Optimizing Task Parallelism with Library-Semantics-Aware Compilation

Peter Thoman, Stefan Moosbrugger, and Thomas Fahringer

University of Innsbruck, Austria
{petert,stefanm,tf}@dps.uibk.ac.at

Abstract. With the spread of parallel architectures throughout all areas of computing, task-based parallelism is an increasingly commonly employed programming paradigm, due to its ease of use and potential scalability. Since C++11, the ISO C++ language standard library includes support for task parallelism. However, existing research and implementation work in task parallelism relies almost exclusively on runtime systems for achieving performance and scalability. We propose a combined compiler and runtime system approach that is aware of the parallel semantics of the C++11 standard library functions, and therefore capable of statically analyzing and optimizing their implementation, as well as automatically providing scheduling hints to the runtime system.
We have implemented this approach in an existing compiler and demonstrate its effectiveness by carrying out an empirical study across 9 task-parallel benchmarks. On a 32-core system, our method is, on average, 11.7 times faster than the best result for Clang and GCC C++11 library implementations, and 4.1 times faster than an OpenMP baseline.

1 Introduction

Task-based parallelism is one of the most fundamental parallel abstractions in common use today [1], with applications in areas ranging from embedded systems, over user-facing productivity software, to high performance computing clusters. In all of these fields, the C++ programming language is one of the first choices for performance-sensitive applications.

The C++11 standard, which is now implemented in all the most widely-used C++ compilers, introduced several parallelism-related functions and classes in the standard library. One of the most interesting of these from both the perspective of an application developer and a library implementation is the `async` function template. It has the potential to express both coarse- and fine-grained task parallelism, and can serve as a building block for more complex and feature-rich parallel patterns.

While relatively easy to implement and use, achieving good efficiency with task parallelism can be challenging not only for application developers but also for runtime systems, particularly in the case of fine-grained tasks [3]. The *granularity* of tasks is defined by the length of the execution time of a single task between interactions with the runtime system, such as spawning new tasks. It has recently been demonstrated that the performance of fine-grained task-parallel programs written in C++11 is insufficient in all mainstream compilers and standard libraries [4].

In order to achieve high performance with fine-grained tasks, the overhead of interactions with the runtime system needs to be minimized, and both task

distribution and communication need to be implemented in a scalable and efficient fashion. Previous work in this area has focused mostly on new libraries, dynamic optimization at runtime, or user-controlled tuning parameters. Conversely, we propose an approach that combines a *library-semantics-aware optimizing compiler* with a high-performance runtime system which is statically tuned by leveraging knowledge analytically derived at the compiler level. Our goal is to maximize the efficiency of task execution without requiring any additional effort or systems-level knowledge on part of the application programmer, and without introducing any tuning overhead at runtime.

We implemented our method within the Insieme compiler and runtime system [5], but its principles are equally applicable in any other framework. Our concrete contributions are the following:

- A library-semantics-aware compilation process, in which an existing compiler is enriched with the capability to comprehend C++11 standard library semantics, and thus recognize, analyze and optimize task-parallel programs written using these libraries.
- A set of analyses which statically determine several performance-relevant properties of task-parallel code regions, and a heuristic which automatically tunes various runtime system parameters based on these properties.
- An implementation of our approach within the Insieme compiler and runtime system.
- Evaluation and analysis of the performance of our method on a set of 9 task-parallel benchmarks. We compare to existing C++11 implementations, as well as OpenMP versions of the benchmarks in order to provide a more optimized and mature performance baseline.

The remainder of this paper is structured as follows. In Section 2 we discuss some initial results that motivated our work. We then describe our library-semantics-aware compilation method in detail in Section 3, and our static analyses as well as the tuning heuristics derived from them in Section 4. The performance of our implementation is evaluated in Section 5, followed by an overview of related work in Section 6. Finally, Section 7 summarizes and concludes our findings.

