LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-778700

Making OpenMP Ready for C++ Executors

T. R. W. Scogland, D. Sunderland, S. L. Olivier,
D. S. Hollman, N. Evans, B. R. de Supinski

June 20, 2019

international workshop on OpenMP: iWOMP
Auckland, New Zealand
September 1, 2019 through September 1, 2019

Disclaimer

Making OpenMP Ready for C++ Executors

Thomas R.W. Scogland[1], Dan Sunderland[2], Stephen L. Olivier[2], David S. Hollman[2], Noah Evans[2], and Bronis R. de Supinski[1]

[1] Lawrence Livermore National Laboratory
{tscogland,bronis}@llnl.gov
[2] Center for Computing Research, Sandia National Laboratories
{dsunder,slolivi,dshollm,nevans}@sandia.gov

Abstract. For at least the last 20 years, many have tried to create a general resource management system to support interoperability across various concurrent libraries. The previous strategies all suffered from additional toolchain requirements, and/or a usage of a shared programing model that assumed it owned/controlled access to all resources available to the program. None of these techniques have achieved wide spread adoption. The ubiquity of OpenMP coupled with C++ developing a standard way to describe many different concurrent paradigms (C++23 executors) would allow OpenMP to assume the role of a general resource manager without requiring user code written directly in OpenMP. With a few added features such as the ability to use unbound threads to execute tasks and to specify a task "width", many interesting concurrent frameworks could be developed in native OpenMP and achieve high performance. Further, one could create concrete C++ OpenMP executors that enable support for general C++ executor based codes, which would allow Fortran, C, and C++ codes to use the same underlying concurrent framework when expressed as native OpenMP or using language specific features. Effectively, OpenMP would become the de facto solution for a problem that has long plagued the HPC community.

Keywords: C++ Executors, OpenMP Tasks

1 Introduction

As high performance simulations reach extreme scales, the software engineering and resource management challenges have become increasingly important. In particular, managing machine-level parallelism, large numbers of threads, and memory access patterns can be essential as individual machine nodes become more capable and as the costs of data movement become prohibitive.

To manage the complexity of these systems, performance portability layers (e.g. RAJA [13], Kokkos [7]) that support platform independent code written in a higher-level abstraction are gaining wide adoption. In the broader computer science community, a similar approach of using higher-level work-runner abstractions has taken hold to allow algorithms to be expressed independently of the

underlying execution system. These various efforts have spawned the current effort to define a fundamental executor concept and interface for C++, currently targeting C++23. This concept would support the use of user or vendor-defined executors with standard library algorithms to execute those algorithms in arbitrary contexts. Executors would generalize many aspects of RAJA and Kokkos, and would provide a common interface in which the next generation of platform-independent libraries could be written.

While these execution interfaces can make a component portable across different programming models or architectures, only code written using those interfaces gains those benefits. Simulations often consist of multiple components, libraries, and even languages. Composability between components in these complex systems can complicate their correct and effective use. The problem is often most severe when different components use different runtime systems, with each runtime system competing for resources. Attempts to solve the composability problem would provide application-level resource management [10] or a common substrates for resource management [15,8]. An especially promising approach uses OpenMP as a common thread pool and resource layer beneath other abstractions. Integrating a code written with OpenMP with a code using the RAJA or Kokkos OpenMP backend is no harder than integrating OpenMP codes, allowing modern C++ to interface (relatively) seamlessly with the occasional 30-year-old Fortran library that nobody admits to needing in their code but always seems to be there.

In a C++ executors world, this approach requires an implementation of the executor concept on top of OpenMP. This requirement is not, in itself, a problem. A parallel loop or a runner is straightforward to implement in OpenMP but executors and Kokkos and, to some extent, RAJA use a model that does not ideally match OpenMP. Some patterns cannot be expressed in OpenMP while adhering to their interfaces. Performance will suffer if these patterns are not enabled.

