Analysis of Explicit vs. Implicit Tasking in OpenMP using Kripke

Charles Jin
Reservoir Labs
New York, NY
jin@reservoir.com

Muthu Baskaran
Reservoir Labs
New York, NY
baskaran@reservoir.com

***Abstract*—Dynamic task-based parallelism has become a widely-accepted paradigm in the quest for exascale computing. In this work, we deliver a non-trivial demonstration of the advantages of explicit over implicit tasking in OpenMP 4.5 in terms of both expressiveness and performance. We target the Kripke benchmark, a mini-application used to test the performance of discrete particle codes, and find that the dependence structure of the core "sweep" kernel is well-suited for dynamic task-based systems. Our results show that explicit tasking delivers a 31.7% and 8.1% speedup over a pure implicit implementation for a small and large problem, respectively, while a hybrid variant also underperforms the explicit variant by 13.1% and 5.8%, respectively.**

***Index Terms*—Tasks, Runtime, OpenMP, Kripke, Optimizing Compiler.**

I. Introduction

As the race for exascale computing gains momentum, node architectures have increasingly moved toward compositions of heterogeneous processors that form a computational unit. With clock speeds quickly approaching foreseeable limits set by physical phenomena such as cooling and global power constraints, the focus has turned to improving performance by increasing parallel throughput across cores [1]. To fully exploit of the computational capabilities of tomorrow's systems, it has thus become increasingly important that software is designed to expose the maximal amount of parallelization to the hardware, while also maintaining resilience and managing other optimizations such as locality.

Dynamic task-based parallelism has become a widely-accepted paradigm for addressing such demands. In this approach, units of computation are separated into dynamically spawned tasks with dependences expressed as a Directed Acyclic Graph. A separate software layer, called the runtime system, dynamically schedules the tasks to take advantage of the various available computing resources while respecting dependences and ensuring a correct execution path.

There are multiple dynamic task-based runtimes being developed in the community for exascale systems, such as Open Community Runtime (OCR) [2], Concurrent Collections (CnC) [3], Legion [4], Charm++ [5], Habanero [6], PaRSEC [7], X10 [8], DARMA [9], and others. In this paper, we focus on OpenMP [10], which has emerged as a viable framework for programming on systems with shared-memory semantics. In particular, OpenMP exposes an intuitive API for parallel programming that maintains much of the original serial syntax. While OpenMP originally focused on do-all parallelism, where the compiler assumes that loop iterations can be safely work-shared across threads and creates an implicit task for each iteration, OpenMP 3.0 added support for explicit tasks, and later versions continue to introduce features that augment the performance and flexibility of explicit tasking.

This work explores the perceived performance gap between explicit and implicit tasking in OpenMP 4.5 by implementing several variants of the Kripke benchmark with different performance characteristics. In particular, though we observe higher overhead costs for explicit tasking as expected, the flexibility of expressing dependences between tasks explicitly allows the runtime to extract a higher level of parallelism than is available to implicitly-defined tasks.

II. Background

A. OpenMP

OpenMP is an open-standard programming model that provides an API for shared-memory parallelism, primarily via compiler directives. The OpenMP API follows the fork-join model of parallelism, whereby a master thread forks many slave threads and allocates each thread some portion of work. While OpenMP presents a rich set of APIs for implicit tasking, and, in particular, loop-based for-all parallelism, we focus on OpenMP's support for explicit task-based parallelism in this work, specifically the `task` construct introduced in OpenMP 3.0.

The main concepts provided by OpenMP for explicit tasking are: (1) single task definition via the `task` directive, (2) synchronization using the `taskgroup`, `taskwait`, and `taskyield` directives, and (3) imposing scheduling constraint by expressing dependences through the `depends` clause. Note that while the runtime is responsible for exploiting parallelism in OpenMP, it is still the programmer's responsibility to ensure all permissible paths maintain correctness using synchronization and dependences.

In particular, dependences in OpenMP are defined via data dependences as opposed to between tasks. In other words, each task specifies the set of data locations it reads and writes, and

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-SC0018480.

the runtime is responsible for ensuring that any reordering of tasks produces the same semantics as a serial execution. For instance, consecutive tasks that write to the same location cannot be reordered; however, the runtime is free to reorder consecutive tasks that read from the same location.

One main benefit of OpenMP is that it is intuitive and easy to develop. Because removing OpenMP directives should yield a correct serial program, development, debugging, and maintenance are vastly easier than frameworks that require restructuring code to expose parallelism to the runtime; oftentimes, developing in OpenMP is as simple as writing a serial code and then adding in directives where necessary. However, this simplicity comes at a tradeoff, namely in that the programming model is not intended for distributed memory systems. Indeed, much of the concision in OpenMP is due to the lack of a need for explicit or implicit data movement primitives. We also explore how this terseness leads to subtle performance considerations that result from code structure.

B. Kripke

Our main target is the Kripke benchmark [11], a mini-application developed by LLNL as a proxy for 3D Sn deterministic particle transport codes. Kripke is designed to allow programmers to explore how data layout affects performance of particle transport codes by supporting multiple nestings of the main program data.

