Implementing Data-Parallel Patterns for Shared Memory with OpenMP

Michael Suess and Claudia Leopold

University of Kassel, Research Group Programming Languages / Methodologies,
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
E-mail: {msuess, leopold}@uni-kassel.de

AthenaMP[1] is an open-source parallel pattern library in C++ and OpenMP that is currently under development. This paper reports on the implementations of several data-parallel patterns: `modify_each`, `transmute`, `combine`, `reduce`, and `filter`. We describe the patterns as well as our experiences implementing them in C++ and OpenMP. The results of two benchmarks showing no performance losses when compared to a pure OpenMP implementation are also included.

1 Introduction

With the advent of multi-core processors, parallel programming for shared memory architectures is entering the mainstream. However, parallel programming in general is considered difficult. One solution to this problem lays in the use of libraries that hide parallelism from the programmer. This has been practised in the field of numerics for a long time, where premium quality libraries that encapsulate parallelism are available. For the more general field of programming, design patterns are considered a good way to enable programmers to cope with the difficulties of parallel programming[2].

Although there are a variety of libraries available both commercially and in a research stage, what is missing from the picture is a pattern library in OpenMP. Since its introduction in 1997, OpenMP has steadily grown in popularity, especially among beginners to parallel programming. Its ease of use is unmatched by any other system based on C/C++ or Fortran available today, while at the same time offering an at least acceptable performance. Therefore, design patterns implemented in OpenMP should be a viable learning aid to beginners in the field of concurrent programming. At the same time, they are able to encapsulate parallelism and hide it from the programmer if required. For this reason, the AthenaMP project[1] was created that implements various parallel programming patterns in C++ and OpenMP.

The main goal of AthenaMP is to provide implementations for a set of concurrent patterns, both low-level patterns like advanced locks (not described here), and higher-level patterns like the data-parallel patterns described in this paper or task-parallel patterns like taskpools or pipelines. The patterns demonstrate solutions to parallel programming problems, as a reference for programmers, and additionally can be used directly as generic components. Because of the open source nature of the project, it is even possible to tailor the functions in AthenaMP to the need of the programmer. The code is also useful for compiler vendors testing their OpenMP implementations against more involved C++-code. A more extensive project description is provided by one of the authors in his weblog[a].

[a] http://www.thinkingparallel.com/2006/11/03/a-vision-for-an-openmp-pattern-library-in-c/

Another contribution of this paper is a description of implementation problems that we faced while developing our patterns: the insufficient state of the compilers with regard to OpenMP and C++, as well as the restricted ability to influence the scheduling of parallel loops at runtime.

Sect. 2 starts with a general introduction to the features of our patterns/generic components and goes on to highlight each one of them in detail. The performance of our solutions is described in Sect. 3 using two benchmarks. Some implementation details and problems with current compilers are described in Sect. 4. Sect. 5 shows related work, and Sect. 6 closes the paper with a short summary.

2 Data-Parallel Patterns in Detail

This section introduces the data-parallel patterns contained in AthenaMP to date. First, we explain some features of our implementation common to all of them.

The interfaces of our functions realizing the patterns have been designed using the Standard Template Library (STL) as an example where applicable. Just like in the STL, all functions presented here are polymorphic through the use of templates. It is possible to mix the library calls freely with normal OpenMP code. This is useful e.g. for employing nested parallelism, where the user specifies the first level of parallelism in his own code and the second level is opened up by the library.

All functions take two or more iterators specifying one or more ranges of elements as parameters, as is customary in the STL. Behind the scenes, depending on the iterators supplied, there are two versions of the patterns: one that takes random-access iterators (as supplied by e.g. std::vectors or std::deques), and one that takes forward or bidirectional iterators (as supplied by e.g. std::lists). Using template metaprogramming (iterator traits), the right version is selected transparently to the user by the compiler.

The first version makes use of the `schedule`-clause supplied with OpenMP. Although it is set to static scheduling, this is easily changeable in the source of the library. When the functor supplied takes a different amount of time for different input values, using a dynamic schedule is a good idea. This random-access version is fast and has easy to understand internals, because it uses OpenMP's `parallel for` construct.

