OpenMPIR

Implementing OpenMP Tasks with Tapir

George Stelle
Los Alamos National Laboratory
stelleg@lanl.gov

William S. Moses
MIT CSAIL
wmoses@mit.edu

Stephen L. Olivier
Center for Computing Research
Sandia National Laboratories
slolivi@sandia.gov

Patrick McCormick
Los Alamos National Laboratory
pat@lanl.gov

ABSTRACT

Optimizing compilers for task-level parallelism are still in their infancy. This work explores a compiler front end that translates OpenMP tasking semantics to Tapir, an extension to LLVM IR that represents fork-join parallelism. This enables analyses and optimizations that were previously inaccessible to OpenMP codes, as well as the ability to target additional runtimes at code generation. Using a Cilk runtime back end, we compare results to existing OpenMP implementations. Initial performance results for the Barcelona OpenMP task suite show performance improvements over existing implementations.

1 INTRODUCTION

When writing task-parallel programs today, there is a large selection of potential programming models and implementations to consider. Unfortunately, despite having some significant overlaps in semantics, parallel programming models such as OpenMP [20], Cilk [3], Kokkos [6], HPX [11], Charm++ [12], Qthreads [25], pthreads[19], MPI [8], Chapel [4], UPC [7], OpenCL [24], and OpenACC [26], among others, have limited ability to interoperate either at the level of compile-time analysis or at run-time execution. Moreover, many of these programming models are implemented as run-time libraries with only primitive compiler support. As a consequence, most are not able to take advantage of potential compiler analysis and optimization capabilities. While there has been some recent work on specializing compilers

to reason about parallel programming libraries [17], fragmentation among models presents a significant challenge for compiler writers who must choose a single model, or multiple intermediate forms, for analysis and optimization. An ideal solution would be a common representation of parallel semantics and constructs.

Compilers implement the majority of their analyses and optimizations on intermediate representations (**IRs**) that allow such transformations to be written once with relative ease and applied to a variety of source front-end languages such as C or C++. Historically, the IR of mainstream compilers such as LLVM [15] or GCC's Gimple [16] haven't supported parallel constructs. As a consequence, compilers using these IRs haven't had the ability to reason about parallelism. The recent work of Schardl et al. has shown that the LLVM IR can be extended to represent fork-join parallelism without requiring a major rewrite [23]. They did so by extending the LLVM instruction set with three instructions capable of representing fork-join parallelism. To benchmark performance and test accuracy they wrote a front end which translated programs written with Cilk, a parallel programming model for C/C++, into their Tapir IR. They also implemented a back end to **lower** Tapir programs to vanilla LLVM IR with embedded Cilk runtime calls. They suggested that a similar approach could be taken with OpenMP and other such frameworks.

In this work we put that suggestion to the test, implementing OpenMP tasks by compiling them to Tapir instructions. This allows us to both analyze and optimize OpenMP programs as well as compile programs written in OpenMP to use other runtimes. For this paper our prototype implementation uses the Cilk runtime as a back end, and we run our prototype implementation on the Barcelona OpenMP task suite [5]. We discuss the kinds of optimizations this enables and how the work can be extended to include other parallel constructs and semantics in OpenMP. We also discuss the possibility of extending this work to other programming models, which would help to alleviate some of the problems with fragmentation previously discussed.

The contributions of this work include:

- A prototype front end that transforms OpenMP task constructs to Tapir IR;

- Benchmarks of the implementation using a Cilk run-time back end, showing performance improvements over existing OpenMP implementations; and
- An initial investigation into the reasons for these performance improvements.

Our hope is to move the community towards further conversations on a shared IR that can be used for many programming models. The work presented here only represents a small step in that direction and there is still a herculean amount of design and implementation details to be considered. We argue that the benefits vastly outweigh the costs and we hope to see future work explore additional possibilities.

The remainder of the paper is organized as follows. In Section 2 we give background and motivation for the presented work. Following that, we describe Tapir in Section 3, and OpenMP tasking in Section 4. We then discuss the implementation, and how we compile OpenMP task constructs to Tapir IR in Section 5. In Section 6 we describe the evaluation setup, including which implementations we compare to and how. We discuss the results of evaluation in Section 7, including possible explanations for discrepancies. We discuss the implications of the results and the significance of this work in a large context in Section 8. Finally, we cover some of the many ways the work could be extended in Section 9, and conclude in Section 11.

2 BACKGROUND

Internal representations are a crucial part of modern compiler design and implementation. By representing code in a generic, hardware-agnostic format, they allow both multiple frontends and backends to all take advantage of a common/shared series of compiler analyses and optimizations.

LLVM has gained traction in the broad community for its simplicity, relative ease of both experimentation and modifications, and extensibility [15]. Historically, LLVM has only had sequential semantics: there is no explicit notion of concurrent or parallel processes. This has resulted in existing compilers treating parallel constructs as thin wrappers for calls into run-time systems. These calls are infeasible for the compiler to reason about because of the complexity and flexibility of the runtimes being called, and the challenge of reasoning about the corresponding semantics in a uniform fashion. The addition of parallel constructs was recently explored by Schardl et al. By adding three first-class instructions to LLVM, Tapir enables analyses and optimizations of parallel constructs. We briefly describe the key aspects of Tapir in Section 3.

Arguably the most common parallel programming model for on-node parallelism in HPC, OpenMP has grown from being a method for implementing parallel loops into a large and complex specification for parallelism. This paper focuses on a relatively recent addition to the OpenMP specification: the `task` and `taskwait` constructs. A detailed description of these constructs is presented in Section 4 and a more complete description can be found in the OpenMP specification [20].

2.1 Fragmentation

In this section we further discuss a significant challenge in HPC: fragmentation of parallel programming models. There are many ways to write parallel programs [2–4, 6–8, 20, 25]. From libraries, to language extensions, to standalone languages, to combinations of these, the fragmentation of parallel programming models is a problem that is only getting worse. This means that when writing tooling, optimizations, or analyses, one must choose which models to target as well as understand points of contention and interoperability between them. A common difficulty among all of these is that reasoning about parallel programs is hard and compilers can potentially help reduce this burden. This is exactly analogous to the problem LLVM solves: compiler optimizations are hard, so sharing them is valuable. Tapir is a demonstration that the LLVM approach to compilation of serial code can be effectively extended to help reason about parallel code.

