The GNU libstdc++ parallel mode: Software Engineering Considerations

Johannes Singler *
Universität Karlsruhe
Kaiserstraße 12
76128 Karlsruhe, Germany
singler@ira.uka.de

Benjamin Kosnik
Redhat, Inc.
101 N. Wacker Drive, Suite 150
Chicago, IL 60606
bkoz@redhat.com

ABSTRACT

The C++ Standard Library implementation provided with the free GNU C++ compiler, *libstdc++*, provides a "parallel mode" as of version 4.3. Using this mode enables existing serial code to take advantage of many parallelized STL algorithms, an approach to making use of multi-core processors which are now or will soon will be ubiquitous. This paper describes the software engineering issues discovered during implementation, the results of user testing, and presents possible solutions to outstanding issues. Design issues with configuring the software environment to a wide variety of multi-core hardware options, influencing algorithm and parameter choices at compile and run time, standards compliance, and the interplay between execution speed, the executable size, the library code size, and the compilation time are addressed.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Design, Languages, Standardization

Keywords

Parallel Algorithm Libraries

1. INTRODUCTION

Nowadays, nearly all personal computers ship with more than one processor core on a chip. However, the increased computing performance from the underlying hardware is often rendered useless due to insufficiently parallelized software. Thus, the high performance possibilities inherent in multi-CPU hardware is at present largely theoretical.

Parallelizing software is difficult. There exist a wide variety of approaches to extracting parallelism: new languages, language extensions, specialized libraries, specialized compilers. The tools for parallel software engineering are quite primitive as compared to existing tools for traditional serial software development.

In addition, writing properly parallelized programs is hard. Software developers not only have to think about the implementation of threading, but also about how to actually parallelize the algorithms, take into account hardware variance, and then do testing and debugging. This is beyond the abilities of many programmers, and thus costly and error-prone when attempted.

C++ is a very well-established programming language with a rich syntax, supporting both object-oriented and generic programming. It comes with no inherent performance penalty, as do languages requiring a virtual machine, e. g. Java or C#. This makes it a ideal candidate for applications executing compute-intensive algorithms. The standardization of C++ [4] has set a common ground for language and library implementations, which are widely available and often of high quality. The C++ standard is also evolving, as can be seen by the proposed upcoming standard, termed C++0x [3]. The Standard Template Library [12] is part of the existing library specification, and provides many useful generic data structures and algorithms to its user. Since this interface is long-established, well-understood, and widely used, it is a very good way to bring easy-to-use parallel algorithms to the programmer. By utilizing parallelized STL algorithms as black boxes, the programmer can *completely abstract* from parallel programming, but still benefit from multi-core power. If activated, a usual call to an STL algorithm will cause its parallel execution. So this is a very good example for the concept of encapsulation, too. It even works for existing code, and the parallelization can optionally happen in an incremental way.

The implementation of the parallel mode was begun under the name *Multi-Core Standard Template Library (MCSTL)* [13] in 2006, followed by its integration into the STL implementation of the free[1] GNU C++ compiler (g++)

*Author partially supported by DFG grant SA 933/3-1.

IWMSE'08, May 11, 2008, Leipzig, Germany.

[1]Both free as beer and free as speech. The compiler is open-source and licensed under the GNU Public License,

in 2007. g++ is available for a large number of platforms, including x86, x86_64, IA-64, and Sparc. It provides state-of-the-art optimization.

The parallel mode includes functionality like load balancing embarrassingly parallel work, sorting, merging two or more sorted sequences, random shuffling, partition of a sequence by a pivot, finding single elements or substrings in sequences, prefix sums, and many more. Thus, it enables the programmer to use parallelized standard algorithms as subtasks of the actual problem solving.

This paper states the goals of libstdc++ parallel mode and shows up the software engineering issues we encountered while developing and integrating it. For many of them, we present resolutions, which might include trade-offs between desirable properties. Some of them cannot be resolved satisfactorily in the current setting. In this case, the paper intends to foster research for development of the C++ language and its standard library. The solutions presented here are propositions, may not be integrated into the current parallel mode code (yet), and are subject to change.

The algorithmic aspects have been considered in [13]. The general performance and scalability of the algorithms is comparable to the ones in the MCSTL, and has been evaluated in that context before. Still, we will discuss performance implications stemming from the fact that a library must be general, thus making specific optimizations difficult.

2. SOFTWARE ENGINEERING ASPECTS

Major software-engineering-related goals are:

1. Transparent integration of parallel algorithms, i. e. compile in sequential mode and parallel mode, without changes to the user code. Pragmatic balance between adherence to the C++ standard and parallelization/software engineering benefits.
2. Possibility to choose and tune algorithms, and to adjust the degree of parallelism, in order to achieve best performance.
3. Maintainability of the library code, i. e. little code duplication, much code reuse.
4. Limited increase in compilation time and executable size, with respect to the user application.

