OpenMP as a High-Level Specification Language for Parallelism

And its use in Evaluating Parallel Programming Systems

Max Grossman, Jun Shirako, and Vivek Sarkar

Rice University
Department of Computer Science
{max.grossman,shirako,vsarkar}@rice.edu

Abstract. While OpenMP is the de facto standard of shared memory parallel programming models, a number of alternative programming models and runtime systems have arisen in recent years. Fairly evaluating these programming systems can be challenging and can require significant manual effort on the part of researchers. However, it is important to facilitate these comparisons as a way of advancing both the available OpenMP runtimes and the research being done with these novel programming systems.

In this paper we present the OpenMP-to-X framework, an open source tool for mapping OpenMP constructs and APIs to other parallel programming systems. We apply OpenMP-to-X to the HClib parallel programming library, and use it to enable a fair and objective comparison of performance and programmability among HClib, GNU OpenMP, and Intel OpenMP. We use this investigation to expose performance bottlenecks in both the Intel OpenMP and HClib runtimes, to motivate improvements to the HClib programming model and runtime, and to propose potential extensions to the OpenMP standard. Our performance analysis shows that, across a wide range of benchmarks, HClib demonstrates significantly less volatility in its performance with a median standard deviation of 1.03% in execution times and outperforms the two OpenMP implementations on 15 out of 24 benchmarks.

1 Motivation

The OpenMP specification offers a high-level way of expressing shared-memory parallelism. The level of abstraction offered by OpenMP provides a number of benefits, and has contributed significantly to OpenMP's success in three ways. First, OpenMP offers users a high-level programming model to work in, by default only requiring that they express the parallelism in their application and not the low-level details of how to exploit the parallelism on a given hardware platform. Second, OpenMP's abstractions offer runtime builders flexibility in the optimizations and techniques they can use to produce a well-performing

OpenMP implementation. Third, OpenMP's abstractions map well across hardware platforms and generations, offering both backwards compatibility and the promise of portability to future platforms.

However, there are an increasing number of alternative shared memory parallel programming systems, including HClib [10], Cilk [4], TBB [16], C++ language extensions for parallelism [13], Kokkos [8], Raja [12], X10 [6], and more. None of these models has experienced the widespread acceptance of OpenMP, but facilitating comparisons among them and OpenMP helps move the entire parallel programming models community forward.

To encourage and improve the rigor of these comparisons, this paper looks at OpenMP as a high-level specification of parallelism, a format that is universally understood and applicable and can be mapped to other programming models and runtimes through source-to-source transformations. In this way we can validate both the performance and programmability of these novel parallel programming systems. If a novel parallel programming model is sufficiently flexible to support a reasonable subset of the latest OpenMP standard, we can assume it has the potential for broad applicability. We can also say that if a novel parallel runtime is sufficiently well-performing to match the performance of the various OpenMP runtimes, we can assume that it has the potential to handle real world applications. Otherwise, this comparison can help to identify performance bottlenecks in current runtime efforts.

While these comparisons can clearly motivate improvements to the novel parallel programming system being evaluated, they can also benefit the development of the OpenMP standard. For example, recently the OpenMP standard underwent a massive expansion in the programming constructs and hardware platforms supported, e.g. task-parallel programming and accelerator programming. However, these constructs were not new and unique in their introduction to OpenMP: all were based on existing implementations in existing programming models.

When proposing, defining, and building these new OpenMP features it is important to be able to provide an early prototype of the extension, and then verify that production implementations of accepted extensions are well implemented. The OpenMP-to-X framework solves both of these problems for new OpenMP features by facilitating their construction on top of, and direct comparison to, existing programming systems.

This work contributes the OpenMP-to-X framework for implementing the OpenMP standard on top of other parallel programming models. We see two primary contributions of this work:

1. Using OpenMP as a high level specification of parallelism enables a more direct and fair performance and programmability comparison among programming models, both OpenMP vs. X and X vs. Y.
2. By enabling the OpenMP standard on top of other parallel programming models, we facilitate the extension of OpenMP with experimental abstractions on top of a pre-existing implementation.