With fragmentation comes questions of compatibility or composability of different programming models. To illustrate this, consider the example of a program using a library that uses OpenMP to implement parallelism internally. If this program uses a different programming model for parallelism, e.g. Cilk, there is currently no good solution for ensuring that the two different back ends will cooperate with management of hardware resources.

Addressing these concerns is a major motivation for our use of Tapir. By standardizing an IR, in addition to easing the development of optimizations, one avoids duplicating work across different implementations. It also enables the possibility of multiple programming models coexisting peacefully. We will return to this consideration later in Section 9 when discussing future work.

3 TAPIR

Tapir is an extension to LLVM seeking to resolve issues in optimizing parallel code. Prior to the introduction of Tapir, compilers would represent parallel programs by directly translating their syntax to opaque runtime calls. This allowed compilers to support parallel programs but meant that traditional optimizations such as code motion or common-subexpression elimination weren't able to reason about parallel programs. This often led to parallel programs running significantly slower than expected. In the Tapir project the authors show that it is possible to represent and optimize fork-join parallel programs with relatively minimal modifications to the compiler, namely the introduction of three instructions designed to interface well with an existing compiler.

3.1 Compilation without Tapir

Consider the first code segment defined in Figure 1, a simple OpenMP program which performs divide-and-conquer search. However, as written, there is a simple optimization that can be performed, that the value `(low+high)/2` may be computed once before either parallel task, thereby avoiding redundant computations. For the serial version of this program optimizations like this are performed automatically. However,

a

```
01  void search(int low, int high) {
02    if (low == high) search_base(low);
03    else {
04      #pragma omp task
05      search(low, (low+high)/2);
06      #pragma omp task
07      search((low+high)/2 + 1, high);
08      #pragma omp taskwait
09    }
10  }
```

b

```
11  void search(int low, int high) {
12    if (low == high) search_base(low);
13    else {
14      int mid = (low+high)/2;
15      #pragma omp task
16      search(low, mid);
17      #pragma omp task
18      search(mid + 1, high);
19      #pragma omp taskwait
20    }
21  }
```

Figure 1: Example of common-subexpression elimination on an OpenMP program. a The function `search`, which uses parallel divide-and-conquer to apply the function `search_base` to every integer in the closed interval [`low`, `high`]. b An optimized version of `search`, where the common subexpression `(low+high)/2` of the original version is computed only once and stored in the variable `mid` in the optimized version.

without the use of Tapir, these sorts of optimizations are not run on parallel programs.

If one were to compile this program using the traditional technique of lowering `#pragma` directives into runtime calls, one would find a program similar to Figure 2. The OpenMP pragmas are effectively treated as syntactic sugar for runtime calls because LLVM has no concise way to represent such parallel constructs. Importantly, these runtime calls tend to obfuscate the program, making it infeasible to analyze and/or optimize parallel code.

3.2 Tapir Internals

Tapir is a compelling solution to this problem because it allows existing analysis and optimizations to work on parallel programs without significant modifications. To properly describe Tapir requires a brief introduction to key aspects of LLVM. LLVM represents expression/evaluation order through a control-flow graph (CFG) of blocks that contain sequences of instructions. Blocks always end with terminator instructions that define the edges of the control flow graph. For example, Figure 3 shows a simple function and its corresponding LLVM code.

To represent parallelism Tapir introduces the `detach`, `reattach`, and `sync` instructions. The `detach` statement is syntactically similar to a conditional branch in that it is a terminator instruction with two blocks. However, unlike the conditional

```
22  void task_1(int* clos) {
23    search(*clos[0], (*clos[0] + *clos[1])/2);
24  }
25
26  void task_2(int* clos) {
27    search(*clos[0], (*clos[0] + *clos[1])/2);
28  }
29
30  void search(int low, int high) {
31    if (low == high) search_base(low);
32    else {
33      int* closure_1[] = {&low, &high};
34      omp_run_task(task_1, closure_1);
35      int* closure_2[] = {&low, &high};
36      omp_run_task(task_2, closure_2);
37      omp_taskwait();
38    }
39  }
```

Figure 2: Simplified compilation of the unoptimized search code from Figure 1. a The parallel tasks are moved into their own functions and accompanying closures. These functions are then passed to OpenMP runtime calls. This obfuscates the program, making it infeasible for an optimizer to reason about what is happening.

branch which denotes a choice between successors, the `detach` instruction denotes that its two successors may run in parallel. Semantically, the first successor of a `detach` instruction is referred to as the **detached block** and represents a task that can, but is not required to, run in parallel with the **continuation block**, or the second successor to the detach statement. The `detach` is analogous to a `fork` in most fork-join models.

The end of a task created by a `detach` instruction must end with a `reattach` instruction to the corresponding continuation block. The `reattach` instruction is used to denote the end of a parallel task. Finally, one may use the `sync` instruction to wait for any outstanding tasks created by `detach` instructions within the current function to complete. The `sync` instruction is analogous to a `join` in most fork-join frameworks.

4 OPENMP TASKS

Initially, the cross-vendor OpenMP [20] shared memory programming model focused on the execution of data parallelism by a cooperating team of threads, e.g., dividing the iterations of a loop among the threads. Version 3.0 of the OpenMP API specification introduced support for lightweight asynchronous tasks, designated by the application developer and scheduled onto the team of threads by the OpenMP runtime implementation. The `task` construct applied to a structured block of code creates an explicit task, and the `taskwait` construct waits for completion of all tasks generated by the current task.

Recursive task creation and synchronization using the constructs result in an implicit directed acyclic graph (DAG) that allows both reasoning about and visualization of the program execution. Figure 4 shows some example code, a view

a

```
40  int f(int x) {
41    int y = 10;
42    if (x == 0) {
43      y = 12;
44    }
45    return y;
46  }
```