2.1 Threading

We chose the OpenMP compiler extension and API in version 2.5 [11] for managing the threads. It has many advantages over strictly library-based approaches like `pthread`.

- OpenMP provides so-called "fork-join" parallelism, which is right what we need for parallelizing single algorithms. The work is divided, the threads start, they re-join, the function call returns.
- Low-level issues like thread pooling, and getting information about the environment, e. g. the number of cores, can be left to the OpenMP runtime. The user can configure the runtime environment using standardized procedures.

the libstdc++ under the GPL with runtime exception. That means that even commercial programs may use the library without any having to publish the program's source code.

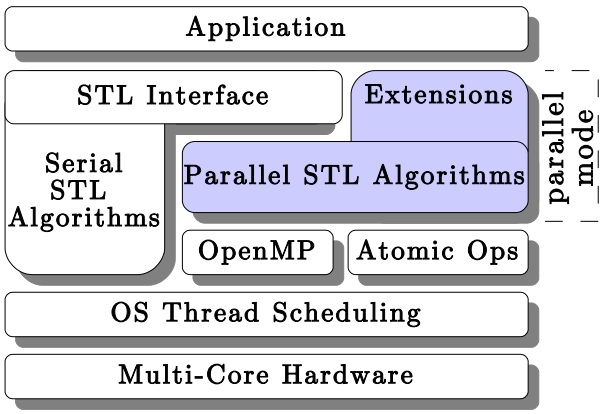

Figure 1: Abstraction layers of the parallel mode.

- Augmented code also compiles and runs correctly (though not in parallel) with OpenMP stub support, which can be added to every compiler, easily by the user.
- OpenMP is platform-independent. GCC supports OpenMP 2.5 from version 4.2. It is also supported by all major commercial C++ compilers.

In addition to OpenMP, the parallel mode uses atomic operations like `fetch-and-add`, as provided by the GCC's lower-level libraries. The abstraction layers are visualized in Figure 1.

2.1.1 *OpenMP Compliance*

A program incorporating algorithms from the libstdc++ parallel mode tries to behave like any other OpenMP-parallelized program. This includes obedience to the environment variables defined in the OpenMP standard. Most important are `OMP_NUM_THREADS` and `OMP_DYNAMIC`, which influence the number of threads. By default, `OMP_NUM_THREADS` is usually set to the number of cores available in the system, `OMP_DYNAMIC` is switched off. Although an OpenMP `parallel` clause takes an argument on the desired number of threads, this value poses an upper limit only. If the dynamic mode is switched on, the runtime can decide to schedule a smaller number of threads to the parallel region. In the end, this means that the number of threads is clear only after the `parallel` statement, i. e. in the parallel region. The code must not make any assumptions about the number of threads beforehand. In many algorithms, this leads to `single` block right after the beginning of the parallel region, in order to distribute the work appropriately. For best performance, the compiler should identify this case, and start the threads only after the `single` block, having fixed the number of threads in advance, and thus returning the right value from `omp_get_num_threads`.

The environment variable `OMP_SCHEDULE` is less important, since it only influences OpenMP `parallel for` loops, which are not used by the parallel mode at the moment.

Nested parallelism *is* used by algorithms, namely the parallel quicksort variants. It is a classic example for parallel divide-and-conquer. Conceptually, the threads partition the

sequence into two parts, in parallel. After that, the threads are split into two groups corresponding to the sizes of the sub-sequences. When only one threads remains in a group, it solves the problem on its own, possibly helping out slower threads via work-stealing.

In the actual implementation, the threads are not split, but newly started, relying on the OpenMP runtime's thread pool in terms of efficiency. The desired effect is achieved by having a `parallel` region with two `sections`. In each of this sections, an appropriate number of worker threads are spawned for the parallel partitioning. After that, the sorting step is called recursively, which leads to nested parallelism. There is a termination condition that stops the spawning of new threads such that the total number of threads never exceeds a certain maximum number, determined by calling `omp_get_max_threads` at the start of the algorithm. The runtime deciding to execute two sections one after the other does not threaten correctness, either.

It the user switches off nested parallelism using the `OMP_NESTED` environment variable, this scheme is destroyed. The algorithm will still run correctly, but its speedup will be limited to 2.

