Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores

Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch

University of Münster, Germany
{p.kegel,schellmann,gorlatch}@uni-muenster.de

Abstract. We compare two parallel programming approaches for multi-core systems: the well-known OpenMP and the recently introduced Threading Building Blocks (TBB) library by Intel®. The comparison is made using the parallelization of a real-world numerical algorithm for medical imaging. We develop several parallel implementations, and compare them w.r.t. programming effort, programming style and abstraction, and runtime performance. We show that TBB requires a considerable program re-design, whereas with OpenMP simple compiler directives are sufficient. While TBB appears to be less appropriate for parallelizing existing implementations, it fosters a good programming style and higher abstraction level for newly developed parallel programs. Our experimental measurements on a dual quad-core system demonstrate that OpenMP slightly outperforms TBB in our implementation.

1 Introduction

Modern CPUs, even on desktop computers, are increasingly employing multiple cores to satisfy the growing demand for computational power. Thus, easy-to-use parallel programming models are needed to exploit the full potential of parallelism. Early parallel programming models (e.g. Pthreads) usually allow for flexible parallel programming but rely on low-level techniques: The programmer has to deal explicitly with processor communication, threads and synchronization, which renders parallel programming tedious and error-prone. Several techniques exist to free programmers from low-level implementation details of parallel programming. One approach is to extend existing programming languages with operations to express parallelism. OpenMP is an example of this approach. Another approach is to introduce support for parallel programming within a library. Recently, Intel® published one such library, Threading Building Blocks (TBB), that adds support for high-level parallel programming techniques to C++.

The OpenMP application programming interface is quite popular for shared-memory parallel programming [3]. Basically, OpenMP is a set of compiler directives that extend C/C++ and Fortran compilers. These directives enable the user to explicitly define parallelism in terms of constructs for single program multiple data (SPMD), work-sharing and synchronization. OpenMP also provides some library functions for accessing the runtime environment. At compile time, multi-threaded program code is generated based on the compiler directives. OpenMP has been designed to introduce parallelism in existing sequential programs. In particular, loop-based data parallelism can be easily exploited.

Threading Building Blocks (TBB) is a novel C++ library for parallel programming, which can be used with virtually every C++ compiler. In TBB, a program is described in terms of fine-grained tasks. Threads are completely hidden from the programmer. TBB's idea is to extend C++ with higher-level, task-based abstractions for parallel programming; it is not just a replacement for threads [6]. At runtime, TBB maps tasks to threads. Tasks are more light-weight than threads, because they cannot be preempted, such that no context switching is needed. They may be executed one after another by the same thread. This reduces scheduling overhead, which usually occurs when each program task is defined and performed by a single thread. Tasks also can be re-scheduled at runtime to load-balance threads. This mechanism is referred to as "task-stealing" [6]. TBB provides skeletons for common parallel programming patterns, like map, reduce or scan. Thereby it facilitates high-level parallel programming.

This paper studies the use of OpenMP and TBB for parallelizing an existing sequential implementation of an algorithm for image reconstruction. We compare parallel implementations of this algorithm based on OpenMP and TBB, using such criteria as programming effort, programming style and abstraction, and runtime performance. In Section 2, the algorithm that we want to parallelize and its field of application are presented. The actual parallelization is described in Section 3. Here, we provide some code examples and a detailed description of several parallel implementations of the algorithm based on OpenMP and TBB. In Section 4, we compare our TBB- and OpenMP-based implementations. We present runtime performance and discuss programming style. Finally, we conclude with an evaluation of TBB and OpenMP in Section 5 with respect to our algorithm and describe our ongoing and future work.

2 Positron Emission Tomography (PET) Imaging

In Positron Emission Tomography (PET), a radioactive substance is injected into a specimen (mice or rats in our case). Afterwards, the specimen is placed within a scanner, a device that is facilitated with several arrays of sensors. As the particles of the applied substance decay, two positrons are emitted (hence the name PET) in opposite directions. The "decay events" are detected by the sensors, usually by two opposite sensors in parallel. The scanner records these events, with each record comprising a timestamp and the positions of the two sensors.

