Bronis R. de Supinski
Pedro Valero-Lara
Xavier Martorell
Sergi Mateo Bellido
Jesus Labarta (Eds.)

LNCS 11128

Evolving OpenMP for Evolving Architectures

14th International Workshop on OpenMP, IWOMP 2018
Barcelona, Spain, September 26–28, 2018
Proceedings

Springer

Bronis R. de Supinski · Pedro Valero-Lara
Xavier Martorell · Sergi Mateo Bellido
Jesus Labarta (Eds.)

Evolving OpenMP for Evolving Architectures

14th International Workshop on OpenMP, IWOMP 2018
Barcelona, Spain, September 26–28, 2018
Proceedings

Springer

Editors
Bronis R. de Supinski
Lawrence Livermore National Laboratory
Livermore, CA
USA

Sergi Mateo Bellido
Barcelona Supercomputing Center
Barcelona, Barcelona
Spain

Pedro Valero-Lara
Barcelona Supercomputing Center
Barcelona, Barcelona
Spain

Jesus Labarta
Universitat Politècnica de Catalunya
Barcelona, Barcelona
Spain

Xavier Martorell
Universitat Politècnica de Catalunya
Barcelona
Spain

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-98520-6 ISBN 978-3-319-98521-3 (eBook)
https://doi.org/10.1007/978-3-319-98521-3

Library of Congress Control Number: 2018950652

LNCS Sublibrary: SL2 – Programming and Software Engineering

Preface

OpenMP is a widely accepted, standard application programming interface (API) for high-level shared-memory parallel programming in Fortran, C, and C++. Since its introduction in 1997, OpenMP has gained support from most high-performance compiler and hardware vendors. Under the direction of the OpenMP Architecture Review Board (ARB), the OpenMP specification has evolved up to and beyond version 4.5. The 4.5 version includes several refinements to existing support for heterogeneous hardware environments, many enhancements to its tasking model including the task-loop construct, and support for doacross loops. As indicated in TR7, OpenMP 5.0, will include significant new features, such as mechanisms for memory affinity and the standardization of tool APIs, and improvements in existing ones, such as the device and tasking constructs.

The evolution of the standard would be impossible without active research in OpenMP compilers, runtime systems, tools, and environments. OpenMP is important both as a standalone parallel programming model and as part of a hybrid programming model for massively parallel, distributed memory systems built from multicore, manycore, and heterogeneous node architectures. Overall, OpenMP offers important features that can improve the scalability of applications on expected exascale architectures.

The community of OpenMP researchers and developers is united under the cOMPunity organization. This organization has held workshops on OpenMP around the world since 1999: the European Workshop on OpenMP (EWOMP), the North American Workshop on OpenMP Applications and Tools (WOM-PAT), and the Asian Workshop on OpenMP Experiences and Implementation (WOMPEI), which attracted annual audiences from academia and industry. The International Workshop on OpenMP (IWOMP) consolidated these three workshop series into a single annual international event that rotates across Europe, Asia-Pacific, and the Americas. The first IWOMP workshop was organized under the auspices of cOMPunity. Since that workshop, the IWOMP Steering Committee has organized these events and guided the development of the series. The first IWOMP meeting was held in 2005, in Eugene, Oregon, USA. Since then, meetings have been held each year, in: Reims, France; Beijing, China; West Lafayette, USA; Dresden, Germany; Tsukuba, Japan; Chicago, USA; Rome, Italy; Canberra, Australia; Salvador, Brazil; Aachen, Germany; Nara, Japan; and Stony Brook, USA. Each workshop has drawn participants from research and industry throughout the world. IWOMP 2018 continued the series with technical papers and tutorials. The IWOMP meetings have been successful in large part due to generous support from numerous sponsors.

The IWOMP website (www.iwomp.org) provides information on the latest event, as well as links to websites from previous years' events. This book contains the proceedings of IWOMP 2018. The workshop program included 16 technical papers, two keynote talks, and a tutorial on OpenMP. The paper "The Impact of Taskyield on the

Design of Tasks Communicating Through MPI" by Joseph Schuchart, Keisuke Tsugane, Jose Gracia, and Mitsuhisa Sato was selected for the Best Paper Award. All technical papers were peer reviewed by at least three different members of the Program Committee.

September 2018

Bronis R. de Supinski
Sergi Mateo Bellido

Organization

Program Committee Co-chairs

Bronis R. de Supinski	Lawrence Livermore National Laboratory, USA
Sergi Mateo Bellido	Barcelona Supercomputing Center (BSC), Spain

General Chair

Jesus Labarta	Universitat Politècnica de Catalunya, Spain

Publication Chair

Pedro Valero-Lara	Barcelona Supercomputing Center (BSC), Spain

Publicity Chairs

Xavier Teruel	Barcelona Supercomputing Center (BSC), Spain
Matthijs van Waveren	OpenMP Architecture Review Board (ARB)

Local Chair

Xavier Martorell	Universitat Politècnica de Catalunya, Spain

Program Committee

Martin Kong	Brookhaven National Laboratory, USA
Christian Terboven	RWTH Aachen University, Germany
Terry Wilmarth	Intel, USA
Nasser Giacaman	University of Auckland, New Zealand
Alejandro Duran	Intel, Spain
Mark Bull	EPCC, University of Edinburgh, UK
Chunhua Liao	Lawrence Livermore National Laboratory, USA
Stephen Olivier	Sandia National Laboratories, USA
James Beyer	Nvidia, USA
Thomas R. W. Scogland	Lawrence Livermore National Laboratory, USA
Hal Finkel	Argonne National Laboratory, USA
Oliver Sinnen	University of Auckland, New Zealand
Rosa M. Badia	Barcelona Supercomputing Center (BSC), Spain
Mitsuhisa Sato	RIKEN AICS, Japan
Eduard Ayguade	Universitat Politècnica de Catalunya, Spain
Larry Meadows	Intel, USA
Deepak Eachempati	Cray Inc., USA

Joachim Protze	RWTH Aachen University, Germany
Priya Unnikrishnan	IBM Toronto Laboratory, Canada
Eric Stotzer	Texas Instruments, USA

IWOMP Steering Committee

Steering Committee Chair

Matthias S. Müller	RWTH Aachen University, Germany

Steering Committee

Dieter an Mey	RWTH Aachen University, Germany
Eduard Ayguadé	BSC and Universitat Politècnica de Catalunya, Spain
Mark Bull	EPCC, University of Edinburgh, UK
Barbara Chapman	Stony Brook University, USA
Bronis R. de Supinski	Lawrence Livermore National Laboratory, USA
Rudolf Eigenmann	Purdue University, USA
William Gropp	University of Illinois, USA
Michael Klemm	Intel, Germany
Kalyan Kumaran	Argonne National Laboratory, USA
Federico Massaioli	CASPUR, Italy
Lawrence Meadows	Intel, USA
Stephen L. Olivier	Sandia National Laboratories, USA
Ruud van der Pas	Oracle, USA
Alistair Rendell	Australian National University, Australia
Mitsuhisa Sato	University of Tsukuba, Japan
Sanjiv Shah	Intel, USA
Josemar Rodrigues de Souza	SENAI Unidade CIMATEC, Brazil
Christian Terboven	RWTH Aachen University, Germany
Matthijs van Waveren	KAUST, Saudi Arabia

Contents

Best Paper

The Impact of Taskyield on the Design of Tasks Communicating Through MPI

Joseph Schuchart[1](✉), Keisuke Tsugane[2], José Gracia[1], and Mitsuhisa Sato[2]

[1] High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Stuttgart, Germany
{schuchart,gracia}@hlrs.de
[2] University of Tsukuba, Tsukuba, Japan
tsugane@hpcs.cs.tsukuba.ac.jp, msato@cs.tsukuba.ac.jp

Abstract. The OpenMP tasking directives promise to help expose a higher degree of concurrency to the runtime than traditional worksharing constructs, which is especially useful for irregular applications. In combination with process-based parallelization such as MPI, the `taskyield` construct in OpenMP can become a crucial aspect as it helps to hide communication latencies by allowing a thread to execute other tasks while completion of the communication operation is pending. Unfortunately, the OpenMP standard only provides little guarantees on the characteristics of the `taskyield` operation. In this paper, we explore different potential implementations of `taskyield` and present a portable black-box tool for detecting the actual implementation used in existing OpenMP compilers/runtimes. Furthermore, we discuss the impact of the different `taskyield` implementations on the task design of the communication-heavy Blocked Cholesky Factorization and the difference in performance that can be observed, which we found to be over 20 %.

Keywords: OpenMP tasks · Task-yield
Blocked Cholesky Factorization · Hybrid MPI/OpenMP · OmpSs

1 Motivation

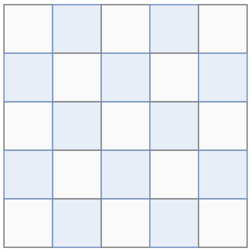
Task-based programming in OpenMP is gaining traction among developers of parallel applications, both for pure OpenMP and hybrid MPI + OpenMP applications, as it allows the user to expose a higher degree of concurrency to the scheduler than is possible using traditional work-sharing constructs such as parallel loops. By specifying data dependencies between tasks, the user can build a task-graph that is used by the scheduler to determine partial execution ordering of tasks. At the same time, OpenMP seems to become an interesting choice for higher-level abstractions to provide an easy transition path [9,10].

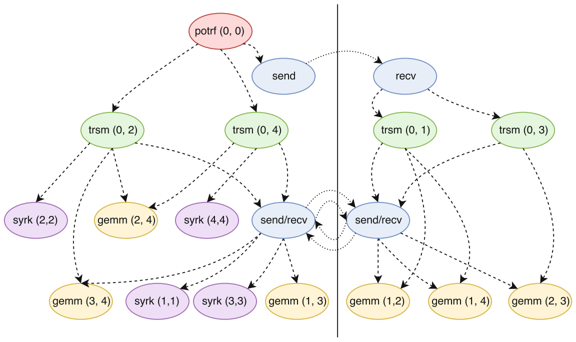
The OpenMP standard specifies the `taskyield` pragma that signals to the scheduler that the execution of the current task *may* be suspended and the thread may execute another task before returning to the current task (in case of `untied` tasks, the execution of the latter may be resumed by a different thread as well) [6]. This feature has the potential to hide latencies that may occur during the execution of a task, e.g., waiting for an external event or operation to complete. Among the sources for such latencies are I/O operations and (more specifically) inter-process communication such as MPI operations.

Traditionally, the combination of MPI + OpenMP using work-sharing constructs requires a fork-join model where thread-parallel processing interchanges with sequential parts that perform data exchange with other MPI ranks. Using OpenMP tasks, the user may structure the application in a way that communication can be performed concurrently with other tasks, e.g., perform computation that is independent of the communication operations [1,4]. In that case, communication may be split into multiple tasks to (i) parallelize the communication and (ii) reduce synchronization slack by achieving a fine-grained synchronization scheme across process boundaries, e.g., start the computation of a of subset of boundary cells as soon as all required halo cells from a specific neighbor have been received. The MPI standard supports thread-parallel communication for applications requesting the support for `MPI_THREAD_MULTIPLE` during initialization [5], which is supported by all major implementations by now.

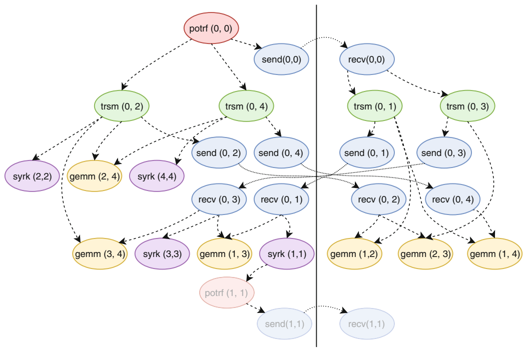
Fine-grained synchronization is especially useful for communication-heavy hybrid applications such as the distributed Blocked Cholesky Factorization, which decomposes a real-valued, positive-definite matrix A into its lower triangular matrix L and its transpose, i.e., $A = LL^T$. The computation of a block can naturally be modeled as a task that may start executing as soon as the blocks required as input are available, either coming from previous local tasks or received from another MPI rank. Figure 1 depicts two different approaches towards decomposing the computation and communication of blocks with a cyclic distribution across two processes (Fig. 1a) into tasks: Fig. 1b shows the use of coarse communication tasks, i.e., all send and receive operations are funneled through a single task. In Fig. 1c the communication is further divided into individual tasks for send and receive operations, which has the potential of exposing a higher degree of concurrency as tasks requiring remote blocks, e.g., some of the tasks at the bottom of the depicted graph, may start executing as soon as the respective receive operation has been completed. In particular, the synchronization stemming from the communication of blocks between tasks on different processes only involves a single producing task and a set of consuming tasks instead of including all producing and consuming tasks. While this may seem trivial for the depicted dual-process scenario, the difference can be expected to be more significant with increasing numbers of processes.

With MPI two-sided communication, a receive operation can only complete once a matching send operation has been posted on the sending side, and vice versa. At the same time, a portable OpenMP application should neither rely on the execution order of runnable tasks nor on the number of threads available to execute these tasks [6, Sect. 2.9.5]. Thus, it should not rely on a sufficient number of threads to be available to execute all available communication tasks, e.g., to make sure all send and receive operations are posted. In order to perform


(a) Cyclic block distribution



(b) Coarse-grain communication tasks



(c) Fine-grain communication tasks

Fig. 1. Task graph of the first iteration of the Blocked Cholesky Factorization on a matrix with cyclic distribution of 5×5 blocks across 2 processes with coarse- and fine-grain communication tasks. Local dependencies are depicted as strong dashed lines and communication as fine-dashed lines.

fine-grained synchronization using MPI two-sided communication, the user may be tempted to rely on `taskyield` to force the task scheduler to start the execution of all available tasks. Otherwise, the communication in Fig. 1c may deadlock as only either `send` or `recv` may be posted concurrently on both processes.

Unfortunately, the OpenMP standard is rather vague on the expected behavior of `taskyield` as it only specifies that the execution of the current task *may* be interrupted and replaced by another task. The standard does not *guarantee* that the current thread starts the execution of another runnable task (if available) upon encountering a `taskyield` directive. An implementation may safely ignore this directive and continue the execution of the currently active task. At the same time, OpenMP does not offer a way to query the underlying implementation for the characteristics of `taskyield`, making it hard for users to judge the extent to which fine-grained communication tasks may be possible.

In this paper, we discuss possible implementations of `taskyield` in Sect. 2. We further present a simple and portable black-box test to query the characteristics of `taskyield` and summarize the results on different OpenMP implementations in Sect. 3. Section 4 provides a performance comparison of different implementations of the distributed Blocked Cholesky Factorization that are adapted to the previously found characteristics. We draw our conclusions and outline future work in Sect. 5.

2 Potential Implementations of Taskyield

As mentioned above, the OpenMP standard leaves a high degree of freedom to implementations as to whether and how `taskyield` is supported. While this provides enough flexibility for implementations, it decreases the portability of application relying on specific properties of `taskyield`. In this section, we present a classification of possible implementations and outline some of their potential benefits and drawbacks.

No-op. Figure 2a depicts the simplest possible implementation: ignoring the `taskyield` statement simplifies the scheduling of tasks as a thread is only executing one task at a time without having to handle different task contexts (the context of a task is implicitely provided by the thread's private stack). For the purpose of hiding latencies, this implementation does not provide any benefit as tasks trying to hide latencies will simply block waiting for the operation to complete without any further useful work being performed in the meantime. Consequently, the <`slack`> of task T1 cannot be hidden and the runnable tasks T2 and T3 are only scheduled once T1 has finished its execution (① and ②). This may lead to deadlocks in two-sided MPI communication if not enough tasks are executed to post a sufficient number of communication operations.

Stack-Based. In Fig. 2b, a stack-based implementation of `taskyield` is depicted. Upon encountering a `taskyield` directive in task T1, the current thread retrieves the next runnable task T2 from the *Queue* (if any) and starts its execution on top of T1 (①). The execution of T1 resumes only after the execution of T2 finished. T1 may test the state of the operations it waits on and yield the thread to another task, e.g., T3 (②). This scheme provides a simple way for hiding latencies: tasks are not required to maintain their own execution context as again the stack of the current thread hosts the execution context and tasks are not enqueued into a queue upon encountering a `taskyield` directive. However, thread-private stacks are typically limited in size (albeit user-configurable) and thus implementations may have to limit the task-yield stack depth. Another drawback is the potential serialization of tasks: if all tasks that are being stacked upon each other happen to call `taskyield`, the re-entrance of the lower tasks is delayed until all tasks on top of them have finished their execution. Deadlocks in MPI two-sided communication may still occur if the number of operations is greater than the accumulated task-yield stack limit of the available threads.

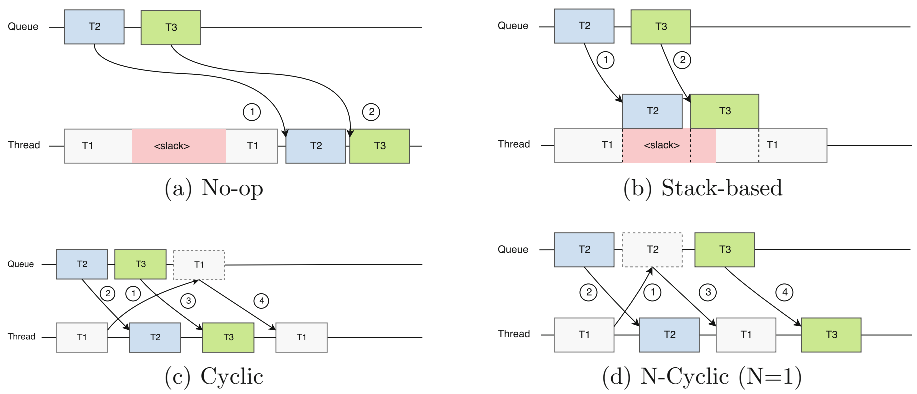


Fig. 2. Possible implementations of task-yield (single thread, three tasks).

Cyclic. An implementation supporting cyclic `taskyield` has the ability to reschedule tasks into the ready-queue for later execution by the same or another thread (in case the task is marked as `untied`). As depicted in Fig. 2c, upon encountering a `taskyield` (①) the thread moves the current task with its context to the end of the ready-queue and starts executing the next available task (②). Since the task T1 has been inserted at the end of the queue, its execution will only resume after the execution of T3 has completed or suspended through a call to `taskyield` (③ and ④). Provided that a sufficient number of runnable tasks is available, the latency of the operation started by T1 can be completely hidden behind the execution of T2 and T3. However, cyclic `taskyield` requires each task to maintain its own execution context as tasks are swapped in and out and thus cannot rely on the thread's stack to host their context. Even in the presence of a large number of communication tasks, cyclic `taskyield` guarantees the execution of all communicating tasks, thus avoiding deadlocks with MPI two-sided communication.

N-cyclic. The N-cyclic `taskyield` depicted in Fig. 2d is a generalization of the cyclic taskyield in that it inserts the yielding task at position N into the queue. This may be desireable by the user for two reasons: (1) a large number of tasks have been created and are runnable but the yielding task should resume its execution only after a few other tasks have been executed and before all of the runnable tasks executed, or (2) the available memory is scarce and may not be sufficient for all remaining tasks to allocate contexts and yield the thread, which would require a large number of contexts to be allocated at the same time. However, a hard-coded (small) limit N would introduce a similar potential for deadlocks in communication-heavy applications as the stack-based yield with limited stack-depth.

3 Existing Taskyield Implementations

We attempted to determine the actual variant of `taskyield` implemented by different OpenMP implementations. While some implementations are open source (GCC, Clang, OmpSs [3]), other implementations have to be treated as a black-box (Cray, PGI, Intel). Thus, we created a black-box test that tries to determine the characteristics of the underlying implementation.[1]

The test is presented in Listing 1.1. The main logic consists of a set of untied tasks with each task containing two `taskyield` regions (lines 15 and 34). The test logic relies on the constraint that only one thread is involved in its execution. Thus, all but the master thread are trapped (lines 9 through 11) in order to avoid restricting the enclosing parallel region to a single thread, which would otherwise skew the result.

It is important that the tasks are marked as `untied` as otherwise threads may not start the execution of sibling tasks, as mandated by the OpenMP task scheduling constraints [6, p. 94].

Before issuing the first task-yield, each task increments a shared variable and stores its value locally to determine the execution order (line 13). After returning from the first yield, the first task checks whether any other task has been executed in the meantime (line 19). If that is not the case, it is easy to determine that the `taskyield` was a `no-op` (line 20). Otherwise, the task continues to first check whether all or a subset of the remaining tasks have already executed the *second task-yield* and thus completed their execution, in which case the task-yield implementation is an unlimited (line 24) or depth-limited (line 26) `stack-based` implementation, respectively. If this is not the case, the task checks whether all tasks or a subset of tasks have at least reached the *first task-yield*, in which case we infer either a `cyclic` (line 28) or `N-cyclic` (line 30) task-yield.

Table 1 lists the implementations we detected using the black-box test discussed above. The result for GCC matches the expectations we had after examining the available source code of `libgomp`. In the case of both Clang and Intel, we found that more than one thread has to be requested to enable a stack-based yield with a depth limit of 257 tasks. Upon further investigation, it appears that this limit is imposed by the task creation throttling, i.e., the number of tasks the master thread creates before it starts participating in the task processing. We have not found a way to control the throttling behavior.

The Cray compiler behaves similarly with a stack-based yield limited by task creation throttling, although the imposed limit appears to be 97 tasks. For the PGI compiler, we determined a `no-op` task-yield.

The OmpSs compiler uses a task throttling mechanism by default, which imposes a configurable upper limit with a default of $T \times 500$ on the number of tasks active, with T being the number of threads [2]. Consequently, the task-yield should be considered `N-cyclic` with $N = T \times 500$. The throttling mechanism can be disabled, which leads to a `full-cyclic` task-yield.

[1] The full code is available at https://github.com/devreal/omp-taskyield.

Listing 1.1. Black-box test for `taskyield` implementations.

```
  volatile int flag_one_cntr = 0;
  volatile int flag_two_cntr = 0;

#pragma omp parallel
#pragma omp master
  for (int i = 0; i < NUM_TASKS+omp_get_num_threads()-1; ++i) {
#pragma omp task firstprivate(i) untied
    {
      if (omp_get_thread_num() > 0) {
        // trap all but thread 0
        while(flag_one_cntr != NUM_TASKS) { }
      } else {
        int task_id = ++flag_one_cntr;

#pragma omp taskyield

        // when come back we only care about the first task
        if (task_id == 1) {
          if (flag_one_cntr == 1) {
            printf("NOOP\n");
          }
          // some other tasks were running in between
          else if (flag_two_cntr == (NUM_TASKS - 1)) {
            printf("STACK (unlimited)\n");
          } else if (flag_two_cntr == flag_one_cntr-1) {
            printf("STACK (depth=%d)\n", flag_one_cntr);
          } else if (flag_one_cntr == NUM_TASKS) {
            printf("CYCLIC\n");
          } else if (flag_one_cntr > 0) {
            printf("N-CYCLIC (N=%d)\n", flag_one_cntr-1);
          }
        }

#pragma omp taskyield

        ++flag_two_cntr;
      } // thread-trap
    } // pragma omp task
  } // for()
```

Table 1. Detected `taskyield` implementations using the black-box test depicted in Listing 1.1. T represents the number of threads.

Runtime	Version tested	Task-yield
GCC	7.1.0	No-op
Clang, Intel	Clang 5.0.1, Intel 18.0.1	No-op
	`OMP_NUM_THREADS > 1`	Stack (257)
Cray CCE	8.6.5	No-op
	`OMP_NUM_THREADS > 1`	Stack (97)
PGI	17.7	No-op
OmpSs	17.06, 18.04	Cyclic ($T \times 500$)
	`NX_ARGS="--throttle=dummy"`	Cyclic

Overall, our black-box test successfully detects the yield characteristics of existing OpenMP implementations. We hope that this test helps users experimenting with the OpenMP `taskyield` directive to choose the right task design depending on the OpenMP implementation at hand.

4 Evaluation Using Blocked Cholesky Factorization

To evaluate the impact of the different `taskyield` implementations on the performance of a communication-heavy hybrid application, we implemented several variants of the Blocked Cholesky Factorization.[2] The benchmark employs BLAS level 3 routines inside OpenMP tasks with defined input and output dependencies to create task graphs similar to the ones depicted in Fig. 1.

4.1 Implementations of Blocked Cholesky Factorization

Funneled Communication. We start from a version that funnels communication through a single task to exchange the blocks computed by the `trsm` tasks, as depicted in Fig. 1b. This version of the benchmark is guaranteed to work with all available OpenMP implementations and any number of threads, as no deadlock in the MPI communication may occur.

The single-task implementation comes in two flavors: in the `funneled` variant the communication task calls `taskyield` while waiting for the communication to finish whereas with `funneled-noyield` task-yield is not used. The latter serves as the baseline as it mimics the `no-op` task-yield on all OpenMP implementations.

Fine-Grained Communication Tasks. In contrast to `funneled`, the `fine` version of the benchmark creates a task per communication operation, leading to fine-grained task dependencies as depicted in Fig. 1c and a potentially higher degree of concurrency exposed to the scheduler. For this version, it is essential that a sufficient number of communication tasks can initiate block transfers to avoid starvation, e.g., $N + 1$ with N being either the number of receiving or sending tasks. Creating this version for `cyclic` task-yield only requires declaring input and output dependencies between the tasks, leaving it to the scheduler to properly execute them. As mentioned above, it is important that communication tasks are marked as `untied` as otherwise threads will not switch between tasks.

Unfortunately, employing fine-grained communication tasks with implementations offering stack-based task-yield requires more effort and has been an tedious process. Specifically, we were required to introduce *dummy tasks* to avoid communication tasks from different communication phases, i.e., tasks communicating the results of `potrf` and `trsm`, to overlap. This partial serialization of tasks is necessary as otherwise `recv` tasks (which naturally do not have local input dependencies) from a later communication phase may be scheduled on top of earlier `recv` tasks. The later `recv` tasks might then stall and never return to

[2] All code is available at https://github.com/devreal/cholesky_omptasks.

the earlier tasks due to implicit transitive dependencies through MPI between them. As an example, consider the `recv(1,1)` task depicted on the right in Fig. 1c being scheduled on top of `recv(0,0)`, which has the transitive implicit dependency `recv(0,0)` $\rightarrow$ `trsm(0,1)` $\rightarrow$ `send(0,1)` $\rightarrow$ `recv(0,1)` $\rightarrow$ `syrk(1,1)` $\rightarrow$ `potrf(1,1)` $\rightarrow$ `send(1,1)` $\rightarrow$ `recv(1,1)`. As a consequence, the user has to identify and expose these implicit dependencies stemming from two-sided communication, which OpenMP is otherwise not aware of.

Per-Rank Communication Tasks. We also made an attempt to reduce the number of communication tasks by combining all `send` and `recv` operations for a specific rank into a single task. This variant is called `perrank` and requires support for dependency iterators (as proposed in [7, p. 62]) to map dependencies from the block domain to the process domain, e.g., to collect dependencies on all blocks that have to be sent to a specific process. At the time of this writing, only OmpSs offered support for dependency iterators (called multi-dependencies).

Even with support for dependency iterators, the `perrank` version is not guaranteed to run successfully on `no-op` task-yield implementations as there are no formal guarantees on the execution order of tasks. Given a guaranteed similar relative execution order of the communication tasks on all processes, e.g., in the (reverse) order in which they were created, `perrank` will run successfully even on a single thread. However, if the execution order is random across processes, e.g., with a reverse order on some processes as a worst case, a minimum number of communication operations has to be in flight to avoid a deadlock. While we have not established a formal definition of this lower bound, we expect it to be below $\frac{(p-1)}{2}$, with p being the number of processes participating in the (potentially all-to-all) block exchange. However, the scalability in terms of processes may be limited by the number of threads available.

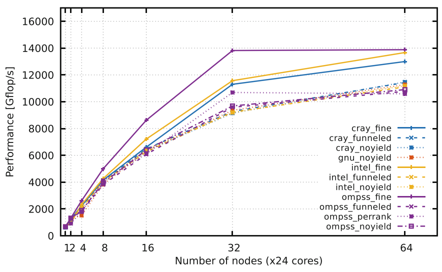
4.2 Test Environment

We ran our tests on two systems: *Oakforest PACS*, a KNL 7250-based system installed at the University of Tsukuba in Japan with 68-core nodes running at 1.4 GHz, and *Hazel Hen*, a Cray XC40 system installed at HLRS in Germany, which is equipped with dual-socket Intel Xeon CPU E5-2680 v3 nodes running at 2.5 GHz. On Oakforest PACS, we employed the Intel 18.0.1 compiler and Intel MPI 2018.1.163. On Hazel Hen, we used the Intel 18.0.1, GNU 7.2.0, and Cray CCE 8.6.5 compilers as well as Cray MPICH 7.7.0. On both systems, we used the OmpSs compiler and runtime libraries in version 17.06 in `ompss` mode.

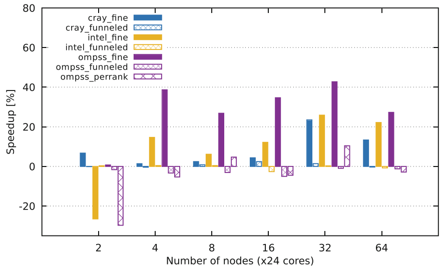
On Oakforest PACS, we relied on the runtime system implementation to ensure proper thread pinning, i.e., using `KMP_AFFINITY=granularity=fine, balanced` for Intel. The OmpSs runtime performs thread pinning by default, which we had to explicitely disable on the Cray system as it would otherwise interfere with the thread pinning performed by `aprun`.

4.3 Results

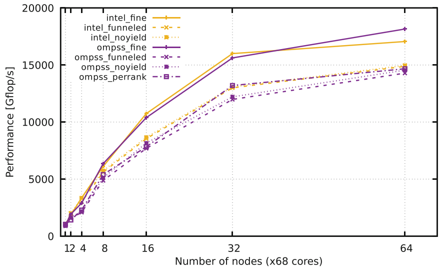
Using the information from the black-box test presented in Sect. 3, we executed the different benchmark implementations discussed above with any suitable OpenMP compiler. We first ran the Cholesky factorization on matrices of size 64k^2 double precision floating point elements with a block size of 512^2 elements.



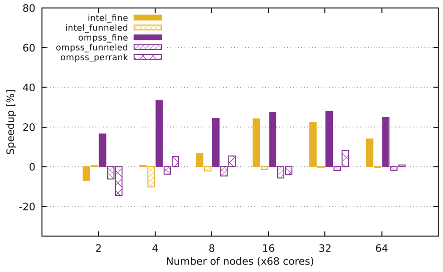
(a) Performance on Hazel Hen



(b) Speedup relative to `noyield` on Hazel Hen



(c) Performance on Oakforest PACS



(d) Speedup relative to `noyield` on Oakforest PACS

Fig. 3. Strong scaling performance and speedup relative to `noyield` of Blocked Cholesky Factorization on a 64k^2 matrix with block size 512^2 using different OpenMP compilers.

The measured performance of the benchmark on the Cray system is presented in Fig. 3a. The most important observation is that the implementations using fine-grained communication tasks (solid line) by far outperform the variant using funneled communication (dashed lines). The speedup of fine-grained communication tasks (depicted in Fig. 3b) range up to 42 % for OmpSs and up to 24 % for both Intel and Cray. For fine-grained dependencies, the OmpSs implementation outperforms the fine-grained implementations of the stack-based `taskyield` present in the Cray and Intel implementations. It is notable that OmpSs appears to saturate performance earlier – at 32 nodes – than the latter two implementations, which approach the saturation point at 64 nodes. It can also be observed that the difference between the `noyield` and the `funneled` version using `taskyield` is marginal, which can be attributed to the fact that yielding a single thread does not constitute a significant increase in resource availability given that a single thread occupies less than 5 % of the node's overall performance.

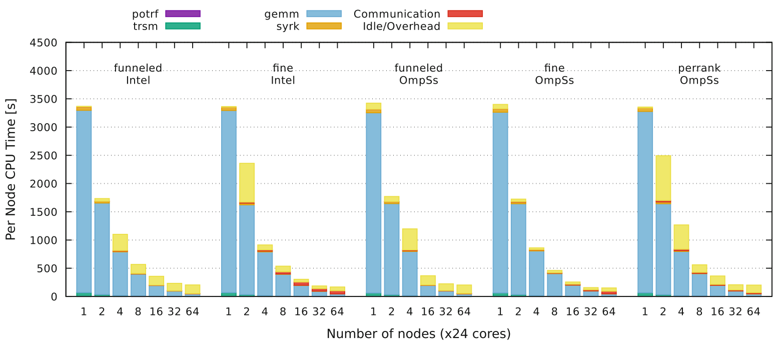
On Oakforest PACS, except for 64 nodes the different benchmark variants perform slightly better when using the Intel OpenMP implementation as compared to running under OmpSs, as depicted in Fig. 3c and d. This may be attributed to the generally higher overhead of task dependency handling observed in OmpSs [8]. However, relative to `noyield`, the versions using fine-grained communication tasks exhibit up to 34 % speedup with OmpSs and 25 % with Intel, with the main improvements seen in mid-range node numbers.

An interesting observation can be made on both systems regarding fine-grained communication tasks on the Intel runtime: for two nodes fine-grained communication tasks can have a negative impact on the performance, which diminishes and turns into performance improvements with increasing node numbers. This effect shrinks again for the highest node numbers.

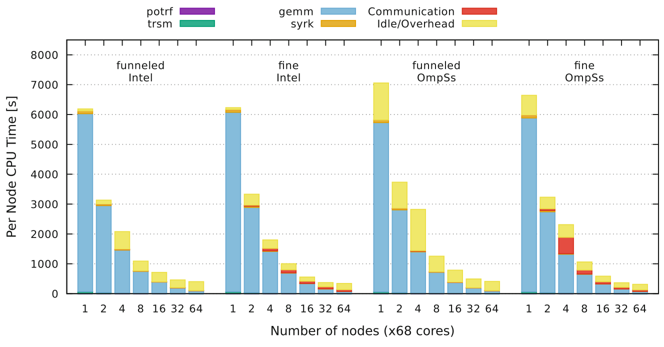
We also note that `perrank` communication tasks do not seem to provide significant benefits over `funneled` communication.

Breakdown of CPU Time. Figure 4 presents a breakdown of the accumulated time spent by the threads on the four BLAS operations and MPI communication as well as idle state and overhead. The latter contains task creation, task instantiation, task switching, and idle times as we have not found a way to further break down these numbers accurately. In all cases, the `gemm` kernel is the dominating factor but it is interesting to note that the overhead/idle times significantly increase with increasing node numbers for the `funneled` versions, indicating a lack of concurrency due to the coarse-grain synchronization. With the fine-grained synchronization, the idle times seem to be lower. However, at least for the Intel compiler the time spent waiting on MPI communication rises with increasing node numbers while this effect is less pronounced in OmpSs. We attribute this to the limiting effect on the available concurrency of stack-based task-yield: tasks below the currently executing task are blocked until the tasks above (and potentially the communication handled by them) have finished. In contrast to this, communication tasks in the `cyclic` task-yield in OmpSs finish as soon as the respective communication operations have finished and their execution has resumed once, allowing idle threads to poll for completion.

Scaling the Problem Size. Figure 5 depicts the strong scaling results of the Blocked Cholesky Factorization on a matrix with 128k^2 elements. Due to the computational complexity of $O(n^3)$, the total number of computation tasks increases by a factor of eight compared to a matrix size of 64k^2, leading to a total number of computation tasks of 2.8 million and putting significantly higher pressure on the task scheduler. On the Cray XC40 (Fig. 5a), OmpSs using fine-grained dependencies again outperforms all other implementations as it benefits from the larger number of available computational tasks, followed by the Intel compiler with fine-grained dependencies. All `funneled` runs show only limited

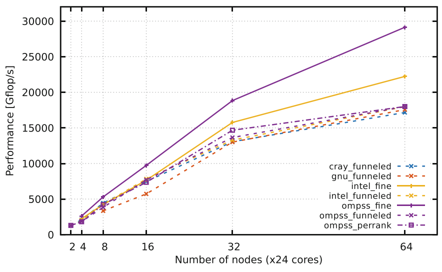


(a) Hazel Hen

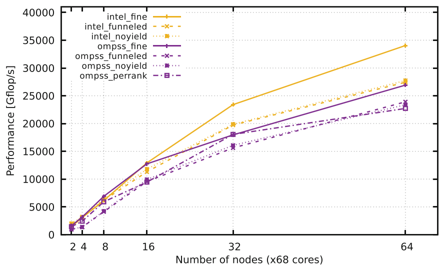


(b) Oakforest PACS

Fig. 4. Breakdown of CPU time of different implementations of the Blocked Cholesky Factorization for a $64k^2$ matrix with block size 512^2.



(a) Hazel Hen



(b) Oakforest PACS

Fig. 5. Strong scaling Performance of Blocked Cholesky Factorization on a $128k^2$ double precision floating point matrix with block size 512^2 using different OpenMP compilers.

scaling as they cannot exploit the full degree of concurrency present due to the coarse-grained synchronization. We should note that we were unable to gather reliable data for fine-grained communication tasks with the Cray compiler as we saw frequent deadlocks in the MPI communication, presumably due to the lower limit on the task-yield stack depth.

On Oakforest PACS (Fig. 5b), the scaling of OmpSs is rather limited compared to the Intel OpenMP implementation. We attribute these limitations to a relatively higher overhead involved in the OmpSs task management, which becomes more significant with the larger number of tasks and the low serial performance of a single KNL core. Again with both OmpSs and the Intel compiler, however, fine-grained communication tasks outperform the version using `funneled` communication on the same compiler. The `perrank` version appears perform slightly better, albeit with a far smaller benefit than the fine-grained communication tasks.

4.4 Discussion

The results presented above demonstrate that the available implementation of `taskyield` in OpenMP can have a significant impact on hybrid applications attempting to hide communication latencies, both in terms of task design, incl. correctness, and in terms of performance. While users can rely on a deadlock-free execution of communication tasks with `cyclic` task-yield, more care has to be given to the synchronization of tasks when using a `stack`-based or `N-cyclic` yield with fine-grained communication tasks. With both `no-op` yield and – less significant – stack-based yield the user has to ensure that a sufficient number of communication operations can be in-flight, e.g., by ensuring a sufficient number of OpenMP threads being available.

The variations of task-yield across OpenMP implementations make the transition from a correct, i.e., deadlock-free, sequential MPI application to a correct task-parallel MPI program tedious. In many cases, it might not be sufficient to simply encapsulate data exchange between individual computation tasks into communication tasks to achieve fine-grained synchronization and rely on OpenMP `taskyield` to ensure scheduling of all necessary communication tasks. Instead, users will have to work around the peculiarities of the different implementations and will thus likely fall back to funneling MPI communication through a single task to guarantee correct execution on all OpenMP implementations, potentially losing performance due to higher thread idle times.

However, our results strongly indicate that fine-grained communication tasks outperform more restricted synchronization schemes such as `funneled` and `perrank` as the former has the potential to significantly increase the concurrency exposed to the runtime. Unfortunately, an application has no way to query the current OpenMP implementation for information on the properties of `taskyield` to adapt the communication task pattern dynamically. Introducing a way to query these properties in OpenMP would allow users to adapt the behavior of their application to best exploit the hardware potential under any given OpenMP implementation. Similarly, providing a way to control the limits of task-yield in all implementations would help (i) raise awareness of the potential pitfalls, and (ii) provide the user with a way to adapt the runtime to the application's needs. Such configuration options could include the task creation throttling limit (as already offered by OmpSs) and any further limitation affecting the effectiveness of task-yield in the context of distributed-memory applications relying on two-sided communication.

5 Conclusion and Future Work

In this paper, we have presented a classification and evaluation of potential implementations of the `taskyield` construct in OpenMP. We discussed advantages and disadvantages of the different implementations and how a communication-heavy application may have to be adapted to successfully employ task-yield for communication latency hiding. Using a black-box test we were able to determine the characteristics of task-yield in different OpenMP implementations. We have shown that fine-grained communication tasks may outperform more coarse-grained approaches while requiring additional reasoning about implicit transitive dependencies from two-sided communication in `stack`-based task-yield implementations to avoid deadlocks.

Looking ahead, we plan to investigate other types of applications beyond Blocked Cholesky Factorization. While we do not expect the performance impact to be that pronounced on traditional stencil applications, applications from areas such as graph processing may benefit from taskified communication while potentially suffering from similar correctness problems with non-cyclic task-yield as presented in this paper. To help users tackle the issue of implicit transitive dependencies, it might be worth investigating ways to signal their existence to the OpenMP scheduler or provide users means to query or even control some scheduler characteristics, e.g., the relative task execution order and the properties of task-yield. From a user's (idealistic) perspective, the standard would eventually mandate a `cyclic` task-yield to avoid the issues described in this paper.

Acknowledgements. We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA) in the SmartDASH project. Part of this work has been supported by the European Community through the project Mont Blanc 3 (H2020 program, grant agreement N° 671697). This work was supported by a fellowship within the FITweltweit program of the German Academic Exchange Service (DAAD).

References

1. Akhmetova, D., Iakymchuk, R., Ekeberg, O., Laure, E.: Performance study of multithreaded MPI and OpenMP tasking in a large scientific code. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2017. https://doi.org/10.1109/IPDPSW.2017.128
2. BSC Programming Models: OmpSs User Guide, March 2018. https://pm.bsc.es/ompss-docs/user-guide/OmpSsUserGuide.pdf
3. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multicore architectures. Parallel Process. Lett. (2011). https://doi.org/10.1142/S0129626411000151

4. Meadows, L., Ishikawa, K.: OpenMP tasking and MPI in a lattice QCD benchmark. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 77–91. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_6
5. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (Version 3.1) (2015). http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
6. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 4.5 (2015). http://www.openmp.org/mp-documents/openmp-4.5.pdf
7. OpenMP Architecture Review Board: OpenMP Technical report 6: Version 5.0 Preview 2 (2017). http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf
8. Schuchart, J., Nachtmann, M., Gracia, J.: Patterns for OpenMP task data dependency overhead measurements. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 156–168. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_11
9. Tsugane, K., Lee, J., Murai, H., Sato, M.: Multi-tasking execution in PGAS language XcalableMP and communication optimization on many-core clusters. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. HPC Asia 2018. ACM (2018). https://doi.org/10.1145/3149457.3154482
10. YarKhan, A., Kurzak, J., Luszczek, P., Dongarra, J.: Porting the PLASMA numerical library to the OpenMP standard. Int. J. Parallel Program. (2017). https://doi.org/10.1007/s10766-016-0441-6

Loops and OpenMP

OpenMP Loop Scheduling Revisited: Making a Case for More Schedules

Florina M. Ciorba[1](✉), Christian Iwainsky[2], and Patrick Buder[1]†

[1] University of Basel, Basel, Switzerland
{florina.ciorba,p.buder}@unibas.ch
[2] Technische Universität Darmstadt, Darmstadt, Germany
christian.iwainsky@sc.tu-darmstadt.de
http://hpc.dmi.unibas.ch, http://www.sc.tu-darmstadt.de, http://www.hkhlr.de

Abstract. In light of continued advances in loop scheduling, this work revisits the OpenMP loop scheduling by outlining the current state of the art in loop scheduling and presenting evidence that the existing OpenMP schedules are insufficient for all combinations of applications, systems, and their characteristics. A review of the state of the art shows that due to the specifics of the parallel applications, the variety of computing platforms, and the numerous performance degradation factors, no single loop scheduling technique can be a 'one-fits-all' solution to effectively optimize the performance of all parallel applications in all situations. The impact of irregularity in computational workloads and hardware systems, including operating system noise, on the performance of parallel applications results in performance loss and has often been neglected in loop scheduling research, in particular the context of OpenMP schedules. Existing *dynamic loop self-scheduling techniques*, such as trapezoid self-scheduling, factoring and weighted factoring, offer an unexplored potential to alleviate this degradation in OpenMP due to the fact that they explicitly target the minimization of load imbalance and scheduling overhead. Through theoretical and experimental evaluation, this work shows that these loop self-scheduling methods provide a benefit in the context of OpenMP. In conclusion, OpenMP must include more schedules to offer a broader performance coverage of applications executing on an increasing variety of heterogeneous shared memory computing platforms.

Keywords: Dynamic loop self-scheduling · Shared memory · OpenMP

1 Introduction

Loop-level parallelism is a very important part of many OpenMP programs. OpenMP [3] is the decades-old industry standard for parallel programming on shared memory platforms. Due to wide support from industry and academia, a

† Authors in order of contribution.

broad variety of applications from science, engineering, and industry are parallelized and programmed using OpenMP [26].

Applications in science, engineering, and industry are complex, large, and often exhibit irregular and non-deterministic behavior. Moreover, they are frequently *computationally-intensive* and consist of large *data parallel loops*. High performance computing platforms are increasingly complex, large, heterogeneous, and exhibit massive and diverse parallelism. The optimal execution of parallel applications on parallel computing platforms is NP-hard [22]. This is mainly due to the fact that the individual processing times of application tasks[1] cannot, in general, be predicted, in particular on machines with complex memory hierarchies (e.g., non-uniform memory access) [18].

The performance of applications can be degraded due to various "overheads", which are synchronization, management of parallelism, communication, and load imbalance [6]. Indeed, these overheads cannot be ignored by any effort to improve the performance of applications, such as the loop scheduling schemes [9].

Load imbalance is the major performance degradation overhead in computationally-intensive applications [15,19]. It can result from the uneven assignment of computation to units of work (e.g., threads) or the uneven assignment of units of work to processors. The former can be mitigated via a fine-grained decomposition of computation into units of work. The latter is typically minimized via the use of a central (ready) work queue from which idle processors remove units of work. This approach is called *self-scheduling*. The use of a central work queue facilitates a dynamic and even distribution of load among the processors and ensures that no processor remains idle while there is work do be conducted. The dynamic nature of the self-scheduling schedules combined with their centralized work queue characteristic makes them *ideal* for use in OpenMP. Self-scheduling is already supported in OpenMP by the scheduling mechanisms of parallelizing work constructs, namely the `parallel` for loop constructs. The scheduling responsibility falls onto the OpenMP runtime system, rather than on the operating system or the (potentially scheduling non-expert) programmer. The OpenMP runtime system can be precisely optimized for scheduling and shared memory programming, while the operating system must be generic to accommodate a variety of programming models and languages.

No single loop scheduling technique can address all sources of *load imbalance* to effectively optimize the performance of all *parallel applications* executing on all types of *computing platforms*. Indeed, the characteristics of the loop iterations *compounded* with the characteristics of the underlying computing systems determine, typically during execution, whether a certain scheduling scheme outperforms another. The impact of system-induced variability (e.g., operating system noise, power capping, and others) on the performance of parallel applications results in additional irregularity and has often been neglected in loop scheduling research, particularly in the context of OpenMP schedules [23,34].

[1] *Tasks* and *loop iterations* denote independent units of computation and are used interchangeably in this work.

There exists a great body of work on loop scheduling and a taxonomy of scheduling methods is included in Sect. 3. In essence, the present work makes the case that the existing OpenMP schedules (`static`, `dynamic`, and `guided`) are insufficient to cover all combinations of applications, systems, and variability in their characteristics [23,37].

The present work revisits the loop schedules in the OpenMP specification [3], namely `static`, `dynamic`, and `guided` [31], and challenges the assumption that these are sufficient to efficiently schedule all types of applications with parallel loops on all types of shared memory computing systems. Moreover, this work makes the case that OpenMP must include more schedules, and proposes the *loop self-scheduling* class, to offer a broader performance coverage of OpenMP applications executing on an increasing variety of shared memory computing platforms. The additional loop scheduling techniques considered herein are trapezoid self-scheduling [35], factoring [21], weighted factoring [20], and random. The four dynamic loop self-scheduling (DLS) techniques have been implemented in the LaPeSD libGOMP [1] based on the GNU OpenMP library and evaluated using well-known benchmarks. The experimental results indicate the feasibility of adding the four loop self-scheduling techniques to the existing OpenMP schedules. The results also indicate that the existing OpenMP schedules only partially covered the achievable performance spectrum for the benchmarks, system, and pinnings considered. The experiments confirm the original hypothesis and that certain DLS outperform others without a single DLS outperforming all others in all cases. Thus, the results strongly support the case of ample available room for improving applications performance using more OpenMP schedules.

The remainder of this work is organized as follows. Section 2 reviews the work related to implementing various loop schedules into compilers and runtime systems. Loop scheduling is revisited in Sect. 3, with a focus on the class of dynamic loop self-scheduling techniques and the existing OpenMP schedules. Section 4 describes the implementation of the dynamic loop self-scheduling techniques considered in this work, the selection of the benchmarks used to evaluate their performance, as well as offering details of the measurement setup and the experiments. This work concludes in Sect. 5 with a discussion and highlights of future work aspects.

2 Related Work

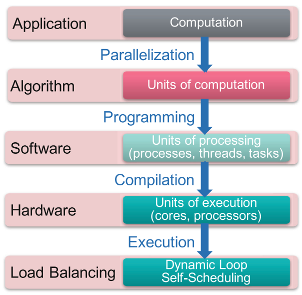
The motivation behind the present work is reinforced by the plethora and diversity of the existing related work, indicating that scheduling of loops in OpenMP is a long-standing and active area of research with diverse opportunities for improvement. The most recent and relevant efforts of implementing additional scheduling methods into parallelizing compilers and runtime systems are briefly discussed next.

A preliminary version of this work [12] contains the first prototype implementation, in LaPeSD libGOMP [1], of three self-scheduling techniques considered herein, trapezoid self-scheduling [35], factoring [21], and weighted factoring [20], while random is a newly added schedule in the present work. The major focus therein was the feasibility of implementing new loop self-scheduling techniques in libGOMP. An exploratory evaluation of the self-scheduling techniques for a broad range of OpenMP benchmarks from various suites confirmed that static scheduling is beneficial only for loops with uniformly distributed iterations; it also confirmed the need for dynamic scheduling of loops with irregularly distributed iterations. Most of the loop schedules proposed over time in OpenMP employ affinity-based principles [4], while the remaining employ work stealing [17,25] or other variations of static scheduling [25,29,30]. A number of approaches rely on profiling information about the application on a given architecture [13,34,38]. Only very little work has considered additional loop self-scheduling methods [23,37], while the present work is the first to consider weighted factoring [20] and random self-scheduling. The impact of system-induced variations has only been considered in limited instances [34], whereas the present work considers the *cumulative impact* of variabilities in problem, algorithm, and system. Most related work considers benchmarks from well-known suites. The present work provides targeted experiments for a particular class of real-world applications, namely molecular dynamics. Existing efforts also implemented their scheduling prototypes in libGOMP [12,13,29,30,34], similar to this work. No studies explore various pinning strategies, which are used in this work to highlight the benefit of dynamic loop self-scheduling. Moreover, in this work experiments are conducted with 20 threads on a state-of-the-art 10$\times$2-way cores Intel Broadwell architecture.

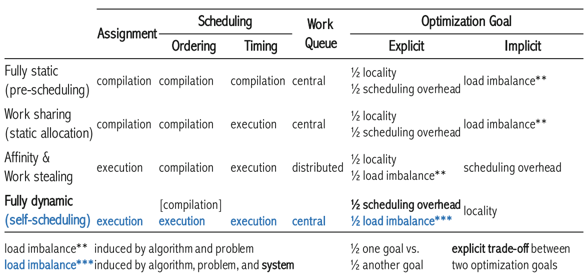
3 Revisiting Loop Scheduling

Parallelization is the process of identifying units of computations that can be performed in parallel on their associated data, as well as the ordering of computations and data, in space and time, with the goal of decreasing the execution time of applications. The process of parallelization consists of three steps: partitioning (or decomposition), assignment (or allocation), and scheduling (or ordering). *Partitioning* refers to the decomposition of the computational domain and of the corresponding computations and data into parallel subparts, called *units of computation* (see Fig. 1a). Through programming, the units of computation are mapped to software *units of processing*, such as processes, threads, or tasks[2] (see Fig. 1a). *Assignment* refers to the allocation of the units of processing to *units of execution* in space, such as cores or processors. The allocation can be performed *statically*, during compilation or before the start of the application execution, or *dynamically*, during application execution (see Fig. 1b). *Scheduling* refers to the ordering and timing of the units of computations to achieve a load balanced execution. The ordering can be *static*, determined at compilation time, *dynamic*, performed during application execution, or *adaptive*, which changes either the static or the dynamic ordering during execution (see Fig. 1b).

[2] Task denotes, in this context, the form of concurrency at the level of a programming paradigm, e.g., OpenMP 3.0 tasks.



(a) Load balancing via dynamic loop self-scheduling



	Assignment	Scheduling		Work Queue	Optimization Goal	
		Ordering	Timing		Explicit	Implicit
Fully static (pre-scheduling)	compilation	compilation	compilation	central	½ locality ½ scheduling overhead	load imbalance**
Work sharing (static allocation)	compilation	compilation	execution	central	½ locality ½ scheduling overhead	load imbalance**
Affinity & Work stealing	execution	compilation	execution	distributed	½ locality ½ load imbalance**	scheduling overhead
Fully dynamic (self-scheduling)	execution	[compilation] execution	execution	central	**½ scheduling overhead** ½ load imbalance***	locality

load imbalance** induced by algorithm and problem
load imbalance*** induced by algorithm, problem, and **system**
½ one goal vs. ½ another goal
explicit trade-off between two optimization goals

(b) Taxonomy of loop scheduling by assignment, scheduling decisions, and target optimization goals

Fig. 1. Loop scheduling and its relation to load balancing, overhead, and locality.