The dependences between tasks in the main kernel follow a "wavefront" pattern, allowing for levels of parallelism that are difficult to exploit in a traditional do-all parallelism model, but well-suited for frameworks with dynamic task-based runtimes.

We implemented two microkernels following the same wavefront dependence pattern to help tune performance in a more controlled environment. We use these microkernels to demonstrate the characteristics of the wavefront dependence pattern as well as our basic implementation techniques in the following section.

III. Implementations

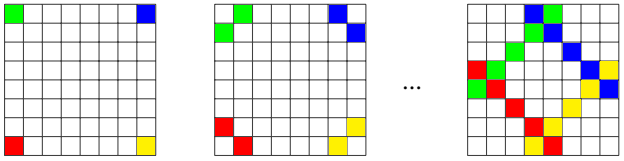
A. Gauss-Seidel Microkernel

The Gauss-Seidel method is an iterative method for solving a linear system of equations (Fig. 1). We set up the problem such that the iteration domain is a two-dimensional square matrix. A point (i, j) in the iteration space depends on the values of the points $(i-1, j)$ and $(i, j-1)$ in the current timestep and the points (i, j), $(i+1, j)$, and $(i, j+1)$ in the previous timestep. Thus, in the first timestep, the only point that can be calculated is $(0, 0)$; in the second timestep, the points $(1, 0)$ and $(0, 1)$ can be calculated; and so forth, where successive diagonals defined by $\{(i, j) \mid i + j = n\}$ can be calculated in the n^{th} timestep (Fig. 2).

The basic example in Fig. 3 demonstrates the difficulty of using do-all parallelism to solve wavefront dependence patterns. The amount of work across timesteps ranges from 1 to N, where N is the dimension of the matrix. However, there is an implicit barrier at the end of the worksharing construct `for`. As a result, this prevents any parallelism across

```
for n = 0 .. niter-1:  // iteration counter
  for i = 0 .. N-1:
    for j = 0 .. N-1:
      A[i][j] = C0*A[i][j] +
                C1*(A[i+1][j] + A[i][j+1] +
                    A[i-1][j] + A[i][j-1])
```

Fig. 1: Gauss-Seidel microkernel. `C0` and `C1` are constants derived from the system of linear equations, and `N` is the dimension of the square matrix. Notice that the update step only depends on `A[i-1][j]` and `A[i][j-1]` from the current iteration; all other values are from previous iterations.



(a) step 0 (b) step 1 (c) step 4

Fig. 2: A two-dimensional wavefront dependence pattern with four independent sweeps; each grid point has four subtasks. In the Gauss-Seidel microkernel, only one sweep is performed and each grid point has only one subtask.

diagonals (Fig. 4). To implement a properly skewed loop to expose maximal parallelism also requires additional work on the part of the programmer and moves away from the ideal of serial syntax with parallel performance.

Conversely, in a task-based model (Fig. 5), finer-grained parallelism is available to the runtime. In particular, the program does not block on the line that calls `doWork(i, j)`, as the `task` directive packages the invocation in a deferrable

```
for n = 0 .. niter-1:  // iteration counter
  for t = 0 .. 2N-2:   // timestep counter
    #pragma omp for
    for i + j = t:
      doWork(i, j)
```

Fig. 3: Sweep algorithm, variant one: do-all parallelism. In this version, each iteration of the inner loop is fully independent and can thus be executed in parallel.

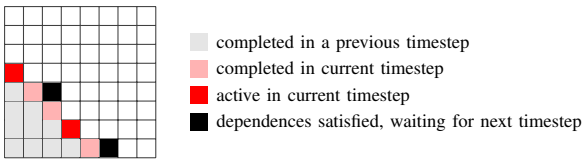


Fig. 4: An example execution path for the Gauss-Seidel sweep using the algorithm in Fig. 3. In this case, two tasks in the next timestep have already satisfied their dependences but are waiting on the current timestep ($t = 4$) to complete. Without dynamic task-based parallelism, it is difficult to extract full parallelism from the sweep algorithm.

```
for n = 0 .. niter-1:
  for i = 0 .. N-1:
    for j = 0 .. N-1:
      #pragma omp task \
        depends(in:  A[i-1,j,n], \
                     A[i,j-1,n], \
                     A[i+1,j,n-1], \
                     A[i,j+1,n-1]) \
        depends(out: A[i,j,n])
      doWork(i, j)
```

Fig. 5: Sweep algorithm, variant two: task-based parallelism. By spawning tasks with explicit dependencies, extracting parallelism is left to the runtime.

```
for n = 0 .. niter-1:
  for t = 0 .. 2N-2:
    for i + j = t:
      #pragma omp task \
        depends(in:  A[i-1,j,n], \
                     A[i,j-1,n], \
                     A[i+1,j,n-1], \
                     A[i,j+1,n-1]) \
        depends(out: A[i,j,n])
      doWork(i, j)
```

Fig. 6: Sweep algorithm, variant three: task-based parallelism, loop skewed. This combines the outer loop of the do-all variant with the explicit tasks of the task-based variant.

task. Thus, in theory, the loop simply spawns all the tasks with the proper dependences, and it is up to the runtime to schedule the tasks for execution.

