INVITED PAPER

The Ongoing Evolution of OpenMP

This paper discusses the OpenMP framework's past, current status, and anticipated future in the face of the evolving CPU and accelerator landscape.

By Bronis R. de Supinski, Thomas R. W. Scogland, Alejandro Duran, Michael Klemm, Sergi Mateo Bellido, Stephen L. Olivier, Christian Terboven, and Timothy G. Mattson

ABSTRACT | This paper presents an overview of the past, present and future of the OpenMP application programming interface (API). While the API originally specified a small set of directives that guided shared memory fork-join parallelization of loops and program sections, OpenMP now provides a richer set of directives that capture a wide range of parallelization strategies that are not strictly limited to shared memory. As we look toward the future of OpenMP, we immediately see further evolution of the support for that range of parallelization strategies and the addition of direct support for debugging and performance analysis tools. Looking beyond the next major release of the specification of the OpenMP API, we expect the specification eventually to include support for more parallelization strategies and to embrace closer integration into its Fortran, C and, in particular, C++ base languages, which will likely require the API to adopt additional programming abstractions.

KEYWORDS | Accelerator architectures; computer architecture; computer science; computers and information processing; memory management; multicore processing; multithreading; parallel architectures; parallel processing; parallel programming; programming

Manuscript received March 9, 2017; revised June 18, 2018; accepted June 27, 2018. Date of publication August 13, 2018; date of current version October 25, 2018. This article has been authored by Lawrence Livermore National Security, LLC under Contract DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains, a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or to allow others to do so, for United States Government purposes. Released as LLNLJRNL- 754009.
(Corresponding author: Bronis R. de Supinski.)
B. R. de Supinski and **T. R. W. Scogland** are with Livermore Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551-0808 USA (e-mail: bronis@llnl.gov).
A. Duran is with Intel Corporation Iberia, Madrid 28020, Spain.
M. Klemm is with Intel Deutschland GmbH, 85622 Feldkirchen, Germany.
S. Mateo Bellido is with the Barcelona Supercomputing Center, 08034 Barcelona, Spain.
S. L. Olivier is with Sandia National Laboratories, Albuquerque, NM 87185 USA.
C. Terboven is with RWTH Aachen University, 52074 Aachen, Germany.
T. G. Mattson is with Intel, Hillsboro, OR 97124 USA.

Digital Object Identifier 10.1109/JPROC.2018.2853600

I. INTRODUCTION

The OpenMP effort began in 1996 when a handful of vendors (DEC, HP, IBM, Intel, Kuck and Associates, and SGI) were brought together by the Accelerated Strategic Computing Initiative (ASCI) of the Department of Energy (DOE) to create a portable application programming interface (API) for shared memory computers based on their various implementations of, and extensions to, the Parallel Computing Forum directives [26]. Vendors do not typically work well together unless an outside force compels cooperation. Mary Zosel and the ASCI parallel tools team provided that compulsion by communicating that ASCI would only purchase systems with a portable API for shared memory programming. Their role in the beginning of OpenMP ensured that it met the needs of HPC application programmers.

Early public presentations about the project [13] clearly defined the initial group's goals:

- to support portable, efficient and comprehensible shared-memory parallel programs;
- to produce specifications based on common practice that could be readily implemented;
- to provide a consistent API for Fortran, C and C++ to the most reasonable extent possible;
- to be lean and mean, i.e., to be only as large as required to express important control-parallel, shared-memory programs but no larger;
- to ensure API versions are backwards compatible;
- to support *serial equivalence*, i.e., for OpenMP programs to produce the same result whether run serially or in parallel, to the greatest possible extent.

The first OpenMP specification was released in November 1997 at SC97. The early OpenMP community knew that other parallel programming standardization efforts, such as High Performance Fortran (HPF) and MPI 2.0, suffered from multiyear delays as implementors struggled to produce robust, application-ready

```
#pragma omp parallel // fork
{
  // share the loop across threads,
  // reducing into total
  #pragma omp for reduction(+:total)
  for (int i=0; i<N; ++i) {
    total += foo(i);
  }
} // join
```

Fig. 1. ***Basic OpenMP.***

implementations. Thus, OpenMP by design narrowly focused on current practice. This focus led to the availability of multiple vendor-supported implementations within a year of the release of the first specification.

Over time, additional vendors and research organizations joined the effort. A nonprofit corporation, the OpenMP Architecture Review Board (ARB), was created to prevent any single vendor from dominating the standard. The current 32 members of the OpenMP ARB continue to own and to evolve the API to serve the needs of parallel application programmers. The ARB retains many of the original goals in its current mission, which is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable. The OpenMP API now provides a simple and flexible model for developing parallel applications for platforms ranging from embedded systems and accelerator devices to multicore systems. Fig. 1 shows a simple OpenMP example in which `parallel` and `for` directives specify that the basic fork-join parallelism model should create threads and share the iterations of the loop across them, with a reduction performed on the values computed in those threads. This OpenMP syntax has been valid since the release of the first C version of the specification.

OpenMP retains all but two of its original goals. Specifically, OpenMP has evolved to support almost all parallel programming patterns, which necessarily implies a larger specification than originally envisioned. Further, while serial equivalence is still achievable, that range of patterns necessarily leads to many opportunities to deviate from it. Otherwise, the only change to the original goals is that the scope of OpenMP has extended beyond shared memory.

We comprehensively examine the state of OpenMP in anticipation of the imminent release of version 5.0 of the API. We first review the evolution of OpenMP through version 4.5 (Section II). We then provide a more detailed examination of the philosophy that has guided its evolution (Section III). Next, we briefly review the basic concepts and mechanisms that support implementation of the evolving API (Section IV). We then detail some recent (Section V) and impending (Section VI) additions to OpenMP. Finally, we discuss and anticipate some possible directions for its longer term evolution (Section VII).

II. OVERVIEW OF OPENMP'S EVOLUTION

OpenMP is a living language that reflects the needs of its many users. Versions adopt new features, major or minor, for various reasons. Performance motivates the adoption of some features, while expressiveness or maintainability motivate others. In general, Language Committee members identify potential extensions through interactions with their customers or users or through knowledge of the activities in the research community. They bring the potential extensions to the committee and describe how they will improve the specification. If the improvement is based on performance, then they will provide documentation of the potential performance advantages. Even when the benefit involves some other facet, they will usually provide evidence that they do not impede performance, particularly when they are not used. The features are adopted if the Language Committee is convinced that they improve the specification.

As hardware capabilities and the range of supported algorithms have grown, the complexity of the specification has also expanded. Fig. 2 lists the number of pages of the versions of the specification (not including front matter, appendices or indices). The initial OpenMP specification [15] (OpenMP Version 1.0 for Fortran) was 40 pages long. The latest specification [22] (OpenMP 4.5 for Fortran, C and C++) is 303 pages long.

Fig. 2 details OpenMP's evolution. Prior to the release of version 2.5 [18] in 2005, each OpenMP specification addressed a particular base language (i.e., Fortran or C and C++). This division simplified writing the text of the specification, but also created difficulties. First, most of the people working on the Fortran specifications also worked on the C/C++ specifications. Thus, the evolution of the API was hampered since we could not run the two language committees in parallel. Thus, updates to the specification were produced slowly relative to their amount of new material. For example, OpenMP 2.0 for C/C++ [17] (50 pages) was released almost 4 years after OpenMP 1.0 for C/C++ [16] (45 pages) despite the relatively simple extensions that it included.

Not only was the progress of the API slower due to the separate specifications, the separation also allowed the API to have subtle differences across the languages. The

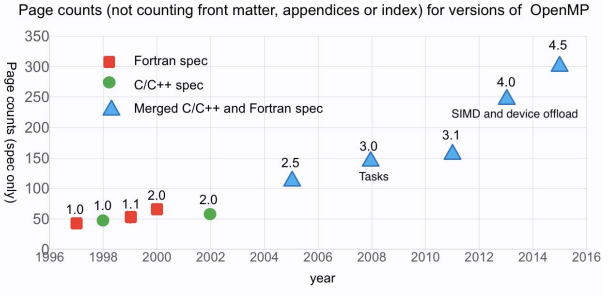

Fig. 2. ***OpenMP specification growth across versions.***

process of merging the separate APIs for the languages into a single specification was a much larger undertaking than any of us expected. That process required us to recast OpenMP's core abstractions much more carefully so they would apply across the languages. The resulting OpenMP version 2.5 specification (117 pages) took three years to create despite adding few new capabilities.

