A Case for Extending Task Dependencies

Tom Scogland(✉) and Bronis de Supinski

Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{scogland1,bronis}@llnl.gov

Abstract. Tasks offer a natural mechanism to express asynchronous operations in OpenMP as well as to express parallel patterns with dynamic sizes and shapes. Since the release of OpenMP 4 task dependencies have made an already flexible tool practical in many more situations. Even so, while tasks can be made asynchronous with respect to the encountering thread, there are no mechanisms to tie an OpenMP task into a truly asynchronous operation outside of OpenMP without blocking an OpenMP thread. Additionally, producer/consumer parallel patterns, or more generally pipeline parallel patterns, suffer from the lack of a convenient and efficient point-to-point synchronization and data passing mechanism. This paper presents a set of extensions, leveraging the task and dependency mechanisms, that can help users and implementers tie tasks into other asynchronous systems and more naturally express pipeline parallelism while decreasing the overhead of passing data between otherwise small tasks by as much as 80 %.

Keywords: Tasks · Producer/consumer · Interoperability

1 Introduction

The addition of tasks to OpenMP marked a fundamental shift in the programming paradigms available to OpenMP users. Programs are no longer tied to statically sized and shaped parallel algorithms, but support recursive or dynamic structures as well. The addition of task dependencies in OpenMP 4.0, and more recently the addition of the `taskloop` construct, have continued to build on this support for dynamic parallelism. Despite these advances, some patterns remain elusive. This paper presents proposed extensions to OpenMP to target two such patterns: efficient interoperability with other asynchronous models, frameworks or hardware; and efficiently expressing fine-grained producer/consumer relationships.

Much as OpenMP has incorporated progressively more dynamic parallelism through the last several versions, many programming models and frameworks have been building in support for asynchronous operations. Some of the more

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-694789).

N. Maruyama et al. (Eds.): IWOMP 2016, LNCS 9903, pp. 130–140, 2016.
DOI: 10.1007/978-3-319-45550-1_10

commonly used of the programming models include CUDA [1] and OpenCL [2], both of which rely on asynchronous data motion and kernel execution for efficiency. At the same time libraries and frameworks such as MPI [7], libuv, or even Linux native asyncio expose programmers to ever more options for asynchronous communication independent of their programming model. All of these can be incorporated into the OpenMP task graph by wrapping a call to them in a task, to be sure, but there is a downside to this approach. The call must block, or be caused to block, within that task for dependencies to resolve correctly in OpenMP. As a result, all of these potentially asynchronous options either must not be integrated into the task graph, or must consume an OpenMP thread for their entire duration, forcing them to effectively be synchronous. We propose an extension to the task and dependency system to support unstructured tasks, tasks encompassing a dynamic region, to incorporate these models and frameworks more efficiently and cleanly with the OpenMP model.

Producer/consumer models have been poster-children for OpenMP tasking, especially since the advent of dependencies. Tasks are, in general, a natural way to express producer/consumer and more general pipeline models, and OpenMP tasks are no exception in this respect. Unfortunately however, OpenMP lacks an efficient point-to-point data passing mechanism to support very fine-grained producer/consumer parallelism. The built-in way to pass an element from a producer to a consumer is for the producer to create a task, which is a relatively expensive operation that must be amortized. We propose adding a new type of dependency, queue dependencies, that carry typed concurrent queues to reduce the necessity to create additional tasks while providing an efficient point-to-point data transfer mechanism for OpenMP.

Our contributions are as follows:

- A design for unstructured tasks, allowing general interoperability between the OpenMP task graph and external asynchronous models
- An extension to task dependencies to carry data channels, enhancing support for fine-grained producer/consumer codes
- An evaluation of our queue-based producer/consumer dependencies.

The rest of this paper is laid out as follows. First we present the base global dependency extension on which the others are built in Sect. 2. Our design for task interoperability with unstructured task regions follows in Sect. 3. Next we discuss and evaluate our design for queue dependencies in Sect. 4, followed by related work in Sect. 5.

2 Global Dependencies

Our extensions revolve around improving point to point synchronization between tasks and integration of external synchronization mechanisms into the OpenMP task graph. Both of these goals benefit from one base extension, the ability to define a global, or at least cross-task, dependency or synchronization directed

synchronization mechanism. Depending on the shape this takes, different extensions become possible as either API routines or directives. This section discusses the global dependency mechanism itself, as well as two extensions that either make use of it or exploit the added potential for point-to-point synchronization in a system that incorporates the concept. Specifically we propose extensions for writing advanced interoperability interfaces that can help integrate other asynchronous runtimes or mechanisms with the OpenMP task graph and integrating producer consumer relationships either through dependency directives, or through an analogous queue API.