A problem occurs when the number of threads is small relative to the loop dimensions. Specifically, it is not true that all tasks in a region will be made available to the runtime for scheduling. Rather, the master thread is allowed to stop spawning tasks and switch to execution if the backlog of deferred tasks is large. Performance begins to degrade significantly when the deferred tasks do not exhibit high levels of parallelism, as the master thread may pause before the parallel tasks are spawned. Thus, it is actually crucial that tasks are spawned in an order that yields sufficient parallelism should the master thread switch to execution (Fig. 6).

Note that for both task-based parallelism examples, we have augmented the input matrix `A` with the iteration count n. This is a syntactic representation to clarify the dependences, but in practice, the third dimension can be dropped as the order in which the tasks are spawned guarantees that the dependences between timesteps are satisfied.

B. Kripke Microkernel

The Kripke microkernel consists of eight independent sweeps from each corner of a 3-dimensional matrix. Because the sweeps are independent, the outer loop can be parallelized fully using do-all parallelism, however, the inner loops suffer from the same bottleneck effect as the Gauss-Seidel microkernel (Fig. 7). The call to `sweep` is identical in structure to the

```
#pragma omp for
for dir = 0 .. 7:
  sweep(dir)
```

Fig. 7: Simplified Kripke microkernel code. The inner loop is identical to the 3-dimensional analogue of the Gauss-Seidel microkernel.

Gauss-Seidel microkernel; refer to Fig. 2 for a 2-dimensional visualization of the dependences.

It is worth exploring several possible variants of this structure. Assume that this code is to run on a machine with 64 cores. In Fig. 8, the outer loop is executed by 8 threads, and each thread spawns its own team of 8 threads for the tasks spawned in `sweep`. However, there is a tradeoff between the overhead of having nested `parallel` regions and improved locality of having a dedicated thread team for each direction.

In Fig. 9, the `parallel` construct has spawned a team of 64 threads, and the following `for` construct provides that only one thread executes each iteration of the for loop. As a result, tasks for each direction are spawned in parallel, but the threads are shared between each direction.

Finally, in Figure 10, we permute the loop over the indexes with the loop over the directions. The effect is that tasks are spawned in the globally "optimal" order and reuse one thread pool, however the spawning can only be done by one thread.

```
#pragma omp parallel num_threads(8)
#pragma omp for
for dir = 0 .. 7:
  #pragma omp parallel num_threads(8)
  #pragma omp single
  sweep(dir)
```

Fig. 8: Outer direction loop, variant one: 8 teams of 8 threads. Each `sweep` is executed in parallel with its own team of 8 threads.

```
#pragma omp parallel num_threads(64)
#pragma omp for
for dir = 0 .. 7:
  sweep(dir)
```

Fig. 9: Outer direction loop, variant two: one team of 64 threads. The 8 sweep directions share the same team of 64 threads.

```
#pragma omp parallel num_threads(64)
#pragma omp single
for steps = 0 .. max_steps:
  for i + j + k = steps:
    for dir = 0 .. 7:
      #pragma omp task depend(...)
      doWork(i, j, k, dir)
```

Fig. 10: Outer direction loop, variant three: one team of 64 threads with fused inner loop.

C. Kripke Application

The full Kripke application is hard-coded for the 3D Kobayashi radiation benchmark, problem 3i [12], which is a specific case of the steady state form of the Boltzmann transport equation in three dimensional geometry. The objective is to solve for angular flux, $\psi(r, \Omega, E)$, where $r \in \mathcal{D} \subset \mathbf{R}^3$ is position, $\Omega \in \mathcal{S}^2$ is direction (in the unit sphere in $\mathbf{R}^3$), and $E \in (0, \infty)$ is energy.

First, the problem is discretized using the standard discrete-ordinates method. This is accomplished (1) for the spatial operator r using a Diamond Difference approximation over Z zones, (2) for the integral over Ω with a quadrature rule of D points and weights, and (3) for the energy variable E by binning into G groups. In order to coarsen the granularity of computation, standard implementations of Kripke further partition the zones, directions, and groups into "zone sets", "direction sets", and "group sets", respectively; this allows operations to be implemented over chunks of data, improving locality and reducing overhead.

The problem is then solved using an iterative method, with each iteration consisting of two steps: (1) a right hand side calculation that consists of several matrix operations which are fully parallelizable in the space domain, and (2) a matrix inversion using the diamond differences method implemented via the sweep algorithm detailed in the Kripke microkernel.

Our implementation of Kripke closely follows the Charm++ port of Kripke, and overdecomposes along zonesets in the spatial domain, yielding the sweep with task dependences in three dimensions.

In addition to a basic serial code, we implement the following variants of the full Kripke application, most of which which can be seen as different pairings between (1) the two variants for the outer direction loops described in the Kripke microkernel and (2) the three inner loops of the sweep algorithm detailed in the Gauss-Seidel microkernel. Pseudocode for selected implementations can be found in the Appendix.