Of course, an equivalent algorithm could be implemented using a single parallel region, by scheduling all arising work "manually". However, this would imply the following disadvantages:

- The parallel partitioning step could not be done by the already existing parallelized `partition` call, because the threads are already in existence. A less elegant solution, minimizing code duplication, would be to tear apart the partition algorithm, and to double-use its parts in both the quicksort and the `partition` implementation.
- Threads of a group have to barrier synchronize once in a while, which can imply (OS-based) waiting. However, the OpenMP `barrier` statement takes all threads in a *team* into account, not only the ones in the algorithm-specified and data-dependent group. Solving this case inside the OpenMP specification could only be done by using `omp_set_lock` and `omp_get_lock`, which is beyond their intended purpose. OpenMP lock functionality does not support the monitor concept where multiple threads can be notified at once. Thus, each thread would have to wait for its predecessor thread to finish, with respect to a predetermined order. This adds up the inefficiencies.

A future elegant resolution without the problems noted here, using OpenMP 3.0, is sketched in Section 2.11.

2.2 Activation of the Parallel Mode

The use of the parallel mode STL should be optional, i. e. by default, it is deactivated by the compiler. If the programmer wants to use the parallel mode, there should be two options:

1. Activate the parallel mode by default, i. e. all STL algorithms that have parallel implementations available, will have parallel code included into the program.
2. Activate the parallel mode on request, i. e. specific algorithm calls can be annotated by the programmer to call the parallel version of the algorithm.

By inserting a `using namespace ... __attribute__ ((__strong__));` declaration[2], the contents of a namespace can be transparently imported into another one, superseding the already existing symbols there. This procedure is called *namespace association* [1]. The parallel implementations from namespace `__gnu_parallel` are imported into the unaltered namespace `std`, the parallel calls thereby hiding the sequential implementations[3]. The sequential versions are not completely inaccessible in this case, it is just that the imported functions are chosen by default. This is just what we want for the first case. If the programmer wants to force sequential execution at a certain spot, he just adds `__gnu_parallel::sequential_tag()` behind all other call arguments.

In the second case, the programmer just has to prepend the namespace qualifier `__gnu_parallel::` to his call, to switch to the parallel version explicitly.

2.3 Sequence Access through Iterators

Most STL algorithms take one or more sequences as their main argument(s). To abstract from the underlying representation, the iterator concept is used. A sequence is usually specified by passing an iterator referencing the first element `begin()`, and an iterator referencing a virtual element behind the last element `end()`. This allows a consistent treatment of arrays, regarding pointers as iterators. Also, different iterator types or instantiations can access a single sequence in different ways, e. g. selecting a range, advancing with a stride, going backwards. On the one hand side, this adds a lot of flexibility.

On the other hand side, information gets lost when passing iterators. The best example for that is the length of a linked listed. A linked list container might implement the size query efficiently, just by book keeping the number of inserted and deleted elements. When an algorithm is passed the corresponding `begin()` and `end()` iterators, however, this information is lost, since the iterators do not include a reference to the underlying container. Since the iterators are not random-access iterators, the distance between them cannot be computed efficiently. The only way to find out the length of the sequence is to traverse the whole sequence, taking sequential linear time, and impossible to parallelize.

Since all parallel mode algorithms are *data-parallel*, the input must be split into parts, the splitting positions being scattered along the whole sequence. This can only be done in sublinear time, if random access to the elements is granted. Hence, parallel execution seems to be limited to random-accessible sequences. We have given thought on how to workaround this problem with using as little sequential computation as possible [7], but this is a secondary choice.

The decision on whether random access is possible happens at compile time, using partial template specialization. For each used iterator, an appropriate specialization of `std::iterator_traits` must be available, as requested for all iterators by the standard [4, Section 24.3.3].

2.4 Code Reuse

Parallel algorithms, in general, are far more complicated

[2]This declaration currently is a GNU extension, but will probably be integrated into the upcoming standard.

[3]A similar construction has already existed in the libstdc++, namely for the *debug mode*.