b

```
47  define i32 @f(i32 %x) {
48  entry:
49    %y = alloca i32, align 4
50    store i32 10, i32* %y, align 4
51    %cmp = icmp eq i32 %x, 0
52    br i1 %cmp, label %if.then, label %if.end
53
54  if.then:
55    store i32 12, i32* %y, align 4
56    br label %if.end
57
58  if.end:
59    %1 = load i32, i32* %y, align 4
60    ret i32 %1
61  }
```
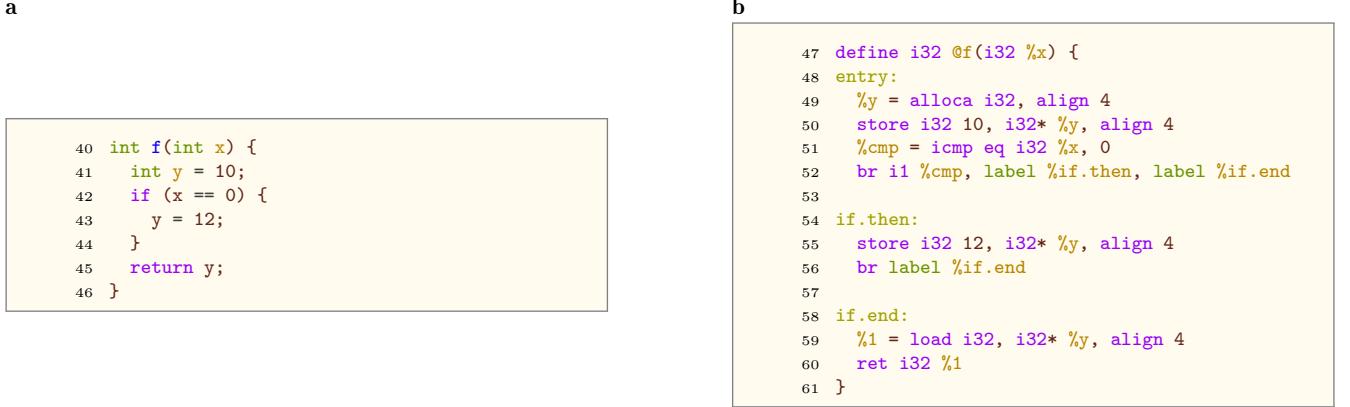
Figure 3: Example code snippet and the corresponding LLVM code. This code segment contains three blocks: `entry`, `if.then`, and `if.end`. The conditional statement is implemented in LLVM by the conditional branch at the end of the entry block.

of the task DAG, and a simplified execution schedule mapping the tasks to a team of two threads. In the example, the nth Fibonacci number is calculated by recursively generating tasks to calculate the $(n-1)$th and $(n-2)$th Fibonacci numbers. The `taskwait` ensures that the child tasks have completed before their answers are combined to yield the final result.

The initial design of the OpenMP task model established a basic framework for asynchronous task parallel execution in OpenMP programs [1]. Subsequent versions of the OpenMP specification, up to the current version 4.5, have added new features to the tasking model. The `depend` clause codifies data dependences among tasks, indicating that a data location is an input or output of a task. The runtime system ensures that a task is not scheduled until its input dependencies are fulfilled. The `taskloop` construct combines groups of independent loop iterations into explicit tasks, enabling composition of concurrent loop execution and independent explicit tasks within the same OpenMP parallel region. The `taskgroup` construct waits on not only all child tasks, but all descendent tasks, providing a deep synchronization. The `taskwait` construct allows the application to indicate a point at which the implementation may suspend the current task to work on other tasks, as may be desired for long-running tasks that generate many others. The task concept was also leveraged to provide asynchronous offload of data and computation to accelerators by applying the `nowait` clause to device constructs such as the `target` construct to generate an asynchronous *target task*.

Several clauses for the `task` construct aim to optimize execution of tasks but can be safely ignored by an implementation that chooses to do so. The `mergeable` clause allows the implementation to omit creation of a new data environment for a descendent of a task marked with the `final` clause. The `priority` clause assigns an integer priority to the task and recommends the prioritization of tasks with higher priority values. The `untied` clause allows a task to be migrated between threads after suspension, which enables practical use

of *work-first scheduling* (suspending the parent task in favor of executing each child task immediately on the thread where it is generated).

5   IMPLEMENTATION

Tapir is implemented as an extension to the LLVM instruction set. Clang is a C family compiler that has support for OpenMP extensions and targets LLVM [14]. The existing Clang OpenMP implementation maps OpenMP constructs directly to OpenMP runtime library calls by wrapping a C statement in a `CapturedStmt`. This replaces a C statement with a function call into the OpenMP runtime library, along with a machine generated function whose body contains that statement.

For this work we replaced a subset of the OpenMP implementation to generate Tapir IR instead of vanilla LLVM IR with OpenMP runtime calls. Specifically, we replace the two primary `task` and `taskwait` pragmas for task parallelism. By re-using code from Schardl et al. for code generation of Cilk constructs, we were able to easily generate Tapir code for these OpenMP pragmas. The ease with which this was completed is a testament to the quality of the Tapir implementation.

While we did implement the codegen for the `task` and `taskwait` constructs, it's worth noting that even for these pragmas the implementation is incomplete. Currently, any clauses modifying the behavior of the pragmas is ignored. Probably the most common semantics this will change is for the semantics of variables, e.g. `shared` vs. `private`. Surprisingly, this had little effect on the correctness of the entire Barcelona OpenMP task suite. Indeed, it is likely that fixing this problem would marginally increase performance by reducing the number of memory copies for variables declared `private` in OpenMP code. The fact that behavior wasn't changed significantly is worth revisiting, and we do so in Section 8.

The work represented in this paper is only front-end implementation, which means we use the only existing back end

a

```
62  int fib(int n)
63  {                       // A
64    if (n < 2)
65      return n;
66    else
67    {
68      int x, y;
69      #pragma omp task
70        x = fib(n-1);  // B
71      #pragma omp task
72        y = fib(n-2);  // C
73      #pragma omp taskwait
74      return (x+y);     // D
75    }
76  }
```

b

c

Time	t_0	t_1	t_2
Thread 0	A	B	D
Thread 1		C	
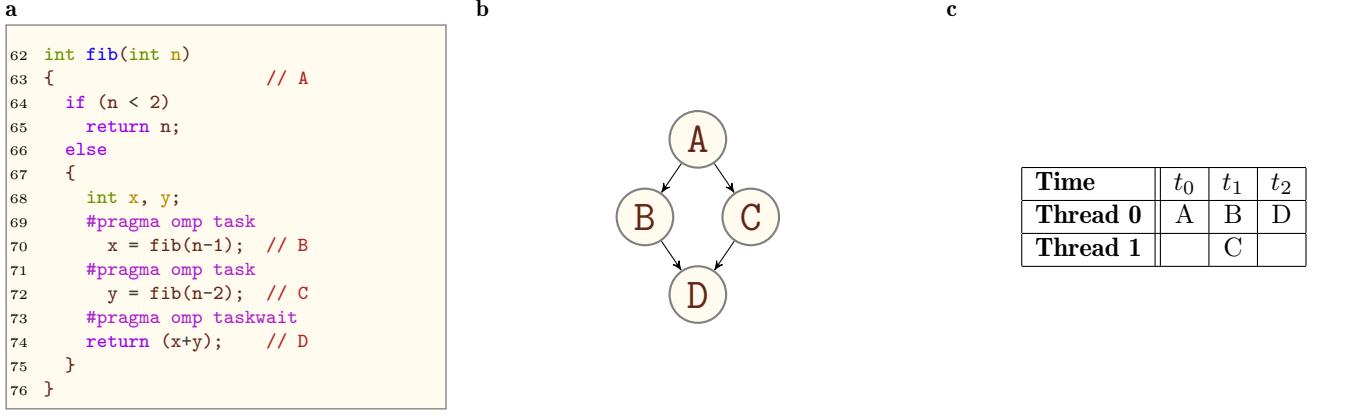
Figure 4: Code, task graph, and schedule of a simple brute force recursive Fibonacci number calculation on two threads.