1) Tasking using 8 by 8 thread teams.
2) Tasking using 8 by 8 thread teams, inner sweep loops skewed.
3) Tasking using one team of 64 threads.
4) Tasking using one team of 64 threads, inner sweep loops skewed.
5) Tasking using one team of 64 threads, inner sweep loops fused and skewed.
6) Do-all parallelism.
7) Hybrid do-all parallelism using the `taskgroup` construct to synchronize diagonals.

IV. Results

We present experimental results for the full Kripke application below. These results highlight the performance advantages of a dynamic task-based runtime where the potential for parallelism is high, but complex dependences limit the amount of do-all parallelism. We ran our experiments on an 8-core (16 threads) quad socket Intel Xeon (Ivy Bridge) server. Code was compiled with GCC 7.3 (OpenMP 4.5). OpenMP was limited to 64 threads.

```
for iter = 1 .. niter-1:
  #pragma omp for
  for i, j, k;
    rhs(i, j, k)

  #pragma omp for
  for dir = 0 .. 7:
    sweep(dir)
```

Fig. 11: Simplified Kripke code. Each call to `rhs` is fully independent, while the `sweep` portion is identical to the Kripke microkernel.

TABLE I: Kripke - Small Problem[a]

Implementation Version	Performance Metrics[b]		
	Execution time (seconds)	*Average CPU utilization*	*Instructions per cycle*
serial, no OpenMP	17.036	0.998	2.15
tasking, 8 by 8	3.327	14.248	1.07
tasking, 8 by 8, skewed	3.966	11.270	1.11
tasking, 64	2.894	22.412	0.78
tasking, 64, skewed	2.760	16.287	0.98
tasking, 64, skewed, fused	2.924	23.143	0.77
do-all	4.044	15.956	0.96
hybrid, do-all + taskgroup	3.178	15.486	1.03

[a]Small problem has 16 energy groups, 64 direction groups, 16x16x16 zones, and 10 iterations.
[b]Results are the average of 10 runs as reported by `perf`.

TABLE II: Kripke - Large Problem[a]

Implementation Version	Performance Metrics[b]		
	Execution time (seconds)	*Average CPU utilization*	*Instructions per cycle*
serial, no OpenMP	485.348	0.997	2.07
tasking, 8 by 8	75.245	12.523	1.25
tasking, 8 by 8, skewed	70.841	14.219	1.22
tasking, 64	73.466	14.986	1.09
tasking, 64, skewed	75.152	12.756	1.20
tasking, 64, skewed, fused	73.740	14.788	1.10
do-all	77.073	13.318	1.18
hybrid, do-all + taskgroup	75.204	14.296	1.12

[a]Large problem has 32 energy groups, 128 direction groups, 32x32x32 zones, and 10 iterations.
[b]Results are the average of 10 runs as reported by `perf`.

Both problems were decomposed into zonesets of 4x4x4. With 8 simultaneous sweeps, this yields a total of 512 tasks per iteration. The decompositions were selected independently to yield the fastest solution (i.e. execution time) across all implementations and decompositions; in other words, we first identified the best decomposition for each implementation,

then took the implementation with the best overall performance and used the corresponding decomposition for our benchmark results. As it turns out, this decomposition is either optimal, or very close to optimal, for all implementations.

A. Explicit vs. Implicit Tasking

We now present our main result, which addresses the performance of explicit versus implicit tasking in Kripke. Generally speaking, spawning implicit tasks via the `for` construct is preferable to spawning explicit tasks within the loop using the `task` construct, as the overhead costs for spawning explicit tasks can outweigh the performance improvements, especially when the parallelism is too fine. Conversely, overhead costs are incurred only the first time the `for` construct is encountered, improving the scalability with loop width.

In Kripke, the wavefront nature of the sweep means that any implementation using do-all parallelism (i.e. `for`) will introduce artificial synchronization points, reducing the amount of parallelism extracted. However, this theoretical improvement in performance must be compared with the additional overhead of spawning explicit tasks.

The results in Tables I and II demonstrate that in both problems, there were significant gains from explicit tasking. The fastest pure tasking variant in the small problem was 31.7% faster than the pure do-all variant and 13.1% faster than the hybrid variant. For the large problem, the fastest pure tasking variant outperformed the pure do-all variant by 8.1% and the hybrid variant by 5.8%. The large problem also saw significantly higher throughput in the tasking variant at 1.22 IPC (instructions per cycle) for the fastest tasking variant versus 1.18 IPC for the pure do-all variant and 1.12 IPC for the hybrid variant. Note that this measurement is less predictive of performance for the smaller problem because overhead is more significant.

Finally, because our tests were run on a NUMA machine with four sockets, performance in the key sweep kernel could be affected by OpenMP thread affinity. This is addressed in the implicit case by pinning the threads responsible for a given direction to the same socket, as no data is shared during the sweep between directions (Fig. 16). Conversely, OpenMP 4.5 does not support task affinity, so a given task could be picked up by a thread on any socket, rendering pinning useless. As such, we do not use affinity for the explicit tasking variants, allowing threads to migrate freely.