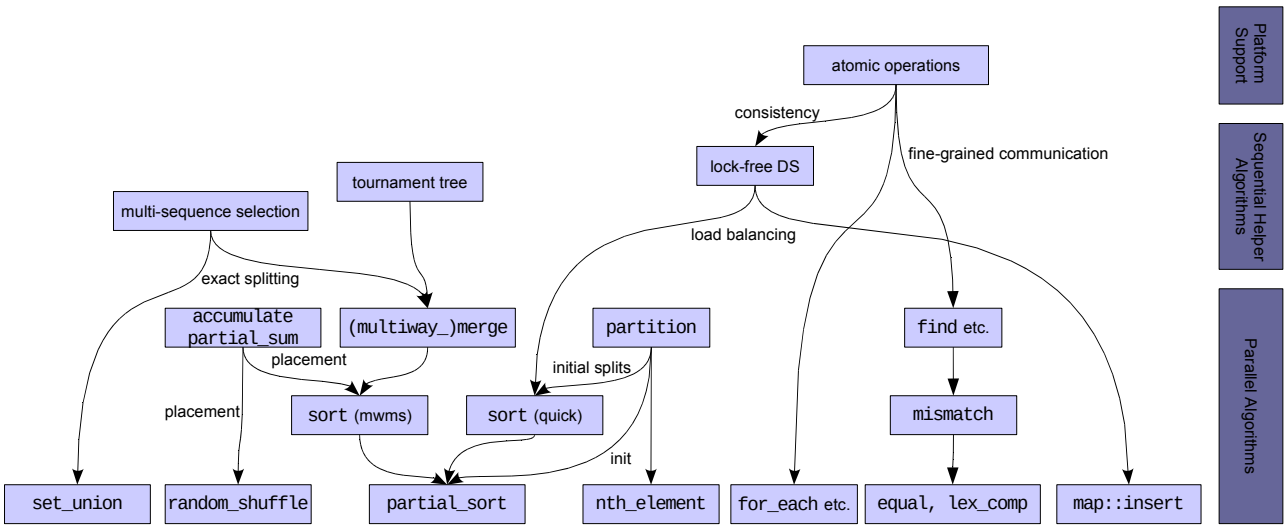

Figure 2: Code reuse inside the parallel mode.

than sequential ones. For example, the one-liner describing `for_each` explodes into a parallel version of many hundred lines, implementing complex load balancing mechanisms like work-stealing.

On the one hand, this fact makes the usage of a library even more valuable, since you do not have to write that code yourself. On the other hand, the library itself is threatened by hard-to-maintain complexity. For this reason, the library programmers must make efforts to keep the code as maintainable as possible, without diminishing benefits. Two important aspects in that are fostering code reuse, and avoiding code duplication.

Code duplication is particularly tempting for the STL, since it contains many algorithm calls that act similarly. `for_each` and (unary) `transform`, for example, only differ in the fact that the functor passed to `transform` is allowed (even supposed) to modify its input in-place. There are several other similar calls, basically all of them being "embarrassingly parallel", traversing one or two input sequences. One could write the code for `for_each`, copy and paste it several times, and change some little details. This is feasible for the sequential versions, but unmaintainable for the parallel ones. A bug fix or an algorithmic improvement would have to be applied in many places, making the process extremely error-prone.

In particular the algorithmic portions, must be reused already by the library itself. For a family of algorithms, a most general version of the parallel algorithm is factored out, and subsequently called by all members of the family. The surplus arguments must be filled with dummies, e. g. noop-functors and dummy iterators. Similar action is taken for default functors. For example, the sequential implementation provides two completely independent implementations for both versions of `accumulate`, one having explicitly specified the addition functor, the other taking the default `operator+` for the respective type. For the complex parallel version, the implicit version must be redirected to the explicit one by filling in an equivalent functor as soon as possible (see Figure 3), to avoid code duplication. We have to rely on the compiler's ability to inline simple operators, and to optimize away empty functions. Otherwise, the performance penalty will be considerable. Compiling without optimizations will therefore impede the parallel implementation to a greater extent than the sequential one.

2.5 Extensions to the Standard Library

There are some algorithms that are a sensible extensions of the standard library. One of them is the generalization of `merge` to accept multiple sequences, instead of only two. This is internally used for the mergesort implementation, therefore called multiway mergesort. Of course, multiple sequences can be merged together one after the other, but this is less efficient, not in asymptotic running time, but due to additional accesses to memory. So we pass on this functionality to the programmer under the name `multiway_merge`.

Other functions are helpers for exploiting parallelism in higher levels. This includes the simple, but error-prone function for splitting a range into parts of about the same size (plus/minus 1). Much more complex is the routine for selecting the element of a certain global rank, from several sorted sequences. Both routines are used intensively by the parallel mode internals: it is only natural to make them accessible to the programmer. One approach would be to document the calls in `__gnu_parallel`, and attempt to then standardize most-used functionality in `namespace std` for some future version of C++.

2.6 Sequential Performance

Since parallelization is about performance, the question arises whether the existing sequential algorithms are sufficiently optimized. An asymptotic penalty in running time is accepted for integer sorting, since the STL provides only comparison-based sorting, taking $\mathcal{O}(n \log n)$ time. Specialized integer sorting algorithms take only $\mathcal{O}(n)$ time for this task. We could specialize the `sort` routine for all kinds of integer elements, to support at least ascending and

descending order. However, the library user usually wants to sort a more complex object including an integer key. Thus, the interface would have to be changed quite radically. Not a comparison functor would have to be passed, but a functor extracting the key from the element. This could lead to a syntax similar to the upcoming hash maps.

2.7 Algorithm Variants

For some functions, the parallel mode offers multiple algorithmic variants to choose from, in order to allow best execution speed in a variance of circumstances. This is particularly sophisticated for sorting, which comes in three variants. This sounds a lot, but also counting `stable_sort`, the sequential implementation also has already two completely independent versions.