The second version is for forward iterators. Use of the scheduling functionality supplied by OpenMP would be very costly in that case, as to get to a specific position in the container, $O(n)$ operations are necessary in the worst case. This operation would need to be carried out n times, resulting in a complexity of $O(n^2)$. For this reason, we have implemented our own static scheduling. The internal `schedule_iter` template function is responsible for it. It takes references to two iterators. One of them marks the start and the other the end of the range to be processed. The function calculates the number of elements in the range, divides it among the threads, and changes the iterators supplied to point to the start and the end of the range to be processed by the calling thread (by invoking `std::advance`). This results in a total complexity of $O(n * t)$, where t is the number of threads involved. Iff each thread in a team calls the function and works on the iterators returned, the whole range is processed with only a few explicit calls to `std::advance`. However, the forward iterator version is slower and less flexible (because changing the scheduling policy is not possible with this version) than the random access iterator version, as can be observed in Sec. 3.

The parallelism inherent in the patterns described is totally hidden from the user, except of course when looking into the AthenaMP source code. This does not necessarily lead to smaller sources (as a lot of scaffolding code for the new functors is required), but to less error prone ones. A lot of mistakes are common when working with OpenMP[3] (or any other parallel programming system), and these can be reduced by deploying the patterns.

The last parameter to each function is the number of threads to use in the computation. If the parameter is left out, it defaults to the number of threads as normally calculated by OpenMP. Unless otherwise specified in the description of the pattern, the original order of the elements in the range is preserved and no critical sections of any kind are required.

It is also possible to nest patterns inside each other, e.g. by calling `reduce` inside a functor supplied to `modify_each` for two-dimensional containers. Nested parallelism is employed in this case, which becomes especially useful if you have a large number of processors to keep busy.

The following patterns and their implementations are discussed in the next sections:

- `modify_each`: also commonly known as *map*, modifies each element supplied
- `transmute`: also known as *transform* applies a function to each element provided and returns a new range containing the results of the operation
- `combine`: combines elements from two sources using a binary function and returns a new range containing the results of the operation
- `reduce`: also known as *fold*, combines all elements into a single result using an associative operation
- `filter`: filters the elements supplied according to a predicate function and returns the results using a new container

2.1 modify_each

The `modify_each` pattern provides a higher-order template function to apply a user-supplied functor to each element in a range in parallel. No provisions are made to protect any internal data in the functor or any side-effects of the functor from concurrent access. Fig. 1 shows an example, where the pattern is used to add ten to all elements in an std::vector. The user is relatively free with regards to the functor supplied, as long as it contains an `operator()` method that is a unary function and takes a non-const reference as argument. If `operator()` has a return value, it is ignored.

2.2 transmute

The `transmute` pattern is similar to `modify_each` in the way that it applies a user-supplied unary functor to all elements in a range in parallel. While `modify_each` works on the original elements and modifies them, `transmute` stores its results in a different range and is therefore even able to change the type of each element. It is known in the STL as `transform` (but similar to many of our patterns, we could not use this name to avoid name-clashes with the STL-version).

```
/** a user-supplied functor that adds diff to its argument */
class add_func : public std::unary_function<int, void> {
  public:
    add_func (const int diff) : diff_ (diff) {}
    void operator() (int& value) const { value += diff_; }
  private:
    const int diff_;
};

/* add 10 to all targets in place. */
athenamp::modify_each (targets_.begin(), targets_.end(),
    add_func (10));
```

Figure 1. `modify_each` in action

```
/* combine all elements from sources1_ with all elements from
 * sources2_ and store the results in sources1_ */
athenamp::combine (sources1_.begin(), sources1_.end(),
    sources2_.begin(), sources1_.begin(), std::plus<int>());
```

Figure 2. `combine` in action

The list of parameters it takes is similar to `modify_each` again, except for the fact that it takes an additional iterator that points to the location where the results are to be stored. The user is responsible for making sure that there is enough room to store all results. The user-supplied functor is similar to the one supplied for `modify_each` again, except for the fact that it cannot work on its argument directly, but returns the result of its computation instead. The result is then put into the appropriate location by our library function `transmute`.

No provisions are made to protect any internal data in the functor or any side-effects of the functor from concurrent access. While it is possible and correct to use `transmute` to apply changes inplace by overlapping its supplied ranges, it is recommended to use `modify_each` in that case because it is faster, since it involves less copying of elements.

An example is omitted here to keep the paper short and because of the similarity of this method to the already explained `modify_each` pattern.