2 Motivation

Our primary motivation for this work is the desire to be able to employ C++11 threading constructs as building blocks for task parallel programs. Clearly, this approach should offer significant advantages over third-party and homegrown solutions: it is easier to teach and read, thereby increasing programmer productivity, it can be more closely integrated and supported within a given compiler and its associated runtime library, thereby potentially offering superior performance, and it is portable to any standard-conformant implementation of C++ without external dependencies.

However, the primary reason for parallelization is generally the desire to improve program performance. As Figure 1 illustrates, both the performance and scalability of state-of-the-art C++11 compilers and runtime systems is insufficient to serve as a replacement for existing parallel languages. The figure depicts

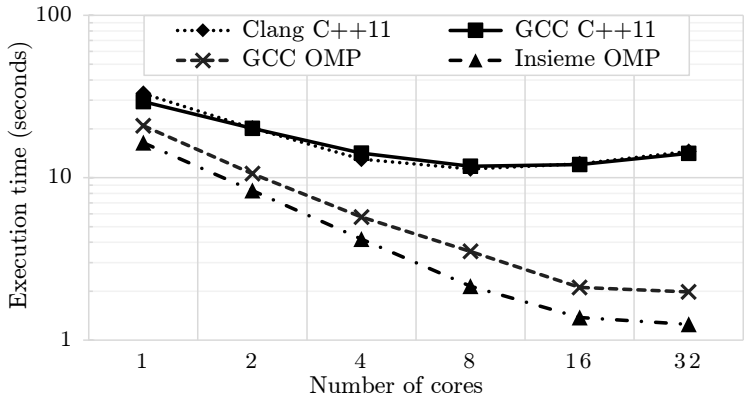

Fig. 1: Performance of the *pyramids* benchmark across APIs and compilers

the execution time over varying degrees of parallelism for the *pyramids* benchmark from the INNCABS [4] C++11 benchmark suite, as well as an OpenMP implementation of the same benchmark provided for reference. The hardware and software setup for this test is the same as used for the evaluation in Section 5, where it is described in detail. At the maximum degree of parallelism of 32, the production-ready OpenMP implementation of GCC outperforms the C++11 versions generated by both GCC with libstdc++ and Clang with libc++ by a *factor of 7*, and the research OpenMP implementation in Insieme is a full order of magnitude faster.

While some degree of improvement of the C++11 results could be achieved purely at the library level, we believe that providing high efficiency rivaling existing parallel languages over several distinct task-parallel patterns without the overhead of runtime tuning requires the co-operation of a library-semantics-aware compiler with a high-performance runtime system.

3 Semantics-aware Compilation

A fundamental issue with effectively implementing parallelism in mainstream compilers and languages is that it is often expressed by means of library function calls, opaque to the compiler and thus impossible for it to optimize. Furthermore, even parallelism expressed at the language (extension) level – e.g. using OpenMP constructs – is usually translated to internal library calls [6] before reaching the main compiler intermediate representation (IR), once again rendering important semantic information inaccessible to the compiler.

The Insieme source-to-source compiler is based on the INSPIRE intermediate representation, which is designed to inherently support unified parallel language semantics. It has been successfully employed in OpenMP [7], Cilk [8], and OpenCL [9] compilation. We lack the space to detail INSPIRE semantics in this paper, and point interested readers instead to the summary provided by Jordan et al. [10].

In order to enable semantics-aware compilation, analysis, and optimization of C++11 task-parallel programs, we have extended the Insieme C++ frontend to i) identify relevant C++11 thread support library calls and data types, ii) an-

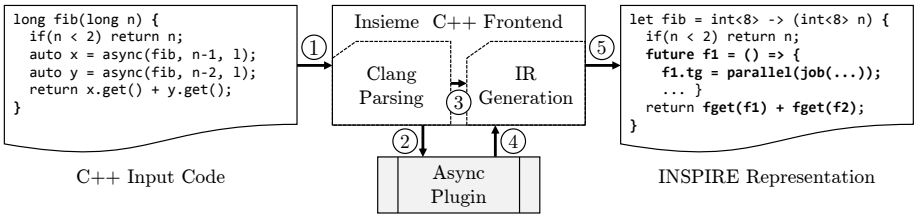

Fig. 2: Semantics-aware frontend conversion of library calls to INSPIRE

alyze their suitability for direct semantic translation, and iii) translate them to appropriate INSPIRE constructs.

