Mapping OpenMP to a Distributed Tasking Runtime

Jeremy Kemp[1(✉)] and Barbara Chapman[2]

[1] Department of Computer Science, University of Houston, Houston, TX, USA
jakemp@uh.edu
[2] Department of Applied Mathematics and Statistics and Computer Science, Stony Brook University, Stony Brook, NY, USA
Barbara.Chapman@stonybrook.edu

Abstract. Tasking was introduced in OpenMP 3.0 and every major release since has added features for tasks. However, OpenMP tasks coexist with other forms of parallelism which have influenced the design of their features. HPX is one of a new generation of task-based frameworks with the goal of extreme scalability. It is designed from the ground up to provide a highly asynchronous task-based interface for shared memory that also extends to distributed memory. This work introduces a new OpenMP runtime called OMPX, which provides a means to run OpenMP applications that do not use its accelerator features on top of HPX in shared memory. We describe the OpenMP and HPX execution models, and use microbenchmarks and application kernels to evaluate OMPX and compare their performance.

1 Introduction

OpenMP [6] is a directive-based parallel programming interface that provides a convenient means to adapt Fortran, C and C++ applications for execution on shared memory parallel architectures. In response to the growing complexity of shared memory systems, OpenMP has evolved significantly in recent years. Today, it is suitable for programming multicore or manycore platforms, including any attached accelerator devices.

Multiple research efforts have explored the provision of an application level interface based on the specification of tasks or codelets (e.g. PARSEC [9], HPX [12], OCR [14], OmpSs [10]) and their dependencies. This is due to considerable interest in the potential of dataflow programming approaches to provide very high performance via the minimization of synchronization.

Tasking interfaces have also been proposed as a low-level execution layer for very large computing systems, including the anticipated exascale platforms. Given this interest, we have explored the mapping of OpenMP to one such interface, HPX [12]. HPX is a C++ library with an extensive set of features that support task-parallel programming and that made it a good candidate for this work. Our translation of OpenMP (with the exception of accelerator features) to

B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 222–235, 2018.
https://doi.org/10.1007/978-3-319-98521-3_15

HPX, called OMPX, began as a modification of the Intel OpenMP Runtime [1]. We ended up rewriting major portions of it, as the use of OpenMP threading and scheduling mechanisms prevented us from benefiting from key HPX features.

The rest of the paper is organized as follows. We briefly introduce OpenMP and HPX in Sect. 2, and describe their runtimes in Sect. 3. Section 4 provides a description of the implementation itself, and Sect. 5 gives the results of our evaluation. We then outline related work and reach some conclusions.

2 Overview of OpenMP and HPX

OpenMP defines a set of directives for the specification of parallelism in C, C++ and Fortran applications with minimal code change. Code within OpenMP parallel regions are executed by a team of threads, each of which may access shared data and may also have some private data that is not accessible to other threads. OpenMP has features for specifying parallel loops and sections, which will be executed by the threads participating in the enclosing parallel region, constructs for offloading code and data to GPUs, and a means to set/get execution parameters such as a thread's ID or the number of threads in a team.

OpenMP 3.0 introduced task parallelism and redefined itself in terms of tasks. Explicit tasks are created with the task directive; implicit tasks are created to implement other constructs. Code associated with an explicit task construct can be executed asynchronously on any thread in the parallel region at any time prior to their next synchronization, which may be a taskwait construct, that waits on all tasks created by the current task, or a barrier that waits for all tasks created in the parallel region to finish.

Tasks may be suspended by the implementation at certain points during their execution. By default OpenMP tasks are tied, which prevents a task from moving to a new thread when its execution is resumed, and which implies that it may consistently access a specific thread's private data. Implicit tasks are tied. Untied tasks may be suspended and subsequently continued by another thread.