In particular, we target the HClib [10] parallel programming library and compare its performance and feature set against the GNU and Intel OpenMP implementations.

The rest of this paper is structured as follows. In Section 2 we provide background on the HClib parallel programming library that we experiment with in this work. In Section 3 we describe the design and implementation of our OpenMP-to-X framework. In Section 4 we use this framework to directly compare HClib and various OpenMP implementations. In Section 5 we discuss the qualitative benefits and potential applications of our OpenMP-to-X framework. We conclude this paper in Section 6.

2 HClib

HClib is a C/C++ library for programming multi-core and heterogeneous systems. It uses a task-based programming model, and supports constructs such as parallel-for, finish-async, promises, and atomic variables.

HClib sits on top of a lightweight work-stealing runtime that load balances user-created tasks across persistent worker threads. Locality is a first-class citizen in the HClib runtime and programming model.

The HClib runtime scheduler is built around the concept of *places*. A single place represents a single hardware component, e.g. an L1 cache, a GPU's memory, the DRAM attached to a socket. Locales are bi-directionally connected to each other so as to emulate the structure of the underlying hardware. For example, a user might configure an L1 place to be connected to an L2 place, rather than being connected directly to system memory because this would more accurately represent the hardware component connectivity.

When making load balancing decisions at runtime, the HClib scheduler uses this graph of places to encourage locality-aware scheduling decisions. These places are also exposed through the programming model so that users can choose to explicitly pin task execution to a particular set of places, or allow more flexibility in their scheduling, thereby improving load balancing opportunities.

HClib tasks consist of an executable body and a closure (lambda) capturing any context from the task's creation point. HClib tasks can block on certain operations, in which case HClib uses runtime-managed stacks to switch the task off of its worker thread, allowing that worker thread to pick up more useful work to complete. The blocked task is automatically made eligible to execute again once its dependencies have been satisfied.

HClib shares many constructs with OpenMP, such as parallel-for, asynchronous task creation, and accelerator offload. However, each programming model also includes constructs that are unique to it; for example, HClib's futures do not require an underlying sequential ordering in the way that OpenMP's task dependencies do. Hence, there are sufficient similarities between HClib and OpenMP that a fair and one-to-one performance comparison can be made at the programming model level, but also enough difference in both their programming constructs and runtime implementation to make that comparison interesting.

3 Methods

In this section, we describe our OpenMP-to-X source-to-source compilation framework, provide more details on the HClib APIs that this framework currently targets, and describe the mapping from OpenMP APIs to HClib APIs. OpenMP-to-X source code and tests are available open source at `https://github.com/agrippa/omp-to-x`.

3.1 OpenMP-to-X Compile-Time Mechanics

OpenMP-to-X is constructed on top of Clang LibTooling [1], a framework for traversing and transforming the AST of a C/C++ program.

OpenMP-to-X iterates over each function in the source program. Inside each function, OpenMP-to-X constructs a representation of the nesting of OpenMP pragmas. At the same time, OpenMP-to-X also tracks visible variables at each OpenMP pragma to enable later closure creation if necessary.

Once the full OpenMP pragma tree is constructed, OpenMP-to-X traverses it from the leaves to the root and applies a pragma-specific transformation at each node. At the completion of this postorder traversal of the OpenMP pragma tree, the current function will have been entirely converted from OpenMP to a different parallel programming model.

3.2 The HClib APIs Targeted by OpenMP-to-X

In this work, we use the HClib parallel programming library as a case study of the OpenMP-to-X framework. Most of this work was performed using HClib's C APIs. However, below we discuss the equivalent C++ APIs as they are more concise.