In the following sections, we delve deeper into the structural differences between the variants and how these implementation decisions affected performance.

B. Overhead and Scaling in OpenMP

One question central to tuning the performance of parallel programs is determining the optimal level of granularity. If the granularity is too fine (i.e. the amount of time spent in a task is small), then the overhead will subsume any additional parallelism; if the granularity is too coarse, then the program will not be extracting the full amount of parallelism available.

We implement several variants to demonstrate this difference. In particular, some tasking implementations and the do-all implementations use 8 teams of 8 threads. This has high initial overhead costs due to the nested parallel regions, in exchange for the guarantee that threads will be evenly distributed across directions (each of which corresponds to an equal share of computation). Conversely, the tasking implementations with one team of 64 threads will have lower overhead costs but leaves the allocation of resources up to the runtime. Additionally, we implement two versions of do-all parallelism. In the first, the outer direction loop spawns 8 teams, while the inner loop iterates over the sweep diagonals, spawning a separate team for each step. The second is a hybrid approach, where the outer direction loop again spawns 8 teams, but the inner loop reuses the same thread team for each iteration by spawning explicit tasks (with no dependences) and synchronizing with the `taskgroup` directive.

We would expect that implementations that minimize overhead costs will perform better for the smaller problem, as the granularity of the tasks is finer by design. The results for the explicit tasking variants in Tables I and II align with these intuitions, as the best explicit tasking variant uses one team of 64 threads in the small problem versus 8 teams of 8 threads in the large problem.

Additionally, the hybrid do-all version that uses explicit tasks for the inner sweep loop outperforms the pure do-all variant in both problems, though the gap is significantly narrower in the larger problem (21.4% vs. 2.4% speedup). This is because the reduction in overhead stays roughly constant while the size of the problem is significantly larger.

Finally, an incidental benefit of both problems yielding the same decomposition is that we are able to compare how the performances of the implementations scale over different levels of granularity. As a rough estimate, a task in the large problem is approximately 32 times larger than a task in the small problem. However, note that the serial implementation takes about 28.5 times longer on the large problem, whereas the OpenMP versions all achieve better scaling (Table III).

In fact, two OpenMP versions perform quite well with respect to this metric: the skewed 8 by 8 tasking version and the plain do-all version. This is to be expected, as both versions have significant overhead costs but exhibit high levels of parallelism that are more easily observed when the tasks are computationally heavy.

Care should be taken when extrapolating these results for larger (or smaller) problems, as the optimal decomposition may not hold constant as the problem scales.

C. Loop Optimizations

We also experiment with two common loop optimization techniques to explore their interaction with OpenMP.

Loop skewing involves rewriting a nested loop such that the inner loop of the transformed code can be executed fully in parallel. In the sweep algorithm, this is done primarily by iterating over diagonals in the sweep, i.e. Fig. 3 (skewed) versus Fig. 5 (unskewed). This exposes maximal parallelism

TABLE III: Scaling on Kripke[a]

Implementation Version	Performance Metrics[a]	
	Execution time (Large / Small)	*Average CPU utilization (Large / Small)*
serial, no OpenMP	28.49	1.00
tasking, 8 by 8	22.62	0.88
tasking, 8 by 8, skewed	17.86	1.26
tasking, 64	25.39	0.67
tasking, 64, skewed	27.23	0.78
tasking, 64, skewed, fused	25.22	0.64
do-all	19.06	0.83
hybrid, do-all + taskgroup	23.66	0.92

[a]Results taken from Tables I and II.

in the inner loop, though the work is not as evenly distributed across the outer loop iterations, and the memory access pattern for C-style arrays is less optimal.

This is confirmed by our results. Skewing is only beneficial (1) for the large problem, in the task variant with 8 teams of 8 threads, and (2) for the small problem, in the task variant with 64 threads. In the first case, the improved parallelization is worth the overhead costs when the gain is greater due to the increased size of the problem, as can be seen from the lower IPC but improved execution time and average processor utilization; in the second case, skewing allows for greater throughput, which is the primary weakness of the single team variant, resulting in significantly greater IPC (0.98 vs. 0.78).

Loop fusion, also known as **loop jamming**, involves combining adjacent loop bodies that have the same number of iterations (see Fig. 17 for sample fused code). While this generally reduces memory locality, in our case, we combine the iterations over the 8 directions into one loop, which allows the directions to proceed in parallel without needing to nest parallel regions.

Similar to loop skewing, fusion has the largest benefit when applied to implementations that suffer from lack of parallelization. Indeed, we see that in the larger problem, the skewed 64-thread variant of tasking has improved average processor utilization and execution time, though the IPC suffers.

It is important to note that these loop optimization techniques are necessary only insofar as OpenMP does not allow for the user to specify when to stop adding tasks to the task graph. In particular, OpenMP provides the `priority` clause for tasks as a hint to the scheduler, however, this does not improve performance when the master task defers spawning to pick up work instead. Allowing the user to spawn all tasks within a given region might hurt performance if the number of spawned tasks is high, but in this case–i.e. each full iteration of the sweep, the delay would be minimal, and the runtime would be exposed to the maximal amount of parallelism without the overhead of nested parallel regions.