Figure 2 provides a simplified overview of this conversion process, which we will now describe in more detail. The Insieme C++ frontend is based on Clang [11] and features a plugin system allowing multiple entry points for custom INSPIRE generation. For this work, we have created a *C++11 Async plugin*, resulting in the following frontend conversion process:

① The input program is parsed by Clang.
② For every language construct encountered, the Async plugin is invoked.
③ In case of the vast majority of language constructs, the plugin ignores them and they are passed directly to the default IR generation phase.
④ However, the relevant subset of suitable library calls and data structures are intercepted and converted appropriately, as detailed below.
⑤ Finally, the full INSPIRE representation including a semantically equivalent implementation of the library functions is generated.

Table 1 lists the most relevant subset of C++11 library functions and types the Async plugin acts upon, as well as their INSPIRE equivalent. Several implementation details – such as the management of the `valid` state of each `future` – are omitted for brevity. The same is true for the `future::wait` operation, as it is simply equivalent to a `future::get` operation ignoring its return value.

Focusing on the essentials, the conversion is relatively straightforward. Future type templates are converted to structures comprising the return value (of automatically deduced type `'a`) and a `threadgroup`, which is the fundamental INSPIRE type allowing operations on an asynchronously executing process. Async calls are converted to a call to a function which takes an arbitrary closure `() => 'a f` as its argument and returns a pointer to a future `ref<ref<future>>`. It allocates the new future structure on the heap, launches a new parallel job executing the closure `f` and storing its result in the future structure, and stores the result of this parallel call – a `threadgroup` – in the future structure as well. Finally, it returns a pointer to this new future structure. When `get` is invoked on a future, its associated threadgroup is first `merge`d to ensure that it has completed, the return value is stored, and the heap allocation for the future structure is freed.

The crucial feature of this conversion process is that, after it has completed, the entire parallel program semantics are expressed in pure INSPIRE. This uniformity allows the compiler core to perform analysis as it would on e.g. an OpenMP, Cilk or OpenCL program. Furthermore, it enables the compiler back-

Table 1: Semantic mapping of standard library constructs

C++11	INSPIRE
`future<T>`	`let future = struct { 'a result; threadgroup tg; }`

`async(f)`	`(() => 'a f) -> ref<ref<future>> {`
	`  ref<ref<future>> x = var.new(future);`
	`  (*x)->tg = parallel( job { (*x)->result = f(); } );`
	`  return x; }`

`future::get()`	`(ref<ref<future>> f) -> 'a {`
	`  merge((*f)->tg);`
	`  auto var = (*f)->result;`
	`  ref.delete(*f);`
	`  return var; }`

end to generate code targeting the highly optimized Insieme runtime system, instead of relying on the implementation provided by a given C++11 standard library.

One important prerequisite during the conversion of `async` calls is checking the specification of the `std::launch` parameter. Our semantics-aware compilation applies if and only if this parameter is either i) not supplied, thereby leaving the choice up to the compiler, or ii) supplied and set to to `async | deferred`. Other cases, that is settings of exclusively `async` or exclusively `deferred`, prescribe the desired behavior exactly, and leave little room for compiler- and runtime-level optimization. Therefore, the Async plugin forwards those cases directly to the default IR generation phase, maintaining their correctness.

4 Static Optimization and Compiler-assisted Tuning

Library-semantics-aware compilation as described up to now is quite useful in and of itself, as it allows C++11 programs to automatically benefit from all backend and runtime optimization work carried out for any other parallel language compiled to INSPIRE. However, its full set of advantages can only be leveraged in combination with static compiler-level optimization and analysis.

In this section, we will discuss both *static optimization*, which is always attempted by the compiler and invariably improves performance when applicable, as well as *feature analysis and tuning*, whereby compiler analysis is used to derive code features which determine runtime tuning parameters according to some heuristics.