Our position paper proposes that two new developments, the executors proposal for the 2023 C++ standard and the increasing use of OpenMP as a resource manager, enable unique and synergistic solutions to these problems. The common timeframe for these standards provides a unique opportunity to codesign them. C++ is already the lingua franca for performance portability layers in HPC, and OpenMP is becoming the de facto runtime composition layer included in every major compiler implementation. Marrying the two in a way that provides best-in-class performance and composability for and between both models will open new possibilities for more performant, more maintainable, and more easily composed components and scientific applications.

This paper makes the following contributions:

- An analysis of the state of OpenMP tasking and offload from the perspective of abstraction layers and C++ executors;
- A proposal of two extensions to OpenMP to improve the composability of tasks, target regions, and parallel loops, as well as making asynchronous tasks more amenable to abstractions; and

– A discussion of the feasibility of implementing the extensions in both a research runtime and the OpenMP standard.

2 Background

For nearly a decade, the C++ standards committee (ISO/IEC JTC1/SC22/WG21) has iterated on numerous designs of generic abstractions for execution, known as executors. Representing one of the most ambitious generic library design efforts of its kind, the current proposal [11] aims to address the needs of vastly different application domains, from embedded computing to high performance computing and everything in between. At least a subset of the features proposed therein are likely to be merged into the C++ standard working draft early in the C++23 cycle [18], with other portions expected to follow shortly thereafter.

While the exact syntactic details of executors remain undecided, the various designs have fairly consistently focused several important axes in the design space. The most prominent of these is the expression of cardinality of work, distinguished by the elaboration of separate interfaces for single and bulk execution, somewhat akin to providing both a `parallel for` and `task` interface. Different prominent stakeholders have tended to see either of these extremes as fundamental: GPGPU stakeholders, for instance, tend to consider bulk execution to be fundamental. Networking stakeholders, on the other hand, tend to see single execution as the fundamental operation. Designs that can incorporate both of these world views have led to new paradigms in generic programming [12].

Another fundamental design axis that has appeared consistently across the history of executors is the distinction between one-way execution ("fire-and-forget" work) and two-way execution—that is, work that requires some means of signaling completion, failure, and/or cancellation. Programming models based on promises and futures, dating back to at least the late 1970s [4], are a traditional example of the latter. Recently, the design of two-way executors has begun to converge on push-style programming models [17] due to their ability to unify the observer pattern [9] with future/promise semantics.

Across all of these dimensions, the basic interfaces of all proposals has had one thing in common: Much like OpenMP's `task`s, when in a parallel region, they abstract over asynchrony. Work is allowed to be queued for later execution, or run immediately, possibly singly or in bulk, and possibly with or without a propagated value, but the basic expression of algorithms using any of these interfaces is based upon the ability to asynchronously generate work. At present, OpenMP can model single or bulk execution with or without signaling completion. Enabling asynchronous scheduling of these units of work however requires that all the associated code can be wrapped in a `parallel` region, which is not possible due to the interface itself as well as interference with the rest of the program. There is also currently no way to model an asynchronous check for completion of a task. Though it can be modeled with atomics or similar, we leave exploration of this aspect for future work.

3 Requirements and Proposed Features

Since its introduction in version 3.0 of the specification, OpenMP support for task parallelism has evolved into an increasingly powerful tool to expose parallelism in application to be exploited by the OpenMP runtime system. Expressiveness has been expanded by allowing more sophisticated dependences and synchronizations between tasks, and the scope of task parallelism in OpenMP has expanded to encompass asynchronous offload to accelerators. However, the awkward relationship of task parallelism to thread parallelism has changed little from OpenMP 3.0 to 5.0. Otherwise promising use cases for task parallelism, of which C++ executors implementation is but one, are rendered difficult or impossible by the limitations of this relationship. We outline some of the issues below, along with a high-level view of some potential future changes to the specification to address them. The changes are comparatively light on new syntax, and the first is only semantic.