For the purpose of this paper, we assume the existence of an extra attribute, `global`, on the `depend` clause that can be specified before the dependency direction. For example, `depend(global, inout: a)` would introduce an input dependency on all tasks, regardless of their parent task or sibling relationship, that have an output dependency on `a`. Likewise it specifies an output dependency that applies to all tasks created by all threads that specify an input dependency on `a` thereafter. This kind of dependency can be thought of much like an `omp single` directive for structured parallel regions, in that it ensures that only that one task, of all tasks that have dependencies on a given list item, can be running at a time. One example use would be to block work on a given data element, or set of same, to synchronize or otherwise communicate with other processes in a distributed memory setting. The nature of a global dependence is that its ordering with respect to specific tasks encountered in other threads is not guaranteed without external synchronization, but it can provide a useful serialization point.

3 Unstructured Tasks

Especially with the uptake of heterogeneous architectures and computational coprocessors, asynchronous operations are becoming more and more common in high performance computing. OpenMP provides an interface to asynchronously offloading work with the target directives, and memory motion if necessary, with the device data constructs. If there is an operation that isn't directly provided by OpenMP however, integrating it without wasting a thread is less than straightforward for even the most advanced users.

As an example, a user writing an asynchronous version of `omp_target_memcpy` with an underlying asynchronous call provided by their system might write the code in Fig. 1. The resulting implementation is asynchronous with respect to the encountering task and the encountering thread. Even so, if they take the simple option, it requires a CPU thread to remain blocked in the `async_wait()` call in order to ensure the memcpy is done before the task dependence is satisfied. To avoid blocking the thread the entire time, the user may add polling using `taskyield` to re-invoke the task scheduler. While this certainly works, it introduces additional invocations of the task scheduler and of a polling interface to function, where an asynchronous response mechanism may already be available and may further be more efficient or may be the only mechanism available. To address cases like this, where an already asynchronous event should be integrated into the task dependency graph, we propose to introduce *unstructured tasks*.

```
void naive_async_memcpy(void *dst, void *src,
                        size_t size, void **dep) {
    // Wait for all OpenMP tasks that have out depends on *dep
    #pragma omp task depend(inout: *dep)
    {
        // start asynchronous memcpy
        async_memcpy(dst, src, size);
        // wait for the memcpy to complete
        async_wait();
        // OR
        while(!async_poll_done()) {
            #pragma omp taskyield
        }
    }
}
```

Fig. 1. A naive attempt at an asynchronous `memcpy`

Figure 2 shows the constructs of the extension we propose for this use case. Much like the device data constructs, specifically `target enter data` and `target exit data`, the `taskenter` and `taskexit` directives form an unstructured region. Rather than an unstructured data environment however, these form an unstructured task region to allow manipulation of dependency satisfaction based on external factors. The `taskenter` directive takes an identifier clause, which is used to fill a passed variable with the identifier later used to satisfy the task dependency, and optionally accepts an associated block and a `depend` clause. A `taskenter` without a depend clause is allowed, but is treated as though it begins with a `taskwait` construct. If no block is provided, the encountering thread blocks on a task scheduling point until all input dependencies are satisfied. Otherwise, if a block is provided, it is treated as a deferrable task that will be executed when dependencies are resolved, allowing the encountering thread to continue immediately.

The main difference between `taskenter` and a regular OpenMP task is that output dependencies *are not satisfied* at the end of the `taskenter` construct. In order to satisfy the output dependencies on an unstructured task, a `taskexit` directive with a matching `identifier` clause must be used. If the dependency is a standard, local, dependency then the `taskexit` must be encountered by the task that encountered the `taskenter` construct. For a global dependency, any thread executing on the same device is allowed to encounter the `taskexit`.