2.3 combine

`combine` is a relatively simple pattern that is used to combine elements from two different ranges using a binary operation and put the result into a third range. It is similar to the `transmute` pattern explained above, except that it works on two ranges instead of one. Similar restrictions as for `transmute` also apply here: the user is responsible for making sure there is enough room to put the results in and the internals of the functor are also not protected from concurrent access. It is also possible to store the results inplace, as shown in Fig. 2. What is also shown in this figure is that it is possible to use the functors provided by the STL for this pattern (`std::plus` in this example). If this is the case, many lines of code can be saved when compared to the parallel version without patterns.

```
/** A functor that can be used to find the maximum and sum of all
 * elements in a range by repeatedly applying operation() to all
 * elements in the range. */
class max_sum_functor : public std::unary_function<int, void> {
  public:
    max_sum_functor (int max, int sum) : max_(max), sum_(sum) {};
    int max () const { return max_; }
    int sum () const { return sum_; }

    void operator() (const int arg1)
    {
        max_ = std::max (arg1, max_);
        sum_ += arg1;
    }

    void combine (const max_sum_functor& func)
    {
        max_ = std::max (func.max (), max_);
        sum_ += func.sum ();
    }

  private:
    int max_;
    int sum_;
};

max_sum_functor func (-1, 0);
/* calculate maximum and sum of all elements in vector targets_ */
athenamp::reduce (targets_.begin(), targets_.end(), func);
/* check result */
std::cout << func.max() << std::endl << func.sum() << std::endl;
```

Figure 3. `reduce` in action

2.4 reduce

The `reduce` pattern combines all elements in a given range into a single result using a binary, associative operation. It is also commonly known as *fold* or *for_each*. Many parallel programming systems feature a reduce-operation, among them OpenMP. The reduce-operation in OpenMP has a disadvantage, though: it is limited to a few simple, predefined operators, such as $+$ or $*$. These operators can only be applied to a few data-types, such as `int`'s.

Our `reduce`-method solves these problems, as a user-defined functor is specified as operator. This functor can work on any data-type. Multiple reductions in a single pass are possible as well, with a functor that does two or more operations at the same time. An example (see Fig. 3) will make things clearer, before we go into more details. The functor in the example stores the variables `max_` and `sum_` internally. They are initialized appropriately in the constructor and can be read after the operation has completed using the `max` and `sum` methods. `operator()` is applied to each element in the range to find the maximum element and the sum of all elements.

Internally, the `reduce`-method creates a copy of the functor for each thread involved in the calculation. For this reason, the functors supplied must also have a copy constructor. In our example, the compiler takes care of this correctly, therefore we have omitted it. At the end of the reduction, the `combine`-method (which has nothing in common with the `combine` pattern mentioned in the last section!) is used to correctly combine the different functors from each thread. As can be seen in the example, there is no need to

```
/* filter all odd numbers and store them into results_ */
results_ = athenamp::filter<std::list<int> > (targets_.begin(),
    targets_.end(), std::bind2nd (std::modulus<int>(), 2));
```

Figure 4. `filter` in action

protect anything from concurrent access, as the `reduce`-method does this automatically. As a downside, this also means that our implementation has a critical region that must be carried out once by all threads involved, which of course decreases the performance slightly, especially for quick operations on few elements.

2.5 filter

The `filter` pattern filters a range of objects according to a predicate functor and stores all results for which the predicate returns true into a new container. The target container must be specified as template parameter and must offer both `push_back` and `insert`-methods in its interface (all STL sequence containers do). A small example to make things clearer is shown in Fig.4.

In this example, all odd numbers are filtered out of the targets_-vector and stored into the results_-list. The predicate functor can either be a standard one from the STL (as shown in the example) or a user-defined one. In all cases, no protection of the internals of the functor from concurrent access is guaranteed. This should not be a problem, as most functors used in this case do not have any internals to protect.

The implementation has no critical sections, but a small sequential part at the end where the contents of each threads accumulated objects are copied together into a new container that is returned afterwards.

3 Performance

Measuring performance of generic components such as the ones provided here is hard, as it depends heavily on the user code that is to be parallelized. Most patterns do not need locks or perform at most one locking operation per thread. Therefore, they are able to scale to a large number of processors. If the amount of work to be done in the user-supplied functor is too small, however, bus contention will become an issue and performance may not be satisfactory – but this is the case for pure OpenMP as well.

We have performed two different tests on our components. For brevity, we can only report the results for the `modify_each` pattern here. For the first test, we incorporated the pattern into the game-like application OpenSteerDemo[4], which is a testbed for the C++ open-source library OpenSteer written by Reynolds. It simulates and graphically displays the steering behaviour of autonomous computer-controlled characters, called agents, in real-time. No difference at all was measurable between the pure OpenMP version and our pattern version. We expect this to be the case in most applications.