Load imbalance is the major performance degradation factor in computationally-intensive applications [15]. Applications suffer from load imbalance when certain processors are idle and yet there is work ready to be performed that no processor has started. Load imbalance can result from the uneven assignment of computation to units of work (e.g., threads) or the uneven assignment of units of work to processors (see Fig. 1a). The causes of both types of uneven assignment can be due to the *problem* (e.g., non-uniform data distribution), *algorithm* (boundary phenomena, convergence, conditions and branches), or induced by the *system* (memory access interference, operating system noise, or use of shared resources). The uneven assignment of computation to units of work can be mitigated via a fine-grained decomposition of computation into units of work. The uneven assignment of units of work to processors is typically minimized via the use of a central (ready) work queue from which idle and available processors remove units of work. This approach is called *self-scheduling*. Self-scheduling facilitates a dynamic and even distribution of load among the processors and ensures that *no processor remains idle* while there is work to be conducted.

Loop scheduling refers to the third parallelization step, described above, applied to the iterations of a loop. As illustrated in Fig. 1b, depending on the assignment, the ordering and timing of the scheduling decisions, as well as the target optimization goals, existing loop scheduling solutions can be classified into: fully static or *pre-scheduling*, *work sharing*, *affinity* and *work stealing*, and fully dynamic or *self-scheduling*. In the *absence* of load imbalance, approaches that employ pre-scheduling or static allocation are highly effective as they favor data locality and have virtually no scheduling overhead [27]. Affinity-based scheduling strategies [17,24,28,32] also deliver high data locality. In the *presence* of load imbalance, which is the major application performance degradation factor in computationally-intensive applications, strategies based on work stealing only mitigate load imbalance caused by problem and algorithmic characteristics [11,17]. Both affinity and work stealing incur non-negligible scheduling overhead.

Scheduling that employs affinity and work stealing only partially addresses load imbalance and trades it off with data locality. None of the aforementioned scheduling approaches explicitly minimize load imbalance *and* scheduling overhead. *Fully dynamic self-scheduling* techniques, such as those considered in this work, explicitly address *all* sources of load imbalance (caused by problem, algorithmic, and systemic characteristics) and attempt to minimize the scheduling overhead, while implicitly addressing allocation delays and data locality [8,10,36].

This work considers dynamic loop *self-scheduling* in the context of OpenMP scheduling, to explicitly address *load imbalance* and *scheduling overhead* for the purpose of minimizing their impact on application performance. The OpenMP specification [3] provides three types of loop schedules: `static`, `dynamic`, and `guided`, which can be directly selected as arguments to the OpenMP `parallel for schedule()` clause. The loop schedules can also automatically be selected by the OpenMP runtime via the `auto` argument to `schedule()` or their selection can be deferred to execution time via the `runtime` argument to `schedule()`.

The use of `schedule(static,chunk)` employs *straightforward parallelization* or *static block scheduling* [27] (STATIC) wherein N loop iterations are divided into P chunks of size $\lceil N/P \rceil$; P being the number of processing units. Each `chunk` of consecutive iterations is assigned to a processor, in a round-robin fashion. This is only suitable for *uniformly distributed* loop iterations and in the *absence* of load imbalance. The use of `schedule(static,1)` implements *static cyclic scheduling* [27] wherein single iterations are statically assigned consecutively to different processors in a cyclic fashion, i.e., iteration i is assigned to processor $i \texttt{ mod } P$. For certain non-uniformly distributed parallel loop iterations, cyclic produces a more balanced schedule than block scheduling. Both versions achieve high locality with virtually no scheduling overhead, at the expense of poor load balancing if applied to loops with irregular loop iterations or in systems with high variability. The dynamic version of `schedule(static,chunk)` which employs *dynamic block scheduling* is `schedule(dynamic,chunk)`. The only difference is that the assignment of chunks to processors is performed during execution. The dynamic version of `schedule(static,1)` is `schedule(dynamic,1)` which employs *pure self-scheduling* (PSS), the easiest and most straightforward dynamic loop self-scheduling algorithm [33]. Whenever a processor is idle, it retrieves an iteration from a central work queue. PSS[3] achieves good load balancing yet may introduce excessive scheduling overhead. *Guided self-scheduling* (GSS) [31] is implemented by `schedule(guided)`, one of the early self-scheduling techniques that trades off load imbalance and scheduling overhead.

The above OpenMP loop schedules are insufficient to cover the needs for efficient scheduling of all types of applications with parallel loops on all types of shared memory computing systems. This is mainly due to overheads, such as synchronization, management of parallelism, communication, and load imbalance, that cannot be ignored [9].

[3] For simplicity and consistency with prior work, PSS is herein denoted SS.

This work proposes the use of self-scheduling methods in OpenMP to offer a broader performance coverage of OpenMP applications executing on an increasing variety of shared memory computing platforms. Specifically, the loop self-scheduling techniques to consider are *trapezoid self-scheduling* (TSS) [35], *factoring "2"* (FAC2) [21], *weighted factoring "2"* (WF2) [20], and *random*. It is important to note that the FAC2 and WF2 methods evolved from the probabilistic analysis that gave birth to FAC and WF, respectively, while TSS is a deterministic self-scheduling method. Moreover, WF2 can employ workload balancing information specified by the user, such as the capabilities of a heterogeneous hardware configuration.

Based on their underlying models and assumptions, these self-scheduling techniques are expected to trade load imbalance and scheduling overhead as illustrated in Fig. 2.

Random is a self-scheduling-based method that employs the uniform distribution between a lower and an upper bound to arrive at a randomly calculated chunk size between these bounds.

A comparison of the prior existing and newly added OpenMP loop schedules is illustrated in Fig. 3 for scheduling 100 (uniformly distributed) tasks on 4 homogeneous processors. STATIC denotes `schedule(static,chunk)`, GSS denotes `schedule(guided)`, and SS denotes `schedule(dynamic,1)`. The ordering of work requests made by the threads as illustrated in Fig. 3a is only an instantiation of a dynamic process, and may change with every experiment repetition. Even though the scheduling overheads and allocation delays are not accounted for in this example (Fig. 3), one can easily identify the differences between the chunk sizes, the total number of chunks, their assignment order, and the impact of all these factors on the total execution time. STATIC and SS represent the two extremes of the *load balance vs. scheduling overhead* trade-off (see Fig. 2). The remaining DLS techniques are ordered by their efficiency in minimizing scheduling overhead and load imbalance. The `schedule(guided,chunk)`[4] and `schedule(dynamic,chunk)`[5] versions of the respective OpenMP schedules are not considered here due to the fact that they are simple variations of GSS and SS and fall between the two extremes of the trade-off illustrated in Fig. 2, which, for reasons of clarity, only illustrates the schedules considered in this work.

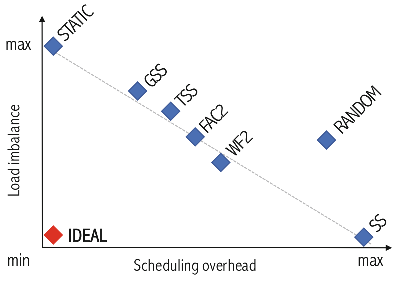
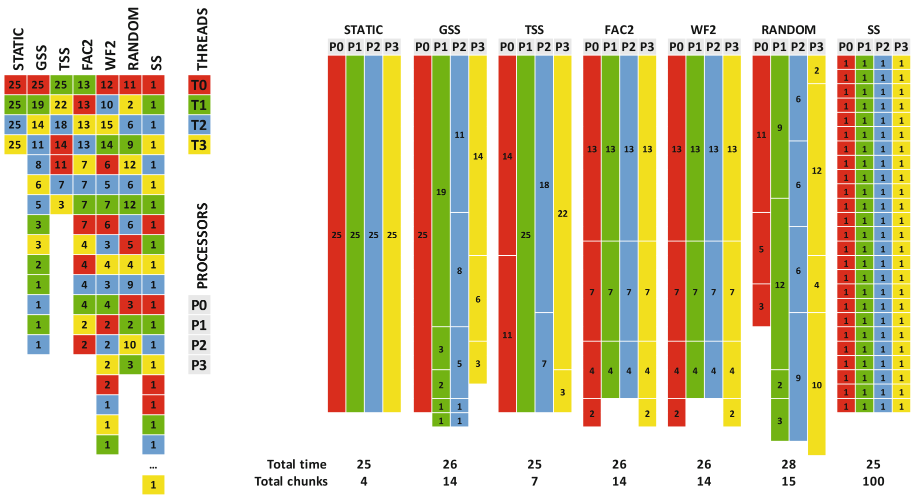


Fig. 2. Load imbalance vs. scheduling overhead trade-off for various dynamic loop self-scheduling schemes.

[4] Here `chunk` denotes the smallest chunk size to be scheduled.

[5] Here `chunk` denotes a fixed chunk size to be scheduled.


STATIC
GSS
TSS
FAC2
WF2
RANDOM
SS
THREADS
T0
T1
T2
T3
PROCESSORS
P0
P1
P2
P3
Total time 25 26 25 26 26 28 25
Total chunks 4 14 7 14 14 15 100

(a) Chunk sizes (b) Chunk execution

Fig. 3. The use of different DLS techniques for (a) calculation of chunk sizes (numbers within colored rectangles) for 100 tasks and (b) their execution over time using 4 threads (T0,..,T3) assigned to 4 processing units (P0,..,P3). While certain DLS achieve comparable "absolute" execution times, the incurred scheduling overhead, directly proportional with the number of self-scheduled chunks, may prohibitively degrade performance. (Color figure online)

4 Evaluation of Selected DLS Techniques

In its current state, the OpenMP standard contains the two extremes of the scheduling spectrum and one self-scheduling example (see Fig. 2). While the state of the art has produced important additional scheduling methods over the last decades, the OpenMP standard has not yet adopted these advances.

To evaluate the benefit of more advanced loop schedules, four additional schedules were added to an OpenMP runtime library for the GNU compiler called `LaPeSD-libGOMP` [1] and tested on molecular dynamics (MD) benchmarks, which are known use non-uniformly distributed `parallel for` loops resulting in imbalanced workloads (problem and algorithmic imbalance) [7]. CPU binding and pinning was used to inject additional load imbalance (system imbalance) in those applications. The four added loop scheduling techniques were compared against the schedules available in current OpenMP using this libGOMP version, the MD benchmarks and special measurement setup.

Benchmarks and Schedule Selection: This work considers applications simulating MD, which typically consist of large loops over complex molecular structures with variable degree of interactivity between different molecules. While full MD applications are highly complex and challenging to be used for comparison in a performance testing environment, the comparatively short runtimes allowed an efficient exploration of the intended parameter space. The choice of MD benchmarks is made from various well-established suites and includes: `c_md` from the OpenMP Source Code Repository (SCR) suite [16], `lava.md` from the RODINIA suite [14], `NAS MG` from the NAS OpenMP suite [5], and `350.md` from the SPEC OpenMP 2012 suite [2].

An additional test application considered in this work is an in-house implementation of a linear algebra kernel, *adjoint convolution* with decreasing task size, called `ac`[6]. This benchmark provides an ideal input to self-scheduling due to its high imbalance caused by the decreasing task size towards the end of the kernel. Overall, the added benefit of using benchmarks is the avoidance of issues typically present in full production-ready codes, such as variations and influences of the MPI parallelization and I/O activities.

The four additional dynamic loop self-scheduling (DLS) techniques considered herein are trapezoid self-scheduling (TSS) [35], factoring (FAC2)[21], weighted factoring (WF2) [20], and random (RAND). While the literature offers more advanced DLS, this work considers DLS that require no additional runtime measurement during their execution, i.e., all used DLS are dynamic and non-adaptive[7]. Recall that the FAC2 and WF2 methods evolved from the probabilistic analysis behind FAC and WF, respectively. Moreover, recall that WF2 can employ workload balancing information specified by the user, such as the capabilities of a heterogeneous hardware configuration.

Prototype Implementation: The development of a dedicated OpenMP compiler and runtime implementation is beyond the scope of this work. Therefore, the GNU implementation of OpenMP is used, as the scheduling mechanisms are independent from the actual compilation of OpenMP constructs and of their separate implementation in a separate runtime library. For the implementation[3] of the additional loop schedules the `LaPeSD-libGOMP` is used as basis.

Measurement Setup and Evaluation Policy: The experiments for this work were conducted on a single node of the *miniHPC* system at the University of Basel, a fully-controlled research system with a dual-socket 10-core Intel Xeon E5-2640 v4, with both hyper-threading and TurboBoost enabled. In all experiments the hyper-threaded cores with ID 20–39 were left idle.

All experiments were conducted exclusively, each experiment being repeated 20 times. For the results shown in this work, the *median* and the *standard deviation* of the execution times were gathered for all repetitions. As discussed in Sect. 3, load imbalance has three potential causes: either the workload of the iteration space is imbalanced, the work assignment is imbalanced, or the hardware is imbalanced, e.g., as in a heterogeneous system. Since the work assignment

[6] In-house codes available at https://bitbucket.org/PatrickABuder/libgomp/src.

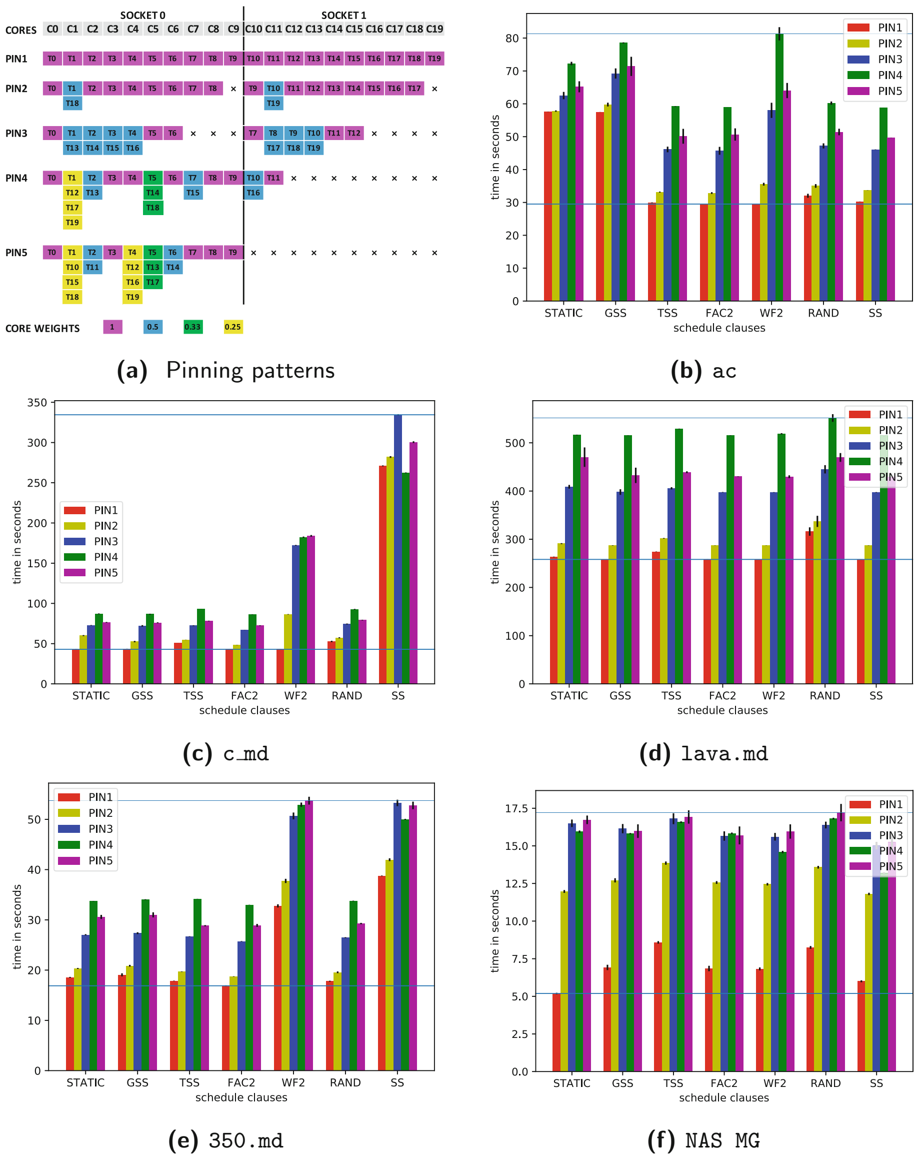
[7] In non-adaptive dynamic scheduling, the schedule does not adapt to runtime performance observations other than to the dynamic consumption of tasks.

(scheduling) is defined by the used schedule clause, any load imbalance originates in either or in both, variability in the benchmark itself or from performance variability in the hardware.

For this work, the principal load imbalance originates in the workloads in OpenMP codes of the MD benchmarks. However, as the benchmarks, in their original form, are rather limited in loop sizes and input data, the inherent imbalance is not extreme. To highlight the impact of the loop scheduling choice, additional artificial hardware irregularity is added by omitting specific CPU cores for use in the experiments and by binding the 20 OpenMP threads to the remaining cores, reducing the achieved performance of those threads in the process. Figure 4a provides an overview of the five pinning and oversubscription configurations used. In the experiments, *PIN1* to *PIN5* were used as the pinning and binding schemes. *PIN1* uses all the cores of the system and binds each thread to a single core. *PIN2* considers cores 9 and 18 as not present in the system and binds two additional threads to cores 1 and 11 (respectively), reducing the performance of threads $1, 10, 18$, and 19. *PIN3* increases the hardware load imbalance by removing cores $7, 8, 16, 17$ and 18 from the set of available CPU cores, hence, reducing the performance of threads $1, 2, 3, 4, 8, 9, 10, 13, 14, 15, 16$, and 18. *PIN4* adds further inhomogeneity into the system; cores 12 to 19 remain idle, while the twenty threads are bound to cores $1, 2, 5, 7$, and 10. This creates four classes of overloaded CPUs: with 4 threads, with 3 threads, and with 2 threads. The last setup, *PIN5*, uses only one (out of two) socket. Note that pinnings *PIN3*–*PIN5* explicitly avoid a symmetric configuration to increase the level of imbalance in the system. This decreases the efficiency of threads being executed on those cores, hence, implicitly generating a substantial compound load imbalance for the application. An efficient load balancing scheme is expected to mitigate such performance variations due to compound load imbalance such as results from the *PIN3*–*PIN5* setups.

Experimental Evaluation: The results of all selected benchmarks are shown in Fig. 4b to f. For this work, a specific schedule is considered to be advantageous if it provides a performance benefit directly to the parallel loops or if it provides a performance benefit in coping with the hardware heterogeneity. The `ac` has approximately 10E6 iterations to schedule, having a coefficient of variation of the iterations execution times (CV) of 57% and is, as such, a suitable example for self-scheduling. As expected, the STATIC and GSS schedules struggle to cope with the highly irregular workload, other schedules with a less strict scheduling regiment perform well, with FAC2 performing the best. The STATIC exhibits the least sensitivity to the different *PIN*-regiments with WF2 exhibiting the highest sensitivity. For `ac`, FAC2 offers the overall best performance in all configurations with SS offering comparable performance for *PIN4* and *PIN5* as it can offset its high overhead by achieving the best load balancing effect.

For `c_md` with its roughly 16E3 iterations and CV of 57%, the FAC2 offers the best performance in the regular setup (*PIN1*) with STATIC, GSS and WF2 coming very close; TSS and RAND provide a slightly lower performance while the



(a) Pinning patterns
(b) ac
(c) c_md
(d) lava.md
(e) 350.md
(f) NAS MG

Fig. 4. Pinning Patterns and Measurements. (a) Pinning patterns: T0–T19 represent OpenMP threads, C0–C19 represent physical cores. Core weights depict fraction of the performance of a single core available for a given thread. (b)–(f) The existing OpenMP schedules, STATIC, GSS, and SS, as well as the additional TSS, FAC2, WF2, and RAND schedules from the present own implementation were evaluated for each benchmark. Minimum and maximum runtimes are marked by horizontal blue lines. (Color figure online)

SS exhibits a 5 times lower performance – likely due to the small iteration space and high scheduling overhead. For the imbalanced PIN configurations all schedules, except WF2 and SS, fare well, offering comparable performance. WF2 is substantially affected by the hardware-induced load imbalance – likely due to its initial calibration with the attempt to account for the hardware imbalance and, subsequently, to the lack of iterations to benefit form the high setup-overhead. The SS exhibits an interesting behavior for this benchmark: while it suffers unproportionally for *PIN3*, it performs better for *PIN4*, even against its *PIN1* configuration.

The `lava.md` benchmark has a comparable low peak CV of 14% in its 13E4 iterations. For this reason, it is not surprising that all but the RAND schedule offer comparable performance; RAND even induces a load imbalance into the benchmark. For the *PIN2–PIN5* configurations there is little difference between the schedules, again with the exception of RAND.

For the `350.md` the iteration count is about 27E3 with a high CV of 8700%. Again FAC2 has the best performance advantage, making use of the high variations in the loop iterations. The STATIC, GSS, TSS, and RAND schedules also perform well and reach about 10% of the peak performance. SS and WF2 provide the lowest performance, failing to exploit the high CV in the workload due to the overhead of scheduling a much larger number of chunks.

The `NAS MG` benchmark has a very low loop iteration space, between 2 and 1k, with a very strongly variable CV between 0, i.e., no variation, and 1. However, many such loops are instantiated in this benchmark which results in approximately 5k instantiations. Here the STATIC schedule offers the best performance for the *PIN1* configuration as its overhead is minimal and the iteration space and, thus, the aggregated load imbalance is low. For this reason, the SS offers the best performance in the hardware imbalanced *PIN2–PIN5* configurations, as it has few iterations to schedule and a high hardware-based imbalance to mitigate. For the same reason, the WF2 also performs much better, even though the cost to calculate a chunk is higher – it is amortized by the well balanced scheduling.

Discussion: In this work an outline of a prototype implementation of additional schedules in OpenMP has been provided together with an evaluation thereof using various benchmarks and hardware setups. While it is clear that the additional schedules do not provide a one-fits-all solution, the greater variety of choice offers an opportunity for improvement in many instances. The precise circumstances of such a benefit must be taken into account when deciding which scheduling choice is the best. The number of iterations to be scheduled, the chunk sizes, and the overheads associated with each approach factor heavily into the decision of which schedule to use. The experience from the prototype implementation and the subsequent test on benchmarks, support the original hypothesis that indeed, *additional schedules provide benefit.* While a single schedule was applied for all work-sharing loops in a code, a more diverse use of the scheduling clause will further increase the benefit. This can, for example, be achieved by leveraging domain expert knowledge, not considered in this work. A critical observation is that many loops from the considered benchmarks *did not use any scheduling clause* in the various work-sharing loops. It is well known that explicitly using the OpenMP `schedule` is critical for improving performance. This can bee seen in `c_md`, `ac`, `350.md` and `NAS MG`, where the performance difference between the fastest and slowest schedule exceeded a factor of two.

5 Summary

This work revisits the scheduling of OpenMP work-sharing loops and offers an overview of the current state-of-the-art self-scheduling techniques. Scheduling is a performance critical aspect of loops that is currently receiving insufficient attention. This work investigates alternative state-of-the-art loop self-scheduling schemes from the literature and reflects on their use in OpenMP loop scheduling. In Sect. 3, it was shown that the existing OpenMP schemes STATIC and SS are only the extremes of a broad spectrum of state-of-the-art loop self-scheduling methods, and that GSS represents a well-known, yet obsolete variant within that spectrum, with many newer more efficient schedules being available. Using a selection of more recent loop-scheduling techniques, prototype implementations were developed for the GNU OpenMP runtime library and used in viability experiments. Section 4 provides an overview of this effort as well as measurements results. These results show that more recent loop self-scheduling techniques exhibit, in three out of five test cases, an improved performance than the schedules already available in OpenMP. With the theoretical underpinnings of loop self-scheduling and the results presented, it is clear that OpenMP should include a greater variety of loop schedules.

Fueled by the increasing heterogeneity of hardware, i.e., classic CPUs in combination with accelerators and future heterogeneous CPUs, a fixed set of schedules will not suffice, even if it were to outperform all currently available scheduling techniques (in the absence of load imbalance). Therefore, the OpenMP community is invited to address this issue, especially with the addition of tasking that brings additional challenges in the area of scheduling in OpenMP. From this perspective, an interface for user-defined schedules, comparable to [25], would be preferable, as it allows users and library developers to add scheduling techniques without having to update the OpenMP standard every time a new approach is developed. In addition, the interface should enable developers to provide the runtime system with more expert knowledge and information regarding the variability of the iteration space of the loops in their applications.

Future work will explore more schedules with more complex underlying models, such as the adaptive self-schedules, which employ feedback loops and require performance measurements. The applicability of self-scheduling is also of high interest, as for example the global load balancing of task scheduling in taskloops remains unaddressed. Further research in the domain of heterogeneity and NUMA and its impact on scheduling is also planned.

Acknowledgment. This work is partly funded by the Swiss National Science Foundation in the context of the "Multi-level Scheduling in Large Scale High Performance Computers" (MLS) grant, number 169123, and by the Hessian State Ministry of Higher Education by granting the "Hessian Competence Center for High Performance Computing".

References

1. An Enhanced OpenMP Library. https://github.com/lapesd/libgomp. Accessed 27 Apr 2018
2. SPEC OMP2012. https://www.spec.org/omp2012/. Accessed 27 Apr 2018
3. The OpenMP API specification for parallel programming. http://www.openmp.org. Accessed 27 Apr 2018
4. Ayguadé, E., et al.: Is the *schedule* clause really necessary in OpenMP? In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 147–159. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45009-2_12
5. Bailey, D.H., et al.: The NAS parallel benchmarks: summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing 1991, pp. 158–165. ACM, New York (1991)
6. Banicescu, I.: Load Balancing and data locality in the parallelization of the fast multipole algorithm. Ph.D. thesis, New York Polytechnic University (1996)
7. Banicescu, I., Flynn Hummel, S.: Balancing processor loads and exploiting data locality in N-body simulations. In: Proceedings of IEEE/ACM SC 1995 Conference on Supercomputing, p. 43 (1995)
8. Banicescu, I., Liu, Z.: Adaptive factoring: a dynamic scheduling method tuned to the rate of weight changes. In: Proceedings of 8th High Performance Computing Symposium, pp. 122–129. Society for Computer Simulation International (2000)
9. Bast, H.: Dynamic scheduling with incomplete information. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1998, pp. 182–191. ACM, New York (1998)
10. Bast, H.: Scheduling at twilight the EasyWay. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 166–178. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45841-7_13
11. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
12. Buder, P.: Evaluation and analysis of dynamic loop scheduling in OpenMP. Master's thesis, University of Basel, November 2017
13. Cammarota, R., Nicolau, A., Veidenbaum, A.V.: Just in time load balancing. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 1–16. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37658-0_1
14. Che, S., Sheaffer, J.W., Boyer, M., Szafaryn, L.G., Wang, L., Skadron, K.: A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC 2010), pp. 1–11. IEEE Computer Society, Washington, DC (2010)
15. Dongarra, J., Beckman, P., et al.: The international exascale software roadmap. Int. J. High Perform. Comput. Appl. **25**(1), 3–60 (2011)

16. Dorta, A.J., Rodriguez, C., Sande, F.d., Gonzalez-Escribano, A.: The OpenMP source code repository. In: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2005, pp. 244–250. IEEE Computer Society, Washington, DC (2005)
17. Durand, M., Broquedis, F., Gautier, T., Raffin, B.: An Efficient OpenMP loop scheduler for irregular applications on large-scale NUMA machines. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 141–155. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_11
18. Durand, M.D., Montaut, T., Kervella, L., Jalby, W.: Impact of memory contention on dynamic scheduling on NUMA Multiprocessors. In: Proceedings of International Conference on Parallel Processing, vol. 1, pp. 258–262, August 1993
19. Flynn Hummel, S., Banicescu, I., Wang, C.-T., Wein, J.: Load balancing and data locality via fractiling: an experimental study. In: Szymanski, B.K., Sinharoy, B. (eds.) Languages, Compilers and Run-Time Systems for Scalable Computers, pp. 85–98. Springer, Boston (1996). https://doi.org/10.1007/978-1-4615-2315-4_7
20. Flynn Hummel, S., Schmidt, J., Uma, R.N., Wein, J.: Load-sharing in heterogeneous systems via weighted factoring. In: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1996, pp. 318–328. ACM, New York (1996)
21. Flynn Hummel, S., Schonberg, E., Flynn, L.E.: Factoring: a practical and robust method for scheduling parallel loops. In: Proceedings of ACM/IEEE Conference Supercomputing (Supercomputing 1991), pp. 610–632, November 1991
22. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1990)
23. Govindaswamy. K.: An API for adaptive loop scheduling in shared address space architectures. Master's thesis, Mississippi State University (2003)
24. Kale, V.: Low-overhead scheduling for improving performance of scientific applications. Ph.D. thesis, University of Illinois at Urbana-Champaign, August 2015
25. Kale, V., Gropp, W.D.: A user-defined schedule for OpenMP. In: Proceedings of the 2017 Conference on OpenMP, Stonybrook, New York, USA, November 2017 (2017)
26. Klemm, M.: Twenty years of the OpenMP API. Scientific Computing World, April/May 2017
27. Li, H., Tandri, S., Stumm, M., Sevcik, K.C.: Locality and loop scheduling on NUMA multiprocessors. In: Proceedings of the 1993 International Conference on Parallel Processing, ICPP 1993, vol. 02, pp. 140–147. IEEE Computer Society, Washington, DC (1993)
28. Markatos, E.P., LeBlanc, T.J.: Using processor affinity in loop scheduling on shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. **5**(4), 379–400 (1994)
29. Penna, P.H., Castro, M., Plentz, P., Cota de Freitas, H., Broquedis, F., Méhaut, J.-F.: BinLPT: a novel workload-aware loop scheduler for irregular parallel loops. In: Simpósio em Sistemas Computacionais de Alto Desempenho, Campinas, Brazil, October 2017
30. Penna, P.H., et al.: Assessing the performance of the SRR loop scheduler with irregular workloads. Procedia Comput. Sci. **108**, 255–264 (2017). International Conference on Computational Science, ICCS 2017, 12–14 June 2017, Zurich, Switzerland
31. Polychronopoulos, C.D., Kuck, D.J.: Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans. Comput. **C–36**(12), 1425–1439 (1987)

32. Subramaniam, S., Eager, D.L.: Affinity scheduling of unbalanced workloads. In: Proceedings of Supercomputing 1994, pp. 214–226, November 1994
33. Tang, P., Yew, P.-C.: Processor self-scheduling for multiple-nested parallel loops. In: Proceedings of International Conference on Parallel Processing, vol. 12, pp. 528–535. IEEE (1986)
34. Thoman, P., Jordan, H., Pellegrini, S., Fahringer, T.: Automatic OpenMP loop scheduling: a combined compiler and runtime approach. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 88–101. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_7
35. Tzen, T.H., Ni, L.M.: Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. IEEE Trans. Parallel Distrib. Syst. **4**(1), 87–98 (1993)
36. Wang, Y., Ji, W., Shi, F., Zuo, Q., Deng, N.: Knowledge-based adaptive self-scheduling. In: Park, J.J., Zomaya, A., Yeo, S.-S., Sahni, S. (eds.) NPC 2012. LNCS, vol. 7513, pp. 22–32. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35606-3_3
37. Zhang, Y., Burcea, M., Cheng, V., Ho, R., Voss M.: An adaptive OpenMP loop scheduler for hyperthreaded SMPs. In. Proceedings of International Conference on Parallel and Distributed Computing Systems (PDCS) (2004)
38. Zhang, Y., Voss, M., Rogers, E.S.: Runtime empirical selection of loop schedulers on hyperthreaded SMPs. In: 19th IEEE International Parallel and Distributed Processing Symposium, p. 44b, April 2005

A Proposal for Loop-Transformation Pragmas

Michael Kruse(✉) and Hal Finkel

Argonne Leadership Computing Facility,
Argonne National Laboratory, Argonne, IL 60439, USA
{mkruse,hfinkel}@anl.gov

Abstract. Pragmas for loop transformations, such as unrolling, are implemented in most mainstream compilers. They are used by application programmers because of their ease of use compared to directly modifying the source code of the relevant loops. We propose additional pragmas for common loop transformations that go far beyond the transformations today's compilers provide and should make most source rewriting for the sake of loop optimization unnecessary. To encourage compilers to implement these pragmas, and to avoid a diversity of incompatible syntaxes, we would like to spark a discussion about an inclusion to the OpenMP standard.

Keywords: OpenMP · Pragma · Loop transformation · C/C++
Clang · LLVM · Polly

1 Motivation

Almost all processor time is spent in some kind of loop, and as a result, loops are a primary targets for program-optimization efforts. One method for optimizing loops is annotating them with OpenMP pragmas, such as `#pragma omp parallel for`, which executes the loop iterations in multiple threads, or `#pragma omp simd` which instructs the compiler to generate vector instructions.

Compared to manually parallelizing the relevant code (e.g. using the pthreads library) or manually vectorizing the relevant code (e.g. using SIMD-intrinsics or assembly), annotating a loop yields much higher programmer productivity. In conjunction with keeping the known-to-work statements themselves unchanged, we can expect less time spent on optimizing code and fewer bugs. Moreover, the same code can be used for multiple architectures that require different optimization parameters, and the impact of adding an annotation can be evaluated easily.

B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 37–52, 2018.
https://doi.org/10.1007/978-3-319-98521-3_3

Directly applied transformations also make the source code harder to understand since most of the source lines will be for the sake of performance instead of the semantics.

Pragmas allow separating the semantics-defining code and the performance-relevant directives. Using `_Pragma("...")`, the directives do not necessarily need appear adjacent to the loops in the source code, but can, for instance, be `#include`d from another file.

Most compilers implement additional, but compiler-specific pragmas. Often these have been implemented to give the programmer more control over its optimization passes, but without a systematic approach for loop transformations. Table 1 shows a selection of pragmas supported by popular compilers.

Table 1. Loop pragmas and the compilers which support them

Transformation	Syntax example	Compiler
Threading	`#pragma omp parallel for`	OpenMP [25]
	`#pragma loop(hint_parallel(0)`	msvc [22]
	`#pragma parallel`	icc [18]
Unrolling	`#pragma unroll`	icc [18], xlc [16], clang [2]
	`#pragma clang loop unroll(enable)`	clang [4]
	`#pragma GCC unroll` *n*	gcc [11]
Unroll and jam	`#pragma unroll_and_jam`	icc [18]
	`#pragma unrollandfuse`	xlc [16]
	`#pragma stream_unroll`	xlc [16]
Loop fusion	`#pragma nofusion`	icc [18]
Loop distribution	`#pragma distribute_point`	icc [18]
	`#pragma clang loop distribute(enable)`	clang [4]
Loop blocking	`#pragma block_loop(`*n*`,`*loopname*`)`	xlc [16]
Vectorization	`#pragma omp simd`	OpenMP [25]
	`#pragma simd`	icc [18]
	`#pragma vector`	icc [18]
	`#pragma loop(no_vector)`	msvc [22]
	`#pragma clang loop vectorize(enable)`	clang [3,4]
Interleaving	`#pragma clang loop interleave(enable)`	clang [3,4]
Software pipelining	`#pragma swp`	icc [18]
Offloading	`#pragma omp target`	OpenMP [25]
	`#pragma acc kernels`	OpenACC [24]
	`#pragma offload`	icc [18]
Assume iteration independence	`#pragma pragma ivdep`	icc [18]
	`#pragma GCC ivdep`	gcc [11]
	`#pragma loop(ivdep)`	msvc [22]
Iteration count	`#pragma loop_count(`*n*`)`	icc [18]
Loop naming	`#pragma loopid(`*loopname*`)`	xlc [16]

```
for (int i = 0; i < M; i+=1)
  for (int j = 0; j < N; j+=1) {
    #pragma omp section id(zero)
    { C[i][j] = 0; }
    for (int k = 0; k < K; k+=1)
      C[i][j] += A[i][k] * B[k][j];
  }

#if __haswell__
  #pragma omp loop(i,j) distribute sections(zero,k)
  #pragma omp loop(i,j,k) tile sizes(96,2048,256) \
      pit_ids(i1,j1,k1) tile_ids(i2,j2,k2)
  #pragma omp loop(i1,...,j2) interchange permutation(j1,k1,i1,j2,i2)
  #pragma omp loop(i2,k2) pack array(A)
  #pragma omp loop(j2,k2) pack array(B)
  #pragma omp loop(i2) simd
#elif __skylake__
  [...]
```

Listing 1.1: Optimization of matrix-matrix multiplication using our proposed pragmas. The tile sizes were derived using the analytical model in [21] for Intel's Kaby Lake architecture.

Our vision is to enable optimizations such as in Listing 1.1, which separates the algorithm from its optimization thus keeping the code readable and maintainable. It also illustrates how different transformations can be applied for different compilation targets. Many BLAS implementations of dgemm use this pattern [21], but pre-transformed in the source. As demonstrated in Sect. 4, the performance is comparable to that of optimized BLAS libraries.

Many existing loop annotation schemes, including OpenMP, require the user to guarantee some "safety" conditions (i.e., that the loop is safe to parallelize as specified) for the use of the annotations to be valid. Making the user responsibility for the validity of a loop transformation may not always be practical, for instance if the code base is large or the loop is complex. Compiler assistance, e.g. providing warnings that a transformation might not be safe, can be a great help. Our use case is an autotuning optimizer, which by itself has only a minimal understanding of the code semantics. Thus, we propose a scheme whereby the compiler can ensure that the semantics of the program remains the same (by ignoring the directive or exiting with an error) when automatically-derived validity conditions are unsatisfied.

2 Proposal

In this publication, we would like to suggest and discuss the following ideas as extensions to OpenMP:

1. The possibility to assign identifiers to loops and code sections, and to refer to them in loop transformations

```
#pragma omp unroll factor(2)
#pragma omp reverse
for (int i=0; i<n; i+=1)
  Statement(i);
```

(a) Reversal followed by partial unrolling

```
#pragma omp reverse
#pragma omp unroll factor(2)
for (int i=0; i<n; i+=1)
  Statement(i);
```

(b) Partial unrolling followed by reversal

Listing 1.2: Transformation composition of loop unrolling and loop reversal

```
#pragma omp loop(outer) thread_parallelize
#pragma omp loop(inner) vectorize
#pragma omp stripmine strip_width(4) strip_id(inner) pit_id(outer)
for (int i = 0; i < n; i+=1)
  Statement(i);
```

Listing 1.3: Loop strip mining with follow-up transformations

2. A set of new loop transformations
3. A common syntax for loop transformation directives

In the following, we discuss these elements of the proposal and a proof-of-concept implementation in Clang. We do not discuss syntax or semantics specific to Fortran, but the proposal should be straightforward to adapt to cover Fortran.

2.1 Composition of Transformations

As loop transformations become more complex, one may want to apply more than one transformation on a loop, including applying the same transformation multiple times. For transformations that result in a single loop this can be accomplished by "stacking up" transformations. Like a regular pragma that applies to the loop that follows, such a transformation directive can also be defined to apply to the output of the following transformation pragma.

The order of transformations is significant as shown in Listing 1.2. In the former the order of execution will be the exact reversal of the original loop, while in the latter, groups of two statements keep their original order.

2.2 Loop/Section Naming

In case a transformation has more than one input- or output-loop, transformations or its follow-up transformations require a means to identify which loop to apply to.

As a solution, we allow assigning names to loops and refer to them in other transformations. An existing loop can be given a name using

```
#pragma omp id(loopname)
for (int i=0; i<n; i+=1) ...
```

and loops from transformations get their identifier as defined by the transformation, for instance as an option. As an example, Listing 1.3 shows a loop that is strip-mined. The inner loop is vectorized while the outer is parallelized.

```
#pragma omp section(A,B) distribute distributed_ids(loopA, loopB)
for (int i=0; i<n; i+=1) {
  #pragma omp id(A)
  { StatementA(i); }
  #pragma omp id(B)
  { StatementB(i); }
}
```

Listing 1.4: Loop distribution example

```
#pragma omp loop(i,j) fuse
for (int i=0; i<n; i+=1) { .. }
for (int j=0; j<n; j+=1) { .. }
```

(a) Working example

```
#pragma omp loop(i,i) fuse
for (int i=0; i<n; i+=1)  { .. }
for (int i=0; i<n; i+=1)  { .. }
```

(b) Ambiguous implicit name

Listing 1.5: Implicit loop name example

In some cases, it may be necessary to not only assign an identifier to the whole loop, but also to individual parts of the loop body. An example is loop distribution: The content of the new loops must be defined and named. Like for (canonical) for-loops, OpenMP also has a notion of sequential code pieces: sections and tasks. We reuse this idea to also assign names to parts of a loop, as shown in Listing 1.4. The transformation results in two loops, named `loopA` and `loopB`, containing a call to `StatementA`, respectively `StatementB`.

Loop and section names form a common namespace, i.e. it is invalid to have a section and a loop with the same name. When being used in a loop transformation, a section name stands for the loop that forms when distributed from the remainder of the loop body.

To avoid boilerplate pragmas to assign loop names, loops are assigned implicit names. Every loop is assigned the name of its loop counter variable, unless:

– it has a `#pragma loop id(..)` annotation,
– some other loop has an annotation with that name,
– or there is another loop counter variable with the same name.

For instance, fusing the loops in Listings 1.5a is legal, but the compiler should report an ambiguity in Listings 1.5b.

2.3 Transformations

In addition to the loop transformations mentioned in Table 1, there are many more transformations compilers could implement. Tables 2 and 3 contain some pragmas that could be supported.

Table 2 contains the directives that apply to loops, many of which change the order of execution. Transformations such as unrolling and unswitching do not affect the code semantics and therefore can always be applied. The additional assume- and expect-directives do not transform code, but give hints to the compiler about intended semantic properties.

Table 2. Directives on loops

Directive	Short description
`parallel for`	OpenMP thread-parallelism
`simd`	Vectorization
`unroll`	Loop unrolling
`split`	Index set splitting
`peel`	Loop peeling (special kind of index set splitting)
`specialize`	Loop versioning
`unswitch`	Loop unswitching (special kind of loop versioning)
`shift`	Add offset to loop counter
`scale`	Multiply loop counter by constant
`coalesce`	Combine nested loops into one
`concatenate`	Combine sequential loops into one
`interchange`	Permute order of nested loops
`stripmine`	Strip-mining
`block`	Like strip-mining, but with constant sized outer loop
`tile`	Tiling (combination of strip-mining and interchange)
`reverse`	Inverse iteration order
`distribute`	Split loop body into multiple loops
`fuse`	Loop Fusion/Merge: Inverse of loop distribution
`wavefront`	Loop skewing
`unrollandjam`	Unroll-and-jam/register tiling
`interleave`	Loop interleaving
`scatter`	Polyhedral scheduling
`curve`	Space-filling curve (Hilbert-, Z-curve or similar)
`assume_coincident`	Assume no loop-carried dependencies
`assume_parallel`	Assume parallelism
`assume_min_depdist`	Assume minimum dependence distance
`assume_unrelated`	Assume statements access only disjoint memory
`assume_termination`	Assume that a loop will eventually terminate
`expect_count`	Expect an average number of loop iterations

Most clauses are specific to a transformation (e.g. tile sizes), but some clauses apply to all transformations, such as:

`(no)assert`. Control whether the compiler has to abort with an error if, for any reason, a transformation can not be applied. The default is `noassert`, but the compiler may still warn about not applied transformations.

`noversioning`. Disable code versioning. If versioning is required to preserve the loop's semantics, do not apply the transformation unless `assume_safety` is

```
#pragma omp stripmine strip_size(2) \
        pit_id(outer) strip_id(inner)
for (int i=0; i<128; i+=1)
  Statement(i);
```

(a) Strip-mining pragma

```
#pragma omp id(outer)
for (int pit_i=0; pit_i<128; pit_i+=2)
  #pragma omp id(inner)
  for (int i=pit_i; i<pit_i+2; i+=1)
    Statement(i);
```

(b) Output loop

Listing 1.6: Strip-mining example

used as well. The combination `assert noversioning` can be used to ensure that the transformed code always runs instead of a some fallback version.

`assume_safety`. Assume that the transformation is semantically correct in all well-defined cases. This shifts the responsibility of correctness to the programmer. If the compiler was able to apply the transformation using code versioning, in general, this will remove the runtime checks.

`suggest_only`. By default, a transformation pragma overrides any profitability heuristic the compiler might use to apply a transformation. This clause can be used together with `assume_safety` to only imply that the transformation is semantically correct, but leave it to the profitability heuristic to decide whether to actually apply it, with only a bump in favor of applying the transformation and/or its parameters. The compiler might also apply a different transformation. For instance, `parallel for assume_safety suggest_only` implies that a loop is parallelizable, but the compiler might choose to vectorize it instead.

Directives with `assume_`-prefix inform the compiler that a property is always true. It is the programmer's responsibility to ensure that this the case and executions that violate the assumption have *undefined behavior*. Directives with an `expect_`-prefix suggest that compiler optimize the code assuming that the property likely applies. Executions that violate the expectation may run slower, but the behavior remains the same.

Directives in Table 3 apply to sections of code, which might be in, or include, loops. For instance, `assume_associative` and `assume_commutative` inform the compiler that a section behaves like a reduction, so reduction detection can apply to more than a fixed set of operators.

In the following we present a selected subset of these transformation in more detail.

Loop Strip-Mining/Blocking/Collapse/Interchange are *vertical* loop transformations, i.e., transformations on or to perfectly nested loops.

Strip-mining is the decomposition of a loop into an inner loop of constant size (called the *strip*) and an outer loop (which we call the *pit*) executing that inner loop until all loop bodies have been executed. For instance, the result of Listing 1.6a is the loop in Listing 1.6b.

The difference of loop blocking is that the pit's size is specified in a clause. However, the case if the iteration count is not known to be a multiple of the strip/block size requires a different handling.

Table 3. Directives on code, including loops

Directive	Short description
`id`	Assign an unique name
`parallel sections`	OpenMP multi-threaded section
`target`	Accelerator offloading
`ifconvert`	If-conversion
`reorder`	Execute code somewhere else
`pack`	Use copy of array slice in scope
`assume_associative`	Assume a calculation is associative
`assume_commutative`	Assume a calculation is commutative
`assume_disjoint_access`	Memory accesses do not alias
`assume_nooverflow`	Assume (unsigned) integer overflow does not occur
`assume_noalias`	Assume pointer ranges do not alias
`assume_dereferenceable`	Assume that a pointer range is dereferenceable
`expect_dead`	Expect that code in a branch is not executed

Collapsing is the reverse operation: Combine two or more vertical loops into a single loop. Only the case where the number of iterations does not depend on anything in the outer loop needs to be supported. The transformation is already available in OpenMP's `collapse`-clause of the `for`-pragma.

Interchange is permuting the order of perfectly nested (i.e. vertical) loops. Classically, only two loops are interchange, but we can generalize this to allow any number of loops as long as they are perfectly nested. In contrast to the previous transformations, interchange may change the execution order of iterations and therefore requires a legality check.

Using these transformations, other transformations can be constructed. For instance, tiling is a combination of strip-mining and interchange:

```
#pragma omp interchange permutation(outer_i,outer_j,inner_i,inner_j)
#pragma omp stripmine strip_size(4) pit_id(outer_i) strip_id(inner_i)
for (int i=0; i<128; i+=1)
  #pragma stripmine strip_size(4) pit_id(outer_j) strip_id(inner_j)
  for (int j=0; j<128; j+=1)
    Statement(i);
```

For convenience, we also propose a `tile`-transformation which is syntactic sugar for this composition.

Loop Distribution/Fusion/Reordering are *horizontal* transformations, i.e. apply on loops that execute sequentially (and may nest other loops).

Loop distribution splits a loop body into multiple loops that are executed sequentially. An example was already given in Listing 1.4. The opposite is loop fusion: Merge two or more loops into a single loop. The `reorder`-pragma changes the execution order of loops or statements.

```
#pragma omp split indices(i > n/2)
for (int i=0; i<n; i+=1)
  Statement(i);
```

(a) Loop split pragma

```
for (int i=0; i<=n/2; i+=1)
  Statement(i);
for (int i=n/2+1; i<n; i+=1)
  Statement(i);
```

(b) Transformed code

Listing 1.7: Index set splitting

```
#pragma omp loop(A,B) concatenate
#pragma omp id(A)
for (int i=0; i<n; i+=1)
  StatementA(i);
#pragma omp id(B)
for (int i=0; i<n; i+=1)
  StatementB(i);
```

(a) Loop concatenation pragma

```
for (int i=0; i<n+n; i+=1) {
  if (i < n)
    StatementA(i);
  else
    StatementB(i-n);
}
```

(b) Transformed code

Listing 1.8: Index set concatenation

Loop Counter Shifting/Scaling/Reversal modifies the iteration space that the compiler associates with a loop. By itself, this does not do anything, but might be required for other transformations, especially loop fusion. By default, loop fusion would match iterations that have the same numeric value into the same iteration of the output loop. If different iterations should be executed together, then the iteration space must be changed such that the instances executed together have the same numeric value.

The scaling transformation only allows positive integer factors. A factor of negative one would reverse the iteration order which accordingly we call loop reversal. It may change the code's semantics and therefore requires a validity check.

Index Set Splitting/Peeling/Concatenation modify a loop's iteration domain. They are *horizontal* loop transformations. Index set splitting creates multiple loops, each responsible for a fraction of the original loop's iteration domain. For instance, the result of Listing 1.7a is shown in Listing 1.7b.

The difference of *loop peeling* is that the split loop is specified in number of iterations at the beginning or end of the original loop. Therefore, it can be seen as syntactical sugar for index set splitting.

Loop concatenation is the inverse operation and combines the iteration space of two or more consecutive loops. For instance, the result of Listing 1.8a is Listing 1.8b.

2.4 Syntax

The generalized loop transformation syntax we propose is as follows:

`#pragma omp` [`loop(`*loopnames*`)`] *transformation option*`(`*argument*`)` *switch* ...

The optional `loop` clause can be used to specify which loop the transformation applies on. How many loops can be specified depends on the transformation. If the clause is omitted, the transformation applies on the following horizontal or vertical loops, depending on the transformation. Alternatively, a number specified to mean the next n vertical or horizontal loops, like OpenMP's `collapse`-clause.

OpenMP's current `simd` and `for` constructs require canonical for-loops, but implementations may lift that restriction to support more kinds of transformable loops, e.g. while-loops.

3 Implementation

To serve as a proof-of-concept, we are working on an implementation in Clang. Independently from this proposal, we also want to improve Clang/LLVM's loop transformations capabilities to make it useful for optimizing scientific high-performance applications.

3.1 Front-End: Clang

Clang's current general loop transformation syntax, also shown in Table 1, is

`#pragma clang loop` *transformation*`(`*option*`)` *transformation*`(`*option*`)` ...

and therefore differs from the proposed OpenMP syntax: the first keywords (`omp` vs. `clang loop`), but also for the transformations themselves. Multiple options can be given by using different variants of the same transformation at the same time, which is ambiguous when composing transformations.

Instead, we implement a hybrid of both syntaxes, which is:

`#pragma clang loop`[`(`*loopnames*`)`] *transformation option*`(`*argument*`)` *switch* ...

The current syntax still needs to be supported for at least the transformations currently possible with Clang.

Clang's current architecture has two places where loop transformations occur, shown in Fig. 1.

1. OpenMP's `parallel for` is implemented at the front-end level: The generated LLVM-IR contains calls to the OpenMP runtime.
2. Compiler-driven optimizations are implemented in the mid-end: A set of transformation passes that each consume LLVM-IR with loops and output transformed IR, but metadata attached to loops can influence the passes' decisions.

This split unfortunately means that OpenMP-parallel loops are opaque to the LLVM passes further down the pipeline. Also, loops that are the result of other transformations (e.g. LoopDistribute) cannot be parallelized this way because it must have happened before. An exception is, `#pragma omp simd` which just annotates a loop inside the IR using `llvm.loop.vectorize.enable` to be processed by the LoopVectorizer pass.