OpenMP 4.0 introduced the taskgroup synchronization construct and the depend clause for the task directive. The taskgroup construct waits on all tasks created in a region, and not only on those created by the current task. The depend clause is used to specify data dependencies between tasks. It takes 2 parameters: the type of dependency (in, out or inout) and a list of variables. OpenMP uses the address of a specified variable to match it to other tasks so that it can execute them in the order in which they were created. These dependencies are restricted to tasks created by the same parent task.

HPX comprises both high-level features for creating parallel C++ applications consisting of a collection of tasks, as well as low-level features that support their efficient execution, e.g. enabling the creation of a custom scheduler. It provides a uniform API for both shared and distributed memory systems. HPX includes the means to create and schedule tasks, and specify task dependences, without any notion of user-level threading. Under active development, it encompasses features for convenience such as a parallel loop (implemented similarly to OpenMP's taskloop), and utilities such as parallel sort and search routines.

HPX tasks are created using `dataflow` and `async` (see Fig. 1), extensions of C++ async and future constructs. Async tasks use available data and are able to be executed immediately. The dataflow keyword is used when the input for the task is not yet ready, but futures corresponding to that data are. They can be used to create directed acyclic graphs (DAGs) of data-dependent tasks without change to the internals of the functions involved.

```
int val = func(42);
future<int> f_val1 = async(func, val);
future<int> f_val2 = dataflow(unwrapping(func), f_val1);
f_val2.wait();
```

Fig. 1. An example comparing the use of dataflow and async to a normal function call.

Futures are a key element of HPX. They coordinate the flow of data between tasks, as well as the order of execution of tasks. Ideally, futures are used as input and output for dependent tasks, thereby avoiding explicit synchronization inside the tasks. Wait and get methods are available for situations where this is not possible: with them, a task can wait for the data in a future, or retrieve it, returning control back to the runtime to schedule other work until the data is available. On distributed memory systems, dependencies may occur between tasks running on different nodes. To handle this, the user creates a special kind of object. When methods are called on that object, the implementation will insert the necessary communication. Note that in other contexts, this might be resolved via polling or blocking communications, but in HPX the task involved will relinquish resources until the data is available. HPX also provides a full set of legacy synchronization mechanisms, including mutex, lock and barrier features that can be used inside tasks. Unlike their OpenMP counterparts, the HPX variants exist entirely in user space and do not block a thread. Instead they return control to the runtime so other work can be done.

Advanced features in HPX include executors. These are containers that tasks can be created in; synchronization on all tasks in a container is similar to OpenMP's taskgroup. Executors can also be used for a high-level specification of how tasks are scheduled, how the runtime task queues are structured and how tasks may be stolen from them.

Finally, HPX also provides low-level APIs for direct interaction with the threading subsystem. This API is very verbose, and is not generally intended for application development. It can be used to place tasks precisely on threads, set priorities and queues, and influence work stealing mechanisms.

3 Runtime Implementation Overview

3.1 OpenMP Runtime Task Management

This section covers the runtime level details of OpenMP that are pertinent to understanding OMPX. We focus on how a task is handled by the Intel OpenMP runtime, starting from its creation.

In the Intel OpenMP Runtime, both tied and untied tasks are stored in a single local queue. Each thread accesses its own queue from one end, and steals from the other end of other thread's queues. Whenever a task is stolen, that task must be checked to see if it is ready to execute, and if it there are constraints that might not allow it to be executed on the thread that stole it.

Untied tasks can be executed on any thread in a parallel region without restriction. In contrast, the runtime must ensure that a new tied task is scheduled according to certain scheduling constraints [4] to prevent deadlock. The constraints on execution order arising from OpenMP task dependencies are only possible among sibling tasks. The input and output variables used to specify them are tracked by the runtime in a hash table. No synchronization is needed to access the hash table, since only one task will ever write to an entry.

3.2 The HPX Runtime

Unlike OpenMP, HPX does not implement the fork-join execution model. A worker thread is spawned for each OS thread, but these do not begin executing application code until a task is scheduled on them. The necessary functionality for creating a task in a single function call is already present in C++ constructors. The arguments passed to it are copied or moved as specified by the constructors of the objects being passed in. When a task is initially created, memory is allocated for a small task data object, similar to OpenMP.