for Tapir. The existing back end generates code that calls the Cilk runtime `libcilkrts`. This has implications for adhering to the OpenMP specification. For example, environment variables such as `OMP_NUM_THREADS` are ignored and replaced by `CILK_NWORKERS`. We discuss some of these issues following and return to address others in the discussion of future work.

There are other issues that had to be overcome to run full OpenMP programs using Tapir. Generally, in addition the `task` and `taskwait` pragmas, a program must contain pragmas to initialize OpenMP. A standard pattern for building task-parallel OpenMP programs is to insert a `parallel` pragma to start the necessary hardware threads, followed immediately by a `single` pragma to have only one thread continue on the specified statement. Then the other threads will get work from spawned tasks instead of implicitly executing the same code in a data parallel manner. For the purpose of this work we took the shortcut of replacing these pragmas with no-ops. This worked well for all examples except for one in which after the `parallel` and `single` pragmas the top level task was called with an unnecessary `task` pragma. Because the wait was implicit and unhandled by our implementation, the simple fix was to remove the `task` pragma. This was the only change required to the source code. Our temporary no-op shortcut, of course, doesn't follow the OpenMP specification, and should be addressed by future work. Section 9 discusses how the implementation can be improved.

All code used for the implementation is available at `https://github.com/lanl/openmpir-clang`. The version used for this paper is tagged LLVM17.

5.1 Example

To better understand how the OpenMP to Tapir compiler works, we turn to the parallel fragment of the `fib` example from Section 4. In Figure 5 we show the input OpenMP code and the Tapir code that is generated.

Upon entering the `else` branch, the code immediately detaches the basic block corresponding to the call to `fib(n-1)`, labeled `det.achd`. The continuation for the first call also immediately detaches the basic block corresponding to the

call of `fib(n-2)`. Finally, the continuation for the second call immediately calls `sync`. This is a nice example of code that will be optimized by Tapir. The LLVM code shown in Figure 5 is not optimized. With optimizations turned on, Tapir will replace a `detach` of a function call followed immediately by `sync` with a simple function call, resulting in faster code.

6 EVALUATION

To evaluate the use of Tapir we compared performance on the Barcelona OpenMP Task Suite to several existing OpenMP implementations. The suite is a set of tests intended to test performance of OpenMP implementations using both irregular and regular tasking.[1] The following list provides abridged descriptions of each benchmark taken verbatim from the test suite distribution [5].

- **Alignment** aligns all protein sequences from an input file against every other sequence using the Myers and Miller algorithm. The alignments are scored and the best score for each pair is provided as a result. The scoring method is a full dynamic programming algorithm. It uses a weight matrix to score mismatches, and assigns penalties for opening and extending gaps. The output is the best score for each pair of them.
- **FFT** computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey algorithm. This is a divide and conquer algorithm that recursively breaks down a Discrete Fourier Transform (DFT) into many smaller DFT's. In each of the divisions multiple tasks are generated.
- **Fibonacci** computes the nth Fibonacci number using a recursive parallelization. While not representative of an efficient Fibonacci computation it is still useful because it is a simple test case of a deep tree composed of very fine grain tasks.
- **Floorplan** kernel computes the optimal floorplan distribution of a number of cells. The algorithm gets an

---

[1] Since original publication, an unbalanced tree search benchmark has been added to the benchmark suite. Unfortunately, due to time constraints, we weren't able get results for this additional benchmark.

a

```
77  #pragma omp task
78    x = fib(n-1);
79  #pragma omp task
80    y = fib(n-2);
81  #pragma omp taskwait
```

b

```
82  if.end:
83    detach label %det.achd, label %det.cont
84
85  det.achd:
86    %2 = load i32, i32* %n.addr, align 4
87    %sub = sub nsw i32 %2, 1
88    %call = call i32 @fib(i32 %sub)
89    store i32 %call, i32* %x, align 4
90    reattach label %det.cont
91
92  det.cont:
93    detach label %det.achd1, label %det.cont4
94
95  det.achd1:
96    %3 = load i32, i32* %n.addr, align 4
97    %sub2 = sub nsw i32 %3, 2
98    %call3 = call i32 @fib(i32 %sub2)
99    store i32 %call3, i32* %y, align 4
100   reattach label %det.cont4
101
102 det.cont4:
103   sync label %sync.continue
```

Figure 5: Transformation from OpenMP task constructs to Tapir.

input file with cells' descriptions and it returns the minimum area size which includes all cells. This minimum area is found through a recursive branch and bound search. We hierarchically generate tasks for each branch of the solution space. The state of the algorithm needs to be copied into each newly created task so they can proceed. This implies that additional synchronizations have been introduced in the code to maintain the parent state alive.

- `Health` simulates the Columbian Health Care System. It uses multilevel lists where each element in the structure represents a village with a list of potential patients and one hospital. The hospital has several double-linked lists representing the possible status of a patient inside it (waiting, in assessment, in treatment or waiting for reallocation). At each time step all patients are simulated according with several probabilities (of getting sick, needing a convalescence treatment, or being reallocated to an upper level hospital). A task is created for each village being simulated. Once the lower levels have been simulated synchronization occurs.

- `NQueens` computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an $n \times n$ chessboard such that none of the queens attack any other. It uses a backtracking search algorithm with pruning. A task is created for each step of the solution.

- `Sort` sorts a random permutation of n 32-bit numbers with a fast parallel sorting variation of the ordinary mergesort. First, it divides an array of elements in two halves, sorting each half recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. Tasks are used for each split and merge. When the array is too small, a serial quicksort is used to increase the task granularity. To avoid the overhead of quicksort, an insertion sort is used for very small arrays (below a threshold of 20 elements).