Following the merger of the specifications and the growth in the popularity of the API, as evidenced by the expanding membership of the OpenMP ARB, the pace of the evolution of OpenMP has increased. Today, OpenMP is no longer a simple API for which its full breadth can be learned in less than a day. Nonetheless, the core features of the version 1.0 specifications remain and the goal of backwards compatibility has largely been achieved.

The OpenMP version 3.0 API specification [19] (151 pages) added task-based parallelism. This addition supports irregular parallelism, unlike the original loop-based constructs. OpenMP 3.0 also provided much more control over the existing support for structured parallelism. OpenMP version 3.1 [20] (160 pages) extended the support for structured parallelism, for example, by adding straightforward control of the number of threads used at each level of nested parallelism. OpenMP 3.1 also further refined tasking support. In general, the continued evolution of OpenMP has advanced existing features while also expanding the types of parallel algorithms that the specification supports.

The OpenMP version 4.0 API specification [21] (248 pages) added support for accelerator-based systems through its device constructs. Echoing the API's original purpose, OpenMP 4.0 also standardized directives for single-instruction–multiple-data (SIMD) parallelism, which had become widely supported by many compilers but with subtly different semantics. OpenMP 4.5 [22] (303 pages) added many refinements to those additions. As we later discuss in detail, OpenMP 5.0 will support mechanisms to control data placement in complex, multi-level memory systems. It will also include support for first-party and third-party tools as well as the customary major extensions for the types of parallelism that OpenMP already supports.

In evolving the OpenMP API, we have added features that address nonuniform memory architectures, more complex concurrency control, irregular algorithms, accelerators, and much more. The specification has not grown due to a lack of discipline in its designers. Instead, its growth reflects user demands for new features and how hardware has changed. In that light, a 7.5X increase in size over the course of almost 20 years is not surprising.

III. GUIDING PHILOSOPHY OF OPENMP

OpenMP's general philosophy reflects the ARB's mission to standardize directive-based multi-language high-level parallelism that is performant, productive, and portable. Portability is achieved first and foremost through broad adoption and support. At the highest level, a directive-based approach supports productivity through incremental parallelization and refinement through which user code remains as close to its original serial version as possible while still achieving performance goals. Directives allow the programmer to specify information that a compiler would otherwise not be able to determine but that is often known to the user, or that might require complex and error-prone analysis.

OpenMP provides sensible defaults that often result in high performance but also allows low-level control of aspects for which the compiler and runtime may not deduce high-quality settings. Programmers can thus start from simple usage of OpenMP directives and incrementally increase the level of complexity to expose more and more control over the code transformations applied and parallel execution to achieve higher performance. Despite the growing complexity of OpenMP directives, the OpenMP language is designed to maintain this core principle of directives building on top of each other to support this incremental program evolution.

As we discussed earlier, OpenMP retains many of its original goals, which embodied a general philosophy. However, like the specification, this philosophy has evolved as OpenMP has expanded to support a wider range of parallel programming patterns. This section discusses the evolution of two key aspects of the original philosophy, language independence and serial equivalence, as well as the issue of descriptiveness versus prescriptiveness, a philosophical issue for programming models that has recently received significant attention.

A. Relationship to Base Languages

Although OpenMP began with separate specifications for C/C++ and Fortran, as we discussed in Section II, OpenMP 2.5 merged them into a single document. Although that choice was partly pragmatic—it reduced the effort to move the base languages forward—the original goal of a consistent API across the base languages, which remains a key part of OpenMP's guiding philosophy, was the primary reason. This language independence is one of OpenMP's core strengths since OpenMP has greater portability and generality, not only across C, C++, and Fortran but also in its design as a result.

OpenMP, by itself, is not a language. It provides an API for portably expressing parallelism and concurrency across three independent base languages. As discussed above, OpenMP attempts to provide the same experience and easy interoperability between all three while also being consistent with the specific base language. Thus, to the extent possible, OpenMP relies on the base language for sequential programming constructs. However, some mistakenly claim that a directive-based approach is necessarily limited in scope. In reality, the approach can be Turing complete and a directive could provide any construct that is available in a base language.

```
void example() {
  int a = 0;
#pragma omp parallel
  {
#pragma omp single
    {
      int b = 0;
#pragma omp task
      while (b == 0) {
#pragma omp atomic read seq_cst
        b = a;
      }
#pragma omp task
      {
#pragma omp atomic update seq_cst
        a++;
      }
    }
  }
}
```

Fig. 3. *Trivial OpenMP program.*

Traditionally, OpenMP has limited its scope in several ways. However, we are finding that as the API grows and addresses more programming patterns we must support a larger set of basic programming constructs. As we discuss in Section VI-B, one example is the concept of iterators, which provide structured looping functionality inside the directives themselves. Closures are another example under consideration, as we discuss in Section VII-F. Support for these constructs increases the complexity of the OpenMP compilation pass so some implementers are resisting their addition to the API. In general, we are currently debating the extent to which OpenMP should provide basic programming constructs. Nonetheless, we expect the degree to which OpenMP feels like a general programming language to increase.

Regardless, OpenMP will not become a standalone language and will continue to rely on base languages to specify the bulk of the computation that is to be performed. It will continue to rely heavily on each base language to define the behavior of a given construct within each thread of execution or block of code. Further, we are actively updating OpenMP to support recent base language standards. OpenMP 4.0 added Fortran 2003 [5] as a normative reference while OpenMP 5.0 will add Fortran 2008 [7], C11 [9], C++11 [8], 14 [10], and 17 [11].

The evolution of the base languages in their normative references complicates OpenMP's relationship to them. Before the release of C11 and C++11, C and C++ did not have any well-defined concept of a data race or threading. In fact, the ISO C99 standard [6] does not contain the term "thread" at all, and only contains the word "race" as part of the term "brace." In general, the original normative references did not address parallelization. Thus, OpenMP has provided all threading and memory model semantics for a program that used OpenMP constructs. In order to provide full support for the later C and C++ standards, which include integrated threading models, acquire and release memory models and other built-in parallel concepts, OpenMP must ensure that its semantics do not conflict with those of the base languages. That process has begun with TR6 [23] (Technical Report 6), which provided a preview of OpenMP 5.0 and will continue beyond OpenMP 5.0.

Finally, while a pragma-based approach is natural for Fortran and C programmers, it is not the most natural one for C++. Besides complex questions related to support for parallelism and for lambdas that arise with the latest C++ standards [11], we are beginning to look at other possible mechanisms for C++, such as attributes.

B. Serial Equivalence

An original goal for OpenMP was to support serial equivalence as much as possible. As a result, many think that all OpenMP programs, or at least all correct OpenMP programs, are guaranteed to produce the same result if the code is executed in parallel as when the compiler completely ignores all OpenMP constructs. However, even OpenMP 1.0 included runtime functions that allow a program to depend on the number of threads or the thread number that executed a region. Thus, trivial programs could fail to exhibit serial equivalence. Today, many more opportunities exist to write OpenMP programs that do not provide serial equivalence.

As OpenMP has evolved, the opportunities to write programs that do not exhibit serial equivalence have increased. Fig. 3 provides a simple tasking program in which the serial version has an infinite loop while the parallel version will complete quickly, assuming that the parallel region uses two or more threads and different threads execute the two tasks. Fig. 4 shows a simple example for accelerators in which "incremented" is always printed, while "incremented again" may or may not print with OpenMP, depending on whether the host and accelerator share memory. Beyond these simple examples, many constructs and clauses are natively unordered. For example since reductions operate in an unknown order using them with floating point types rarely produces serial equivalence.