The example in Fig. 3 illustrates the usage of an unstructured task to improve integration with a native asynchronous memcpy. This version uses a `taskenter` directive to create a global input and output dependency on the value passed in by the user, and since it has an associated block and no `if` clause, the task is deferred. While the encountering thread continues, the task waits for input dependencies to be satisfied. Once they are, the external asynchronous memcpy is invoked, and a callback registered with the external library to satisfy the

```
// Begin an unstructured task region:
#pragma omp taskenter identifier(list-item) \
                      [depend([global,] \
                              [in|out|inout[:]] \
                              <list-items>) ...] \
                      [untied] \
                      [if(<condition>)]
// {} Optional block to be run as a task when in dependencies
//    are satisfied

// End an unstructured task region, satisfy out dependencies
#pragma omp taskexit identifier(list-item)
```

Fig. 2. Unstructured tasks

```
struct dep_ident {void ** dep; void * ident;};
void extended_async_memcpy(void *dst, void *src,
                           size_t size, void **dep) {
    struct dep_ident *i = malloc(sizeof(struct dep_ident));
    i->dep = dep;
    // Wait for all OpenMP tasks that have out depends on *dep
    #pragma omp taskenter identifier(i->ident) \
                          untied \
                          depend(global, inout: *dep)
    {
        async_memcpy(dst, src, size);
        async_callback_on_complete(clear_dependency, i);
    }// Note: *dep out is not satisfied here
}

void clear_memcpy_dependency (void *dep) {
    struct dep_ident *i = dep;
    // Satisfy the currently outstanding out dependency on *dep
    // note: since this is a global dependency, any thread may clear the
    //       dependency
    #pragma omp taskexit identifier(i->ident)
}
```

Fig. 3. Produce and consume as dependence types

OpenMP dependence on `*dep` once the copy is complete. Using the extension, the external memcpy now offers nearly the same level of task integration that unstructured device data constructs do through the `nowait` clause, but through a wholly user-controllable mechanism.

It is worth noting that this interface would be meant for advanced users only, primarily for those writing runtime systems, optimized native libraries supporting OpenMP or integrations with other systems. In giving the user control over an unstructured dependency region, it is possible for a user to create a deadlock by

never completing an unstructured task in the same way it is possible for them to cause a deadlock by never unlocking a lock that threads are waiting on. That said, the potential advantages of being able to integrate more tightly with the dependency mechanisms of system libraries, or even communication libraries such as MPI, are substantial.

4 Queue Dependencies

OpenMP tasks are frequently taught with certain specific dynamic workloads in mind. The most common of these appear to be parallelizing recursive algorithms and parallel processing of dynamically sized containers like linked lists. Right behind these in lists of use-cases however is to model a parallel producer and consumer. While this use-case appears in many lists, actual code for a task-based producer consumer is rarely included in examples, preferring to fall back on `critical` sections to create ad-hoc queues to implement the pattern. The resulting code requires external dependencies, or manual implementation of data transfers, neither of which is necessary for a version using tasks. As an example, see Figs. 4 and 5. Both figures implement the same pattern, but the critical section version requires a queue to store intermediate work, and takes nearly twice as many lines of code as the task version. Admittedly, the critical section version offers a known first-in first-out order, which the task version does not, but not all producer consume problems require such an ordering.

```
1  serial_queue_t sq = INIT_SERIAL_QUEUE;
2  int done = 0;
3  #pragma omp parallel
4  while (!done) {
5      int product = 0;
6      if (omp_get_thread_num () == 0) {
7          done = produce(&product);
8          if (done) {
9              #pragma omp flush
10             break;
11         }
12         #pragma omp critical(product_queue)
13         enqueue(&sq, product);
14     }
15     #pragma omp critical(product_queue)
16     int status = dequeue(&sq, &product);
17     if (status == QUEUE_EMPTY) continue;
18     consume(product);
19 }
```

Fig. 4. A simple single producer multiple consumer with critical sections

```
#pragma omp parallel
#pragma omp serial
while (1) {
    int product = 0;
    int done = produce (&product);
    if (done) break;
    #pragma omp task firstprivate(product)
    consume (product);
}
```

Fig. 5. A simple single producer multiple consumer with tasks

While there are clear upsides to the task version, unfortunately there is also a hidden downside. The task version incurs more overhead, for which we will provide specific numbers below. What both of these have in common is that they share the work of consumption across available threads by passing produced data through an intermediate data structure. In Fig. 4 the structure is an is explicit queue. The task version in Fig. 5 uses an implicit structure, the task queue inside the OpenMP runtime itself, to accomplish the same ends. Each task storing its required data element to be consumed when a thread is available. In order to improve the usability of this idiom in OpenMP, we propose to incorporate a new dependency type that also serves to pass data between tasks, allowing them to be reused rather than rebuilt.