V. Related Work

Several works have explored the explicit tasking model in OpenMP since the feature was introduced in version 3.0. [13] considers applications that tend to generate unbalanced task graphs. Such problems are natural targets for task-based parallelism, as the runtime is able to react dynamically as tasks are created and completed. Performance is measured using the UTS problem, which counts the number of nodes in an implicitly-defined tree. Because the amount of computation per task is small, they find that a thread-based implementation where threads explicitly coordinate load exhibits better performance due to lower overhead.

Other studies have directly compared explicit and implicit tasking in OpenMP and concluded that explicit tasking yields better performance and lower overhead than implicit tasking for certain benchmarks. The kernel in [14] is a nested loop of four levels, none of which carry any dependencies. By tuning the granularity of the tasks, they are able to achieve a significant speedup versus a `for` version that parallelizes the outer loop using dynamic scheduling. [15] also considers a problem with fully independent nested loops and finds that explicit tasks outperform due to the overhead of nested parallel regions.

Finally, [16] implements a version of Kripke in OpenMP that specifically targets GPUs, evaluating OpenMP 4.5 from the perspective of ease of porting and performance versus handwritten CUDA. They find that the state-of-the-art implementation of Kripke uses several C++14 features that are not supported by OpenMP 4.5, though despite these difficulties, they conclude that performance portability is achievable in OpenMP.

VI. Conclusion and Future Work

We have substantiated the utility of explicit task-based parallelism in OpenMP using a full implementation of Kripke as a benchmark. In particular, pure explicit tasking variants as well as a hybrid variant outperform pure implicit parallelism variants, demonstrating that the flexibility of explicit tasks allows the runtime to exploit higher levels of parallelism than is possible with implicit tasking alone. Kripke also proved to be an excellent benchmark, as it was relatively easy to implement while being of greater interest to the scientific computing community, and the data layout and complex task dependences were well-suited to exploring different programming paradigms and, in particular, task parallelism.

Though the explicit tasks already outperform implicit tasks, there are still several avenues of improvement to this approach. First, OpenMP is continuing to improve on support for its tasking framework, and in particular, the introduction of the `affinity` hint for tasks in version 5.0 could improve locality. Additionally, introducing more flexibility in specifying task spawning and scheduling behavior would obviate much of the need to rewrite code to reduce overhead and expose more parallelism.

Finally, we intend to expand on this work in two ways. First, many of the loop optimizations here could be offloaded to a compiler. As such, we intend to augment R-Stream [17] [18], a source-to-source compiler that performs advanced loop optimizations techniques, with support for OpenMP, thereby allowing programmers to work with completely serial code

without sacrificing performance. Secondly, these results are currently limited to shared memory architectures. In order to allow applications to target distributed memory systems, we intend to explore layering a second runtime on top of the OpenMP runtime to manage the distributed memory semantics, including data movement and synchronization, while using OpenMP for on-node parallelism. Building support for both types of parallelization into R-Stream would enable programmers to write truly architecture-independent serial code that can then be compiled to fully exploit any target architecture by taking into account heterogeneity and performance at both the intra- and inter-node level.

REFERENCES

[1] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science–VECPAR 2010*, 2011.

[2] T. Mattson, R. Cledat, V. Cave, V. Sarkar, Z. Budimlic, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tarislar, J. Teller, and N. Vrvilo, "The Open Community Runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, September 2016.

[3] Intel, "Concurrent Collections," http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking Storage and Analysis (SC'12)*, Salt Lake City, UH, USA, November 2012.

[5] M. P. Robson, R. Buch, and L. V. Kale, "Runtime coordinated heterogeneous tasks in Charm++," in *Proceedings of the Second International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2. Piscataway, NJ, USA: IEEE Press, 2016, pp. 40–43. [Online]. Available: https://doi.org/10.1109/ESPM2.2016.7

[6] 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*, ser. PPPJ '11. New York, NY, USA: ACM, 2011, pp. 51–61. [Online]. Available: http://doi.acm.org/10.1145/2093157.2093165

[7] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in PaRSEC: A data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:8. [Online]. Available: http://doi.acm.org/10.1145/3148226.3148233

[8] IBM, "X10," http://x10-lang.org/.

[9] Sandia National Laboratories, "DARMA," https://share-ng.sandia.gov/darma/.

[10] OpenMP Architecture Review Board, "The OpenMP specification for parallel programming," 2015, http://www.openmp.org/.

[11] A. J. Kunen, T. S. Bailey, and P. N. Brown, "Kripke - a massively parallel transport mini-app," in *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method (ANS MC '15)*, Nashville, Tennessee, Apr. 2015.

[12] K. Kobayashi, N. Sugimura, and Y. Nagaya, "3D radiation transport benchmark problems and results for simple geometries with void region," *Progress in nuclear energy*, vol. 39, no. 2, pp. 119–144, 2001.