The `async` API creates a single-threaded task running asynchronously with respect to the task which created it:

```
hclib::async([=] { ... });
```

HClib asynchronous tasks can be chained through the use of futures and promises. For example, to ensure some work `B()` is not performed until some other work `A()` has completed one could use the `async_future` and `async_await` APIs:

```
hclib::future_t *fut = async_future([] { A(); });
async_await([] { B(); }, fut);
```

Promise and future objects can also be explicitly created, satisfied, and waited on by the programmer.

Another way to synchronize on tasks is to use `start_finish` and `end_finish`, whose semantics are the same as those of the `finish` statement in X10, Habanero-Java [5], and Habanero-C [7]. `end_finish` waits for all tasks spawned after the preceding `start_finish` to complete:

```
hclib::start_finish();
hclib::async([] {
  hclib::async([] { B(); });
  A();
});
hclib::end_finish();
// A and B must have completed here.
```

For convenience, HClib also supports a parallel for construct called `forasync`. There is no implicit finish at the end of `forasync` like there is for an `omp parallel for` region. The execution of an `forasync` parallel region can also be dependent on a future, and the completion of all iterations can satisfy a promise.

```
hclib::finish([] {
  hclib::forasync1D(niters, [] (int iter) {
    std::cout << "Hello from iter " << iter << std::endl;
  });
});
```

`forasync` can be combined with HClib places to launch parallel for loops on accelerators, e.g.:

```
hclib::place_t *gpu_place = hclib::closest_place_of_type(GPU);
hclib::finish([] {
  hclib::forasync1D_at(gpu_place, niters, [] (int iter) {
    ...
  });
});
```

Related to accelerators, HClib also supports place-aware memory allocation and copies, e.g.::

```
hclib::place_t *gpu_place = hclib::get_closest_place_of_type(GPU);
void *d_ptr = hclib::allocate_at(nbytes, gpu_place);
hclib::finish([] {
  hclib::async_copy(gpu_place, d_ptr, cpu_place, h_ptr, nbytes);
});
```

3.3 Mapping OpenMP to HClib

Currently, OpenMP-to-X supports the transformation of the OpenMP constructs listed in Table 1. Where relevant, the `private`, `firstprivate`, and `shared` data sharing clauses are also supported. The selection of which constructs and clauses to support was empirically motivated by the constructs used in our benchmarks.

Note that `single` and `master` are not supported in the general case, only as single-threaded task-launching regions. Additionally, the SPMD `parallel` region is not supported by our framework. However, in the benchmarks evaluated in this work these parallel constructs were not used. Our use of OpenMP was partly motivated by its widespread use, and the fact that supporting OpenMP applications on top of another programming model was a strong indicator for the generality of that programming model. However, if in practice particular OpenMP constructs are not widely used (e.g. SPMD `parallel`, `threadprivate`), we do not consider it important that they be used to evaluate other programming models.

Construct	Clauses	Mapping
`critical`		pthread mutexes
`atomic`		Atomic builtins
`task`	depend, if	async
`taskwait`		start-finish/end-finish
`single`		
`master`		async-at
`parallel for`	reduction	for-async

Table 1. A summary of the OpenMP constructs supported by OpenMP-to-X.

Supporting `task` and `taskwait` OpenMP's `task` construct has a natural mapping to the `async` HClib API discussed in Section 3.2. To ensure strict adherence to the OpenMP specification, the OpenMP-to-X framework explicitly constructs a closure for each `async` launched. In this closure, a `private` variable simply has a field declared for it. A `firstprivate` variable has a field declared for it which is initialized from the launching context. A `shared` variable has a field declared for it that is initialized with the address of the shared variable in the launching context.

An OpenMP `task` is translated to an `async` call with the same body. The body of the task must be transformed to unpack `private` and `firstprivate` variables from the closure. Additionally, every reference to a `shared` variable in the task body is translated to be a dereference of the corresponding pointer field in the closure.

HClib has no direct equivalent to OpenMP's `taskwait` construct as there is no implicit tracking of child tasks. Instead, we wrap the body of each `async` in a `start_finish` and `end_finish` pair. When a `taskwait` must be handled, a call to `end_finish` is emitted to ensure all preceding tasks created by the current task have completed. Then, a call to `start_finish` is emitted to open a new task scope. Note that this finish scope does not affect the parallel execution of the `async` it is inside of, it only allows the creating `async` to block on previously created tasks.