Static Optimization Listing 1 depicts a common pattern of async and future usage in parallel programs. While this particular example is highly simplified, the underlying pattern of launching a set of asynchronous tasks, and then waiting for their completion before returning from the current task is exceedingly common in real-world task-parallel applications, including most instances of divide-and-conquer and branch-and-bound algorithms. In fact, Cilk semantics – the original template for task-parallel programming – strictly proscribe this behavior.

Listing 1: Common pattern of `async` and `future` usage

```
let mmul = (int l, int r, int t int b, ...) -> unit {
  auto f1 = async(mmul(l, l+(r-l)/2, t, t+(b-t)/2, ...));
  auto f2 = async(mmul(l+(r-l)/2, r, t, t+(b-t)/2, ...));
  auto f3 = async(mmul(l, l+(r-l)/2, t+(b-t)/2, b, ...));
  auto f4 = async(mmul(l+(r-l)/2, r, t+(b-t)/2, b, ...));
  ...
  f1.wait(); f2.wait(); f3.wait(); f4.wait();
  ...
}
```

The observation that this type of synchronization pattern is common is interesting from an optimization perspective, as synchronizing on the completion of all active child tasks can generally be implemented much more efficiently in a given parallel runtime library than waiting for each of them individually. Therefore, we have created a static optimization we call *synchronization coalescing* to optimize this type of pattern.

Algorithm 1 describes the synchronization coalescing transformation. First, on line 1 to 4, it is ensured that no `threadgroup` object passes out of the current task, as this might allow unknown synchronization and access patterns. This means that e.g. futures stored in global variables or moved outside the function cannot be optimized, but in practice we have not found this to be a significant limitation so far.

From line 5 to 12, all possible static control paths to `merge` calls are examined to ensure that the expected synchronization pattern is maintained. As this check is done on static control paths, repeated `parallel`/`merge` invocations within a loop are not optimized, but the common idiom of first launching a set of tasks in a loop and then waiting on their results in a new loop is captured.

If neither of the two safety checks prevents the optimization, starting from line 13 the code transformation is performed.

It is important to note that the actual implementation of this transformation benefits from the semantics-aware translation of library calls to the unified and inherently parallel INSPIRE representation in several important ways:

Algorithm 1 Synchronization coalescing

	T input/output task function
1:	Determine the set P of all `parallel` invocations in T.
2:	**for all** `parallel` $p \in P$ **do**
3:	**if** the $threadgroup = p()$ can pass outside T **then return**
4:	**end for**
5:	Determine the set M of all `merge` invocations in T.
6:	**for all** `merge` $m \in M$ **do**
7:	Compute the set of all static execution paths F from the entry point of T to m.
8:	**for all** paths $f \in F$ **do**
9:	Reverse f and remove the first entry.
10:	**end for**
11:	**if** $\exists f \in F : f$ encounters a `merge` after a `parallel` **then return**
12:	**end for**
13:	Insert `merge_all` before the lexicographically first $m \in M$.
14:	Remove all $m \in M$ from T.

1. There is no need to deal with slightly different variants of the same underlying operation individually – e.g. it is sufficient to process only `merge` calls rather than `future::get` *and* `future::wait` invocations, as both of these map to INSPIRE functions internally calling `merge`.
2. Existing tools for the analysis of parallel control and data flow in Insieme can be re-used directly, e.g. in the implementation of the safety checks, without requiring specific adaptation for C++11 `async`.
3. The resulting optimization is equally available and applicable to any other input language or library generating INSPIRE.