In general, serial equivalence requires the program or runtime to limit the possible execution orders. As OpenMP has grown to support more parallel programming patterns, the range of execution orders has also grown, which implies more opportunities not to exhibit serial equivalence or would require more execution order limitations, which would limit performance. OpenMP tries to avoid those limitations unless the programmer requires them. Thus, the philosophy of OpenMP remains to provide constructs that can be used to build programs with serial equivalence when desired but often does not guarantee it without additional work.

```
void example2() {
 int a = 0;
#pragma omp target map(tofrom:a)
  {
    a++;
  }

  if (a)
    printf("incremented\n");

#pragma omp target map(to:a)
  {
    a++;
  }

  if (a == 2)
    printf("incremented again\n");
}
```

Fig. 4. *Trivial OpenMP accelerator program.*

C. Descriptive or Prescriptive Semantics

The high performance community is currently debating the value of *descriptive* versus *prescriptive* programming semantics. Semantics are descriptive if programming constructs describe the computation that should be performed but provide the compiler and runtime the flexibility to determine exactly how to perform the computation. Programming constructs with prescriptive semantics prescribe all details of how to perform the required computation.

Our position is that the debate is misguided since it assumes a binary choice between the two types of semantics. However, almost all languages have constructs that are descriptive while others are (more) prescriptive. Specifically within the HPC community, some claim that OpenACC is descriptive while OpenMP is prescriptive [12], [27]. While OpenACC provides more descriptive constructs in its most recent version than OpenMP does, the `acc parallel loop` directive is prescriptive since sometimes users want to *prescribe* that a loop must be parallelized.

Alternatively, most OpenMP defaults allow the compiler freedom to choose details about how the computation is performed. Even the `num_threads` clause of the `parallel` construct, which many believe to be among its most prescriptive mechanisms, allows the compiler and runtime to determine if the number of threads requested is available. If that many threads are not available, the compiler and runtime have the flexibility to determine how many threads to use. So, one may see the issue as where to place a language, or even its constructs, on a continuum of possible semantics.

More importantly, choosing one place on that continuum is overly limited and fails to address the overall preference of programmers. Specifically, they would prefer that the compiler and runtime would always "do the right thing" given a description of the computation to perform. However, in reality, compilers and runtimes often do not. In these instances, programmers prefer to have the ability to override their decisions and to prescribe exactly how to perform the computation.

For these reasons, the emerging OpenMP philosophy is to provide mechanisms that describe the computation to perform and that prescribe as much or as little as the programmer desires about how to perform it. As a first step, OpenMP 5.0 will add the `loop` construct, which only informs the compiler and runtime that a loop nest is easily parallelized. In the longer run, we are exploring mechanisms that specify that the intent of a clause or a construct is fully descriptive or prescriptive.

IV. CONCEPTS AND MECHANICS

OpenMP has expanded greatly in scope and complexity since its inception, but many of its features build on a common set of core mechanics and basic concepts that have changed relatively little over the past 20 years. This section describes two of the most important building blocks of OpenMP, outlining and data environments.

A. Outlining

Compiler outlining is the opposite of inlining. The technique extracts a function from the body of another function. While conceptually simple, outlining forms the basis of the most common implementation of most OpenMP constructs that transform serial code to run in parallel. Specifically it allows the compiler to create the functions required as targets for underlying threading primitives. For example, an implementation may convert a parallel region like that in Fig. 5 into a new function and runtime calls as in Fig. 6.

While our example is simplified, the transformation can outline any block into a function with an appropriate signature for specific parallelization mechanisms and capture any necessary state in a compatible data structure or type. Thus, the user does not need to create wrapper functions and single-use structures to encapsulate their code in order to parallelize it. Instead, the compiler does the repetitive work, while the user determines the appropriate form and granularity of parallelism. This technique allows the

```
void foo() {
  int a;
#pragma omp parallel
  {
#pragma omp master
    a = omp_get_num_threads();
  }
}
```

Fig. 5. *Function that uses OpenMP.*

```
struct foo_parallel_0 {
  int * a;
};

void foo_parallel_(void * data_in) {
  struct foo_parallel_0 * data = data_in;
  data->a[0] = omp_get_num_threads();
}

void foo () {
  int a;
  struct foo_parallel_0 data = {&a};
  runtime_parallel(foo_parallel_, &data);
}
```

Fig. 6. ***After outlining.***

user to specify something for which compiler analysis is highly complex while allowing the compiler to handle the repetitive and error-prone portion of the transformation, which is the most enduring aspect of the philosophy of OpenMP, as stated in Section III.

Interestingly, C++11, with lambdas, and C, with the blocks extension, now provide outlining mechanisms directly to the user. Thus, these languages can cover many original OpenMP features (many of the newer OpenMP features would require additional extensions). We revisit this technique and its relationship to the base languages in Section VII when we discuss some potential directions for the continuing evolution of OpenMP.

B. Data Environments

While outlining supports parallelization, it does not directly address the issue of data sharing between threads or tasks. In OpenMP, every task, including implicit tasks that a loop or device construct creates, has its own data environment that represents its view of memory and of state in the OpenMP runtime. The simplest manifestations of a data environment provide variables that are private to the task, thread, team or construct in general without having to refactor variable declarations and initializations in user code.

OpenMP data environments also include ICVs (Internal Control Variables), which are a less familiar but equally important aspect of them. ICVs govern the actions of the OpenMP runtime. Most users know some of the major ICVs by their associated environment variables, such as `OMP_NUM_THREADS`, and the behavior of environment variables and data environments are similar. Each new data environment inherits some values and their behaviors from the enclosing data environment but is otherwise independent of that enclosing environment. Thus, each task can control the behavior of OpenMP in its dynamic scope without changing the behavior of OpenMP constructs outside of that scope. Thus, the mechanism supports composability and control.

Overall, OpenMP data environments are an essential concept that has evolved with OpenMP. For example, we added the concept of separate data environments for each device along with the device constructs. This concept provides a richer memory environment than the original OpenMP shared memory environment. Specifically, distinct device data environments can have copies of the same variable that may share storage–or may not. Thus, OpenMP provides mechanisms to keep the potential copies consistent.

V. RECENT OPENMP EXTENSIONS

OpenMP 4.0 extends the API to cover two additional major forms of parallelism: accelerator offload and SIMD vectorization. Almost all current systems include hardware that require these parallel programming patterns. This section discusses the related extensions as well as several tasking extensions in OpenMP 4.0 and 4.5.

A. SIMD

Compilers have included technology to auto-vectorize loops for many years. However, this support has limited effectiveness for real applications because of the complexity of determining the potential correctness and benefit of vectorization (e.g., are loop iterations free of dependences). These limitations led almost all major compilers to include implementation-defined vectorization directives. While frequently spelled `ivdep`, the semantics often subtly varied across compilers. Due to the similarity with the original motivation for OpenMP with respect to threading directives, we included explicit directives to exploit SIMD parallelism in OpenMP 4.0.

The `simd` directive expresses that a given loop nest has no dependences that would prevent vectorization. The compiler can then vectorize the loop without performing any dependence analysis. The directive accepts several clauses that provide further information and/or restrictions to guide vectorization. The `simd` directive is not prescriptive as the compiler may choose not to vectorize the loop (essentially a vector width of one).

Loops with functions pose a particular problem to vectorization. If the compiler has the function definition available then it could inline it to vectorize the loop fully. In practice, the definition is often in a different compilation unit. Without special treatment, the compiler can still partially vectorize the loops by repeatedly calling the scalar function for each element of the vector. A more efficient solution generates vector variants of the functions that process multiple elements of the vector in a single invocation. The compiler can then use these variants in loops annotated with the `simd` directive.

OpenMP provides the `declare simd` directive to guide generation of vector function variants. The directive accepts several clauses that prescribe generation of efficient variants for specific use cases so a function may be annotated with multiple `declare simd` directives.

```
#pragma omp declare simd uniform(c)
double scale(double v, double c) {
     return v * c;
}

void example(double * v, size_t n) {
    double alpha = 0.5;
#pragma omp simd
    for (size_t i = 0; i < n; i++ )
        v[i] = scale(v[i], alpha);
}
```

Fig. 7. *OpenMP SIMD vectorization example.*

Other clauses generally guide generation of vector variants (e.g., the `uniform` clause indicates that a given argument should be a scalar and not a vector). The compiler can also generate other variants that may be useful for a specific target architecture. The simple example in Fig. 7 uses the OpenMP SIMD directives.

B. Devices

In addition to the pervasiveness of vector units in modern processors, many systems now include additional coprocessors or computational accelerators. These devices include hardware such as graphics processing units (GPUs), digital signal processors (DSPs), and computation offload coprocessors like the Intel Xeon Phi coprocessor. While these hardware devices usually reside in a single node, they pose a particular challenge for OpenMP because they frequently use a different instruction set and programming paradigm. Further, they often do not coherently share memory with the host processors that OpenMP originally targeted.