The parallel mode features two kinds of parallel quicksort, unbalanced and balanced, as well as a the multiway mergesort. The quicksort variants work completely in place, and are quite small in executable size (see Table 1). On the downside, they do not support stable sort, and they require nested parallel regions.

Multiway mergesort, however, provides stability, and does not use nested parallelism. It requires a temporary copy of the data, though. Experiments show that it scales better down in terms of running time, i. e. for small inputs.

While the mergesort can give stronger theoretical guarantees on the running time, the quicksort variants theoretically may not scale at all, in the unlikely worst case. In practice, though, the balanced variant can better handle the case of cores being unexpectedly unavailable because of other processes.

2.8 Code and Binary Size, Compilation Time

Generic programming trades off executable size for optimal adaptation to and integration of the data type, resulting in best performance. Instantiating an algorithm for each data type separately allows high-level programming without any run-time overhead, which could be implied by calling virtual functions or similar run-time decisions on what code to execute next. The increase in executable size is generally accepted. If code size really matters, you can still derive all used data types from a base class with virtualized functions, and instantiating the algorithm for the base class.

However, for the parallel mode, the algorithms are more complex, and hence, the problem gets more important, because of more complex code and variants. Therefore, the code size for the algorithms grow up to a factor of 4 compared to the sequential code. This is shown in Table 1, for the program in Figure 3:

For `TAG`, the corresponding tags are inserted, in order to choose the algorithm variant. The code was compiled using the options `-O3 -g0 -fopenmp -D_GLIBCXX_PARALLEL -DNDEBUG` with a prerelease version of GCC 4.3, running on a x86_64 machine.

The compilation time also differs, it approximately scales with the size of the resulting executable. This is mostly due to the optimization passes, not the parsing, as can be seen by comparing the timings for the debug mode.

As a conclusion from these measurements, it can be stated that including all variants for a run-time choice is to be avoided. The programmer should be empowered to choose at compile time, and the default to include only the parallel algorithm forming the best choice.

```
#include <vector>
#include <algorithm>
int main() {
  std::vector<int> vi(100000000);
  std::sort(vi.begin(),vi.end() TAG); return 0;}

sort(begin,end)
 sort(begin,end,comp=std::less<int>())
  parallel_sort(begin,end,comp,stable=false)
   1. parallel_sort_mwms(...,stable);
   2. parallel_sort_qs(...);
   3. parallel_sort_qsb(...);
    respective OpenMP-parallelized code
```

Figure 3: Minimal sorting example, and call stack for "runtime variant choice", i. e. TAG="" with the three possible runtime outcomes.

Algorithm Variant(s)	Size	Time
Sequential	15 479	0.74
Quicksort	22 387	1.49
Balanced Quicksort	26 989	1.84
Multiway-Mergesort Sampling	36 002	3.49
Default (Multiway-Mergesort Exact)	41 229	4.68
Multiway-Mergesort Exact	41 237	4.78
Multiway-Mergesort (splitting choice at run-time)	46 003	5.48
All Parallel Variants (run-time choice)	61 543	6.50

Table 1: Executable sizes (in bytes) and compilation time (in seconds) for different algorithms variants.

2.9 C++ Standard Compliance

Strict 100% compliance to the C++ standard makes parallelization virtually impossible. This is because the standard was designed for serial use, without having parallel processing in mind at all.

The algorithm descriptions often do not specify only a input/output relation, but do give code examples the operation is "equivalent" to. This implies, for example, the order in which the elements are processed in `for_each`, and thus would ultimately forbid parallelization.

2.9.1 Argument-Dependent Lookup

A problem arising from the strong namespace inclusion is the failure of the so-called argument-dependent lookup (also called Koenig lookup), concerning unqualified function names. This quite exceptional but important C++ rule [4, Section 3.4.2] makes the compiler look up the function name in all namespaces the function arguments' types come from.

Having removed `std::` in front of `sort` in the code from Figure 3, the compiler still looks up `sort` in the `std` namespace, because the type of at least one argument is from `std`. However, compilation fails in parallel mode because associated namespaces are not looked up. This problem can be blamed on a halfhearted specification of namespace association declaration, and could be solved by extending it reasonably to such lookups.

2.9.2 Functors

The library calls take many kinds of encapsulated user code as arguments.

- Explicit functors to process/transform an element, making up the main functionality of the algorithm. The functor object might contain "state" information. Examples: functors for `for_each`
- Implicit methods and functions, inherent to the value type. Examples: Overloaded (copy) constructors and assignment operators
- Implicit methods and functions, inherent to the iterator type. Examples: Increment operator (`operator++`), dereference operator (`operator*`), difference operator (`operator-`), default constructor
- Stateful helper functions. Example: random number generator for `random_shuffle`.
- (Hopefully) stateless helper functions. Example: Comparison functors, e. g. used by `sort`.