OpenMP-to-X also supports the `depend` clause on `task` constructs. While both `depend` and HClib promise/future objects are ways of expressing dependencies between tasks, there are subtle differences that make it challenging to efficiently implement `depend` on top of promises and futures. `depend` uses memory address ranges to specify input and output dependencies, and relies on a sequential creation of dependent tasks to ensure tasks are ordered properly (i.e. a task must be created after all of the tasks it is dependent on). On the other hand, promises and futures are more explicit ways of expressing dependency and must be handled by the programmer, but do not rely on any creation ordering.

To support `depend` on top of promises and futures, OpenMP-to-X stores a mapping from any memory address range designated as an output range of a created OpenMP task to the future object that dependent tasks should be registered on. This design is concerning, as for programs that make heavy use of `depend` this map could grow to be a space and time bottleneck. Indeed, if we ignore small opportunities for compile-time dependency resolution among tasks,

it is hard to see how `depend` could be implemented in any OpenMP runtime without an analogous lookup structure. While OpenMP's design of `depend` does offer an intuitive interface to programmers, it seems it also introduces more overheads than user-managed future and promise objects might.

Supporting `single` and `master` As stated earlier, OpenMP-to-X does not currently support the `single` and `master` contructs in the general case, but only in their use as single-threaded regions inside a wrapping parallel region for the creation of OpenMP tasks, e.g.:

```
#pragma omp parallel
#pragma omp single
{
}
```

`single` is trivial to handle in this case, as it is equivalent to removing both OpenMP pragmas. For `master`, we use HClib locality abstractions and the `async_at` API to force the execution of a `master` region on the main thread of the program.

Supporting `parallel for` While OpenMP-to-X does not support SPMD `parallel` regions, it does support the translation of the combined `parallel for` construct to the `forasync_future` API. For example, the following code:

```
#pragma omp parallel for
for (int i = 0; i < N; i++) { ... }
```

translates in to:

```
hclib::future_t *fut = hclib::forasync_future(...);
fut->wait();
```

Similar code generation could be used to support the new `taskloop` pragma, as the semantics of a `parallel for` are similar in the most common scenario (e.g., without thread-private data). In the case of thread-private OpenMP data, OpenMP-to-X would detect a currently unsupported OpenMP command and exit with an error message.

With the above transformations, a reasonable subset of all OpenMP programs can be automatically and directly converted to use the HClib APIs. This conversion enables a more fair and direct performance comparison between existing OpenMP runtimes and research parallel runtimes (covered in Section 4) and enables prototyping of novel OpenMP constructs and clauses (discussed in Section 5).

4 Experimental Evaluation

In this section, we compare the performance of the HClib, Intel OpenMP, and GNU OpenMP runtimes on a range of benchmarks from the Rodinia, BOTS,

and Kastors benchmark suites. All HClib benchmark implementations were automatically generated using the OpenMP-to-X framework. The automatic nature of the OpenMP-to-X framework enables a more fair and comprehensive performance comparison than would be possible manually.

These experiments were performed on a dedicated 12-core 2.80GHz Intel X5660 CPU node in the Rice DAVINCI cluster with 48GB of system RAM. The GNU compiler toolchain v4.8.5 and Intel compiler toolchain v15.0.2 were used. The version of HClib this work used can be found at `https://github.com/habanero-rice/hclib/tree/resource_workers`. All experiments were repeated ten times, and the `taskset` tool was used to pin threads to cores for both the HClib and OpenMP experiments. The ith software thread is bound to the ith logical core, and all experiments were run with 12 software threads.

When possible, we compare both "small" and "large" datasets for each benchmark. For a complete list of benchmarks and configurations, please refer to `https://github.com/habanero-rice/hclib/blob/resource_workers/test/performance-regression/cpu-only/datasets.sh`.