ListMode Ordered Subset Expectation Maximization [5] (LM OSEM) is a block-iterative algorithm for 3D image reconstruction. LM OSEM takes a set of the aforementioned recorded events and splits them into s equally sized subsets. For each subset $l \in 0, \ldots, s-1$ the following computation is performed:

$$f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A_N^t \mathbf{1}} \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l}. \tag{1}$$

Here $f \in \mathbb{R}^n$ is a 3D image in vector form with dimensions $n = (X \times Y \times Z)$. $A \in \mathbb{R}^{m \times n}$, element a_{ik} of row A_i is the length of intersection of the line between the two detectors of event i with voxel k of the reconstruction region, computed using Siddon's algorithm [9]. Each subset's computation takes its predecessor's output image as input and produces a new, more precise image.

The overall structure of a sequential LM OSEM implementation is shown in Listing 1. It comprises three nested loops, one outer loop with two inner loops. The outer loop iterates over the subsets. The first inner loop iterates over a subset to compute the summation part of c_l. The second inner loop iterates over all elements of f_l and c_l to compute f_{l+1}.

```
for (int l = 0; l < subsets; l++) {
  /* read subset */

  /* compute c_l */
  #pragma omp parallel
  {
    #pragma omp for schedule(static)
    for (int i = 0; i < subset_size; i++) {
      ...
    }
  } /* end of parallel region */

  /* compute f_l+1 */
  #pragma omp parallel for schedule(static)
  for (int k = 0 ; k < image_size; k++) {
    if (sens[k] > 0.0 && c_l[k] > 0.0)
      f[k] = f[k] * c_l[k] / sens[k];  }  }
```

Listing 1. The original sequential implementation comprises one outer loop with two nested inner loops. For parallelization it is augmented with OpenMP compiler directives.

A typical 3D image reconstruction processing 6×10^7 million input events for a $150 \times 150 \times 280$ PET image takes more than two hours on a common PC. Several parallel implementations for systems with shared- and distributed-memory, as well as hybrid systems have been developed to reduce the algorithm's runtime [4,7]. Also an implementation for Compute Unified Device Architecture (CUDA) capable graphics processing units is available [8].

3 Parallelizing the LM OSEM Algorithm

Because of the data dependency between the subset's computations (implied by $f_{l+1} = f_l c_l$), the subsets cannot be processed in parallel. But the summation part of c_l and the computation of f_{l+1} are well parallelizable. (Additional parts of the implementation, e.g., for performing an image convolution to improve image quality, can also be parallelized. However, none of these program parts have to do directly with LM OSEM and will be omitted in the sequel.)

3.1 Loop Parallelization

To parallelize the summation part of c_l and the computation of f_{l+1}, we have to parallelize the two inner loops of the LM OSEM algorithm. OpenMP and TBB both offer constructs to define this kind of data parallelism.

In OpenMP, two constructs are used to parallelize a loop: the *parallel* and the *loop* construct. The parallel construct, introduced by a `parallel` directive, declares the following code section (the *parallel region*) to be executed in parallel by a team of threads. Additionally, the loop construct, introduced by a `for` directive, is placed within the parallel region to distribute the loop's iterations to the threads executing the parallel region. We parallelize the inner loop using these two constructs, as shown in Listing 1. For the first inner loop, each thread must perform some preliminary initializations. The according statements are placed at the parallel region's beginning. The second inner loop does not need any initializations. Therefore, we use the `parallel for` directive, which is a shortcut declaration of a parallel construct with just one nested loop construct. Apart from the additional compiler directives, no considerable changes were made to the sequential program. Thus, an OpenMP-based parallel implementation of LM OSEM is easily derived from a sequential implementation of the LM OSEM algorithm written in C [7], see Listing 1.

OpenMP supports several strategies for distributing loop iterations to threads. The strategy is specified via the `schedule` clause, which is appended to the `for` directive. In our application, we expect that the workload is evenly distributed in iteration ranges of reasonable size (greater than 10000 events). Therefore, a static and even distribution of iterations to threads will be sufficient for our application. OpenMP's `static` scheduling strategy suits our needs. If not specified otherwise, it creates evenly sized blocks of loop iterations for all threads of a parallel region. Thereby, the block size implicitly is defined as n/p, where n is the number of iterations and p is the number of threads.