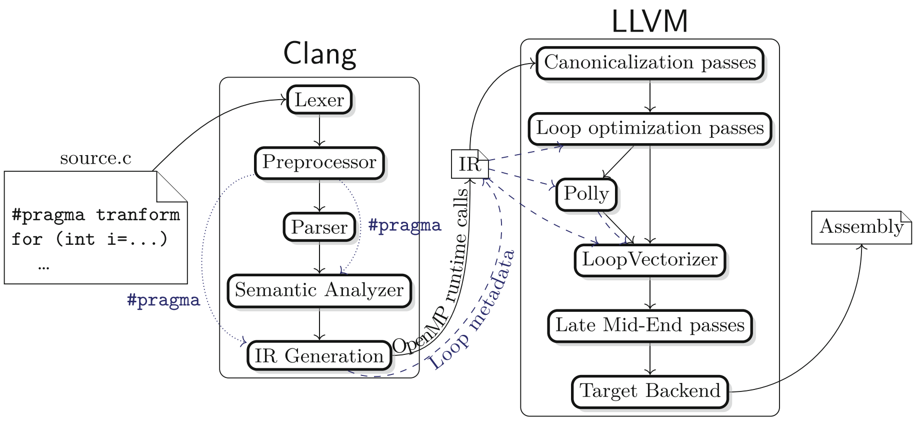


Fig. 1. Clang compiler pipeline

Multiple groups are working on improving the situation by adding parallel semantics to the IR specification [28,29]. These and other approaches have been presented on LLVM's mailing list [5,10] or its conferences [9,27]. Until Clang's implementation of OpenMP supports generating parallel IR, we require users to use a different pragma if they want the mid-end to apply thread-parallelism. In Clang's case, this is `#pragma clang loop parallelize_thread`.

3.2 LLVM-IR Metadata

Only existing loops can be annotated using the current metadata, but not loops that result from other transformations. In addition, there is no transformation order and at most one instance of a transformation type and can be specified. Therefore, a new metadata format is required.

Our changes use loop annotations only to assign loop names. The transformation themselves are instead a sequence of metadata associated with the function containing the loop. Each transformation has to lookup the loop it applies on using the result of the previous transformations.

Current passes that consume the current metadata need to be modified to read the changed format instead. Due to their fixed order in the pass pipeline however, they can only apply on loops that originate in the source or are the result of passes that execute earlier in the pipeline.

3.3 Loop Transformer: Polly

Polly [13] takes LLVM-IR code and 'lifts' is into another representation – *schedule trees* [32] – in which loop transformations are easier to express. To transform loops, only the schedule tree needs to be changed and Polly takes care for the remainder of the work.

We can implement most transformations from Table 2 as follows. First, let Polly create a schedule tree for a loop nest, then iteratively apply each transformation in the metadata to the schedule tree. For every transformation we can check whether it violates any dependencies and act according to the chosen policy. When done, Polly generates LLVM-IR from the schedule tree including code versioning.

If desired, Polly can also apply its loop nest optimizer which utilizes a linear program solver before IR generation. We add artificial *transformational dependencies* to ensure that user-defined transformations are not overridden.

4 Evaluation

Although we intended this to be a proposal for further discussion, and hence do not have a complete implementation yet, we can measure what the effects of such pragmas are. Figure 2 shows the execution times of a single thread double precision matrix-multiplication kernel ($M = 2000, N = 2300, K = 2600$).

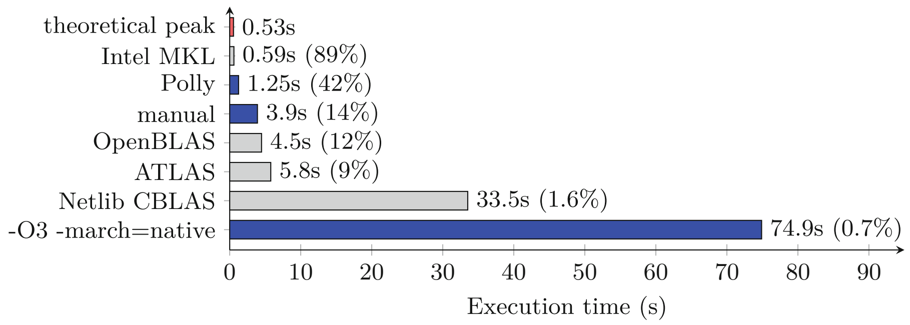


Fig. 2. Comparison of matrix-multiplication execution times on an Intel Core i7 7700HQ (Kaby Lake architecture), 2.8 GHz, Turbo Boost off

The naïve version (Listing 1.1 without pragmas) compiled with Clang 6.0 executes in 75 s (gcc's results are similar); Netlib's reference CBLAS implementation in less than half that time. With the pragma transformations manually applied the execution time shrinks to 3.9 s. The same transformations as automatically applied by Polly runs in 1.14 s, which is 42% of the processor's theoretical floating-point limit. LLVM's loop vectorizer currently only supports vectorizing inner loops, so we applied an additional unroll-and-jam step in the manual version. Polly instead prepares its output for the SLP-vectorizer, which may explain the performance difference.

By comparison, OpenBLAS and ATLAS both reach similar results with 4.5 and 5.8 s. Their binaries were obtained from the Ubuntu 16.04 software repository, therefore are likely not optimized for the platform. Intel's MKL library runs in 0.59 s, which is 89% of the theoretical flop-limited peak performance.

5 Related Work

As already mentioned in Table 1, many compilers already implement pragmas to influence their optimization passes. The most often implemented transformation is loop unrolling, for instance in gcc since version 8.1 [11]. The most advanced transformation we found is xlc's `#pragma block_loop`. It is the only transformation that uses loop names which might have been introduced only for this purpose. The compiler manual mentions special cases where it is allowed to compose multiple transformations, but in most cases it result in only one transformation being applied or a compiler error [16].

Multiple research groups already explored the composition of loop transformations, many of them based on the polyhedral model. The Unifying Reordering Framework [20] describes loop transformations mathematically, including semantic legality and code generations. The Clint [34] tool is able to visualize multiple loop transformations.

Many source-to-source compilers can apply the loop transformations themselves and generate a new source file with the transformation baked-in. The instructions of which transformations to apply can be in the source file itself like in a comment of the input language (Clay [1], Goofi [23], Orio [14]) or like our proposal as a pragma (X-Language [7], HMPP [6]). Goofi also comes with a graphical tool with a preview of the loop transformations. The other possibility is to have the transformations in a separate file, as done by URUK [12] and CHiLL [30]. POET [33] uses an XML-like description file that only contains the loop body code in the target language.

Halide [26] and Tensor Comprehensions [31] are both libraries that include a compiler. In Halide, a syntax tree is created from C++ expression templates. In Tensor Comprehensions, the source is passed as a string which is parsed by the library. Both libraries have objects representing the code and calling its methods transform the represented code.

Similar to the parallel extensions to the C++17 [19] standard library, Intel's Threading Building Blocks [17], RAJA [15] and Kokkos [8] are template libraries. The payload code is written using lambdas and an *execution policy* specifies how it should be called.

Our intended use case – autotuning loop transformations – has also been explored by POET [33] and Orio [14].

6 Conclusion

We propose adding a framework for general loop transformation to the OpenMP standard. Part of the proposal are a set of new loop transformations in addition to the already available thread-parallelization (`#pragma omp for`) and vectorization (`#pragma omp simd`). Some of these have already been implemented using compiler-specific syntax and semantics. The framework allows arbitrarily composing transformation, i.e. apply transformations on already transformed loops.

Loops – existing in the source code as well as the loop resulting from transformations – can be assigned unique identifiers such that the pragmas can be applied on already transformed loops.

Experiments show speedups comparable to hand-optimized libraries without the cost in maintainability. We started implementing the framework using a different syntax in Clang/LLVM using the Polly polyhedral optimizer to carry out the transformations.

The proposal is not complete in that it does not specify every detail a specification would have. As with any proposal, we are looking for feedback from other groups including about applicability, syntax, available transformations and compatibility/consistency with the current OpenMP standard.

Acknowledgments. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nations exascale computing imperative.

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

1. Bagnères, L., Zinenko, O., Huot, S., Bastoul, C.: Opening polyhedral compiler's black box. In: 14th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2016). IEEE (2016)
2. Attributes in Clang. http://clang.llvm.org/docs/AttributeReference.html
3. Auto-Vectorization in LLVM. http://llvm.org/docs/Vectorizers.html
4. Clang Language Extensions. http://clang.llvm.org/docs/LanguageExtensions.html
5. Doerfert, J.: [RFC] abstract parallel IR optimizations. llvm-dev mailing list post, June 2018. http://lists.llvm.org/pipermail/llvm-dev/2018-June/123841.html
6. Dolbeau, R., Bihan, S., Bodin, F.: HMPP^{TM}: a hybrid multi-core parallel programming environment. In: First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007) (2007)
7. Donadio, S., et al.: A language for the compact representation of multiple program versions. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 136–151. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-69330-7_10
8. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014)
9. Finkel, H., Doerfert, J., Tian, X., Stelle, G.: A parallel IR in real life: optimizing OpenMP. EuroLLVM 2018 presentation (2018). http://llvm.org/devmtg/2018-04/talks.html#Talk_1
10. Finkel, H., Tian, X.: [RFC] IR-level region annotations. llvm-dev mailing list post, January 2017. http://lists.llvm.org/pipermail/llvm-dev/2017-January/108906.html

11. Free Software Foundation: Loop-Specific Pragmas. https://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html
12. Girbal, S., et al.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Int. J. Parallel Program. **34**(3), 261–317 (2006)
13. Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., Pouchet, L.N.: Polly - polyhedral optimization in LLVM. In: First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) (2011)
14. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using Orio. In: Proceedings of the 23rd IEEE International Parallel and Distributed Computing Symposium (IPDPS 2009). IEEE (2009)
15. Hornung, R.D., Keasler, J.A.: The RAJA portability layer: overview and status. Technical report LLNL-TR-661403, Lawrence Livermore National Lab (2014)
16. IBM: Product documentation for XL C/C++ for AIX, V13.1.3
17. Intel: Threading Building Blocks. https://www.threadingbuildingblocks.org
18. Intel: Intel C++ Compiler 18.0 Developer Guide and Reference, May 2018
19. International Organization for Standardization: ISO/IEC 14882:2017, December 2017
20. Kelly, W., Pugh, W.: A framework for unifying reordering transformations. Technical report UMIACS-TR-93-134/CS-TR-3193, University of Maryland (1992)
21. Low, T.M., Igual, F.D., Smith, T.M., Quintana-Orti, E.S.: Analytical modeling is enough for high-performance BLIS. Trans. Math. Softw. (TOMS) **43**(2), 12:1–12:18 (2016)
22. Microsoft: C/C++ Preprocessor Reference. http://docs.microsoft.com/en-us/cpp/preprocessor/loop
23. Müller-Pfefferkorn, R., Nagel, W.E., Trenkler, B.: Optimizing cache access: a tool for source-to-source transformations and real-life compiler tests. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 72–81. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27866-5_10
24. OpenACC-Standard.org: The OpenACC Application Programming Interface Version 4.0, November 2017
25. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.0, July 2017
26. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013), pp. 519–530. ACM (2013)
27. Saito, H.: Extending LoopVectorizer towards supporting OpenMP4.5 SIMD and outer loop auto-vectorization. EuroLLVM 2018 presentation (2016). http://llvm.org/devmtg/2016-11/#talk7
28. Schardl, T.B., Moses, W.S., Leiserson, C.E.: Tapir: embedding fork-join parallelism into LLVM's intermediate representation. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2017), pp. 249–265. ACM (2017)
29. Tian, X., et al.: LLVM framework and IR extensions for parallelization, SIMD vectorization and offloading. In: Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC 2016). IEEE (2016)
30. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.K.: A scalable autotuning framework for compiler optimization. In: Proceedings of the 23rd IEEE International Parallel and Distributed Computing Symposium (IPDPS 2009). IEEE (2009)

31. Vasilache, N., et al.: Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. CoRR abs/1802.04730 (2018)
32. Verdoolaege, S., Guelton, S., Grosser, T., Cohen, A.: Schedule trees. In: Fourth International Workshop on Polyhedral Compilation Techniques (IMPACT 2014) (2014)
33. Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.: POET: parameterized optimizations for empirical tuning. In: Proceedings of the 21st IEEE International Parallel And Distributed Computing Symposium (IPDPS 2007). IEEE (2007)
34. Zinenko, O., Huot, S., Bastoul, C.: Clint: a direct manipulation tool for parallelizing compute-intensive program parts. In: 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE (2014)

Extending OpenMP to Facilitate Loop Optimization

Ian Bertolacci[1(✉)], Michelle Mills Strout[1], Bronis R. de Supinski[2], Thomas R. W. Scogland[2], Eddie C. Davis[3], and Catherine Olschanowsky[3]

[1] The University of Arizona, Tucson, AZ 85721, USA
ianbertolacci@email.arizona.edu, mstrout@cs.arizona.edu
[2] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{bronis,scogland1}@llnl.gov
[3] Boise State University, Boise, ID 83725, USA
eddiedavis@u.boisestate.edu, catherineolschan@boisestate.edu

Abstract. OpenMP provides several mechanisms to specify parallel source-code transformations. Unfortunately, many compilers perform these transformations early in the translation process, often before performing traditional sequential optimizations, which can limit the effectiveness of those optimizations. Further, OpenMP semantics preclude performing those transformations in some cases prior to the parallel transformations, which can limit overall application performance.

In this paper, we propose extensions to OpenMP that require the application of traditional sequential loop optimizations. These extensions can be specified to apply before, as well as after, other OpenMP loop transformations. We discuss limitations implied by existing OpenMP constructs as well as some previously proposed (parallel) extensions to OpenMP that could benefit from constructs that explicitly apply sequential loop optimizations. We present results that explore how these capabilities can lead to as much as a 20% improvement in parallel loop performance by applying common sequential loop optimizations.

Keywords: Loop optimization · Loop chain abstraction
Heterogeneous adaptive worksharing · Memory transfer pipelining

1 Introduction

Efficient use of the complex hardware commonly available today requires compilers to apply many optimizations. OpenMP already supports many of these optimizations, such as offloading regions to accelerators like GPUs and the parallelization of code regions through threads and tasks. However, OpenMP currently ignores the large space of sequential optimizations, such as loop fusion, loop fission, loop unrolling, tiling, and even common subexpression elimination.

One might argue that the compiler can automatically apply traditional sequential optimizations. This approach has worked reasonably well for sequential code regions. However, the space of sequential optimizations, when to apply them, and in what order is complex and OpenMP parallelization complicates it further. Thus, hints, or even prescriptive requirements, from the application programmer could substantially reduce the complexity of the compiler's navigation of that space.

More importantly, the semantics of several existing OpenMP constructs preclude the use of some sequential optimizations prior to the transformations represented by those constructs. Many proposed OpenMP extensions, such as ones to support pipelining [3] and worksharing across multiple accelerators [7,9], increase these restrictions. Others, such as loop chaining [1,2] complicate the optimization space further. Thus, the ability to specify sequential optimizations and when to apply them could substantially improve the compiler's ability to exploit the complex hardware in emerging systems.

A key issue is supplying sufficient data to keep the computational units busy. Sequential optimizations frequently improve arithmetic intensity and memory performance. OpenMP parallelization can also improve memory performance by reducing the memory footprint per thread and thus freeing CPU cache for other data. Many proposed data and pipelining optimizations also address these issues. Often the programmer understands which optimizations to apply and the best order in which to apply them. Other times, the programmer can easily determine that the order in which they are applied will not impact code correctness. Thus, OpenMP extensions should allow the programmer to provide this information to the compiler.

This paper makes the following contributions:

- an initial review of limitations that existing OpenMP constructs and proposed extensions impose on the use of sequential optimizations;
- proposed syntax to allow OpenMP programmers to request the use of sequential optimizations; and
- a discussion of the performance implications, including an initial experimental study, of the current limitations on and the proposed support for sequential optimizations.

Overall we find that flexible support to specify sequential optimizations could use existing compiler capabilities to produce significantly more efficient code.

The remainder of the paper is structured as follows. In the next section, we discuss the limitations that OpenMP semantics place on the use of traditional sequential optimizations. Section 3 then presents our proposed syntax for an initial set of sequential optimizations to include directly in OpenMP. We then present an initial performance study in Sect. 4 that demonstrates that these sequential extensions could provide significant performance benefits.

2 Existing Limitations

Existing OpenMP constructs can require that their semantics be instantiated prior to performing several important sequential optimizations. These requirements can limit the scope and the effectiveness of those optimizations. Other

```
#pragma omp for schedule(static, 1) nowait
for( int i = 0; i < n; ++i )
  A[i] += B[i] * c;
```

Fig. 1. A simple static loop, with a chunk size that interferes with optimization

optimizations may be complicated by the order in which a specific OpenMP implementation interprets OpenMP semantics and applies sequential optimizations. In this section, we discuss common subexpression elimination, unrolling, pipelining, CoreTSAR, and loop chaining.

2.1 Common Subexpression Elimination

A simple example that we do not address in this work is common subexpression elimination (CSE). For CSE, the optimization could take the form of code hoisting to outside of an OpenMP parallel region and creation of firstprivate copies that are stored in registers for each thread. CSE would be performed before parallelization.

2.2 Unrolling

Of particular interest in the context of our proposed extensions are limitations implied by the loop construct. Consider, for example the loop unrolling optimization. In Fig. 1, the specification of a static schedule with a chunk size parameter of one implies that unrolling cannot be performed prior to implementing the schedule. Specifically, since OpenMP requires iterations to be distributed in chunks of the given size, a single thread cannot be provided with any more than one consecutive iteration in this case.

2.3 Pipelining

The expression of data access patterns can be leveraged to allow for pipelining of data transfers and computation in loop offload. This is a topic that we have explored before in the context of pipelining individual loops [3]. Given annotations of the data access pattern of a given iteration in a loop nest, the implementation can implement multiple buffering and an overlapped pipeline of data transfers and computation for the user. Figure 2 shows a simple example of a stencil code, which uses one possible way to describe the data access pattern, by using the iteration variable to split on the `pipeline_map` clause. Specifically, referring to the induction variable `k` in the `pipeline_map` clause expands to a reference to all iterations of the range of iterations from one to `nz-1` as defined by the loop.

A weakness of the previously proposed approach is that it cannot easily be extended to work for asynchronous loops with dependencies. It must be synchronous with respect to other loops at least, if not with respect to host execution. Since the first loop must fully complete before the next can start, loop

```
#pragma omp target \
  pipeline(static[1,3])\
  pipeline_map(to:A0[k-1:3][0:ny-1][0:nx-1])\
  pipeline_map(from:Anext[k:1][0:ny-1][0:nx-1])\
  pipeline_mem_limit(MB_256)
for(k=1;k<nz-1;k++) {
#pragma omp target teams distribute parallel for
  for(i=1;i<nx-1;i++) {
    for(j=1;j<ny-1;j++) {
      Anext[Index3D (i,     j,     k)] =
        (A0[Index3D (i,     j,     k + 1)] +
         A0[Index3D (i,     j,     k - 1)] +
         A0[Index3D (i,     j + 1, k)] +
         A0[Index3D (i,     j - 1, k)] +
         A0[Index3D (i + 1, j,     k)] +
         A0[Index3D (i - 1, j,     k)])*c1
       - A0[Index3D (i,     j,     k)]*c0;
    } }
}
```

Fig. 2. An example stencil kernel using pipelining.

fusion is effectively impossible, at least unassisted. Similarly pipelining complicates tiling since both modify the loop bounds, possibly dynamically. Given extensions to express how the data flows from one loop to the next, this kind of pipelining might be applied to multiple loops in sequence without having to complete one loop in full before beginning the next.

2.4 CoreTSAR

As with the pipelining extensions, CoreTSAR [7,9], the Core Task Size Adapting Runtime, leverages data access pattern information to coschedule loops across different resources such as CPUs and GPUs or coprocessors. The main abstraction is a loop iteration mapping to the accesses made by any given iteration. Figure 3 shows a simple partitioning of a GEMM kernel by its outer loop, splitting the loop by rows according to the `i` index. CoreTSAR uses a somewhat more abstract syntax to describe the data access pattern. Its `hetero` clause selects the devices and schedule to use, and defines that the associated loop over `i` should be split. Each `part_map` clause overloads the array section syntax to represent `split by loop iter:length`.

The information provided by the user, along with the usual loop trip count information, feeds into a scheduler that selects how much of the given iteration range should be provided to each device, and distributes the data accordingly. This scheduler can be any of a wide range, but the `static` and `adaptive` schedules are most common. The `static` schedule is similar to the `static` schedule on the loop construct but adjusted proportionally by the performance of

```
void runGemm(T **a_a, T **b_a, T **c_a) {
  T *a = *a_a, *b = *b_a, *c = *c_a;
#pragma omp target teams distribute parallel \
         for part_map(c[1:N][0:N])          \
         part_map(to: a[1:N][0:N]) map(to: b[0:N*N])\
         hetero(1, all, adaptive, default, 10)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; ++j) {
      c[(i * N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
      }
    }
  }
}
```

Fig. 3. An example GEMM kernel with CoreTSAR.

the hardware targets, while `adaptive` starts out as static and uses a linear-optimization approach to attempt to complete the work given to each device in as close to the same amount of time as possible while minimizing time. The resulting code precludes loop fusion for the same reason as pipelining does. If the loop transformations to generate the dynamically sized inner chunks are applied before sequential optimizations, tiling is also effectively precluded since the loop distribution extension would require information about the tiling to be correct.

2.5 Loop Chain Abstraction and Optimization

A *loop chain* is N $(N > 1)$ loop nests with no code between them that explicitly share data. Figure 4 shows an example of a simple loop chain. These are common in stencil applications and present an opportunity for both data-reuse optimizations and temporary-storage optimizations. The loop chain abstraction [5] represents a loop chain as a sequence of loops domains $L_1, L_2, \ldots, L_N$, well defined data space domains $D_1, D_2, \ldots, D_M$, read access functions $Read_{l,d} : L_l \rightarrow D_d$, and write access functions $Write_{l,d} : L_l \rightarrow D_d$. This model of a loop chain provides a framework for developing and implementing scheduling and storage optimizations between loops.

The key optimization applied to loop chains is loop fusion, which is used in conjunction with other loop optimizations such as tiling and wavefront parallelism. However these optimizations can be difficult to apply manually to complex scientific codes, obfuscate the primary computation, and may not be portable. Further, there are many combinations of possible optimizations, including the parameterization of tile sizes; a developer would need significant expertise to know ahead of time which optimizations would be useful.

```
for( int i = lb; i <= ub; i += 1 )
  A[i] = (B[i-1] + B[i] + B[i+1]);

for( int i = lb; i <= ub; i += 1 )
  A[i] = A[i] * (1.0/3.0);
```

Fig. 4. Example of simple loop chain.

The loop chain abstraction allows automated loop fusion, tiling, and parallel wavefronting of loop chains through a source-andto-source compiler [1,2]. A major contribution of that work is an OpenMP-style annotation language that allows the developer to state explicitly the loop chain domains, access functions, and a list of desired optimizations to be applied to the loop chain. Figure 5 shows how the loop chain in Fig. 4 would be annotated with this language. The separation of the `schedule(..)` specification in only one location for the entire loop chain enables the use of autotuning across different potential schedules.

The annotations allow the developer to identify the entire loop chain, each loop nest of the chain, the domains of the loops, and the read and write access functions. The annotations can also be applied to sequences of inlineable functions whose bodies contain the loops and accesses. The annotations then provide the transformation tool information about the code (loop domains and data access patterns) required to perform the optimizations, replacing the need for complicated analysis. Additionally, loop chain scheduling operators (`fuse`, `tile`, `wavefront`, `serial`, and `parallel`) allow the developer to list specific optimizations to apply to the loop chain. Combined, the loop chain and related annotations support easy application of complicated optimizations on new and existing code without requiring cumbersome and unnecessary rewriting and redesigning of the applications. These annotations can be written by non-experts, who can easily change the optimizations that are applied. Thus, the developer can use these optimizations and experiment to find the fastest schedule by changing the scheduling annotations and not the application code.

3 Sequential Optimizations

In this section, we discuss some of the most effective sequential optimizations for scientific codes: loop fusion, tiling, and unrolling. Each have been implemented with pragmas in various tools.

3.1 Fusion

Loop fusion is a common optimization that takes two or more loop bodies and combines them into one. The loop domains may also be shifted to respect data-dependencies that occur between the loops. Loop fusion improves caching behavior by reducing the distance between when data is produced and when it is used.

```
#pragma omplc loopchain schedule( ... )
{
  #pragma omplc for domain(lb:ub) \
     with (i) \
        write A {(i)}, \
        read  B {(i-1),(i),(i+1)}
   for( int i = lb; i <= ub; i += 1 )
     A[i] = (B[i-1] + B[i] + B[i+1]);

  #pragma omplc for domain(lb:ub) \
     with (i) \
        write A {(i)}, \
        read  A {(i)}
   for( int i = lb; i <= ub; i += 1 )
     A[i] = A[i] * (1.0/3.0);
}
```

Fig. 5. Example of annotated source code (schedule omitted)

```
for( int i = lb; i <= ub; i += 1 ){
  A[i] = (B[i-1] + B[i] + B[i+1]);
  A[i] = A[i] * (1.0/3.0);
}
```

Fig. 6. Example of a fused loop chain using `schedule( fuse() )`

Loop fusion can also enable reduction of the amount of temporary storage that a computation requires.

In the loop chain transformation framework [1,2], the `fuse()` scheduling operator specifies the fusion of all loops in the loop chain. The tool uses the read/write information to form a dependency graph and to find the smallest shifts required to make fusion legal. Alternatively, the developer can explicitly specify the shifts that they require. Figure 6 shows the result of the schedule `fuse()` applied to the example loop chain in Fig. 5.

3.2 Tiling

Tiling is a common optimization that breaks a loop into contiguous chunks [4, 12,13]. Tiling reduces reuse distances in order to improve caching behavior.

In the loop chain transformation framework [1,2], the `tile( )` scheduling operator specifies tiling of the loops in the loop chain. This operator has three required arguments. First, a tuple indicates the tile-size in each dimension. Currently, tile-size is not parameterizable (a current limitation of the polyhedral code generator, ISL [11]), and requires a constant value. The tiling can be of fewer dimensions than the loop being tiled (for example, strip mining is a one dimensional tiling of a two-or-more dimension space). The second argument to

```
#pragma omp parallel
for( int io = lb; io <= ub; io += 10 ){
  for( int i = io; i <= io+10; i += 1 ){
    A[i] = (B[i-1] + B[i] + B[i+1]);
  }
  for( int i = io; i <= io+10; i += 1 ){
    A[i] = A[i] * (1.0/3.0);
  }
}
// clean-up loop
```

Fig. 7. Tiled loop chain with `schedule( tile( (10), parallel, serial ) )`

the tiling operator specifies the schedule to apply *over* the tiles. The current tool supports several scheduling operators: `tile`; `wavefront`; `serial`; and `parallel`. For example, using the `serial` operator over the tiles would lead to each tile being visited serially, whereas using the `parallel` operator allow each tile to be visited in parallel (with no guarantee of order). Similarly, the third argument to the tiling operator is the schedule that is applied *within* a tile; it can also be any one of the scheduling operators (`tile`, `wavefront`, `serial`, or `parallel`). For example, using the `serial` operator within a tile would lead to each each point *within* a tile being visited serially, whereas using the `parallel` operator would allow all points in a tile to be visited in parallel (with no guarantee of order). Figure 7 show the result of the schedule `tile( (10), parallel, serial )`.

3.3 Unrolling

Many compilers support loop unrolling. Unfortunately no common mechanism for invoking loop unrolling is consistently available. For example, the Intel and IBM XL compilers accept `pragma unroll(n)`. On the other hand, gcc supports `pragma GCC unroll n`. In contrast, the PGI and Microsoft C compilers do not offer any unrolling pragma.

The state of standardization for unrolling directives is similar to the range of non-portable parallel directives before OpenMP, and extending OpenMP to support it should carry the same overall benefit. Adding a new `unroll(n)` clause to the loop directive or the same as a loop chain operator can deliver this behavior portably. As a simple example, Fig. 8 shows a simple loop with the unrolling annotation and the result of unrolling it.

3.4 Interaction Between Optimizations

In the loop chain source-to-source transformation framework, each scheduling operator produces some function mapping from one polyhedral space to another. These functions are composed together when performing the complete transformation. In the more general case, any of the optimization operators or clauses

```
// Annotated
#pragma omp for unroll(2) schedule(static, 1) nowait
for( int i = 0; i < n; ++i )
  A[i] += B[i] * c;
// Expanded
#pragma omp for schedule(static, 1) nowait
for( int i = 0; i < n; i+=2 ) {
  A[i] += B[i] * c;
  A[i+1] += B[i+1] * c;
}
```

Fig. 8. Example of applying an unrolling clause to the loop directive.

Table 1. Descriptions for each of the execution schedules presented.

Legend label	Description
Baseline	Original implementation, series of loops
Fuse	Loop fusion
Tiled 8X8X8	Loop fusion then tile
Tiled 16X16X16	Loop fusion then tile
Tiled 32X32X32	Loop fusion then tile

that we have discussed could be composed, and in some cases even repeated. For example, a loop might be tiled for one size, parallelized, then tiled again for an inner cache on a given core. How these optimizations compose together is beyond the scope of this paper.

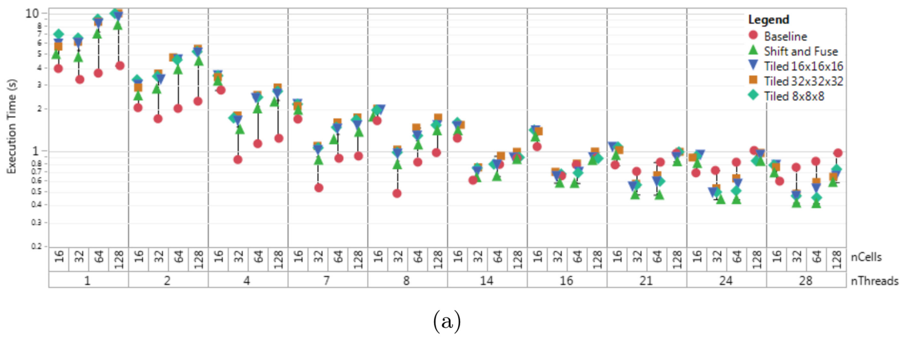
4 Experimental Results

Current results with the proposed loop chain optimizations [2] indicate that stencil applications optimized with loop chain optimizations perform better than the baseline at high thread counts.

4.1 Loop Fusion and Tiling

MiniFluxDiv[1] captures a subset of the stencil computations implemented in PDE solvers for computational fluid dynamics simulations. It was developed to emulate the behavior of the shared memory portion of code in the Chombo framework [6]. A series of sequential optimizations including loop fusion, iteration space shifting followed by loop fusion, and tiling were applied. The schedules presented here are described in Table 1.

[1] The code for the mini-flux-div benchmark can be found in the Variations on a Theme [10] benchmark repository.



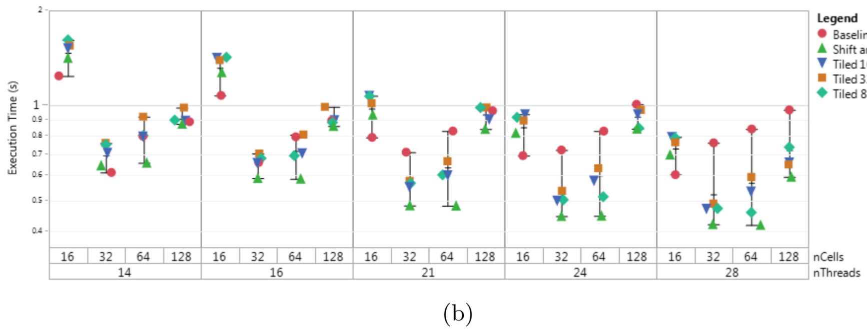


Fig. 9. Experimental results of Mini-Flux-Div (a stencil CFD code) micro-benchmark, (a) overall performance results, (b) zoomed view of results for threads 14 through 28

These experiments were run on a single node of the multi-node R2 cluster, at Boise State University. Each node contains a dual socket, Intel Xeon E5-2680 v4 CPU at 2.40 GHz clock frequency with 28 cores (14 per socket).

The sequential optimizations improves performance the most with larger domains and larger core counts. Figure 9 shows that shared memory scaling was improved for the cases with larger domains (128^3). This improvement impacts the full application performance as the interprocess communication can be reduced when larger shared memory domains are utilized.

The fusion optimizations that loop chains can enable also address some of the issues that arise with abstractions that produce many workshared loops nearly in direct contact [8]. They provide an easy way to eliminate the overheads of extra barriers and joins.

4.2 Loop Unrolling

The unrolling optimization can benefit non-chained loops as well as chains. To explore those benefits, we modify the clompk benchmark from the CORAL2 benchmark suite, which measures OpenMP overheads for execution patterns that are roughly consistent with a sparse matrix hydrodynamics code. We produced three unrolling factors for each loop: no unrolling; unroll by two; and unroll by four. We compile with OpenMP in parallel, and separately without any OpenMP directives. Figure 10 presents the performance results.

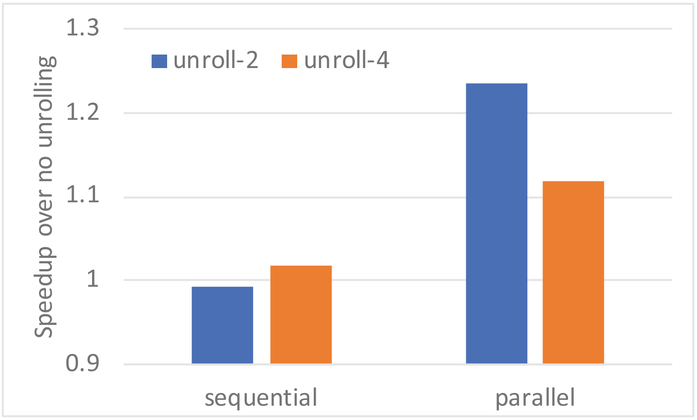


Fig. 10. Performance of clompk with and without unrolling applied.

For the sequential loop variant, an unroll factor of two produces a slowdown of about one percent, while a factor of four yields a speedup of a little more than one percent. The differences there are nearly in the noise, and show little change from the baseline. On the other hand, the OpenMP loops speed up by over 23% with an unroll factor of two, and almost 12% for a factor of four. These differences underscore a key point of this paper: manual unrolling makes almost no difference for serial execution since the compiler can accurately determine the degree to which to perform the transformation. However, manual unrolling can provide substantial benefit in the context of OpenMP. Thus, the version of clompk with OpenMP active runs 10% slower with one thread than the version compiled without OpenMP.

5 Conclusion

Modern systems are complex. They utilize powerful hardware and diverse architectures. Optimizations on loops, such as heterogeneous adaptive worksharing, memory transfer pipelining, and loop chain scheduling, are necessary to achieve the performance offered by these advanced architectures. However, these, and other necessary optimizations, are difficult to implement in the compiler while maintaining all required invariants of parallel programming models like OpenMP, and can be quite costly if added manually by the developer.

While the scope of OpenMP has traditionally been limited to an easy-to-use API for adding parallelism, we argue that these sequential optimizations could be made more accessible and portable by extending OpenMP to support them. The proposed extensions follow the OpenMP model by providing developers pragmas that explicitly prescribe specific optimizations. We combined them with descriptive information that gives the compiler the necessary information, particularly data access patterns, to perform the optimizations legally and efficiently.

Some of these extensions require the compiler to be more intelligent. For example, the `fusion` optimization in loop chaining requires the compiler to determine the required loop shifts to make fusion legal for stencil computations. However, this intelligence is augmented by the descriptive information provided by the annotations. Thus, developers can easily apply advanced and complex optimizations to their application without restricting their ability to maintain and to improve the application around and beyond these optimizations.

References

1. Bertolacci, I.J., Strout, M.M., Guzik, S., Riley, J., Olschanowsky, C.: Identifying and scheduling loop chains using directives. In: Proceedings of the Third International Workshop on Accelerator Programming Using Directives, pp. 57–67. IEEE Press (2016)
2. Bertolacci, I.J., Strout, M.M., Riley, J., Guzik, S.M., Davis, E.C., Olschanowsky, C.: Using the loop chain abstraction to schedule across loops in existing code. Int. J. High Perform. Comput. Netw. (To be published)
3. Cui, X., Scogland, T.R., de Supinski, B.R., Feng, W.C.: Directive-based partitioning and pipelining for graphics processing units. In: International Parallel and Distributed Processing Symposium, pp. 575–584. IEEE (2017)
4. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of the 15th Annual ACM SIGPLAN Symposium on Priniciples of Programming Languages, pp. 319–329 (1988)
5. Krieger, C.D., et al.: Loop chaining: a programming abstraction for balancing locality and parallelism. In: Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), May 2013
6. Olschanowsky, C., Strout, M.M., Guzik, S., Loffeld, J., Hittinger, J.: A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In: The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November 2014
7. Scogland, T.R.W., Feng, W., Rountree, B., de Supinski, B.R.: CoreTSAR: core task-size adapting runtime. IEEE Trans. Parallel Distrib. Syst. **26**, 2970–2983 (2015)
8. Scogland, T.R.W., Gyllenhaal, J., Keasler, J., Hornung, R., de Supinski, B.R.: Enabling region merging optimizations in OpenMP. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 177–188. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24595-9_13
9. Scogland, T.R.W., Feng, W., Rountree, B., de Supinski, B.R.: CoreTSAR: adaptive worksharing for heterogeneous systems. In: Kunkel, J.M., Ludwig, T., Meuer, H.W.

(eds.) ISC 2014. LNCS, vol. 8488, pp. 172–186. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_11
10. Strout, M., Olschanowsky, C., Davis, E., Bertolacci, I., et al.: Varitions on a theme (2017). https://github.com/CompOpt4Apps/VariationsOnATheme
11. Verdoolaege, S.: Integer Set Library (2016). http://isl.gforge.inria.fr/
12. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Programming Language Design and Implementation. ACM, New York (1991)
13. Wolfe, M.J.: Iteration space tiling for memory hierarchies. In: Third SIAM Conference on Parallel Processing for Scientific Computing, pp. 357–361 (1987)

OpenMP in Heterogeneous Systems

Manage OpenMP GPU Data Environment Under Unified Address Space

Lingda Li[1(✉)], Hal Finkel[2], Martin Kong[1], and Barbara Chapman[1]

[1] Brookhaven National Laboratory, Upton, USA
{lli,mkong,bchapman}@bnl.gov
[2] Argonne National Laboratory, Lemont, USA
hfinkel@anl.gov

Abstract. OpenMP has supported the offload of computations to accelerators such as GPUs since version 4.0. A crucial aspect in OpenMP offloading is to manage the accelerator data environment. Currently, this has to be explicitly programmed by users, which is non-trival and often results in suboptimal performance. The unified memory feature available in recent GPU architectures introduces another option, implicit management. However, our experiments show that it incurs several performance issues, especially under GPU memory oversubscription. In this paper, we propose a compiler and runtime collaborative approach to manage OpenMP GPU data under unified memory. In our framework, the compiler performs data reuse analysis to assist runtime data management. The runtime combines static and dynamic information to make optimized data management decisions. We have implement the proposed technology in the LLVM framework. The evaluation shows our method can achieve significant performance improvement for OpenMP GPU offloading.

Keywords: Data management · Unified memory
OpenMP offloading · Compiler · Runtime · LLVM

1 Introduction

Today's computing systems rely on accelerators to achieve performance and energy efficiency goals. As the most popular accelerator nowadays, the massive threading ability of GPUs can especially benefit applications with large amounts of parallelism, such as scientific computing and machine learning. Therefore, GPU is and will remain a crucial component of supercomputing systems in the foreseeable future. For instance, in the next OLCF supercomputer, Summit, each node is equipped with 6 NVIDIA Volta GPUs while the number of CPUs remains 2 [2].

In order to leverage accelerators like GPUs, OpenMP 4.0 introduced the ability to offload computations to accelerators [3]. It is called device offloading. Compared to native GPU programming models such as CUDA [15] and OpenCL [19], using OpenMP for GPU programming has a shorter learning curve for users and is more performance portable. Compared to other directive based methods like OpenACC [1], OpenMP has a broader user community and better compiler support. Therefore, we expect the number of OpenMP+GPU users will continue to grow.

However, writing efficient GPU programs is still a non-trivial job with OpenMP. One of the biggest challenges in GPU programming is how to efficiently program GPU memory. Normally, CPU and GPU are attached with separate memory since they have different memory preferences. While CPU prefers low access latency, GPU performance is more sensitive to memory bandwidth compared with latency. Separate memory also helps reduce memory contention caused by sharing. Traditionally, CPU and GPU use separate memory spaces for their individual memory. As a result, they cannot access each other's memory, and data exchange has to be managed explicitly by programmers.

To ease the programming of GPU memory, a feature called *unified memory (UM)* is introduced in recent NVIDIA GPU architectures. Unified memory introduces a single memory space which covers both CPU and GPU memory. From programmers' perspective, they do not need to worry about the location of accessed data, and data is moved between CPU and GPU by the underlying system software automatically if necessary. The burden of programming data transfer is relieved.

The other major advantage of unified memory is that it enables running kernels with memory footprints larger than the GPU memory capacity. Without on demand page migration of unified memory, GPU offloading is possible only if the dataset fits into the GPU memory. While with it, part of data can reside in the CPU memory, and they will be fetched into the GPU memory when actually required at runtime. These advantages promote more usage of unified memory in future GPU programming.

Every story has two sides. As we will show, unified memory also brings many challenges along with its benefits. First of all, page fault overhead can be significant in cases when data transfers dominate in the execution. More importantly, although unified memory is able to address working sets that exceed the GPU memory capacity, significant data thrashing often happens in such scenarios. Programmers often have no clues about these issues. Therefore, we believe it is crucial to address the performance issues of unified memory for OpenMP offloading, and it would be preferable if the solution is transparent to programmers.

This paper makes the following contributions for this goal.

- First, we analyze the performance of unified memory. The results reveal that its performance mainly depends on accessed data properties, including size, access density and reuse situation (Sect. 3).
- We design a compiler-runtime collaborative framework to optimize unified memory performance and implement it in Clang and LLVM OpenMP runtime [12]. The proposed method analyzes data object properties to find out proper optimization strategies, which are applied at runtime (Sect. 4).
- The experimental results demonstrate that our technique can improve unified memory performance significantly while having low overhead (Sect. 5).

2 Related Work

Since the introduction of device offloading in OpenMP 4.0, several compilers have adopted this feature. For instance, [5] describes how to implement this extension in the LLVM framework. Our optimization uses this work as the baseline.

There are several proposals to simplify and optimize the GPU memory management. CGCM [10] provides compiler and runtime support to automatize the GPU memory management for CUDA programs. Pai *et al.* propose a software coherence mechanism to reduce redundant data transfers between the CPU and GPU [17]. Zhao and Xie propose to leverage hybrid DRAM and NVM GPU memory systems and a data migration mechanism to reduce GPU power consumption [20]. These works aim at traditional GPU programs where data movement is managed explicitly by users, and do not consider unified memory.

Some recent research aims to study or improve the performance of unified memory. In the presence of heterogeneous memory, Agarwal *et al.* propose that the ratio of data allocation in each memory should be proportional to the memory bandwidth in order to achieve the highest total bandwidth [4]. The method we propose in this paper is orthogonal to this work.

Several research efforts have studied and optimized OpenMP device data management. Grinberg *et al.* introduce a method to use unified memory within the current OpenMP implementation [8]. Mishra *et al.* study the OpenMP offloading performance under unified memory [14]. Cui *et al.* propose a pipeline directive to break down OpenMP parallel loops and thus achieve device computation and communication overlapping [7]. Hahnfeld *et al.* propose to use existing OpenMP 4.5 directives for similar purposes [9]. Olivier *et al.* discuss double buffering for Intel Xeon Phi processors in OpenMP [16]. These methods are limited to cases where data access patterns are analyzable, and they also require programming efforts. In contract, our work is able to address unpredictable memory access pattern using unified memory, and does not require inputs from users.

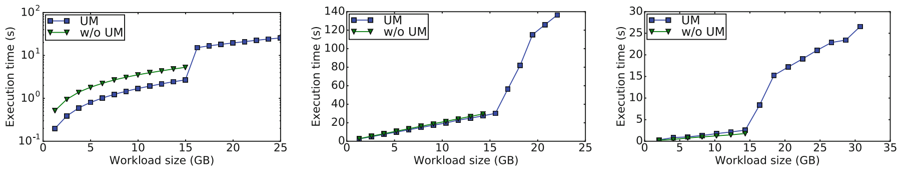
3 Unified Memory Analysis

As the first step, we compare the performance of unified memory with that of traditional GPU memory management approach, and analyze how it performs in different scenarios. Table 1 shows our benchmarks. We use the OpenMP offloading version of BFS, CFD, and SRAD from the Rodinia benchmark suite in our experiments [6,14]. For each benchmark, we generate inputs with various sizes to study the performance impact of workload sizes. The detailed experimental setup is described in Sect. 5.1. We modified the LLVM OpenMP runtime so that it supports the placement of data in unified memory.

Figure 1 illustrates the GPU execution time when data is placed in unified memory and transferred implicitly, versus when data is transferred by OpenMP runtime explicitly. The x axis represents the working set size and y axis represents the execution time. The measured execution time captures both the GPU

Table 1. Benchmarks.

Name	Domain	Description
Breadth first search (BFS)	Graph algorithms	Breadth first search traverses all the connected components in a graph
Computational fluid dynamics solver (CFD)	Fluid dynamics	The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow
Speckle reducing anisotropic diffusion (SRAD)	Image processing	SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs)



(a) BFS. (b) CFD. (c) SRAD.

Fig. 1. GPU performance under traditional approach and unified memory.

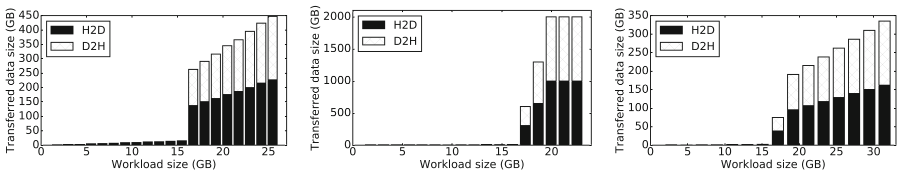
computation time and data transfer time between CPU and GPU. Note that the y axis of BFS is on the logarithmic scale due to the dramatic performance change in the presence of memory oversubscription. Our key observations are as follows.

1. For working sets that fit into the GPU memory, unified memory outperforms when less data is actually required at runtime. While traditional approach needs all data to be present in the GPU memory before computation starts, unified memory only transfers the actually accessed data at runtime and thus may result in less data transfer. However, the data transfer bandwidth is lower under unified memory, because it incurs extra address translation and page fault processing overhead.

Here, we define the ratio of actually accessed data size and total data size as *access density*. The lower the density is, the less data is transferred under unified memory. When it is lower than a threshold (mostly depends on the hardware), the benefit brought by less data transfer outweighs the lower bandwidth disadvantage of unified memory. Therefore, unified memory outperforms in such cases. BFS belongs to this category. For other programs including CFD and SRAD, traditional approach outperforms.

In summary, for data with high density, we would like to explicitly transfer all data beforehand to reduce page fault overhead. Otherwise, we should let unified memory fetch data on demand at runtime.

2. Unified memory suffers from poor performance for oversubscribing workloads with data reuse. For working sets that are larger than the GPU memory size, unified memory is able to work correctly while traditional approach fails. Its performance is largely decided by data reuse for such workloads. When large amount of data gets reused, it is likely that reused data will thrash between CPU and GPU memory. While all 3 benchmarks exhibit various degree of data thrashing behavior, BFS incurs the largest performance loss.



(a) BFS. (b) CFD. (c) SRAD.

Fig. 2. Data transfer volume under unified memory. H2D and D2H represent traffic from CPU to GPU and that from GPU to CPU respectively.

Table 2. Unified memory performance summary and optimization strategies.

Data size	Reuse	Density	Performance	Optimization
≤GPU memory size	\	High	Slightly worse (page fault overhead)	Explicit data copy
		Low	Better (less data transfer)	None
>GPU memory size	High	\	Poor (data thrashing)	Data pinning
	Low		Good	None

GPU programs are more likely to suffer from data thrashing because of the following reason. In the GPU execution paradigm, different threads usually perform similar operations on different data items to exploit its massive threading and data parallel ability. As a result, a large volume of data gets accessed in a single OpenMP target region (i.e., kernel) call, which fills up the GPU memory and evicts old data out. Since data reuse usually happens across different GPU kernel invocations, soon-to-be-reused data is not likely to survive in the GPU memory under the default replacement algorithm, LRU [11,13,18].

Figure 2 shows the data transfer volume of both directions under different workloads. For oversubscribing workloads, the dramatic traffic increment of both directions demonstrates the existence of data thrashing. Data thrashing not only adversely affects performance, but also wastes a lot energy on redundant data transfer.

To avoid data thrashing, we propose to pin data in a certain memory to prevent harmful data movement. The pinned location, GPU or CPU memory, should be selected based on the overall locality of a data object. For instance, data with good locality should be pinned to GPU, and data with poor locality should be pinned to CPU instead.

3. Unified memory performs well for oversubscribing working sets with little data reuse. If little data reuse exists, the performance of unified memory does not show significant difference whether GPU memory is oversubscribed or not. Since there is no reuse, all data is brought to the GPU memory once and replacement decisions do not affect performance. On demand data fetching works well in this case.

Conclusion. Table 2 summarizes the performance of unified memory and corresponding optimization strategies in different scenarios. Data size, access density and data reuse (i.e., locality) play important roles in the performance of unified memory. Later we will introduce how we identify different scenarios and apply optimization strategies accordingly, in order to improve unified memory performance.

4 Unified Memory Management

In this section, we propose a compiler-runtime combined framework to optimize GPU unified memory management. The key idea of our framework is to analyze the properties of data objects in unified memory, and apply optimization according to the analysis results for each object. The data analysis is performed by both compiler and runtime in our framework, while the optimization is applied by runtime. We will introduce each part separately in the rest of this section.

4.1 Static Analysis

The compiler identifies unified memory data objects and performs static analysis on them. The proposed compiler analysis includes 3 stages: data allocation analysis, GPU data usage analysis, and data access frequency analysis. All analysis is performed on the LLVM IR level. We briefly describe them as follows.

Data Allocation. As the first step, we identify all data objects in unified memory space and record them. Such objects can be allocated through CUDA APIs (e.g., `cudaMallocManaged()`), and OpenMP memory allocation APIs (e.g., `omp_target_alloc()`). Note that we modified the implementation of `omp_target _alloc()` in the LLVM OpenMP runtime to support unified memory allocation. The compiler employs a *GPU object table (GOT)* to keep records of detected unified memory objects and their information obtained in the following steps.

GPU Usage. Then, we would like to find where these data objects are used in GPU execution (i.e., which OpenMP offloading regions) for later analysis. We implement a pass to check all usage of a unified memory object's allocated memory space. When a related address of the object is passed to an OpenMP target launching function (e.g., `__tgt_target()`), we identify one instance of its GPU usage and record this OpenMP target region in the corresponding GOT entry.

Data Access Frequency. At last, we design a compile-time analysis pass to help understand data access frequency within target regions, which will be used to estimate *data reuse* and *access density* that is critical for unified memory optimization as shown in Sect. 3. First, for a certain data object in GOT, we would like to calculate the access frequency for every OpenMP target region that uses it, namely *local access frequency (LAF).* The existing LLVM pass `BlockFrequencyInfo` can help achieve this purpose. This pass takes the taken probability of branch instructions as input, to derive the execution frequency of every basic block. It achieves so with static information. Using the information provided by `BlockFrequencyInfo`, we get the execution frequency of each memory access instruction. Then we accumulate the frequency of all memory instructions within a target region that relate to a data object to get its LAF.

We would also like to get the *global access frequency (GAF)* of each data object, which represents the overall GPU access frequency across the whole program. For this purpose, we implement an inter-function/module analysis pass to build a global call graph that includes both CPU and GPU functions. In this graph, we calculate the call frequency of each parent and child function pair, using the results from `BlockFrequencyInfo`. Then we estimate the overall execution frequency of each GPU function by traversing all leaf nodes in the built call graph. Combining the execution frequency and LAFs of all GPU functions, the GAF of a unified memory object is calculated.

Data Reuse. Since the desired optimization only needs a relative not absolute data reuse results, i.e., it is good enough to tell object A gets better reuse than object B, the *data reuse* of a data object is derived from its GAF directly. We rank all unified memory objects based on their GAFs, and use the ranking number to represent the data reuse. Assuming object A has the highest GAF and B has the lowest GAF, the ranking (i.e., data reuse) of A and B will be 1 and n respectively, where n is the total number of objects. We modify the OpenMP runtime interface, so that for every unified memory argument, both data reuse and LAF information is passed to the OpenMP runtime on offloading. LAFs will be used by the runtime to estimate access density as will be introduced in Sect. 4.2.

Completely compiler-based data access analysis has the drawback of low accuracy and non-awareness of dynamic execution pattern. The latter is critical for GPU execution since a code fragment can be executed by millions to billions of threads. For data reuse, a relative result is good enough and thus we use static analysis results for simplicity. For data size and access density, we leave a significant part of analysis to the runtime, as will be introduced in Sect. 4.2.

Overhead. The proposed analysis utilizes results from existing LLVM passes and does not need to be performed recursively. Compared to dozens of default analysis and optimization passes in Clang, its time overhead is negligible. In our experiments, we do not observe notable compile time change when the proposed analysis is enabled.

4.2 Runtime Analysis

Our runtime analysis utilizes runtime information to help the compiler finalize data analysis results. Particularly, data size and access density are estimated combining both runtime and compile-time information.