The default scheduler in HPX places tasks in lockless lifo queues. Each worker thread has a queue for each of three priorities: high, normal and low. There are no constraints on how tasks can be scheduled or stolen once they are ready for execution, but there are several modular schedulers included with HPX which can change the queue organization and how work is stolen.

HPX uses dynamically allocated stacks for its tasks, unlike OpenMP, which will continue to use the stack of the original thread. The allocation of the stack for a task can be delayed until immediately before execution. The implementation can potentially recycle a previous stack frame, if the task that used it is complete. Tasks that suspend on a wait, get or yield still need their stack frames. Thus creating a large number of tasks that suspend can consume large amounts of memory and hurt performance.

A future is used to coordinate shared state between two or more dependent tasks. If a task hasn't completed when a second, dependent task is being created, the latter will append the remainder of its task creation to the end of the first task. Once complete, the first task will resume creation of the second task. If it does not depend on any other inputs, then the second task will begin executing

immediately. If the second task has other outstanding dependencies, the first task will pass the remainder of the second task's creation to the next task it depends on, and another task will be pulled from a work queue and executed.

4 The OMPX Implementation

HPX uses tasks as the primary form of parallelism, and task dependencies as the primary form of synchronization. Since OpenMP has tasks underlying all of its parallelism, it can potentially be mapped to a purely task-based programming model. We describe our OMPX runtime, an adapted version of the Intel OpenMP runtime, and explain how it translates OpenMP features into HPX code. Since certain features (primarily, tied tasks) do not have a straightforward mapping, we created two versions of OMPX in order to assess the cost of providing full compliance with the OpenMP standard.

4.1 Initialization and Parallel Regions

Like most OpenMP runtimes, OMPX is not loaded until the first time an OpenMP construct or library call is encountered. As part of its initialization, OMPX reads and processes environment variables for both OpenMP and HPX. The execution of a synchronization function passed to `hpx::start()` signals that the HPX runtime has started. A function is registered with `atexit()` that will shut down the HPX runtime when the process exits. If HPX has already been started without initializing OMPX, then the application is a hybrid OpenMP HPX application, in which case the number of threads is queried from HPX and used for the OMPX runtime.

```
void thread_setup( microtask_t t_func, arg_struct arg, int tid)
{
    omp_task_data task_data(tid, arg.parent);
    set_thread_data( &task_data );
    kmp_invoke( t_func, arg );
    while (task_data.num_tasks > 0) {
        hpx::this_thread::yield();
    }
}
void fork(microtask_t t_func, void *args)
{
    vector<future<void>> threads;
    for(int i = 0; i < num_threads; i++) {
       threads.push_back( async( thread_setup, t_func, args, i, ...));
    }
    hpx::wait_all(threads);
}
```

Fig. 2. The implementation of fork-join in OMPTX.

Parallel regions are translated into runtime calls to a fork function, where one of the arguments passed is a compiler-generated function containing the code

inside the parallel region. The initial implementation of this in HPX is shown in the fork function call in Fig. 2. The `thread_setup` function initializes OpenMP metadata for that implicit task (e.g. num_threads). This data is stored local to the HPX task, which is called a thread in HPX nomenclature (but called a task elsewhere in this paper to avoid confusion), using the HPX function `set_thread_data`. This data can later be retrieved with the `get_thread_data` call. The `thread_setup` function continues to stay in scope as long as there are explicit tasks that have not completed. This was subsequently replaced by an implementation that makes calls to HPX's lower level threading interface to place tasks on specific threads, and synchronize them with lower level synchronization. This translation places implicit tasks on a specific thread but does not prevent them from being stolen, which can cause inconsistencies with any thread-specific constructs, e.g. accesses to threadprivate data or `omp_get_thread_num()`. An HPX construct called an executor can provide the requisite functionality. We modified the default scheduler to remove work stealing and created an executor with a suitable work stealing scheduler to handle explicit tasks. The use of HPX executors may lower performance and thus we created two versions of OMPX: a compliant version that binds tied tasks to a thread with executors and a non-compliant version that does not bind tied tasks and uses atomic counters instead. The version can be selected when compiling the runtime.