The core of the extension is to add the new dependency types `produce` and `consume`, along with corresponding directives `omp produce` and `omp consume`. When a `produce` or `consume` dependency is encountered, a multi-producer multi-consumer queue is logically created and associated with the address of the list-item passed to the `depend` clause. If one already exists in the current thread, it is found and reused, or if the dependency uses the `global` attribute it would be found in any thread in the contention group. The queue thus created would persist until either all producer tasks using it have completed and it is empty or until the parallel region ends, whichever comes first. Within the region of a producer task the `omp produce(<arg>[[, <full>], <complete>])` directive may be used to add items to the queue. The first argument to the directive is copied into the queue as they would be by a `firstprivate` clause. If the queue is full and the `<full>` variable is omitted, the operation blocks on a task scheduling point until room is available. If the `<full>` argument is specified, then the operation is non-blocking, and sets that argument a boolean representing whether the `<arg>` was successfully placed into the queue or not. If the `<full>` parameter is true, the application is responsible for caring for the argument value until it can be passed. Finally, when all producer tasks for a given queue are complete, the queue becomes closed, allowing consumers to detect that there will be no more values to consume by specifying something to be set for the closed argument.

```
int done = 0;
#pragma omp parallel
while (!done) {
    int product = 0, more = 0;
    #pragma omp master
    #pragma omp task depend(global, produce: product)
    while (1) {
        done = produce (&product);
        if (done) {
            #pragma omp flush
            break;
        }
        more = 1;
        #pragma omp produce(product)
    }
    #pragma omp task depend(global, consume: product)
    while(1) {
        #pragma omp consume(product, more)
        if (!more)
            break;
        consume (product);
    }
}
```

Fig. 6. A simple single producer multiple consumer with queue dependencies

On the opposite side, a task with a `consume` dependence becomes a consumer task, and uses the same mechanism to locate the queue it is to read from. Within the region, the `omp consume(<arg>[[, <more>], <complete>])` directive can be used to retrieve a value from the queue into `<arg>`, a boolean representing whether the value retrieved is new or the queue was empty in `<more>` and `<closed>` which denotes whether the queue has been closed. If a consumer is encountered without a matching producer having been encountered, the consumer will act as though the producer exists and is both empty and closed.

Using these constructs, we can produce the example presented in Fig. 6. This example uses producer and consumer tasks to create a pipeline that can continue as long as the queue has room in a parallel context, and yet retains a correct serial elision when OpenMP is not used. This example shows the master encountering a producing task while all the other threads, and subsequently the master itself, encounter consuming tasks. All of these become logically connected by the same multi-producer multi-consumer queue because the dependency specification on each is global. The producer blocks when the queue is full, but due to the task scheduling point therein it can be re-scheduled to continue in the original loop and help process consuming tasks until the queue runs dry. Threads may spin through the outer loop, but because the lifetime of the queue is tied to the enclosing parallel region, it is only created and destroyed once in this construct.

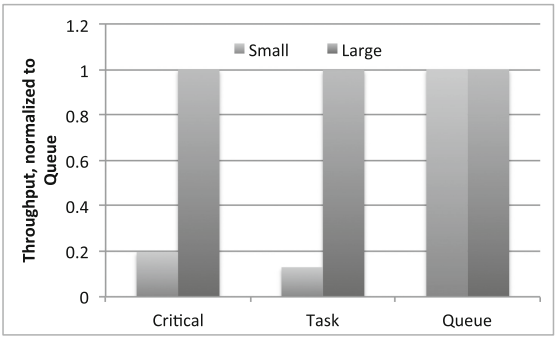

Fig. 7. Producer/consumer microbenchmark normalized element throughput

The main downside to this construction is the relative verbosity and complexity, it is in fact longer than using a third-party queue with critical sections, but it holds an advantage in potential throughput. The main complication is maintaining serial elision correctness while using directives for the data passing and synchronization mechanism. This could likely be alleviated by shifting to an API-based approach somewhat, or certainly by providing a progress guarantee between tasks even when run with one thread, but those options are outside the scope of this paper.