Data Size. The size of data objects depends on input in many cases and thus it is natural for runtime to get this information. In the OpenMP offloading runtime interface, the size of arguments is passed along with arguments themselves, and it is thus free for runtime to get data size. One problem is the existing data size is measured in bytes, while we would also like the number of elements for the access density estimation introduced below. Luckily the element size can be easily obtained by the compiler through type checking, and we pass it to the runtime so that the number of elements can be computed by dividing the total size by the element size.

Access Density. Density is calculated as the actual accessed element number divided by total element number. We already discussed how to get the total element number. The difficult part is how to estimate the number of actually accessed elements.

Fortunately, the regular code structure of OpenMP target regions makes a simple solution to calculate the number of accessed elements possible. In the common scenario which covers more than 90% of offloading regions, an outer `for` loop contains all work of an offloading region. Its iterations are distributed across all GPU threads. The loop body is usually short and has simple control flow. In such scenarios, the accessed element number in a single iteration is easy to estimate thanks to the simple loop body. The total accessed number mainly depends on how many iterations are executed.

We design a compiler-runtime combined scheme to compute total accessed element number and thus access density. The runtime is responsible to obtain the number of outer loop iterations, while the compiler estimates the number of accessed elements in a single iteration, using LAF. If we assume memory accesses distribute evenly across different elements in a data object, which is quite reasonable for GPU, the number of accessed elements in a loop body is equal to its LAF. By multiplying LAF and loop iteration number together at runtime, we get an estimation of the total accessed element number in a object. Then the access density can be computed as min(accessed element number/total element number, 100%).

Discussion. The limitation of our density estimation method is that it assumes a unified access distribution and does not distinguish elements within a data object. For instance, if a small fraction of elements receive all data accesses in an object, the access density estimated using the proposed method will be larger than the actual value.

In order to get more accurate analysis results, methods such as profiling and instrumentation can potentially be used. However, unlike our proposed method which puts little burden on compiler and runtime, these methods suffer from significant compiling/runtime overhead and often need help from programmers. Applying them will also add significant complexity to the implementation. As Sect. 5 will show, the proposed compiler-runtime combined analysis can already achieve significant performance improvement.

4.3 Runtime Management

This subsection will describe how we manage unified memory objects based on the above analysis results, namely data size, data reuse and access density. The runtime makes two key decisions for each object: (1) where it should be mapped, GPU or CPU memory, and (2) how it should be transferred if it is mapped to GPU, explicitly or implicitly (i.e., data transfer is performed on demand). Table 2 has listed our optimization strategies.

Data Mapping. When encountering an OpenMP target call, the runtime will follow these steps to map the arguments in unified memory before execution starts. After all properties are collected (i.e., size, reuse and density) as described earlier, all unified memory objects involved in this call are ranked based on their reuse. Then we select the proper mapping strategy for each object using the reuse order (from large to small), so that data with better locality has higher priority to be mapped to the GPU memory. If there is enough space in the GPU memory, the current object is mapped to the GPU memory. Otherwise, it is mapped to that of CPU to prevent data thrashing. In this case, we use the CUDA API `cudaMemAdvise` to pin data into the CPU memory.

Data Transfer. If an object is mapped to the GPU memory, we further decide how it should be transferred based on its density. Objects with small density (<0.6, an empirical number obtained based on experimental results) should be transferred implicitly to reduce data transfer volume, otherwise explicit transfer is used. For explicit transfer, a GPU memory object with the same size is allocated using `cudaMalloc` for GPU usage, and data transfer primitive `cudaMemcpy` is used to synchronize original and new copies. This is the default policy followed by the current OpenMP implementation. In the case of implicit transfer, we simply pass the original object to GPU kernels, and let the unified memory driver handle on demand data transfer during execution.

Book Keeping. To implement the method described above, several book keeping needs to be done at runtime. To keep track of GPU memory, the runtime uses two 64-bit counters for each GPU. They record the size of GPU memory objects that are transferred explicitly and implicitly, respectively. We can calculate the free GPU memory size with these two counters.

The proposed runtime also maintains a table to keep records of active data objects in the GPU memory. For each object, it records the size, data reuse, mapping place (GPU or CPU), and transfer mechanism (explicit or implicit).

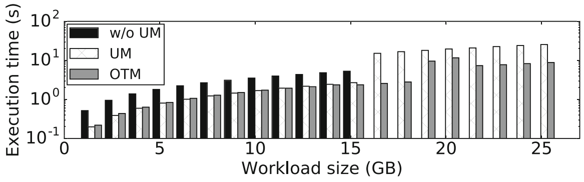
Overhead. The proposed runtime is integrated seamlessly within the existing LLVM OpenMP target offloading runtime. We do not introduce any expensive operation into it. In our experiments, we find that there is virtually no difference for the runtime execution time with or without our modification. Besides, all performance results in Sect. 5 include the runtime overhead, if there is any.

5 Evaluation

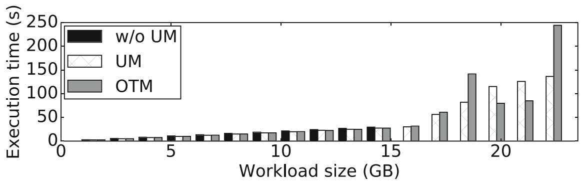
5.1 Experimental Methodology

To evaluate the performance of our benchmarks, we use the OLCF SummitDev, which is the prototype machine of Summit. Each SummitDev node is equipped with 2 POWER8 CPUs and 4 Tesla P100 GPUs. They are connected through NVLink 1.0, which provides up to 160 GB/s IO bandwidth per GPU. The Tesla P100 GPU has 56 SMs and is equiped with 16 GB HBM2 memory. It supplies a local memory bandwidth of 732 GB/s.

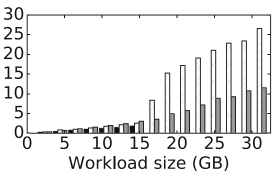
We use the up-to-date Clang [5] that supports OpenMP GPU offloading to compile benchmarks. To enable offloading for NVIDIA GPUs, we pass the flag `-fopenmp-targets=nvptx64-nvidia-cuda` along with `-fopenmp` to Clang. All benchmarks are compiled under the O2 optimization level. The Linux kernel version is 3.10.0 and the CUDA version is 9.0.69 on SummitDev.



(a) BFS.



(b) CFD.



(c) SRAD.

Fig. 3. Performance of various schemes.

5.2 Performance Results

Figure 3 illustrates the GPU performance under unified memory (UM), traditional approach (w/o UM), and our compiler-runtime collaborative OpenMP Target data Management framework (OTM). Again, note that the execution time (y axis) of BFS is on the logarithmic scale. While OTM helps BFS and SRAD achieve significant performance improvement, it fails to do so on some CFD workloads. Detailed analysis is as follows.

BFS. BFS receives the most performance gain from our approach. Under fitting workloads, OTM outperforms w/o UM by 113% on average and has similar performance compared to UM. Under oversubscribing workloads, OTM achieves a dramatic average speedup of 3.37× compared with UM.

There is a large amount of data reuse existing in BFS, because the same vertex and neighbor vertices are likely to be accessed in multiple iterations. However, the traversal happens in an irregular order, and thus it is difficult to optimize its performance using traditional methods. With OTM, the data structure that is used to store edges of each vertex, which is less frequently accessed but has the largest size, is often pinned to the CPU memory. This prevents it from thrashing other more important data in the GPU memory, so that data locality can be exploited within the GPU memory.

Note that there is a performance drop for OTM at the workload of 20 GB. This is because at this point, OTM decides to pin several small data objects into the CPU memory instead of a larger one, due to the GPU memory capacity limitation. As a result, having multiple objects in the CPU memory collectively has a larger impact on performance. When the workload is larger, once again, a large data object is pinned to the CPU memory. We will address this issue by enabling finer grained data mapping control in the near future, to further improve performance.

CFD. On average, OTM and UM has similar performance across all workloads. On some workloads, OTM is outperformed by UM. The reason that OTM fails to improve CFD performance, is that OTM currently does not handle complex scenarios well. Compared with BFS and SRAD, CFD has more complex data structures and control flow. Multiple target regions interleave with each other in multiple ways, and different regions use different sets of data objects as well as share some of them. Under some workloads, we find that for every target region, OTM pins some of its data objects into the CPU memory, which slows down all target region execution. The smarter choice here is to keep all data required by some target regions in the GPU memory to accelerate their execution, while have mixed data location for other kernels. We will develop technology to solve this problem soon.

SRAD. OTM helps SRAD achieve an average speedup of 2.55× across all oversubscribing workloads compared with UM. Since the data reuse in SRAD is limited compared with that in BFS, the speedup of OTM is more moderate.

For smaller workloads, the performance of OTM is similar to that of UM while lower than the traditional approach by 30.7% on average. By taking a closer look, we find that OTM transfers some highly reused data objects using unified memory's on demand fetching, while they should be transferred explicitly. This is because loop nests exist in some target regions, which confuses the proposed compiler analysis. More accurate analysis methods can be used to alleviate this problem. Luckily selecting the incorrect data transfer manner does not impose a large performance penalty. The data mapping location has much more significant performance impact, in which OTM makes optimized decisions for all 3 benchmarks.

In all benchmarks, the data transfer between CPU and GPU is reduced significantly for large workloads under OTM. Since there are no existing tools that can extract data transfer volume when data pinning is applied, we do not compare the data transfer of different methods.

6 Conclusion

In this paper, we develop a compiler-runtime collaborative technology to improve OpenMP GPU data management under unified memory. There are several future directions worth exploring besides what we have mentioned in Sect. 5. First, application experts may wish to provide data locality hints directly rather than relying on compiler analysis. We plan to explore new OpenMP directives/clauses for this purpose. Second, ideas presented in this paper are not limited to unified memory but also applicable to more generic scenarios. We plan to further develop our techniques to have a generic optimized OpenMP GPU data management framework.

Acknowledgement. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

1. OpenACC. http://www.openacc.org
2. Summit. https://www.olcf.ornl.gov/summit
3. OpenMP 4.0 specifications (2013). http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
4. Agarwal, N., Nellans, D., Stephenson, M., O'Connor, M., Keckler, S.W.: Page placement strategies for GPUs within heterogeneous memory systems. In: ASPLOS 2015, pp. 607–618. ACM, New York (2015)
5. Antao, S.F., et al.: Offloading support for OpenMP in Clang and LLVM. In: LLVM-HPC 2016, pp. 1–11. IEEE Press, Piscataway (2016)
6. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, pp. 44–54. IEEE (2009)
7. Cui, X., Scogland, T.R.W., de Supinski, B.R., Feng, W.C.: Directive-based partitioning and pipelining for graphics processing units. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 575–584, May 2017

8. Grinberg, L., Bertolli, C., Haque, R.: Hands on with OpenMP4.5 and unified memory: developing applications for IBM's hybrid CPU + GPU systems (Part I). In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 3–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_1
9. Hahnfeld, J., Cramer, T., Klemm, M., Terboven, C., Müller, M.S.: A pattern for overlapping communication and computation with OpenMP* target directives. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 325–337. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_22
10. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. In: PLDI 2011, pp. 142–151. ACM, New York (2011)
11. Jaleel, A., Theobald, K.B., Steely, Jr., S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). In: ISCA 2010, pp. 60–71. ACM, New York (2010)
12. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO 2004, p. 75. IEEE Computer Society, Washington, DC (2004)
13. Li, L., Tong, D., Xie, Z., Lu, J., Cheng, X.: Optimal bypass monitor for high performance last-level caches. In: PACT 2012, pp. 315–324. ACM, New York (2012)
14. Mishra, A., Li, L., Kong, M., Finkel, H., Chapman, B.: Benchmarking and evaluating unified memory for OpenMP GPU offloading. In: LLVM-HPC 2017, pp. 6:1–6:10. ACM, New York (2017)
15. NVIDIA: Compute unified device architecture programming guide (2007)
16. Olivier, S.L., Hammond, S.D., Duran, A.: Double buffering for MCDRAM on second generation Intel® Xeon Phi™ processors with OpenMP. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 311–324. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_21
17. Pai, S., Govindarajan, R., Thazhuthaveetil, M.J.: Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In: PACT 2012, pp. 33–42. ACM, New York (2012)
18. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J.: Adaptive insertion policies for high performance caching. In: ISCA 2007, pp. 381–391. ACM, New York (2007)
19. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)
20. Zhao, J., Xie, Y.: Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration. In: ICCAD 2012, pp. 81–87. ACM, New York (2012)

OpenMP 4.5 Validation and Verification Suite for Device Offload

Jose Monsalve Diaz[1], Swaroop Pophale[2](✉), Oscar Hernandez[2], David E. Bernholdt[2], and Sunita Chandrasekaran[1]

[1] University of Delaware, Newark, DE, USA
{josem,schandra}@udel.edu
[2] Oak Ridge National Lab, Oak Ridge, TN, USA
{pophaless,oscar,bernholdtde}@ornl.gov

Abstract. OpenMP has been widely adopted for shared memory systems for over a decade. With the heterogeneity trend in architectures rapidly growing, the programming model needed to evolve such that applications could not only be ported to traditional CPUs but also to accelerators often acting as discrete or integrated devices to CPUs. To that end, OpenMP started to provide support for heterogeneous systems since 2013 when the version 4.0 of the specification was ratified. OpenMP 4.5 is being enhanced to cover major requirements of Exascale Computing Project (ECP) applications. As a result it is time-critical to ensure that the implementations of the 4.5 features are correct and conforming to the specification. This paper focuses on building a Validation and Verification testsuite that will test and present results for several offloading features implemented in compilers such as Clang, IBM XL C/C++, CCE, and GCC. We have results for our testsuite on TITAN, Summitdev and Summit at the Oak Ridge National Lab. We will highlight some of the ambiguities we encountered in the process of validating and verifying feature implementations. We also make the testsuite available for anyone to use and will walk the readers through the infrastructure and the workflow of the testsuite. A website has been built to capture our efforts narrated in this paper https://crpl.cis.udel.edu/ompvvsollve.

Keywords: OpenMP · Validation and Verification · Testsuite

J. M. Diaz and S. Pophale—Contributed Equally
This manuscript has been co-authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

1 Introduction

In 2013 the OpenMP specification made a significant shift to provide support for heterogeneous systems. They introduced a set of directives for identifying code as well as data to be moved to the target device for computation. Other than support for accelerators, `SIMD` support for vectorization, thread affinity control, user defined reductions, updates to `task` construct by introducing task groups and dependency clauses were also introduced. This led to the release of Version 4.0. Significant further improvements for device support along with runtime routines for device memory management was introduced in Version 4.5 in 2015 along with new `taskloop` construct that would enable loops to be divided into tasks avoiding the requirement that all threads execute the loop. Support for `doacross` loops to parallelize loops with well-structured dependences were provided. Further support for tasks in the form of task priority was introduced. `SIMD` extensions included the ability to specify exact SIMD width and additional data-sharing attributes.

As the OpenMP specification continues to grow and evolve with all its existing and new features, it is critical to ensure that the different implementations that claim conformance are functionally correct and true to the specification. Having confidence in the correctness of implementations will encourage *users* to adopt OpenMP 4.5 for large applications and port them to heterogeneous systems. Some of the applications that have been ported to GPUs at DOE laboratories using OpenMP 4.5 include Pseudo-Spectral Direct Numerical Simulation-Combined Compact Difference (PSDNS-CCD3D) [1], a computational fluid dynamics code on turbulent flow simulation using GPUs and run to scale on the Titan super-computer, Quick Silver [12], a proxy app of Mercury, that solves a simplified dynamic Monte Carlo particle transport problem. Both these papers discuss the challenges such as heterogeneous memory model, thread safety and thread management, and common programming patterns that are not portable, faced by the application developers before the code ran on a GPU using OpenMP 4.5.

As hardware continues to evolve, the trend for systems to be equipped with specialized accelerators or co-processors attached to a CPU-based system is only going to continue. Top500 reports that a hundred and two systems in the list are configured with accelerators and coprocessors among which, eighty six use NVIDIA GPUs, twelve use Intel Xeon Phi cards, five uses PEZY technology, and two systems use a combination of NVIDIA GPUs and Intel Xeon Phi coprocessors [13]. Directive-based programming models, like OpenMP, can prove to be very effective and useful especially when there are legacy codes that need to be migrated to such evolving platforms. Programming models do not require application developers to be fully aware of hardware details or learn newer programming languages thus allowing developers to spend more time on the algorithms and the scientific findings and lesser time on deciphering the intricate details of hardware and language.

Going forward we already know that ORNL's Summit will be a heterogeneous platform consisting of IBM Power9 and NVIDIA's Volta GPUs. We also know that Argonne National Lab will receive an Intel-based system (not Knights Hill), Aurora, in the time frame of 2021. Having all this compute power is useless unless we have vetted OpenMP 4.5 implementations that deliver what the specification promises.

Compilers [11] that support OpenMP features include GCC 7.1 where OpenMP 4.5 is fully supported for C and C++. IBM XL C/C++ for Linux V13.1.5 on little endian distributions and XL Fortran for Linux V15.1.15 on little endian distributions (available in Dec 2016) support OpenMP 3.1 and features in OpenMP 4.5 (include device constructs for offloading to NVIDIA GPU), Intel 17.0 supports OpenMP 4.5 for C/C++ and Fortran, Cray Compiling Environment (CCE) 8.5 (June 2016) supports OpenMP 4.0, with OpenMP 4.5 support for device constructs, LLVM Clang 3.9 release supports all non-offloading features of OpenMP 4.5. Clang that supports offloading features is currently in development.

Such a wide range of compilers and interpretation of different implementers can lead to differing implementations of OpenMP features. Our previous publications have captured such discrepancies [6,15]. To that end, this paper continues to leverage our previous contributions on Validation and Verification testsuite for OpenMP and build a number of functional test cases along with use cases to test 4.5 offloading constructs and clauses or combinations of occurrences of clauses on constructs in order to check for correctness and conformance of features specifically to the 4.5 specification. Once our implementation of the new OpenMP 4.5 features is complete (this is currently on-going work and not fully complete yet), we plan to tie in the 3.1 testsuite and make them all available under one repository to enable testing the entire 4.5 Specification.

This paper makes the following contributions:

- Releases a Validation and Verification testsuite currently comprising of approximately 60 tests that includes unit tests covering extensively target, target data, target enter and exit data features, target update, target teams distribute, and target teams distribute parallel for. We will continue to add more test cases to this suite.
- The testsuite also contains use cases that represent kernels extracted from production DOE applications and other frequently used computation kernels.
- Describe the testsuite infrastructure and add relevant license thus allowing anyone to contribute and to the testsuite.
- Present bugs identified and their potential workarounds thus informing application developers of challenges using key constructs.
- Evaluate implementations of different compilers and verify their conformation to 4.5 specification (these results are posted on https://crpl.cis.udel.edu/ompvvsollve).
- List ambiguities in the specification that we came across while interpreting definitions of clauses and constructs and relevant steps taken to clarify them.
- Assemble a user-friendly website with easy to find status update on compiler implementations among other details.

The remainder of the paper is organized as follows: Sect. 2 discusses some of the most relevant testsuite work by the authors and others. Section 3 gives more insight into the new 4.x features. Section 4 explains the workflow and the process of developing such a suite. Section 5 explains the testsuite infrastructure. Sects. 6 and 7 present discussions, conclusion and future work.

2 Related Work

Related work on OpenMP Validation and Verification suite includes [6,7] that present validation of OpenMP 2.0 implementations which was further extended and improved in [15] to develop a more robust testsuite and provide up-to-date test cases covering all the features until OpenMP 3.1. Some of our other efforts include an OpenACC Validation and Verification testsuite [2,16] where we discuss the compiler status of OpenACC 2.0 and 2.5 specifications respectively and present ambiguities identified in the specification. Some of the tests also focus on challenges with creating unit test cases covering possible combinations of directives/clauses under study. The papers also highlights reported bugs being fixed and the improvement in the compiler's status over a period of time.

Other related efforts to building and using a testsuite include Csmith [17], a comprehensive, well-cited work where the authors perform a randomized test-case generator exposing compiler bugs using differential testing. Such an approach is quite effective to detecting compiler bugs but does not quite serve the validation purpose since it is hard to automatically map a randomly generated failed test to a bug that actually caused it.

LLVM has a testing infrastructure [5] that contains regression tests and whole programs. The tests itself are driven by *lit* testing tool, which is part of LLVM. Recently [3,4] published examples on how a *user* may program with OpenMP 4.5 on IBM's heterogeneous platforms with GPU support. Though these provide a very good overview on how to use OpenMP 4.5 offload features, they assume that the underlying implementation is as per specification and correct.

3 New in OpenMP 4.x

Going from OpenMP Specification version 3.1 to 4.0, the most significant change was the support for accelerators. OpenMP 4.0 provides directives to describe when data and/or computation should be moved to another computing device/accelerator. This is usually indicated by the presence of `target` keyword which in itself and as a part of others forms the new directives for device offload. Other changes that OpenMP 4.0 brought are: SIMD constructs that enable vectorization of serial and parallelized loops, addition of error handling capabilities, thread affinity mechanisms through OMP_PROC_BIND and OMP_PLACES, tasking extensions for support task-to-task synchronization, Fortran 2003 support that allows interoperability of Fortran and C, user defined reductions, and ability to enforce sequential consistency in atomics [9].

In November 2015 OpenMP Specification 4.5 was released. The significant changes there were the introduction of the `taskloop` construct, allowing the use of `depend, threads and simd` clauses on `ordered` directive, data sharing changes allowing C++ methods to privatize accessible non-static members of an object (with restrictions) [10], and changes in default mapping for offloading.

Scalar variables in OpenMP 4.0, in the absence of explicit mapping, were implicitly mapped `tofrom` but in OpenMP 4.5 (without explicit mapping) the scalars are implicitly privatized. They have the same effect as if they were declared `firstprivate` on the target construct. Now the *User* has to explicitly copy the scalar value back if the value is needed on the host. With OpenMP 4.5 C/C++ pointers are implicitly mapped. Hence the host pointer is translated to the corresponding device pointer in case the pointed object is already mapped else it is NULL.

The `use_device_ptr` clause to `target data` construct and `is_device_ptr` clause to `target construct` allow using device specific memory routines. Also new directives such as `target enter data` and `target exit data` were added to enable mapping and un-mapping of variables independently (synchronously or asynchronously) in separate functions or methods. Additional support for mapping C++ references made possible to map structure elements individually in OpenMP 4.5. Asynchronous offloading on the `target` directive through the `nowait` and `depend` clauses is now possible. Lastly the `declare target` directive was extended to be able to mark global variables for deferred mapping.

4 Testsuite Workflow

The testsuite presented in this work aims at providing tests that provide a comprehensive coverage of different offload directives in OpenMP 4.5. The tests are intended to establish accuracy of the interpretation of the OpenMP specification by an implementation and verify the correctness of the functionality. Development of the tests is an on-going iterative process where we address both the functional tests for different combinations of directives and clauses and application kernels tests abstracted out of real world applications. We evaluate all tests by verifying with the specification through a peer review process and testing them on different OpenMP implementations available to us. Our current testbeds (Summitdev, Titan and Summit) at ORNL. They have LLVM, IBMXL, GNU and CCE compilers available, each implementation, in our experience, has different levels of support for OpenMP 4.5. We solicit community feedback, especially from OpenMP developers, to review and refine the tests. For a given directive we first refer to the OpenMP 4.5 specifications to list the different directives and their constructs. Although it is hard to assess how many tests are needed per directive, we try to keep an organized list of the tests we have created. We start by listing all the directives, followed by the multiple constructs that can be used with such directive. For each construct we expand the list with possible modifiers, options, or cases that could apply to that particular construct. Our goal is cover as many valid combinations as possible. We have found it difficult

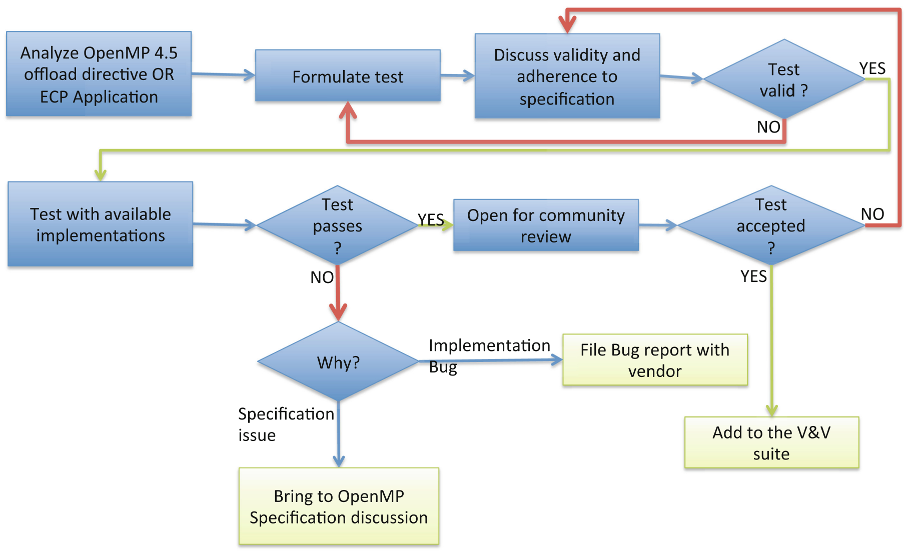

Analyze OpenMP 4.5 offload directive OR ECP Application
Formulate test
Discuss validity and adherence to specification
Test valid ?
YES
NO
Test with available implementations
Test passes ?
YES
NO
Open for community review
Test accepted ?
NO
YES
Why?
Implementation Bug
File Bug report with vendor
Specification issue
Bring to OpenMP Specification discussion
Add to the V&V suite

Fig. 1. Workflow for developing the validation and verification suite

to assess the exact number of tests that are needed per combination of directive and clause(s). In most of the cases, new tests will often come up during the discussion, as there are corner cases either from the description of the specifications, or the interaction with the C and C++ programming model that need to be addressed.

For our second category of tests we collect different application motifs and distill them down to tests through our interaction with different DOE applications. These tests provide insights into how OpenMP 4.5 directives are used in real-world applications and could potentially bring to light unexpected side effects or performance degradation due to interactions between different `target` directives. Currently, our Validation and Verification suite is hosted on bitbucket for easy collaboration and though not complete (in coverage of all offloading directives) we plan to open it to the OpenMP community with this paper. A summary of the results can be seen at our website https://crpl.cis.udel.edu/ompvvsollve. The framework, discussed in more detail in Sect. 5, is geared to be stand-alone, with options to compile for different OpenMP implementations.

Figure 1 represents the development cycle of a test-case. There are three possible positive outcomes of the process we have adopted. Either a test passes through all the checks and makes it to the validation suite, or it uncovers a bug in the vendor implementation of the OpenMP 4.5 standard, or highlights a contentious concept or text that is easy to misinterpret and brings it to the attention of the OpenMP community and specification developers. All tests are written agnostic to where they are executed (host vs. target). We do this to facilitate execution of tests in situations where the host is also configured to be the target or no device is available at the execution time where the computation could be offloaded. After a test executes the output indicates if the test passed or failed and where it was executed (host or target). We have encountered cases where we cannot confidently confirm the correctness. One such example is the `nowait` clause. For such cases we do not use affirmative output messages. If the test fails, it could have been because of a number of reasons other than the incorrect implementation and we try to capture it as best as possible. Failure could mean failure to compile or execution time failure that lead to crashes or cryptic error codes. By providing feedback to the vendors we hope to make the error codes more *user* friendly.

5 Validation Suite Infrastructure

In addition to having a well defined workflow to guarantee correctness (as described in Sect. 4) it is equally important to provide a well defined and flexible infrastructure that supports such a workflow. The design process of this infrastructure has been thoroughly discussed and incrementally improved, resulting in a set of requirements that justify the different features that we have implemented so far. These requirements are as follows:

1. In order to support the previously described workflow, we must define communication mechanisms between active compiler developers, the OpenMP community and the application developers. This requires creation of a web portal and a code repository.
2. It is necessary to provide the *user* with an easy, flexible and simple to use interface. This interface must allow fast deployment and execution of the testsuite. To this end, the testsuite must adapt to new execution environments, providing customization of compilers' options configuration and flags etc. Additionally, it should be able to support different batch schedulers and Linux environment modules.
3. Those who would like to use this testsuite must be able to obtain and export compilation and execution results for an off-line system evaluation. Hence, the designed infrastructure should allow them to obtain results in either a well defined raw logfile format, an intermediate format for exporting to other analysis tools and scripts, or create a well presented report.

With these requirements defined, we present our infrastructure in the rest of this section.

5.1 Development Environment and Website

As described in our workflow, sharing and evaluating tests is an important part of the test creation process. This requires code sharing and tracking changes. To this end, a repository was created in Bitbucket. We use the features available in a git repository, plus the additional components provided by Bitbucket to support our development workflow. This repository can be found in [14].

In addition to the Bitbucket repository, we have created a website that contains all project related information, including project objectives, documentation on how to use and contribute to this software, publications, and repository guidance. Furthermore, we envision this website as a point of contact with those that would use this testsuite, where they can find documentation, examples, and results obtained from the systems evaluated. Our website is https://crpl.udel.edu/ompvvsollve.

5.2 Makefile

A Makefile has been created and included in this project. It is used as an entry point to our testsuite. The Makefile allows users to compile, run, and report test results. A set of make rules has been created for each purpose, together with a set of options that modify each rule's behavior (e.g. verbosity, log creation and tests selection). Furthermore, it is possible to use the standard CC and CXX flags to select different compilers. Our testsuite uses the following syntax:

```
make CC=ccompiler CXX=cppcompiler [OPTIONS] [RULE]
```

Rules for Makefile. See Table 1 for a list of all the rules that can be used.

Table 1. Set of rules available in the Makefile

Rule	Description
compile	Compile tests using the compilers specified by CC and CXX
run	Run tests that have been previously compiled
all	Compile and run tests using the compilers specified by CC and CXX
compilers	Print a list of available compilers
report_json	Given a set of logfiles, create a JSON file containing all the results
report_html	Create an HTML-based results report that allows filtering and search
clean	Remove all compiled tests

Options. A set of options can be used to modify the behavior of the rules. The `SOURCES_C` and `SOURCES_CPP` options can be used with `compile` and `all` rules to select which tests to compile. To select what tests to run the `TESTS_TO_RUN` option should be used. The `VERBOSE` option can be used to increase verbosity level of the make command, while `VERBOSE_TESTS` will increase verbosity level of the tests outputs. This is, each test can display additional information at runtime about what it is executing and where the error is encountered. The option `LOG` switches logs on and off, while `LOG_ALL` changes if the output of the Makefile commands should be included in the log files. `NO_OFFLOADING` can be used to turn off offloading capabilities in the compiler.

Other options have been created to adapt the testsuite to the underlying system. It is common to use environment modules to provide multiple software, compilers and libraries within the same system. Additionally, batch schedulers are used to guarantee exclusive and fair access to systems in environments where many users access the same resources. However, this creates new challenges to our testsuite. The options MODULE_LOAD and ADD_BATCH_SCHED are available for this purpose. The first one will execute a `module load...` command before compiling or running the tests. The second one will pre-append a batch scheduler command (e.g. jsrun and aprun) to tests that are running. However, since these elements change from system to system it is necessary to create a system description file and use the SYSTEM option to select this description file.

The following use case examples will compile and run all the tests, in verbose mode enabled in both the tests and the make command. Logs will be created including all output from compilers, tests runtime outputs, and make commands outputs. According to the Summit [8] system description file, we will add the `jsrun` command before running each tests, and we will load all the required modules before compiling and running each tests.

```
make CC=clang CXX=clang++ SYSTEM=summit VERBOSE=1 VERBOSE_TESTS=1 \
LOG=1 LOG_ALL=1 ADD_BATCH_SCHED=1 VERBOSE_TESTS=1 MODULE_LOAD=1 all ;
```

Customizations. As mentioned before, it is possible to customize the testsuite to adapt it to the system environment. A template for a system description file is provided which contains the following options: BATCH_SCHEDULER, C_COMPILER_MODULE, CXX_COMPILER_MODULE, C_VERSION, CXX_VERSION, and CUDA_MODULE. The VERSION commands will be used during the log creation. For further information and example, refer to the documentation on our website.

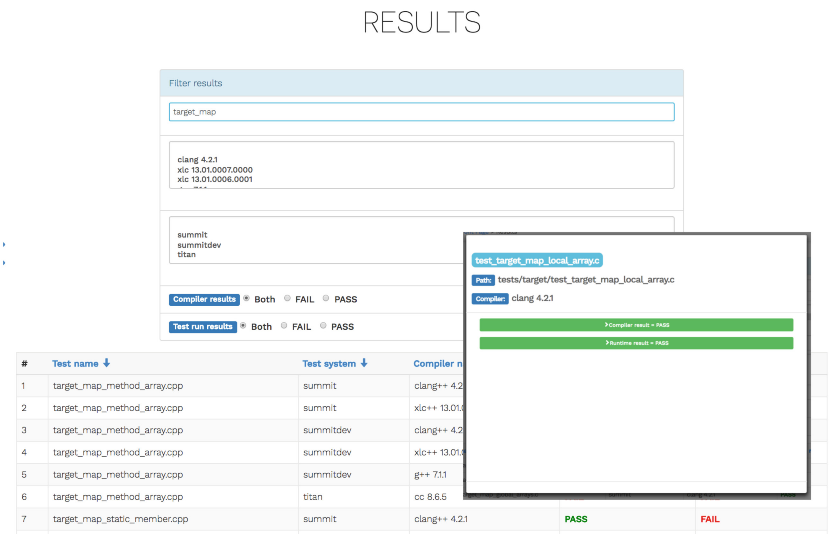
5.3 Results, Logs and Reports

Although it is possible to obtain results directly from the standard output. We have made it easier for the user to create logs and reports for offline results evaluation. So far there are three options to obtain results: Raw format, JSON format, or HTML format.

Raw Format. When the LOG option is used in the make command, a new folder called `logs` is created. It contains a log file per test that was compiled and/or executed. Log files are accumulative in the sense that if the make command is issued multiple times, they will all be registered within the same log files. To differentiate multiple runs, as well as compilation from run, we have created a header and footer line per entry, containing system information, compiler used, source file, compiler command, and time. Refer to our website for Header and footer formats.

JSON Format. Although the raw format is easy to read, it is not well structured and it would be hard to parse and automate to generate final reports. For this reason, we have created the `report_json` rule that uses a script to parse the raw format and output a JSON file. The structure of the JSON file is as follows:

```
[{
        "Binary path": "...",
        "Compiler command": "...",
        "Compiler ending date": "...",
        "Compiler name": "...",
        "Compiler output": "...",
        "Compiler result": "PASS/FAIL",
        "Compiler starting date": "...",
        "Runtime ending date": "...",
        "Runtime only": false/true,
        "Runtime output": "...",
        "Runtime result": "PASS/FAIL",
        "Runtime starting date": "...",
        "Test comments": "...",
        "Test name": "...",
        "Test path": "...",
        "Test system": "..."
}, ...]
```


RESULTS

Filter results

target_map

clang 4.2.1
xlc 13.01.0007.0000
xlc 13.01.0006.0001

summit
summitdev
titan

Compiler results [✓ Both] FAIL PASS

Test run results [✓ Both] FAIL PASS

test_target_map_local_array.c

Path: tests/target/test_target_map_local_array.c

Compiler: clang 4.2.1

Compiler result = PASS

Runtime result = PASS

#	Test name	Test system	Compiler n		
1	target_map_method_array.cpp	summit	clang++ 4.2		
2	target_map_method_array.cpp	summit	xlc++ 13.01.		
3	target_map_method_array.cpp	summitdev	clang++ 4.2		
4	target_map_method_array.cpp	summitdev	xlc++ 13.01.		
5	target_map_method_array.cpp	summitdev	g++ 7.1.1		
6	target_map_method_array.cpp	titan	cc 8.6.5		
7	target_map_static_member.cpp	summit	clang++ 4.2.1	PASS	FAIL

Fig. 2. Snapshot of the HTML report generated by the testsuite

HTML Format Using the JSON file, it is possible to create a user-friendly and readable report. This report uses a pre-defined HTML template with advanced javascript and css libraries that allows the listing of all the tests in a table, access to more information for each test, filter out results by compiler, systems and PASS/FAIL results or even search an specific name or clause. An snapshot of these results can be seen in Fig. 2

6 Discussion

Each test is compiled with four different compilers that are available to us. Some of these compilers provide complete support for OpenMP 4.5 constructs while others have indicated to offer only partial support (at the time of writing this paper). By using multiple implementations we are able to analyze the validity of the tests and at the same time how each compiler's implementation behaves for a particular construct under study. The compilers we use include GCC Version 7.1.1, IBM XL Version 14.1 Beta 7, Cray CCE Version 8.6.1 and Clang Version 3.8.0. It is worth noticing that this version of Clang has been modified for our running environment, and it is not exactly the same available in the main LLVM trunk.

We uncovered many implementation bugs, misunderstanding/misinterpretation of the definitions in the specification that led to us (incorrectly) reporting as an implementation bug, as well as ambiguities in the specification throughout this process. Due to space constraints we only discuss the more interesting cases.

6.1 Implementation Bugs

- `Target` construct in methods of a class
  During testing `target` directive in C++ methods, we noticed that Clang failed to map an array `tofrom` in one of the OpenMP implementations. Since there is no explicit restriction in the OpenMP specification such usage scenarios are valid. Similarly, a `target` construct in static method of class failed to map class static variables. The later was fixed as a result of the bug report.
- Compiler crashes
  We noticed that the Clang compiler crashed when `collapse` clause was used with dependent iteration spaces. Though the behavior is invalid the implementers agreed that it should present an error to the *user* and not crash. With the Cray implementation, when trying to use `map delete` or `release` of a variable that was originally mapped by `target enter` data, led to compiler crashes. The bug reported was promptly addressed to correct such a behavior.
- Scalar values and `defaultmap`
  For the Cray compiler implementation, we uncovered that the `defaultmap` would not correctly map to the scalar values. Scalar variables in OpenMP 4.0, in the absence of explicit mapping, were implicitly mapped `tofrom` but in OpenMP 4.5 (without explicit mapping) the scalars are implicitly privatized.

From our experience most of the errors on our part were from not accounting for the default mapping on the `target` clause. Earlier in the development phase, especially while trying to test `target data` directive we would run into errors such as an array segment mapping to device with default length (e.g. map(array[1:])) would fail at runtime or we would get errors saying that the test was trying to map data that was already mapped. We understand that this behavior is going to change in OpenMP 5.0, where the implicit data mapping on the `target` construct will work differently from explicit data mapping. In the presence of partial mapping, the reference counter will get incremented and it will no longer be classified as undefined behavior.

6.2 Specification Ambiguities

There have been moments when interpreting the specifications document has been problematic. This brought intense discussions in our meetings and/or led to the submission of invalid bug reports. Here we discuss three cases that we would like to go through with the OpenMP community.

When using classes in C++, it is a common practice to use the `this` pointer to refer to the current object, or to access methods or data members of the class. There are difficulties for a programmer using OpenMP to use the `this` pointer for setting data environment as `map(to: this)` or `map(to: this->attribute)` to map the object or an attribute. In the former case, the problem is the interpretation of the `this`. It is interpreted as a pointer but as a special expression. This will cause most of the compilers to complain. In the latter case, the operation `this->attribute` is an arbitrary expression and the `map` clause expects a list item that is mappable.

In the second case, we were attempting to test the `private` and `firstprivate` clauses in combination with offload directives. The specifications document [10] in section 2.15.3.3 says that *"Inside the construct, all references to the original list item are replaced by references to the new list item. In the rest of the region, it is unspecified whether references are to the new list item or the original list item."*. When used with device regions, it is not easy to understand what is the meaning of "rest of the region", as there are three different concepts that must be distinguished: region, target region and task region. It has not been possible for us to clearly identify and separate all these regions, and to understand the rest of the paragraph.

Finally, we present an issue with the array section dependencies. When using the `depend()` clause, it is possible to specify array sections in the form of `depend(inout: a[10:5])`. This clause will map 5 elements of the array `a` starting from position 10. However, in the description of the `depend` clause, section 2.13.9, restriction 1 "List items used in depend clauses of the same task or sibling tasks must indicate identical storage locations or disjoint storage locations.". This implies that it is not possible to map array sections that overlap. Additionally, this section mentions that dependencies only take into consideration storage locations, which creates doubts with respect to the difference between specifying an array section, compared to specifying the element where the array section begins. That is, the difference between `depend(inout: a[10:5])` and `depend(inout: a[10])`. We hope that the specification committee would make requirements more explicit.

7 Conclusion and Future Work

Our ongoing work on the OpenMP validation and verification test-suite targets features of the OpenMP 4.5 standard in the order of priority as identified by DOE applications. Particularly the offloading mechanisms for *target devices*. Although the majority of our current set of tests are implemented in C and C++, we plan to have Fortran versions in the near future. Our workflow discussed in Sect. 4 ensures that we make every effort to catch and mitigate implementation or interpretation errors while developing the tests. We try to capture corner cases that we believe might be prone to implementation errors or that are important to applications. Section 5 detailed how to access, execute and customize the test-suite along with how to visualize the results. A website has been built to capture our efforts narrated in this paper https://crpl.cis.udel.edu/ompvvsollve. As of this writing we have not covered the entire OpenMP 4.5 API but we hope that this paper will encourage compiler developers and anyone else interested to use the testsuite and provide feedback. While we implement tests for the remainder of the OpenMP 4.5 offloading and new API we want to begin concurrent dialogue with such *users* of the testsuite to ensure smooth adaptability of the V&V suite.

References

1. Clay, M.P., Buaria, D., Yeung, P.K.: Improving scalability and accelerating petascale turbulence simulations using OpenMP (2017, to appear). http://openmpcon.org/conf2017/program/
2. Friedline, K., Chandrasekaran, S., Lopez, M.G., Hernandez, O.: OpenACC 2.5 validation testsuite targeting multiple architectures. In: Kunkel, J.M., Yokota, R., Taufer, M., Shalf, J. (eds.) ISC High Performance 2017. LNCS, vol. 10524, pp. 557–575. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67630-2_39
3. Grinberg, L., Bertolli, C., Haque, R.: Hands on with OpenMP4.5 and unified memory: developing applications for IBM's hybrid CPU + GPU systems (Part I). In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 3–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_1
4. Grinberg, L., Bertolli, C., Haque, R.: Hands on with OpenMP4.5 and unified memory: developing applications for IBM's hybrid CPU + GPU systems (Part II). In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 17–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_2
5. LLVM: LLVM Testing Infrastructure Guide. https://llvm.org/docs/TestingGuide.html#test-suite-overview
6. Müller, M., Neytchev, P.: An OpenMP validation suite. In: Fifth European Workshop on OpenMP, Aachen University, Germany (2003)

7. Müller, M.S., Niethammer, C., Chapman, B., Wen, Y., Liu, Z.: Validating OpenMP 2.5 for Fortran and C/C++. In: Sixth European Workshop on OpenMP. KTH Royal Institute of Technology, Stockholm, Sweden (2004)
8. Oak Ridge National Laboratory: SUMMIT: scale new heights. discover new solutions. https://www.olcf.ornl.gov/summit/
9. OpenMP: OpenMP - enabling HPC since 1997. http://www.openmp.org/about/openmp-faq
10. OpenMP: OpenMP 4.5 specification. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
11. OpenMP: OpenMP compilers. http://www.openmp.org/resources/openmp-compilers/
12. Richards, D.F., Bleile, R.C., Brantley, P.S., Dawson, S.A., McKinley, M.S., O'Brien, M.J.: Quicksilver: a proxy app for the Monte Carlo transport code mercury. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 866–873. IEEE (2017)
13. Top500: Top500 November 2017 list highlights. https://www.top500.org/lists/2017/11/highlights/
14. University of Delaware and Oak Ridge National Laboratory: OpenMP 4.5 validation and verification suite. https://bitbucket.org/crpl_cisc/sollve_vv/src
15. Wang, C., Chandrasekaran, S., Chapman, B.: An OpenMP 3.1 validation testsuite. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 237–249. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_18
16. Wang, C., Xu, R., Chandrasekaran, S., Chapman, B., Hernandez, O.: A validation testsuite for OpenACC 1.0. In: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW), pp. 1407–1416. IEEE (2014)
17. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: ACM SIGPLAN Notices. vol. 46, pp. 283–294. ACM (2011)

Trade-Off of Offloading to FPGA in OpenMP Task-Based Programming

Yutaka Watanabe[1](✉), Jinpil Lee[2], Taisuke Boku[1,3], and Mitsuhisa Sato[1,2]

[1] Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki, Japan
ywatanabe@hpcs.cs.tsukuba.ac.jp

[2] RIKEN Center for Computational Science, Kobe, Hyogo, Japan
{jinpil.lee,msato}@riken.jp

[3] Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki, Japan
taisuke@cs.tsukuba.ac.jp

Abstract. In High-Performance Computing (HPC), Field Programmable Gate Array (FPGA) is attracting increased attention as an accelerator because its performance has been dramatically improved in recent years. On the other hand, task-based programming recently supported in OpenMP 4.0 enables to expose much parallelism by executing several tasks of the program in the form of a task graph. To accelerate the task-based parallel program by FPGA, it is useful for some dominant tasks frequently executed in parallel to be offloaded to FPGA as an asynchronous FPGA task. We present a performance optimization based on the trade-off between the kernel size and the number of asynchronously executed kernels in parallel in OpenMP task-based programming with FPGA tasks to make use of FPGA hardware resources efficiently. Since a "program" for FPGA is directly converted into the hardware, the hardware resource limitation raises a new issue in optimization on which and how to offload a task to FPGA. Taking task-based block Cholesky factorization as a motivating example, we present the trade-off on how to offload dominant "GEMM" task frequently executed in parallel in the execution of the task-graph. We found that under the limitation of the hardware resource, multiple small kernels are better than a single big high-performance kernel because of higher throughput and higher kernel frequency.

Keywords: Accelerator · FPGA · OpenMP
Task-based programming

1 Introduction

In recent years, in the field of High-Performance Computing (HPC), Field Programmable Gate Array (FPGA) is getting more attracted as an accelerator since the performance of some applications can be dramatically improved.

FPGA is programmable hardware and is especially good if the operations can be fully pipelined. The computation and communication performance have been improved dramatically and are enough to use as an accelerator in HPC. For example, the latest Intel FPGA "Stratix 10" has 10 TFlops of theoretical peak performance in single precision. It has been reported that offloading to FPGA achieves better performance compared with a GPU and CPU in some applications such as Fast Fourier Transform [1].

On the other hand, the recent trend of the many-core processor, task-based programming in OpenMP [2] has gained the interests of developers of parallel applications. It allows users to expose a higher degree of parallelism in tasks compared with traditional work-sharing constructs of OpenMP such as parallel loops. Tasks exposed by task-based programming are scheduled to be executed in each core according to data dependency between them.

OmpSs [3] is the forerunner in task-based programming, and its idea has been transferred into the OpenMP standard. OpenMP 4.0 has the task directive with data dependency for task-based programming. OmpSs supports task-based programming for accelerators like Xeon Phi, GPU, and Xilinx Zynq FPGA. When a task is offloaded to FPGA, the runtime system creates a helper thread that manages FPGA offloading tasks.

In this paper, we present the trade-off between kernel size and the number of kernel instances that are simultaneously executed asynchronously in OpenMP task-based programming with FPGA tasks in order to use FPGA hardware resources efficiently. As a prerequisite, we define a program using OpenMP task-based programming, and then offload specific tasks to FPGA. The OpenMP runtime will create and manage the tasks. We also assume that we have the optimized OpenCL kernel implementation as a prerequisite. Unlike CPU and GPU, a "program" for FPGA is directly converted into the hardware. Thus, one of the important optimization methods is how to use the limited hardware resources of FPGA. In task-based programming, the tasks to be offloaded to FPGA should be carefully generated and scheduled among candidate tasks for efficient utilization of limited FPGA hardware. When the program is decomposed into tasks and task graph, the most dominant task should be chosen for offloading to FPGA. In the example of block Cholesky factorization to be shown later, the task of general matrix multiplication, "GEMM" is the most dominant task. In addition, a large number of similar tasks can often be executed in parallel.

An interesting trade-off shown in this paper is how to offload "GEMM" using FPGA: one complex high-performance kernel which needs large hardware resources, or multiple simple lower performance kernels which need fewer resources for each kernel. In this paper, we focus on the trade-off between the performance of a single kernel and multiple kernels under hardware resources limitation of FPGA for a large number of parallel tasks exposed by task-based programming.

To program FPGA, we used Open Computing Language (OpenCL) [4] as a programming model. OpenCL is an open standard programming framework for heterogeneous systems that use accelerators. Many vendors of CPU, GPU, or

FPGA support the OpenCL framework though each device has unique optimization techniques. We used Intel FPGA SDK for OpenCL [5], which is a toolchain to develop FPGA applications with OpenCL. It mitigates the complexity of traditional development with HDL such as Verilog. There plenty of works to utilize an FPGA in HPC using OpenCL [6,7].

As a result of a real-world application, we found that in the example of task-based block Cholesky factorization, multiple small GEMM kernels outperform single large GEMM kernels in a large number of GEMM tasks offloaded in FPGA because of higher throughput and higher kernel frequency owing to simplicity. Our contributions are as follows:

– We present an asynchronous offloading algorithm to FPGA by exploiting OpenMP task programming model.
– We discuss the trade-off between the complexity of high performance and the throughput under hardware resource limitation of FPGA for the task-based parallel programming with FPGA offloading.
– We demonstrate the effectiveness of this trade-off by implementing asynchronous offloading to FPGA.
– We propose a directive for task offloading to FPGA with controllable hardware parameters.

The rest of this paper is organized as follows. The next section describes the motivation and objective of this research. Section 3 describes the implementations followed by the experimental results in Sect. 4. In Sect. 5, we propose a programming model for task-based FPGA computing. Section 6 shows the related works, and the conclusion and future work are presented in Sect. 7.

2 Motivation and Objective

As described above, we are interested in the trade-off involved when using limited FPGA hardware resources for a large number of parallel tasks exposed by OpenMP task-based programming. We want to offload calculations to FPGA under OpenMP task programming. Our motivating example is a task-based version of block Cholesky factorization, which is a blocking calculation method for Cholesky factorization. Cholesky factorization decomposes from a Hermitian matrix A to lower triangular matrix L and the transposed conjugate matrix $L\dagger$. This is a typical application that can be implemented using the task parallel model.

Block Cholesky factorization consists of following calculations:

– Symmetric rank-k update (SYRK)
– Cholesky factorization (POTRF)
– Solving a triangular matrix equation (TRSM)
– General Matrix Multiplication (GEMM)

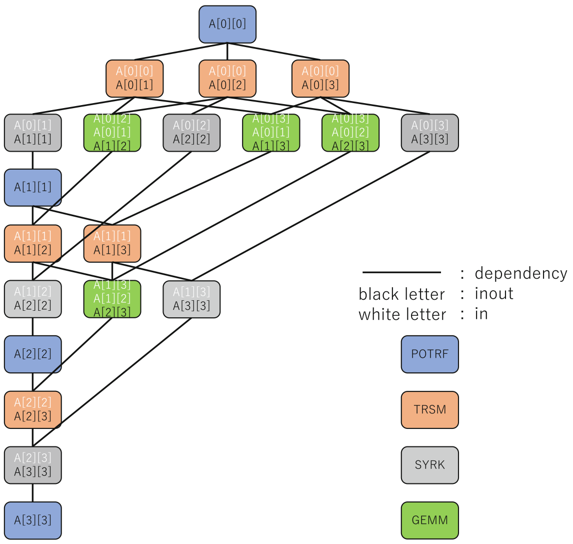


Fig. 1. Task flow of 5 × 5 block cholesky factorization

Figure 1 shows the task flow of 5×5 block Cholesky factorization. Each block presents the task. Each task has in dependency or inout dependency connected with the black line. While there are only four GEMM tasks out of 20, the number of GEMM tasks increase as the number of blocks increase. When the matrix is divided into 32×32 blocks, 4960 out of 5984 tasks are GEMM. Thus, the GEMM task is chosen for offloading to FPGA.

The outline of the task-based version with FPGA is shown in Program 1.1. This is a persuade program with OpenMP and offloading GEMM to FPGA. The *write_data_to_fpga* function is for host-to-FPGA data transfer, the *read_data_from_fpga* function is for FPGA-to-host data transfer, and the *enqueue_request_to_fpga* function is for offloading request to FPGA.

This program assumes that the Hermitian matrix is already copied to the FPGA DDR memory. As shown in Fig. 1, the block data required by GEMM is only updated by the TRSM task. Thus, by updating the data on FPGA in the TRSM task, we can reduce the host-to-FPGA data transfer.