- `SparseLU` computes an LU matrix factorization over sparse matrices. A first level matrix is composed by pointers to small submatrices that may not be allocated. Due to the sparseness of the matrix, a lot of imbalance exists. Matrix size and submatrix size can be set at execution time. While a dynamic schedule can reduce the imbalance, a solution with tasks parallelism seems to obtain better results. In each of the sparseLU phases, a task is created for each block of the matrix that is not empty.

- `Strassen` uses hierarchical decomposition of a matrix for multiplication of large dense matrices. Decomposition is done by dividing each dimension of the matrix into two sections of equal size. For each decomposition a task is created.

There are two classes of input sizes for the benchmarks. Some, like Floorplan, require input files. For each of these we chose the largest provided. For the benchmarks with a parameter to adjust the size of the input, we attempted to choose a parameter size so that the fastest implementation ran on the order of seconds. The one exception was for `Fibonacci`. Because, as discussed further below, the Intel implementation required a large stack size, we stopped at 42. The exact inputs used are:

- `Alignment: -f prot.100.aa`

- FFT: `-n 335544320`
- Fibonacci: `-n 42`
- Floorplan: `-f input.20`
- Health: `-f large.input`
- NQueens: `-n 15`
- Sort `-n 335544320`
- SparseLU: `-n 100 -m 100`
- Strassen: `-n 8192`

The Intel compiler required increasing the stack size for `Fibonacci` and `FFT`. The implementation seemed to use significantly more stack space than the others, requiring setting `OMP_STACKSIZE` to `256M` for computing the 42nd Fibonacci number. `FFT` ran fine with an increase to `16M`. Clang required a similar increase in `OMP_STACKSIZE` for `SparseLU`, needing to be increased to `64M` to avoid stack overflows. GCC and the Tapir implementation required no such modifications.

For other implementations, we compare to GCC 7.1, Clang 4.0.1, and Intel 17.0.0. The machine used is a two socket Intel Xeon E5-2683 v3 machine with 132GB of memory running Linux 4.11.4. Each Xeon has 16 cores running at 2.1GHz, with 20M of L3 cache. All benchmarks were run using all 64 hyper-threads. 32-thread tests were run as a sanity check and showed qualitatively similar results. It's worth noting that each of the other implementations comes with its own runtime. In this sense, performance is a function of both the compiler and any optimizations it is able to perform, and the runtime. Understanding the interaction of these two parts is non-trivial as we will see when discussing results. Each compiler was run with the flags `-O3 -fopenmp`. Our compiler was also run with `-ftapir` to enable Tapir instructions to be lowered to a parallel runtime (in this case, the default Cilk runtime).

As mentioned in the Section 5, the gaps in the implementation changed program behavior in a couple of cases. In the case of `FFT`, we were forced to remove an unnecessary `task` pragma, as the current implementation doesn't insert the implicit barrier at the end of a `parallel` region. This was a one line fix in the benchmark, and would be fixed properly by handling the `parallel` and `single` pragmas correctly in our implementation.

The second change in program behavior due to our incomplete implementation was caused by the lack of the `critical` and `atomic` pragmas used in the `Floorplan` benchmark. The `critical` pragma ensures that only one thread can execute a particular statement at a time. This should be fixable in the implementation by adding a simple code-localized synchronization generation in the IR. Similarly, the `atomic` pragma can be addressed by retaining a pointer to the shared stack variable, much like existing OpenMP implementations. It is worth noting that this behavior was non-deterministic, and that roughly half of the time `Floorplan` still returned correct results. As we will see, this makes for an interesting trade-off given the witnessed performance increase.

We set a time-out of 10 minutes, as at least one implementation was always finishing within 10 seconds so anything running more than 60 times slower becomes irrelevant. Because

of time constraints correctness checks were not performed on every run so it is possible that some non-determinism was missed.

Each variant was run 10 times, with the height of the bars representing the mean and the error bars representing standard deviation. In cases where there was timeout or segmentation fault no time is reported and the run is marked as faulty.

7 RESULTS

In this section we discuss the results of running our evaluation on our implementation, and how its performance compared to the existing implementations listed in Section 6. Figures 6 and 7 give simple visualizations of performance. Each graph represents a single benchmark from the previously enumerated Barcelona benchmarks, and each bar represents one of the implementations.

As mentioned earlier regarding `Floorplan` our implementation returned correct results nondeterministically because of not having an implementation of the OpenMP `critical` and `atomic` pragmas. While ICC and Clang finished in roughly 1-2 seconds, the Tapir implementation was finished in 0.02 seconds and computed the correct result roughly half of the time. This raises interesting questions on the cost/benefit relation of the missing OpenMP pragmas.

GCC was the only culprit for timeouts. Recall that the timeout was set at 10 minutes, so any timeout means GCC was running at least approximately 60 times slower than the fastest implementation. For example, on the `FFT` benchmark, while our Tapir implementation was running in under 5 seconds, the GCC implementation was timing out at 600 seconds, so was at least 100 times slower. GCC results are omitted from graphs for the three timeout cases, `FFT`, `Fibonacci`, and `NQueens`.

Putting failures aside, performance for our implementation is quite strong, generally outperforming or matching the best of existing implementations. For benchmarks where the overhead-to-work ratio is high is where the Tapir + Cilk implementation shows significant benefit. For example, for the `Fibonacci` benchmark, our implementation runs significantly faster than any of the others. `NQueens` and `FFT` show similar behavior but to lesser degrees. In contrast, benchmarks that have a lower overhead to work ratio unsurprisingly differ less in performance.