4.2 Worksharing Constructs and Synchronization

Relatively little effort is needed to implement worksharing constructs in OMPX, since the same logic as in standard OpenMP can be used, and the metadata needed for computing chunks of work and handling constructs like ordered is passed to the corresponding tasks. Atomic counters and locks local to each parallel region support the implementation of single, master and critical constructs in a manner that is very similar to a standard OpenMP implementation. Since OpenMP barriers wait on all tasks created in the parallel region before returning, we must keep track of their completion. In OMPX, task completions are tracked by an atomic counter in the non-compliant version, or the executor in the compliant version. Once all tasks are complete, each implicit thread waits on the HPX barrier local to the parallel region.

4.3 Tasking

Outside of scheduling, the biggest challenge in implementing explicit tasks is the parent-child relationship which OpenMP tasks have, and HPX tasks do not. Each task requires a small struct to hold the data to implement this behavior. Care must be taken when creating tasks to avoid referencing the metadata of the parent task, as the parent may be finished and deallocated by the time the child task begins executing. To accomplish this, the needed data is copied into the child task, including a shared pointer to an atomic variable that tracks the number of child tasks. This atomic is used to implement the wait in the `omp_taskwait` runtime call.

```
void task_setup(kmp_task_t *task, omp_task_data *parent )
{
    auto task_func = task->routine;
    omp_task_data task_data(parent->team, parent->icv);
    hpx::set_thread_data( &task_data );
    task_func(task);
    parent->num_child_tasks--;
    team->num_tasks--;
    delete[] (char*)task;
}
int omp_task(kmp_task_t * task_struct)
{
    omp_task_data *task = get_task_data();
    task->num_child_tasks++;
    async(task_setup, task_struct, task);
    return 1;
}
```

Fig. 3. The implementation of task creation in OMPTX.

Task creation is implemented by two calls to the runtime: `task_alloc`, which allocates memory and `omp_task`, which creates the task and passes control of it to the runtime. The compiler computes the amount of memory needed for shared and firstprivate data plus a small struct used by the runtime. The size is passed to `task_alloc` which allocates the memory and returns a pointer to it. This memory is freed once the task is completed at the end of `task_setup` shown in Fig. 3. The `omp_task` call is implemented with async, similar to implicit task creation. It returns immediately, and allows the HPX runtime to manage the scheduling of the task. Similar to the `thread_setup` function for spawning implicit tasks, the `task_setup` function initializes the metadata for the task and decrements two counters after the task function call returns. The first of the counters decremented in `task_setup` records the number of active, or not completed, tasks in a parallel region. This information is tracked by the executor in the compliant version. The second counter maintains the number of active child tasks spawned by a given task. These are necessary for the correct implementation of barrier and taskwait respectively.

Tasks with dependencies are translated to an Intel OpenMP runtime call, `omp_task_with_deps`, a function similar to the `omp_task` call, but with additional parameters for dependencies. These parameters include the number of dependencies and an array of structs populated with the address of the variable used and a flag indicating the type of dependence.

Since HPX uses futures to coordinate task dependencies and OpenMP uses the address of the variables in the depend clause, a map is needed to match these addresses with the corresponding future. This future is the one returned from the last task that output a dependency to the given address. To do this, each task has its own `std::map<int64_t, shared_future<void>>` to map variable

```
1  vector< shared_future< void > > dep_futures;
2  for(int i = 0; i < ndeps;i++) {
3      auto dep_addr = deplist[i].base_addr;
4      if(df_map.count(dep_addr) > 0)
5          dep_futures.push_back(df_map[dep_addr]);
6  }
7  shared_future new_task;
8  if(dep_futures.size() == 0 ) {
9      new_task = async(task, args);
10 } else {
11     auto deps = when_all(dep_futures);
12     new_task = dataflow(df_setup, args, deps);
13 }
14 for(int i = 0; i < ndeps;i++) {
15     int64_t dep_addr = deplist[i].base_addr;
16     if(df_map[i].flags.out)
17        df_map[dep_addr] = new_task;
18 }
```

Fig. 4. Adding the newly created task to the map for later usage.

addresses to futures. Since each task has its own map, no synchronization is needed to access it.