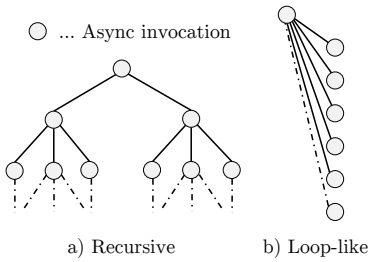

Fig. 3: Parallel patterns

Parameter	Possible Values
Push position	$P =$ {front, back}
Queue length	$L = 2^n$ $n \in \mathrm{N}, n > 1$
Meaningful choices	$8, 16, 32, 64$

Table 2: Runtime system settings

Feature Analysis and Tuning As task parallelism is a versatile abstraction, it can model a variety of parallel patterns. Among those, two highly relevant ones for runtime system optimization are *recursive parallelism* and *loop-like parallelism*, both of which are illustrated in Figure 3. The former occurs e.g. in divide-and-conquer and branch-and-bound algorithms, while the latter is common whenever lists or arrays are processed. The crucial difference between the two, which directly affects how they are most efficiently executed, is the fact that in recursive parallelism each task generally generates further sub-tasks, while this is not the case for loop-like parallelism.

Many task-parallel runtime systems offer tuning options, which can significantly influence the achieved performance. The same is true for the Insieme runtime system we employ. Two of its most relevant settings are listed in Table 2: *push position* and *queue length*. These describe, respectively, whether newly generated tasks are inserted at the front or the back of each work queue, and the number of full parallel tasks which will be generated before falling back to sequential execution (lazy task creation).

These settings relate directly to the differences between recursive and loop parallelism: as recursively parallel tasks generate new tasks, long queues are not necessary to maintain good utilization, and newly generated tasks should be inserted at the back of the queue so that other workers have a chance to first steal large blocks of work (further up in the task tree). Conversely, for loop-like parallelism, longer queues are desireable to maintain enough available tasks for all workers to be utilized effectively, and new tasks should be inserted at the front of the queue to maintain cache locality on the local worker.

In a conventional runtime system or parallel library, these settings need to be taken care of by cautious selection of defaults, or, at best, by studying the behavior of the application at execution time and gradually converging towards an optimum. With library-semantics-aware compilation, we are able to classify ap-

plications at compile time by means of static analysis, and automatically choose appropriate runtime system settings based on this classification.

Currently, our classification is based on two relatively simple analyses: i) a *recursion check* which determines whether a task function may invoke itself recursively, and ii) a *loop check* which investigates the invocation context of a given parallel call to find out whether it occurs within any loop structure. Describing these inter-procedural analyses in detail is not possible within the constraints of this paper, but they are actually relatively simple to accomplish within the Insieme infrastructure.

Based on the result of these analyses, classification is trivial:

1. if *recursion check* succeeded, classify as recursive, P = back and $L = 8$;
2. else, if *loop check* succeeded, classify as loop-like, P = front and $L = 64$;
3. else, use the defaults (P = front and $L = 32$).

While the arguments for the choice of P and the relative queue length for each category were outlined above, the question for best choice of absolute value for L has not been fully solved. Our current selection for each category is based on empiric experience, with a more rigorous mechanism planned in future work.

5 Evaluation

We evaluate the effectiveness of our semantics-aware compilation approach on 9 task-parallel C++11 benchmarks from the INNCABS suite [4]. We have selected benchmarks for which equivalent OpenMP versions exist so as to provide an additional reference measurement. Relying exclusively on current C++11 library implementations as the sole point of comparison seems insufficient – as illustrated in Section 2, their performance is not competitive for fine-grained tasks.

Experimental Setup Our evaluation platform is a quad-socket shared-memory system equipped with Intel Xeon E5-4650 processors, each offering 8 cores clocked at a nominal frequency of 2.7 GHz (up to 3.3 GHz with Turbo Boost). The software stack consists of Clang 3.4.2 using libc++ 3.4.2 and gcc 4.9.0 using libstdc++ 3.4.20, both with -O3 optimizations, on a Linux operating system with kernel version 2.6.32-431. The thread affinity for all benchmark runs was fixed using a fill-socket-first policy, and all reported numbers are medians over five runs.

Presentation Due to a lack of space, we are unable to give a detailed account of all our results. In order to provide some more in-depth discussion as well as a comprehensive impression of the overall performance of our approach, we have decided to discuss the results of three individual benchmarks – each representative of a broader category – in detail, as well as provide a separate overview across the entire set of benchmarks. In all cases, we discuss 4 metrics:

cpp11 best defined as the best result obtained by either gcc or Clang using the highest-performing of the three available task launch policies available for `async`. This summarized metric maintains readability on the charts while presenting the state of the art in C++11 production compilers in the best possible light.

omp indicating the performance achieved by the OpenMP version of each benchmark compiled using gcc.