7.1 Profiling

In this section we investigate *why* performance varies in the ways it does. While we have yet to identify every discrepancy in performance, we hope to get some ideas for why the performance of our implementation is generally better and where each of the implementations is spending the bulk of the execution time. Our primary tool for this task is profiling, acknowledging that many kinds information are difficult to infer from profiling, such as sources of contention, reasons for memory locality, etc.
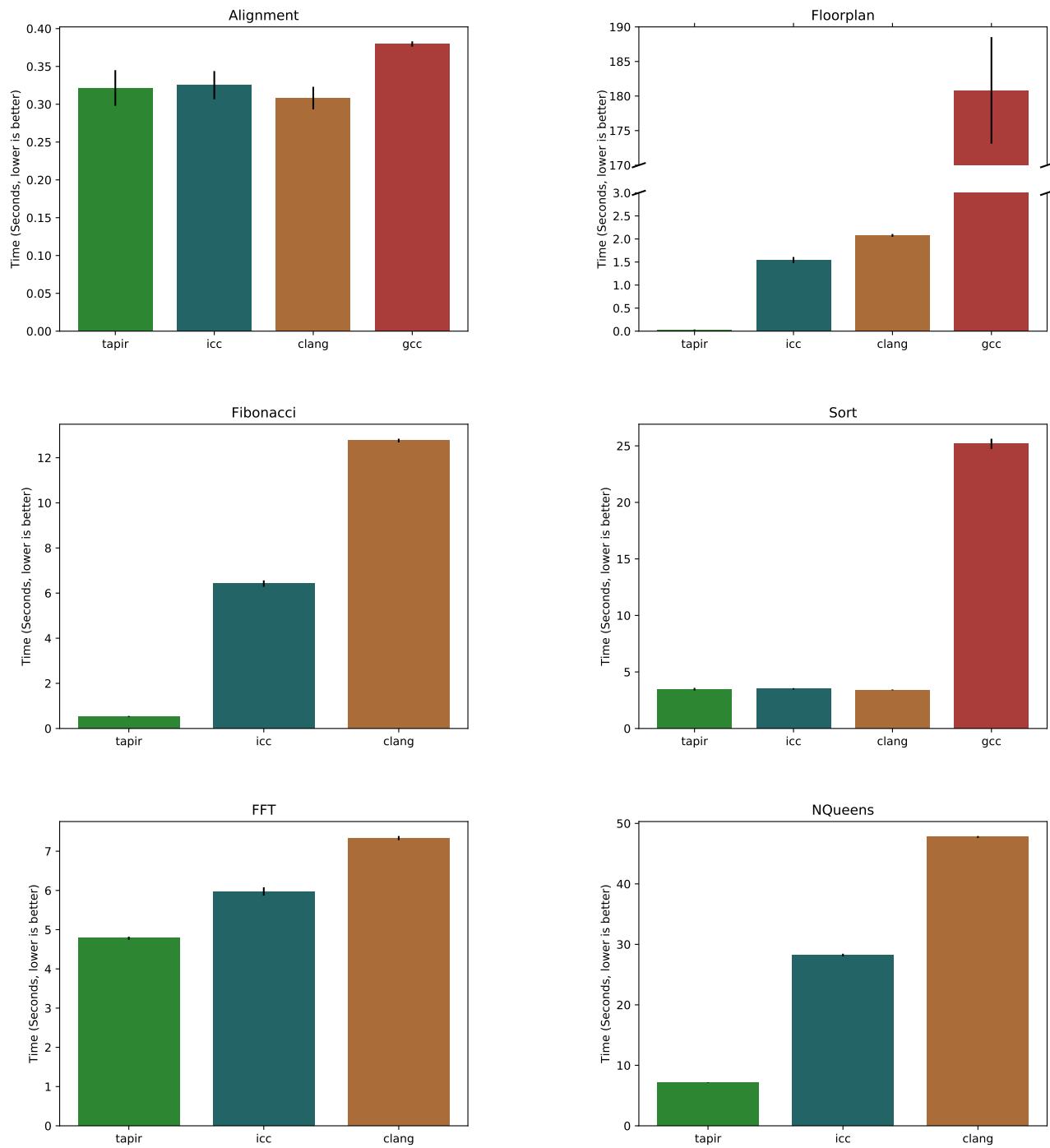
**Figure 6: Barcelona OpenMP Task Suite results**
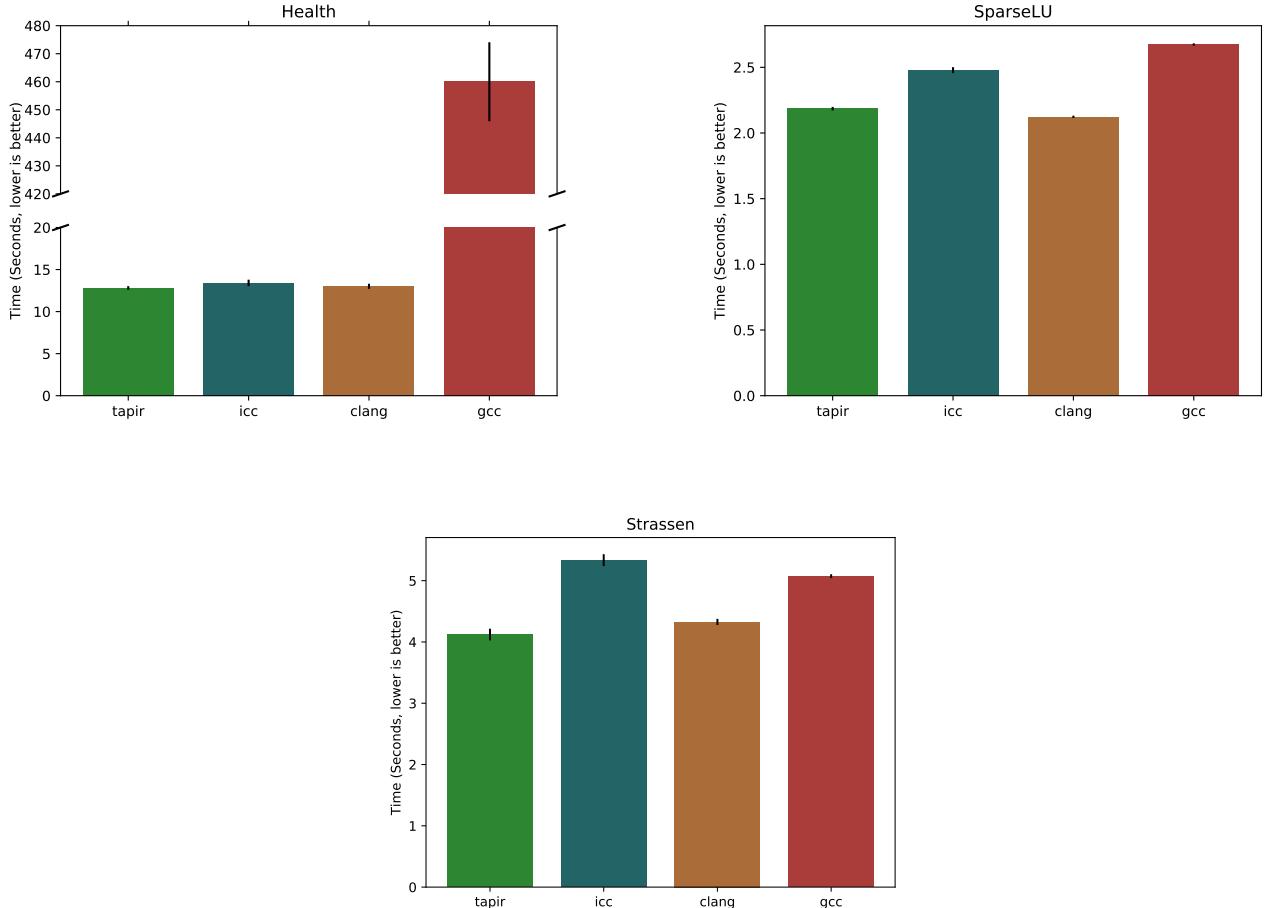
Figure 7: Barcelona OpenMP Task Suite results (continued)