3.1 "Free-agent" Threads

The first issue, and the one encountered even by programmers writing the simplest OpenMP program using tasks, is the requirement to create a team of threads even for a program comprised entirely of explicit tasks. In the absence of such a team of threads, the tasks would be executed only sequentially. This leads to the frequent idiom combining `parallel` and `single` or `master` to start a team of threads and then begin task creation on only one thread of the team, as shown in Figure 1.

```
1  int main()
2  {
3      #pragma omp parallel
4      #pragma omp single
5      {
6          #pragma omp task
7              func1();
8          #pragma omp task
9              func2();
10     } // tasks join here
11 }
```

Fig. 1. Asynchronous tasking without free-agent threads

Related shared memory tasking frameworks like OmpSs [6], Cilk [14], Argobots [16], and Qthreads [19] simply make threads available for executing tasks immediately at program startup. While many OpenMP implementations already create threads upon initialization of the run time library, the current semantics of OpenMP forbid using those threads to execute tasks until one or more teams

have been created. The constraint is more than an inconvenience, because the creation of teams segregates the available threads. Since neither threads nor tasks can be exchanged between two different teams of threads, the effect is to limit composability and load balancing.

The solution proposed for future OpenMP versions is to allow a pool of "free agent" threads maintained by the implementation to exist outside of a team and available to execute tasks. This new semantic would allow a program to execute tasks asynchronously on an implementation's thread pool without creating a parallel region, as shown in Figure 2.

```
1  int main()
2  {
3     #pragma omp task
4         func1();  // executes on an available thread in the pool
5     #pragma omp task
6         func2();  // executes on another available thread in the pool
7     #pragma omp taskwait // tasks join here
8  }
```

Fig. 2. Asynchronous tasking with free-agent threads

The effect of the code, assuming that the implementation has a pool of threads ready to execute the tasks, is equivalent to Figure 1. While the difference is just a few lines, it not only simplifies reasoning about how to use of tasks, a boon especially for new users, but also places fewer constraints on interleaving tasks with parallel regions or parallel loops.

3.2 Task Width

Another important issue is that OpenMP currently provides no way for the programmer to indicate when creating a task that the task includes further parallelism inside the task or to what degree. The implementation becomes aware of the nested parallelism only at the time the nested constructs within the task are encountered. If, however, the implementation had knowledge of the nested parallelism at task creation, it could plan to execute the task where and when adequate threads are available for the nested parallelism. The solution proposed for future OpenMP versions is to admit a clause on task-generating constructs to specify the degree of nested parallelism present in the task.

We propose to add a `width` clause to the `task` directive. The argument to new clause would indicate the amount of nested parallelism created within the task, as shown in Figure 3. A more restrictive way to accomplish the same effect would be to allow a `nowait` clause on the `parallel` construct, transforming its region into a task. The example in Figure 4 shows the equivalent code using this alternate approach.

```
#pragma omp task
    func1();  // no width specified, so assume 1 thread only
#pragma omp task width(5)
{
    // width(5) indicates internal parallelism
    #pragma omp parallel for num_threads(5)
        for (int i = 0; i < MAX; ++i)
            func2(i);
}
```

Fig. 3. Parallelism inside a task with a specified width

```
#pragma omp task
    func1();  // no width specified, so assume 1 thread only
#pragma omp parallel for num_threads(5) nowait
    for (int i = 0; i < MAX; ++i)
        func2(i);
}
```

Fig. 4. Asynchronous parallel regions

A point in favor admitting the `nowait` clause on the `parallel` construct would be symmetry with the `target` construct, which already admits the clause. It would also be a convenient way to express asynchronous bulk parallelism. However, it does not support some use cases that are supported by task width for interoperability of OpenMP users' programs with libraries that use OpenMP internally. Consider the example shown in Figure 5, in which the function call is made to a math library routine. Because the nested parallelism is hidden inside the library routine, the more restricted `parallel nowait` idiom does not support this use case.