Figure 7 shows results for a simple producer/consumer microbenchmark implemented with each of these three mechanisms. Each run uses a single producer, the master thread, and eight consumers (one of which is also the producer), one per thread on the 8-core Intel i7 CPU, 500,000 elements are produced, retrieved and consumption is a simple sum for the small case or a loop over 5,000 elements in the large case. The underlying queue implementation for the concurrent queue is a blocking multi-producer multi-consumer queue equivalent to that proposed by Scogland et al. [10]. These results show that for producer/consumer problems with relatively small work, or a requirement for high throughput, the queue extensions can be highly beneficial. For medium or larger, long running, tasks the overhead is dwarfed by the task execution time.

5 Related Work

Integrating task systems with one another has been a popular topic of research in recent years. OmpSs [4,6], a precursor to many of OpenMP's tasking features, incorporates both CUDA and OpenCL asynchronous communications into its runtime, and provides the results to the user at the beginning of their tasks. The runtime backing the StarPU [3] task system performs similar background management of data as well. This approach differs from the one we propose for OpenMP in that we want to give users control over the machinery that allows OpenMP to coexist with another asynchronous model peacefully. The automated asynchronous data movement these models use is a good candidate

for optimizations of the device data constructs, and possibly other areas, but does not address quite the same problem.

The concept of using channels, or queues, between independent tasks or threads of control to provide synchronization and communication is certainly not a new one. Discussions of a method like it go at least as far back as discussions of communicating or cooperating sequential processes and coroutines by Dijkstra [5] and Hoare [8] who discuss communication between coroutines via a variety of mechanisms as fundamental to programming. In more recent times the use of blocking channels in combination with lightweight tasks has been popularized by the Go [9] language, which incorporates both a coroutine analog and synchronous typed channels as basic language types. The main distinctions between our extension and channels as used by Go are that Go's channels support serial execution only with guaranteed progress despite blocking on channels, where our extension is designed to always result in a valid serial elision that does not require task swapping to be correct. A progress guarantee of this type might be a useful future extension, but it is beyond the scope of this paper.

6 Conclusions

We have presented three extensions to OpenMP's tasking model. First, at the base, is allowing a `global` modifier to make a task dependency apply across a contention group, rather than just to sibling tasks. Leveraging that, we present unstructured tasks, which give users and runtime implementers a way to more closely and efficiently integrate their asynchronous mechanisms with OpenMP's task graph. Finally we presented an extension for producer/consumer dependencies, allowing OpenMP runtimes to provide a queue-like data-passing and synchronization abstraction for producer/consumer models. We show that using the queue mechanism allows a user to generate far fewer tasks, and pay less overhead per iteration of each of the producer and consumer. For very small consumer workloads, we found an improvement of as much as 80 % and no performance decrease for larger tasks. In the future we would like to investigate these mechanisms in terms of larger applications, and explore the possibility of gracefully handling an unstructured task left unsatisfied as well as investigating the possibility of a formal progress model for OpenMP tasking.

References

1. CUDA programming guide (2007). http://docs.nvidia.com/cuda/cuda-c-programming-guide/
2. The OpenCL Specification, November 2012. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009). http://www.springerlink.com/index/h013578235633mw3.pdf

4. Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguadé, E., Labarta, J.: Productive programming of GPU clusters with OmpSs. In: International Parallel and Distributed Processing Symposium, pp. 557–568 (2012). http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6267858&contentType=Conference+Publications&matchBoolean%3Dtrue%26rowsPerPage%3D30%26searchField%3DSearch_All%26queryText%3D%28%22Productive+Programming+of+GPU+Clusters+with+OmpSs%22%29
5. Dijkstra, E.W.: Cooperating sequential processes. In: Hansen, P.B. (ed.) The Origin of Concurrent Programming, pp. 65–138. Springer, New York (1968)
6. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(2), 173–193 (2011). http://www.worldscinet.com/abstract?id=pii:S0129626411000151
7. Forum, M.P.I.: MPI: a message-passing interface standard. Technical report (1994). http://citeseer.ist.psu.edu/article/forum94mpi.html
8. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). http://portal.acm.org/citation.cfm?doid=359576.359585
9. Pike, R.: The go programming language. Talk given at Google's Tech Talks (2009)
10. Scogland, T.R.W., Feng, W.: Design and evaluation of scalable concurrent queues for many-core architectures. In: ACM/SPEC International Conference on Performance Engineering (ICPE), February 2015