The `omp_task_with_deps` function was re-implemented in three stages: building the dependency vector, spawning the task, and updating the dependency map. In the first stage, shown on line 1 of Fig. 4, the addresses of dependencies are translated to futures that can be used as arguments when creating tasks with `dataflow`. This is done by traversing each dependency in the list, looking their addresses up in the map, and, if that entry in the map holds a future, appending it to the dependency vector. The first tasks created this way will have no input dependencies in the vector, as the map starts off empty. Once the dependency vector is built, the task can be created, as shown on line 7 through 13 of Fig. 4. If the dependency vector is empty, then the task is spawned using async. Otherwise, the task is created using `dataflow`, with the dependency vector as input. No data is passed through the futures in this dependency vector, as OpenMP tasks don't have a return value. The futures only serve to signal that a task is ready to begin. Finally, as shown in the loop beginning on line 14 of Fig. 4, the future returned from the `async` or `dataflow` is inserted into the map for each output dependency, to be used by later tasks as input dependencies.

5 Evaluation

In this section we compare and contrast overheads of OpenMP, HPX, and our two implementations of OpenMP on top of HPX - which we call the "compliant" and the "non-compliant" versions - and also show their performance on several application kernels. These benchmarks were run on a single Haswell node of the

NERSC Cori system. The Cori nodes contain 2 Haswell CPUs for a total of 32 cores and 64 threads. The OpenMP benchmarks were compiled with icc 18.0, while HPX benchmarks were compiled with gcc 7.1. The performance of HPX is best with gcc, and new language features used by HPX do not always work with icc. So, for the kernel applications, we show speedup relative to the serial version compiled with the corresponding compiler, gcc for HPX and icc for all others.

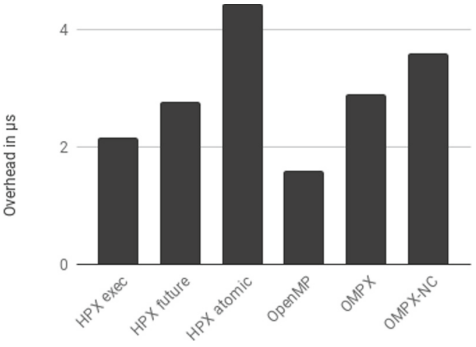

Fig. 5. Task creation overhead per task

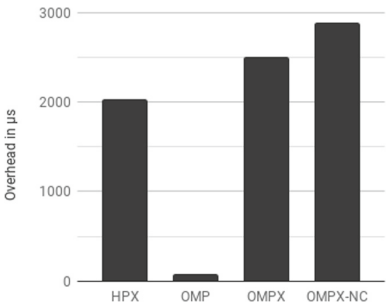

Fig. 6. EPCC barrier benchmark

The microbenchmarks consist of a task creation benchmark, and the EPCC taskbench benchmark suite [7]. The task creation benchmark is a variation of the task creation benchmark included with HPX. An OpenMP version of this benchmark was written for comparison, as well as 2 new HPX versions that use the same synchronization as the OMPX runtimes. HPX future represents typical application level HPX, HPX exec and OMPX use executors, and HPX atomic and OMPX-NC both use atomic counters to synchronize tasks. The overhead for this task creation benchmark can be seen in Fig. 5.