All this user code might be able to interact with the environment in an non-thread-safe way through side effects. These are most legitimate for the first class of functors. However, for some routines, the standard states that they "shall not have any side effects", namely `transform` [4, Section 25.2.3] and the numerical algorithms `accumulate`, `inner_product`, `partial_sum`, and `adjacent_difference`.

For the other classes, they usually imply a bad programming style, or an abuse of functionality.

The extensive code re-use, justified in Section 2.4, yields another detail problem. Functors have to be passed to subroutines (analogous to the call stack in Figure 3), and the question arises whether this should by done by value or by reference. The common prototypes of the algorithm calls take the functors by value. This prohibits returning any "state" to the caller, which is good news. To avoid copying the same thing multiple times, one should pass on *references* to subroutines.

Another hitch stemming from more complex algorithms is the use of user data types with unfortunate properties. For example, many algorithms have to copy elements from the input to some temporary storage. However, the library may not assume that the given data type is default-constructable, so the temporary storage cannot be easily allocated. The algorithms never produce "new" elements, so we can always circumvent this problem. One option is to use pointers to the existing elements, the other is to reserve raw memory and to construct the element using placement new, calling the copy constructor. In the latter case, you have to be extremely careful in order not to construct multiple objects in the sample place, without calling the destructor for the former ones.

Determining the return type of a functor is a rather minor problem. The most elegant way for the programmer to declare a functor, is to (indirectly) derive from a standard base class like `unary_function` or `binary_function`. These base classes take the argument type and the return type as template parameters, so they can be read out from the type members `(first)_argument_type`, `second_argument_type`, and `result_type`. However, it is more compatible not to rely on the types being defined appropriately. For the return type, there is an alternative, namely applying the `typeof` keyword to a "fake" function call. This function call is in fact never executed, but just serves to find out the return type: `typedef typeof(functor(input)) return_type;`. The return type is needed for `inner_product`, where the type of the intermediate values (after multiplying) need not coincide with the return type. Example:

```
struct A;
bool operator*()(const A& a1, const A& a2)
  { ... }
void foo() {
  A* S1 = new A[100], * S2 = new A[100];
  double c =
    std::inner_product(a, a+100, b, 0U);
}
```

The return type `bool` of `operator*` is needed in order to pass on a regular functor to the "embarrassingly parallel" infrastructure, instead of just using the `+` and `*` operators.

2.9.3 Exception Safety

Two things that avoid each other like the plague are parallelism and exceptions. However, there is no principal reason for this.

The problem surfaces when a routine in a thread throws an exception which is not caught inside the thread. When the parallelization is done explicitly, this situation can always be avoided by doing just that. The further consequences on the program execution can be handled then, e. g. forcing the other threads to terminate, or to somehow sensibly resume execution.[4]

For a library which takes functors, such a specific solution is not applicable, and the search for a more robust solution is ongoing. Some general restrictions should apply, as follows. First, the algorithms themselves should not throw any exceptions. This is easy to achieve since "wrong" input is not possible to happen, in most cases. As an exception, some algorithms rely on the given sequences being sorted. However, not even the sequential versions check for this precondition, and may deliver wrong results, may not terminate, or may crash the whole program. For all algorithms taking a comparison functor, inconsistent return values (failing to be a strict weak ordering) can lead to undefined behavior, just as in the sequential case, again.

So, exceptions have to be expected only from user-defined code, as listed in Section 2.9.2. But what is a sensible reaction to an exception being thrown? Probably, rethrowing the exception after the clean termination of the algorithm is a reasonable choice. All involved threads should be stopped as soon as possible. If more than one thread throws[5] before the execution can be stopped, one of the exceptions would have to be selected in a sensible way. Unfortunately, this is all nice talk, but the current C++ does not allow this kind of reaction [14]. A `catch` clause can either specify an exception type to catch, or catch all exceptions, as marked by an ellipsis (`...`). In the first case, the type of the exception is known, so it could be rethrown. In the second case, however, the exception object is completely inaccessible, so rethrowing it is impossible. And finally, the exceptions to expect are not known to the library. Throwing some general exception in place of the original one would be an work-around.

[4]This is an area that is not detailed in the current C++ standard, and still under development in C++0x: some C++ implementations allow a "per-thread" exception handler, and others do not. Lack of portability has hampered experimentation in this area.

[5]This is particularly likely to happen for out-of-memory-related exceptions.

The only and best way to handle this would probably be to extend OpenMP to a defined handling of exceptions. So far, throwing exceptions over parallel region boundaries is just forbidden.