It is worth noting again that the scope of the work that this paper reports is only the front end. All performance gains attributable to the existing Tapir back end and Cilk runtime represent prior work. We can separate reasons for performance discrepancies into a few categories:

- Front-end code generation;
- IR Optimizations; and
- Runtime efficiency.

While we would like to distinguish between these, it is difficult to do so with only profiling information. A more thorough investigation would require deep knowledge of each of the front ends and runtimes. We hypothesize that much of the disparity present in these results is more a function of runtime implementation differences. As justification for this hypothesis, consider the `Fibonacci` benchmark. There are no memory accesses to the heap, and the code leaves little room for optimization (algorithmic rewrites aside), so it is effectively

a benchmark of how fast a runtime can execute fork-join parallelism.

There are two cases we examine using the profiler. The first is the case when there is a large difference in run times between implementations. For these our hypothesis is that the slower implementations are for some reason spending more time in the runtime. As mentioned, it is difficult to understand exactly why an implementation is spending more time in the runtime without further investigation, but it is a useful sanity check nonetheless. For this case, we turn to the `Fibonacci` benchmark. We get roughly the following breakdowns of work-to-overhead ratio according to the Linux `perf` tool.

- `tapir`: 30% runtime overhead, 50% work, 20% other
- `clang`: 85% runtime overhead, 5% work, 10% other
- `icc`: 85% runtime overhead, 5% work, 10% other
- `gcc`: 100% runtime overhead, 0% work, 0% other

This supports our hypothesis that much of the slowdown incurred is overhead in the runtime. For Tapir the overhead is relatively small given the size of the work chunks, for `clang` and `icc` it is significantly larger, and for `gcc` it is overwhelming. `clang` and `icc` have effectively the same runtime, so the fact that their overheads are similar is unsurprising.

For the cases where performance is close between the implementations, such as `sparselu`, we expect run times to be dominated by actual task workloads. This expectation is confirmed by investigating performance counters for `sparselu` where every implementation spends roughly 90% of its time in the task bodies.

Another way we can discriminate the cause of performance discrepency is by disabling Tapir optimizations. This allows us to directly measure the effect that Tapir optimizations are having on runtime. For `fibonacci`, for example, disabling optimizations results in a roughly 2x performance loss. This is likely due to Tapir inlining the second task call and moving some memory operations into registers. This is strong evidence that Tapir optimizations are an important factor in the performance of compiled task-parallel code. Again, note that this compounds with runtime performance differences to give the measured performance reported in Section 7.

8 DISCUSSION

In this section we discuss the relevance of the results and in the context of the literature. As discussed in Section 1, there have been many programming models for implementing parallel programs. While this paper does not take a stance on which one *should* be used, it does accept that OpenMP is currently/potentially the most widely used model that has a notion of tasks. The goal of this work is to show that a single intermediate representation can be used to implement multiple programming paradigms with good performance. Schardl et al. showed that Tapir works well as an IR target for Cilk and surmised that it could be an effective IR for a large subset of OpenMP. This work has taken the first step in that direction.

One interesting aspect of this work is that it compiles a language extension intended for one runtime (OpenMP) to use another (Cilk). While in many ways this is a weakness of this work, it does open the possibility of utilizing this in a more general way to address many significant challenges such as performance portability. Having a parallel-aware IR enables the possibility of multiple high-level parallel languages and multiple parallel runtime targets in the same way that LLVM does for serial code. While the OpenMP specification's requirement to have access to runtime routines complicates things, for many programming models it seems at least conceivable to map them onto multiple different runtimes. This opens the possibility of mixing programming models at compile time by targeting the same runtime. For example, if both OpenMP and Cilk targeted Tapir instructions, one could have a large application where some parts were written using Cilk, while others used OpenMP, and the result would not suffer the incompatibility problems often seen today.

The surprising fact that program behavior wasn't changed significantly is worth discussing. We surmise that it is likely due to the semantics of variables borrowed from the Cilk codegen and their relation to the OpenMP specification of semantics. In particular, *OpenMP specifies a relaxed memory model for shared variables* in which writes by one thread need only be visible at to other threads at explicit or implicit flushes. This results in the Cilk implementation of only writing to a shared variable at the end of a parallel section being a valid implementation of OpenMP's specification. In contrast, existing OpenMP implementations sometimes write shared variables to memory on every write access. This is a potential for performance discrepancy because of differing implementation behavior. There is of course the fact that technically, by copying the value of variables back to the surrounding context even for `private` variables, one isn't following the specification. We surmise that in practice this hasn't had an effect because `private` variables are often used for inputs to calculations that are not accessed after the tasks complete, or for performance rather than their semantic properties. Also, `firstprivate` is the default data-sharing attribute for tasks.

9 FUTURE WORK

One of the important questions in this work is whether the Tapir instructions can be used to implement the full OpenMP semantics. Runtime calls aside, implementing the full semantics of OpenMP pragmas would be a significant undertaking.

For two examples, consider the problematic `atomic` and `critical` pragmas from the `Floorplan` benchmark. The `atomic` pragma would likely not be too difficult to translate from existing OpenMP codegen: one would need to simply maintain and then modify the contents of the original stack pointer to the shared variable. The `critical` section would likely be a little more difficult. Still, it should be possible to modify existing OpenMP code to block on that section.

Still, whether or not it's *possible* to implement these extra features is a separate question from whether or not it's *useful*. If implementing extra features require using special synchronization primitives, or calls into runtimes, or any other tool other than the Tapir instructions, then the compiler cannot reason about code using those features, and many of the advantages we've discussed, e.g., backend compatibility, generic optimization, etc., become moot, Features like the OpenMP `atomic` pragma don't require any changes to control flow, and therefore are likely to work well with Tapir analyses and optimizations. On the other hand, features like OpenMP's `critical` section *do* require changes to control flow that seem challenging to map onto Tapir's instructions. Furthermore, even if it is possible to implement features using the Tapir instructions, it may require such radical entangling of the code that even if analysis is technically possible it becomes infeasible.