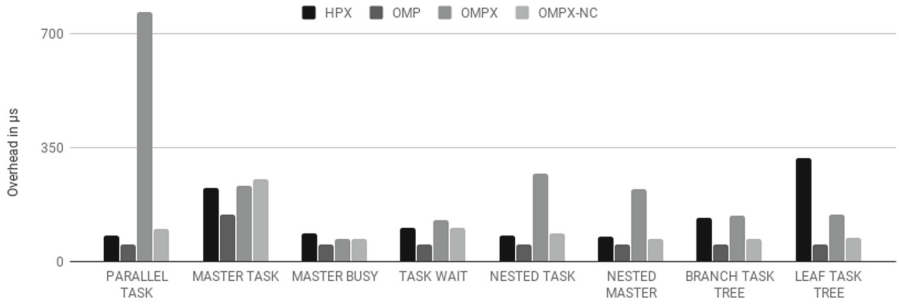

Fig. 7. EPCC Taskbench overhead

EPCC taskbench is an OpenMP based microbenchmark suite that measures overheads of different methods of task creation and synchronization. An HPX version of these benchmarks was written for comparison. This benchmark suite is comprised of 9 benchmarks, 8 of which are shown in Fig. 7, and the barrier benchmark in Fig. 6. The performance gap between runtimes with the barrier benchmark is so large that it needed to be included as a second figure. To summarize these benchmarks: The barrier benchmark creates tasks on each thread with a barrier after each task is created. Parallel task creation creates tasks on each thread. The master task benchmark creates tasks on the master thread. The master busy benchmark creates tasks on master, while all other threads execute a large serial workload. The task wait benchmark creates tasks on each thread, with a taskwait after every task is created. The nested task benchmark creates tasks on each thread, which create nested tasks. The nested master benchmark creates nested tasks on the master thread. The tree based benchmarks create a tree of tasks recursively, with the branch version executing work on the branches, and the leaf version on the leaves.

There are several noteworthy observations in these microbenchmarks. When comparing the parallel task and master task benchmarks, we can see that the OMPX version and the underlying executor have a substantial increase in run time for concurrent task creation. The nested task and nested master benchmarks also have concurrent task creation, but to a lesser extent than the parallel task benchmark. This also corresponds to increased task creation time in the executor based runtime. In the tree based benchmarks, we see the pure HPX versions take longer, as the tree structure does not map well to futures when there are no data dependencies involved. Overall, OpenMP performs consistently better than HPX and both versions of OMPX on the microbenchmarks.

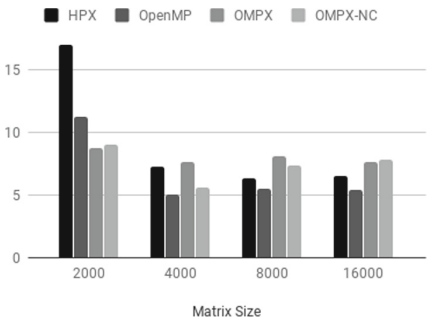

Fig. 8. Jacobi speedup

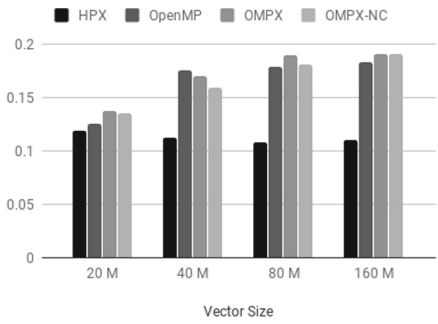

Fig. 9. Stencil speedup

We have four kernel benchmarks to evaluate the OMPX runtime: LU, Jacobi, 1D Stencil, and Nqueens. The LU decomposition benchmark divides the matrix into blocks and works on it in place, with each task writing to one block and reading from multiple. Jacobi iteratively solves the 2D heat equation for a matrix

that is divided up into chunks of rows, with each task writing one chunk and reading from three. 1D stencil solves the 1D heat equation for an array which is divided into chunks that the tasks operate on, similar to Jacobi. In each of these kernels, the only synchronizations are task dependencies and a final wait after the tasks have been created. The Nqueens benchmark solves the Nqueens problem on a given board size, and has no data dependencies, only taskwait.