Little has been published about solving this problem, so far. One paper [8] only considers terminating the program in a standard-compliant way, not giving any thoughts to resuming it sensibly. To our knowledge, the only proposal for handling exceptions in the context of OpenMP, is in fact a paper proposing integration of OpenMP into Java [9, Section 4.3]. Their suggestion is well reasoned. After the occurrence of an exception, the threads should terminate as soon as possible. To achieve this, the catching thread sets a cancellation flag, which is read out by the other threads at certain cancellation points, namely `barrier`, `critical`, `parallel`, and all OpenMP work-sharing directives. This approach could lead to an effective exception treatment in the parallel mode.

2.9.4 Tag Parameters

In order to allow compile time algorithm choices, be it only forcing sequential execution, the function prototypes must be augmented by additional "tag" parameters. This makes some of the standard compliance tests fail just because of a unknown method signature. For user code, this should not make any difference, since all standard-compliant calls will still succeed.

2.9.5 Conclusion on Standard Compliance

So far, the libstdc++ parallel mode works with "reasonably" programmed code. This includes assumptions as side-effect-safe and exception-free functors passed to the parallelized algorithms.

Some of the problems could be avoided by changing the library integration, e. g. the argument-dependent lookup, making it less elegant, though. Others are inherent to parallel computing, like side-effect issues and exceptions. Those could be mitigated by providing thread-safe containers, as provided by the Intel Threading Building Blocks [2].

2.10 Benefits from Upcoming C++ Standard

The upcoming next version of the C++ standard will provide functionality that comes in particularly handy for the parallel mode.

The easy availability and standardization of *thread-local storage* could simplify the parallel mode code.

Concepts and *static assertions* simplify checking for preconditions, e. g. an iterator being random-accessible. *Lambda functions* allow for more elegant definition of functors, thereby making the usage of STL algorithms easier and more intuitive.

So-called *rvalue references* allow to avoid unnecessary copying of (potentially large and complex) objects. Instead, they can be "moved" explicitly, implying that the data must not be accessed in the original location any longer. By providing a specific *move constructor*, the programmer can thus make moving much more efficient than the combination of copy construction/assignment and deletion. This helps the STL algorithms in general, because many of them actually only move data, e. g. `sort` and `random_shuffle`. In particular, the parallel mode profits because copying data loads the memory connection, which is a common bottleneck for parallel speedup.

2.11 Benefits from Upcoming OpenMP

Although the OpenMP standard version 3.0 is not yet final, some of the proposed innovations by the draft look promising.

The standard draft specifies the `parallel for` loop to work also on C++ random access iterators, in addition to integer loop variables. This supplements the `for_each` functionality. However, no scheduling providing a worst-case guarantee on running time is demanded.

A major addition is the `task` construct, which allows (recursive) asynchronous processing of code blocks, scheduled by the OpenMP runtime. This could greatly simplify the code for recursive routines like the quicksort variants, and avoid the nested parallelism drawback. The use of the platform-specific `yield` call could also be made obsolete by this construct. The yield call proposes to the OS to give control to some other thread, and is usually called when out of work, in order to avoid other threads to wait unnecessarily long. It is most needed when there are more threads scheduled than cores (currently) available. A publication actually proposed to include an equivalent directive into OpenMP [15].

Unfortunately, OpenMP 3.0 is unlikely to improve on the field of atomic operations. Simple atomic arithmetic routines are provided, but none of those is equivalently powerful as `fetch-and-add` or `compare-and-swap`. Also, nothing new is said about exception handling.

2.12 Testing

Testing the libstdc++ parallel mode must cover various aspects. The test suite accompanying the libstdc++ checks most standard compliance issues. Currently, 99% of the tests succeed, the failing are explained by Section 2.9.4.

However, the functionality and conformance checks are often limited to very simple, hand-written examples. This is not sufficient for the parallel implementation, since many of its subroutines have fallbacks for very small inputs, e. g. a sequence length smaller than the desired thread number. Thus, there should be some improvement in this direction. Larger inputs could be generated on the fly, comparing the results of the sequential and the parallel implementation for accordance.

The confidence in the correctness of the algorithms are based on the test program available from the original MCSTL, so far. Integrating this program as a whole into the libstdc++ test suite is questionable because of the quite different paradigms used. So, the best compromise is to integrate its test cases into the existing testing framework.

2.13 Performance Aspects

To what extent should an implementation of a general purpose standard library should aim for best execution speed? On the one hand, such an implementation may be used by very many people, which makes optimization worthwhile. On the other hand, such a library must be very robust and general, which impedes finely-tuned optimization. By definition, hand-written code can always be as least as good as a library because nobody keeps you from doing the same thing as the library does. However, the genericity is gone, and thus it will be too much of an effort to optimize all algorithm instantiations of a program

to this extent. Such solutions tend to be fragile, and require constant maintenance.

