Submit New Art

When, and when not, to use the Intel ArBB _for loops

Intel® Array Building Blocks (Intel® ArBB) provides loop constructs such as **_for**, **_while**, and **_do/_until**. Th are similar to the **for**, **while**, and **do/while** loops in C/C++. Users sometimes get confused as to when to use the ArBB loops and when to use the regular C/C++ loops. Users who are familiar with the "parallel for" loops in OpenMP* and Intel® Threading Building Blocks may also have some incorrect assumptions about Intel ArBB' loop constructs, especially about the **_for** loops. This article uses the **_for** loops as an example to illustrate the proper usage of Intel ArBB loop constructs.

You cannot use _for loops for parallel execution.

The most important thing to note is that **_for** loops (and all other loop constructs in Intel ArBB) are regular seri loops. Iterations of a **_for** loop are executed sequentially. Intel ArBB does not auto-parallelize **_for** loops, nor it provide a "parallel for" loop construct. You cannot use **_for** loops to express data parallelism. The correct wa express data parallelism in Intel ArBB is to use container operations and/or the **arbb::map()** function.

As a simple example, consider the problem of multiplying a matrix with a vector. Here is an Intel ArBB implementation that works but does not have any parallelism:

- collapse sourceview plaincopy to clipboardprint?

```
void matvec_product(const dense<f32, 2>& matrix, const dense<f32>& vector, dense<f32>& result)
{
    usize rows = matrix.num_rows();
    usize cols = matrix.num_cols();
    _for (usize i = 0, i < rows, ++i) {  // SERIAL LOOP
        f32 sum(0.0);
        _for (usize j = 0, j < cols, ++j) {  // SERIAL LOOP
            sum += matrix(j, i) * vector[j];
        } _end_for
        result = replace(result, i, sum);
    } _end_for
}

```

Except for the use of Intel ArBB types, operators and keywords, this code looks very similar to the serial C implementation. However, just like the C version, it runs sequentially. This is not the right way to compose an efficient parallel program using Intel ArBB.

A much better implementation that shows the simplicity of Intel ArBB syntax and expresses the intrinsic parall
nature of the algorithm looks like this:

- collapse sourceview plaincopy to clipboardprint?

```
void matvec_product(const dense<f32, 2>& matrix, const dense<f32>& vector, dense<f32>& result)
{
    result = add_reduce(matrix * repeat_row(vector, matrix.num_rows()));
}

```

Notice the use of container operators instead of scalar operators, as well as the use of collective operators. Not
is this code simpler, it also allows the Intel ArBB runtime to parallelize the computation through vectorization and/or multithreading. It is also possible to express this algorithm using an **arbb::map()** function. To learn mor
about how to express parallelism using Intel ArBB containers, container operators, and the arbb::map() functio
refer to this and this sections in the Intel ArBB User's Guide. Also, see the tutorials for more code samples.

When should you use **_for** loops?

The **_for** loop should be used in the following situations:

- Inside Intel ArBB functions.
- To express serially dependent iterative computation. This is the case where a computation must be done incrementally, with the current step depending on the result of the previous step. A good example would heat dissipation using an iterative stencil:

  - collapse sourceview plaincopy to clipboardprint?

  ```
  void apply_stencil(dense<f64, 2>& grid, i32 iterations) {
      _for(i32 i = 0, i < iterations, ++i) {
          map(stencil)(grid);
      } _end_for
  }

  void stencil(f64& cell) {
      arbb::array<usize, 2> coord;
      position(coord);
      usize x = coord[0], usize y = coord[1];
      _if(x != 0 && y != 0 && x != WIDTH-1 && y != HEIGHT-1) {
          cell = 0.25 * (neighbor(cell, -1, 0) + neighbor(cell, 1, 0) +
   neighbor(cell, 0, -1) + neighbor(cell, 0, 1));
      } _end_if
  }

  ```

  In this code, computing each stencil-based update step is parallelized through the use of the **arbb::map()**
  function. But the updating must be done multiple times repetitively in a sequence in order to compute th
  solution over time.

When should you use regular C/C++ **for** loops inside Intel ArBB code?

***Trademarks**

We have been so far concentrating on the usage of **_for** loops. Some users may now be wondering, why not jus the regular C/C++ **for** loops inside Intel ArBB function to control repetitive execution? Is using regular C/C++ loops inside Intel ArBB functions permitted?

The answer is it is legal to use a regular **for** loop inside Intel ArBB code. In fact, it is often very useful, but it executes at *capture* time, not at run time. Remember a regular **for** statement can only involve C/C++ types, sin is a regular C/C++ statement. Such C/C++ statements inside Intel ArBB code also only get evaluated once at *capture* time. Then its effects, which are frozen at the point of capture, are carried over to all subsequent Intel ArBB executions. For a regular **for** loop whose body contains Intel ArBB statements, its effect is to unroll the l body by however many times it ran at *capture* time. This feature makes the regular **for** loop useful in creating different computation specializations (i.e. different versions of a same Intel ArBB function with different unroll factors). See the example below:

- collapse sourceview plaincopy to clipboardprint?

```
// A C/C++ int type
int unroll_factor;

// An Intel ArBB function
void kernel(f32& a)
{
    for (int i = 0; i < unroll_factor; ++i) {
        stmt1(a);
        stmt2(a);
    }
}

int main()
{
    // Create a closure that unrolls 4 times
    unroll_factor = 4;
    closure<void(f32&)> unroll_4 = capture(kernel);

    // Create a closure that unrolls 8 times
    unroll_factor = 8;
    closure<void(f32&)> unroll_8 = capture(kernel);

    ......

}

```

One more thing ...

It is incorrect to write a **_for** loop like this:

- collapse sourceview plaincopy to clipboardprint?

```
int i;
_for (i = 0, i < n, ++i) {  // WRONG!

    ......

} _end_for
```

To understand why this is wrong, consider the loop condition statement (i < n). Because the induction variable C/C++ type, this loop condition statement returns a regular C/C++ **bool** type. However, the **_for** loop works w only Intel ArBB types and it expects an **arbb::boolean** type here. The problem is actually more than type mismatching. Consider the loop step statement (++i). This statement is C/C++ code rather than Intel ArBB cod because it involves only a C/C++ type. Remember that a C/C++ statement inside Intel ArBB code gets execute the *capture* time and its effects are "baked in" and won't change during the Intel ArBB execution. What this me here is that the induction variable **i** is only incremented once and then stays unchanged! In fact, as far as Intel is concerned, the arguments to the **_for** loop are empty. Intel ArBB can only capture computations expressed w Intel ArBB types. To understand more about Intel ArBB's *capture* concept and the related *closure* concept, rea other two KB articles, here and here.

(Note: This should be an error at compile time. However, in the current beta version of Intel ArBB, this error is caught and no warning is issued. In fact, it triggers a bug in the ArBB beta that leads to a crash at O2 and O3, although it works at O0. However, in summary, you should always use ArBB types as induction variables in _f loops.)

Do you need more help?
Click tags links for related articles Search Knowledge Base Visit User Forums Get other Support options

This article applies to: Intel® Array Building Blocks Knowledge Base