In TBB, parallel loops are defined using the `parallel_for` template, which is applied in two steps as proposed by Reinders [6]:

1. A class is created, which members correspond to the variables that are outside the scope of the loop to be parallelized. The class has to overload the `()` operator method so that it takes an argument of type `blocked_range<T>` that defines a chunk of the loop's iteration space which the operator's caller should iterate over. The method's body takes the original code for the loop. Then, the arguments for the loop's iteration bounds are replaced by calls to methods that return the iteration space's beginning and end.
2. The code of the original loop is replaced by a call to the `parallel_for` pattern. The pattern is called with two arguments, the loop's iteration space and an instance of the previously defined class, called the *body object*. The iteration space is defined by an instance of the `blocked_range<T>` template class. It takes the iteration space's beginning and end and a *grain size* value.

A parallel implementation of LM OSEM using TBB thus requires to re-implement the two inner loops. The most obvious difference to OpenMP is the use of C++ instead of C. TBB's `parallel_for` construct is a C++ template and takes a C++ class as parameter. Actually, this technique is a workaround for C++'s missing support of *lambda expressions*. With lambda expression, blocks of code can be passed as parameters. Thus, the code of the body object could be passed to the `parallel_for` template *in-place* without the overhead of a separate class definition [6].

According to step 1, we create a class for each of the loops. Within these classes, we change the loops' iteration bounds and provide member variables that save the original outer context of the loops (see Listing 2). We determined these variables by carefully analyzing our existing implementation of LM OSEM.

```
class ImageUpdate {
  double *const f, *const c_l;
  double *const sens,

public:
  ImageUpdate(double *f, double *sens, double *c_l) :
    f(f), sens(sens), c_l(c_l) {}

  void operator() (const blocked_range<int>& r) const {
    for (int k = r.begin(); k != r.end(); k++) {
      if (sens[k] > 0.0 && c_l[k] > 0.0)
        f[k] *= c_l[k] / sens[k];  }  }
};
```

Listing 2. The loop to be parallelized is embedded into a separate class that defines a method `operator()`

Then, following step 2, we are able to replace the loops by calls to TBB's `parallel_for` template (see Listing 3).

```
for (int l = 0; l < subsets; l++) {
  /* read subset */

  /* compute c_l */
  parallel_for(
    blocked_range<int>(0, subset_size, GRAIN_SIZE),
    SubsetComputation(f, c_l, event_buffer, precision));

  /* compute f_l+1 */
  parallel_for(
    blocked_range<int>(0, image_size, GRAIN_SIZE),
    ImageUpdate(f, sens, c_l));  }
```

Listing 3. A call to the `parallel_for` template replaces the loop

The grain size value specifies a reasonable maximum number of iterations that should be performed by a single processor. Unlike OpenMP, which provides multiple scheduling strategies, TBB always uses a fixed scheduling strategy that is affected by the selected grain size. If the size of an iteration range is greater than the specified grain size, the `parallel_for` template recursively splits it into disjoint iteration ranges. The grain size value thus serves as a lower bound for the tasks' size. If each task comprised a single iteration, this would result in a massive task scheduling overhead. Internally, the `parallel_for` template creates a task for each iteration range in a divide and conquer manner. Finally, all tasks are processed independently; i.e., in parallel.

TBB's grain size value is comparable to an optional *block size* value, which can be specified for OpenMP's `schedule` clause. Both specify a threshold value for chunks of iterations, to avoid too large or too small work pieces for each thread. However, the semantics of the block size value slightly differ from the grain size value and depend on the selected scheduling strategy. Reinders [6] proposes to select a grain size, such that each iteration range (task) takes at least 10,000 to 100,000 instructions to execute. According to this rule of thumb we select a grain size value of 1,000 for the first inner loop and a value of 10,000 for the second inner loop.

3.2 Thread Coordination

Within the first inner loop (summation part of c_l) all threads perform multiple additions to arbitrary voxels of a common intermediate image. Hence, possible race conditions have to be prevented. There are two basic techniques for this:

1. Mutexes: The summation part is declared mutually exclusive, such that only one thread at a time is able to work on the image.
2. Atomic operations: The summation is performed as an atomic operation.