An open question regarding semantics is whether the number of threads in the clause indicates maximum or minimum nested parallelism within the task. Additionally, should it reflect only first nesting level of parallelism, or all levels, if more than one level of parallelism is present within the task? This information may need be readily available even to the programmer if the nested parallelism is inside library calls. Even the basic indication that there exists nested parallelism with in the task, regardless of size gives the runtime system more information than it currently has for scheduling.

3.3 Broader Applicability

Progress on these issues is important not only for the success of OpenMP as an implementation platform for C++ executors, but also for other important use case scenarios. Among these use cases are real-time systems and GUI-based programs, in which an event loop runs continuously and spawns new work periodically or based on user input and sensors. Ever-increasing levels of hardware

```
void user_func(...)
{
    #pragma omp task width(5)
    {
        blas_library_call(...);   // allowed to use 5 threads internally
    }
}

// (Inside the library)
void blas_library_call(...)
{
    #pragma omp parallel for // gets up to "width" threads
        for (int i = 0; i < MAX; ++i)
            ...
}
```

Fig. 5. Task with a width calling library code

parallelism also motivate more flexible mechanisms to expose application parallelism and provide more information to inform run time task scheduling.

Increasingly, single-source programming models for portable utilization of heterogeneous compute resources, in which applications provide a single implementation that is generic over execution model and resources, are a popular approach to heterogeneous library design. Kokkos [7] is one such library that has had a significant impact on major portions of the ISO-C++ executor design process. Kokkos provides the concept of an `ExecutionSpace` that closely resembles an executor. Users write code that is generic over the specific `ExecutionSpace` type in order to express, with a single source, an algorithm that can run with multiple execution models.

The obvious concern in the design of the `ExecutionSpace` concept is restricting the programming model enough to provide low-overhead performance (relative to an execution-model-specific implementation) on all supported `ExecutionSpace` types. Specifically, Kokkos provides `ExecutionSpace` implementations for OpenMP, CUDA, thread-pool-based execution, and serial execution, among others. The restrictions on the `ExecutionSpace` design thus include abstractions that can map to a notional "intersection" of execution model restrictions for all of the supported backends. (ISO-C++ executor design is very similar in this respect.)

The extensions to the OpenMP programming model presented herein do not represent an expansion of that intersection, since (for instance) serial execution will always be a supported execution model. However, expanding the "intersection" of a subset of the supported execution models often enables an increase in the precision of the user's mental performance model for some generic code because programming model abstractions can be mapped to a smaller "outer product" of performance characteristics.

In this context the `nowait` clause on the `parallel` construct has a semantic much more similar to that of a CUDA kernel launch than the traditional use of the `parallel` construct. The restricted programming model that encompasses both the synchronous `parallel` construct's semantics and the asynchronous semantics of a CUDA kernel launch requires the user to assume that the *earliest* an algorithm's execution can begin is immediately upon invocation, and the *latest* the algorithm can finish execution is upon return from the next call to an explicit `Kokkos::fence()` on the `ExecutionSpace` used by the algorithm. They cannot rely on the encountering thread to block, or not to block. Presentation of a consistently asynchronous model, or at least a potentially asynchronous one, can help reduce the variability in behavior of the code across platforms.

4 Feature Interactions and Feasibility

The main challenge with this set of extensions is deciding how arrangements of asynchronous execution, tasks, parallel regions, and widths that were not previously possible can interact without harming backward compatibility or performance unduly. This section will discuss the various trade-offs and considerations necessary to integrate free-agent threads and task widths into OpenMP.

4.1 Task Joining

As discussed previously, OpenMP tasks either execute immediately in the encountering thread, in a serial context, or are joined at the end of their enclosing parallel region. As a result, there is currently no way for tasks to logically "run off" the end of a program. If however we allow tasks to run asynchronously at the top level of the program, we need to define what happens if tasks are still executing when `main` ends. For example, take the code in Figure 2, if there were no `taskwait` at the end of `main` there would be no guarantee that either `func1` or `func2` would be done at the end of the program.