4.1 Variance of each Runtime

The first metric to consider is how much variance exists across ten runs of the same benchmark for a given runtime. The percent standard deviation of each benchmark and dataset is plotted in Figure 1. We measured a median percent standard deviation of 10.59%, 3.44%, and 1.03% for IOMP, GOMP, and HClib, respectively.

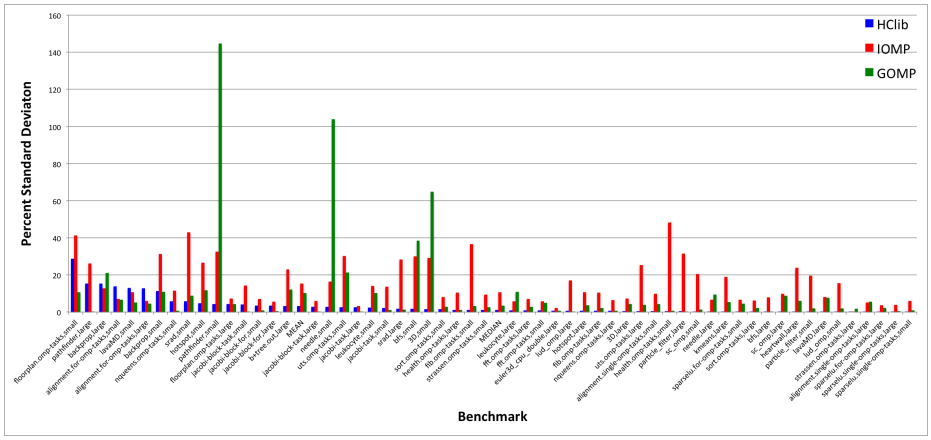

Fig. 1. The percent standard deviation of each benchmark and dataset on all runtimes, sorted from most to least variance on HClib. Higher values indicate more variance from run to run.

The main trend of importance is the lower variance offered by the HClib runtime. Our first thought on seeing these results was that some OpenMP initialization code was being measured, and that it was causing high overheads and high variance. However, even when we investigate the two benchmark configurations with the highest variance on the IOMP runtime (`srad,small` and `floorplan.omp-tasks,small`) and manually modify them to ensure any OpenMP initialization must have occured prior to the timed code region, we continue to see high variance.

Additionally, these two high-variance benchmarks have significantly different patterns of parallelism: `srad` consists of an outer sequential loop wrapped around two inner parallel-for regions, and `floorplan` uses the `task` and `taskwait` constructs. This suggests that this volatility is not a localized problem, but rather one that might be exposed by many applications. Admittedly, these are both short-running benchmarks so a small amount of volatility can appear as a large percentage. However, the fact that this volatility is not reflected in the HClib results suggests that it is not an intrinsic characteristic of these benchmarks.

Further analysis of these two benchmark configurations using HPCToolkit [2] reveals that HClib does a better job of utilizing worker threads. For example, in the `srad,small` benchmark configuration 45.5% of IOMP execution time is spent under two IOMP internal functions: `__kmp_fork_barrier` and `__kmp_join_barrier`. In contrast, HClib spends 24.9% of time in runtime internal functions related to load balancing. These high runtime overheads and thread idleness might contribute to the observed volatility in IOMP and GOMP execution times.

4.2 Overall Performance

We also consider the overall performance achieved by HClib, IOMP, and GOMP for benchmarks where a percent standard deviation below 10% was observed. Focusing only on benchmarks that achieve reasonably consistent performance on all runtimes allows us more confidence in any conclusions drawn. Figure 2 plots the median speedup for all consistent benchmarks, normalized to GOMP performance.

For 15 out of a total 24 consistent benchmarks, HClib outperforms IOMP and GOMP. For 6, IOMP is the highest performer, and for 3 we see the best performance from GOMP. While the mean speedup for HClib relative to GOMP is slightly lower than IOMP at 2.45x and 2.59x respectively, HClib demonstrates more consistent results by having a median speedup of 1.34x compared to IOMP's median of 1.02x.