In OpenMP, both techniques are declared by appropriate directives. Mutexes are declared by using the *critical* construct. Similarly to the aforementioned parallel construct, it specifies a mutual exclusion for the successive code section (see Listing 4).

```
/* compute c_l */
...
#pragma omp critical
while (path_elements[m].coord != −1) {
  c_l[path_elem[m].coord] += c * path_elem[m].length
} /* end of critical section */
```

Listing. 4. Mutex in OpenMP: The *critical* construct specifies a mutex for the successive code section (critical region)

OpenMP's *atomic* construct is used similarly to the *critical* construct (see Listing 5). It ensures that a specific storage location is updated without interruption (atomically) [1]. Effectively, this ensures that the operation is executed only by one thread at a time. However, while a critical region semantically locks the whole image, an atomic operation just locks the voxel it wants to update. Thus atomic operations in our case may be regarded as fine-grained locks.

```
/* compute c_l */
...
while (path_elements[m].coord != −1) {
  path_elem[m].length *= c;
  #pragma omp atomic
  c_l[path_elem[m].coord] += path_elem[m].length;  }
```

Listing 5. Atomic operation in OpenMP: The *atomic* construct ensures that the voxels of the intermediate image are updated atomically

Besides, mutual exclusion can be implemented explicitly by using low-level library routines. With these routines, in general, a *lock variable* is created. Afterwards a lock for this variable is acquired and released explicitly. This provides a greater flexibility than using the *critical* or *atomic* construct.

TBB provides several mutex implementations that differ in properties like scalability, fairness, being re-entrant, or how threads are prevented from entering a critical section [6]. *Scalable* mutexes do not perform worse than a serial execution, even under heavy contention. Fairness prevents threads from starvation and, for short waits, *spinning* is faster than sending waiting threads to sleep. The simplest way of using a mutex in TBB is shown in Listing 6. The mutex lock variable is created explicitly, while a lock on the mutex is acquired and released implicitly. Alternatively, a lock can be acquired and released explicitly by appropriate method calls, similarly to using the OpenMP library functions.

```
tbb::mutex mtx; /* create mutex */
...
{ /* beginning of lock scope */
  tbb::mutex::scoped_lock lock(mtx); /* acquire lock on mutex */
  while (path_elements[m].coord != −1) {
    c_l[path_elem[m].coord] += c * path_elem[m].length;  }
} /* end of scope, release lock */
```

Listing 6. Mutex in TBB: The constructor of `scoped_lock` implicitly acquires a lock, while the destructor releases it. The brackets keep the lock's scope as small as possible.

Atomic operations can only be performed on a template data type that provides a set of methods (e.g. `fetchAndAdd`) for these operations. Thus, the original data type (e.g., the image data type of LM OSEM) would have to be changed throughout the whole program. In our application, the voxels of an image are represented by double-precision floating-point numbers. However, TBB only supports atomic operations on integral types [6]. Therefore, with TBB, we cannot use atomic operations in our implementation of the LM OSEM algorithm.

4 Comparison: OpenMP vs. TBB

To compare OpenMP and TBB concerning our criteria of interest, we create several parallel implementations of LM OSEM using OpenMP and TBB. Two versions of an OpenMP-based implementation are created using the *critical* and the *atomic* construct, respectively. Moreover, we create three versions of our implementation based on TBB using TBB's three basic mutex implementations: a `spin_mutex` (not scalable, not fair, spinning), a `queuing_mutex` (scalable, fair, spinning) and a wrapper around a set of operation system calls (just called `mutex`) that provides mutual exclusion (scalability and fairness OS-dependent, not spinning).

4.1 Programming Effort, Style, and Abstraction

The previous code examples clearly show that TBB requires a thorough redesign of our program, even for a relatively simple pattern like `parallel_for`. Our TBB-based implementations differ greatly from the original version. We had to create additional classes, one for each loop to be parallelized, and replace the parallelized program parts by calls to TBB templates. Basically, this is because TBB uses library functions that depend on C++ features like object orientation and templates. Consequently, the former C program becomes a mixture of C and C++ code. The most delicate issue was identifying the variables that had to be included within the class definition of TBB's `parallel_for` body object.