The second test refers to the case that the user-supplied functor is too small. Our benchmark uses `modify_each` to add one to all elements in a vector or list, respectively. The results are shown in Fig. 1. Of course, this benchmark is not representative in any way,

Platform (Threads)	`modify_each` (lists)	OpenMP variant (lists)
AMD (4)	0.11 (8.06)	0.09 (7.56)
SPARC (8)	2.39 (35.8)	2.38 (34.8)
IBM (8)	0.15 (3.35)	0.14 (3.40)

Table 1. Wall-clock times in seconds for the `modify_each` function. The values in braces are measurements for the version of `modify_each` using forward iterators. All tests carried out on containers with 100.000 elements, using 10.000 repetitions.

as it is clearly limited by the available memory bandwidth. Yet, since this is true for the pure OpenMP-version as well, it shows that the overhead introduced by using the pattern is negligible even for this case.

It can also clearly be observed that the version of our patterns using forward iterators (as found in lists) is several orders of magnitude slower than the one for random access iterators. The reasons for this behaviour have been explained in Sec. 2. For our other patterns, similar results can be observed.

4 General Remarks and Evaluation

This section lists some implementation details and problems that were common to all patterns described here. The biggest problems while implementing the patterns was the state of the compilers available today. OpenMP and C++ are rarely used together in practice, and therefore many compilers are still buggy when it comes to this combination. This manifests itself in valid code that cannot be compiled on some compilers, or in subtle bugs that creep in as soon as high optimization settings are turned on. These high optimization settings are needed, though, because the way of programming shown here heavily relies on inlining. For example, without inlining there will be one function called for every element in the range to be processed by our patterns. When the function itself does not do much (as the one shown in Sect. 2.1), all performance gains from parallelization are lost. Good optimization by the compilers is therefore crucial to our work.

Another problem was the lack of support for changing the scheduling policy for parallel loops at runtime. With this feature, it would be possible to pass a parameter to any data-parallel pattern and let the user decide which scheduling policy works best for his particular problem. As an example, when the functors supplied to the pattern take a different amount of time to execute depending on their input, dynamic or guided scheduling usually work best. Currently, the user has to change the parameter inside the library or copy the code into his own program, where it can be modified.

Changing the scheduling policy at runtime is only possible with an environment variable in OpenMP. This feature is mostly used as a debugging aid, because one can only change the schedule of all loops (with a schedule set to `runtime`) at once. Use of environment variables is also not practical for a library. Letting the `schedule`-clause take a user-supplied parameter (e.g. a string) would solve this problem and is presently discussed in the OpenMP language committee.

5 Related Work

Many of the data-parallel patterns described here are similar to functionality provided by the Standard Template Library (STL). There are a variety of parallel STL-implementations in a research stage, among them STAPL[5] or PSTL[6]. Similar functionality is also starting to appear in commercial projects, such as the Intel Threading Building Blocks or in QT Concurrent. What differentiates AthenaMP from these projects is its strong focus on OpenMP on one hand (which no other project we know of provides), and its ability for programmers to study the source and learn from that. AthenaMP does not restrict itself to patterns found in the STL, but there are also task-parallel patterns (e.g. taskpool, pipeline), synchronization patterns (e.g. scoped locks, guard objects) and many more available in the library, which are not described in this paper, though. By using the expressiveness of OpenMP, its code is far easier to understand and adapt than any of the other libraries we are aware of.

6 Summary

This paper shows how it is possible to implement data-parallel patterns modelled after the Standard Template Library (STL) with OpenMP. The implementations we have described are part of the AthenaMP project[1]: `modify_each`, `transmute`, `combine`, `reduce`, and `filter`. Common characteristics are their iterator-based interface, the dynamic selection of the best suited implementation depending on the iterator-type supplied, and their ability to nest inside each other. The patterns were described in detail and with examples, along with performance measurements for a simple benchmark and the game-like application OpenSteerDemo. Afterwards, some implementation problems we had with OpenMP were shown, the biggest one being the state of the OpenMP-compilers today and the lack of support for runtime scheduling in OpenMP. The paper closed with a summary of related work.

References

1. Michael Suess. AthenaMP. `http://athenamp.sourceforge.net/`, 2006.
2. Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming (Software Patterns Series)*. Addison-Wesley Professional, 2004.
3. Michael Suess and Claudia Leopold. Common mistakes in OpenMP and how to avoid them. In *Proceedings of the International Workshop on OpenMP - IWOMP'06*, June 2006.
4. Bjoern Knafla and Claudia Leopold. Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP. In *Proceedings of the International Conference on Parallel Computing (PARCO 2007), Juelich, Germany*, 2007.
5. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. *Workshop on Languages and Compilers for Parallel Computing*, pages 193–208, 2001.
6. E. Johnson and D. Gannon. HPC++: experiments with the parallel standard template library. *Proceedings of the 11th international conference on Supercomputing*, pages 124–131, 1997.