To better understand the cause of the performance difference between the two highest performing runtimes, HClib and IOMP, we also investigated various hardware counters. The main trends we observed for both runtimes was that higher performance was strongly correlated with 1) fewer last-level cache misses, and 2) fewer instructions executed. This indicates the importance of locality, and the importance of keeping overheads to a minimum. For example, on the Kastors jacobi-block-task benchmark with the large dataset, IOMP's 70%

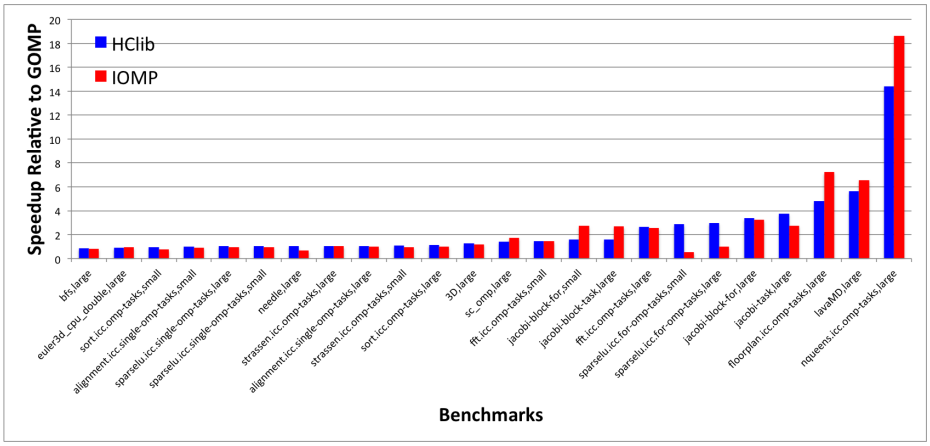

Fig. 2. The speedup of each benchmark and dataset that demonstrated a percent standard deviation below 10% on all runtimes, normalized to the GOMP results and sorted from lowest to highest speedup on the HClib runtime. Higher values are better.

performance improvement over HClib was correlated with a 4.5x reduction in last-level cache misses. On the BOTS sparselu benchmark, HClib demonstrated a 5.50x and 3.04x speedup on the small and large datasets, respectively. This was correlated with a 2.94x and 4.25x reduction in instructions executed. These hardware counter results continue to support the observations from Section 4.1: HClib's work-stealing runtime implementation keeps useful computation on the worker threads, rather than internal runtime logic.

We also used built-in HClib metrics to analyze these benchmarks. We observe that for many of the benchmarks where HClib experienced the highest speedup there is a single worker thread producing most of the tasks in the application, with the others all stealing from it. This insight combined with the more consistent results shown in Section 4.1 and the HPCToolkit investigation above indicates that HClib's work-stealing scheduler may be more aggressive about load balancing while exhibiting lower overhead than IOMP's or GOMP's task schedulers.

It is important to point out that one of the common shortcomings of source-to-source code generation is that naive techniques can often break compiler optimizations by making application source code more difficult to analyze (e.g., by taking the address of variables that would normally be stored in registers). While it is difficult to isolate the performance side effects OpenMP-to-X's transformations would have, the side effects of the optimization-limiting transformations (e.g., taking addresses of shared variables, passing function pointers, adding volatile qualifiers) are fundamentally required for correctly implementing OpenMP semantics, regardless of programming model. For example, while taking the address of a shared OpenMP variable as part of OpenMP-to-X's transformations might force the compiler to allocate stack space for it, that space must have been allocated in the OpenMP version of the program as well in order for multiple threads to access it. Therefore, we believe the optimization-

limiting side effects of OpenMP-to-X transformations would be similar to the loss of optimization necessary to support OpenMP semantics and would not significantly affect these performance results.