When focusing on the GEMM task, it is suspended by the taskyield clause after executing the *enqueue_request_to_fpga* function and the *read_data_from_fpga* function. While the task is suspended and host executes another task, FPGA performs these operations in the background, which enables overlapping CPU/FPGA computations. When the suspended task is resumed, it checks whether these operations are finished. If not, then the task is suspended again.

```
void cholesky(const int ts, const int nt, float* A[nt][nt])
{
#pragma omp parallel
#pragma omp single
    for (int k = 0; k < nt; k++) {
//create POTRF task
#pragma omp task depend(out:A[k][k])
        {
            omp_potrf(A[k][k], ts, ts);
        }
        for (int i = k + 1; i < nt; i++) {
//create TRSM task
#pragma omp task depend(in:A[k][k]) depend(out:A[k][i])
            {
                omp_trsm(A[k][k], A[k][i], ts, ts);
                write_data_to_fpga(A[k][k], event0);
                waitForFinish(event0);
            }
        }
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
//create GEMM task
#pragma omp task depend(in:A[k][i], A[k][j]) depend(out:A[j][i])
                {
                    enqueue_request_to_fpga(gemmKernel, ..., event1)
    ;
                    do {
#pragma omp taskyield
                        checkStatus(event1, &ret);
                    } while (ret != done);
                    read_data_from_fpga(A[j][i], event2);
                    do {
#pragma omp taskyield
                        checkStatus(event2, &ret);
                    } while (ret != done);
                }
        }
//create SYRK task
#pragma omp task depend(in:A[k][i]) depend(out:A[i][i])
            {
                omp_syrk(A[k][i], A[i][i], ts, ts);
            }
        }
    }
#pragma omp taskwait
}
```

Program 1.1. Outline of block Cholesky factorization with OpenMP

Since the hardware resources of FPGA are limited, determining how to use them is a significant factor. In the example, a large number of GEMM tasks are requested in parallel. Our question here is which is better among the following choices:

– implement a big kernel and handle serially
– implement multiple smaller kernels and handle in parallel

In the typical use of FPGA in HPC applications, a big kernel occupies a large portion of FPGA logic to enhance the computational performance, and it is called sequentially according to the host program steps. Since the kernel invocation to FPGA can be controlled asynchronously from the host CPU through the PCIe interface, it is logically possible to run multiple kernel instances of a device program (e.g., GEMM routine) asynchronously in parallel if the application implies such parallelism. There is a possibility of optimization based on the trade-off between the “single big kernel” and “parallel small kernels” on FPGA.

We used OpenCL described in Sect. 3.1 for implementation. We used the SIMD length of OpenCL kernel described in Sect. 3.2 as a parameter and controlled the resource utilization of a single kernel. If the kernel is small, we replicated it in the logic element space as far as possible. The possible FPGA kernel frequency of small kernels will be higher than that of the big kernel because of the simplicity. Thus, the small kernels might be able to handle many tasks faster than the big kernel.

We claim that, as a result of exposing parallelism in task-based programming, a large number of the same kind of simple tasks are requested in parallel in several applications, as shown in our example.

3 Implementation

In this section, we describe our implementation.

3.1 OpenCL

Figure 2 presents the overview of OpenCL offloading flow. The figure assumes that one kernel is implemented and one Command Queue is prepared. The Command Queue is a queue that manages offloaded tasks. The host program enqueues offloading requests into the queue with the host API named “clEnqueueNDRangeKernel”. If the kernel is ready to be executed, then one request is offloaded into the kernel and executed. After that, the host calls the API “clWaitForEvents” or “clGetEventInfo” to check whether the task is completed. “clWaitForEvents” is an API that waits for the completion of tasks. “clGetEventInfo” is an API that checks the task status.

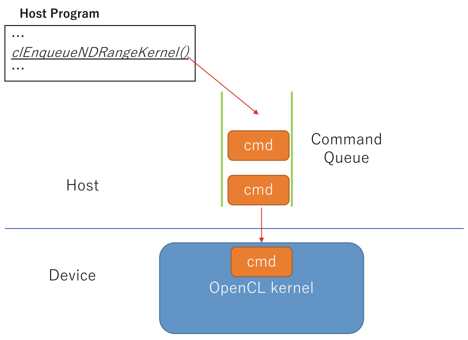


Fig. 2. Offloading flow in OpenCL

OpenCL has two kernel models: single work-item and NDRange. The single work-item kernel model creates the kernel that calculates all data as one task, just like using a single CPU core without parallelism. The NDRange model creates the kernel that calculates data in parallel, like using multiple CPU cores with data parallelism.

3.2 Intel FPGA SDK for OpenCL

For the implementation, we used Intel FPGA SDK for OpenCL, which is a toolchain for developing FPGA applications with OpenCL [15,16]. The toolchain includes an OpenCL-to-Verilog HLS compiler. Programmers do not have to consider peripheral circuits like memory controllers or PCIe controllers because the vendor provides the Board Support Packages that contain them.

The toolchain has some extensions to the OpenCL specification. An example is the num_simd_work_items attribute. The attribute can be used in the NDRangeKernel model and allows SIMD operation. The extensions are necessary for optimizing the OpenCL program for Intel FPGA.

3.3 Implementation of GEMM OpenCL Kernel

The OpenCL kernel that calculates the offloaded GEMM task uses the NDRange kernel model. The kernel is implemented using a blocking algorithm. The block size is fixed to 64×64 in our implementation. All blocks are loaded to M20K BRAM to reduce access to global memory (DDR memory). In addition, by using the num_simd_work_items attribute as a parameter, we can control the resource utilization of the kernel. In this way, we can implement one big kernel or multiple small kernels to FPGA. As another type of optimization, we specify memory the bank for each buffer explicitly because we know the access pattern of GEMM operation.

3.4 Task-Based Programming with Asynchronous Offloading to FPGA

GEMM task created by OpenMP task offloads GEMM calculation to FPGA.

After offloading to FPGA with "clEnqueueNDRangeKernel" API, the host suspends the GEMM task and switches to another task using the taskyield operation provided by OpenMP. The offloading request is waiting in a queue and is eventually scheduled to run while the host GEMM task is suspended. When the suspended GEMM task resumes, it checks whether the offloading FPGA request is completed. If completed, the host reads the result from FPGA and exits. If not, it switches to another task again.

When multiple kernels are implemented on FPGA, the host offloads them as described in Fig. 3. Each device kernel is connected to the independent Command Queue. The requests are scheduled into the queue of each kernel in a round-robin manner. The execution of each kernel does not affect other kernels and executed asynchronously.

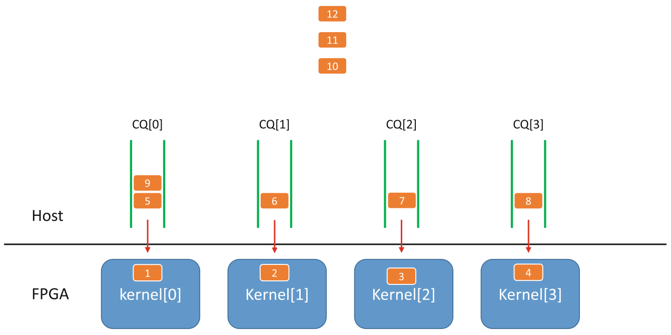


Fig. 3. Enqueue requests with multiple kernels

We found that the taskyield directive does not work properly in Intel OpenMP since it sometimes does not cause the task switch effectively. This is because the OpenMP specification allows the exact action of taskyield to be implementation dependent. In our experiment, we implemented an own task scheduling system and described the task-based program based on Program 1.1. Figure 4 presents the overview of this system.

First, OS thread is bound to CPU core. The OS threads have a user-level thread queue. The runtime creates software-based user-level thread shown as the orange rectangle, which corresponds the tasks in OpenMP, and scheduling into the user-level thread queue. The OS thread monitors the user-level thread queue. If there is a task, then the OS thread pick it up from the head of the queue and executing. If the thread requests taskyield operation, then the OS thread suspends it and push it back to the tail of the queue. After that, next task will be executed. These operations like execution and suspension are not depending

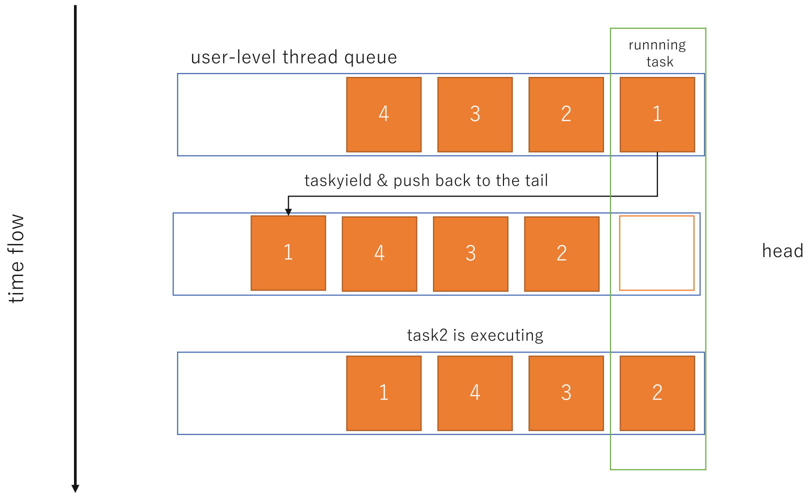


Fig. 4. Overview of task queue implementation

on the OS threads and implemented with user level API. We used Argobots [8], which is a lightweight threading and tasking framework developed by Argonne National Laboratory, to implement this system. Nevertheless, we could execute the same code by OpenMP if the taskyield was properly implemented.

4 Evaluation

4.1 Evaluation Environment

We used the Pre-PACS version 10 (PPX) system at Center for Computational Sciences of University of Tsukuba for evaluation. Table 1 shows the details of the PPX environment. The system has two CPU sockets with Intel Broadwell and Bittware's A10PL4 FPGA [9] board. The FPGA board has Intel's Arria10 FPGA [10] and two 4 GB memory banks. This system has GPU and InfiniBand HCA, but we did not use them in the evaluation.

Table 1. Evaluation environment

CPU	Xeon E5-2660 v4 @ 2.00 GHz x 2
Host DRAM	DDR4-2400 16 GB x 4
GPU	NVIDIA Tesla P100 PCIe x 2
FPGA Board	BittWare A10PL4
FPGA	Intel Arria 10 (10AX115N3F40E2SG)
FPGA DRAM	DDR4-2133 4 GB x 2
InfiniBand	Mellanox ConnectX-4 EDR
OS	CentOS 7.3 64bit
FPGA Compiler	Intel FPGA SDK for OpenCL 17.1.2
CPU Compiler	Intel C Compiler 18.0.1
CPU BLAS Library	Intel Math Kernel Library
Thread Library	Argobots 1.0b1

4.2 Resource Utilization

The numerical calculation performance of a GEMM kernel is theoretically defined by SIMD width. Then, we defined three types of kernels according to the SIMD width. Each kernel type uses the same OpenCL kernel using the NDRangeKernel model, except the SIMD length and the number of replications. SIMD lengths of Type1, Type2, and Type3, are 16, 8, and 4 respectively. Type1 has one replication, while Type2 and Type3 have two replications and four replications respectively. When several kernel instances are implemented like Type3, they can be executed in parallel.

Table 2. Resource utilization

	SIMD length	Frequency	ALMs	DSP	BRAM
Type1	16	164.71 MHz	25%	69%	60%
Type2	8	179.59 MHz	27%	69%	69%
Type3	4	213.94 MHz	31%	70%	90%

It should be noted that the kernel frequency of Type1 is 164 MHz, and the kernel frequency of Type3 increases to 213 MHz. These frequencies are low because of the lack of optimization. With respect to Digital Signal Processing (DSP) utilization, the ratio of each kernel type is about 70%. By reducing Block RAM (BRAM) usage, we can utilize more DSP resources, which will enhance kernel performance. We can generate more kernels by reducing the SIMD length as in from type1 to Type2 and 3 (Table 2).

4.3 Basic Characteristics of GEMM Kernels

First, we measured latency and throughput of each type of GEMM kernel as basic characteristics in single precision floating point. Latency includes the kernel execution time and kernel invocation time through PCIe interface. We only use single kernel instance even if the device program has several kernel instances like Type3.

Figure 5 shows the latency of each type with four different matrix sizes. The vertical axis is shown in log scale. Type1, which has the widest SIMD length among three types, has the minimum latency. On the other hand, the latency of Type3 becomes eight times as the one side of the matrix becomes double while the latencies of Type1 and Type2 increase by less than eight times. The performance of Type3 is stable if the matrix size is 512×512 or more while the performance of Type1 and Type2 depends on the matrix size.

Figure 6 shows the throughput for each type kernel when executing a large number of GEMM operations. If multiple kernel instances are implemented such as Type3, we used all kernel instances in the throughput result. When the matrix is small as 512×512, the performance of Type1 and Type3 are almost same. However, when the matrix size increases, Type3 achieves the highest performance, while Type1 has the lowest throughput. One of the reasons for the difference in performance is the difference in kernel frequency. Note that the performance of Type3 is 350 GFlops at best while the theoretical performance is over 400 GFlops. We guess that this is due to the lack of optimization for task management with OpenCL. By improving task management, Type3 might be improved to obtain better performance.

These results show that when offloading a large number of task to FPGA, the performance of many small kernels is better than that of the big kernel in terms of throughput.

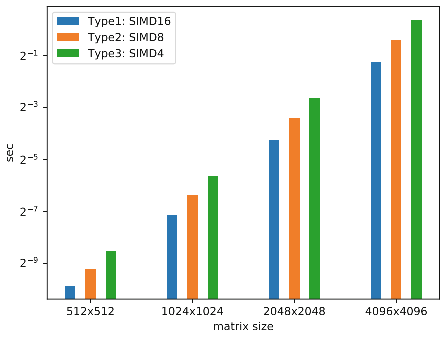


Fig. 5. Latency of OpenCL kernels

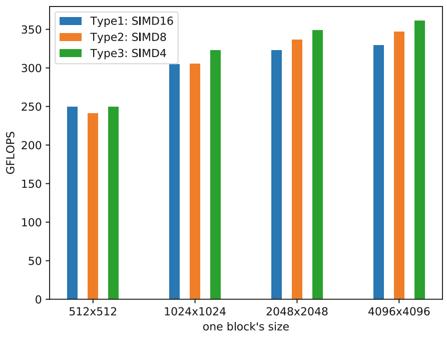


Fig. 6. Throughput of OpenCL kernels

4.4 Result of Block Cholesky Factorization by Offloading to FPGA

Figure 7 shows the result of block Cholesky factorization shown in Program 1.1 with offloading GEMM operation to FPGA in single-precision floating point. The vertical axis indicates the overall performance, and the horizontal axis presents the DSP utilization. When implementing multiple kernels like Type3, the number of DSPs shows with the number of kernel instances used. The Hermitian matrix size is 32768×32768, and divided into 32×32 blocks.

As shown in the figure, Type3 achieves the highest performance among the three types. The performance of Type3 is about 100 GFlops higher than Type1 even though the throughput difference between each of these kernels is approximately 30 GFlops. The result indicates that there may be another reason for the performance difference, and not just the difference in kernel frequency. We

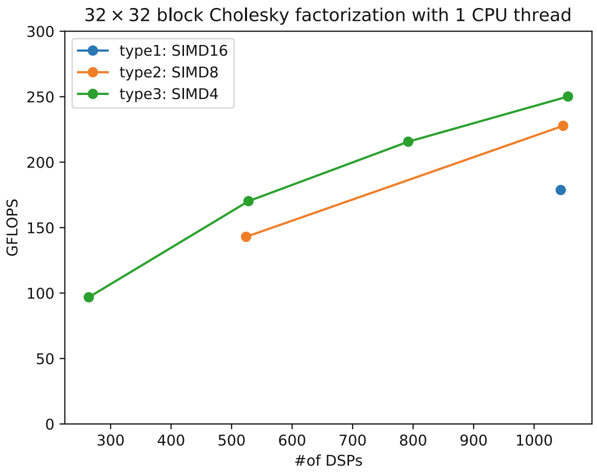


Fig. 7. Result of block Cholesky factorization with FPGA

are now working on investigating the reason. Note that in Type3, the more kernels we used, the better the performance. However, it did not scale well as we expected. One of the reasons may be the task scheduling problem. This will be improved by implementing a priority queue for tasks that offload to FPGA asynchronously using taskyield operation.

Overall, the results indicate that implementing several small kernels is better than the big kernel in the example of block Cholesky factorization.

5 A Proposal of OpenMP Directive for Offloading to FPGA

In this section, we define the interface offloading to FPGA using OpenCL from OpenMP directive. We propose an extension of the OpenMP 4.5 directive for offloading FPGA kernels in task-based programming. We assume that we have a task-based program executing on CPU, and the optimized OpenCL device code is already compiled and synthesized for FPGA. The OpenCL kernel has four instances. This is because the code translation algorithm from OpenMP in C/C++ to OpenCL for Intel FPGA is under development. Program 1.2 shows the OpenMP program with our proposal. We propose one directive: "alias target" and two clauses: "work_size" and "async" for OpenMP specification.

First, we discuss the "alias target" directive. Originally, Lee et al. proposed the alias simd directive [11], which specifies the SIMD implementation of the target function explicitly. If we want to offload the specific task to FPGA, we have to not only adding the target directive but also renaming the existing function name to OpenCL function name. For example, we should rename the "func" to "fpga_func" at line 19. To avoid modifying the existing code, we introduce "alias target" directive to replace the existing function with the explicit OpenCL implementation. In the example Program 1.2, we assume that the name of synthesized OpenCL function is "fpga_func." At line 3, the program aliases "fpga_func" to "func', which is an existing function name found at line 19. At line 18, we inserted the target directive before calling "func" function. This means the operation of "func" will offload to FPGA using "fpga_func". If we do not insert the target directive, the program calls the existing "func" function.

The "num_units" clause indicates the number of the instances of OpenCL kernel we use on FPGA. In this case, we can use up to four instances because the OpenCL kernel has four small instances. This clause is used in "alias target directive". As shown in the evaluation of block Cholesky factorization, we achieved a better performance using several small kernels than the single big kernel with asynchronous offloading. Since there is no way of describing such semantics, we added the "num_units" clause to the "alias target" directive. In the Program 1.2 we use the clause at line 3 and indicate that host will offload the tasks using four instances on FPGA.

The "work_size" clause indicates the size of work-items. The value of this clause is used in the invocation of "clEnqueueNDRangeKernel" API. Since there is no similar clause that describes the problem size in OpenMP specification, and

```
#define SIZE 1024

#pragma omp alias target to(func) num_units(4)
void fpga_func(
    global float* restrict A,
    global float* restrict B,
    global float* restrict C,
    const int size
);

---

#pragma omp parallel
{
#pragma omp single
    {
        for (int i=0; i<nt; i++) {
#pragma omp target map(to:A[i][:]) map(to:B[i][:]) map(tofrom:C[i
    ][:]) work_size(SIZE, SIZE) async
            func(A[i], B[i], C[i], SIZE);
        }
    }
#pragma omp taskwait
}
```

Program 1.2. Example of proposed OpenMP directive for offloading to FPGA

we assume to use synthesized OpenCL kernel, we need a new clause to describe the size of offload task. The value in the "work_size" clause is an extended list in order to adapt the 3-dimensional parallelism used in OpenCL kernel. Note that local work size determined in the kernel compilation time is also required in the invocation of "clEnqueueNDRangeKernel". However, we do not need to specify because it could be found using "clGetKernelWorkGroupInfo" API.

At the last, we discuss "async" clause. The compiler will translate Program 1.2 to the invocation of OpenCL Host runtime and OpenMP task. In this example, we add the "async" clause to target directive. It indicates the offloading is processing asynchronously as described in Sect. 3.4. Once the offloading is requested to FPGA, it suspends with OpenMP taskyield operation. When it resumes, it checks whether the offloading task is finished using the "clGetEventInfo" API. If not finished, it suspends again. If finished, the task reads the result data from FPGA. Since there is no way of describing such semantics, we added the "async" clause to the target directive.

6 Related Works

OpenARC [12] is a research compiler developed by Oak Ridge National Laboratory. It supports offloading to accelerators like GPU and Intel FPGA. OpenARC supports OpenACC directives to program FPGA, and it translates the OpenACC code into OpenCL host code and OpenCL device kernel [1]. This compiler supports OpenCL extension for Intel FPGA by adding some new clauses to OpenACC. This compiler uses synchronous offloading.

OmpSs [3] is an extended OpenMP programming model developed by Barcelona Supercomputing Center. In addition to OpenMP, OmpSs supports accelerators like Xeon Phi, GPU, and Xilinx Zynq FPGA. Zynq FPGA is a System-on-Chip platform with the ARM processor. In the point of FPGA, OmpSs uses OmpSs programming model for the frontend and translates to Vivado HLS program [13,14]. Vivado is an HLS toolchain for Xilinx FPGA. Programmers should define the target function with Vivado oriented optimization techniques. The OmpSs runtime creates a helper thread that manages FPGA offloading tasks.

Unlike from OpenARC and OmpSs, we used explicit asynchronous offloading so that users can overlap CPU/FPGA computations. In this paper, we focus on the differences in implementation between the big kernel and multiple small kernels.

7 Conclusion and Future Work

In contrast to CPU and GPU, the limitation of hardware resources raises a new issue in optimization since a “program” for FPGA is directly converted into the hardware. In this paper, we pointed out a trade-off between the complexity of high performance and the throughput under the limitation of the hardware resource of FPGA for OpenMP task-based parallel programming with FPGA offloading.

Taking task-based block Cholesky factorization as a motivating example, we demonstrated the trade-off in offloading dominant “GEMM” tasks frequently executed in parallel in the execution of the task graph. We found that under the hardware resource limitation, multiple small kernels are better than a single big high-performance kernel because of higher throughput and higher kernel frequency. It should be noted that multiple small kernels can be executed at the higher kernel frequency, resulting in higher performance and better resource utilization.

Our future work will be to apply our technique to other applications. We expect that by exposing parallelism in task-based programming, several applications can execute a large number of the same kind of simple tasks in parallel;hense, our technique will contribute to improving their performance.

References

1. Lee, S., Kim, J., Vetter, J.S.: OpenACC to FPGA: a framework for directive-based high-performance reconfigurable computing. In: 2016 IEEE International Parallel and Distributed Processing Symposium, pp. 544–554. IEEE (2016)
2. OpenMP. http://www.openmp.org/
3. The OmpSs Programming Model. https://pm.bsc.es/ompss
4. OpenCL Overview. https://www.khronos.org/opencl/
5. Intel FPGA SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

6. Zohouri, H.R., Maruyama, N., Smith, A., Matsuda, M., Matsuoka, S.: Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 35. IEEE Press (2016)
7. Kobayashi, R., Oobata, Y., Fujita, N., Yamaguchi, Y., Boku, T.: OpenCL-ready high speed FPGA network for reconfigurable high performance computing. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, pp. 192–201. ACM (2018)
8. Argobots. http://www.argobots.org/
9. A10PL4 PCIe FPGA Board. https://www.bittware.com/fpga/intel/boards/a10pl4/
10. Arria 10 FPGA. https://www.altera.com/products/fpga/arria-series/arria-10/overview.html
11. Lee, J., Petrogalli, F., Hunter, G., Sato, M.: Extending OpenMP SIMD support for target specific code and application to ARM SVE. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 62–74. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_5
12. Open Accelerator Research Compiler. http://ft.ornl.gov/research/openarc
13. Filgueras, A., et al.: OmpSs@Zynq all-programmable SoC ecosystem. In: Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 137–146. ACM (2014)
14. Bosch, J., Filgueras, A., Vidal, M., Jimenez-Gonzalez, D., Alvarez, C., Martorell, X.: Exploiting parallelism on GPUs and FPGAs with OmpSs. In: Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems, p. 4. ACM (2017)
15. Intel FPGA SDK for OpenCL Programming Guide. https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
16. Intel FPGA SDK for OpenCL Best Practices Guide. https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf

OpenMP Improvements and Innovations

Compiler Optimizations for OpenMP

Johannes Doerfert(✉) and Hal Finkel

Argonne Leadership Computing Facility, Argonne National Laboratory,
Argonne, IL 60439, USA
{jdoerfer,hfinkel}@anl.gov

Abstract. Modern compilers support OpenMP as a convenient way to introduce parallelism into sequential languages like C/C++ and Fortran, however, its use also introduces immediate drawbacks. In many implementations, due to early outlining and the indirection though the OpenMP runtime, the front-end creates optimization barriers that are impossible to overcome by standard middle-end compiler passes. As a consequence, the OpenMP-annotated program constructs prevent various classic compiler transformations like constant propagation and loop invariant code motion. In addition, analysis results, especially alias information, is severely degraded in the presence of OpenMP constructs which can severely hurt performance.

In this work we investigate to what degree OpenMP runtime aware compiler optimizations can mitigate these problems. We discuss several transformations that explicitly change the OpenMP enriched compiler intermediate representation. They act as stand-alone optimizations but also enable existing optimizations that were not applicable before. This is all done in the existing LLVM/Clang compiler toolchain without introducing a new parallel representation. Our optimizations do not only improve the execution time of OpenMP annotated programs but also help to determine the caveats for transformations on the current representation of OpenMP.

Keywords: OpenMP · Compiler optimizations · Alias analysis
Variable privatization · Barrier elimination
Communication optimization

1 Introduction

In the LLVM/Clang compiler toolchain [9] the use of OpenMP allows for parallel execution but also introduces immediate drawbacks. Due to early outlining and the indirection though the OpenMP runtime, the Clang front-end introduces an optimization barrier that is impossible to overcome by common

The U.S. government retains certain licensing rights. This is a U.S. government work and certain licensing rights apply middle-end optimizations. As a consequence, the OpenMP-annotated program parts do not benefit from classic compiler transformations like constant propagation or loop invariant code motion. Analysis results, especially alias information, is severely degraded in the parallelized program parts. For these and similar reasons, researchers, industry as well as the general LLVM community are currently looking into alternative representations of parallelism in a modern compiler toolchain [11,14].

In order to provide immediate benefit, and to create a meaningful baseline for these efforts, we investigated the feasibility and limitations of program optimizations based on the current representation of OpenMP programs. In the following we present five distinct program transformations that required reasonable implementation, and thereby maintainability effort. They do however enable some of the most important compiler optimizations. In the process, we were able to determine the caveats for transformations on the current representation of OpenMP programs, which use explicit calls to the OpenMP runtime and early outlined parallel program parts.

The rest of this report is organized as follows. We first present necessary background for our work in Sect. 2. Afterwards, Sect. 3 through Sect. 7 describe the new parallel centric optimizations we introduced in the LLVM pipeline. In Sect. 8 we show preliminary evaluation results of some of these optimizations on different OpenMP benchmarks taken from the Rodinia benchmark suite [4] as well as the LULESH v1.0 benchmark [7]. Finally, we discuss related work in Sect. 9 and conclude in Sect. 10.

2 Background

Clang, the C/C++ front-end of the LLVM compiler framework, immediately lowers OpenMP constructs to runtime library calls. The code in parallel regions (`#pragma omp parallel` and similar) is outlined into separate functions. Their addresses as well as all communicated values (`shared` and `firstprivate` clauses) are then passed to a runtime library call. Inside this library function the outlined parallel region is invoke by each thread in the OpenMP thread team. Due to this opaque indirection, passes that work on LLVM's low-level intermediate representation (LLVM-IR) do not need to be aware of any parallel, or OpenMP specific, semantic. For an example consider the code in Fig. 1a which is lowered by Clang to LLVM-IR similar to the pseudo C code shown in Fig. 1b.

This early outlining approach allows rapid integration of new features and bears little risk of miscompilations due to the function level abstraction and the indirection through the runtime library. Though, this approach will inevitably prevent any optimization to cross the boundary between sequential and parallel code as long as the semantics of the runtime library are not explicitly encoded.

In this work we present several compiler transformations which are aware of the semantics of runtime library calls that implement OpenMP parallel constructs. These transformations are designed to be also applicable to OpenMP tasks runtime calls as well as other parallel language (extensions) expressed in LLVM-IR.

3 Optimization I: Attribute Propagation

Programmers employ attributes, e.g., `const` or `restrict`, to encode domain knowledge in the source code. This is an explicit contract between the programmer and the compiler that limits the set of defined execution traces in the hope of better transformations. Similarly, the compiler might employ attributes to manifest knowledge that was inferred by analyses passes.

The arguably most important use case for attributes are caller-callee boundaries, especially for intra-procedural analyses. Attributes for function parameters provide information about the otherwise unknown inputs while the potential effects of function calls are limited through attributes at the call-site arguments and at the called function.

To improve attributes at the caller-callee boundary of indirectly called parallel program parts, we created an LLVM propagation pass. It communicates the following attributes between pointer arguments in the sequential context and parameter declarations of the parallel work function:

– The absence of pointer capturing[1].
– The access behaviour, thus read-only or write-only.
– The absence of aliasing pointers that are accessible by the callee.
– Alignment, non-nullness and dereferencability information of the pointer.

While all but aliasing information can be simply propagated from the (indirect) call site to the parameter declaration and vice versa, the coarse grained nature of the no-alias attributes, e.g., `restrict` in C/C++ and `noalias` in LLVM's IR, complicates propagation. Even if pointers are known to be alias free in the (sequential) code preceding the parallel region, the `restrict` or `noalias` attribute cannot be simply placed at the parameter declaration to convey this information. First, other arguments could be derived from the alias free pointer which would introduce aliasing opportunities in the parallel work function. Second, the attributes will break dependences that cross barriers thereby allowing code motion across these sequencing constructs. An example to showcase the second problem is given in Fig. 1. The OpenMP annotated C source code in Fig. 1a is translated by the front-end to LLVM's IR corresponding to the pseudo code shown in Fig. 1b. Since the parameter `int*` is known to be alias free in the (sequential) context of `foo` we want to `restrict` qualify it in the parallel work function as shown in Fig. 1c and d. This qualification will break the dependence between the accesses to `p` and the call to `bar`, allowing the store-load forwarding performed in Fig. 1c. However, `bar` could contain a barrier which would require all increments to be performed prior to any multiplication.

To enable optimizations in the outlined parallel work function we still want to propagate alias information. However, existing analysis and optimization passes might change the semantics if we propagate `restrict`/`noalias` attributes to non-read-only argument pointers. For such alias-free pointer arguments, we have to ensure that their dependences with potential barriers are not eliminated. Since

[1] A pointer is captured if a copy of it is made inside the callee that might outlive it.

the semantics of both `restrict` and `noalias` are defined based on accesses through the syntactic pointer expression, we can prevent any unsound transformation by providing potential barriers `access` to this syntactic expression. This representation is illustrated in Fig. 1d. All existing compiler analyses and transformations have then to assume the memory pointed to can be inspected and modified by the potential barrier, ensuring the original memory state when a (potential) barrier is executed.

```
int foo() {
  int a = 0;
#pragma omp parallel shared(a)
  {
#pragma omp critical
    { a += 1; }

    bar();

#pragma omp critical
    { a *= 2; }
  }
  return a;
}
```

(a) OpenMP annotated C source input featuring a call to an unknown function `bar` inside the parallel region.

```
int foo() {
  int a = 0;
  int *restrict p = &a;
  omp_parallel(pwork, p);
  return a;
}
void pwork(int tid, int *p) {
  if (omp_critical_start(tid)) {
    *p = *p + 1;
    omp_critical_end(tid);
  }
  bar();
  if (omp_critical_start(tid)) {
    *p = *p * 2;
    omp_critical_end(tid);
  }
}
```

(b) Pseudo C-style representation of the lowered LLVM-IR produced by Clang for the input in Figure 1a.

```
void pwork(int tid,
           int *restrict p) {
  if (omp_critical_start(tid)) {
    omp_critical_end(tid);
  }
  bar();
  if (omp_critical_start(tid)) {
    *p = 2 * (*p + 1);
    omp_critical_end(tid);
  }
}
```

(c) Unsoundly transformed work function after alias information propagation *if* the call to `bar` contains a barrier.

```
void pwork(int tid,
           int *restrict p) {
  if (omp_critical_start(tid)) {
    *p += 1;
    omp_critical_end(tid);
  }
  bar()[p]; // May "use" pointer p.
  if (omp_critical_start(tid)) {
    *p *= 2;
    omp_critical_end(tid);
  }
}
```

(d) Sound representation after alias information propagation with a pretended use by the potential barrier call `bar`.

Fig. 1. Example illustrating the problematic propagation of `restrict` or `noalias` information from the parallel region context to the parallel work function.

4 Optimization II: Variable Privatization

Writing OpenMP code involves the tedious and error-prone classification of all variables declared outside and used inside the parallel region. Since this classification can have a crucial performance impact we provide a transformation that reclassifies the variables based on their actual usage. Our optimization is performed on low-level LLVM IR and aims to improve both the sequential code as well as the parallel work function. In part, the transformation can be interpreted as strengthening of the OpenMP clauses described below from top to bottom:

– Shared, which indicates any modification might be visible to other threads as well as after the parallel region.
– Firstprivate, which is a private variable but initialized with the value the variable had prior to the parallel region.
– Private, which is a thread-local uninitialized copy of the variable, thus similar to a shadowing re-declaration in the parallel region.

Note that the clause strengthening from `shared` or `firstprivate` to `private` allows the use of separate variables for the sequential and parallel program parts, which enables additional optimizations in both parts. This kind of variable privatization is legal if all of the below stated legality conditions hold:

– The variable is (re-)assigned on all paths from the end of the parallel region that might reach a use,
– Each use of the variable inside the parallel region is preceded by an assignment that is also part of the parallel region.
– There is no potential barrier between the use of a variable and its last preceding assignment.

In addition, we try to communicate variables by-value instead of by-reference. This is sound if they are live-in (`firstprivate` or `shared`) but not live-out nor used for inter-thread communication. Thus, if only the first and last of the above conditions holds we pass the value of the variable instead of the variable container, e.g., the stack allocation.

Finally, non-live-out variables that might be used for communication inside the parallel region can be privatized prior to the parallel region. Hence, if the first of the above conditions holds we replace the variable in the parallel region with a new one declared in the sequential code. This new one is initialized with the value of the original variable just prior to the parallel region. This transformation decouples the variable uses in the two code region and thereby allows for further optimization of the original one in the sequential part.

Note that all transformations have to be aware of potential aliases that could disguise a user variable. In addition, by-value privatization requires the involved type changes to be legal and potentially even a register file transfer[2].

[2] The kmpc OpenMP library used by LLVM/Clang communicates variables via variadic functions that require the arguments to be in integer registers. When a floating point variable is communicated by-value instead of by-reference we have to insert code that moves the value from a floating point register into an integer register prior to the runtime library call and back inside the parallel function.

5 Optimization III: Parallel Region Expansion

Parallel regions introduce an optimization barrier at their boundary. In addition, the start and end of parallel execution can, depending on the hardware, add significant cost. As an example consider the code shown in Fig. 2a.

In each iteration of the sequential outer loop two *new* OpenMP thread teams are started to work on the current value of `ptr`, first in forward and then in backward direction. Due to the early outlining, there is no analysis information transfer between the outer loop and the parallel regions nor between the two parallel loops. Furthermore, starting and ending the task teams will eventually accumulate non-trivial cost on the critical path. To decrease this cost and to improve intra-procedural analyses we extend adjacent parallel regions as shown in Fig. 2b.

To eliminate the task spawning overhead further, the parallel section can be expanded around sequential constructs as well. This is only possible if the sequential constructs can be guarded appropriately and they do not interfere with the parallel semantics, i.a., they do not throw exceptions. The final code after parallel region expansion is illustrated in Fig. 2c.

If a new expanded parallel region is created the contained existing parallel regions are flattened. Thus, the original `#pragma omp parallel` annotations, or alternatively the indirection through the corresponding runtime calls, contained in the extended region are removed. Since there is an implicit barrier at the end of a parallel region, we insert an explicit `#pragma omp barrier`, or an appropriate runtime call, when parallel regions are flattened. This allows later passes to remove the former implicit, and thereby irremovable, barriers. However, it is important to note that the expanded parallel region will introduce a new implicit barrier at its end.

Accounting for the two implicit barriers after the parallel for loops, the number of barriers in this example increased by one compared to the original code. However, there is now only one parallel region that starts a thread team and there is far more context in the parallel region to enable further optimizations.

6 Optimization IV: Barrier Elimination

Barriers are synchronization points that can be placed manually by the programmer or occur implicitly due to the use of certain OpenMP annotations, i.e., `#pragma omp parallel for` without the `nowait` clause. Since synchronization can significantly increase the runtime it should always be used with caution. However, the minimal placement of barriers is an inherently hard and error-prone task even for expert programmers. Since precise dependency information is required to argue about the need for a barrier, and program transformations might be necessary to obtain such information, compilers are well suited to perform this task. To this end, we implemented an OpenMP barrier elimination pass that uses alias information to remove redundant barriers. A barrier is considered redundant if there is no dependence crossing it, thus from the code after to the

```
while (ptr != end) {
  #pragma omp parallel for firstprivate(ptr)
  for (int i = ptr->lb; i < ptr->ub; i++)
    forward_work(ptr, i);
  #pragma omp parallel for firstprivate(ptr, a)
  for (int i = ptr->ub; i > ptr->lb; i--)
    backward_work(ptr, a, i - 1);
  ptr = ptr->next;
}
```

(a) Example featuring two adjacent parallel regions each containing a parallel for loop.

```
while (ptr != end) {
  #pragma omp parallel for firstprivate(ptr, a)
  {
    #pragma omp for firstprivate(ptr) nowait
    for (int i = ptr->lb; i < ptr->ub; i++)
      forward_work(ptr, i);
    #pramga omp barrier // explicit loop end barrier
    #pragma omp for firstprivate(ptr, a) nowait
    for (int i = ptr->ub; i > ptr->lb; i--)
      backward_work(ptr, a, i - 1);
    #pramga omp barrier // explicit loop end barrier
  }
  ptr = ptr->next;
}
```

(b) Expanded version of the code shown in Figure 2a with a parallel region containing two adjacent parallel for loops.

```
#pragma omp parallel shared(ptr) firstprivate(a)
{
  while (ptr != end) {
    #pragma omp for firstprivate(ptr) nowait
    for (int i = ptr->lb; i < ptr->ub; i++)
      forward_work(ptr, i);
    #pramga omp barrier // explicit loop end barrier
    #pragma omp for firstprivate(ptr, a) nowait
    for (int i = ptr->ub; i > ptr->lb; i--)
      backward_work(ptr, a, i - 1);
    #pramga omp barrier // explicit loop end barrier
    #pragma omp master
    { ptr = ptr->next; }
    #pramga omp barrier // barrier for the guarded access
  }
}
```

(c) Final code after parallel region expansion. The two adjacent parallel for loops as well as the sequential pointer chasing loop are now contained in the parallel region.

Fig. 2. Example to showcase parallel region expansion.

code prior. Note that our transformation is intra-procedural and therefore relies on parallel region expansion (ref. Fig. 5) to create large parallel regions with explicit barriers. In the example shown in Fig. 2c it is possible to the eliminate the barrier between the two work sharing loops if there is no dependence between the `forward_work` and `backward_work` functions, thus if they work on separate parts of the data pointed to by `ptr`. If this can be shown, the example is transformed to the code shown in Fig. 3. Note that the explicit barrier prior to the *#pragma omp master* clause can always be eliminated as there is no *inter-thread* dependence crossing it.

```
#pragma omp parallel shared(ptr) firstprivate(a)
{
  while (ptr != end) {
    #pragma omp for firstprivate(ptr) nowait
    for (int i = ptr->lb; i < ptr->ub; i++)
      forward_work(ptr, i);
    #pragma omp for firstprivate(ptr, a) nowait
    for (int i = ptr->ub; i > ptr->lb; i--)
      backward_work(ptr, a, i - 1);
    #pragma omp master
    ptr = ptr->next;
    #pramga omp barrier // synchronize the guarded access
  }
}
```

Fig. 3. Example code from Fig. 2 after parallel region expansion (ref. Fig. 2c) and consequent barrier removal.

7 Optimization V: Communication Optimization

The runtime library indirection between the sequential and parallel code parts does not only prohibit information transfer but also code motion. The arguments of the runtime calls are the variables communicated between the sequential and parallel part. These variables are determined by the front-end based on the code placement and capture semantics, prior to the exhaustive program canonicalization and analyses applied in the middle-end. The code in Fig. 5a could, for example, be the product of some genuine input after inlining and alias analysis exposed code motion opportunities between the parallel and sequential part. Classically we would expect `K` and `M` to be hoisted out of the parallel loop and the variable `N` to be replaced by `512` everywhere. While the hoisting will be performed if the alias information for `Y` has been propagated to the parallel function (ref. Sect. 3), the computation would not be moved into the sequential code part and `N` would still be used in the parallel part. Similarly, the beneficial[3] recompute of `A` inside the parallel function (not the parallel loop) will not happen as

[3] Communication through the runtime library involves multiple memory operations per variable and it is thereby easily more expensive than one addition for each thread.

```
__attribute__((const)) double norm(const double *A, int n);

void norm(double *restrict out, const double *restrict in, int n) {
  #pragma omp parallel for shared(out, in) firstprivate(n)
  for (int i = 0; i < n; ++i)
    out[i] = in[i] / norm(in, n);
}
```

Fig. 4. Parallel loop containing an invariant call to `norm` that can be hoisted.

no classic transformation is aware of the "pass-through" semantic of the parallel runtime library call.

The code motion problem for parallel programs is already well known since the example shown in Fig. 4 is almost the motivating example for the Tapir parallel intermediate language [11]. Note that the programmer provided (almost[4]) all information necessary to hoist the linear cost call to `norm` out of the parallel loop into the sequential part. Even after we propagate alias information to the parallel function (see Sect. 3), existing transformations can only move the call out of the parallel loop but not out of the parallel function. Our specialized communication optimization pass reorganizes the code placement and thereby the communication between the sequential and parallel code. The goal is to *minimize* the cost of *explicit communication*, thus variables passed to and from the parallel region, but also the *total number of computations* performed.

Our communication optimization pass will generate a weighted flow graph in which all variables are encoded that are executable in both the parallel and sequential part of the program. For our example in Fig. 5a the flow graph is shown in Fig. 5c[5]. Each variable is split into two nodes, an incoming one (upper part) and an outgoing one (lower part). The data dependences are represented as infinite capacity/cost (c_∞) edges between the outgoing node of the data producer and the incoming node of the data consumer. The two nodes per variable are connected from in to out with an edge that has the capacity equal to the minimum of the recomputation and communication (c_ω) cost. Since recomputation requires the operands to be present in the parallel region, its cost is defined as the sum of operand costs plus the operation cost, e.g., c_{ld} for memory loads. We also add edges from the source to represent this operation cost flow. If an expression is not movable but used by movable expression it will have infinite cost flowing in from the source. Globally available expressions, thus constants and global variables, have zero communication cost and are consequently omitted. Though, loads of global variables are included as they can be moved. Finally, all values required in the immovable part of the parallel region are connected to a sink node with infinite capacity edges.

The minimum cut of this graph defines an optimal set of values that should be communicated between the sequential and parallel code. For the example code in

[4] We need to ensure that `norm` is only executed under the condition `n > 0`.

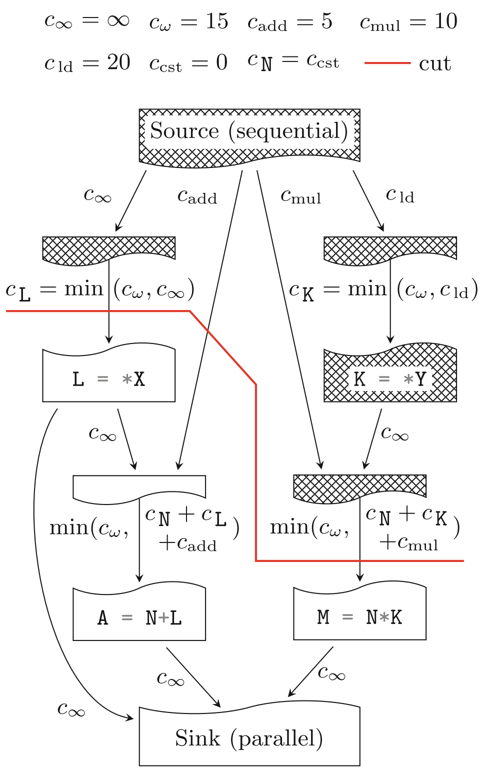
[5] The nodes for `X` are omitted for space reasons. They would look similar to the ones for `L`, though not only allow c_ω flow to the sink but also into the incoming node of `L`.

```
void f(int *X, int *restrict Y) {
  int N = 512;          //   movable
  int L = *X;           // immovable
  int A = N + L;        //   movable
  #pragma omp parallel for         \
      firstprivate(X, Y, N, L, A)
  for (int i = 0; i < N; i++) {
   int K = *Y;          //   movable
   int M = N * K;       //   movable
   X[i] = M+A*L*i;      // immovable
  }
}
```

(a) Function that exposes multiple code movement opportunities between the sequential and parallel part.

```
void g(int *X, int *restrict Y) {
  int L = *X;         // immovable
  int K = *Y;         // c_ld > c_ω
  int M = 512 * K;    // c_mul + c_K > c_ω
  #pragma omp parallel              \
          firstprivate(X, M, L)
  {
   int A = 512 + L;  // c_add < c_ω
   #pragma omp for                  \
        firstprivate(X, M, A, L)
   for (int i = 0; i < 512; i++)
    X[i] = M+A*L*i;  // immovable
  }
}
```

(b) Function shown in Figure 5a after communication optimization.


$c_\infty = \infty$ $c_\omega = 15$ $c_{\text{add}} = 5$ $c_{\text{mul}} = 10$
$c_{\text{ld}} = 20$ $c_{\text{cst}} = 0$ $c_{\text{N}} = c_{\text{cst}}$ cut
Source (sequential)
c_∞
c_{add}
c_{mul}
c_{ld}
$c_{\text{L}} = \min(c_\omega, c_\infty)$
$c_{\text{K}} = \min(c_\omega, c_{\text{ld}})$
L = *X
K = *Y
c_∞
c_∞
$\min(c_\omega, c_{\text{N}} + c_{\text{L}} + c_{\text{add}})$
$\min(c_\omega, c_{\text{N}} + c_{\text{K}} + c_{\text{mul}})$
A = N+L
M = N*K
c_∞
c_∞
c_∞
Sink (parallel)

(c) Communication flow graph[5] for the code shown in Figure 5a. Variable nodes that are partially reachable from the source after the minimal cut are hatched and placed in the sequential part of the result (see Figure 5b).

Fig. 5. Communication optimization example with the original code in part 5a, the constructed min-cut graph in part 5c and the optimized code in part 5b.

Fig. 5a, the communication graph, sample weights and the minimal cut are shown in Fig. 5c. After the cut was performed, all variables for which the incoming node is reachable from the source will be placed in the sequential part of the program. If an edge between the incoming and outgoing node of a variable was cut, it will be communicated through the runtime library. Otherwise, it is only used in one part of the program and also placed in that part. For the example shown in Fig. 4 the set of immobile variables would necessarily include `out`,`in` and `n` as they are required in the parallel region and cannot be recomputed. However, the result of the norm function will be hoisted out of the parallel function if the recomputation cost is greater than the communication cost, thus if $c_\omega > c_{\text{call}}$.

In summary, this construction will perform constant propagation, communicate arguments by-value instead of by-reference, minimize the number of communicated variables, recompute values per thread if necessary, and hoist variables not only out of parallel loops but parallel functions to ensure less executions.

8 Evaluation

To evaluate our prototype implementation we choose appropriate Rodinia 3.1 OpenMP benchmarks [4] and the LULESH v1.0 kernel [7]. Note that not all Rodinia benchmarks communicate through the runtime library but some only use shared global memory which we cannot yet optimize. The Rodinia benchmarks were modified to measure only the time spent in OpenMP regions but the original measurement units were kept. All benchmarks were executed 51 times and the plots show the distribution as well as the median of the observed values. We evaluated different combinations of our optimizations to show their individual effect but also their interplay. However, to simplify our plots we only show the optimizations that actually changes the benchmarks and omit those that did not. The versions are denoted by a combination of abbreviation as described in Table 1. Note that due to the prototype stage and the lack of a cost heuristics we did not evaluate our communication optimization.

Table 1. The abbreviations used in the plots for the evaluated optimizations as well as the list of plots that feature them

Version	Description	Plots
base	Plain "-O3", thus no parallel optimizations	Figs. 6, 7 and 8
ap	Attribute propagation (ref. Sect. 3)	Figs. 6, 7 and 8
vp	Variable privatization (ref. Sect. 4)	Figs. 6, 7 and 8
re	Parallel region expansion (ref. Sect. 5)	Fig. 7
be	Barrier elimination (ref. Sect. 6)	Fig. 8

When we look at the impact of the different optimizations we can clearly distinguish two groups. First, there is *ap* and *vp* which have a positive effect on every benchmark. If applied in isolation, attribute propagation (*ap*) is often slightly better but the most benefit is achieved if they are combined (*ap_vp* versions). This is mostly caused by additional alias information which allows privatization of a variable. The second group contains parallel region expansion (*re*) and barrier elimination (*be*). The requirements for these optimizations are only given in some of the benchmarks (see Fig. 7 and respectively Fig. 8). In addition, parallel region expansion is on its own not always beneficial. While it is triggered for cfd, srad, and pathfinder it will only improve the last one and slightly decrease the performance for the other two. The reason is that only for pathfinder the overhead of spawning thread teams is significant enough to improve performance, especially since barrier elimination was not able to remove the now explicit barriers in any of the expanded regions. These results motivate more work on a

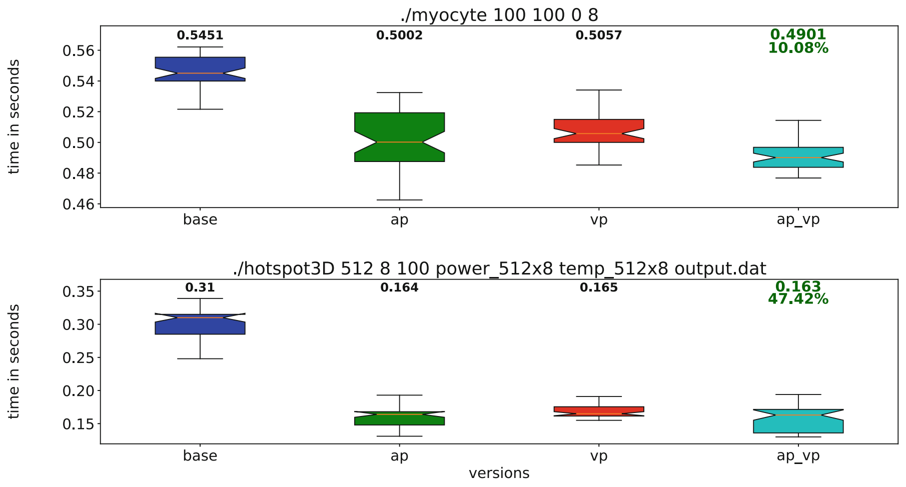


Fig. 6. Performance improvements due to attribute propagation (ref. Sect. 3) and variable privatization (ref. Sect. 4).

better cost heuristic and inter-pass communication. It is however worth to note that the LULESH v1.0 kernel [7] already contained expanded parallel regions without intermediate barriers. This manual transformation could now also be achieved automatically with the presented optimizations.

9 Related Work

To enable compiler optimizations of parallel programs, various techniques have been proposed. They often involve different representations of parallelism to enable or simplify transformations [6,8,15].

In addition, there is a vast amount of research on explicit optimizations for parallel programs [1–3,5,10]. In contrast to these efforts we introduce relatively simple transformations, both in terms of implementation and analysis complexity. These transformations are intended to perform optimizations only meaningful for parallel programs, but in doing so, also unblock existing compiler optimizations that are unaware of the semantics of the runtime library calls currently used as parallel program representation.

The Intel compiler toolchain introduces "OpenMP-like" annotations in the intermediate representation IL0 [12,13]. While effective, this approach require various parts of the compiler to be adapted in order to create, handle, and lower these new constructs. In addition, each new OpenMP construct, as well as any other parallel language that should be supported, will require a non-trivial amount of integration effort. Partially for these reasons, Intel proposed a more native embedding [14] of parallel constructions (especially OpenMP) into the intermediate representation of LLVM. Similarly, Tapir [11] is an alternative to integrate task parallelism natively in LLVM. In contrast to most other solutions,

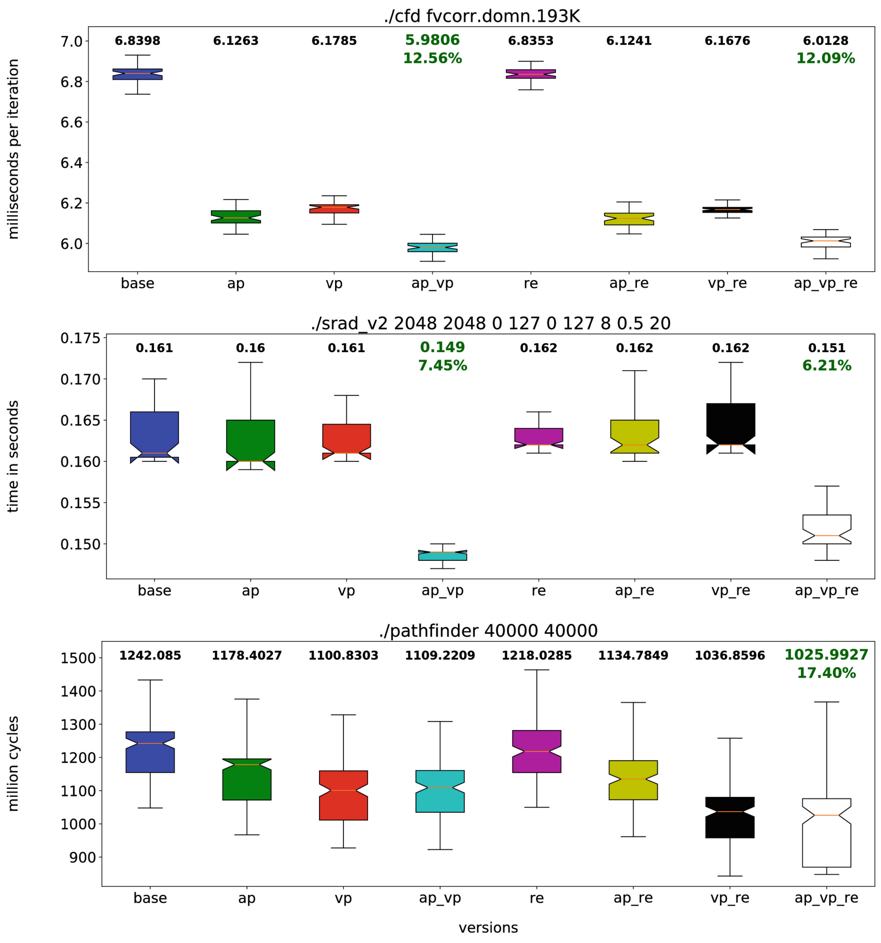


Fig. 7. Performance results for three of our parallel optimizations (*ap*, *vp*, *re*).