The Jacobi results in Fig. 8 use the best chunk size for each runtime and input size. The performance at most chunk sizes was similar across all 4 approaches, but with the HPX versions, the performance jumped substantially at chunk sizes that were small enough to fit into cache. This was in the range of 2–8 rows per chunk, depending on the problem size, while the OpenMP version did best when the overall number of chunks was close to the number of threads. The 1D stencil is similar to Jacobi, using task dependencies, but with less data reuse. We can see in Fig. 9, the overhead introduced with OMPTX increases, but still close to the OpenMP version. The HPX version achieves near linear scaling. The only change introduced in the HPX version is the initialization of futures, which is done parallel, and does not need to be done in OpenMP.

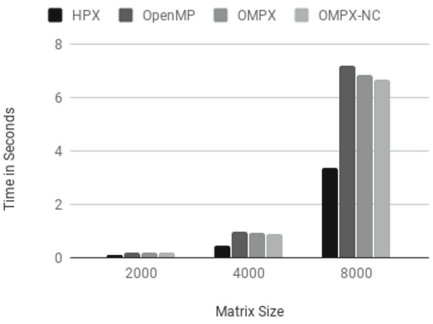

Fig. 10. LU execution time

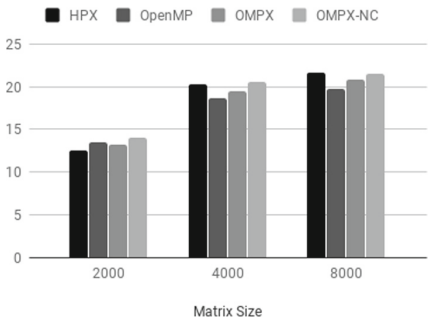

Fig. 11. LU speedup

We see the speedup of LU in Fig. 11, with OpenMP having the best speedup. However, if we look at the execution times for LU in Fig. 10 we see that that HPX has the best overall execution time. The 1D-Stencil and Jacobi kernels do not have such an anomaly, but the Nqueens kernel does. We see in Fig. 12 the performance of OpenMP is consistently the best. However, HPX shows better scaling in Fig. 13.

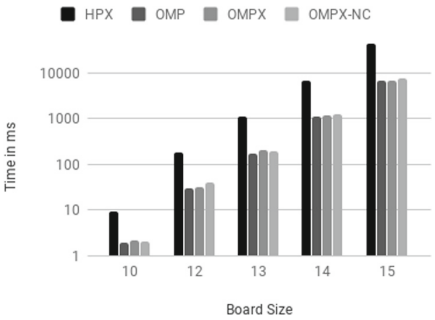

Fig. 12. Nqueens execution time

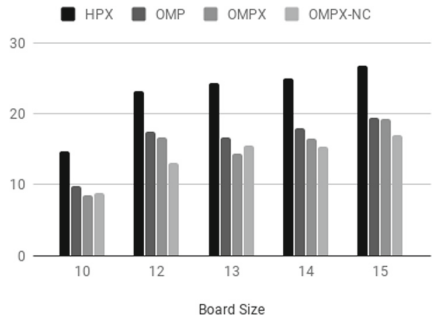

Fig. 13. Nqueens speedup

6 Related Work

OCR [14] is a distributed tasking runtime with a very restricted interface. All work is done in tasks and synchronization is done with task dependencies that form a directed acyclic graph, or DAG.

Parsec [9] is another purely task based distributed framework. It provides an abstract interface for defining tasks and their dependencies, which is translated to C by a compiler. The underlying runtime uses MPI with predefined data layouts the programmer can choose from. Legion [5] is a library based approach using C++. Like Parsec, it provides an abstract syntax to define tasks and their dependencies. The data is not simply wrapped like it is in OCR, the programmer must describe how abstract sets or logical regions of data should be populated, so the runtime can precisely place tasks and segments of data on separate memory or devices. Regent [16] is a higher level interface to Legion that is easier to use.