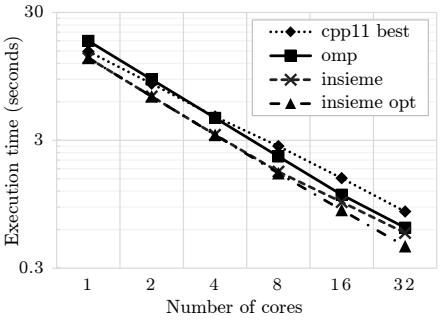

Fig. 4: *Alignment* benchmark results

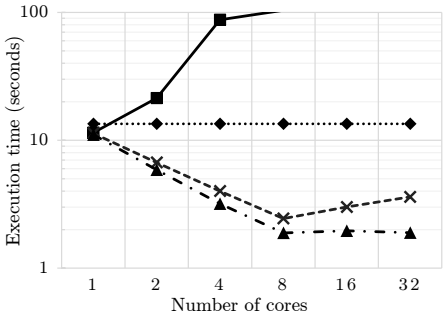

Fig. 5: *Health* benchmark results

insieme our result using library-semantics-aware compilation in the Insieme infrastructure, without heuristic runtime tuning.

insieme opt the same as above, but with the inclusion of the compiler-assisted runtime tuning described in Section 4.

Alignment The *alignment* benchmark is loop-like in structure, and features coarse-grained tasks. As Figure 4 illustrates, its parallel scaling is reasonable with all tested technologies. However, it is worth noting in this context that the best C++11 version shows worse scaling than the other options, likely due to higher threading overhead. The *insieme* and *insieme opt* results are almost indistinguishable for up to 8 cores, with insieme opt scaling better beyond that. This fits perfectly with expectations, as the *alignment* benchmark is classified correctly by the compiler as loop-like, increasing the runtime system queue size which in turn improves utilization at higher degrees of parallelism.

While the log-log presentation in the chart hides it to some extent, the improvement achieved by our approach is tangible even in this coarse-grained case. At 32 cores, the *insieme opt* execution time is 47% shorter than *cpp11 best*, 28% better than *omp*, and an improvement of 21% over *insieme*.

Health This benchmark is recursive in structure, and features extremely fine-grained tasks. Therefore, as depicted in Figure 5, the best C++11 result remains flat as the `deferred` launch policy – which is not parallel – is always the fastest. Even the OpenMP implementation suffers from slowdown, rather than speedup, with increasing thread counts. The low-overhead Insieme runtime system and synchronization coalescing allow our system to achieve scaling up to 8 cores. Once again, the benchmark is correctly categorized by the compiler, with *insieme opt* scaling better up to 8 threads, while also not suffering from the performance drop-off incurred by the base *insieme* version at 16 and 32. This is due to new tasks being pushed to the back of work queues, resulting in larger tasks being spread across all cores and preventing the severe overheads with higher core counts that affect all other versions.

Sort This divide-and-conquer implementation of a mergesort is another example of recursive task parallelism, but its tasks are significantly more coarse-grained than those of *health*. Consequently, the OpenMP version performs much better. However, as seen in Figure 6, the task granularity is still too low for either gcc or Clang to achieve any speedup in the C++11 code. One interesting artifact