An obvious first step towards better OpenMP coverage, other than the features listed above, is handling OpenMP `parallel for` loops. These are similar enough to `cilk_for`

loops in semantics that this also should be a relatively painless next step to adapt from the existing Cilk implementation.

While OpenMP and Cilk have similar fork-join style semantics that Tapir implements naturally, many allow a more flexible mechanism for synchronization, namely futures [4, 11, 25]. The primary difference between the fork-join parallelism represented by Tapir and these approaches is that child tasks can outlive their parents. Representing this class of parallelism in Tapir represents a significant challenge for future work.

One interesting area for future work would be combining Tapir with work on reasoning about locality in LLVM. For example, Hayashi et al. use address spaces in LLVM to reason about memory locality in partitioned global address space (PGAS) systems [9]. By combining reasoning about code concurrency and locality, one could potentially apply optimizations to better schedule and co-locate code and data.

10 RELATED WORK

There has been significant work on internal representations for parallel programs. These can be roughly broken into four categories.

First, compilers can attach metadata to an existing IR. For example, LLVM has a parallel loop metadata construct [22]. This has the benefit of being flexible and requiring minimal work, but historically can be lost during optimizations, and any code movement can break the semantics. The flexibility of this approach can also be viewed as a negative, as it can compromise what can otherwise be a simple, well defined semantics for the IR.

A second approach is to use intrinsic functions to define parallel tasks [18]. This, like the metadata approach, has the advantage of being flexible and relatively easy to implement, but at the cost of being less well defined. There has recently been proposals to standardize intrinsics extensions for LLVM to represent parallelism, but we would caution that a large, flexible set of intrinsics being added to an already only-partially defined [27] language could make reasoning about parallel programs, even in an IR setting, infeasible.

A third approach is to have a separate instruction set for parallelism. This is the approach taken by projects like HPIR [28], SPIRE [13], and INSPIRE [10]. This approach has the downside of being unable to re-use existing optimization and analysis infrastructure of an existing IR.

Finally, the last approach, and that taken by Tapir, is to implement parallel instructions as an extension of an existing IR. This allows for integration into existing analyses and optimizations. In the case of Tapir, this allows one to leverage years of development into program analysis and optimization, extending only where necessary.

While there is general agreement that there needs to be IR support for parallelism in parallel programs, there isn't consensus on which of these approaches is best. Approaches that are easier at first, such as metadata or intrinsic approaches, are similarly easy to extend but can become unwieldy. There is a reason that it's difficult to add instructions to LLVM: any added IR construct needs to be considered in many locations.

From debugging by printing out IR, to case analysis for different control flow constructs, having a proper set of instructions to denote parallelism has numerous advantages. Additionally, any hope of having a formal semantics for an IR depends on a simple, concise set of parallelism constructs, something that metadata and intrinsics approaches can promptly defeat.

A related approach to composing programming models is to keep the same code generation approach, but replace calls into built-in runtimes with calls into a shared runtime. This is the approach taken by Lithe, where Pan et al. compose programs using both GNU OpenMP and Intel TBB [21]. This approach succeeds in composing different programming models at run-time, but doesn't enable the static analyses that are enabled by Tapir.

11 CONCLUSION

We have shown that Tapir is an excellent IR target for OpenMP tasks. While not all of the semantics are covered, we have a path forward for many of them. We have shown that compiling to Tapir instructions allows for a straightforward compilation of OpenMP tasking programs to use the Cilk runtime system. We've also shown that this combination leads to better performance than existing OpenMP tasking implementations on the Barcelona OpenMP Tasking benchmark suite. Moving forward, we hope efforts like this one help to reduce the fragmentation of parallel runtimes, as well as make it easier to write optimization for parallel programs.

REFERENCES

[1] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.* 20 (March 2009), 404–418. Issue 3.

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 66.

[3] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69.

[4] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[5] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *ICPP '09: Proceedings of the 38th International Conference on Parallel Processing*. IEEE, 124–131.

[6] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[7] Tarek El-Ghazawi and Lauren Smith. 2006. UPC: unified parallel C. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 27.

[8] William Gropp and Ewing Lusk. 2005. Using mpi-2. In *12th European PVM/MPI Users' Group Meeting-Recent Advances in Parallel Virtual Machine and Message Passing Interface*.

[9] Akihiro Hayashi, Jisheng Zhao, Michael Ferguson, and Vivek Sarkar. 2015. LLVM-based communication optimizations for PGAS programs. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 1.

[10] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. INSPIRE: The Insieme parallel intermediate representation. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 7–18.

[11] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 6.

[12] Laxmikant V Kale and Sanjeev Krishnan. 1993. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, Vol. 28. ACM, 91–108.

[13] Dounia Khaldi, Pierre Jouvelot, François Irigoin, and Corinne Ancourt. 2012. SPIRE: A methodology for sequential to parallel intermediate representation extension. In *17th Workshop on Compilers for Parallel Computing (CPC 2013)*.

[14] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.

[15] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86.

[16] Jason Merrill. 2003. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*. 171–179.

[17] Nicholas Moss. 2016. Kokkos GPU Compiler. (Jul 2016). http://www.osti.gov/scitech/servlets/purl/1310548

[18] Nick Moss, Kei Davis, and Patrick McCormick. 2016. The ARES high-level intermediate representation. In *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*. IEEE Press, 32–36.

[19] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. 1996. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.".

[20] OpenMP Architecture Review Board. 2015. OpenMP Application Program Interface Version 4.5. (November 2015). http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[21] Heidi Pan, Benjamin Hindman, and Krste Asanović. 2010. Composing Parallel Software Efficiently with Lithe. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 376–387. https://doi.org/10.1145/1806596.1806639

[22] LLVM Project. 2015. *LLVM Language Reference Manual*. http://llvm.org/docs/LangRef.html

[23] Tao B Schardl, William S Moses, and Charles E Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 249–265.

[24] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[25] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–8.

[26] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC—first experiences with real-world applications. *Euro-Par 2012 Parallel Processing* (2012), 859–870.

[27] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 427–440.

[28] Jisheng Zhao and Vivek Sarkar. 2011. Intermediate language extensions for parallelism. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*. ACM, 329–340.