XcalableMP [13] extends OpenMP to include distributed computing, using a directive based approach, and has recently [17] added task related features for distributed computing. The previous generation of distributed memory taking frameworks include Habanero [8] and Chapel. Some of the national labs have developed distributed tasking frameworks that have gained widespread use with Argobots [15] and Kokkos [11]. The necessary functionality to implement OpenMP that is provided by HPX is also introduced in other tasking implementations that are not distributed, like OmpSs [10], StarPU [3], quark [19], intel TBB [2], and qthreads [18].

7 Conclusions

We have introduced OMPX, an HPX based implementation of the Intel OpenMP runtime, and discussed how a multi-paradigm programming interface like OpenMP can be mapped to a purely task based library like HPX. We have used microbenchmarks and kernel applications to to compare the performance of HPX, the Intel OpenMP runtime, and two versions of OMPX. In the benchmarks where data dependencies existed, HPX and OMPX were able to leverage

locality in a way that OpenMP currently does not to achieve superior performance. Additional benchmarks using larger applications would be desirable, but applications that are task dependency based are not widely available.

With the microbenchmarks, we can also see that constructs that do not translate directly to HPX, specifically barrier and thread related constructs, have worse performance in HPX and OMPX. Executors initially had consistently worse performance than atomics, but they were improved and optimized. Now the version of OMPX that uses executors is often faster than the non compliant version that uses atomics. The HPX library continues to expand, and includes a new resource manager to control how and where tasks are executed. This could be integrated into future versions of OMPX to further improve performance.

The next major extension to OMPX would include support for a distributed environment and evaluate the different approaches to do so. This would require initial exploration to determine how much compiler work and restrictions to OpenMP would be needed. This would also include evaluating automatic task distribution and the benefits of manually placing tasks using some existing OpenMP abstraction or adding totally new construct to OpenMP.

References

1. Intel OpenMP* runtime. https://www.openmprtl.org/
2. Intel threading building blocks user guide. https://software.intel.com/en-us/node/506045
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). https://doi.org/10.1007/978-3-642-03869-3_80
4. Ayguade, E., et al.: The design of OpenMP tasks. IEEE Trans. Parallel Distrib. Syst. **20**(3), 404–418 (2009)
5. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage And Analysis, p. 66. IEEE Computer Society Press (2012)
6. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 4.0
7. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for OpenMP tasks. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 271–274. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_24
8. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the new adventures of old x10. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, pp. 51–61. ACM (2011)
9. Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: PTG: an abstraction for unhindered parallelism. In: 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), pp. 21–30. IEEE (2014)
10. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Proces. Lett. **21**(02), 173–193 (2011)

11. Carter Edwards, H., Sunderland, D.: Kokkos array performance-portable manycore programming model. In: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2012, pp. 1–10. ACM, New York (2012)
12. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: 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, p. 6. ACM (2014)
13. Lee, J., Sato, M.: Implementation and performance evaluation of XcalableMP: a parallel programming language for distributed memory systems. In: 2010 39th International Conference on Parallel Processing Workshops (ICPPW), pp. 413–420. IEEE (2010)
14. Mattson, T.G., et al.: The open community runtime: a runtime system for extreme scale computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7, September 2016
15. Seo, S., et al.: Argobots: a lightweight low-level threading and tasking framework. IEEE Trans. Parallel Distrib. Syst. **29**(3), 512–526 (2018)
16. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: a high-productivity programming language for HPC with logical regions. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, TX, USA, November, Austin (2015)
17. Tsugane, k., Lee, J., Murai, H., Sato, M.: Multi-tasking execution in PGAS language XcalableMP and communication optimization on many-core clusters. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, pp. 75–85. ACM, New York (2018)
18. Wheeler, K.B., Murphy, R.C., Thain, D.: Qthreads: an API for programming with millions of lightweight threads. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–8. IEEE (2008)
19. Yarkhan, A., Kurzak, J., Dongarra, J.: Quark users guide. Innovative Computing Laboratory, University of Tennessee, Electrical Engineering and Computer Science (2011)