Given the way OpenMP is currently defined, there is logically a parallel region around the entire program comprising only the initial thread. If we naively extend this to make free agent threads accessible, we would assume that these tasks should join on return. Given the considerations of implementations however, and the fact we want OpenMP to be usable when `main` is compiled without it, our recommendation is that tasks are allowed to be cancelled by the end of the program. Users always have the option to use either `taskwait` or `taskgroup` to join tasks if they want them joined.

4.2 Threads Available for `parallel`

Since threads may now be executing task work alongside the initial thread, it is possible to encounter a synchronous `parallel` region while some threads are busy. There are a number of options available to handle this situation:

1. Run the parallel region immediately with fewer threads.

2. Make `parallel` wait for the concurrent tasks to pause or finish before starting with all threads it would otherwise have been allotted.
3. Begin the parallel region with available threads and join others in as the tasks either finish or reach scheduling points.

Given the potential performance implications, the user will almost certainly want control over the choice of the options above. However the choice of default has implications both for performance and for backwards compatibility. When an OpenMP `parallel` region starts, it is provided with some number of threads. The actual number is always implementation-defined, and can be affected by a variety of environment variables through OpenMP's Internal Control Variables (ICVs). That said, when the *dyn-var* ICV [3] is set to false, the number of threads in each parallel region is fixed, and codes are allowed to rely on this property to access thread-local state and for various other reasons.

Given the requirements imposed by *dyn-var*, we propose that either option two or three is used when *dyn-var* is true, and allow only option one when it is false. The user can then control the general behavior they prefer with an existing ICV, get a more specific thread count with a task with a width or asynchronous parallel region, or use a taskwait to ensure tasks have joined before the `parallel` region starts.

4.3 Interactions Between Width and Num-Threads

The concept of the `width` clause for a task is simple on the surface–it tells the runtime that the task being created should be provided with a given level of parallelism, and that something in the dynamic scope of that task will make use of it. Unlike with `parallel nowait` there is no guarantee precisely *when* that parallelism will be used, so that many threads don't necessarily need to be immediately available. Given the way OpenMP is specified today, the simplest way to think about translating a task with a width of six would be to set the `nthreads-var` ICV to six inside a task as in Figure 6.

```
1 #pragma omp task // width(6)
2 {
3     omp_set_num_threads(6);
4 }
```

Fig. 6. A naive de-sugaring of a task with a width

This approach provides the desired behavior of controlling the number of threads used in a dynamic scope, and allows different values of width for tasks nested within one another while re-using a well established mechanism. It gets

[3] The value set by the `OMP_DYNAMIC` environment variable.

surprisingly close to the overall goal, even to providing the appropriate level of parallelism when calling into a library, although it does not provide the runtime or compiler with appropriate scheduling information. The downside is that if the number of threads is set this way it overrides the value from `omp_get_max_threads()`, and it can be overridden relatively easily. It may be more appropriate to employ a mechanism like the thread limit on teams to resist called code expanding past the resources allotted, and to provide a method to interrogate the total number of threads available. While a parallel run in the tasks context could only have the number specified by the limit, an asynchronous task there could request more.

4.4 Feasibility

In order to explore the design space, we created an initial prototype runtime implementing the new semantics we describe for tasks outside of a parallel context. We considered implementation in the LLVM OpenMP runtime, GOMP, and BOLT [1] which is a user-level threaded version of the LLVM runtime implemented on top of argobots [16]. The LLVM and GOMP runtimes could both implement the pattern we have discussed, but currently rely on the scoping of parallel regions for memory management of their tasking runtimes. For example, while the task-running threads and per-thread contexts persist across parallel regions the task queues and attendant metadata do not. However, BOLT does not, instead relying on the argobots system to manage some of these details. As a result, a naive prototype is as simple as removing the checks for whether tasks should be allowed to be run asynchronously outside of a parallel context[4].