OpenMP 4.0 added the `target` directive and related directives and routines to address these devices. These additions provide an offload model that uses the existing shared-memory model on each device. Since many accelerators are many-core devices, we added the `teams` and `distribute` directives, which create leagues of independent thread teams and share loop iterations among them. Accelerators can execute these teams efficiently since synchronization across them is highly restricted while all OpenMP functionality (except the device constructs) may be used within each team. The code in Fig. 8 offloads a simple loop to the default device and divides its work across teams of threads. The `map` clauses map data into the device data environment and, if desired, update the view of the data on the device (host) before (after) execution of the `target` region.

In addition to the `map` clause on the `target` directive, OpenMP provides several other options for device data management. These options include directives for the definition of structured target data regions and also for unstructured transfers or updates between host and device data. The `nowait` clause can be used on the `target` directive and on these device data management directives to enable the implementation to treat them as asynchronous tasks. This feature allows overlap of host and device computation and data transfers. It can also be combined with task dependences, described in Section V-C, for data-driven asynchronous execution.

Similarly to `simd` regions, `target` regions that contain function calls are particularly challenging to support. Unlike with `simd` regions, however, if the function definition is not available to the compiler then the compiler may not generate any variant that can be executed, even inefficiently, on the device. Thus, in OpenMP 4.0 and 4.5, if any target region calls a function then the user must annotate the function definition and its declarations with the `declare target` directive. The directive can also be applied to global variables. The compiler then must generate a variant of a function or a static lifetime variable for the target device.

C. Tasking Extensions

OpenMP 3.0 introduced directives to support asynchronous task parallelism. Those extensions were carefully designed to support that unstructured parallel pattern while coexisting with OpenMP's existing support for structured parallelism [1]. They generate tasks with the `task` construct and synchronize them through the `taskwait` construct and barriers. The `taskwait` construct specifies a wait on the completion of child tasks of the current task, and a barrier requires complete execution of all tasks in the current parallel region before any threads in the team can continue execution beyond the barrier. However, these simple synchronization mechanisms often lack the expressiveness to expose all available parallelism. OpenMP 4.0 addressed these limitations with two additional synchronization mechanisms: task dependences and task groups.

The `depend` clause in OpenMP 4.0 uses variable names to indicate dependences between tasks (i.e., restrictions on their execution order). Fig. 9(a) and (b) shows task code for a producer-consumer pattern in OpenMP 3.0 and 4.0. The time lines below it illustrate the scheduling of the tasks on two threads. Task dependences support fine-grained, data-driven synchronization, as Fig. 9(d) shows, which allows more flexible scheduling compared to the coarse-grained synchronization that OpenMP 3.0 supported [Fig. 9(c)]. Fig. 10 compares the parallel

```
#pragma omp target teams distribute \
            parallel for            \
            map(to: in_arr[0:n])    \
            map(from: out_arr[0:n])
for(size_t i = 0; i < n; ++i)
  out_arr[i] = in_arr[i] * in_arr[i];
```

Fig. 8. *OpenMP device offload example.*

```
#pragma omp task
produce(a);
#pragma omp task
produce(b);
#pragma omp task
produce(c);


// wait on all
// children here
#pragma omp    \
        taskwait

#pragma omp task
consume(a, b);
#pragma omp task
consume(b, c);
#pragma omp task
consume(a, c);
```

(a) OpenMP 3.0

```
#pragma omp task \
    depend(out: a)
produce(a);
#pragma omp task \
    depend(out: b)
produce(b);
#pragma omp task \
    depend(out: c)
produce(c);

#pragma omp task \
   depend(in: a, b)
consume(a, b);
#pragma omp task \
   depend(in: b, c)
consume(b, c);
#pragma omp task \
   depend(in: a, c)
consume(a, c);
```

(b) OpenMP 4.0

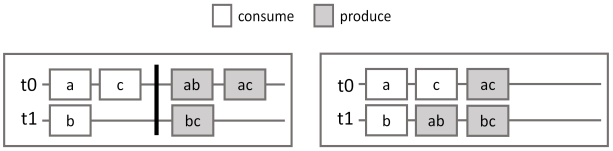

Fig. 9. ***Tasking examples without and with dependences.***

speedup achieved on a 48-core system. A basic task-based implementation of Cholesky edges out a highly optimized version using the loop construct, and using dependences improves performance more significantly. For Gauss-Seidel, a basic task-based implementation performs worse than a version based on the loop construct, but a version that uses task dependences provides the best performance.

As stated previously, the `taskwait` construct requires that all child tasks of the current task must complete. The `taskgroup` construct allows the current task to wait on

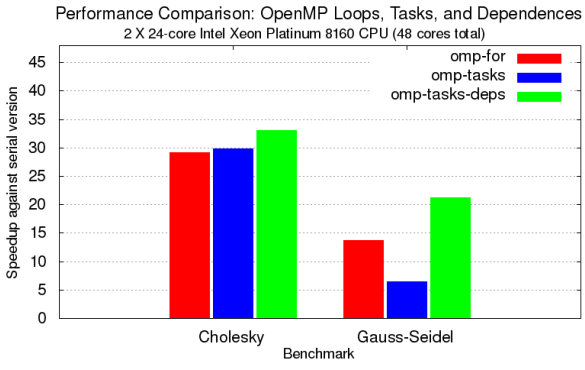

Fig. 10. ***Performance benefit of dependence support.***

```
void sapxy_tasks(float * a, float * b,
                 float s, size_t n) {
#pragma omp taskloop simd                \
            num_tasks(NTASKS)            \
            shared(a,b) firstprivate(s)
  for (size_t i = 0; i < n; i++) {
    a[i] = a[i] * b[i] * s;
  }
}
```

Fig. 11. ***Task loop example.***

only a subset of its children, while others may continue executing beyond the synchronization point. Also, the construct requires that all descendant tasks of that subset complete execution, which we call *deep synchronization*. Because some children of the current task can be excluded from a task group, those tasks can perform long-running background activities that proceed alongside successive computational kernels.

With OpenMP 3.0 tasking support, a user could manually decompose a loop into chunks that OpenMP tasks execute. This cumbersome and error-prone manual transformation is inconsistent with the philosophy of OpenMP. Thus, OpenMP 4.5 added the `taskloop` construct to automate it. Fig. 11 uses the construct to parallelize a *saxpy* operation. The `num_tasks` clause specifies the number of tasks to create for the loop. Alternatively, users specify the minimum number of loop iterations per task with the `grainsize` clause. OpenMP 4.5 also includes a combined `taskloop simd` construct to use SIMD parallelism in the generated tasks.

D. Cancellation

OpenMP 4.0 introduced cancellation, which ends an OpenMP region *early* to enable efficient error handling and more efficient algorithms. When a thread encounters a `cancel` construct, it cancels execution of the innermost associated region (as indicated by a `parallel`, `sections`, `for` or `do` clause) or associated set of tasks (as indicated by the `taskgroup` clause).

Cancellation must occur with well-defined semantics so users can ensure that their data is in an expected state. Since the user can manage the state immediately before the `cancel` construct, the thread that encounters it immediately proceeds to the end of the canceled region (e.g., the end of the current task for the `taskgroup` clause). Other threads must encounter a *cancellation point*, prior to which the user can manage state, in order to process the cancellation. Cancellation points are implied at barriers and are explicitly indicated by `cancellation point` and `cancel` constructs. If a thread observes that another thread has canceled the associated region at a cancellation point, it also proceeds to the end of the canceled region (e.g., the end of the current task). With the `taskgroup`

```
bin_tree_t *
search_tree(bin_tree_t * tree, int val) {
  bin_tree_t * found = NULL;
  if (tree) {
    if (tree->val == val)
      found = tree;
    else {
#pragma omp task shared(found)
      {
        bin_tree_t * found_left =
          search_tree(tree->left, val);
        if (found_left) {
#pragma omp atomic write
          found = found_left;
#pragma omp cancel taskgroup
        }
      }
#pragma omp task shared(found)
      {
        // similar code for right branch
      }
#pragma omp taskwait
    }
  }
  return found;
}
```

Fig. 12. ***Cancellation example.***

clause, tasks that have not begun to execute are simply discarded since they cannot have state from partial execution.