Parallelizing the two inner loops of the original LM OSEM implementation using OpenMP is almost embarrassingly easy. Inserting a single line with compiler directives parallelizes a loop without any additional modifications. Another single line implements a mutual exclusion for a critical section or the atomic execution of an operation. Also, in contrast to TBB, we do not have to take any measures to change variable scopes. OpenMP takes care of most details of thread management.

Regarding the number of lines of code, OpenMP is far more concise than TBB. The parallel constructs are somewhat hidden in the program. For example, the end of the parallel region that embraces the first inner loop of LM OSEM is hard to identify in the program code. TBB, on the other hand, improves the program's structure. Parallel program parts are transferred into separate classes. Though this increases the number of lines of code, the main program becomes smaller and more expressive through the use of the `parallel_for` construct.

The parallel constructs of TBB and OpenMP offer a comparable level of abstraction. In neither case we have to work with threads directly. TBB's `parallel_for` template resembles the semantics of OpenMP's parallel loop construct. OpenMP's parallel loop construct can be configured by specifying a scheduling strategy and a block size value, whereas TBB relies solely on its task-scheduling mechanism. However, OpenMP's parallel loop construct is easier to use, because the `schedule` clause may be omitted.

Regarding thread coordination, OpenMP and TBB offer a similar set of low-level library routines for locking. However, OpenMP additionally provides compiler directives which offer a more abstract locking method. Also, for our application, TBB has a serious drawback regarding atomic operations: these operations are only supported for integral types (and pointers).

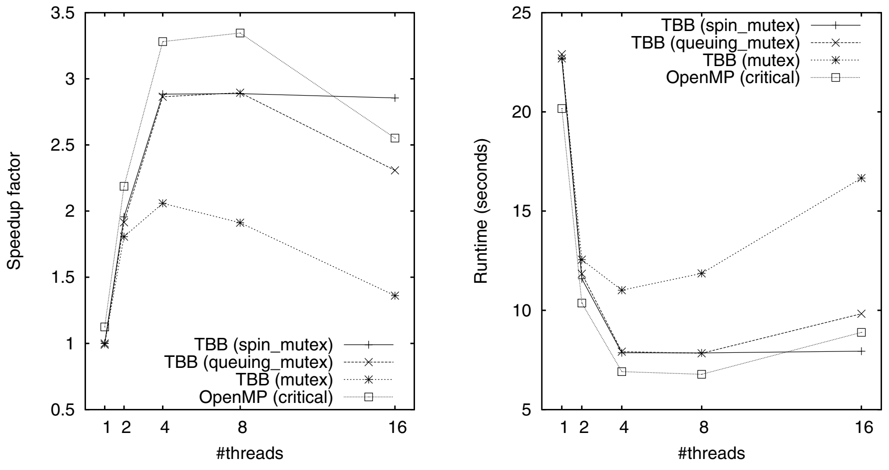
4.2 Runtime Performance

To give an impression of the performance differences between OpenMP and TBB with respect to our application, we compare the OpenMP-based parallel implementations with the ones based on TBB.

We test our implementations on a dual quad-core (AMD Opteron™ 2352, 2.1 GHz) system. Each core owns a 2×64 KB L1 cache (instruction/data) and 512 KB L2 cache. On each processor, four cores share 2 MB L3 cache and 32 GB of main memory. The operating system is Scientific Linux 5.2.