5 Discussion

In our experience, OpenMP-to-X has proven to be a powerful tool for runtime comparisons. It can motivate changes and offer insights in to both the current state of the OpenMP standard and runtimes, as well as current research runtimes.

5.1 Insights Gained into HClib

The comparative analysis enabled by the OpenMP-to-X framework and performed as part of this work identified several bottlenecks in the HClib runtime. In particular, memory allocations for finish-scope management, runtime-managed stacks, and future/promise object management were identified as limiting the scalability of the HClib runtime, and addressed. These results validate the OpenMP-to-X framework as a tool for motivating improvements to parallel research runtimes.

This work also motivated the addition of atomic variables as a first-class citizen of HClib, as a result of the heavy usage of `omp atomic` in many of the benchmarks used in this work. The use of atomic compiler intrinsics as the target for `omp atomic` in the OpenMP-to-X framework was observed as a performance bottleneck for some generated HClib codes. While HClib atomic variables were not used in the performance analysis in Section 4, they are an example of how comparisons enabled by the OpenMP-to-X framework can motivate improvements to the programming model of a research runtime.

As part of these experiments, we identified several features of OpenMP that are desirable for performance or programmability but which have no analogue in HClib. For example, static scheduling of parallel for loops is not something currently supported in HClib, but would reduce runtime overheads further. Tied task continuations could benefit locality in fork-join style programs. The `master` construct is useful when interacting with third-party libraries which have restrictions on the calling thread. One interesting note about these constructs is that while they are distinct entities in OpenMP, they could also be unified if OpenMP had the concept of locality built in to its programming model (as HClib does with hierarchical places). A statically scheduled parallel for loop is simply one which places the constituent tasks of that parallel for at specific cores for execution, precluding any dynamic load balancing. A tied task is simply a task whose continuation is launched at the same core as was originally executing it. A `master` region is a code block launched at the core on which the master thread resides. This suggests that a well-defined locality model in the OpenMP standard would allow for a unification of many constructs that are disjoint today.

One challenge in this work was supporting the `task depend` clause on HClib. While HClib supports dependent tasks, it does so through programmer-managed promise and future objects. In OpenMP, dependencies are programmer-managed, but there are no explicitly managed dependency objects. While we find both models easy to work with, we found the underlying sequential semantics requirements of the OpenMP approach somewhat restricting, and have concerns over the ability to scale this approach to many tasks since large numbers of tasks will require large numbers of lookups on an ever-growing table of tasks and their output relations.

Perhaps the largest difference between OpenMP and the HClib runtime is that the OpenMP approach is able to take advantage of compile-time optimizations and code transformations, while HClib is purely a library. While in theory the hybrid compiler-runtime approach should have performance benefits, these results show that a purely library-based approach can also produce consistent and well-performing parallel programs.

5.2 Motivating Extensions to OpenMP

As part of this work, we also reflected on the current OpenMP standard and what features of HClib could lead to potential OpenMP extensions in the future.

As part of our experimental evaluation, we found metrics exposed by the HClib runtime to be useful when reasoning about the behavior of parallel programs. However, without matching metrics from the OpenMP runtimes it is difficult to make strong conclusions. The proposed OpenMP Tools API [9] seems to be a synergistic project with the OpenMP-to-X framework which could help provide this missing functionality.

One of the largest differences between HClib and the current OpenMP specification is the inclusion of locality constructs in the programming model. HClib's hierarchical place trees allow programmers to indicate both where tasks may run and where memory should be allocated. In future systems with deeper memory hierarchies, this locality support within the programming model itself may be crucial for productive parallel programming. However, the quantitative benefits of these constructs were not evaluated as part of this work.

5.3 Other Targets for OpenMP-to-X

While HClib was selected as the target of this work because of the authors' familiarity with it, many other shared-memory programming models could benefit from evaluation using OpenMP-to-X.