Fig. 12 shows how to cancel a binary tree search when the value is found. Without the OpenMP directives, the code recursively examines children nodes and stops if the value of the current tree node matches the search value. With OpenMP tasking, the subtree searches execute in parallel. Without cancellation, once a task finds the search value, it does not generate any more tasks but the other branches of the parallel search continue. With cancellation, any executing tasks complete their check but any generated tasks that have not begun execution (including those generated by the executing tasks) are discarded so that unnecessary work is greatly reduced while still executing the search in parallel.

VI. NEXT EVOLUTIONARY STEP

We will release OpenMP 5.0 in November 2018. We have already made substantial progress on its content, as TR6 [23] demonstrates. Based on TR6, OpenMP 5.0 will increase the page count of the specification more than any prior version. However, most new pages will detail additions to OpenMP that support performance analysis and debugging tools that we do not discuss further. Nonetheless, OpenMP 5.0 will also include several extensions to the user-level API that significantly enhance its support for a wide range of architectures. We now discuss many of those extensions.

A. Device Extensions

While OpenMP introduced support to offload computation regions to target devices in version 4.0 and subsequently expanded that support significantly in 4.5, the space is changing quickly. Thus, we have already adopted several extensions and refinements for OpenMP 5.0 including changes that greatly simplify the use of functions in those regions. Further, a new general mechanism to specify application-specific requirements will enable straightforward use of unified memory spaces across devices. Nonetheless, we have also adopted a unique deep-copy mechanism that will significantly improve usability on systems that do not provide unified memory spaces. Importantly, we expect this deep-copy support will often provide performance advantages even on systems that do provide them.

Many offload models, such as CUDA and OpenCL, require function annotations. However, OpenMP 5.0 will ease the use of functions on devices by relaxing its annotation requirements. OpenMP 5.0 will eliminate the requirement to annotate function declarations. Essentially, the compiler must assume that a device variant will be available at link time. Also, the compiler must automatically generate a device variant for any function with a definition in the same translation unit as a call from a `target` region. Essentially, the definition implicitly includes the `declare target` annotation. Because these changes significantly improve usability, many compilers have already implemented them and they have allowed entire large codebases (particularly in C++ due to the pervasiveness of templates) to offload to devices using OpenMP without a single explicit `declare target` directive; other models require hundreds or thousands of annotations to compile them at all.

In order to assume coherent memory between the host and a target device, the user must assert to the compiler that their code requires that support. Given this assertion, if the code is run on a device without that support, it may exhibit unspecified behavior (i.e., the code is broken). Overall, these assertions are a contract between the application and the compiler, which is a general mechanism for which unified memory spaces are just one instance. Thus, OpenMP 5.0 will provide a new `requires` directive that allows OpenMP to specify a set of rules for a given requirement and users to specify that their code conforms to those rules. This directive supports the definition of subsets of the OpenMP specification; one 5.0 subset will support systems that do not require memory to be mapped explicitly into a data environment for target devices. Effectively, the user can assume shared memory between the host and the devices. For example, the code in Fig. 13 is only valid for systems with a unified view of memory. It is nonconforming in OpenMP up to 4.5 but will be correct on systems that meet the requirement. Importantly, the `requires` directive applies to an entire translation unit, which offers usability benefits similar to the implicit `declare target` annotations.

```
#pragma omp requires          \
        unified_shared_memory
struct list {
  void * data;
  struct list * next;
};

void foo() {
  struct list * l = make_linked_list();
#pragma omp target
  {
    struct list * cur;
    while(cur) {
      do_something_with_data(cur->data);
      cur = cur->next;
    }
  }
}
```

Fig. 13. ***Requiring unified memory.***

```
class Vec {
  size_t len;
  double * data;
public:
  // Normal vector methods
#pragma omp declare mapper(Vec v) \
    use_by_default                \
    map(v, v.data[0:v.len])
};

void foo() {
  Vec v1(100), v2(100);
  fill_vec(v1);
#pragma omp target teams distribute \
      parallel for                  \
      map(to:v1)   // explicitly map v1
  for(auto i = 0; i < v1.size(); ++i) {
    v2[i] = v1[i]; // implicitly map v2
  }
  // v2[0-100] == v1[0-100]
}
```

Fig. 14. ***User-defined mapper example.***

The deep-copy support in OpenMP 5.0 will simplify the use of pointer-based data structures like the linked list in Fig. 13 on systems that do not provide coherent unified memory. With OpenMP 4.5, the user must map each piece of the structure and must then assign the pointers on the device to those pieces either with explicit assignments or with further mapping actions. The user often must repeat this verbose, complex and error-prone code sequence every time an instance of the data structure is needed on the device. Instead, the `declare mapper` directive in OpenMP 5.0 will allow the user to describe how to map an instance of the data structure including the targets of pointers. The user can then use this definition in a `map` clause whenever an instance of the data structure is needed on the device. Overall, the descriptions in the `declare mapper` directive are simpler than the OpenMP 4.5 mechanism and eliminate the repetition. Fig. 14 shows an example that maps a multi-level data structure with the `declare mapper` directive. The directive in the `Vec` class uses a `map` clause to describe how to map the data that is the target of the pointer member for any instance of the class. This version works for any target platform, including those that do not support unified memory.

We plan to refine the deep-copy mechanism further. Specifically, we will provide a mechanism that can replace any phase of the mapping process with user-defined expressions or functions written in the base language. This mechanism, which will provide equivalent functionality to data serialization and deserialization for transmission over a network, will support mapping of arbitrary, complex data structures. Further, it will enable data-dependent data transformations that support highly efficient kernel computations. We expect OpenMP 5.1 to include this functionality.

```
void func(double * a, size_t n) {
#pragma omp task \
    depend(iterator(i=0:n):inout:a[i])
   work(a);
}
```

Fig. 15. ***Iterated task dependences.***

B. Iterators

Many OpenMP clauses accept lists of parameters. In OpenMP 4.5 or earlier, while many OpenMP clauses accept expressions, the expressions (but not their values) must be fully determined at compile time. Thus, the number of elements in each list is static and, for example, the `depend` clause can specify a dependence on multiple elements of an array but the number of elements (or array sections) must be known at compile time. This requirement can prevent the expression of some algorithms or make their expression more complex. For example, if a corner cell has fewer dependences than an inner cell then the user may need to modify the base language code to provide separate annotations for each case. Further, the limitation can require the use of long error-prone lists even when the number of list elements is static. This limitation arises from the lack of general programming constructs in OpenMP directives, which we plan to reduce as discussed in Section III-A.

To overcome this lack of expressiveness, OpenMP will add the concept of iterators. This mechanism can iterate through a range of values to produce list-items at runtime. Thus, a clause can have a dynamic number of list elements. Fig. 15 shows how this feature supports a `task` construct with a variable number of dependences.

C. Further Evolution of Tasking Support

OpenMP 5.0 continues to evolve the tasking model to address use cases. Task reductions, task affinity, and additional forms of task dependences enhance performance and ease of use. Prior to OpenMP 5.0, lack of support for explicit task reductions required users to implement their own reductions by collecting and later combining per-thread partial values, passing partial values through the tree of tasks, or using locks or atomics that serialize those operations. The `task_reduction` clause allows a reduction over a task group, and the `reduction` clause is available on task loops. The `in_reduction` clause appears on tasks that participate in the reduction, which can include target tasks that offload computation or transfer data to devices.

Support for task dependences is extended in two new ways. First, use of iterators is allowed in the `depend` clause, as described previously in Section VI-B. Second, a new dependence type allows a set of tasks to commute with respect to one another with the constraints that their executions are mutually exclusive and that they satisfy any dependences with respect to tasks outside the set.

Like task dependences, task affinity indicates the data used by a task. However, task affinity is a hint that guides the scheduling of tasks to threads, rather than enforcing an ordering among the threads. Tasks that use the same data can be scheduled to the same thread or to threads that execute on cores in the same NUMA domain. An advanced runtime may also use the information to tune work stealing for better locality. Future versions of OpenMP may apply the `affinity` clause to other constructs besides the `task` construct.

