examples will explain different use-cases and the several arguments.

begin, end	begin and end form the range on which `bam::parallel_reduce` works on
worker	worker is the predicate which takes a range and does the work on it
join_worker	join predicate which joins the results of the two binary splitted parts
grainsize	sets the grainsize parameter for the minimal range of splitting

1.3.1 Example 1: Accumulate

```
typedef std::vector<int> container;
typedef container::iterator iter;

container v = {1, 2, 3, 4, 5, 6};

using namespace std::placeholders;
std::function<int(iter, iter)> accumulate_wrapper =
    std::bind(std::accumulate<iter, int>, _1, _2, 0);

int result = bam::parallel_reduce<int>(std::begin(v), std::end(v),
    accumulate_wrapper, std::plus<int>());

std::cout << result << std::endl; // prints 21
```

This example shows the very basics when using `bam::parallel_reduce`. At first we are creating a vector whose elements we want to accumulate in parallel. Then we define a wrapper, which calls `std::accumulate` as we can't just pass `std::accumulate` to our algorithm as that one can't provide the third parameter to `std::accumulate` which is needed for type deduction. In our call to `bam::parallel_reduce` we do explicitly name the return type as, that can't be deduced from the template arguments. We then pass our range in form of the iterators, our function to work on, the accumulate wrapper, the `std::plus<int>` functor which is the needed join function to combine the splitted parts which the different threads are handling.

1.3.2 Example 2: Find Max

```
typedef std::vector<int> container;
typedef container::iterator iter;

container v = {1, 2, 3, 4, 5, 6};

auto join_max_helper = [] (iter a, iter b) -> iter {
    return *a > *b ? a : b;
```

```
};

iter result = bam::parallel_reduce<iter>(std::begin(v), std::end(v), std
    ::max_element<iter>, join_max_helper);

std::cout << *result << std::endl; // prints 6
```

In this example we want to find the biggest element in a given range. Therefore we use the new C++11 standard function `std::max_element` as a worker function, however this time we have to provide a custom join function which returns the iterator with the biggest associated element of the two given iterators.