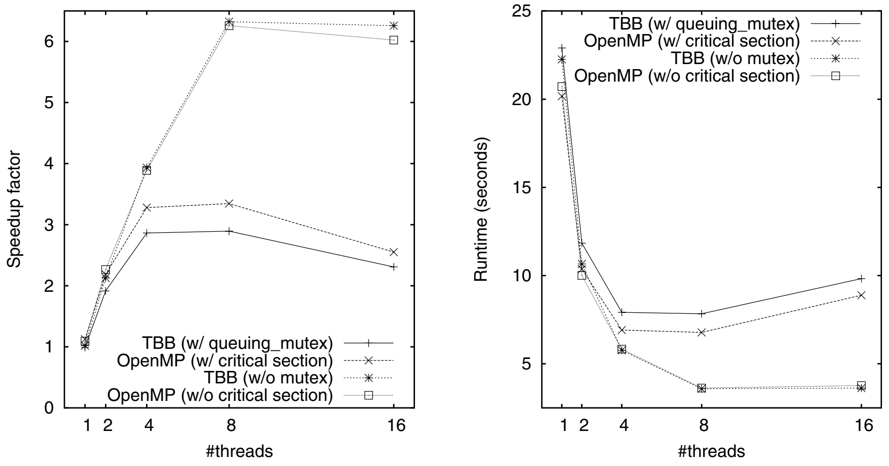
We run the LM OSEM algorithm on a test file containing about 6×10^7 events retrieved from scanning a mouse. To limit the overall reconstruction time in our tests, we process only about 6×10^6 of these events. The events are divided into ten subsets of one million events each. Thus, we get subsets that are large enough to provide a reasonable intermediate image (c_l), while the number of subsets is sufficient for a simple reconstruction. Each program is re-executed ten times. Afterwards, we calculate the average runtime for processing a subset. We repeat this procedure using 1, 2, 4, 8, and 16 threads. The number of threads is specified by an environment variable in OpenMP or by a parameter of the task scheduler's constructor in TBB, respectively.

Figure 1 shows a comparison of the results of the implementations of LM OSEM using mutexes. Most of the implementations show a similar scaling behavior. Exceptions are the implementations based on TBB's `mutex` wrapper and



(a) speedup (b) runtime

Fig. 1. Runtime and speedup for a single subset iteration of different implementations of LM OSEM

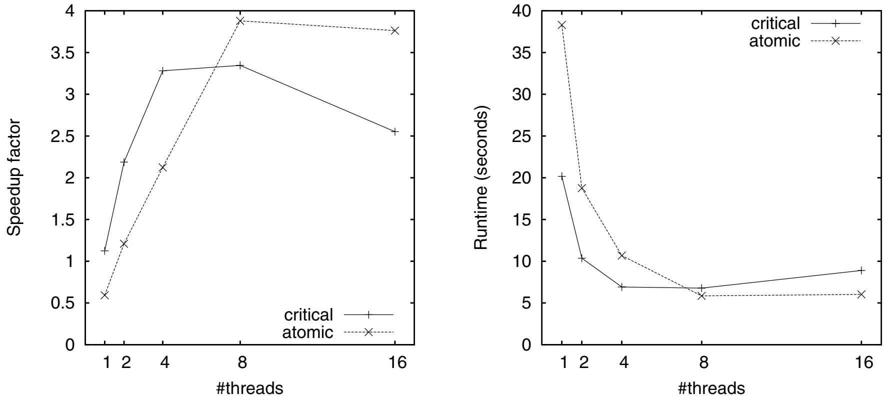


(a) speedup (b) runtime

Fig. 2. Runtime and speedup for a single subset iteration of different implementations of LM OSEM. Without thread coordination both implementations perform equally.

`spin_mutex`. The implementation based on the `mutex` wrapper performs worst of all implementations. In particular, it shows a much worse scaling behavior. With 16 threads (oversubscription), the `spin_mutex`-based implementation performs best of all. However, OpenMP's *critical* construct outperforms most of TBB's mutex implementations.

To clarify the reasons for TBB's inferior performance as compared to OpenMP, we repeat our experiments with two implementations that do not prevent race



(a) speedup (b) runtime

Fig. 3. Runtime and speedup for a single subset iteration of the OpenMP-based LM OSEM implementations

conditions in OpenMP or TBB, respectively. The results (see Figure 2) show that without a mutex (or critical section), TBB and OpenMP perform virtually equally well.

A comparison of the two OpenMP-based implementations of LM OSEM (see Figure 3) reveals a remarkable difference. The runtime of the single-threaded case shows that using atomic operations slows down our implementation by about 50%. However, when the number of threads increases, the implementation using atomic operations provides an almost linear speedup. With about 8 threads, it finally outperforms the OpenMP-based implementation using the *critical* construct and all implementations using TBB's mutexes.

One should keep in mind that the performance of an OpenMP program heavily depends on the used compiler (Intel C/C++ compiler version 10.1 in our case). While the TBB library implements parallelization without any special compiler support, with OpenMP the parallelization is done by the compiler. Hence, the presented results do not apply to every OpenMP-capable compiler in general.