D. Memory Allocation

Memory hierarchies will become deeper in future systems with the use of technologies such as high-bandwidth memory and nonvolatile RAM. Each of these technologies has a different programming interface and distinct performance characteristics. Programming mechanisms must address these differences and support intelligent data placement since the fastest resources typically have limited capacity. To enable programmability of these technologies and portability across platforms, OpenMP 5.0 will include a consistent and portable interface for placement within the memory hierarchy.

A *memory space* is a memory resource that is available in the system. Memory spaces differ in their characteristics, for instance in bandwidth or capacity. OpenMP will define intuitive predefined memory spaces that map to memory resources in HPC systems. An *allocator* object allocates and frees memory from the resources of the memory space to which it is associated when it is created. OpenMP 5.0 will provide predefined memory allocators that match its predefined memory spaces. For example, the predefined memory allocators can select a memory space with large capacity, high bandwidth or low latency, or local to a particular thread or thread team.

```
double * A = (double * ) omp_alloc(N,
                 omp_high_bw_mem_alloc);
```

Fig. 16. *High-bandwidth memory allocation.*

OpenMP 5.0 will include the `omp_alloc` and `omp_free` routines as supersets for `malloc` and `free`. The `allocate` directive can specify allocation properties of variables that are not allocated through an API call such as global or stack variables. The `allocate` clause will directly specify the use of an allocator for any construct that accepts data sharing clauses. It enables the allocation of `private` variables in a particular memory space. Fig. 16 illustrates the use of the predefined `omp_high_bw_mem_alloc` allocator to allocate memory from the high bandwidth memory space.

In order to support rapid adaptation of existing programs to a specific memory configuration, the predefined allocators have type `omp_allocator_t *` and can be used as regular pointers. Thus, they can be passed by argument and once memory allocation uses the OpenMP API function, these code places do not have to be modified again just to use a different memory space; the allocator passed to the function only needs to be adjusted. Fig. 17 illustrates how to select the memory policy that a function used to allocate the private array *some_array*.

Besides predefined allocators, OpenMP 5.0 will support creation of custom memory allocators through which the user can specify additional traits. Current traits can specify the desired memory alignment, the maximum pool size, the fallback behavior when failing to allocate memory and some hints that specify the context in which the memory is expected to be used or the expected contention on the allocator. Fig. 18 shows an example that creates a custom allocator. This allocator returns memory from the *default memory space* with 64-byte alignment that only the thread that allocates the memory can access. This allocator can then be used in the previously presented API calls, directives and clauses.

```
void some_function (omp_allocator_t * allocator) {
  double some_array[N];
  #pragma omp parallel private(some_array)        \
                       allocate(allocator:some_array)
  {
     ...
  }
}

some_function(omp_high_bw_mem_alloc);
some_function(omp_default_mem_alloc);
```

Fig. 17. *Separate memory selection and allocation.*

```
omp_alloctraits_t *traits[]=
                {{OMP_ATK_ALIGNMENT,64},
                 {OMP_ATK_ACCESS,OMP_ATV_THREAD}};
omp_allocator_t *allocator =
          omp_init_allocator(omp_default_mem_space,
                             2,traits);
```

Fig. 18. ***Custom memory allocator.***

VII. LONGER TERM DIRECTIONS

While OpenMP 5.0 is clearly a major step in the evolution of OpenMP, we already know that we will not address every issue that remains. We plan to release a minor revision of 5.0 in November 2020 (nominally, it will be OpenMP 5.1) that we again do not expect to address every open issue. We have established a five-year cadence of major releases of the OpenMP specification, which we plan to continue. In this section, we anticipate the long term evolution of OpenMP; we may realize some of these directions in OpenMP 5.1 but we will defer many of them to OpenMP 6.0 or later.

A. Pipelining of Target Data Transfers

Data transfers between host and device memory is a common bottleneck for heterogeneous applications. A basic optimization overlaps computation with those data transfers. Further, devices often have limited memory capacities, which leads to optimizations that divide the computation into pieces and stage in the data of upcoming pieces and stage out the data of preceding ones while the current piece is executed. While this *pipelining* of data transfer and computation is well understood, manual transformations to implement it involve many complex and error-prone source code changes.

```
void func(double * A, double * An,
          int nx, int ny, int nz) {
#pragma omp target teams distribute     \
  pipeline(static)                      \
  map(pipeline,to:A[k-1:3][0:ny][0:nx]) \
  map(pipeline,from:An[k:1][0:ny][0:nx])
  for(k=1;k<nz;k++) {
    #pragma omp parallel for
    for(i=1;i<nx;i++) {
      for(j=1;j<ny;j++) {
        An[Index3D (i, j, k)] =
         (A[Index3D (i, j, k + 1)] +
          A[Index3D (i, j, k - 1)] +
          A[Index3D (i, j + 1, k)] +
          A[Index3D (i, j - 1, k)] +
          A[Index3D (i + 1, j, k)] +
          A[Index3D (i - 1, j, k)])* c1
        - A[Index3D (i, j, k)] * c0;
      }
    }
  }
}
```

Fig. 19. ***Pipelining example.***

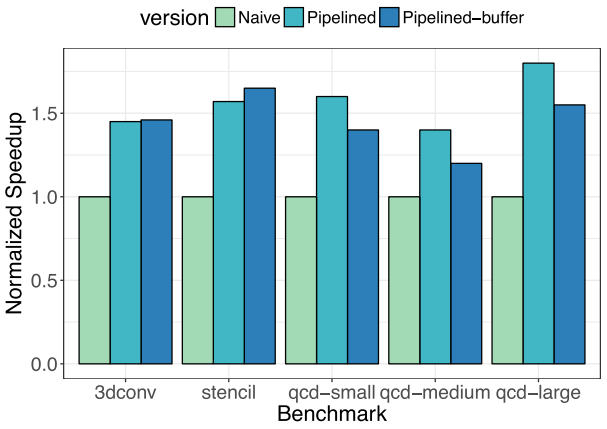

Fig. 20. ***Pipelined versus buffered data transfers.***

We are developing interfaces that associate data transfers with loop iteration spaces and, thus, support automated pipelining. Fig. 19 shows an example that pipelines a stencil computation. The `map` clauses use the distributed loop's iteration variable to indicate that the compiler can divide the arrays along their first dimension and that three elements along that dimension of the input array are required to compute each element along that dimension of the output array. Thus, the compiler can transform the loop to perform chunks of the computation while pipelining the data.

Fig. 20 compares a prototype of this interface to a naive version that does not pipeline the loop [2]. We present two pipelining strategies. Pipelined uses a buffer of the same size and layout as the naive version so it does not save memory space but splits the computation to overlap transfers. Pipelined-buffer uses smaller buffers and transforms the accesses in the loop to decrease the memory capacity that is required. In some cases, particularly the 3dconv and stencil kernels, the buffered version's greater locality actually improves performance. For the quantum chromodynamics kernel however it loses about 20% performance compared to using the full amount of memory, but allows much larger problems to be run than otherwise fit on the device.

B. Memory Affinity

While OpenMP 5.0 will specify task affinity based on memory locations as discussed in Section VI-C, a longer term goal is to support more general memory affinity. Intuitive interfaces for this complex problem are difficult to specify. Nonetheless, we have explored interfaces that associate data to computation and then appropriately locate, transform or replicate the data based on the distribution of the computation of existing mechanisms in OpenMP [24], [25].

Fig. 21 shows a more recent direction that specifies how to partition computation and to map the associated data range to the threads of a `parallel` region and then to a set of devices. This example partitions the GEMM loop into 2-D tiles by columns across sockets and rows across devices

```
int i, is=0, ie=isz, j, js=0, je=jsz;
float A[ksz][jsz], B[isz][ksz];
float * C = (float*)malloc(sizeof(C[0])*isz*jsz);
int   C_pitch = jsz;

#pragma omp parallel devices(socket)                    \
 part(adaptive: j_id=js; j_id<je; ++j_id)               \
 map(to: A[0:ksz][:,part=j_id], B[0:isz][0:ksz])        \
 map(tofrom: C[0:isz,pitch=C_pitch][:,part=j_id])
{ // Partitioned parallel region
#pragma omp target teams distribute parallel for        \
  devices(all) partition(dynamic)                       \
  map(to: A[:][:], B[:,partition=i][:])                 \
  map(tofrom: C[:,partition=i][:])
  for (int i = is; i < ie; ++i) {//Partitioned loop
    for (int j = js; j < je; ++j) {
      float sum = 0.0;
      for (int k = 0; k < ksz; ++k) {
        sum += A[k][j] * B[i][k];
      }
      C[i * C_pitch + j] = sum;
    }
  }
}
```

Fig. 21. ***Possible memory affinity interface.***

associated with a given socket. Extensions like this require careful consideration due to their potentially large number of changes, and high complexity, but the information can support significant optimizations. Beyond providing affinity information, these annotations are sufficient to allow for cross-device coscheduling across nonshared-memory devices.