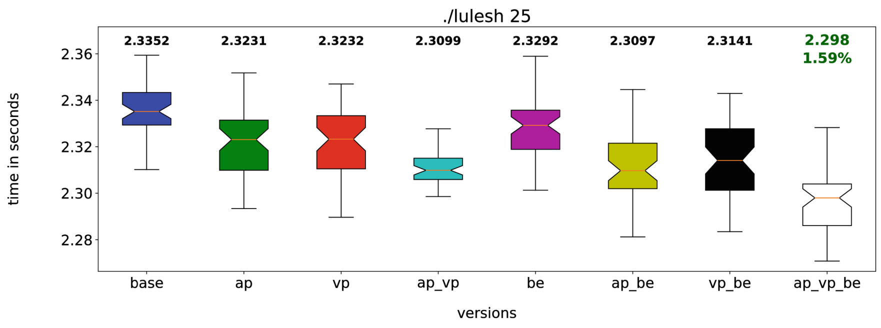


Fig. 8. Performance results for three of our parallel optimizations (*ap*, *vp*, *be*).

it only introduces three new instructions, thus requiring less adaption of the code base. However, it is not possible to express communicating/synchronizing tasks or parallel annotations distributed over multiple functions.

10 Conclusion

In this work we present several transformations for explicitly parallel programs that enable and emulate classical compiler optimizations which are not applicable in the current program representation. Our results show that these transformations can have significant impact on the runtime of parallel codes while our implementation does not require substantive implementation or maintainability efforts. While further studies and the development of more robust and cost aware optimizations are underway, we believe our initial results suffice as an argument for increased compiler driven optimizations of parallel programs.

It is worthwhile to note that we are currently proposing to include the presented optimizations into LLVM. To this end we generalized them to allow optimization not only of OpenMP programs lowered to runtime library calls but a more general set of parallel representations.

Acknowledgments. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

References

1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel analysis of X10 programs. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, 14–17 March 2007, pp. 183–193 (2007). https://doi.org/10.1145/1229428.1229471
2. Barik, R., Sarkar, V.: Interprocedural load elimination for dynamic optimization of parallel programs. In: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, PACT 2009, 12–16 September 2009, Raleigh, North Carolina, USA, pp. 41–52 (2009). https://doi.org/10.1109/PACT.2009.32
3. Barik, R., Zhao, J., Sarkar, V.: Interprocedural strength reduction of critical sections in explicitly-parallel programs. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, 7–11 September 2013, pp. 29–40 (2013). https://doi.org/10.1109/PACT.2013.6618801
4. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, 4–6 October 2009, Austin, TX, USA, pp. 44–54 (2009). https://doi.org/10.1109/IISWC.2009.5306797

5. Grunwald, D., Srinivasan, H.: Data flow equations for explicitly parallel programs. In: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, 19–22 May 1993, pp. 159–168 (1993). https://doi.org/10.1145/155332.155349
6. Jordan, H., Pellegrini, S., Thoman, P., Kofler, K., Fahringer, T.: INSPIRE: the insieme parallel intermediate representation. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, 7–11 September 2013, pp. 7–17 (2013). https://doi.org/10.1109/PACT.2013.6618799
7. Karlin, I., et al.: LULESH programming model and performance ports overview. Technical report LLNL-TR-608824, December 2012
8. Khaldi, D., Jouvelot, P., Irigoin, F., Ancourt, C., Chapman, B.M.: LLVM parallel intermediate representation: design and evaluation using OpenSHMEM communications. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, 15 November 2015, pp. 2:1–2:8 (2015). https://doi.org/10.1145/2833157.2833158
9. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004), 20–24 March 2004, San Jose, CA, USA, pp. 75–88 (2004). https://doi.org/10.1109/CGO.2004.1281665
10. Moll, S., Doerfert, J., Hack, S.: Input space splitting for OpenCL. In: Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, 12–18 March 2016, pp. 251–260 (2016). https://doi.org/10.1145/2892208.2892217
11. Schardl, T.B., Moses, W.S., Leiserson, C.E.: Tapir: embedding fork-join parallelism into LLVM's intermediate representation. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, 4–8 February 2017, pp. 249–265 (2017). http://dl.acm.org/citation.cfm?id=3018758
12. Tian, X., Girkar, M., Bik, A.J.C., Saito, H.: Practical compiler techniques on efficient multithreaded code generation for OpenMP programs. Comput. J. **48**(5), 588–601 (2005). https://doi.org/10.1093/comjnl/bxh109
13. Tian, X., Girkar, M., Shah, S., Armstrong, D., Su, E., Petersen, P.: Compiler and runtime support for running OpenMP programs on pentium-and itanium-architectures. In: Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2003), 22 April 2003, Nice, France, pp. 47–55 (2003). https://doi.org/10.1109/HIPS.2003.1196494
14. Tian, X., et al.: LLVM framework and IR extensions for parallelization, SIMD vectorization and offloading. In: Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, 14 November 2016, pp. 21–31 (2016). https://doi.org/10.1109/LLVM-HPC.2016.008
15. Zhao, J., Sarkar, V.: Intermediate language extensions for parallelism. In: Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH 2011, Proceedings of the Compilation of the Co-located Workshops, DSM 2011, TMC 2011, AGERE! 2011, AOOPES 2011, NEAT 2011, and VMIL 2011, 22–27 October 2011, Portland, OR, USA, pp. 329–340 (2011). https://doi.org/10.1145/2095050.2095103

Supporting Function Variants in OpenMP

S. John Pennycook[1](✉), Jason D. Sewall[1], and Alejandro Duran[2]

[1] Intel Corporation, Santa Clara, USA
john.pennycook@intel.com
[2] Intel Corporation Iberia, Madrid, Spain

Abstract. Although the OpenMP API is supported across a wide and diverse set of architectures, different models of programming – and in extreme cases, different programs altogether – may be required to achieve high levels of performance on different platforms. We reduce the complexity of maintaining multiple implementations through a proposed extension to the OpenMP API that enables developers to specify that different code paths should be executed under certain compile-time conditions, including properties of: active OpenMP constructs; the targeted device; and available OpenMP runtime extensions. Our proposal directly addresses the complexities of modern applications, allowing for OpenMP contextual information to be passed across function call boundaries, translation units and library interfaces. This can greatly simplify the task of developing and maintaining a code with specializations that address performance for distinct platforms and environments.

1 Introduction

As the OpenMP* programming model has evolved to keep pace with evolving architectures, it has introduced many new features (*e.g.* task-based parallelism, offloading to accelerators and explicit SIMD programming). These features are both sufficient to ensure that OpenMP is portable to a wide range of devices, and also expressive enough that developers are able to write codes that are capable of extracting a high level of performance from their targeted device.

However, writing a code that is able to run well on *all* of the devices that OpenMP supports – a code that exhibits high levels of "performance portability" [12,13] – remains a challenge; different devices and/or OpenMP implementations may prefer different ways of expressing parallelism, and some may prefer different algorithms altogether. Several frameworks have been developed on top of OpenMP in an attempt to simplify development when targeting multiple architectures [4,7]; our interpretation of this is that many find the current tools in OpenMP lacking in expressibility.

The proposed `concurrent` directive from OpenMP TR6 [11] addresses part of this problem, effectively providing a mechanism for developers to request that the decision of which OpenMP construct(s) should be used on a given loop nest be made by the implementation, based on analysis of the loop nest and the implementation's knowledge of the targeted device. Such a *descriptive* approach

© Springer Nature Switzerland AG 2018
B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 128–142, 2018.
https://doi.org/10.1007/978-3-319-98521-3_9

covers simple cases well, but we believe it is insufficient for the needs of expert programmers: the implementation's decisions cannot be overridden when the developer has additional information (or simply believes that they know better), and a general-purpose OpenMP implementation will likely not be able to identify and replace algorithms (although this may be possible for common idioms). In this paper, we focus on providing a *prescriptive* complement to the functionality of `concurrent`, enabling users to assert direct control over which code is executed and under which conditions.

It should be noted that our proposal is primarily focused on furnishing expressibility rather than providing new functionality to developers; our intent is to make existing functionality more accessible, and to present a simple mechanism with common syntax that can be employed across all base languages supported by OpenMP. There are already myriad options for maintaining different code paths for different devices and compilation contexts: preprocessors and `#ifdef` guards; template functions (in C++); and the aforementioned "performance portability" frameworks to name but a few. In our own attempts to use such approaches, we have found them wanting: standardized preprocessors are not available for all languages, and handling multiple conditions through nested `#ifdef` clauses can quickly lead to unreadable code; templates may be too complex for average users to reason about, and are not available in C or Fortran; the use of non-standard interfaces may lead to interoperability or composability challenges; and developing bespoke solutions to this problem for all codes (or even all domains) is not productive.

2 Related Work

Since OpenMP 4.0 [10], developers have been able to request that a compiler create alternative versions of a function specialized for execution in SIMD or on accelerator devices via the `declare simd` and `declare target` directives respectively. Developers are able to influence the code generation inside such functions using clauses to these directives (*e.g.* specifying `uniform` or `linear` will lead to different optimizations for `simd` functions), but are unable to exert any direct control using standard OpenMP functionality. Furthermore, since both directives only alter the way in which a single implementation of a function is compiled, the optimizations that can be employed are limited to those that compilers can identify and (safely) implement automatically after static analysis of the function, with no further input from the developer.

To address these issues, there have been several previous efforts to extend OpenMP to enable developers to provide drop-in replacements for the functions generated by `declare simd` and `declare target`. OmpSs supports an extension to `target` – the `implements` clause [3] – which specifies that one function is an alternative implementation of another, specialized for particular devices; this idea has been adopted, with the same syntax and semantics, in OpenMP TR6 [11]. The Intel® C/C++ compiler provides similar functionality for functions specialized for particular SIMD widths and instruction sets – so-called "vector variants" [8] – via function attributes; RIKEN and Arm explored the same concept via an OpenMP extension – `alias simd` [9] – with different syntax but similar semantics. None of these proposals address situations in which a developer wishes to specialize a function for SIMD *and* a particular device simultaneously, nor considers the utility of extending function implementation/-variant/alias support to other situations. We proposed yet another version of this functionality – `declare version` – for memory allocators [14] in previous work. In this paper, we attempt to unify all of the directives discussed above into a single directive, which permits the specialization of functions based on multiple criteria and which is designed to be extensible to future OpenMP constructs.

It should also be noted that similar functionality has already been adopted outside of OpenMP. Thrust [2] and the parallel extensions to the C++ Standard Template Library (STL) [6] allow an execution policy (encapsulating the programming model, device, etc) to be passed to a function call, enabling different implementations to be selected by the library; the Kokkos [4] framework provides a similar facility through the use of tags passed to functors. The PetaBricks [1] language takes a different approach, providing a mechanism to declare a function in terms of multiple candidate algorithms from which the compiler and an autotuning framework can construct an optimized application.

3 Specialization

Specialization is a valuable concept to deploy in software development, optimization, and maintenance. At its core, it is simply a programming construct – a function, a type declaration or even a snippet of code – paired with a mechanism for expressing when that construct should be used. The most common forms of specialization in real code are highly manual in nature: the programmer decides that a particular function (for example) needs to be used in a particular context and makes it happen by forking the codebase, or by employing a conspicuous branch in the code to perform the discrimination.

Consider a parallel histogram computation, where many inputs are divided among threads and reduced into a relatively small array. A developer has a common implementation in a portable language that runs on many of their platforms of interest, and which uses atomic additions found in the language; this implementation provides correct results in all cases. The developer may discover that the atomics primitives on some platforms are very slow, and that giving each thread its own copy of the accumulation array (*i.e. privatizing* the array) runs with much greater efficiency on those platforms. This could be described as specialization for performance's sake, but it is also easy to imagine a platform with no support for atomics at all, in which case the privatized implementation would be necessary for compatibility.

That specialization is useful and even necessary in real code bases is evident; the challenge – which our proposal aims to address – is to make the *mechanism* for deploying the appropriate specialization as easy-to-use and expressive as possible.

4 Enabling Specialization in OpenMP

We propose an extension that uses functions and function calls as the language-level granularity of specialization; developers are able to use a new directive, `declare variant`, to indicate that a given *function variant* is intended to be a specialization of another *base function* with a compatible type signature. The specialization mechanism is then used to decide which function calls to the base function are replaced with function calls to specialized variants (if any). Our proposal allows the user to annotate function variants with *selector* information that guides the specialization mechanism; these selectors generally interact with the *context* around function calls to select meaningful specializations.

Specialization of function calls enables composability (*e.g.* across translation units and library interfaces) via a mechanism that is simple to understand, familiar due to its similarity to other approaches (*e.g.* template specialization) and in keeping with good software engineering practices (*i.e.* modular design). A simple example of a function *variant family* is given in Fig. 1.

```
// Default behavior (i.e. the base function)
float my_rsqrt(float x)
{
  return 1.0f / sqrt(x);
}

// Variant that uses an approximation
float my_rsqrt_approx(float x)
{
  // e.g. result after several Newton-Raphson iterations
}

// Variant vectorized with AVX-512
__m512 my_rsqrt_avx512(__m512 x)
{
  return _mm512_rsqrt_ps(x);
}

// Variant which forwards to library implementation (where it exists)
float my_rqsrt_native(float x)
{
  return rsqrt(x);
}
```

Fig. 1. An example of a function variant family for computing reciprocal square roots.

4.1 `declare Variant` Syntax

The C/C++ syntax[1] of our proposed `declare variant` directive is as follows:

```
#pragma omp declare variant(base-function) [match(context-selector)]
<specialized-function-definition-or-declaration>
```

where `base-function` is the name of the function that the programmer wishes to specialize. The function the directive is applied to is a specialized variant that is

[1] Analogous syntax is proposed for Fortran but we omit it for brevity.

defined as a suitable replacement of `base-function`. The scope of this directive is the translation unit where it appears; therefore, we expect the directive to be applied to the specialized function declaration in headers and also to the specialized function definition itself.

If no `match` clause is provided, then all calls to the `base-function` will become calls to the specialized function. Figure 2 shows how the `declare variant` directive is used to provide an OpenMP specialization, `my_rsqrt_omp_approx`, of the function `my_rsqrt`. All calls to `my_rsqrt` will be replaced by calls to `my_rsqrt_omp_approx`. As the `variant` directive is only recognized by OpenMP-enabled compilers, this provides a mechanism for users to have different code used when compiling for OpenMP (which could also be achieved by means of `#ifdef _OPENMP` with preprocessors).

```
#pragma omp declare variant(my_rsqrt)
float my_rsqrt_omp_approx(float x) { ... }

void foo (float x)
{
   ... = my_sqrt(x); // will call my_rsqrt_omp_approx
}
```

Fig. 2. Example of the `declare variant` directive.

The `match` clause allows the developer to specify a *context-selector* that specifies the context in which calls to *base-function* should be substituted with calls to the specialized function. The syntax of a *context-selector* is as follows:

```
match(trait-class-name={trait[(trait-properties)][,...][,...]})
```

We propose a number of traits for specialization, organized into four trait classes that can specified in the `match` clause: OpenMP construct traits, for specialization based on OpenMP constructs; device traits, for specialization based on the target device; implementation traits, for specialization based on characteristics of the underlying OpenMP implementation; and user-specified traits. Section 4.2 details the traits of each class and their properties (if any).

Multiple specializations of the same base function can be specified using a `declare variant` directive on each specialization. Figure 3 shows an example where two variants have been defined: `my_rsqrt_omp_approx`, to be called when the base function appears in the context of a `simd` directive; and `rsqrt` (possibly provided by a library), to be called when the base function appears in the context of a `target` directive.

In the previous example, it was unambiguous which specialization should be used in each call to `my_rsqrt`, but that is not always the case. For example, if the call were to happen inside a `target parallel for simd` directive, it would be unclear which specialization should be called. We handle such cases by assigning different priorities to each variant, and selecting the variant with the highest priority at a callsite; this algorithm is described in detail in Sect. 4.3.

```
#pragma omp declare variant(my_rsqrt) match(construct={simd})
float my_rsqrt_omp_approx(float x) { ... }

#pragma omp declare variant(my_rsqrt) match(construct={target})
float rsqrt(float x); // library provided

void foo ( float x )
{
  #pragma omp simd
  for ( ... ) { ... = my_rsqrt(x); } // will call my_rqsrt_omp_approx

  #pragma omp target
  {
     ... = my_rsqrt(x); // will call rsqrt
  }
}
```

Fig. 3. Example of `declare variant` directives with match clauses.

4.2 Context Selection Traits

We have identified several concepts that give rise to a need for specialization, and for the purposes of their use and description, we have organized them into *classes* of *traits*. This taxonomy aids the user in clearly expressing their intents for when a particular variant takes precedent over another for a given context. This proposal identifies four distinct classes of traits that help distinguish the conditions for specialization.

OpenMP Construct Traits. The traits in the `construct` class are related to existing OpenMP constructs that might impact a developer's choices for specialization. Table 1 describes the traits in the `construct` class. Each trait specified for this class restricts the associated variant to calls to the base function that appear in the context of the directive of the same name.

Table 1. Traits in the `construct` trait class

Trait name	Example uses
`target`	Code paths that track host/target allocations and perform transfers
`parallel`	Code paths that choose between serial & parallel algorithms; code paths that discriminate based on memory model (*e.g.* atomics, critical, etc)
`teams`	Code paths that choose algorithms or implementations based on synchronization behavior; code paths that perform differently when synchronization is expected to be fast within a team
`simd`	Code paths that deploy *horizontal* vector operations (*e.g.* conflict detection); code paths that override default auto-vectorization behavior

For the `simd` trait, we also propose to allow different *trait properties* that represent clauses available in the `declare simd` directive. These properties further restrict the context in which a variant can be selected. In Fig. 4 two variants are defined to be used in the context of a `simd` construct. The first variant can be used in any `simd` context but the second one can only be used when the `simd` context also determines that the argument of the function is `linear`. Consequently, the first invocation of *foo* in Fig. 4 will be substituted with the first variant as the compiler cannot determine that the argument is `linear`, whereas the second call to *foo* will be substituted with the second variant as the compiler can determine that i is `linear`.

```
float foo(float *x);

#pragma omp declare variant(foo) match(construct={simd})
__mm512 foo_simd_gather(__m512 *x); // needs to use gather instructions

#pragma omp declare variant(foo) match(construct={simd(linear(x))}
__mm512 foo_simd_linear(__mm512 *x); // can avoid gather instructions

#pragma omp parallel for simd linear(i)
for ( i = 0; i < N; i++ ) {
    ... = foo(x[rand()]); // will call foo_simd_gather
    ... = foo(x[i]);      // will call foo_simd_linear
}
```

Fig. 4. Example of `simd` trait properties.

Device Traits. The traits in the `device` class are based on properties of the hardware that the code is being compiled for. Therefore, they restrict the contexts where a variant can be selected to only those where the specified device traits are true. Table 2 describes the traits in the `device` class. Multiple `isa` traits can be specified for a single variant: all of them must be supported by the target device for a variant to be selected. We propose that implementations not be required to be able to compile the function body for variants with `device` traits that are not supported (*e.g.* an unknown ISA), thereby simplifying the use of device-specific intrinsics by programmers. However, we still require that extensions used in the context of a variant function should at least allow other implementations to properly parse (and ignore) the function (*i.e.* by allowing to find the closing bracket of the variant function).

Table 2. Traits in the `device` trait class.

Trait name	Example uses
`uarch`(*uarch-name*)	Code paths that use different optimizations for different microarchitectures; code paths that care about particular implementations of instruction sets or compute capabilities
`isa`(*isa-name*)	Code paths that use specific instruction sets

OpenMP Implementation Traits. Traits in this class are concerned with properties of the particular OpenMP implementation that will be used to run the generated code. Only implementations that support the traits specified in a selector can select that variant as a replacement of the base function. Table 3 describes the traits in the `implementation` class.

Table 3. Traits in the `implementation` trait class.

Trait name	Example uses
`unified_shared_memory` `unified_address`	Code paths that require runtime support for unified shared memory/address spaces across devices. The behavior of these traits is documented in OpenMP TR6 [11]
`vendor`(*vendor-name* [,*extensions*])	Code paths that require vendor-specific and/or prototype concepts

User Traits. In addition to the above trait classes associated with OpenMP contexts, hardware, and runtime capabilities, there is a `user` class that accepts logical expressions as traits. These logical expressions, expressed in the base language, must be able to be evaluated at compile-time, and they can be used to add arbitrary user-specified conditions that can inform variant selection. Figure 5 shows an example of a `user` trait in use; the logical test for the value of the static constant variable `layout` may enable or disable each of the variants in the example.

```
void foo(float *x);

typedef enum {AoS=0, SoA} layout_t;

#pragma omp declare variant(foo) match(user={condition(layout==AoS)})
void foo_AoS(float *x);

#pragma omp declare variant(foo) match(user={condition(layout==SoA)})
void foo_SoA(float *x);

...
static const layout_t layout = AoS;
...
foo(z); // will call foo_AoS
```

Fig. 5. Example of a `user` trait; the value of the compile-time variable `layout` determines the logical value of the `user` selector in each variant.

These classes and their traits are those that we have chosen as the most important and tractable in this initial proposal; they are all static notions that allow contextual information to be tracked or inferred during compile-time. Many other concepts – for example, the values of certain variables or certain arguments at runtime, system conditions, and other language constructs – would be interesting to explore in future work.

4.3 Caller Context and Variant Selection

Our proposed selection mechanism is not explicit; instead, the choice of which variant to call is performed through the interaction of the various selectors and the *context* around the function call. This is necessary to correctly handle cases in which the arguments to a function may be generated or modified by the compiler (*e.g.* during auto-vectorization); to leverage contextual information not exposed to the developer (*e.g.* during auto-parallelization); and to enable selection to be employed transparently for functions maintained by other developers (*e.g.* library functions). Such an implicit mechanism does not remove any control from the user; they remain free to call specific variants explicitly by name.

In the remainder of this section, we discuss how such contextual information can be established and tracked by way of: lexical scope; compiler configuration for a given translation unit; and a function variant's selector information.

Lexical Scope. The OpenMP construct trait class described in Sect. 4.2 contains traits related to a number of OpenMP directives that establish lexical blocks with specific behavior. Conceptually, as each such directive is encountered in lexical order, the corresponding trait is added to the context. As the block for each directive closes, the corresponding trait is eliminated from the context. See Fig. 6 for an example of how contexts vary with the presence of OpenMP directives.

```
void main()
{
  // construct context = {} (i.e. empty)
  #pragma omp target
  {
    // construct context = {target}
    #pragma omp parallel
    {
      // construct context = {target,parallel}
      #pragma omp simd
      {
        // construct context = {target,parallel,simd}
      }
      // construct context = {target,parallel}
    }
    // construct context = {target}
  }
  // construct context = {} (i.e. empty)
}
```

Fig. 6. Example of how the construct context is changed upon entering and exiting lexically scoped OpenMP regions.

Traits in the OpenMP construct context may also be set implicitly, as the result of certain optimizations: regions that are automatically parallelized and/or vectorized without use of the corresponding OpenMP directives may add `parallel` and/or `simd` to the context; similarly, implementations may alter the context to reflect decisions taken as a result of certain descriptive constructs (*e.g.* `concurrent`). Such transformations still imply a lexical scope, albeit one that is usually not exposed to the developer.

Translation Units. Many traits are not established by explicit directives that annotate lexical structures; the traits found in the device and OpenMP runtime trait classes (see Sect. 4.2) are generally established for a translation unit implicitly, by the compiler itself. The exact behavior of these traits will vary from compiler to compiler, but tracking them as part of the context is necessary for matching variants effectively.

For example, a user may specify options to a compiler instructing it to generate code for a specific instruction set or to optimize for the characteristics of a particular microarchitecture; in the presence of such flags, the `isa` and `uarch` traits should be defined appropriately in the context. Specific examples of how such compiler options can impact the device context are given in Fig. 7.

```
icpc -xMIC-AVX512:
device context = { uarch(knl), isa(avx512f, avx512er, avx512cd, ...) }

icpc -xCORE-AVX512:
device context = { uarch(skx), isa(avx512f, avx512cd, ...) }

gcc -msse2 -msse3:
device context = { isa(sse2, sse3) }

clang++ --cuda-gpu-arch=sm_70:
device context = { isa(sm_70) }
```

Fig. 7. Example of how the context is changed by compiler flags. Flags for enabling and configuring OpenMP are omitted.

Functions. Generally speaking, contexts are established within a function body without accounting for any surrounding contexts that hypothetical callers may establish, and the user should not assume that contextual information is passed across function boundaries. Our proposal makes two exceptions: a function's initial context may be modified by its variant selector; and compilers are free to broaden contexts when inlining. An example with both behaviors is shown in Fig. 8.

Variant Selectors. The context of a variant is defined to contain *at least* the traits specified in the variant's selector; additional traits may be present (defined for the translation unit) but only if they are compatible with the selector. This behavior allows for traits derived from lexical scope to be passed explicitly across translation unit boundaries.

```
void bar();

#pragma omp declare variant(bar) match(construct={teams})
void baz()
{
  // without inlining: construct context = {teams}
  //    with inlining: construct context = {parallel, teams}
  ...
}

void foo()
{
  // without inlining: construct context = {}
  //    with inlining: construct context = {parallel}
  #pragma omp teams
  {
    // without inlining: construct context = {teams}
    //    with inlining: construct context = {parallel, teams}
    baz();
  }
}

void main()
{
  #pragma omp parallel
  {
    foo();
  }
}
```

Fig. 8. Example of context propagation via variant selectors and/or inlining.

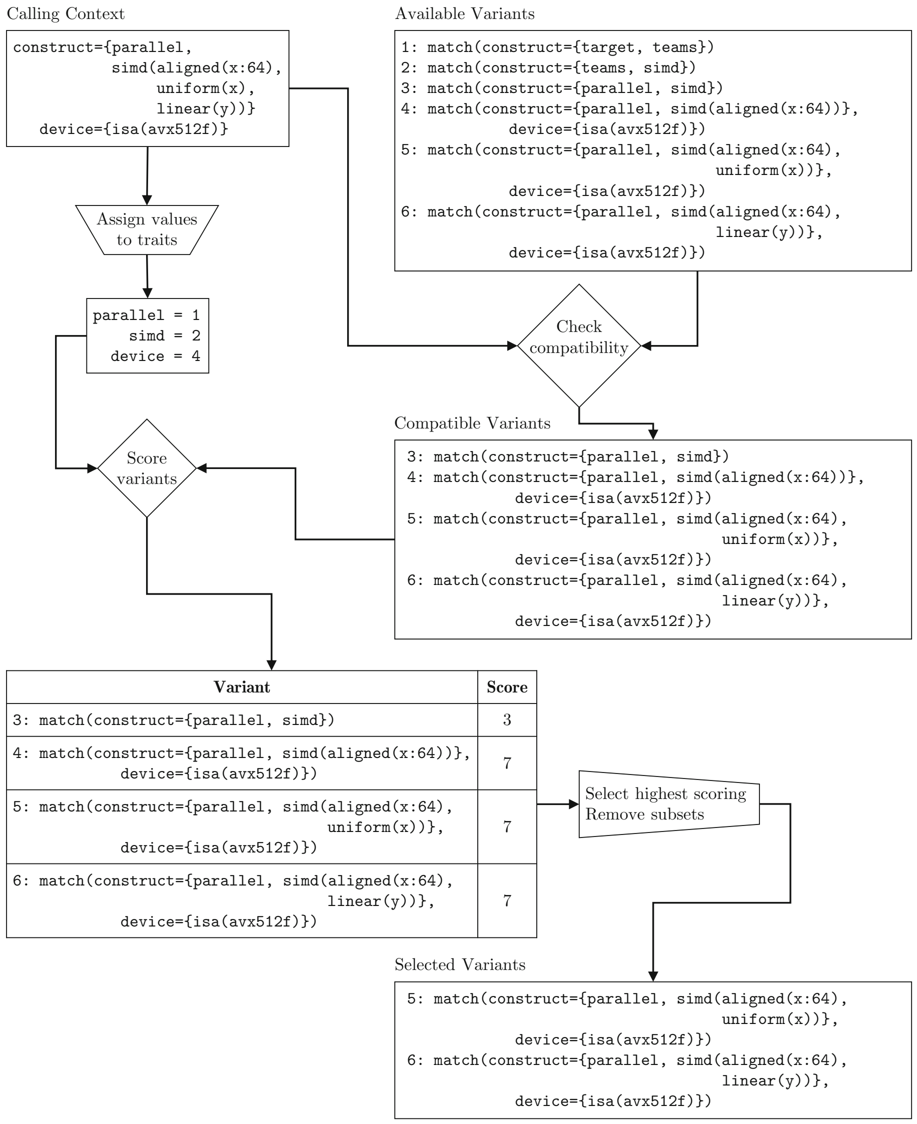
Inlining Behavior. It is common for a compiler to inline function bodies to satisfy user requests and to enable many optimizations. In such cases, the compiler may be able to supply additional context to the inlined function body based on where it has been inlined. While we would like to have consistent behavior of how OpenMP constructs should behave with respect to inlining, the OpenMP specification is not clear on this point and implementations vary in their interpretation. As such, developers should not *depend* on particular inlining behavior – since it is compiler-specific – but it does not introduce problems for our selection mechanism.

Selecting a Variant Based on Context. Given a calling context C and a variant family V, the variant selection algorithm proceeds as follows:

1. Eliminate all variants from V with selectors that are incompatible with C.
2. Compute a *specificity* score associated with each remaining selector.
3. If the most specific (highest scoring) selector is unique, return its variant.

A selector's specificity score is computed by assigning a value of 2^i to each context trait, where i reflects the trait's position in the context, and summing these values. Traits are ordered according to their class – `construct`, `device`, `implementation`, `user` – and level of nesting in lexical scope (if appropriate). If the most specific score is not unique, but the selectors can be ordered by a strict subset/superset relationship of their properties, the selector with the largest superset should be chosen; otherwise, the choice between the most specific selectors is unspecified.

Figure 9 shows this algorithm applied to an example calling context and a family of six variants. First, the variants are compared with the calling context to assess their compatibility: the first two variants are eliminated in this step, as neither `target` nor `teams` is present in the calling context. Second, the traits



Calling Context

```
construct={parallel,
           simd(aligned(x:64),
                uniform(x),
                linear(y))}
   device={isa(avx512f)}
```

Available Variants

```
1: match(construct={target, teams})
2: match(construct={teams, simd})
3: match(construct={parallel, simd})
4: match(construct={parallel, simd(aligned(x:64))},
            device={isa(avx512f)})
5: match(construct={parallel, simd(aligned(x:64),
                                   uniform(x))},
            device={isa(avx512f)})
6: match(construct={parallel, simd(aligned(x:64),
                                   linear(y))},
            device={isa(avx512f)})
```

```
parallel = 1
    simd = 2
  device = 4
```

Compatible Variants

```
3: match(construct={parallel, simd})
4: match(construct={parallel, simd(aligned(x:64))},
            device={isa(avx512f)})
5: match(construct={parallel, simd(aligned(x:64),
                                   uniform(x))},
            device={isa(avx512f)})
6: match(construct={parallel, simd(aligned(x:64),
                                   linear(y))},
            device={isa(avx512f)})
```

Variant	**Score**
`3: match(construct={parallel, simd})`	3
`4: match(construct={parallel, simd(aligned(x:64))}, device={isa(avx512f)})`	7
`5: match(construct={parallel, simd(aligned(x:64), uniform(x))}, device={isa(avx512f)})`	7
`6: match(construct={parallel, simd(aligned(x:64), linear(y))}, device={isa(avx512f)})`	7

Selected Variants

```
5: match(construct={parallel, simd(aligned(x:64),
                                   uniform(x))},
            device={isa(avx512f)})
6: match(construct={parallel, simd(aligned(x:64),
                                   linear(y))},
            device={isa(avx512f)})
```

Fig. 9. Example of the variant selection algorithm.

are assigned a value according to their position in the calling context, and these values are used to assign specificity scores to the variants: the variants receive scores of 3, 7, 7 and 7 based on the values assigned to the `parallel`, `simd` and `device` traits. The last step selects the variants with the highest score and, from those, removes variants that are a subset of other variants: the variant with just `simd(aligned(x:64))` is eliminated, since it is included in the other two variants. The algorithm selects two variants: the `simd` properties for the selected variants contain `aligned(x:64)` as a common subset, but since `uniform(x)` and `linear(y)` cannot be ordered an implementation is free to choose either.

4.4 Relation to Existing Directives

The behavior of `declare variant` as defined in our proposal is orthogonal to the behavior of the `declare simd` and `declare target` directives: it provides a mechanism for associating user-provided variants to base functions, but does not provide a mechanism for requesting compiler-generated variants of base functions. At the time of writing, it is unclear whether or not consolidating these functionalities into a single directive is desirable. By design, extending `declare variant` to support more contextual information is easier than extending the existing directives, but deprecating existing functionality may break existing user code. Should it be decided that deprecating `declare simd` and `declare target` *is* desirable, then modifying our proposed syntax to support this could be as straightforward as making the base-function part of an optional clause (as shown in Fig. 10), or introducing a separate `create variant` directive (as shown in Fig. 11).

```
// Request compiler-generated variant of foo, specialized for simd context
// Base-function omitted; equivalent to "declare simd"
#pragma omp declare variant match(construct={simd})
void foo();

// Associate user-provided variant of foo, specialized for simd context
// Uses "implements" clause; equivalent to current syntax
#pragma omp declare variant match(construct={simd}) implements(foo)
void bar();
```

Fig. 10. Example of using a modified `declare variant` directive to replace `declare simd` and `declare target`.

```
// Request compiler-generated variant of foo, specialized for simd context
// Equivalent to "declare simd", but user provides name for generated function
#pragma omp create variant(foo_simd) match(construct={simd})
void foo();
```

Fig. 11. Example of using a new `create variant` directive to replace `declare simd` and `declare target`.

5 Summary

Application developers hoping to achieve high levels of performance on multiple platforms require a mechanism for selecting and executing different code paths based on properties of the current execution context. This paper proposes a set of extensions to the OpenMP API that provide such a mechanism, introducing the ability to perform function dispatch based on contextual information known to OpenMP at compile-time. The specific contributions of this work are as follows:

1. We review the complex interaction between modern OpenMP constructs and representative OpenMP devices, thus motivating the introduction of powerful and expressive developer tools for specializing code for different execution environments.
2. We propose a new directive, `declare variant`, for declaring variants of functions that should be preferentially selected under certain conditions. Our proposal unifies several previous proposals, and is designed to be easily extended to cover future additional functionality.

We have designed `declare variant` to ensure that the contextual information it supports can be extended as the OpenMP API evolves, and there are many exciting future directions to explore. Incorporating an ability for dynamic (run-time) dispatch is the most obvious: many performance-impacting variables in OpenMP can be chosen dynamically (*e.g.* number of threads, scheduling policies); users may wish to select different devices or algorithms based on properties of program input; and just-in-time (JIT) compilation for problem size has been demonstrated to significantly improve performance in some cases [5]. When considering this extension, it will be important to consider the cost of run-time selection and dispatch.

Acknowledgements. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
* Other names and brands may be claimed as the property of others.

References

1. Ansel, J., et al.: PetaBricks: a language and compiler for algorithmic choice. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 38–49. ACM, New York (2009)
2. Bell, N., Hoberock, J.: Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems Jade Edition, pp. 359–371. Elsevier (2011)
3. Duran, A.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
4. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing

5. Heinecke, A., Henry, G., Hutchinson, M., Pabst, H.: LIBXSMM: accelerating small matrix multiplications by runtime code generation. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 84:1–84:11. IEEE Press, Piscataway (2016)
6. Hoberock, J.: Technical specification for C++ extensions for parallelism. Technical report ISO/IEC TS 19570:2015, ISO/IEC JTC 1/SC 22 (2015)
7. Hornung, R.D., Keasler, J.A.: The RAJA portability layer: overview and status. Technical report LLNL-TR-661403, Lawrence Livermore National Laboratory (2014)
8. Intel Corporation: `vector_variant`. https://software.intel.com/en-us/node/523350
9. Lee, J., Petrogalli, F., Hunter, G., Sato, M.: Extending OpenMP SIMD support for target specific code and application to ARM SVE. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 62–74. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_5
10. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 4.0 (2013)
11. OpenMP Architecture Review Board: OpenMP Technical Report 6: Version 5.0 Preview 2 (2017)
12. Pennycook, S., Sewall, J., Lee, V.: A metric for performance portability. In: Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (2016)
13. Pennycook, S., Sewall, J., Lee, V.: Implications of a metric for performance portability. Future Gen. Comput. Syst. (2017). https://doi.org/10.1016/j.future.2017.08.007
14. Sewall, J.D., Pennycook, S.J., Duran, A., Tian, X., Narayanaswamy, R.: A modern memory management system for OpenMP. In: Proceedings of the Third International Workshop on Accelerator Programming Using Directives, WACCPD 2016, pp. 25–35. IEEE Press, Piscataway (2016)

Towards an OpenMP Specification for Critical Real-Time Systems

Maria A. Serrano[1,2](✉), Sara Royuela[1,2], and Eduardo Quiñones[1]

[1] Barcelona Supercomputing Center (BSC), Barcelona, Spain
{maria.serranogracia,sara.royuela,eduardo.quinones}@bsc.es
[2] Universitat Politecnica de Catalunya (UPC), Barcelona, Spain

Abstract. OpenMP is increasingly being considered as a convenient parallel programming model to cope with the performance requirements of critical real-time systems. Recent works demonstrate that OpenMP enables to derive guarantees on the functional and timing behavior of the system, a fundamental requirement of such systems. These works, however, focus only on the exploitation of fine grain parallelism and do not take into account the peculiarities of critical real-time systems, commonly composed of a set of concurrent functionalities. OpenMP allows exploiting the parallelism exposed within real-time tasks and among them. This paper analyzes the challenges of combining the concurrency model of real-time tasks with the parallel model of OpenMP. We demonstrate that OpenMP is suitable to develop advanced critical real-time systems by virtue of few changes on the specification, which allow the scheduling behavior desired (regarding execution priorities, preemption, migration and allocation strategies) in such systems.

1 Introduction

There is an increasing demand to introduce parallel execution in critical real-time systems to cope with the performance demands of the most advanced functionalities, e.g., autonomous driving and unmanned aerial vehicles. In this regard, OpenMP is a firm candidate [22,40] due to its capability to efficiently exploit highly parallel and heterogeneous embedded architectures, and its programmability and portability benefits. OpenMP is already supported in several embedded platforms for instance, the Texas Instruments Keystone II [37] or the Kalray MPPA [18]. Moreover, OpenMP is being evaluated to be supported in future versions of the Ada language [26], used to develop safety critical systems.

Current critical real-time systems are composed of a set of independent and recurrent pieces of work, known as *real-time tasks*, implementing the functionalities of the system. This model enables to exploit the inherent *concurrency* of the system when the number of available cores is low, as it is the case of the Infineon Aurix, a 32-bit micro-controller used in automotive that features six cores [5]. With the newest highly parallel embedded architectures targeting the critical real-time market, the number of available cores has increased significantly, enabling to

B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 143–159, 2018.
https://doi.org/10.1007/978-3-319-98521-3_10

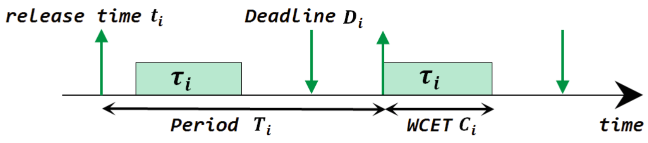


Fig. 1. Real-time task representation.

exploit fine-grain parallelism within each real-time task as well. This is the case for instance, of the Kalray MPPA, featuring a fabric of 256 cores [18].

However, critical real-time systems must provide strong *safety* evidences on the *functional* and *timing* behavior of the system. In other words, the system must guarantee that it operates correctly in response to its inputs, and that system operations are performed within a predefined time budget. A recent work evaluated the suitability of OpenMP from a functional perspective [29]. This paper complements that work and evaluates OpenMP from a timing behavior perspective. In this context, recent studies have shown the similarities between the structure and syntax of the OpenMP tasking model and the *Direct Acyclic Graph* (DAG) scheduling model [40], which enables to verify the timing constraints of parallel real-time tasks [13]. These similarities allow the analysis of the timing behavior of a single real-time task parallelized with OpenMP [35,38].

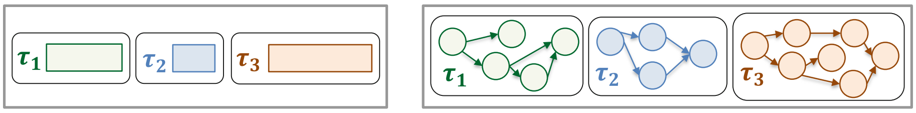
This paper extends previous works, and analyses the use of OpenMP (as it is in version 4.5 [3]) to implement critical real-time systems. We focus on the design implications and the scheduling decisions to efficiently exploit fine grain parallelism within real-time tasks and concurrency among them, while guaranteeing the timing behavior according to current real-time practices.

2 Critical Real-Time Systems

2.1 The Three-Parameter Sporadic Tasks Model

Critical real-time systems are represented as a set of recurrent and independent real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, \dots \tau_n\}$. Each execution of a real-time task is known as a *job*; the time at which a job is triggered is known as *release time* and it is denoted by t_i. A recurrent task can be *periodic*, if there is an exact time between two consecutive jobs, or *sporadic*, if there is a minimum time between jobs. In both cases, they can be triggered either by an internal clock or by the occurrence of an external event, e.g., a sensor.

Traditionally, the *three-parameter sporadic tasks model* [25] is used to characterize critical real-time systems composed of sequential tasks that run concurrently on a platform. In this model, each task τ_i is represented with the tuple $\langle C_i, T_i, D_i \rangle$, where C_i is the Worst-Case Execution Time (WCET), i.e., an estimation of the longest possible execution time of τ_i; T_i is the period, or the minimum time between two consecutive jobs of τ_i; and D_i is the deadline at which τ_i must finish (see Fig. 1). Critical real-time systems must guarantee that, for each task, its deadline is met, i.e., $\forall \tau_i \in \mathcal{T}$, $t_i + C_i \leq D_i$.



(a) 3-parameter sporadic tasks (b) Sporadic DAG tasks

Fig. 2. Real-time system models.

2.2 The Sporadic DAG Tasks Model

In the recent years, the complexity of real-time tasks have significantly increased to incorporate advanced functionalities, e.g., image recognition. With the objective of providing the level of performance needed, the code within each real-time task can be further parallelized. In this context, the use of the *sporadic DAG task model* [13] enables to characterize *parallel real-time tasks*[1] with the tuple $\tau_i = \langle G_i, T_i, D_i \rangle$. $G_i = (V_i, E_i)$ is a DAG representing the parallelism exposed within a real-time task. $V_i = \{v_{i,1}, \ldots, v_{i,n_i}\}$ denotes the set of nodes that can potentially be executed in parallel, where n_i is the number of nodes within τ_i. $E_i \subseteq V_i \times V_i$ denotes the set of edges between nodes, representing the precedence constraints existing between them: if $(v_{i,1}, v_{i,2}) \in E_i$, then $v_{i,1}$ must complete before $v_{i,2}$ begins its execution. In this model, each node $v_{i,j} \in V_i$ is characterized by its WCET, denoted by $C_{i,j}$. Finally, as in sequential real-time tasks, T_i and D_i represent the period and deadline of the parallel real-time task τ_i. This model is considered in the integrated modular avionics (IMA) [2] and the AUTOSAR [20] frameworks, used in avionics and automotive systems, respectively.

Figure 2 shows a taskset composed of three real-time tasks: in Fig. 2a, tasks are modelled with the three parameter sporadic tasks model, i.e., real-time tasks are sequential and run concurrently; in Fig. 2b, tasks are modelled with the sporadic DAG tasks model, i.e., real-time tasks have been parallelized, enabling to exploit both, concurrency and fine-grain parallelism.

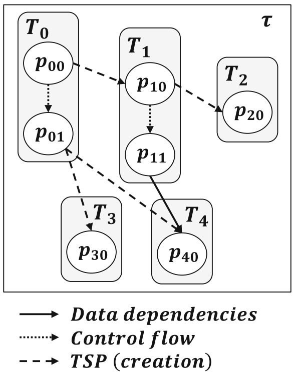
2.3 Parallelizing a Single Real-Time Task with OpenMP

Several works demonstrate that the OpenMP tasking model resembles the sporadic DAG task scheduling model when considering a single real-time task [24,35,38,40,41]. Hence, the OpenMP tasking model can be used to parallelize a real-time task, modelled as a DAG, upon which timing guarantees can be provided. Given an OpenMP-DAG $G = (V, E)$, nodes in V correspond to the portions of code that execute uninterruptedly between two *Task Scheduling Points (TSPs)*, referred as *parts of a task region* in the OpenMP specification, and considered as *task parts* henceforward. Edges in E correspond to explicit synchronizations (for instance, defined by the `depend` clause), TSPs (for instance, defined by the `task` construct) and control flow precedence constraints (defined by the sequential execution order of task parts from the same OpenMP task).

[1] *Parallel real-time tasks* denote real-time tasks which exploit parallelism within them. These tasks are also concurrent among them.

```
 1 void parallel_RT_task() {                    // τ
 2 #pragma omp parallel
 3 #pragma omp single nowait                    // T0
 4 {
 5   part00
 6   #pragma omp task depend(out:x)             // T1
 7   {
 8     part10
 9     #pragma omp task                         // T2
10     { part20 }
11     part11
12   }
13   part01
14   #pragma omp task                           // T3
15   { part30 }
16   #pragma omp task depend(in:x)              // T4
17   { part40 }
18 }}
```

(a) Real-time task parallelized with OpenMP tasks.



(b) OpenMP-DAG.

Fig. 3. Example of an OpenMP real-time task.

Figure 3a shows an example of a real-time task τ parallelized with the OpenMP tasking model, and Fig. 3b shows the corresponding OpenMP-DAG. Nodes of the OpenMP-DAG represent the seven tasks parts generated within the four explicit tasks and the implicit task executing the single region. For instance, task T_1 (line 6 of Fig. 3a), is composed of task parts $part_{10}$ and $part_{11}$ (nodes p_{10} and p_{11} in Fig. 3b). Edges represent (1) the data dependence between $T1$ and T_4; (2) the TSP after the creation of tasks T_1 to T_4, e.g., at the end of task part p_{10} task T_2 is created; and (3) control flow dependences, e.g., task parts p_{10} and p_{11} from T_1 execute sequentially. All threads are synchronized in the implicit barrier at the end of the `parallel` construct (not shown in Fig. 3b).

3 Developing Critical Real-Time Systems with OpenMP

This section analyses the use of the OpenMP tasking model to develop a critical real-time system, from two different perspectives: (1) how to efficiently exploit parallelism within real-time tasks and among them, and (2) how to express the recurrence of real-time tasks.

3.1 Parallelizing Several Concurrent Real-Time Tasks

In critical real-time systems, the scheduler plays a key role as it must guarantee that all real-time tasks execute before its deadline. To do so, real-time schedulers implement the following features (Sect. 4 provides a detailed analysis): (1) *tasks priorities*, which determine the urgency of each real-time task to execute; (2) *preemption strategies*, which determine when a real-time task can be temporarily interrupted if a more urgent task is ready to execute; and (3) *allocation strategies*, which determine the computing resources (cores) in which tasks can execute.

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task priority(p1)  // τ1 : OpenMP−DAG1
    {  RT_task_1()  }
    #pragma omp task priority(p2)  // τ2 : OpenMP−DAG2
    {  RT_task_2()  }
    ...
    #pragma omp task priority(pn)  // τn : OpenMP−DAGn
    {  RT_task_n()  }
}
```

Fig. 4. Critical real-time system implemented with OpenMP parallel tasks.

As introduced in the previous section, current works consider OpenMP only to exploit parallelism within a single real-time task. As a result, each real-time task defines its own OpenMP parallel environment. This becomes a black box for the scheduler, which can not control the resources used by real-time tasks.

In order for the scheduler to have full control over the execution of the real-time tasks (and their parallel execution), the complete taskset must be included within a single OpenMP application. To do so, one option is to exploit nested parallel regions, i.e., to enclose the real-time tasks, each defining its own parallel region (see Fig. 3a), within an outer parallel region. In this case, the OpenMP framework manages two scheduling levels: one in charge of scheduling the real-time tasks (outer parallel region), and another one in charge of scheduling the parallel execution within each real-time task (inner parallel regions). Interestingly, this approach enables the first level scheduler to use the `priority` clause associated to the `task` construct to determine the priority of each real-time task (see Sect. 4). However, this solution is not valid as the first-level scheduler cannot control the parallel execution of each real-time task. In other words, the team of threads of each real-time task is (again) a black box for the first-level scheduler. Hence, preemption and allocation strategies cannot be implemented.

Clearly, the control of the OpenMP threads executing each of the real-time tasks is key to support a fine-grain control over the whole parallel execution. To do so, we propose to define *a common team of OpenMP threads to execute all the real-time tasks*. Figure 4 shows the implementation of a real-time system in which each real-time task $\tau_i \in \mathcal{T}$ is encapsulated within an OpenMP task, and implemented in a function *RT_task_X()*. The code in Fig. 3a could represent an example of one of these functions. However, the `parallel` and `single` constructs (lines 2 and 3, respectively) must be removed, and a `taskwait` synchronization construct must be included at the end of the function (line 18). These changes are not shown due to lack of space. In this design, a single real-time scheduler will be in charge of scheduling both, the OpenMP tasks implementing the real-time tasks (with an associated priority given by the `priority` clause), and the nested OpenMP tasks implementing the parallel execution of each real-time task.

3.2 Implementing Recurrent Real-Time Tasks in OpenMP

The OpenMP tasking model is very convenient to implement critical real-time systems based on DAG scheduling models. However, OpenMP lacks an important feature of these systems, *the notion of recurrency*. As presented in Sect. 2, the execution of real-time tasks can be either periodic or sporadic triggered by an event, e.g., an internal clock or a sensor.

With the objective of including recurrency in the OpenMP execution model, we propose to incorporate a new clause, named `event`, associated to the `task` construct. This clause enables to define the release time of the OpenMP tasks implementing real-time tasks. The syntax of the `event` clause is as follows:

`#pragma omp task event (`*event-expression*`)`

where *event-expression* is an expression, if it evaluates to true the associated OpenMP task is created. This expression represents the exact moment in time[2] at which the real-time task release occurs or the external event that must occur for the real-time task to release a new job. The expression is true whenever the task releases, and shall evaluate to false after the task creation. However, the `event` clause is not enough to state the synchrony between the event that triggers a real-time task and the actual execution of that task. In languages such as Ada, which are intrinsically concurrent, events are treated at the base language level, thus an Ada task triggering an event will launch an *entry* (a functionality) of a different task. But OpenMP is defined on top of C, C++ and Fortran, languages intrinsically sequential, that do not typically provide these kind of features. Following, we analyze three different approaches to associate the occurrence of an event and the execution of a real-time task:

– *Managed by the base language:* a simple approach would use the base language to implement an infinite loop containing the set of real-time tasks with their corresponding events and priorities. This solution however renders one thread useless, executing the control loop. Interestingly, C++11 introduces multi-threading support, adding features to define concurrent execution.
– *Managed by the operating system:* based on the previous approach, the thread executing the control loop may be freed at the end of each iteration, and the operating system may return the thread to the control loop in a period of time shorter than the minimum period of a task (ensuring no job is missed).
– *Managed by the OpenMP API:* a different approach would be implementing the concept of *persistent* task [27] in the OpenMP API, pushing the responsibility for checking the occurrence of an event to the OpenMP runtime.

A deeper evaluation of the most suitable solution to implement critical real-time system is of paramount importance to promote the use of OpenMP in critical real-time environments. This evaluation is out of the scope of this paper and remains as a future work.

[2] Real-Time Operating Systems (RTOS) provide time management mechanisms and timers to determine the release time or deadline of real-time tasks.

Interestingly, this new `event` clause would allow to unequivocally identify which OpenMP tasks implement real-time tasks, differentiating them from the OpenMP tasks used to parallelize each real-time task. The real-time system implemented in Fig. 4 must therefore include the `event` clause associated to each `task` construct at lines 4, 6 and 9.

4 Implementing Real-Time Scheduling Features in the OpenMP Task Scheduler

One of the most important components of critical real-time systems is the *real-time scheduler*, in charge of, not only assigning the execution of real-time tasks to the underlying computing resources, but also guaranteeing that all tasks execute before its deadline. In the context of multitasking systems, the scheduling policy is normally priority driven [16], i.e., real-time tasks have a *priority* assigned and the preference to execute is given to the highest-priority tasks. A scheduler may preempt a running task if a more urgent task is ready to execute. The interrupted task resumes later its execution. Moreover, different scheduling algorithms place additional restrictions as to where real-time tasks are allowed to execute. Overall, real-time schedulers can be classified based on: (1) task priorities, (2) preemption strategies and (3) allocation strategies. Following, we describe how these features can be supported by the OpenMP specification.

4.1 Priority-Driven Schedulers Algorithms

Depending on the restrictions of how to assign priorities to real-time tasks, priority-based schedulers are classified as follows [11]: (1) *Fixed Task Priority (FTP)*, (2) *Fixed Job Priority (FJP)*, and (3) *Dynamic Priority (DM)*. In FTP, each real-time task has a unique fixed priority. This is the case of the rate-monotonic (RM) scheduler that assigns the priorities based on the period (i.e., tasks with smaller periods have higher priority). In FJP, different jobs of the same real-time task may have different priorities. This is the case of the earliest deadline first (EDF) scheduler that assigns greater priorities to the jobs with earlier deadlines. In DM, the priority of each job may change between its release time and its completion. This is the case of the least laxity (LL) scheduler that assigns the priorities based on the laxity[3] of a job.

In OpenMP, the `priority` clause associated to the `task` construct matches the priority representation of real-time tasks for the FTP scheduling. However, the OpenMP specification (version 4.5) states that *"the priority clause is a hint for the priority of the generated task [..] Among all tasks ready to be executed, higher priority tasks are recommended to execute before lower priority ones. [...] A program that relies on task execution order being determined by this priority-value may have unspecified behavior"*. As a result, the current behavior of the

[3] The *laxity* of a job at any instant in time is defined as its deadline minus the sum of its remaining processing time and the current time.

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task deadline(D_1) event(e_1) // τ_1 : OpenMP−DAG_1
  { part_00
    #pragma omp task depend(out:x)     // T_1
    { part_10
      #pragma omp task                 // T_2
      { part_20 }
      part_11
    }
    part_01
    #pragma omp task                   // T_3
    { part_30 }
    #pragma omp task depend(in:x)      // T_4
    { part_40 }
  }
  ...
  #pragma omp task deadline(D_n) event(e_n) // τ_n : OpenMP−DAG_n
  {   ...   }
}
```

Fig. 5. OpenMP real-time system designed for a deadline-based scheduler.