Work-in-progress is using OpenMP-to-X to target CUDA as a backend. CUDA's constrained, data-parallel programming model and discrete address spaces present unique challenges to supporting it under OpenMP-to-X. It has necessitated significant extensions to OpenMP-to-X, mirroring the developer effort normally required to convert an OpenMP program to an equivalent CUDA version. Despite this, the scope of the OpenMP specification that can be supported on CUDA

is far more limited than for HClib. This work has already demonstrated successful automatic conversion of OpenMP `parallel for` loops to CUDA using OpenMP-to-X, similar to past works [14][15][3].

One strong candidate for evaluation would be the Kokkos programming model [8]. While Kokkos has received attention as a fundamental execution layer for the Trilinos project [11], it has also received criticism in its applicability as a general-purpose programming model as a result of its restricted programming model. However, the Kokkos runtime's focus on performance and low overheads would also make for an interesting comparison on the subset of the OpenMP specification it can support.

Both the Intel Thread Building Blocks [16] and Cilk [4] programming models would be good case studies as well, as they are arguably the next most commonly used shared-memory parallel programming models after OpenMP.

We note that HClib, Kokkos, TBB, and Cilk are all syntactically C/C++ programming models. Due to its construction on top of the Clang compiler frontend, targeting C/C++ programming models with OpenMP-to-X is a requirement of the current implementation. While it would be possible to extend this framework to support targeting significantly different programming models or languages, this remains future work.

Targeting other programming models that are not syntactically similar to C/C++ would include all of the usual challenges that come with converting one programming language to another. Programming languages with a C/C++ compatibility layer (e.g. Java) would reduce these challenges.

Targeting parallel programming models that are syntactically C/C++ but diverge significantly from OpenMP in their abstractions (e.g., graph programming models) may also be possible. However, OpenMP-to-X's relevance as a tool for performance or programmability comparison would be lessened. In general, programming models with non-overlapping APIs arise from different motivations, and so the comparison of them might be meaningless.

6 Conclusions and Future Work

In this work, we used the OpenMP-to-X source-to-source code generation framework to enable a variety of experiments. We compared various parallel runtimes, using OpenMP as an intermediate representation for parallelism. We performed feature comparisons by studying how OpenMP operations could be mapped to other parallel programming models, and used that information to motivate extensions to both OpenMP and other parallel programming models. Through these studies, we have improved on the flexibility and performance of the HClib runtime and demonstrated overheads in the Intel and GNU OpenMP runtimes that merit further investigation.

While many past studies have compared OpenMP to other parallel programming models and runtimes, this is the first attempt we are aware of to standarize that process by using the same input OpenMP programs to evaluate different runtimes. By building a framework for consistent and automated generation of

parallel programs from OpenMP programs, we enable more comprehensive and fair performance comparisons between parallel programming models in the future, as well as a path to rapid prototyping of novel OpenMP functionality.

7 Acknowledgments

This work was supported in part by the Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097 and Rice University.

The authors would also like to acknowledge the contributions of Vivek Kumar, Nick Vrvilo, and Vincent Cave to the HClib project.

References

1. Clang libtooling. `http://clang.llvm.org/docs/LibTooling.html`.
2. L. Adhianto. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 2010.
3. M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.
4. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.
5. V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.
6. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.
7. S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with mpi. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 712–725. IEEE, 2013.
8. H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.
9. A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. Ompt and ompd: Openmp tools application programming interfaces for performance analysis and debugging. In *International Workshop on OpenMP (IWOMP 2013)*, 2013.
10. Habanero Research Group. Hclib: a library implementation of the habanero-c language. `http://hc.rice.edu`, 2013.
11. M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.
12. R. Hornung and J. Keasler. The raja portability layer: overview and status. 2014.
13. International Organization for Standardization. The C++ Programming Language Standard. `https://isocpp.org/std/the-standard`, 2014.

14. S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.
15. S. Ohshima, S. Hirasawa, and H. Honda. Ompcuda: Openmp execution framework for cuda based on omni openmp compiler. In *International Workshop on OpenMP*, pages 161–173. Springer, 2010.
16. J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.