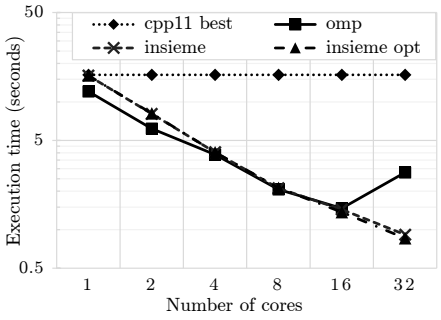

Fig. 6: *Sort* benchmark results

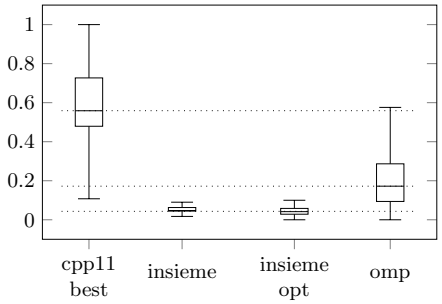

Fig. 7: Overview of results (32 cores)

of note here is that the *omp* version is faster on a single core than any other option, likely due to differences in code generation between pure C and C++11. However, due to its better scaling, the C++11 version compiled and executed with the *insieme* framework catches up to and matches the *omp* version at 4, 8 and 16 cores. At the highest degree of parallelism, the OpenMP version hits a task scheduling wall while our implementation of C++11 continues to scale.

Overall The boxplot in Figure 7 provides a statistical overview of the results across the entire set of 9 benchmarks (*alignment*, *fib*, *floorplan*, *health*, *sort*, *sparselu*, *strassen*, *qap*, and *pyramids*). In order to allow for direct comparison across this diverse set of programs, it was created thusly: i) select the best result across 1 to 32 cores for each benchmark and each of the four previously described versions, ii) normalize these values to the sequential time for the C++11 version of each benchmark, and iii) calculate the required quartiles and medians for the box plot across the 9 resulting benchmark values for each version. Horizontal lines were added at the median for *cpp11 best* and *omp*, and between the two median values for *insieme* and *insieme opt* to improve readability.

These results can be interpreted as follows: with 32 cores at its disposal, the best available C++11 implementation achieves, on average, a parallel speedup of 1.8 (the median is at 0.55) over the sequential version in this set of benchmarks. OpenMP fares better, with a median speedup of 5.9, while our implementation reaches 21.2 without and 23.8 with runtime tuning. In a direct comparison, our tuned results are on average 11.7 times as fast as the *cpp11 best* and 4.1 times as fast as the *omp* baseline.

Looking beyond median performance, it is interesting to note that there is no overlap between *cpp11 best* and *insieme* performance – that is, even at its worst our system performs on par with the best results possible on any of our chosen benchmarks for the existing C++11 implementations. Similarly, the worst cases for *omp* are still on par with the average for *cpp11 best*.

Finally, while *insieme opt* achieves superior median, upper and lower quartile performance than *insieme*, its upper limit is slightly higher. This is due to the *pyramids* benchmark, despite being correctly classified as recursive, performing better at default runtime settings. We believe that this is due to improved cache effectiveness with the default queuing order. We consider statically analyzing memory access patterns and taking them into account for runtime configuration an area for future work.

6 Related Work

There is a large body of existing work in optimizing task parallelism, with a particular focus on scheduling strategies [12, 13] and alleviating task creation overhead [14, 15]. What is common to all of these approaches is that they focus primarily on the runtime level, while we introduce a library-semantics-aware compiler component in order to generate more efficient parallel code, and to provide any given runtime system with static tuning information to use as an initial default. As such, our approach is orthogonal to and compatible with any further runtime-level adaptation and optimization – in fact the runtime system we employ performs adaptive lazy task creation similar to that described by Duran et al. [15].

Looking specifically at the C++ language, parallelism is primarily the domain of libraries [16, 17], and thus also inherently limited to runtime optimization in traditional systems. Meanwhile, existing compiler research related to C++11 parallelism has focused on the correctness of the memory model underlying the standard [18], not on the performance of its library function implementations.

Most compiler research in task parallelism is related to novel, inherently parallel languages [19], or investigates compilation for specific highly-parallel target platforms such as GPUs [20]. Our method is fundamentally different, as it enriches a compiler with understanding of the library-level semantics of a widely-used mainstream language, improving its ability to analyze and optimize the implementation of these semantics.

Liao et al. [21] performed one of the few existing investigations of semantics-aware compilation in parallel computing. However, their goal was improving the applicability of compiler autoparallelization by taking into account STL container semantics in the ROSE compiler framework. Conversely, we propose semantic analysis of programs which are already parallel, in order to more efficiently implement this explicit parallelism.