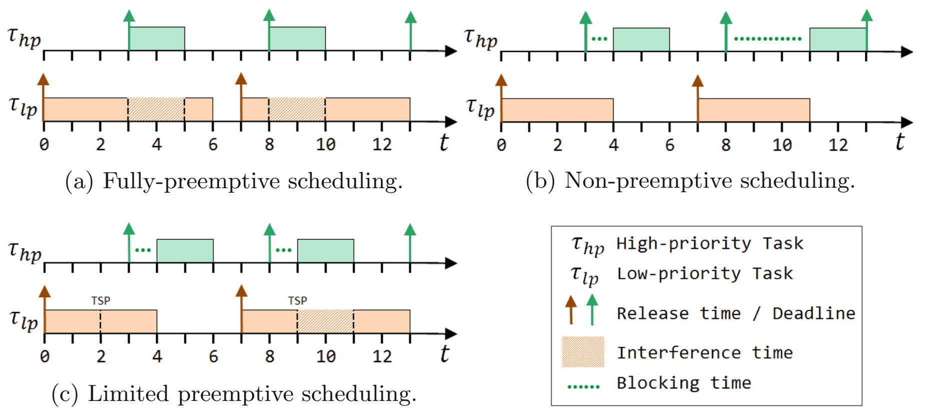
`priority` clause does not guarantee the correct priority-based execution order of real-time tasks. Therefore, the development of OpenMP task schedulers in which the `priority` clause truly leads the scheduling behavior is essential for real-time systems. Moreover, the *priority-expression* value defined at real-time task level must be inherited by the corresponding child tasks implementing parallelism within each real-time task. By doing so, the OpenMP task scheduler can preempt the OpenMP tasks conforming a low priority real-time task in favour of higher priority tasks.

Regarding the implementation of EDF and LL schedulers, a new clause, named `deadline`, associated to the `task` construct is needed. This clause will enable to define the deadline of the real-time task upon which EDF and LL schedulers are based. The syntax of the `deadline` clause is as follows:

`#pragma omp task deadline (`*deadline-expression*`)`

where the *deadline-expression* is the expression that determines the time instant at which the OpenMP task must finish. Similarly to the `priority` clause, the *deadline-expression* associated to an OpenMP task implementing a real-time task must be inherited by all its child tasks. This allows the scheduler to identify those OpenMP tasks with the farthest deadline, and preempt them to assign the corresponding OpenMP threads to those tasks with the closest deadline. The `deadline` clause is not compatible with the `priority` clause, as both are meant for determining the priority of a task for different scheduling algorithms.

Figure 5 shows an example of an OpenMP real-time system, when the scheduler is EDF or LL, and so the `deadline` clause is required. Real-time tasks $\tau_1 \dots \tau_n$ have a deadline and an event associated to them. Notice that, in case of a fixed task priority scheduler, the `deadline` clause would be replaced by a `priority` clause. Real-time task τ_1 corresponds to the real-time task represented



(a) Fully-preemptive scheduling. (b) Non-preemptive scheduling.

(c) Limited preemptive scheduling.

Fig. 6. Scheduling preemption strategies in a single core.

in Fig. 3. All child tasks inherit the deadline of the parent task, for instance, T_1, T_2, T_3 and T_4 inherit the deadline D_1.

4.2 Preemption Strategies

The real-time scheduling theory defines three different types of preemption strategies: (1) *fully-preemptive* (FP), (2) *non-preemptive* (NP), and (3) *limited preemptive* (LP). The FP strategy [9] preempts the execution of low priority tasks as soon as a higher priority task releases. This strategy allows high-priority tasks not to suffer any blocking due to low priority ones. However, it may lead to prohibitively high preemption overheads, mainly related to task context switches and migration delays [15], which may degrade the predictability and performance of the system. The NP strategy [14] executes real-time tasks until completion with no interruption. This strategy offers an alternative that avoids preemption related overheads at the cost of potentially introducing significant blocking effects to higher priority tasks. So far, this strategy has only been considered for sequential real-time tasks. The reason is that parallel real-time tasks may require a different number of computing resources during its execution, and the NP strategy shall guarantee that these resources are always available to avoid preemption operations. Finally, the LP strategy [17] has been proposed as an effective scheduling scheme that reduces the preemption-related overheads of FP, while constraining the blocking effects of NP, thus improving predictability. In LP, preemptions can only take place at certain points during the execution of a real-time task, dividing its execution in non-preemptive regions.

Figure 6 illustrates the three preemption strategies presented above. It considers a task set composed of two tasks: a high-priority task τ_{hp}, with a WCET $C_{hp} = 2$, and a period $T_{hp} = 5$ time units, and a low-priority task τ_{lp}, with a WCET $C_{lp} = 4$, and a period $T_{lp} = 7$ time units. In order to facilitate the explanation, we consider that: a single core is used and real-time tasks are sequential.

Moreover, the deadline is equal to the period, so arrows represent the release time of a given job, and the deadline of the previous job. In the FP strategy (Fig. 6a), as soon as τ_{hp} is released, at times $t = 3$ and $t = 8$, τ_{lp} is preempted and it resumes as soon as τ_{hp} has finished. In the NP strategy (Fig. 6b), although τ_{hp} is released at time instant $t = 3$, it must wait 1 time unit, until τ_{lp} finishes. At the following τ_{hp} release, it must wait again 3 time units. In the LP strategy (Fig. 6c), τ_{lp} defines one preemption point (named as TSP). In the first release of τ_{hp}, at time instant $t = 3$, the preemption point of τ_{lp} has already passed, so τ_{hp} must wait until τ_{lp} has finished. In the second release, at time instant $t = 8$, τ_{hp} must wait only until the preemption point of τ_{lp}, at $t = 9$. Then τ_{lp} is preempted, and τ_{hp} starts its execution.

Interestingly, the OpenMP tasking model implements an LP strategy as explained in Sect. 2.3: OpenMP tasks are preemptable only at TSPs, dividing the task into multiple non preemptive task part regions. Accordingly, the OpenMP runtime can preempt OpenMP tasks and assign its corresponding threads to a different OpenMP task based on the priorities. It is worth noting that OpenMP provides the `taskyield` construct, which allows the programmer to explicitly define additional TSPs. However, regarding task scheduling points, the OpenMP API states that *"the implementation may cause it to perform a task switch"* and regarding the `taskyield` clause, *"the current task can be suspended in favor of execution of a different task"*. This means that an implementation is not forced to perform a task switch in any case. However, in real-time scheduling a TSP must be evaluated, meaning that if a higher priority task is ready at that point, then the lower priority one must be suspended. Therefore, limited preemptive OpenMP schedulers must implement the evaluation of each TSP occurrence.

Interestingly, this laxity in the OpenMP specification, which establishes that threads are allowed to, but not forced to, suspend a task at TSPs, supports the implementation of NP strategies. By simply disabling the suspension of tasks at those points, the OpenMP scheduler would be non-preemptive. In fact, for sequential real-time tasks, this is the default preemption strategy, since there are no implicit TSPs. In this case, it is worth noting that the `taskyield` construct allows the implementation of the LP strategy in sequential real-time tasks as well.

Finally, OpenMP does not support the implementation of FP scheduling strategies because that would require the runtime to preempt the execution of OpenMP tasks at any point of its execution. In any case, as we stated above, FP is not a desirable strategy due to very high preemption overheads it may cause, which can degrade the predictability of the system.

4.3 Allocation and Migration Strategies

There exist three strategies to allocate the execution of real-time tasks to the underlying computing resources (in our case, cores): (1) the *static allocation*, which statically assigns real-time tasks to cores at design time, with the objective of increasing the predictability and minimizing the response time of the overall system; (2) the *dynamic allocation*, in which the allocation is performed based on runtime information, such as the state of the platform (e.g., computing and communication resources available), the set of ready tasks, or the location of input data; (3) the *hybrid allocation*, which statically allocates a subset of real-time tasks, while the rest are dynamically scheduled.

Moreover, real-time schedulers define *migration strategies* to stablish the cores in which real-time tasks are permitted to execute. These strategies can be grouped in three categories: (1) *global scheduling* algorithms allow any real-time task to execute upon any core, allowing jobs from the same real-time task to migrate, (2) *partitioned scheduling* algorithms assign each real-time task to a core so that each job of a real-time task executes always on the same core, and (3) *federated scheduling* algorithms that combine global and partitioned schedulers for a subset of tasks. Typically, the dynamic allocation strategy is built upon global scheduling, whereas static allocation is built upon partitioned scheduling.

Although the OpenMP specification says nothing about allocation strategies, current OpenMP systems are performance-driven, and so all runtime implementations are based on dynamic scheduling. However, static allocation strategies have been proposed for OpenMP as well [24]. In case of migration strategies, the OpenMP *tied* tasking model (the default one) limits the implementation of global schedulers. *Tied* tasks are those that, when suspended, can only be resumed by the same thread that started its execution. As a result, a real-time task implemented as an OpenMP tied task will not be able to migrate. This is not the case of *untied* task, that can be resumed by any thread in the team. In this case the `untied` clause attached to the `task` directive is required.

OpenMP Task to OpenMP Thread Mapping
With the objective of increasing time predictability, most of the real-time schedulers consider a direct mapping between real-time tasks and cores. This includes two conditions: (1) threads are mapped to cores in a one-to-one manner, and (2) threads are not allowed to migrate between cores.

OpenMP threads are an abstraction of the computing resource upon which OpenMP tasks execute. As stated in Sect. 3, this paper considers a single team of threads to execute all OpenMP tasks. This enables the real-time scheduler to have full control over the execution of OpenMP tasks over threads. However, OpenMP threads are further assigned to the operating system, hardware threads and cores (referred to as *places* in OpenMP), existing other levels of scheduling out of the control of the OpenMP scheduler.

Fortunately, the OpenMP specification provides mechanisms to consider a single real-time scheduler and so fulfilling the two conditions stated above. On one hand, the `requires` directive, which will be introduced in the next OpenMP specification, version 5.0 [4], allows to specify the features an implementation must provide in order for the code to compile and execute correctly. This may be useful to express the minimum number of cores that the target architecture must provide to guarantee a one-to-one mapping, as required by the system. On the other hand, OpenMP defines the *bind-var* internal control variable together with the `proc_bind` clause, which allow to control the binding of OpenMP threads

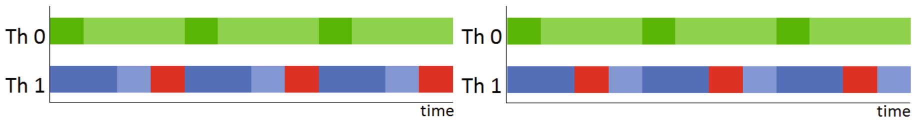
```
#pragma omp parallel
#pragma omp single
{
  while (1) {
    #pragma omp task priority(1) // T1
    {
      work_11();
      #pragma omp taskyield
      work_12();
    }
    work_01();
    #pragma omp task priority(2) // T2
    {  work_21();   }
    work_02();
}}
```

(a) OpenMP code.

🟩 *work*_01()
🟩 *work*_02()

T_1 Low priority task
🟦 *work*_11()
🟦 *work*_12()

T_2 High priority task
🟥 *work*_21()



(b) Execution traces, GCC 8.1. (c) Execution traces, Nanos++.

Fig. 7. Real-time system example: LP scheduling and the `priority` clause. (Color figure online)

to cores, enabling to define different thread-affinity policies. Finally, the *place-partition-var* internal control variable controls the list of places available.

Overall, an OpenMP framework intended to implement a critical real-time system must obey the following constraints: (a) *place-partition-var* := `cores`, so that each OpenMP place corresponds to a single core; and (b) *bind-var* := `close`, so that OpenMP threads are consecutively assigned to places (forbidding threads migration between places). Once OpenMP threads are assigned to cores, this affinity must not be modified. Therefore, the `proc_bind` clause must be forbidden or ignored. Moreover, we propose to use the `requires` directive along with the `ext_min_cores` clause and an integer value, to determine the minimum number of threads (and so, cores) necessary to correctly execute the system.

4.4 Evaluation of Current OpenMP Implementations

This section evaluates how priorities and preemptions are treated in the OpenMP runtime implementation provided by GCC 8.1 [28] and Nanos++ [8]. To do so, we consider the source code presented in Fig. 7a, in which two real-time tasks, T_1 and T_2, are created, T_1 having lower priority than T_2. Moreover, T_1 includes an explicit TSP by means of the `taskyield` construct. Therefore, T_1 is divided into two non-preemptive task parts. Sequential real-time tasks and two threads have been considered for simplicity. Current OpenMP implementations only support dynamic allocation and global scheduling.

Critical real-time systems must honor the priority of each task because these determine preeminence of some tasks over others. Moreover, in the LP strategy, low priority tasks are preempted at preemption points (TSPs) in favour of high priority ones to guarantee that all tasks meet its deadline. Hence, in the example shown in Fig. 7a, T_1 gets first the idle thread as it is created before T_2. However, if T_2 is already created (and ready to execute) at the TSP of T_1 (line 8), T_1 must be preempted and the thread must be assigned to T_2 to honor priorities.

The execution traces[4] of three iterations of the source code presented in Fig. 7a are shown in Fig. 7b, using GCC 8.1, and Fig. 7c, using Nanos++. Green blocks represent the execution of the code within the `single` construct (*work_01* and *work_02*) in the thread *Th* 0. Blue blocks represent the execution of T_1 (*work_11* and *work_12*) in *Th* 1. Red blocks represent the execution of T_2 (*work_21*) in *Th* 1. The exact expected behavior is observed in Nanos++, since T_2 executes between the two task parts of T_1. However, in GCC, T_2 executes after T_1 completes, because the preemption point of T_1 is not honored: T_2 is not executed as soon possible.

Overall, although current OpenMP runtimes are not ready to support the development and execution of critical real-time systems, Nanos++ already implements some of the fundamental features needed by critical real-time systems. This is not the case of GCC 8.1.

5 Related Work

The performance requirements of advanced embedded critical real-time systems entails a booming trend to use multi-core, many-core and heterogeneous architectures. As we stated in early sections of this paper, OpenMP has been already considered to cope with these performance needs [1,21]. In this context, OpenMP has been analyzed regarding the two features that are mandatory in such restricted systems: timing analysis and functional safety.

From a timing perspective, there is a significant amount of work considering the time predictability properties of OpenMP. Despite the fork-join was firstly considered [22], the tasking model seems to be more suitable given its capabilities to define fine grain, both structured and unstructured parallelism. For this reason several works [24,35,38,40] studied the OpenMP tasking model and its similarities with the sporadic DAG scheduling model. However, none of these works consider a complete real-time system, but a unique non-recurrent real-time task. The schedulability analysis of a full DAG task-based real-time system has been addressed for homogeneous architectures under different scheduling strategies [10,12,19,23,33,34]. Recently, a response-time analysis has been proposed for a DAG task supporting heterogeneous computing [36]: the OpenMP accelerator model is proposed to address heterogeneous architectures. From a functional safety perspective, OpenMP is considered as a convenient candidate to implement real-time systems, although some features and restrictions must be addressed [29]. Based on the potential of existent correctness techniques for OpenMP, it could be introduced in safe languages such as Ada [30–32], widely

[4] Traces obtained with Extrae and Paraver performance monitoring tools [6,7].

used to implement safety-critical systems. The Ada Rapporteur Group is considering the introduction of OpenMP into Ada [26] to exploit fine grain parallelism.

Finally, as embedded systems usually have tight constraints regarding resources such as memory (e.g., the Kalray MPPA has 2MB shared memory [18]), different approaches for developing lightweight OpenMP runtime systems [39,41] coexist. These studies are meant to efficiently support OpenMP in such constrained environments. For instance, the memory used at runtime is reduced when the task dependency graph of the applications is statically derived.

6 Conclusions

OpenMP is a firm candidate to address the performance challenges of critical real-time systems. However, OpenMP was originally intended for a different purpose than critical real-time systems, for which guaranteeing the correct output is as important as guaranteeing it within a predefined time budget. In this paper, we evaluate the use of the OpenMP tasking model to develop and execute the sporadic DAG-based scheduling model upon which many critical real-time systems are based on, e.g., IMA and AUTOSAR used in avionics and automotive respectively. Concretely, we propose the use of a single team of threads to implement and execute both, concurrent real-time tasks and the parallelism within them. Two new clauses, `event` and `deadline`, are proposed to allow the implementation of recurrent real-time tasks and FJP and DM schedulers. Moreover, we analyze some important features already provided in the OpenMP API: the `priority` clause and the TSPs. The defined behavior of these two features is not desirable for critical real-time systems. In both cases, it must be a prescriptive modifier, instead of a hint (the case of the `priority` clause) or a possibility of occurrence (the case of TSPs). In order to implement limited preemptive scheduling, the most suitable preemptive strategy for OpenMP real-time systems, it must be guaranteed that, at each preemption point (TSP), if there is a higher priority task ready, the running task is suspended in favor of the highest priority task. Overall, correctly addressing all these features in the specification is of paramount importance to use OpenMP in critical real-time systems.

Nevertheless, some design implications need a deeper analysis and evaluation. This is the case, for instance, of the event-driven execution model not supported in OpenMP, which remains as future work.

Acknowledgments. The research leading to these results has received funding from the Spanish Ministry of Science and Innovation, under contract TIN2015-65316-P, and from the European Union's Horizon 2020 Programme under the CLASS Project (www.class-project.eu), grant agreement No 780622.

References

1. P-SOCRATES European Project (Parallel Software Framework for Time-Critical Many-Core Systems). http://p-socrates.eu
2. ARINC Specification 653: Avionics application software standard standard interface, part 1 and 4 (2012)
3. OpenMP Application Programming Interface (2015). http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
4. OpenMP Technical Report 6: Version 5.0 Preview 2 (2017). http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf
5. AURIX^{TM}Safety Joins Performance (2018). https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/aurix-safety-joins-performance/
6. Barcelona Supercomputing Center: Extrae release 3.5.2. https://tools.bsc.es/extrae
7. Barcelona Supercomputing Center: Paraver release 4.7.2. https://tools.bsc.es/paraver
8. Barcelona Supercomputing Center: OmpSs 1.0 specification (2016). https://pm.bsc.es/ompss-docs/specs/
9. Baruah, S.: Techniques for multiprocessor global schedulability analysis. In: Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS) (2007)
10. Baruah, S.: The federated scheduling of constrained-deadline sporadic DAG task systems. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2015)
11. Baruah, S., Bertogna, M., Buttazzo, G.: Multiprocessor Scheduling for Real-Time Systems. ES. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-08696-5
12. Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A.: The global EDF scheduling of systems of conditional sporadic dag tasks. In: Proceedings of the 27th IEEE Euromicro Conference on Real-Time Systems (ECRTS) (2015)
13. Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., Wiese, A.: A generalized parallel task model for recurrent real-time processes. In: Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS) (2012)
14. Baruah, S.K., Chakraborty, S.: Schedulability analysis of non-preemptive recurring real-time tasks. In: Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS) (2006)
15. Bastoni, A., Brandenburg, B., Anderson, J.: Cache-related preemption and migration delays: empirical approximation and impact on schedulability. In: Proceedings of OSPERT (2010)
16. Buttazzo, G.C.: Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications. Real-Time Systems Series, vol. 24. Springer Science & Business Media, Boston (2011). https://doi.org/10.1007/978-1-4614-0676-1
17. Buttazzo, G.C., Bertogna, M., Yao, G.: Limited preemptive scheduling for real-time systems. A survey. IEEE Trans. Ind. Inform. **9**(1), 3–15 (2013)
18. De Dinechin, B.D., Van Amstel, D., Poulhiès, M., Lager, G.: Time-critical computing on a single-chip massively parallel processor. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2014)
19. Fonseca, J., Nelissen, G., Nelis, V., Pinho, L.M.: Response time analysis of sporadic DAG tasks under partitioned scheduling. In: Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES) (2016)

20. AUTOSAR GbR: AUTomotive Open System ARchitecture (AUTOSAR), Standard v4.1 (2014). http://www.autosar.org
21. Hanawa, T., Sato, M., Lee, J., Imada, T., Kimura, H., Boku, T.: Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 15–27. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02303-3_2
22. Lakshmanan, K., Kato, S., Rajkumar, R.: Scheduling parallel real-time tasks on multi-core processors. In: Proceedings of the IEEE 31st Real-Time Systems Symposium (RTSS) (2010)
23. Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., Buttazzo, G.C.: Response-time analysis of conditional DAG tasks in multiprocessor systems. In: Proceedings of the 27th IEEE Euromicro Conference on Real-Time Systems (ECRTS) (2015)
24. Melani, A., Serrano, M.A., Bertogna, M., Cerutti, I., Quiñones, E., Buttazzo, G.: A static scheduling approach to enable safety-critical OpenMP applications. In: Proceedings of the 22nd IEEE Asia and South Pacific Design Automation Conference (ASP-DAC) (2017)
25. Mok, A.K.: Task management techniques for enforcing ED scheduling on periodic task set. In: Proceedings of the 5th IEEE Workshop on Real-Time Software and Operating Systems (1988)
26. Pinho, L.M., Quiñones, E., Royuela, S.: Combining the tasklet model with OpenMP. In: 19th International Real-Time Ada Workshop (2018)
27. Pop, A., Cohen, A.: A stream-computing extension to OpenMP. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, pp. 5–14. ACM (2011)
28. GNU Project: GNU libgomp, November 2015. https://gcc.gnu.org/projects/gomp/
29. Royuela, S., Duran, A., Serrano, M.A., Quiñones, E., Martorell, X.: A functional safety OpenMP* for critical real-time embedded systems. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 231–245. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_16
30. Royuela, S., Martorell, X., Quiñones, E., Pinho, L.M.: OpenMP tasking model for ada: safety and correctness. In: Blieberger, J., Bader, M. (eds.) Ada-Europe 2017. LNCS, vol. 10300, pp. 184–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60588-3_12
31. Royuela, S., Martorell, X., Quiñones, E., Pinho, L.M.: Safe parallelism: compiler analysis techniques for ada and OpenMP. In: Casimiro, A., Ferreira, P.M. (eds.) Ada-Europe 2018. LNCS, vol. 10873, pp. 141–157. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92432-8_9
32. Royuela, S., Pinho, L.M., Quiñones, E.: Converging safety and high-performance domains: integrating OpenMP into ada. In: Design, Automation Test in Europe Conference Exhibition (2018)
33. Serrano, M.A., Melani, A., Bertogna, M., Quiñones, E.: Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2016)
34. Serrano, M.A., Melani, A., Kehr, S., Bertogna, M., Quinones, E.: An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling. In: Proceedings of the International Symposium on Real-Time Distributed Computing (ISORC) (2017)

35. Serrano, M.A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quiñones, E.: Timing characterization of OpenMP4 tasking model. In: Proceedings of the IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) (2015)
36. Serrano, M.A., Quiñones, E.: Response-time analysis of DAG tasks supporting heterogeneous computing. In: Proceedings of the Annual Design Automation Conference (DAC) (2018)
37. Stotzer, E., et al.: OpenMP on the low-power TI keystone II ARM/DSP system-on-chip. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 114–127. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_9
38. Sun, J., Guan, N., Wang, Y., He, Q., Yi, W.: Scheduling and analysis of real-time OpenMP task systems with tied tasks. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS) (2017)
39. Tagliavini, G., Cesarini, D., Marongiu, A.: Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking. IEEE Trans. Parallel Distrib. Syst. **29**(9), 2150–2163 (2018)
40. Vargas, R., Quiñones, E., Marongiu, A.: OpenMP and timing predictability: a possible union? In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2015)
41. Vargas, R.E., Royuela, S., Serrano, M.A., Martorell, X., Quiñones, E.: A lightweight OpenMP4 run-time for embedded systems. In: Proceedings of the IEEE 21st Asia and South Pacific Design Automation Conference (ASP-DAC) (2016)

OpenMP User Experiences: Applications and Tools

Performance Tuning to Close Ninja Gap for Accelerator Physics Emulation System (APES) on Intel® Xeon Phi™ Processors

Tianmu Xin[1(✉)], Zhengji Zhao[2], Yue Hao[1], Binping Xiao[1], Qiong Wu[1], Alexander Zaltsman[1], Kevin Smith[1], and Xinmin Tian[3]

[1] Collider Accelerator Department, Brookhaven National Lab, Upton, NY 11973, USA
txin@bnl.gov
[2] Lawrence Berkeley National Laboratory, Berkley, CA 94720, USA
[3] Intel Corporation, Santa Clara, USA

Abstract. Radio frequency field and particle interaction is of critical importance in modern synchrotrons. Accelerator Physics Emulation System (APES) is a C++ code written with the purpose of simulating the particle dynamics in ring-shaped accelerators. During the tracking process, the particles interact with each other indirectly through the EM field excited by the charged particles in the RF cavity. This a hot spot in the algorithm that takes up roughly 90% of the execution time. We show how a set of well-known code restructuring and algorithmic changes coupled with advancements in modern compiler technology can bring down the Ninja gap to provide more than 7x performance improvements. These changes typically require low programming effort, as compared to the very high effort in producing Ninja code.

Keywords: APES · RF cavity · Wake field · OpenMP · Xeon Phi™

1 Introduction

Accelerator design and modeling requires intensive computation capability. The need for high performance computation rises not only from simulating billions of particles in the accelerator but also from the dynamics of one particle that depend on the status of the rest of the particles in a certain spatial and temporal range. The latter factor is usually referred to as the 'collective effect', which prevents the embarrassing parallelization of simulating the motion of large amount of particles.

In a synchrotron accelerator, radio frequency (RF) cavities are critical parts which are used in various ways. Although all cavities are designed with a certain frequency in mind, there will always be some coexisting parasitic modes. Usually these modes are in higher frequencies, hence we typically call them higher order modes (HOM). These HOMs could have unwanted, some time destructive influence on the bunches. Therefore, when designing a synchrotron the HOM

B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 163–174, 2018.
https://doi.org/10.1007/978-3-319-98521-3_11

property of RF cavities is one of the most important issue scientists need to address. The way to investigate the problem is to simulate the bunch behavior under the designed cavity, usually for millions of turns, and see if the bunch life time is acceptable. A detailed method will be discussed in the next section.

1.1 Longitudinal Beam Dynamics

In this part we will briefly discuss the dynamics of charged particles in a synchrotron. The typical way of dealing with this problem is to treat the cavity and the ring separately. The RF cavity only updates the momentum of the particles, and the ring maps the 6D coordinates of all particles into a new set of coordinates. In this paper we only care about the so-called "kick" that the particles are getting from each mode of the RF cavity, which is the integrated Lorentz force as shown in Eq. 1,

$$\begin{aligned}\Delta \boldsymbol{p} &= \int_{-\infty}^{\infty} \boldsymbol{F}(x,y,t)dt \\ &= \frac{e}{c}\boldsymbol{V} sin\phi.\end{aligned} \tag{1}$$

where e is the charge of the particle, c is the speed of light, $\boldsymbol{V}$ is defined as the voltage of the cavity, and ϕ is the phase of the mode for the particle. As for the ring part, we are performing a simple linear mapping as shown in Eq. 2 [2]

$$\begin{aligned}
x_{i+1} &= x_i\,(cos\psi_x + \alpha_x\, sin\psi_x) + \beta_x\, sin\psi_x\, \frac{p_{x\,i}}{p_{z\,i}} \\
y_{i+1} &= y_i\,(cos\psi_y + \alpha_y\, sin\psi_y) + \beta_y\, sin\psi_y\, \frac{p_{y\,i}}{p_{z\,i}} \\
p_{x\,i+1} &= -x_i\,\frac{(1+\alpha_x^2)}{\beta_x}\, sin\psi_x\, p_{z\,i} + cos\psi_x\, p_{x\,i} \\
p_{y\,i+1} &= -y_i\,\frac{(1+\alpha_y^2)}{\beta_y}\, sin\psi_y\, p_{z\,i} + cos\psi_y\, p_{y\,i} \\
t_{i+1} &= t_i + T_0\eta\frac{\Delta p_{z\,i}}{p_{z\,0}}
\end{aligned} \tag{2}$$

where t_i represents the arrival time of the particle at i^{th} turn, x_i, y_i, $p_{x\,i}$, $p_{y\,i}$, and $p_{z\,i}$ are spacial and momentum coordinates of the particle, α, β, ψ are the ring parameters, T_0 is the revolution time of the bunch, and η is the factor that links the momentum deviation with the arrival time of the particles. This gives us the equation we need for the tracking process for the given values of V and ϕ for each mode. To get those, we need to do the so called Wake field calculation, discussed in next part of this section.

1.2 Wake Field in RF Cavity

When a charged particle is passing through an RF cavity, it will excite a certain amount of EM field, the so called Wake field. The spectrum of the Wake field is depending on the particle and cavity properties. Any charged particle that follows the pilot particle will be influenced by the Wake field. Assuming the cavity has N different modes, the Wake field will be the linear superposition of these modes [1]. Namely,

$$\boldsymbol{G}(t) = \sum_{n=1}^{N_{mod}} \boldsymbol{V}_n e^{i\omega_n t} \tag{3}$$

where $\boldsymbol{G}$ is the Wake field generated by the point charge (Green's Function), $\boldsymbol{V_n}$ is the amplitude of the n^{th} mode component which depends on the cavity property, ω_n is the angular frequency of the n^{th} mode, and t is the trailing time between the pilot point charge and the probe particle. Equation 3 only describes the Wake field generated from a point charge. In reality, most of the particles will be exposed to the Wake field from all the particles in front of them. As shown in Eq. 4, the j^{th} particle will be exposed to the Wake field from all the $j-1$ previously arrived particles.

$$\boldsymbol{W}(t_j) = \sum_{i=1}^{j-1} \boldsymbol{G}(t_j - t_i) \tag{4}$$

where W is the Wake field that is exerted on the j-th particle. **Therefore, this is a causal process that requires information about previous times**.

1.3 Data Structures, Algorithms and Challenges

Accelerator Physics Emulation System (APES) [3] is a C++ code that simulates particle dynamics in ring-shaped accelerators. The core of the APES code is the implementation of Eqs. 2–4, which calculates the Wake field in the RF cavity, through which particles interact with the cavity and also with each other, and updates the phase space coordinates of the particles (integrated Lorentz force). The top hotspot in the code is the Wake field calculation, which accounts for 90% of the total execution time.

There are two major scaling variables that dominate the execution time of the code. One is the total number of particles N_p (~1,000,000) in one beam, and the other is the total number of modes N_{mod} (~1,000) in the RF cavity. The main data structures are the six 1D arrays, x, y, t, p_x, p_y, and p_z, that store the phase space coordinates of the particles and the six 2D arrays VxR, VxI, VyR, VyI, VzR, and VzI that store the Wake field exerted on each particle, with both real and imaginary parts in each direction (Table 1).

The algorithm to calculate the Wake field is straightforward; in the outer loop, we iterate over all the modes of the cavity, and in the inner loop, we

Table 1. Main data arrays used in the APES code

Name	Dimension
x, y, t, p_x, p_y, p_z	$1 \times N_p$
$VxR, VxI, VyR, VyI, VzR, VzI$	$N_{mod} \times N_p$

iterate over all the particles under each mode in their arrival order to calculate the Wake field exerted on a particle by summing up the Wake field generated by all the particles prior to its arrival. The original code is shown in Fig. 1.

```
 1 // Wake Field calculation
 2 for (int i = 0;i<N_mod;++i){ // iterate over number of modes.
 3   double Vbx = kx[i]*bnch.q;
 4   double Vby = ky[i]*bnch.q;
 5   double Vbz = kz[i]*bnch.q;
 6   double dT = bnch.t[bnch.index[0]] - t_last+bm.delay;
 7   double decay = exp(-dT*tau_invert[i]);// decay of wake from last bunch
 8   double dphi = dT*frq[i]*2.0*pi; // phase shift of the wake from last bunch.
 9   // rotated wake from last bunch:
10   VxR[i][0]=VxR[i][bnch.Np]*decay*cos(dphi)-VxI[i][bnch.Np]*decay*sin(dphi);
11   VyR[i][0]=VyR[i][bnch.Np]*decay*cos(dphi)-VyI[i][bnch.Np]*decay*sin(dphi);
12   VzR[i][0]=VzR[i][bnch.Np]*decay*cos(dphi)-VzI[i][bnch.Np]*decay*sin(dphi);
13   VxI[i][0]=VxR[i][bnch.Np]*decay*sin(dphi)+VxI[i][bnch.Np]*decay*cos(dphi);
14   VyI[i][0]=VyR[i][bnch.Np]*decay*sin(dphi)+VyI[i][bnch.Np]*decay*cos(dphi);
15   VzI[i][0]=VzR[i][bnch.Np]*decay*sin(dphi)+VzI[i][bnch.Np]*decay*cos(dphi);
16   for (j = 1;j<bnch.Np;++j){
17     int tempID = bnch.index[j];
18     double dt = bnch.t[bnch.index[j]]-bnch.t[bnch.index[j-1]];
19     double cosin = cos(2.0*pi*frq[i]*dt);
20     double sine = sin(2.0*pi*frq[i]*dt);
21     VxR[i][j] = VxR[i][j-1]*cosin -(VxI[i][j-1]+Vbx*bnch.x[tempID])*sine;
22     VyR[i][j] = VyR[i][j-1]*cosin -(VyI[i][j-1]+Vby*bnch.y[tempID])*sine;
23     VzR[i][j] = (VzR[i][j-1]+Vbz)*cosin-VzI[i][j-1]*sine;
24     VxTI[i][j] = VxR[i][j-1]*sine+(VxI[i][j-1]+Vbx*bnch.x[tempID])*cosin;
25     VyI[i][j] = VyR[i][j-1]*sine+(VyI[i][j-1]+Vby*bnch.y[tempID])*cosin;
26     VzI[i][j] = (VzR[i][j-1]+Vbz)*sine+VzI[i][j-1]*cosin;
27   }
28   VxR[i][bnch.Np] = VxR[i][j-1];
29   VyR[i][bnch.Np] = VyR[i][j-1];
30   VzR[i][bnch.Np] = VzR[i][j-1]+Vbz;
31   VxI[i][bnch.Np] = VxI[i][j-1]+Vbx*bnch.x[bnch.index[bnch.Np-1]];
32   VyI[i][bnch.Np] = VyI[i][j-1]+Vby*bnch.y[bnch.index[bnch.Np-1]];;
33   VzI[i][bnch.Np] = VzI[i][j-1];
34 }
```

Fig. 1. The loops that perform the Wake field calculation in the original APES code. The outer loop iterates over all the modes in the cavity, and the inner loop iterates over all the particles under each mode in their arrival time order. The scalar variables Vbx, Vby and Vbz store the amplitudes of the Wake field in each direction generated by a single point charge, and the two-dimensional arrays, VxR, VxI, VyR, VyI, VzR, and VzI, store the real and imaginary parts of the Wake field in each direction in each mode.

Note that the particles do not arrive at the RF cavity in their index order. Line 17 in Fig. 1 calculates the particle index (tempID) for the j-th arrival particle. We can readily see the backward dependency in the inner loop (lines 21–26), the major challenge we are facing.

2 OpenMP Parallelization

Despite the loop carried dependencies as shown in Fig. 1, we have added the OpenMP parallel directives (#pragma omp parallel for) to the rest of the major loops where parallelizations are possible, examples being the loops over the number of modes (~1000) for the Wake field calculations and coordinate updates (outer loop), the loops over the number of particles (~1,000,000) for the bunch initialization, etc. In addition, we have also deployed the parallelization from the algorithms, such as sort and accumulate, from the GNU Parallel library. Figure 2 (top) shows the thread scaling of this initial OpenMP implementation

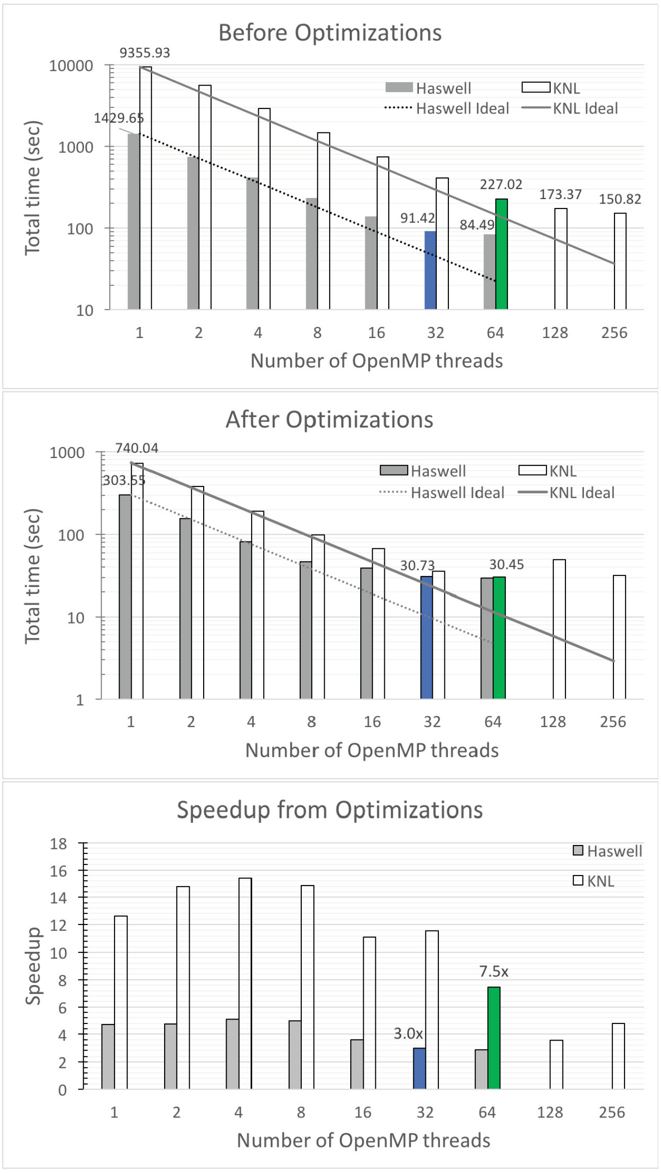


Fig. 2. The OpenMP thread scaling of the APES code before (top) and after (middle) optimizations on Cori Haswell and KNL nodes. The dotted and solid lines show the ideal thread scalings on Haswell and KNL nodes, respectively. The horizontal axis shows the number of threads used, and the vertical axis shows the total runtime of the APES code with the selected benchmark parameters ($N = 1048576$, $N_{mode} = 1024$, $N_{turns} = 10$). The bottom figure shows the total runtime speedup of the optimized APES code in comparison with the initial OpenMP implementation. The highlighted bars in blue (Haswell) and green (KNL) show the results when the number of threads used equals to the number of cores available on the nodes. (Color figure online)

of the APES code on the Cori [4] Haswell and KNL nodes (see the next section for the Cori configuration). The code scales well to the number of available cores on both Haswell and KNL (see the highlighted bars in blue (Haswell, 32 cores) and green (KNL, 64 cores out of 68 available)), and reduces the runtime by >15 and >40 times on Haswell and KNL nodes, respectively. Given the fact that the initial OpenMP implementation is just a straightforward addition of the parallel for directive into the code, the performance gain is impressive, confirming the ease of use of the OpenMP programming model in production codes. Going beyond this to use the Hyper-Threadings on the nodes, the scaling drops quickly, although the run time reduces further.

Following the initial OpenMP implementation, we addressed a few top performance issues, which will be described in detail in the next section and is the main focus of this paper; this achieved a significant performance boost. Figure 2 (bottom) shows the speedup of the optimized code with respect to the initial OpenMP implementation at a number of thread counts. As shown with the highlighted bars (blue for Haswell and green for KNL), we have achieved 3.0 and 7.5 times speedup on Haswell and KNL, respectively, when using all the cores available on the nodes. Figure 2 (middle) shows the thread scaling of the optimized APES on Cori. Note that the vertical axis is scaled down 10 fold in Fig. 2. The thread scaling of the optimized code drops beyond 16 (Haswell) and 32 (KNL) threads, scaling not as well as the original code. This is mainly due to the loop carried dependency in the top loop that is neither parallelizable nor vectorizable and takes similar runtime before and after the optimizations on other parts of the code. However, the optimal runtime is still achieved when using all the cores available.

3 Code Optimizations

In this section, we will describe in detail the optimizations we have deployed in the APES code, including both successful and unsuccessful attempts. Throughout the optimization process, we heavily relied on the Intel compiler optimization reports to identify the performance issues and also to confirm the fixes after addressing them. We also used Intel VTune [5] and Advisor [6] for some of the performance analyses. Our benchmark tests were run on Cori [4] at NERSC, a Cray XC40 system.

3.1 System Configuration and Benchmark Case

Cori has 9688 single-socket KNL (Intel Xeon Phi Processor 7250 ("Knights Landing") nodes @1.4 GHz each with 68 cores/272 threads, a 16 GB high bandwidth on-package memory (HBM or MCDRAM) (>400 GB/sec) and a 96 GB DDR4 2400 MHz memory (102 GB/sec). Each core has two 512-bit vector units, a 64 KB (32 KB instruction, 32 KB data) L1 cache, and shares a 1 MB L2 cache with the other core on the tile (a tile consists of two cores). The MCDRAM is configured as a cache (last level) on Cori. In addition to the KNL nodes, Cori has 2388 dual-socket 16-core Intel Xeon Processor E5-2698 v3 ("Haswell") nodes @2.3 GHz each with 32 cores (64 threads) and a 128 GB 2133 MHz DDR4 memory. Each core has two 256-bit vector units, a 64 KB L1 (32 KB instruction cache and 32 KB data), and a 256 KB L2 cache, sharing a 40-MB L3 cache among the 16 cores on the socket. Cori nodes are interconnected with Cray's Aries network with Dragonfly topology. Cori runs Cray Linux Environment (CLE 6.0 Update 4) and uses SLURM (17.11) as its batch system.

The APES code was compiled with Intel compiler 2018.1.163, and used the GNU parallel libraries in GCC 7.2.0. We have selected the following parameters for the benchmark case:

$$\begin{aligned} Number\ of\ Particles &= 1,048,576 \\ Number\ of\ Modes &= 1024 \\ Number\ of\ Turns\ to\ track &= 10 \end{aligned}$$

In the following sections, we will focus on the KNL timings only. We ran each test more than five times, and reported the average time. All of our runs used 64 cores out of the 68 available on a Cori KNL node, which is the recommended optimal use of the Cori KNL nodes.

3.2 Algorithmic Change to Address Indirect Memory Access

One of the bottlenecks we have identified is the indirect memory access in the code. The relevant code snippet is shown in Fig. 3. We have modified the algorithm to address this issue. Instead of just sorting the particle indices, we sort all the coordinate arrays based on the particle arrival time. This is equivalent to reassigning indices to particles each turn instead of using the pre-assigned particle indices throughout the tracking process. See Fig. 4 for the corresponding code changes. By doing so, we are able to eliminate the indirect memory access in the code. Note that we have sacrificed some time in sorting five more arrays and data rearranging, but the total performance gain from direct memory access has a much higher overall payoff. Table 2 (row 3) shows the performance improvement after this issue was addressed, achieving a two times performance boost in comparison to the initial OpenMP implementation.

3.3 Parallelization of Vector Initialization

In our initial OpenMP implementation, we did not parallelize the initialization of the Wake field arrays, although they can be easily parallelized. Because the initialization itself did not take significant time. We identified this issue from the compiler optimization report, which is especially important for processors with multiple NUMA domains for improved data locality for each thread. We have added the "#pragma omp parallel for" before the Wake filed array initialization loop, as shown in Fig. 5. This small code change has resulted in a large performance increase. As shown in Table 2 (row 4) the accumulative performance has

```
// sort the particles (index) in the order of their arrival times
__gnu_parallel::sort(index.begin(),index.end(),[&](const double& a,const double& b){
     return (t[a]<t[b]);});
//loop over all modes and all particles
  for(int i = 0;i<N_mod;++i){ //trip counts=1024 :
    for (j = 1;j<bnch.Np;++j){ //trip counts = 1048576
      int tempID = bnch.index[j];
      double dt = bnch.t[bnch.index[j]]-bnch.t[bnch.index[j-1]];
      double cosin = cos(2.0*pi*frq[i]*dt);
      double sine = sin(2.0*pi*frq[i]*dt);
      VxR[i][j] = VxR[i][j-1]*cosin -(VxI[i][j-1]+Vbx*bnch.x[tempID])*sine;
      VyR[i][j] = VyR[i][j-1]*cosin -(VyI[i][j-1]+Vby*bnch.y[tempID])*sine;
      VxI[i][j] = VxR[i][j-1]*sine+(VxI[i][j-1]+Vbx*bnch.x[tempID])*cosin;
      VyI[i][j] = VyR[i][j-1]*sine+(VyI[i][j-1]+Vby*bnch.y[tempID])*cosin;
    }
  }
    ...
//loop over all modes and all particles in the order of their arrival times
  for (int i = 0;i<N_mod;++i){ // trip counts = 1024
    #pragma omp for
    for (int j = 0; j<bnch.Np; ++j) { // trip counts = 1048576
      int tempID = bnch.index[j];
      double dphi = 2.0 * pi*fi*bnch.t[tempID]+phiN;
      double cosphi = cos(dphi);
      double sinphi = sin(dphi);
      bnch.px[tempID] += qoc*(V0xR[i]*cosphi-V0xI[i]*sinphi+VxR[i][j]);
      bnch.py[tempID] += qoc*(V0yR[i]*cosphi-V0yI[i]*sinphi+VyR[i][j]);
      bnch.pz[tempID] += qoc*((V0zR[i]*cosphi-V0zI[i]*sinphi+VzR[i][j])+
            (V0xR[i]*cosphi-V0xI[i]*sinphi)*(2*pi*fi)/c*bnch.x[tempID]+
            (V0yR[i]*cosphi-V0yI[i]*sinphi)*(2*pi*fi)/c*bnch.y[tempID]+
            Vbz*0.5);
    }
  }
```

Fig. 3. This code snippet shows the indirect memory access incurred in the original APES code. The variable tempID is the index of the particle that arrives j-th at the cavity. The particles do not arrive in their index order.

```
//sort all six phase space coordinates
void sort_based_on_index(std::vector<double>& x,std::vector<int>& indx){
  std::vector<double> temp(x);
  #pragma omp parallel for
  for(unsigned int i = 0;i<x.size();++i){
    x[i] = temp[indx[i]];
  }
}
__gnu_parallel::sort(index.begin(),index.end(),[&](const double& a,const double& b){
     return (t[a]<t[b]);});
  sort_based_on_index(x,index);
  sort_based_on_index(y,index);
  sort_based_on_index(px,index);
  sort_based_on_index(py,index);
  sort_based_on_index(pz,index);
  __gnu_parallel::sort(t.begin(),t.end());
```

Fig. 4. The optimized code in which all six 1D arrays are sorted in the arrival time order instead of sorting only the particle indices. This eliminates indirect memory access, bnch.x[**tempID**], bnch.y[**tempID**], bnch.px[**tempID**], bnch.py[**tempID**], and bnch.pz[**tempID**] shown in Fig. 3.

been increased by six times in comparison to the initial OpenMP implementation. To show the performance impact from each optimization step, in Table 2 column 3, we have shown the performance boost relative to the previous version. Initializing the Wake filed in a parallel region has resulted in about 3x performance boost from the previous version (sorting all six coordinate arrays).

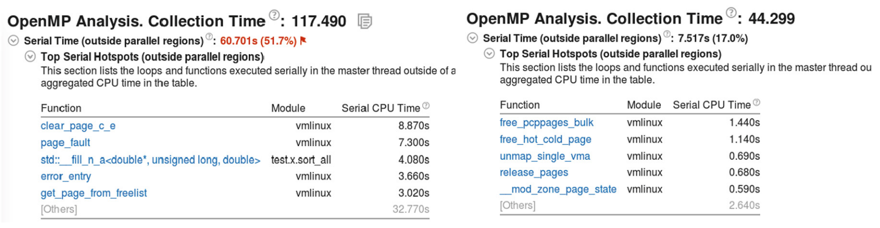
The performance boost appears to be larger than one can usually expect from the first touch, especially on a KNL node with a single socket. We performed the VTune advanced hotspot analyses on the two code versions. VTune showed that when the Wake field was initialized in a parallel region, the master thread had a significantly reduced serial time (7.5 s vs 60.7 s), accounting for ~70% of the

Table 2. APES performance improvement from code optimizations. The first column shows the optimizations deployed, the second column shows the total runtime of the APES code, and the third column shows the performance boost relative to the previous version (to show the performance impact from each optimization step), and the fourth column shows the cumulative performance boost with respect to the initial OpenMP implementations of the APES code.

Optimization	Total runtime (sec)	Progressive boost	Cumulative performance boost
Parallelization of outer loop (initial OpenMP implementation)	227.02	1.00	1.00
Data rearranging for direct memory access	111.91	2.03	2.03
Parallelization of vector initialization	37.33	3.00	6.08
Vectorization of sine and cosine	30.45	1.23	7.46

```
//initialization of Wake Field
#pragma omp parallel for
for (int i = 0;i<N_mod;++i){
  VxR[i].resize(bnch.Np+1,0);
  VyR[i].resize(bnch.Np+1,0);
  VzR[i].resize(bnch.Np+1,0);
  VxI[i].resize(bnch.Np+1,0);
  VyI[i].resize(bnch.Np+1,0);
  VzI[i].resize(bnch.Np+1,0);
}
```

Fig. 5. Initializing the Wake field vector in parallel.



OpenMP Analysis. Collection Time: 117.490

Serial Time (outside parallel regions): 60.701s (51.7%)

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of a
aggregated CPU time in the table.

Function	Module	Serial CPU Time
clear_page_c_e	vmlinux	8.870s
page_fault	vmlinux	7.300s
std::__fill_n_a<double*, unsigned long, double>	test.x.sort_all	4.080s
error_entry	vmlinux	3.660s
get_page_from_freelist	vmlinux	3.020s
[Others]		32.770s

OpenMP Analysis. Collection Time: 44.299

Serial Time (outside parallel regions): 7.517s (17.0%)

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread ou
aggregated CPU time in the table.