Given the structure of other runtimes we expect implementation of this feature to require a rework in the lifetime management of data structures, but relatively little change in implementation logic other than to take advantage of newly available information. We do not provide performance comparisons in this paper as none of the proposed features have a direct impact on performance in our implementation due to the underlying structure of BOLT. As such the prototype performs identically to a stock BOLT library, simply allowing expression of tasks in an alternative manner. We may explore performance impact on applications composed of multiple components and higher-level runtimes in the future.

Overall, free-agent threads, tasks with a width, and asynchronous parallel regions appear feasible from both a runtime and specification perspective. After further experimentation and performance testing with codes in the wild, some defaults and further mechanisms may become desirable. That said the base mechanisms show strong promise for being implementable and providing substantial benefit to composability of OpenMP with itself as well as making it more practical as a substrate for libraries and systems with asynchronous or thread-pool-like interfaces such as C++ executors.

[4] In fact, the original naive prototype only required changing eight lines of code.

5 Related Work

The emergence of manycore and heterogeneous systems and increasing use of hybrid MPI-X programming models has led to a proliferation of frameworks to support performance portability, composition, interoperability, and resource management. Kokkos [7], and RAJA [13] provide performance portability frameworks, however they do so at the middleware level. By specifying memory and concurrency at the language standards level, performance portability policy support becomes a *compiler* rather than a middleware capability. This appraoch ensures support across platforms and provides vendor independent ways of implementing cross platform high performance simulation software.

OmpSs [5] is a task-based OpenMP-like programming model that has inspired many of the current features and behaviors of task parallelism in OpenMP. BOLT [1] provides an alternative implementation of the LLVM OpenMP runtime ABI on top of Argobots [16], which offers user-level threading to support over-decomposition and deeper nesting than is feasible with OS-level thread models. For hybrid MPI+X programming, OmpSs provides direct integration with MPI and BOLT provides integration with the MPICH implementation of MPI [2] using the Argobots [16] runtime framework. StarPU [3] provides integration of heterogeneous computing and software resources in a uniform manner via the runtime. It includes OpenMP 4 with a focus on task parallelism and extensions to support run time scheduling optimizations.

Lithe [15] provides a common runtime substrate to enable coscheduling of runtimes similar to CPU inheritance scheduling [8] while adding a hardware thread abstraction to ensure that multiple runtimes do not oversubscribe system resources. Modified versions of the runtimes (e.g., OpenMP and Threading Building Blocks) are required. The QUO [10] library provides an alternative approach to composing MPI and threading runtimes, managing heterogeneous thread and memory resources at the application level and manually quiescing and running thread groups via the pthreads system interface, thus manually avoid oversubscription of system resources in multiple interacting runtimes.

Our approach is based on the view that the upcoming incoporation of executors into the C++ language standard will make their use commonplace, and that leveraging the many high quality OpenMP implementations in open source and vendor compiler suites is a promising way both to support executors and to integrate C++ programs using them with native OpenMP code. Like other solutions to address the problems of composition and thread resource management, we seek to avoid unintended oversubscription of hardware execution resources. However, using OpenMP as the integration point provides the benefits of greater portability and high-level abstraction compared to ad-hoc and system-level frameworks.

6 Conclusions

Composing multiple frameworks and performance portability layers is an increasingly necessary for high performance computing at scale. However, standardizing

on the portability layer has been difficult, leading to multiple implementations with no clear standard interface. In this paper we have argued that the ubiquity of OpenMP and the coming executors concept in the C++ standard provide a unique opportunity to ensure that both standards grow to a point where they can compose with one another to efficiently and effectively integrate components built with state-of-the-art techniques in C++ with the extensive performance-oriented ecosystem of OpenMP applications and libraries.