The STL implementations are mostly asymptotically efficient, losing only a constant factor to an optimized version. Things are different for parallelization. Under the reasonable assumption that the number of cores in a computer will continue to increase, a sequential implementation of the STL will lose a *growing* factor compared to the parallelized one. This is what makes the parallelization so essential.

In terms of a widely-available highly-used algorithm and data structures library, there is actually not much competition to the STL. LEDA [10] covers much more functionality, but lags behind the development of C++, and as a result does not fully exploit its generic programming capabilities. In addition, an excessive usage of virtual functions leads to bad performance for many cases. A very nice example for a library that pulls out all the stops of C++, is the Computational Geometry Algorithms Library (CGAL) [6]. However, it is limited to geometry and internally re-uses much of the STL. The Boost libraries try to extend the standard library with additional functionality, and many are prospected for inclusion into upcoming versions of the standard.

2.14 Heuristics, Tuning

Even if parallel code is included into the executable, this does not mean that the algorithm is *always* executed in parallel. Heuristic considerations might cause a fallback to the sequential version at runtime, the most obvious case being that there is only one processor core available. Also, the input size and the data type is taken into account to decide whether a function call should be executed in parallel. The corresponding thresholds can be configured at run-time.

Also, in several cases, tuning parameters allow to trade off granularity of load-balancing against synchronization overheads. Load-balancing work in a fine-grained way shows best worst-case performance, but can add too much synchronization overhead to be worthwhile.

2.15 Interaction with Other Libraries

We have integrated the libstdc++ parallel mode with another library akin, namely the STXXL [5]. The STXXL provides functionality alike the STL with respect to external memory, i. e. data on hard disks. Since the library can load several parallel disks very efficiently, it is often actually compute-bound. Using the parallel mode for the internal memory operations results in significant speedups for various applications.

Another natural candidate for using the parallel mode would be CGAL.

3. CONCLUSION AND FUTURE WORK

In this paper, we have presented the transparent integration of parallel algorithm implementations into the libstdc++. We have proposed solutions for most of the arisen issues, many of them already being implemented. For the others, we show starting points.

One of the hardest remaining problems is the decision of whether to switch to parallel execution or not. This is crucial for the resulting program performance in the worst case, since the parallelization overhead cancels out speedups for "to easy" problems, i. e. algorithm calls that have to little or too local data for communication of the data being worthwhile. By testing and profiling, the application programmer can discover break-even points and tuning parameters for a specific algorithm on a specific data type, a specific functors, and so on. However, this is tedious and does not adapt to several platforms at the same time. So, developing methods for auto-tuning the algorithms are underway. Even with the noted deficiencies, the parallel mode user can already work on one level higher than before.

3.1 Acknowledgments

We would like to thank Ulrich Drepper, Paolo Carlini, Jakub Jelinek, and Wolfgang Bangerth for pointing out problems, and valuable discussion.

4. REFERENCES

[1] GCC documentation on namespace association. `http://gcc.gnu.org/onlinedocs/gcc/Namespace-Association.html`.

[2] Intel Threading Building Blocks website. `http://osstbb.intel.com/`.

[3] Working draft, standard for programming language C++. `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2315.pdf`.

[4] *The C++ Standard (ISO 14882:2003)*. 2003.

[5] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for XXL data sets. *Software: Practice and Experience*, August 2007.

[6] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report TR 291, ETH Zürich, 1998.

[7] Leonor Frias and Johannes Singler. Single-pass list partitioning. In *International Workshop on Multi-Core Computing Systems (MuCoCoS) 2008*.

[8] Shi-Jung Kao. Managing C++ OpenMP code and its exception handling. In *WOMPAT*, pages 227–243, 2002.

[9] Michael Klemm, Ronald Veldema, Matthias Bezold, and Michael Philippsen. A proposal for OpenMP for Java. In *IWOMP*, 2006.

[10] Kurt Mehlhorn and Stefan Näher. *LEDA – A platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[11] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 2.5*, May 2005.

[12] P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library*. Prentice-Hall, 2000.

[13] Johannes Singler, Peter Sanders, and Felix Putze. The Multi-Core Standard Template Library. In *Euro-Par 2007 Parallel Processing*.

[14] Michael Süß. Exceptions and OpenMP - an experiment with current compilers. `http://www.thinkingparallel.com/2007/03/02/exceptions-and-openmp-an-experiment-with-current-compilers/`.

[15] Alexander Wirz, Michael Süß, and Claudia Leopold. A comparison of task pool variants in OpenMP and a proposal for a solution to the busy waiting problem. In *IWOMP*, 2006.