Fig. 22 shows performance results from a prototype implementation across five benchmark kernels in terms of speedup over a baseline OpenMP static schedule that uses all cores. The annotations and scheduling improvements that the information enables can increase performance substantially. The optimization space being explored in the figure compares static scheduling to an adaptive scheduler that attempts to predict the best partitioning based on past performance. The CPU adaptive results represent using the adaptive scheduler on the same resources as the baseline. The results also vary the devices across which the runtime system can distribute computation and data. It can use only the CPU cores, only a set of one to four NVIDIA c1060 GPUs, or both. The same GEMM code can target all of these options by changing runtime parameters.

The amount of expressive power this kind of extension can provide is significant, but so is the complexity and the burden on the programmer who is trying to use it. We intend to continue exploring this space in the future to provide an appropriate long-term solution.

C. Memory Allocation Extensions

While OpenMP 5.0 will introduce the major building blocks for memory allocation support (memory spaces, allocators and APIs) we will extend this support. One such direction will be to allow users to determine the memory resources of a particular system and to create memory spaces with a more precise description and not just to rely on predefined ones. The number of allocator traits will increase to allow users to specify a larger range of behaviors. For example, we envision traits that support: memory pinning; distribution of allocations across NUMA domains; fixed-object allocations (i.e., where all allocation invocations have the same size); and allocators with stack semantics. We will also eventually improve support of allocation of variables specified in `map` clauses, as well as C++ variables generally.

D. Free-Agent Threads

Currently, only threads of the parallel region in which an explicit OpenMP task is generated can execute that task. This limitation leads to the unintuitive (if simple) requirement that pure tasking programs in OpenMP must first start a `parallel` region and then must ensure that only one thread executes the code that generates the tasks, for example, by using a `single` region. This limitation can restrict parallelism in more complex applications since other threads (resources) may be idle and available to execute the tasks.

We are exploring a concept of free-agent threads to overcome this limitation. The mechanism would allow any thread that is not assigned to a team to execute any explicit task. It would fully eliminate the limitation; all threads could execute explicit tasks that are generated in the initial thread without requiring an explicit `parallel` region. We need to resolve many details, such as the return values for runtime routines such as `omp_get_thread_num` when executed by a thread that is not part of the team. Since this change will represent a major change in the OpenMP execution model, we do not expect to adopt it before OpenMP 6.0.

E. Event-Based Parallel Programming Pattern

One parallel programming pattern that OpenMP does not yet support is the event-driven model that many interactive applications and networking servers use. In this model, one or more threads run continuously in an event loop to observe external (e.g., user) actions. Other threads then perform the computation that the actions trigger to minimize response times. This event-based pattern naturally suits a task-based model.

OpenMP's current task model does not suit the event-based pattern since it requires the team of the thread that generates a task to execute that task. To support this pattern, OpenMP needs a new capability to allow a thread to direct work toward a team other than its own. This capability would allow the event thread to remain responsive as other teams concurrently handle event processing. In addition, a mechanism that creates reusable tasks could further improve response times.

F. Enabling Language-Level Outlining

As Section IV-A discussed, outlining, or extraction of code into functions by the compiler, is a core mechanism used to implement OpenMP. Some base languages provide

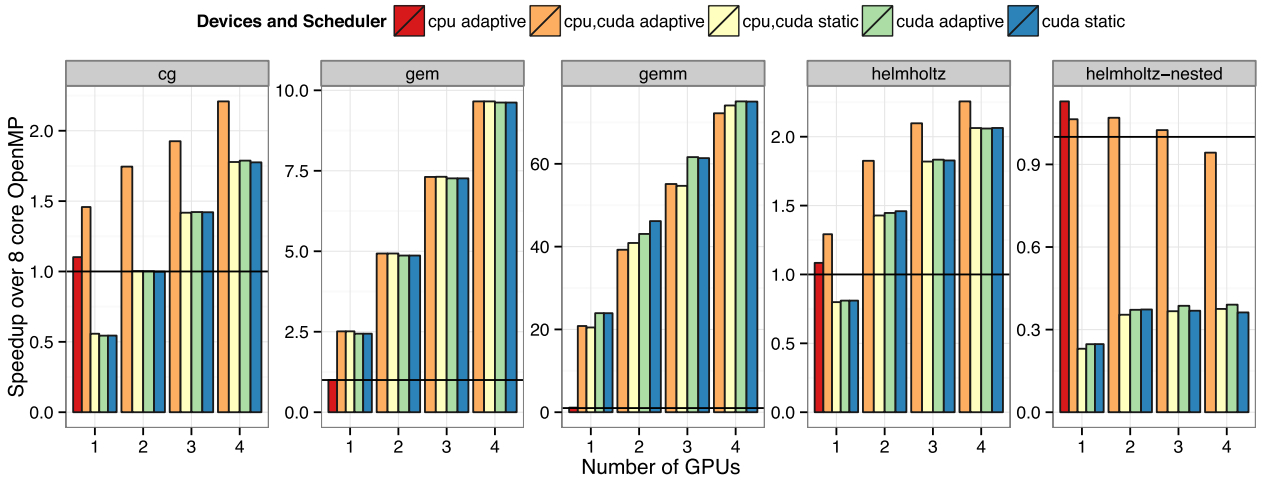

Fig. 22. ***Performance benefit of memory partitioning/affinity.***

outlining mechanisms in the form of closures or lambdas. The writers of libraries and parallel frameworks find these constructs attractive since they can describe abstract patterns and behaviors that then are passed an arbitrarily complex code sequence and associated data. Frameworks like Kokkos [3] and RAJA [4] exploit this mechanism to create flexible looping constructs, like the one in Fig. 23, that can be compiled for host devices, targets or other parallel backends, depending on compile time arguments. These mechanisms pose challenges that OpenMP 5.0 will begin to address.

While OpenMP must evolve to support mechanisms such as lambdas, Fortran users of OpenMP currently cannot exploit the capabilities that we will provide. Although many OpenMP implementations use outlining, they do not expose the resulting functions to the user. However, exposing them could provide many benefits, including a mechanism to support closures in Fortran.

We could extend the `task` directive to create a form of "callable task" or OpenMP closure object that would be portable across C, C++ and Fortran. The extension would significantly reduce the work required to make an arbitrary callable object with state in C and Fortran. It would also support library implementations with functionality like that of Kokkos and RAJA that all three languages could use. Challenges remain, however, such as to integrate the functionality with existing OpenMP constructs and how to make it as efficient as possible at runtime. A simple and portable solution generates a structure, or derived type, and a function pointer. This solution easily integrates with established libraries, but will likely perform poorly for frequently called functions. Despite the challenges, giving users control of outlining could be a major step forward for OpenMP.

```
RAJA::forall<omp_parallel_for>(
    RAJA::range(0, n),
    [=](int a) {
      // loop body
});
```

Fig. 23. ***RAJA loop body example.***

VIII. CONCLUSION

Over 20 years have passed since we released the first OpenMP specification. It has become a mature programming API that continues to support Fortran, C, and C++ as base languages. In its maturation, the size of the API and its specification has grown substantially as we added support for additional parallel programming patterns. Its underlying philosophy has also evolved although we retain many of its core principles. Most of all, the primary purpose of the API continues to be to allow users to specify information about their computation that they easily know but that would require complex compiler analysis to deduce while relying on the compiler to implement repetitive, tedious and error-prone mechanisms that exploit that information in a way that can be carried from compiler to compiler. As of this writing, the OpenMP compilers page [14] lists 16 compilers, nine of which support at least a significant portion of OpenMP 4.5.