[13] S. L. Olivier and J. F. Prins, "Evaluating OpenMP 3.0 run time systems on unbalanced task graphs," in *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, ser. IWOMP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 63–78. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02303-3_6

[14] A. B. Adcock, B. D. Sullivan, O. R. Hernandez, and M. W. Mahoney, "Evaluating OpenMP tasking at scale for the computation of graph hyperbolicity," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 71–83.

[15] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel, "An experimental evaluation of the new OpenMP tasking model," in *Languages and Compilers for Parallel Computing*, V. Adve, M. J. Garzarán, and P. Petersen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 63–77.

[16] I. Karlin, T. Scogland, A. C. Jacob, S. F. Antao, G.-T. Bercea, C. Bertolli, B. R. de Supinski, E. W. Draeger, A. E. Eichenberger, J. Glosli, H. Jones, A. Kunen, D. Poliakoff, and D. F. Richards, "Early experiences porting three applications to OpenMP 4.5," in *OpenMP: Memory, Devices, and Tasks*, N. Maruyama, B. R. de Supinski, and M. Wahib, Eds. Cham: Springer International Publishing, 2016, pp. 281–292.

[17] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, "R-Stream compiler," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1756–1765.

[18] M. Baskaran, B. Pradelle, B. Meister, A. Konstantinidis, and R. Lethin, "Automatic code generation and data management for an asynchronous task-based runtime," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, Nov 2016, pp. 34–41.

APPENDIX A
KRIPKE CODE SAMPLES

```
kripke:
  for iter = 1 .. niter-1:
    rhsAll()
    sweepAll()
rhsAll:
  for i, j, k:
    rhs(i, j, k)
sweepAll:
  for dir = 0 .. 7:
    sweepDir(dir)
sweepDir(0):
  for i = 0; i < i_max; i++:
    for j = 0; j < j_max; j++:
      for k = 0; k < k_max; k++:
        doWork(i, j, k, 0)
sweepDir(1):
  for i = i_max-1; i >= 0, i--:
    for j = 0; j < j_max; j++:
      for k = 0; k < k_max; k++:
        doWork(i, j, k, 1)
...
```

Fig. 12: Serial, no OpenMP. Each direction of the sweep corresponds to starting at a different corner of the 3-dimensional space.

```
sweepDir(0):
  char dummy[1][1][1];
  step_max = i_max-1 + j_max-1 + k_max-1
  for step = 0 .. step_max:
    for i + j + k = step:
        #pragma omp task \
          depends(in: dummy[i-1, j, k], \
                      dummy[i, j-1, k], \
                      dummy[i, j, k-1]) \
          depends(out: dummy[i, j, k])
        doWork(i, j, k, 0)
```

Fig. 13: Tasking, 8 by 8, skewed. This is the same as Fig. 14, except the directional sweeps are skewed to spawn tasks in an order that minimizes dependences.

```
rhsAll:
  #pragma omp parallel for num_threads(64)
  for i, j, k:
    #pragma omp task
    rhs(i, j, k)
sweepAll:
  #pragma omp parallel for num_threads(8)
  for dir = 0 .. 7:
    #pragma omp parallel num_threads(8)
    sweepDir(dir)
sweepDir(0):
  char dummy[1][1][1];
  for i = 0; i < i_max; i++:
    for j = 0; j < j_max; j++:
      for k = 0; k < k_max; k++:
        #pragma omp task \
          depends(in: dummy[i-1, j, k], \
                      dummy[i, j-1, k], \
                      dummy[i, j, k-1]) \
          depends(out: dummy[i, j, k])
        doWork(i, j, k, 0)
```

Fig. 14: Tasking, 8 by 8. The implementation of OpenMP 4.5 in GCC 7.3 does not actually check array bounds when computing dependences, but rather treats array indexes as logical locations. As a result, it is actually possible to index *outside* array bounds in both directions, which makes dependences much easier to express at the edges. An implementation that does not exploit this behavior performs almost just as well, but the resulting code is significantly less clean, as it takes eight different cases to express the dependences.

```
sweepAll:
  #pragma omp parallel for num_threads(64)
  for dir = 0 .. 7:
    sweepDir(dir) // same as tasking, 8 by 8
```

Fig. 15: Tasking, 64. The call to `sweepDir` is the same as Fig. 14; the only difference is that rather than using a nested parallel region to distribute the workload more evenly, all the work shares the same team of threads.

```
// export OMP_PLACES=sockets
// export OMP_PROC_BIND=spread,master
sweepAll:
  #pragma omp parallel for num_threads(8)
  for dir = 0 .. 7:
    for steps = 0 .. max_steps:
      #pragma omp parallel for num_threads(8)
      for x + y + z = steps:
        ... // tasking, 64, skewed, fused
```