7 Conclusion

We have presented a *library-semantics-aware* compilation approach for C++11 tasks. It enables i) static optimization of task parallelism by *synchronization coalescing*, ii) executing C++11 programs on a highly optimized parallel runtime system without any user effort, and iii) automatic tuning of runtime settings based on features derived by compiler analysis.

Our system, implemented as an extension to the Insieme compiler, massively improves performance over existing implementations of C++11 parallelism across a range of 9 benchmarks, by a factor of 11.7 on average. Additionally, while compiling code using standard C++11 library constructs for parallelism, it matches and often exceeds the performance and scalability obtained by C/OpenMP programs.

References

[1] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, 12/2006.

[2] Bjarne Stroustrup. *The C++ programming language.* Pearson Education, 2013.

[3] David N. Turner (ed), Hans Wolfgang Loidl, and Kevin Hammond. "On the Granularity of Divide-and-Conquer Parallelism". *Glasgow Workshop on Functional Programming*. Springer-Verlag, 1995, pp. 8–10.

[4] Peter Thoman, Philipp Gschwandtner, and Thomas Fahringer. "On the Quality of Implementation of the C++11 Thread Support Library". *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro Int. Conf. on*. IEEE. 2015, to appear.

[5] *Insieme Compiler and Runtime Infrastructure*. http://insieme-compiler.org. Distributed and Parallel Systems Group, University of Innsbruck.

[6] Diego Novillo. "OpenMP and automatic parallelization in GCC". *GCC developers summit*. GNU. 2006.

[7] Peter Thoman et al. "Automatic OpenMP loop scheduling: a combined compiler and runtime approach". *OpenMP in a Heterogeneous World*. Springer, 2012, pp. 88–101.

[8] Robert D Blumofe et al. *Cilk: An efficient multithreaded runtime system*. Vol. 30. 8. ACM, 1995.

[9] Klaus Kofler et al. "An automatic input-sensitive approach for heterogeneous task partitioning". *Proceedings of the 27th Int. ACM conference on Int. conference on supercomputing*. ACM. 2013, pp. 149–160.

[10] Herbert Jordan et al. "Inspire: The insieme parallel intermediate representation". *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd Int. Conf. on*. IEEE. 2013, pp. 7–17.

[11] Chris Lattner. "LLVM and Clang: Next generation compiler technology". *The BSD Conf.* 2008, pp. 1–2.

[12] Cédric Augonnet et al. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[13] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. "Scheduling parallel real-time tasks on multi-core processors". *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*. IEEE. 2010, pp. 259–268.

[14] Eric Mohr, David A Kranz, and Robert H Halstead Jr. "Lazy task creation: A technique for increasing the granularity of parallel programs". *Parallel and Distributed Systems, IEEE Transactions on* 2.3 (1991), pp. 264–280.

[15] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. "An adaptive cut-off for task parallelism". *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. Int. Conf. for*. IEEE. 2008, pp. 1–11.

[16] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[17] Ping An et al. "STAPL: An adaptive, generic parallel C++ library". *Languages and Compilers for Parallel Computing*. Springer, 2003, pp. 193–208.

[18] Mark Batty et al. "Clarifying and compiling C/C++ concurrency: from C++ 11 to POWER". *ACM SIGPLAN Notices*. Vol. 47. 1. ACM. 2012, pp. 509–520.

[19] Timothy G Armstrong et al. "Compiler techniques for massively scalable implicit task parallelism". *High Performance Computing, Networking, Storage and Analysis, SC14: Int. Conf. for*. IEEE. 2014, pp. 299–310.

[20] John A Stratton et al. "Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs". *Proceedings of the 8th annual IEEE/ACM Int. symposium on Code generation and optimization*. ACM. 2010, pp. 111–119.

[21] Chunhua Liao et al. "Semantic-aware automatic parallelization of modern applications using high-level abstractions". *Int. journal of parallel programming* 38.5-6 (2010), pp. 361–378.