In this paper, we discussed the $7.5\times$ increase in the size of the OpenMP specification over the course of its lifetime. We provided a glimpse into the evolution of its guiding principles as well as some of the features that the most recent versions added. We also discussed some of the key programming features that OpenMP 5.0 will add and that are under consideration for versions beyond it. These plans will result in a specification that supports essentially every major parallel programming pattern and the latest base language standards. ■

Acknowledgments

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.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information see http://www.intel.com/performance.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information on the specific instruction sets covered by this notice.

REFERENCES

[1] E. Ayguade, "The design of OpenMP tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, Mar. 2009.

[2] X. Cui, T. R. W. Scogland, B. R. de Supinski, and W.-C. Feng, "Directive-based partitioning and pipelining for graphics processing units," in *Proc. Int. Parallel Distrib. Process. Symp.*, May 2017, pp. 575–584.

[3] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014.

[4] R. Hornung and J. Keasler, "The RAJA portability layer: Overview and status," Lawrence Livermore Nat. Lab., Livermore, CA, USA, Tech. Rep. LLNL-TR-661403, 2014.

[5] Information Technology—Programming Languages—Fortran—Part 1: Base Language, ISO International Standard ISO/IEC 1539-1:2004, ISO/IEC, 2004. [Online]. Available: https://www.iso.org/standard/39691.html

[6] Information Technology—Programming Languages, ISO International Standard ISO/IEC 9899:1999/Cor 3:2007, ISO/IEC, 2007. [Online]. Available: https://www.iso.org/standard/50510.html

[7] Information Technology—Programming Languages—Fortran—Part 1: Base Language, ISO International Standard ISO/IEC 1539-1:2010, ISO/IEC, 2010. [Online]. Available: https://www.iso org/standard/50459.html

[8] Information Technology—Programming Languages—C++, ISO International Standard ISO/IEC 14882:2011, ISO/IEC, 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?%csnumber=50372

[9] Information Technology—Programming Languages—C, ISO International Standard ISO/IEC 9899:2011, ISO/IEC, 2011. [Online]. Available: https://www.iso.org/standard/57853.html

[10] Information Technology—Programming Langugages—C++, ISO International Standard ISO/IEC 14882:2014, ISO/IEC, 2014. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029

[11] Information Technology—Programming Languages—C++, ISO International Standard ISO/IEC 14882:2014, ISO/IEC, 2014. [Online]. Available: https://www.iso.org/standard/68564.html

[12] G. Juckeland, "From describing to prescribing parallelism: Translating the SPEC ACCEL OpenACC suite to OpenMP target directives," in *Proc. Int. Supercomput. Conf.*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds. Cham, Switzerland: Springer, 2016, pp. 470–488.

[13] T. Mattson, "The OpenMP architecture review board and the future of OpenMP," in *Proc. 1st Eur. Workshop OpenMP*, 1999. [Online]. Available: http://www.it.lth.se/ewomp99

[14] OpenMP ARB. (Oct.) *OpenMP Compilers and Tools*. [Online]. Available: https://www.openmp.org/resources/openmp-compilers-tools/

[15] OpenMP ARB. (Oct. 1997). *OpenMP Fortran Application Program Interface*. [Online]. Available: http://www.openmp.org/wp-content/uploads/fspec10.pdf

[16] OpenMP ARB. (Oct. 1998). *OpenMP C and C++ Application Program Interface Version 1.0*. [Online]. Available: http://www.openmp.org/wp-content/uploads/cspec10.pdf

[17] OpenMP ARB. (Mar. 2002). *OpenMP C and C++ Application Program Interface Version 2.0*. [Online]. Available: http://www.openmp.org/wp-content/uploads/cspec20.pdf

[18] OpenMP ARB. (May 2005). *OpenMP Application Programming Interface Version 2.5*. [Online]. Available: http://www.openmp.org/wp-content/uploads/spec25.pdf

[19] OpenMP ARB. (May 2008). *OpenMP Application Programming Interface Version 3.0*. [Online]. Available: http://www.openmp.org/wp-content/uploads/spec30.pdf

[20] OpenMP ARB. (Jul. 2011). *OpenMP Application Programming Interface Version 3.1*. [Online]. Available: http://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf

[21] OpenMP ARB. (Jun. 2013). *OpenMP Application Programming Interface Version 4.0*. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[22] OpenMP ARB. (Nov. 2015). *OpenMP Application Programming Interface Version 4.5*. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[23] OpenMP ARB. (Nov. 2017). *OpenMP Technical Report 6: Version 5.0 Preview 2*. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf

[24] T. R. W. Scogland, W.-C. Feng, B. Rountree, and B. R. de Supinski, "CoreTSAR: Core task-size adapting runtime," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 2970–2983, Nov. 2015.

[25] T. R. W. Scogland, B. Rountree, W. Feng, and B. R. de Supinski, "CoreTSAR: Adaptive worksharing for heterogeneous systems," in *Proc. Int. Supercomput. Conf.*, Leipzig, Germany, Jun. 2014, pp. 172–186.

[26] Parallel Computing Forum, "PCF parallel Fortran extensions," *ACM SIGPLAN Fortran Forum*, vol. 10, no. 3, pp. 1–57, Sep. 1991.

[27] M. Wolfe. (Jun. 2016). *Compilers and More: OpenACC to OpenMP (and Back Again)*. [Online]. Available: https://www.hpcwire.com/2016/06/29/compilers-openacc-openmp-back/

ABOUT THE AUTHORS

Bronis R. de Supinski received the Ph.D. degree in computer science from the University of Virginia, Charlottesville, VA, USA, in 1998.

As the Chief Technology Officer for Livermore Computing, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, USA, he formulates LLNL's large-scale computing strategy and oversees its implementation. He is also a Professor of Exascale Computing at Queen's University of Belfast and an Adjunct Associate Professor in the Department of Computer Science and Engineering, Texas A&M University.

Thomas R. W. Scogland received the Ph.D. degree in computer science from Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, in 2014.

He is a Computer Scientist in the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA. His research interests include parallel programming models, heterogeneous computing and resource management at scale. He serves on the OpenMP Language Committee, the C and C++ committees, and as Cochair of the Green500.

Alejandro Duran received the Ph.D. degree in computer architecture by the Universitat Politècnica de Catalunya, Barcelona, Spain, in 2008.

He currently is an Application Engineer at Intel Corporation, Barcelona, Spain. He joined the OpenMP Language Committee in 2006. His areas of interest are parallel programming models, runtime optimizations and performance analysis.

Michael Klemm received the Doctor of Engineering degree from the Friedrich-Alexander-University Erlangen-Nuremberg, Erlangen, Germany, in 2008.

He works in the Developer Relations Division, Intel, Nürnburg, Germany and his areas of interest include compiler construction, design of programming languages, parallel programming, and performance analysis and tuning. He joined the OpenMP organization in 2009 and was appointed CEO of the OpenMP ARB in 2016.

Sergi Mateo Bellido received the bachelors degree in computer science from the Universitat Politècnica de Catalunya, Barcelona, Spain, in 2012.

Since 2011, he has been working as a Compiler Engineer in the Programming Models group, Barcelona Supercomputing Center, Barcelona, Spain. He joined the OpenMP Language Committee in 2014. His areas of interest are compilers, domain specific languages, parallel programming models and performance analysis.

Stephen L. Olivier received the Ph.D. degree in computer science from the University of North Carolina, Chapel Hill, NC, USA, in 2012.

He is a Principal Member of the technical staff in the Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, USA. His research interests include runtime systems, parallel programming models, and power-aware high performance computing. He serves on the OpenMP Language Committee and is Chair of the Tasking Subcommittee.

Christian Terboven received the Doctor of Natural Sciences degree from RWTH Aachen University, Aachen, Germany.

He leads the HPC Group at RWTH, and his research interests include parallel programming models, related software engineering aspects, and the optimization of simulation codes for modern HPC architectures. Since 2006, he serves on the OpenMP Language Committee and is Chair of the Affinity Subcommittee.

Timothy G. Mattson received the Ph.D. degree in chemistry from the University of California Santa Cruz (UCSC), Santa Cruz, CA, USA, in 1985.

He leads the Programming Systems Research group at Intel where he works on parallel programming models, graph algorithms in terms of sparse linear algebra, polystore data management systems, and machine learning applied to software generation. He was part of the original crew that created OpenMP back in 1996 and served as one of the first CEOs of the OpenMP ARB.