We analyzed the requirements for the extensions to tasking as well as the necessary extensions to the OpenMP standard to provide the necessary functionality. Specifically, we propose incorporating the concept of "free-agent" threads into OpenMP, allowing asynchronous execution of tasks and parallelism without a scoping restriction, and extending tasks with a width, allowing a task to represent a quantity of resources allocated to the code executed inside it. Finally we discussed an implementation of "free agent" threads in an OpenMP runtime along with some of the major design considerations for implementing these changes in the specification. While exploring this approach we found a few more potential future extension points, including a non-blocking mechanism for checking if tasks are complete and a mechanism for executing tasks in other teams, but we leave these for future work.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

1. BOLT: A lightning-fast OpenMP implementation, `https://bolt-omp.org/`
2. Argonne National Laboratory: MPICH2: High performance and portable message passing. http://www.mcs.anl.gov/research/projects/mpich2, `http://www.mcs.anl.gov/research/projects/mpich2`
3. Augonnet, C., Thibault, S., Namyst, R.: StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Tech. Rep. RR-7240, Laboratoire Bordelais de Recherche en Informatique - LaBRI, RUNTIME - INRIA Bordeaux - Sud-Ouest (Mar 2010), `http://hal.inria.fr/inria-00467677`
4. Baker, H.C., Hewitt, C.: The incremental garbage collection of processes. ACM SIGPLAN Notices **12**(8), 55–59 (Aug 1977). https://doi.org/10.1145/872734.806932
5. Bueno, J., Duran, A., Martorell, X., Ayguadé, E., Badia, R.M., Labarta, J.: Poster: Programming Clusters of GPUs with OmpSs. In: International Conference for High Performance Computing, Networking, Storage and Analysis

(SuperComputing). ACM (May 2011). https://doi.org/10.1145/1995896.1995961, http://portal.acm.org/citation.cfm?id=1995896.1995961&coll=DL&dl=GUIDE&CFID=61704752&CFTOKEN=92261478
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 Processing Letters **21**(2), 173–193 (2011), http://www.worldscinet.com/abstract?id=pii:S0129626411000151
7. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling many-core performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202 – 3216 (2014). https://doi.org/https://doi.org/10.1016/j.jpdc.2014.07.003, http://www.sciencedirect.com/science/article/pii/S0743731514001257, domain-Specific Languages and High-Level Frameworks for High-Performance Computing
8. Ford, B., Susarla, S.: CPU inheritance scheduling. In: OSDI. vol. 96, pp. 91–105 (1996)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
10. Gutiérrez, S.K., Davis, K., Arnold, D.C., Baker, R.S., Robey, R.W., McCormick, P., Holladay, D., Dahl, J.A., Zerr, R.J., Weik, F., Junghans, C.: Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications. In: 2017 IEEE International Parallel & Distributed Processing Symposium (IPDPS). Orlando, Florida (2017)
11. Hoberock, J., Garland, M., Kohlhoff, C., Mysen, C., Edwards, C., Hollman, D.: P0443r10: A unified executors proposal for C++ (Jan 2019), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0443r10.html
12. Hollman, D., Kohlhoff, C., Lelbach, B., Hoberock, J., Brown, G., Dominiak, M.: P1393r0: A general property customization mechanism (Jan 2019), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1393r0.html
13. Hornung, R., Keasler, J.: The RAJA portability layer: Overview and status. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2014)
14. Leiserson, C.E.: The cilk++ concurrency platform. The Journal of Supercomputing **51**(3), 244–257 (2010)
15. Pan, H., Hindman, B., Asanović, K.: Composing parallel software efficiently with lithe. ACM Sigplan Notices **45**(6), 376–387 (2010)
16. Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castelló, A., Genet, D., Herault, T., et al.: Argobots: A lightweight low-level threading and tasking framework. IEEE Transactions on Parallel and Distributed Systems **29**(3), 512–526 (2018)
17. Shoop, K., Niebler, E., Howes, L.: P1055r0: A modest executor proposal (Apr 2018), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1055r0.pdf
18. Sutter, H.: Trip report: Winter ISO C++ standards meeting (Kona) (Feb 2019), https://herbsutter.com/2019/02/23/trip-report-winter-iso-c-standards-meeting-kona/
19. 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. pp. 1–8. IEEE (2008)