Fig. 16: Do-all. The inner loop body is the same as Fig. 17. The outer loops are permuted to expose 8 by 8 parallelism. This does not sacrifice any parallelism vs. Fig. 17 because the resources are properly partitioned by the nested parallel regions. However, a new parallel region is created per step. Note that the affinity policy binds threads operating on the same direction to the same socket to reduce NUMA overhead.

```
sweepAll:
  step_max = i_max-1 + j_max-1 + k_max-1
  corners = [[0,0,0],
             [0,0,1],
             [0,1,0],
               ...  ]
  directions = [[+1,+1,+1],
                [+1,+1,-1],
                [+1,-1,+1],
                   ...    ]
  #pragma omp parallel num_threads(64)
  #pragma omp single
  {
    // loop over diagonals
    for steps = 0 .. max_steps:
      // loop over all possible steps
      for x + y + z = steps:
        // loop over directions
        for dir = 0 .. 7:
          c = corners[dir]
          d = directions[dir]

          // take steps away from corner
          i = c[0]*i_max + x*d[0]
          j = c[1]*j_max + y*d[1]
          k = c[2]*k_max + z*d[2]

          // is i, j, k a legal index?
          if i < 0 || i >= i_max ...
            continue

          #pragma omp task depends(...)
          doWork(i, j, k, dir)
  }
```

Fig. 17: Tasking, 64, skewed, fused. The 8 sweep directions are combined to enable spawning tasks across directions at the same time, exposing maximal parallelize for the single thread team. This uses the standard idiom of looping first over diagonals of the sweep, however, in practice care must be taken to include proper guards to guarantee that the resulting position is within the original iteration space.

```
sweepAll:
  #pragma omp parallel for num_threads(8)
  for dir = 0 .. 7:
    #pragma omp parallel num_threads(8)
    #pragma omp single
    for steps = 0 .. max_steps:
      #pragma omp taskgroup
      {
        for x + y + z = steps:
          ...  // tasking, 64, skewed, fused
          #pragma omp task
          doWork(i, j, k, dir)
      }
```

Fig. 18: Hybrid, do-all + taskgroup. Compared to Fig. 16, each thread team is created for a given direction only once, and then a master thread spawns explicit tasks, thereby avoiding the overhead of entering a new parallel region for each step. Because tasks are synchronized using the `taskgroup` directive, no dependences are necessary for individual tasks.

APPENDIX B
ARTIFACT DESCRIPTION APPENDIX: ANALYSIS OF EXPLICIT VS. IMPLICIT TASKING IN OPENMP USING KRIPKE

A. Abstract

This artifact contains descriptions of the software versions and hardware specifications used to produce the results, as well as instructions for users seeking to reproduce our experiments.

B. Description

1) Check-list (artifact meta information):

- **Program:** OpenMP 4.5 and C++ code
- **Compilation:** GCC $\geq$6.1 with `-fopenmp` flag
- **Binary:** C++ executable
- **Run-time environment:** Linux environment with OpenMP library
- **Hardware:** Any multi-core shared-memory machine
- **Output:** output from `perf stat`
- **Publicly available?:** No

2) How software can be obtained: GCC is publicly available software. OpenMP 4.5 is implemented in version 6.1 or greater. The latest version of Kripke can be obtained via `git clone` from LLNL's github at https://github.com/LLNL/Kripke. We started with the Charm++ version here: https://computation.llnl.gov/projects/co-design/kripke.

3) Hardware dependencies: Our experiments were run with 64 threads, so we recommend a machine with at least as many (logical) cores.

4) Software dependencies: OpenMP 4.5 requires GCC 6.1 or greater. ICC (Intel), and Clang (LLVM) are other compilers with OpenMP 4.5 support.

5) Datasets: Kripke is hardcoded for the 3D Kobayashi radiation benchmark, problem 3i [12], and will initialize the boundary conditions automatically.

C. Installation

First, install a compiler which supports OpenMP 4.5. We use GCC 7.3, though support began with version 6.1.

Then, install Kripke according to the following instructions: https://github.com/LLNL/Kripke#building-and-running.

D. Experiment workflow

OpenMP provides several environment variables which affect performance. Some can be set using the C++ library, but others must be set directly using environment variables instead, e.g.:

```
$ export OMP_NUM_THREADS=8
$ export OMP_NESTED=1
$ export OMP_PLACES=sockets
$ export OMP_PROC_BIND=spread,master
```

Then, the program must be compiled with the `-fopenmp` flag to enable OpenMP. Finally, use `perf` to time the results. The following command will run the default problem 10 times and report statistics:

```
$ perf stat -r 10 ./kripke ...
```

For the small problem: `./kripke --nest DGZ --niter 10 --grp 4:4 --dir 2:4 --zones 16,16,16 --zset 4:4:4`.

For the large problem: `./kripke --nest DGZ --niter 10 --grp 8:4 --dir 4:4 --zones 32,32,32 --zset 4:4:4`.

E. Evaluation and expected result

`perf stat` will report aggregate statistics for the runs which can be compared against those provided in the paper.

F. Experiment customization

Users can set the input flags to generate differently-shaped problems. The `zset` in particular will change the decomposition of the problem, while the other flags will alter the size of the problem. See the LLNL github for more details on how the other variables affect the problem definition.

Users can also modify the number of threads available to OpenMP via directives or setting environment variables.

G. Notes

Though the base version of Kripke from which we started is publicly available, users will need to implement their own versions of Kripke using OpenMP tasks following the code samples in the Appendix to fully reproduce our experiments.