5 Conclusion

Using OpenMP, we can parallelize LM OSEM with little or no program redesign. It can be easily used for parallelizing our existing sequential implementation. TBB, on the other hand, fosters a structured, object-oriented programming style, which comes along with the cost of re-writing parts of the program. Hence, TBB is more appropriate for the creation of a new parallel implementation of LM OSEM. Besides, TBB is C++ based. Hence, one might argue that using TBB also requires object-oriented programming. *Lambda expressions* (see Section 3.1) will probably become part of the next C++ standard. This might enable TBB to implement loop parallelization *in place*, such that at least the effort for loop parallelization might be reduced to a level comparable to that of OpenMP [6].

With TBB, we can implement the same kind of data-parallelism as with OpenMP. Also, both approaches provide an abstract programming model, which shields the programmer from details of thread programming.

Regarding runtime, we get the best results when using OpenMP. TBB offers a comparable scaling behavior, but does not achieve a similar absolute performance. Our experimental results show that TBB's inferior performance is probably caused by its non-optimal mutex implementations, while its `parallel_for` template provides equal performance. Besides, TBB lacks atomic operations for floating-point data types, while such operations provide the best performance when used in our OpenMP-based implementation.

In conclusion, OpenMP apparently is a better choice for easily parallelizing our sequential implementation of the LM OSEM algorithm than TBB. OpenMP offers a fairly abstract programming model and provides better runtime performance. Particularly thread synchronization, which is crucial in our application, is easier to implemented when using OpenMP.

Nevertheless, apart from the `parallel_for` template, TBB offers some additional constructs (e.g. *pipeline*), which provide a higher level of abstraction. In particular, these constructs allow the implementation of task-parallel patterns. We excluded these constructs from the comparison, because OpenMP contains no comparable features. If we started a new implementation of the LM OSEM algorithm with parallelism in mind, we might much easier exploit parallelism in TBB than in OpenMP by using these high-level constructs. For example, we might concurrently fetch and process events and calculate c_l by using TBB's pipeline construct. Especially, we would not have to take care about thread synchronization explicitly as we have to do in our current implementations. OpenMP does not provide such a construct, so that it would have to be implemented from scratch. Currently, we are working on implementations of LM OSEM using these features to exploit a higher level of programming.

In our future experiments, we also plan to analyze the exact influence of grain and block size and scheduling strategy (see section 3.1) for different amounts of input data on the program performance.

The most important feature of TBB is its task-based programming model. Usually, templates shield the programmers from directly using tasks. The latest version 3.0 of OpenMP also includes a task-based parallel programming model [2]. We are going to study how far this task model is comparable to TBB and whether it increases OpenMP's performance.

References

1. OpenMP.org – The OpenMP API specification for parallel programming, `http://openmp.org/`
2. OpenMP Architecture Review Board. OpenMP Application Program Interface (May 2008)
3. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP - Portable Shared Memory Parallel Programming. MIT Press, Cambridge (2007)
4. Hoefler, T., Schellmann, M., Gorlatch, S., Lumsdaine, A.: Communication optimization for medical image reconstruction algorithms. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 75–83. Springer, Heidelberg (2008)
5. Reader, A.J., Erlandsson, K., Flower, M.A., Ott, R.J.: Fast accurate iterative reconstruction for low-statistics positron volume imaging. Physics in Medicine and Biology 43(4), 823–834 (1998)
6. Reinders, J.: Outfitting C++ for Multi-core Processor Parallelism - Intel Threading Building Blocks. O'Reilly, Sebastopol (2007)
7. Schellmann, M., Kösters, T., Gorlatch, S.: Parallelization and runtime prediction of the listmode osem algorithm for 3d pet reconstruction. In: IEEE Nuclear Science Symposium and Medical Imaging Conference Record, San Diego, pp. 2190–2195. IEEE Computer Society Press, Los Alamitos (2006)
8. Schellmann, M., Vörding, J., Gorlatch, S., Meiländer, D.: Cost-effective medical image reconstruction: from clusters to graphics processing units. In: CF 2008: Proceedings of the 2008 conference on Computing Frontiers, pp. 283–292. ACM, New York (2008)
9. Siddon, R.L.: Fast calculation of the exact radiological path for a three-dimensional CT array. Medical Physics 12(2), 252–255 (1985)