Function	Module	Serial CPU Time
free_pcppages_bulk	vmlinux	1.440s
free_hot_cold_page	vmlinux	1.140s
unmap_single_vma	vmlinux	0.690s
release_pages	vmlinux	0.680s
__mod_zone_page_state	vmlinux	0.590s
[Others]		2.640s

Fig. 6. The VTune advanced hotspot summary report for the optimized version that sorts all six arrays (left) and the version that parallelizes the Wake field initialization in addition to sorting all six arrays (right). This section reports the serial execution time of the master thread. When the Wake filed is initialized in a parallel version, the serial time is significantly reduced.

performance improvement. Since the Wake field initialization invokes the C++ vector resize function, the performance impact from the first touch for the Wake field initialization seems to be more complicated than just initializing arrays with scalar values (Fig. 6).

3.4 Vectorization of Elementary Functions

Another performance issue that we have identified is the repeated invocations of the elementary functions such as sine and cosine in the inner loops. The inner loop in Fig. 1 (line 15–31) is one such example. Since the loop iterates over millions of particles, the time spent computing sine and cosine functions was significant. In addition, due to the loop carried dependencies, this loop was (and still is) not vectorizable. As a result, all the sine and cosine invocations were not able to use their vector versions, as shown in the compiler optimization report. We have separated the sine and cosine calculations from the rest of the loop body, and have added the SIMD pragma to vectorize these function calls. We have confirmed the vectorization of these functions from the compiler optimization report after the code change as well. The change to the code is rather small, as shown in Fig. 7. See Table 2 (row 5) for the performance improvement after this change. The accumulative performance increase was about 7.5 times the initial OpenMP implementation.

```
//vectorization of sine and cosine
  for (int i = 0;i<N_mod;++i){
  ...
    std::vector<double> cosin(bnch.Np,0.0);
    std::vector<double> sine(bnch.Np,0.0);
    #pragma omp simd
    for(unsigned k = 0;k<bnch.Np;++k){
      double dphi = 2.0*pi*frq[i]*bnch.dt[k];
      cosin[k] = cos(dphi);
      sine[k] = sin(dphi);
    }
    for (j = 1;j<bnch.Np;++j){
      VxR[i][j] = VxR[i][j-1]*cosin[j]-(VxI[i][j-1]+Vbx*bnch.x[j])*sine[j];
      ...
    }
    ...
  }
```

Fig. 7. The vectorized sine and cosine invocations in the optimized code.

3.5 Nested Parallel Sections

As we have mentioned previously, one of the major performance bottlenecks in the APES code is the loop carried dependencies incurred in the inner loop, which are neither parallelizable nor vectorizable (see Fig. 1). Therefore, the OpenMP parallel has been deployed in the outer for loop, although it has a much smaller trip count (~1000) in comparison to the inner loop (~1,000,000). Nonetheless, we have identified that the OpenMP parallel sections can be deployed to spread the work load of the inner loop to multiple threads. The changed code is shown in Fig. 8. Since the outer loop is already parallelized, these parallel sections have to run in a nested fashion. Unfortunately, this change did not produce any observable performance improvement.

```
#pragma omp parallel sections num_threads(2)
{
  #pragma omp section
  for (j = 1;j<bnch.Np;++j){
    VzR[i][j] = (VzR[i][j-1]+Vbz)*cosin[j]-VzI[i][j-1]*sine[j];
    VyR[i][j] = VyR[i][j-1]*cosin[j]-(VyI[i][j-1]+Vby*bnch.y[j])*sine[j];
    VxR[i][j] = VxR[i][j-1]*cosin[j]-(VxI[i][j-1]+Vbx*bnch.x[j])*sine[j];
  }
  #pragma omp section
  for (j = 1;j<bnch.Np;++j){
    VxI[i][j] = VxR[i][j-1]*sine[j]+(VxI[i][j-1]+Vbx*bnch.x[j])*cosin[j];
    VyI[i][j] = VyR[i][j-1]*sine[j]+(VyI[i][j-1]+Vby*bnch.y[j])*cosin[j];
    VzI[i][j] = (VzR[i][j-1]+Vbz)*sine[j]+VzI[i][j-1]*cosin[j];
  }
}
```

Fig. 8. Parallelization of the inner loop with the OpenMP parallel sections.

3.6 Outer Loop Vectorizations

Considering the fact that the inner loops have trip counts thousands of times larger than the outer loop, another approach we have tried is to vectorize the outer loops from the beginning, and parallelize the inner loop. Of course, this involves transposition of the data arrays, but it only has to be done once at the beginning of the code. We were able to confirm from the compiler optimization report that the outer loop was indeed vectorized. Unfortunately, this optimization didn't result in a performance increase because of the complexity of the inner loop.

3.7 Other Optimization Efforts

The loop carried dependency remains the major performance issue in APES. We have attempted to use the exclusive and inclusive scans available from the Intel Parallel library (PSTL) and also part of the OpenMP 5.0 specification [7] to parallelize the inner loop body. The exclusive scan works only with the following form of the dependency,

$$B[j] = A[0] + A[1] + ... + A[j-1] \tag{5}$$

This appears to be quite similar to the backward dependency in the APES code, but unfortunately, we were not able to use it in APES because its dependent arrays have complicated scaling factors that fail to fit in the format in Eq. 5. It would be very helpful if exclusive/inclusive scans could be extended to support more general forms of the loop carried dependencies in the future.

We have also tried the compiler prefetch and unroll optimizations, experimenting with the levels of prefetch and unroll. Unfortunately, they did not make observable performance differences either.

4 Conclusion

In this paper, we have described how a set of well-known code restructuring and algorithmic changes coupled with advancements in modern compiler technology can bring down the Ninja gap to provide >7 times performance improvements. Specifically, we have restructured the code and modified algorithms to eliminate the indirect memory access incurred in the large loops (~1,000,000,000 trip counts in total). As a result, we have achieved a 2x performance improvement in comparison to the initial OpenMP implementation. Next, we have initialized the main arrays in parallel regions, which significantly reduces the serial execution time, improving the data locality for threads. We have achieved about a 6x performance improvement cumulatively (combined with the previous optimization). Moreover, we have moved the frequent invocations of the elementary functions (sine and cosine) out of the inner loop, which is neither parallelizable nor vectorizable. By doing so, we were able to vectorize the sine and cosine function calculations. We have achieved an overall performance improvement of over 7x.

The loop carried dependency in the inner loop still remains the major performance bottleneck. Although we have tried a few optimization approaches, such as nested parallel sections, prefetch and unroll compiler optimizations, none of them could effectively address this backward dependency issue in APES. The exclusive and inclusive scans in the OpenMP 5.0 specification looked promising; however, until they support more general forms of the data dependencies, APES would not be able to make use of them.

For future work, we will look into adopting MPI in the code, mainly to mitigate the memory bandwidth pressure of the code. APES is memory bandwidth bound code per the Roofline report from Intel Advisor. Distributing data to multiple nodes may help mitigate this issue, with the performance benefiting more from the MCDRAM as well.

Acknowledgments. This work is supported by Brookhaven Science Associates, LLC under contract No. DE-AC02-98CH10886 with the US DOE. Part of this work was performed at the 2018 Intel Knights Landing Hack-a-thon hosted by Brookhaven National Laboratory (BNL), which was partially supported by the HEP Center for Computational Excellence (KA24001022), the SOLLVE Exascale Computing Project (17-SC-20-SC), and the Office of Science of the U.S. Department of Energy under Contract No. DE-SC0012704. Authors would like to thank the Intel engineers who helped during the hack-a-thon. This work is partially supported by the ASCAR Office in the DOE, Office of Science, under contract number DE-AC02-05CH11231, and used the resources of National Energy Scientific Computing Center (NERSC).

References

1. Zotter, B.W., Kheifets, S.: Impedances and Wakes in High Energy Particle Accelerators. ISBN 978-981-02-2626-8
2. Lee, S.Y.: Accelerator Physics. ISBN 978-981-4374-94-1
3. https://github.com/tianmux/tracking
4. http://www.nersc.gov/users/computational-systems/cori/configuration
5. https://software.intel.com/en-us/intel-vtune-amplifier-xe
6. https://software.intel.com/en-us/advisor
7. http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf

Visualization of OpenMP* Task Dependencies Using Intel® Advisor – Flow Graph Analyzer

Vishakha Agrawal[1], Michael J. Voss[1], Pablo Reble[1], Vasanth Tovinkere[1], Jeff Hammond[2], and Michael Klemm[3](✉)

[1] Intel Corporation, 1300 S MoPac Expy, Austin, TX 78746, USA
{vishakha.agrawal,michaelj.voss,pablo.reble,vasanth.tovinkere}@intel.com
[2] Intel Corp., 2501 Northwest 229th Avenue, Hillsboro, OR 97124, USA
jeff.r.hammond@intel.com
[3] Intel Deutschland GmbH, Dornacher Str. 1, 85622 Feldkirchen, Germany
michael.klemm@intel.com

Abstract. With the introduction of task dependences, the OpenMP API considerably extended the expressiveness of its task-based parallel programming model. With task dependences, programmers no longer have to rely on global synchronization mechanisms like task barriers. Instead they can locally synchronize a restricted subset of generated tasks by expressing an execution order through the `depend` clause. With the OpenMP tools interface of Technical Report 6 of the OpenMP API specification, it becomes possible to monitor task creation and execution along with the corresponding dependence information of these tasks. We use this information to construct a Task Dependence Graph (TDG) for the Flow Graph Analyzer (FGA) tool of Intel® Advisor. The TDG representation is used in FGA for deriving metrics and performance prediction and analysis of task-based OpenMP codes. We apply the FGA tool to two sample application kernels and expose issues in their usage of OpenMP tasks.

1 Introduction

The OpenMP API 4.0 specification introduced the `depend` clause to express data dependences between tasks. Using `depend` clauses, developers can create complex dependence patterns that improve performance by expressing parallelism at a finer granularity, exposing more parallelism and reducing the need for less discriminating synchronization constructs like barriers. However, it can be challenging for developers to understand the ordering constraints they have created, making correctness and performance debugging difficult.

Computational graphs, such as Task Dependency Graphs (TDG), are not unique to OpenMP, in fact they have been increasingly adopted as a way for modern software to express applications that operate on streaming data. The Threading Building Blocks (TBB) flow graph API [7], Microsoft's Asynchronous Agents

B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 175–188, 2018.
https://doi.org/10.1007/978-3-319-98521-3_12

Library [11] and TPL Dataflow [12], as well as OpenCL* [8] or OpenVX* [9] from the Khronos* group are examples of libraries that expose an API for explicitly creating computational graphs. Some of these libraries have graphical tools to support the design and analysis of these computational graphs. In particular, Intel® Advisor Flow Graph Analyzer (FGA) is a tool that provides powerful features for the design and analysis of computational graphs created using the TBB flow graph API.

In this paper, we describe an OMPT-based [15] tracing tool that generates data to be imported to the FGA tool for analysis. Using this OMPT tool and FGA, we analyze two sample applications to show that this tool, although originally designed for the analysis of data flow graphs, provides useful correctness and performance capabilities for analyzing OpenMP task-based applications. FGA helps answer the following important questions:

1. What is the structure of the TDG created by the `depend` clauses?
   (a) If developers have a mental model for the graph, they can see if they have correctly expressed the constraints.
   (b) If developers do not have a mental model, the provided structure can help to develop one.
2. What is the total work (T_w) and critical path length (T_c) in the graph (to provide an ideal upper bound on scaling)?
3. Given the execution trace on a real parallel system, does the application achieve parallelism close to the predicted ideal performance?
4. If the achieved performance does not match, what is limiting performance?
   (a) What are the bottlenecks to focus on?
   (b) What patterns in the graph cause low concurrency during execution?
   (c) How do these patterns map to the task constructs in source code?

The paper is structured as follows. We discuss related work in Sect. 2 and then introduce the existing environment for the proposed FGA extension in Sect. 3. Section 4 describes how the OMPT interface was used to create the TDG for the Flow Graph Analyzer tool. We present case studies of two example applications in Sect. 5. Section 6 concludes the paper and presents opportunities for future work.

2 Related Work

Several tools are targeted at the visualization of dependences in task based programming models. Temanejo [3] is a debugger for task-parallel programming, which includes visualization of task graphs. It supports SMPs, OmpSs, StarPu, PaRSEC and OpenMP. Unlike FGA however, Temanejo is primarily oriented towards debugging and not performance optimization. Tareador includes a task graph visualizer specifically designed for OmpSs [5] with the initial goal to visualize parallelization strategies. Intel® Advisor's Flow Graph Analyzer (FGA) [1] is a tool for the design and analysis of task-based applications that use dependences and data flow graphs. We describe the features of FGA in more detail in Sect. 3.

Besides these tools that specifically target visualization of task graphs, HPC-Toolkit [2] is a collection of tools that support analysis of application performance. The tool suite mainly focuses on a timeline view of the execution of tasks and displays dependences through edges between tasks. FGA provides a graph topology canvas that highlights the graph structure and is synchronized with other views, including a timeline view. Like our proposed extensions to FGA, the HPCToolkit also uses the OMPT interface to retrieve OpenMP context information from the OpenMP implementation executing an application.

Ghane et al. [6] use the OMPT interface to detect false-sharing in multi-threaded code, to identify OpenMP regions, and to correlate events from the processors with OpenMP code. In [10], the OpenMP tools interface is applied to measure performance data of heterogeneous programs that utilize the OpenMP `target` directives to offload computation from the host to accelerator devices. While [6,10] also rely on the OMPT interfaces, our approach solely focuses on task dependencies and thus is orthogonal to these other usages of OMPT.

The work of [16] focuses on task execution performance by investigating memory-access issues as they emerge in machines that contain non-uniform memory. This is in contrast to algorithmic performance issues that arise from task dependences and that are analyzed by FGA, and relates to how tasks are mapped to the underlying machine structure once they are ready for execution after all dependences have been satisfied.

3 Prerequisites

In this section, we review the existing software environment used as the foundation of the FGA extension proposed in this paper. We first provide an introduction to OpenMP task dependences and then review the OpenMP tools interface as specified in Technical Report 6 of the OpenMP API specification. The section closes with a brief review of the capabilities of the Flow Graph Analyzer tool.

3.1 OpenMP Task Dependencies

The OpenMP API supports task-based programming since the introduction of version 3.0 [13] of the specification. Supporting many forms of irregular parallelisms, OpenMP tasks considerably extended OpenMP's capabilities for parallel programming beyond the more traditional worksharing constructs. Augmenting a block of code with the `task` construct indicates that the OpenMP implementation is free to concurrently execute that code block with some other code.

Version 4.0 [14] of the OpenMP API extended the syntax of the `task` construct by adding the `depend` clause to model dependences between a set of tasks. Task dependences can be used to locally synchronize the tasks and to avoid more expensive synchronization mechanisms. Without dependences programmers have to rely on `taskwait`, `taskgroup`, or `barrier` constructs to ensure that a subset of tasks has completed execution before another is scheduled for execution. Any

```
void task_deps() {
    int a, b, c, d;
#pragma omp task depend(out:a,b) // T1
    a = b = 0;

#pragma omp task depend(in:a) \
                 depend(out:c)   // T2
    c = computation_1(a);

#pragma omp task depend(in:b) \
                 depend(out:d)   // T3
    d = computation_2(b);

#pragma omp task depend(in:c,d)  // T4
    computation_3(c,d);
}
```

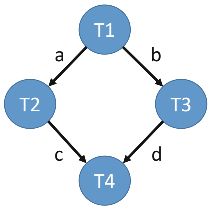


Fig. 1. Example code with OpenMP tasks and resulting task dependence graph.

of the three constructs may synchronize a too big of a task subset and thus overly limits the OpenMP implementation in its scheduling abilities.

Task dependences are based on the `depend` clause that is accepted by the `task` and `target` construct of OpenMP:

```
#pragma omp task depend(type: list-items)
```

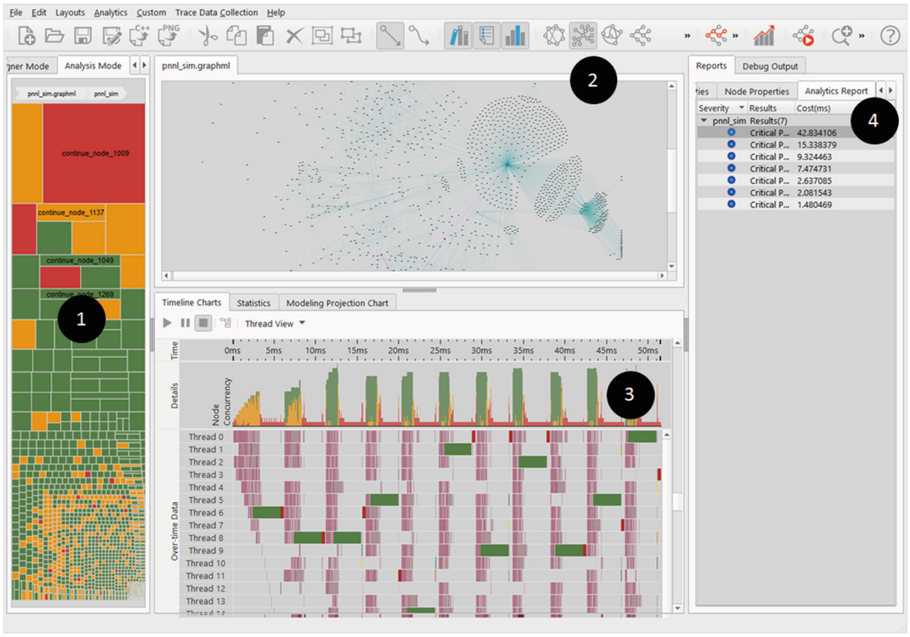
The `type` part of the clause can be one of `in`, `out`, or `inout`, with `inout` being a combination of `in` and `out`. The `list-items` are either a variable (e.g., `a`) or a array section (e.g., `a[0:99]`). If array sections are used, then the OpenMP specification requires that the array sections completely overlap to define a dependence or are disjoint if there is no dependence. An `in` dependence defines that a task's execution depends on the `out` dependence of previously a generated sibling task, if a subset of the tasks' `list-items` appear in both `depend` clauses.

Figure 1 shows a simple example of four tasks T1–T4 with flow dependences and the resulting task dependence graph (TDG). Task T1 defines an `out` dependence for `a` and `b`. T2 shares with T1 variable `a` in its list of `in` dependences; T3 depends on T1 because of variable `b`. The last task, T4, depends on the execution of T2 and T3, as it defines `in` dependences for the list items that appear in the `out depend` clause of T2 and T3.

Through the proper combination of `in` and `out` dependence types, OpenMP programmers can model flow, anti, and output dependences between tasks that are created by the program.

3.2 OpenMP Tools Interface (OMPT)

Technical Report 6 (TR6) of the OpenMP specification [15] defines a tools interface to infer the OpenMP context for performance and debugging purposes.



Severity	Results	Cost(ms)
pnnl_sim	Results(7)	
●	Critical P...	42.834106
●	Critical P...	15.338379
●	Critical P...	9.324463
●	Critical P...	7.474731
●	Critical P...	2.637085
●	Critical P...	2.081543
●	Critical P...	1.480469

Fig. 2. The Intel® Advisor – Flow Graph Analyzer GUI. (Color figure online)

An OMPT tool dynamically links into the application process at link time or execution time and then receives callbacks for OpenMP events while the application executes. The events include callbacks to monitor task execution and task dependences. We discuss our use of the OMPT interface in more detail in Sect. 4.

3.3 Overview of Intel® Advisor – Flow Graph Analyzer (FGA)

The Flow Graph Analyzer (FGA) feature in Intel® Parallel Studio XE 2018 was conceived and primarily developed to support the design, debugging, visualization, and analysis of graphs built using the Threading Building Blocks flow graph API. However, many of the capabilities of FGA are generically useful for analyzing computational graphs, regardless of their origin. In this paper, we focus on FGA's analysis capabilities only and ignore its design features.

The analysis of graphs in FGA can be reduced to the following steps, as highlighted by the numbered circles in Fig. 2: (1) inspect the tree-map view for an overview of the graph performance and use this as an index into the graph topology display, (2) adjust how the graph is displayed on screen using one of the layout algorithms, (3) examine the timeline and concurrency data for insight into performance over time, and (4) run the critical path algorithm to determine the critical path of the computation.

The tree-map view labeled as (1) in Fig. 2 provides an overview of the overall health of a graph and provides two kinds of information: the aggregate CPU time per node and the average concurrency observed while a node is executing in the graph. In the tree map, the area of each rectangle represents the total aggregate CPU time of the node and the color of each square indicates the concurrency observed during the execution of the node. The concurrency information is categorized as poor (red), ok (orange), good (green), and oversubscribed (blue). Nodes with a large area and marked as "poor" are hotspots and have an average concurrency between 0% and 25% of the hardware concurrency and are therefore good candidates for optimization. The tree-map view also serves as an index into a large graph; clicking on a square will highlight the node in the graph and selecting this highlighted node will in turn mark tasks from all instances of this node in the timeline trace view.

The graph topology canvas, labeled as (2) in Fig. 2, shows the graph structure, and supports scroll and zoom features. This view is synchronized with other views in the tool. Selecting a node in the tree-map view, the timeline, or in a data analytics report will highlight the node in the canvas. This lets users quickly relate performance data to the graph structure.

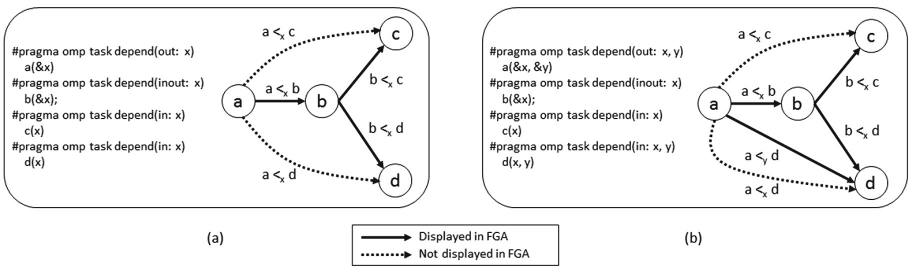
The timeline and concurrency view labeled as (3) in Fig. 2 displays the raw traces in swim lanes mapped to software threads. Using this trace information, FGA computes additional derived data such as the average concurrency of each node and the concurrency histogram over time for the graph execution. Above the per-thread swim lanes, a histogram shows how many nodes are active at that point in time. This view lets users quickly identify time regions with low concurrency. Clicking on nodes in the timelines during these regions of low concurrency, lets developers find the structures in their graph that lead to serial bottlenecks or limited concurrency.

One of the most important analytic reports provided by FGA is the list of critical paths in a graph. This feature is particularly useful when one has to analyze a large and complex graph. Computing the critical paths results in a list of nodes that form the critical paths as shown in the region labeled (4) in Fig. 2. An upper bound on speedup can be quickly computed by dividing the aggregate total time spent by all nodes by the time spent on the longest critical path, T_w/T_c. This upper bound can be used to set expectations on the potential speedup for an application expressed as a graph.

4 Tracing and Visualizing OpenMP Task Dependences

To gather information about task execution for FGA we use the OMPT interface to intercept two events. The event `ompt_callback_task_schedule` monitors the begin and end timestamps for each executed OpenMP task and the event `ompt_callback_task_dependences` records the storage location of items listed in the `depend` clauses of the generated tasks.

These events are then used to create the input for FGA that consists of two parts: (1) a graph to display and (2) the trace of task executions. The graph we build represents the partial order imposed by the `depend` clauses for the set of OpenMP tasks executed by the application, with the nodes representing



```
#pragma omp task depend(out: x)
  a(&x)
#pragma omp task depend(inout: x)
  b(&x);
#pragma omp task depend(in: x)
  c(x)
#pragma omp task depend(in: x)
  d(x)
```

```
#pragma omp task depend(out: x, y)
  a(&x, &y)
#pragma omp task depend(inout: x)
  b(&x);
#pragma omp task depend(in: x)
  c(x)
#pragma omp task depend(in: x, y)
  d(x, y)
```

Fig. 3. Examples of included and removed edges in the TDG.

OpenMP tasks and the edges representing the partial order. To reduce complexity of the graph, we omit some transitive dependences (see Fig. 3). In the figure, we denote that a node a must execute before a node b in the partial order due to a dependence on location x as $a <_x b$.

Figure 3(a) shows an example that only includes dependences due to a single location x. Because $a <_x b$ and $b <_x d$, we remove the transitive edge $a <_x d$. Figure 3(b) shows a case where two locations are involved in determining the partial order, x and y. In this case there are two potential dependence edges from a to d: $a <_x d$ and $a <_y d$. We include an edge from a to d since a is the direct source of y for d. It should be noted that if there are parallel edges between two nodes and at least one of them can be omitted due to transitivity, they all can be omitted without changing the partial order. Even so, we include edges like $a <_y d$ in the graph-topology because we believe including edges to satisfy all required data dependences is the most natural representation.

The main components of the FGA display described in Sect. 3.3 include the tree-map view, the graph-topology canvas, the timeline and concurrency-histogram view, and the critical-path report. The OpenMP task traces map naturally to these views. The tree-map view shows the time spent in each OpenMP task, colored according to the average application concurrency during the time it was executing. The graph topology canvas shows the partial ordering of the tasks, as described in the previous section. The timeline and concurrency histogram view show the execution of each task on the OpenMP runtime threads and the application concurrency over time. And the critical path report shows the most time consuming path from each source to each sink in the graph, sorted with the longest critical path at the top.

5 Case Studies

This section demonstrates the visualization and analysis of two application kernels with task dependencies. We use the *Synch_p2p* kernel from the Parallel Research Kernels [4] and the *Cholesky* application from the KaStORS benchmarks suite [17]. We run the applications on an Intel® Xeon® 6140 processor

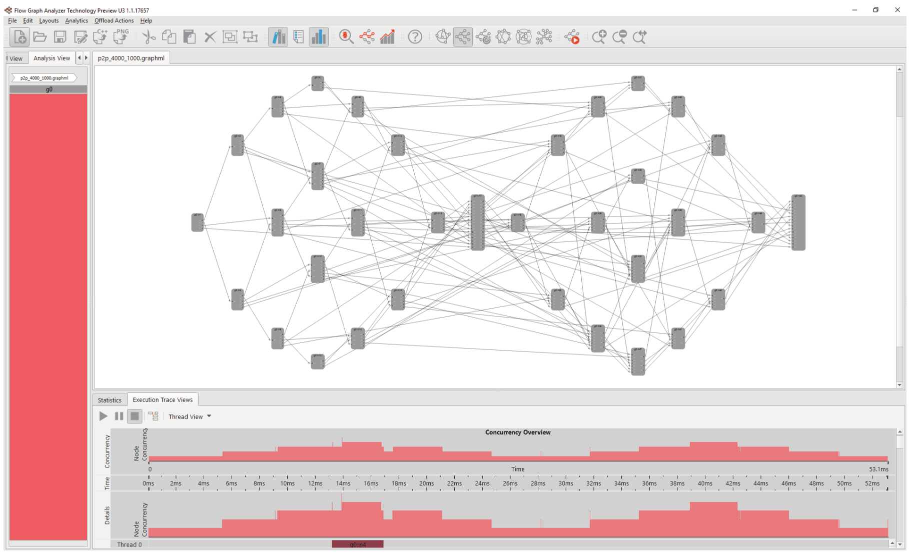


Fig. 4. TDG of the *Synch_p2p* kernel in FGA. (Color figure online)

with microcode security patches for Spectre/Meltdown. The software environment consists of Linux* SLES12 4.4 and Intel® Composer XE for C/C++, version 18.0.0.128.

We perform the following analysis steps with both applications: (1) we use the concurrency histogram and tree map to determine the degree of parallelism achieved, and the timeline view to see allocation of tasks to threads. (2) we analyze the critical path in the TDG to find the lower bound of execution time and (3) we inspect the TDG to find transitive dependences and task execution issues.

5.1 Synch_p2p

We ran the *Synch_p2p* kernel using the settings `KMP_HW_SUBSET=1s,18c,1t` and `KMP_AFFINITY=granularity=fine,compact`; the matrix size was set to 4000 and the tile size to 1000. Figure 4 shows the resulting TDG[1], while the corresponding code is shown in Fig. 5.

The tree map and concurrency histogram shown in red color in Fig. 4 indicates that with this tile and matrix size, we are able to exploit only a limited degree of parallelism. The critical path of the task graph is highlighted in pink in Fig. 6. The latency for the critical path is computed by FGA to be 8.14 msec and the aggregate CPU time for all of the tasks is 18.72 msec. Evaluating T_w/T_c

[1] For improved contrast and readability of the dependence graphs, we have modified the FGA tool to use white background instead of the standard gray color for this paper.

```
pipeline_time = prk_wtime();
int lic = (m/mc-1) * mc + 1;
int ljc = (n/nc-1) * nc + 1;
for (int iter = 0; iter<=iterations; iter++) {
    for (int i=1; i<m; i+=mc) {
        for (int j=1; j<n; j+=nc) {
#pragma omp task depend(in:grid[0],grid[(i-mc)*n+j],\
                                grid[i*n+(j-nc)],\
                                grid[(i-mc)*n+(j-nc)]) \
                 depend(out:grid[i*n+j])
            sweep_tile(i, MIN(m,i+mc), j, \
                           MIN(n,j+nc), n, grid);
        }
    }
#pragma omp task depend(in:grid[(lic-1)*n+(ljc)])\
                 depend(out:grid[0])
    grid[0*n+0] = -grid[(m-1)*n+(n-1)];
}
#pragma omp taskwait
pipeline_time = prk_wtime() - pipeline_time;
```

Fig. 5. OpenMP task dependencies in *Synch_p2p*.

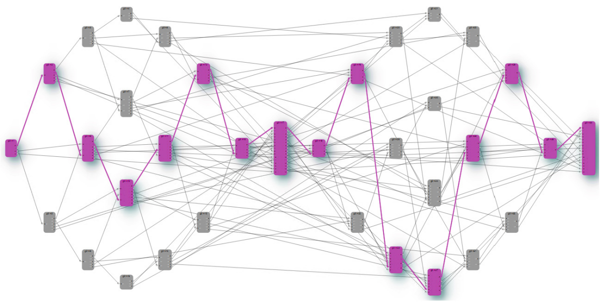


Fig. 6. Critical Path of the *Synch_p2p* TDG. (Color figure online)

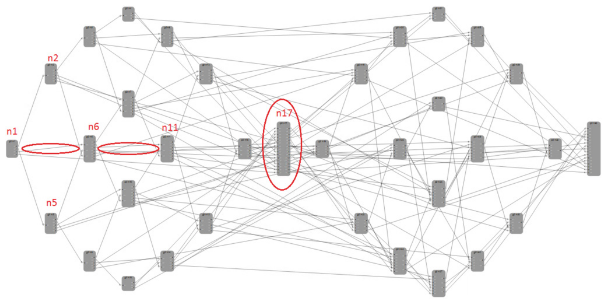


Fig. 7. Transitive edges and barrier node in the TDG of *Synch_p2p*. (Color figure online)

```
pipeline_time = prk_wtime();
for (int iter = 0; iter<=iterations; iter++) {
    for (int i=1; i<m; i+=mc) {
        for (int j=1; j<n; j+=nc) {
#pragma omp task depend(in:grid[0],grid[(i-mc)*n+j],\
                             grid[i*n+(j-nc)]),\
                  depend(out:grid[i*n+j])
              sweep_tile(i, MIN(m,i+mc), j, \
                             MIN(n,j+nc), n, grid);
        }
    }
#pragma omp taskwait
    grid[0*n+0] = -grid[(m-1)*n+(n-1)];
}
pipeline_time = prk_wtime() - pipeline_time;
```

Fig. 8. Optimized *Synch_p2p* code.

confirms that the speedup is limited to at most 2.30x for this graph. To introduce more parallelism, we reduced the tile size to 500, shrinking the critical path to 4.1 msec and increasing the potential speedup to 4.34x. Repeating this for different tile sizes, we found the best performance with a tile size 100 for a matrix size of 4000.

A closer investigation of the task graph reveals the following observations (see Fig. 7) in the structure of the graph. First, all nodes left of *n17* (red circle in Fig. 7) have a dependence to *n17*, which in turn also has dependences to all nodes to the right. Thus, this node effectively constitutes a barrier between tasks on the left side and right side of the task graph. As in this example all tasks of the kernel are sibling tasks with the same parent task, a more efficient approach is to use a `taskwait` construct instead of the barrier task and execute the code of the former task in its parent task. This saves the OpenMP implementation from the bookkeeping overhead to keep track of the task dependences to and from the barrier node.

Second, there are transitive dependences marked with a blue ellipses in Fig. 7, e.g., from node *n1* to `n6`. These edges correspond to the type of dependences highlighted in Fig. 3(b). They do not affect the partial execution order of individual tasks, but incur overhead due to the registration of the dependence with the OpenMP implementation's internal task-execution state. In the example, node *n6* has to be scheduled after *n5*, which has to execute after *n1*. We can remove these transitive dependences by specializing the `task` directives of Fig. 5. We should note that while our kernel retains the same partial ordering of tasks, the source code may be less readable now since we have elided some (redundant) data dependences.

The resulting code with the optimizations applied is shown in Fig. 8. We have removed the `task` construct in line 15 (see Fig. 5) and moved the `taskwait`

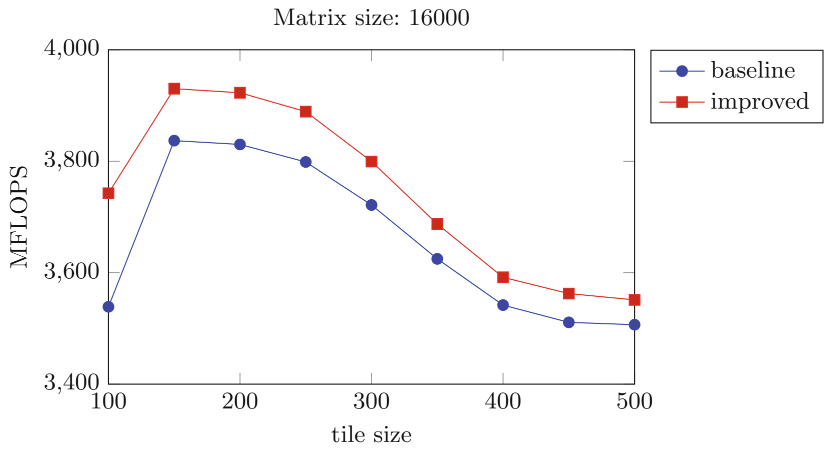


Fig. 9. Performance gain for *Synch_p2p* after the optimization.

construct from line 19 to 15. We also removed the transitive dependence by removing the corresponding `depend` clause if it generates such transitive dependences. Figure 9 shows an improved performance of about 100 MFLOPS for different matrix sizes as a result of these optimizations.

5.2 Cholesky

We ran the *Cholesky* kernel on the same machine used for the previous case study with the settings `KMP_HW_SUBSET=1s,18c,1t`; the matrix size was set to 2048 with a block size of 512. The kernel has been highly optimized by the benchmark authors by carefully adding task dependences to yield a better performance than tasking that uses coarser-grained synchronization. Even so, we traced the application and loaded the trace into FGA as shown in Fig. 10.

Again, we started with a large tile size and so the concurrency histogram and tree-map view shown in the Fig. 10 shows limited concurrency. Iteratively we decreased the tile size to introduce more parallelism until we arrived at a best tile size of 128, which provided a 4.8x improvement in the GFLOPS rate over the initial block size of 512. As in the previous case study, we inspected the TDG for redundant or transitive edges, but in this case were not able to detect any patterns that would unnecessarily inhibit performance.

While we were not able to improve the performance of the *Cholesky* kernel using the FGA tool, we were able to confirm that the dependence structure was properly implemented and we were able to arrive at a tile size that did not limit performance by restricting parallelism.

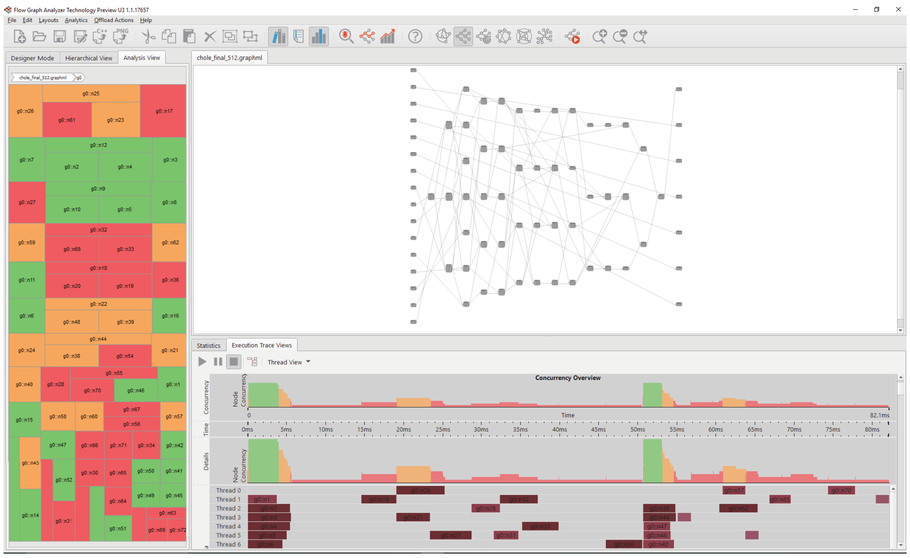


Fig. 10. Trace data of *Cholesky* in FGA.

6 Conclusion and Future Work

In this paper, we have shown how the OMPT interface of TR6 of the OpenMP API specification can be used to create a task dependency graph for the FGA tool. With the `depend` clause, programmers are able to impose an ordering on the execution of OpenMP tasks. While this provides a more fine-grained task synchronization mechanism, it gives rise to potential issues that may harm parallel execution and thus parallel performance. We used this graph as the foundation of an analysis of two sample applications and were able to expose several performance-related tasking issues.

When removing transitive dependences, we found that the set of `depend` clauses for the `task` construct may need to be specialized for some combinations of values of loop counters or other conditions. TR6 of the OpenMP API specification already defines multi-dependences through the *iteration-definition* that can be added to a `depend` clause. A natural extension would be an `if` condition that ignores a `depend` clause if the `if` condition evaluates to *false*. We are planning to bring a corresponding proposal forward with the OpenMP language committee.

There are a number of directions we plan to investigate for the FGA support for OpenMP tasks. FGA could categorize edges through coloring or labels to communicate dependence types and storage locations. The FGA tool could automatically expose transitive `depend` clauses to help remove these additional dependences that may incur bookkeeping overheads in the OpenMP implementation without providing new constraints on the partial order of the task execution. The tool could also help to identify opportunities when a `taskwait` directive may be more efficient than using a barrier task with many dependencies to previously generated tasks. We intend to show relation between edges and task depend clauses or variables in the OpenMP source code. We also plan to make the display and analysis of very large graphs simpler by automatically creating subgraphs that capture iterations or repeating patterns, collapsing these to create a smaller number of nodes at the top-level of the display.

Acknowledgments. We would like to thank Hansang Bae, Johnny Peyton, and Terry Wilmarth for helping us better understand OMPT, and providing useful suggestions.

References

1. Tovinkere, V., Voss, M.: Flow graph designer: a tool for designing and analyzing Intel® threading building blocks flow graphs. In: 2014 43rd International Conference on Parallel Processing Workshops (ICCPW), pp. 149–158. IEEE (2014)
2. Adhianto, L., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurrency Comput.: Practice Exp. **22**(6), 685–701 (2010)
3. Brinkmann, S., Gracia, J., Niethammer, C., Keller, R.: TEMANEJO - a debugger for task based parallel programming models. In: Proceeding of the International Conference on Parallel Computing, ParCo2011 (2011)
4. Van der Wijngaart, R.F., Mattson, T.G.: The parallel research kernels. In: Proceedings of the 2014 IEEE High Performance Extreme Computing Conference, Waltham, MA, pp. 1–6, September 2014
5. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
6. Ghane, M., Malik, A.M., Chapman, B., Qawasmeh, A.: False sharing detection in OpenMP applications using OMPT API. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 102–114. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24595-9_8

7. Intel Corporation. Intel Threading Building Blocks. http://www.threadingbuildingblocks.org/
8. Khronos Group. OpenCL Overview. https://www.khronos.org/opencl
9. Khronos Group. OpenVX Overview. https://www.khronos.org/openvx
10. Llort, G., et al.: The Secrets of the accelerators unveiled: tracing heterogeneous executions through OMPT. In: OpenMP: Memory. Devices, and Tasks - 12th Proceedings of the international Workshop on OpenMP, Nara, Japan, pp. 217–236 (2016)
11. Microsoft. Asynchronous Agents. https://msdn.microsoft.com/en-us/library/dd551463.aspx
12. Microsoft. Microsoft TPL Dataflow. https://www.nuget.org/packages/Microsoft.Tpl.Dataflow
13. OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 3.0, May 2008. http://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf
14. OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 4.0, July 2013. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
15. OpenMP Architecture Review Board. OpenMP Technical Report 6: Version 5.0 Preview 2, November 2017. http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf
16. Schmidl, D., Müller, M.S.: NUMA-aware task performance analysis. In: OpenMP: Memory. Devices, and Tasks - 12th Proceedings of the International Workshop on OpenMP, Nara, Japan, pp. 77–88 (2016)
17. Virouleau, P., et al.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_2

A Semantics-Driven Approach to Improving DataRaceBench's OpenMP Standard Coverage

Chunhua Liao(✉), Pei-Hung Lin, Markus Schordan, and Ian Karlin

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{liao6,lin32,schordan1,karlin1}@llnl.gov

Abstract. DataRaceBench is a benchmark suite designed to systematically and quantitatively evaluate the effectiveness of data race detection tools. Its initial release in 2017 contained 72 C99 microbenchmarks with and without data races and was successfully used to evaluate several popular data race detection tools.

In this paper, we describe a novel semantics-driven approach to improving DataRaceBench's OpenMP standard coverage. Based on a traditional definition of data races, we define several semantic categories for parallelism, data-sharing attributes, and synchronization. This allows us to assign semantic labels to constructs, clauses and data-sharing rules in the OpenMP 4.5 specification. Based on these labels we then analyze the coverage of the initial release of DataRaceBench and add 44 new C and C++ microbenchmarks to improve the OpenMP standard coverage. Finally, we re-evaluate two popular data race detection tools with the new microbenchmarks, and show that the new version of DataRaceBench gives new insights about the selected tools.

1 Introduction

Benchmarks are widely used in many research communities to measure and assess research and development results in a common, reproducible and systematic way. Good benchmarks help a community clarify problems to be solved, build common evaluation metrics, guide future development, and foster collaborations. For example, the SPEC (Standard Performance Evaluation Corporation) [1] and LINPACK [8] play important roles in the high performance computing (HPC) community for performance improvements.

In the HPC community, data race bugs are notoriously damaging while extremely difficult to detect. We have developed a dedicated OpenMP benchmark suite, DataRaceBench [9], to help systematically and quantitatively evaluate data race detection tools for their strengths and limitations. The initial release in 2017, version 1.0.1 of DataRaceBench, included a set of OpenMP microbenchmarks with and without data races. It contained 72 C99 microbenchmarks and was used to generate detailed accuracy reports for four popular data race detection tools [2,3,5,6].

B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 189–202, 2018.
https://doi.org/10.1007/978-3-319-98521-3_13

In this paper, we present a novel semantics-driven approach to analyzing and improving the OpenMP standard coverage of DataRaceBench by examining semantics of a data race. This process involves categorizing semantic categories related to data races, identifying and labeling OpenMP constructs, clauses and data-sharing attribute rules related to these semantic categories, analyzing coverage of existing microbenchmarks with respect to the semantic labels, and finally adding new microbenchmarks to improve coverage. Using this approach, we have added 44 new C and C++ microbenchmarks to DataRaceBench v1.2.0. We used the new version of DataRaceBench to re-evaluate two popular data race detection tools and discovered new insights.

The remainder of this paper is organized as follows. Section 2 gives an overview of the original DataRaceBench. Section 3 describes semantic analysis of data races and how we generate semantic labels for the OpenMP 4.5 specification. Coverage analysis and improvements are described in Sect. 4. Section 5 shows evaluation results. Section 6 presents the conclusion and future work.

2 Original DataRaceBench

DataRaceBench is a dedicated OpenMP benchmark suite to evaluate data race detection tools. The goal of this benchmark suite is two-fold: (1) to capture the requirements related to data race detection in OpenMP programs, and (2) to assess the status of current data race detection tools.

As shown in Fig. 1, the initial release (v.1.0.1) of DataRaceBench contains 72 microbenchmarks written in C99. There are 40 microbenchmarks with known data races. They are called race-yes programs. The other 32 microbenchmarks are called race-no programs which are data race free. To enable scalable experiments, some race-yes programs use C99 variable-length arrays to allow user-specified input sizes as command line options.

Two scripts are also provided to run the benchmark suite and generate reports.

Several design guidelines are followed when creating microbenchmarks for DataRaceBench. The guidelines include:

- Each microbenchmark should be as small as possible to represent a pattern with and without data race. For example, there are programs demonstrating the use of one or more OpenMP constructs or a common parallel computing pattern (for example, reduction, stencil, indirect array accesses, etc.).
- Each microbenchmark program has a main function to support dynamic data race detection.
- We pair up race-yes programs with race-no programs, when possible.
- If possible, a race-yes program should only contain a single pair of source locations that cause data races. For static tools, this is used to check if they can catch the right number of location pairs causing data races. For dynamic tools, we can check if the tool consolidates multiple runtime data races caused by the same pair of source code locations, into one data race.

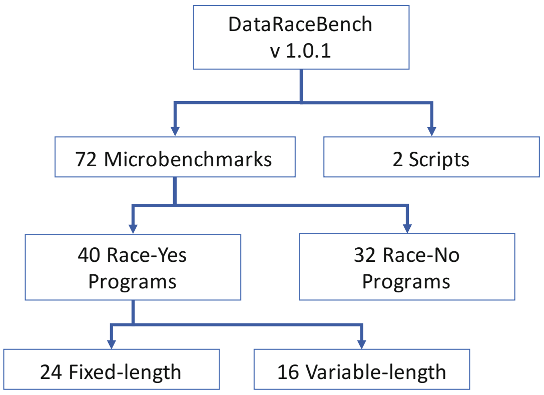


Fig. 1. Overview of initially released DataRaceBench Version 1.0.1

Figures 2 and 3 show a pair of race-yes and race-no programs included in DataRaceBench. The first program has a pair of source code locations (two references to variable **x** at line 5) which will trigger data races. The reason is that there is loop-carried output dependence caused by the writes to the shared variable **x** within a parallel region. The second program fixes the data race bug by introducing a data-sharing clause, **lastprivate**, to make the accesses to x private within the region and copy its local value within the last iteration to its corresponding original variable after the end of the region.

```
//...
int i,x;
#pragma omp parallel for
for (i=0;i<100;i++)
{   x=i;   }
printf("x=%d",x);
```

Fig. 2. Race-yes example

```
//...
int i,x;
#pragma omp parallel for lastprivate(x)
for (i=0;i<100;i++)
{   x=i;   }
printf("x=%d",x);
```

Fig. 3. Race-no example

Using a data race detection tool to analyze a microbenchmark of DataRaceBench will generate several possible results. If the analysis tool detects an existing data race it is called a *true-positive (TP)*. If the tool reports a data race for a given program, but de-facto the data race does not exist, it is called a *false-positive (FP)* analysis result. Similarly, we can have *true-negative (TN)* and *false-negative (FN)* results. With the numbers of positives and negatives reported by the tool, several standard metrics, including *precision (P)*, *recall (R)* and accuracy (A), can be calculated. They are defined as follows: $P = TP/(TP+FP)$, $R = TP/(TP+FN)$, and $A = (TP+TN)/(TP+FP+TN+FN)$. More details of DataRaceBench can be found in a previous paper [9].

3 Semantic Analysis of Data Races

In order to discover what should be included in DataRaceBench, we study the semantics of a traditional definition [13] of data races, i.e., "A *data race* can occur when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous." Based on this definition, the occurrence of a data race depends on satisfying conditions related to at least five kinds of semantics: parallel (or concurrent), shared, variable, read/write access, and synchronization. As a preliminary study, we only focus on parallel, shared and synchronization semantics and examine the relevant C/C++ OpenMP constructs, clauses, and data-sharing attribute rules defined in the latest OpenMP 4.5 specification.

3.1 Parallel Semantics

We define the parallel semantics as the information indicating if a code region will be executed concurrently or not. Based on this definition, we categorize 26 directives (including their combined variants) specified in OpenMP 4.5 into this semantic category. They include **parallel**, **for**, **sections**, **single**, **master**, **simd**, **for simd**, **task**, **taskloop**, **taskloop simd**, **parallel for**, **parallel sections**, **target parallel**, **target teams** and so on. For example, the **sections** construct contains a set of structured blocks that are to be distributed among and executed by the threads in a team. It implies concurrent execution. Similarly, the **taskloop** construct specifies loop iterations will be executed in parallel using OpenMP tasks. Its semantics literally has the word of parallel. Yet another example is the **master** construct, which specifies a structured block that is executed by the master thread of the team. It indicates the region will not be executed concurrently, but by a single thread. Some clauses are also related to parallel semantics. They include **if**, **num_threads**, **collapse** and **num_teams**.

To facilitate coverage analysis, we assign a semantic label (SID) for each relevant directive or clause. The directives related to parallel semantics are labeled as PD01 through PD26. The clauses are labeled as PC01 through PC04.

3.2 Shared Semantics

We define shared semantics as any information describing if a variable is visible and accessible by multiple threads or not. OpenMP 4.5 uses an entire subsection (Sec. 2.15) to describe its data environment, including data-sharing attribute rules and clauses (Sec. 2.15.1). The high-level logic flow of the subsection is shown in Fig. 4. The decision about a variable's data-sharing attribute starts with a question (D1) about if a variable is referenced in some eligible OpenMP regions (dynamic instances of OpenMP code blocks) including **target**, **teams**, **parallel**, **simd**, task generating (**task**, **taskloop**) and worksharing (**for**, **sections**, **single**, and **workshare**). Only a variable referenced in some regions is interesting and checked against the second question (D2): Is the variable referenced in a construct (the lexical extent of an executable directive[1])? If the answer is no, a set of not-in-construct rules apply (defined in Sec. 2.15.1.2 in OpenMP 4.5). If yes, three types of rules apply (defined in Sec. 2.15.1.1 in OpenMP 4.5): predetermined, implicitly determined, or explicitly determined.

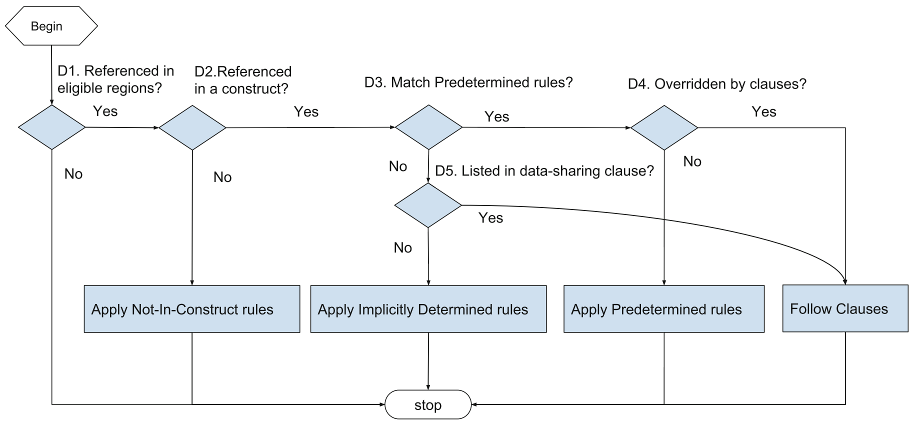

Begin
D1. Referenced in eligible regions?
D2.Referenced in a construct?
D3. Match Predetermined rules?
D4. Overridden by clauses?
D5. Listed in data-sharing clause?
Yes
No
Apply Not-In-Construct rules
Apply Implicitly Determined rules
Apply Predetermined rules
Follow Clauses
stop

Fig. 4. Flowchart of the data-sharing attribute rules

Rules for Not Referenced in a Construct. OpenMP 4.5 uses six sentences to describe when a variable is not referenced in a construct. We label them as NIC1 through NIC6 based on the order the sentences appear in the specification. Since the order of the rules in the specification is rather ad-hoc, we reorganize them as follows:

- Declared inside the called routine
  - NIC1: if the variable uses static storage, it is shared
  - NIC6: otherwise, it is private
- File-scope or namespace-scope variable
  - NIC2.1: threadprivate if the variable is in a threadprivate directive
  - NIC2.2: shared otherwise
- Function arguments in C++
  - NIC5.1: same as actual arguments if passed by reference
  - NIC5.2: private if passed by values (not explicitly listed in OpenMP)
- Dynamic storage:
  - NIC3 - objects with dynamic storage duration are shared
- Static data members
  - NIC4.1: threadprivate if within a threadprivate directive

[1] In OpenMP, an executable directive is a directive that is not declarative. It may be placed in an executable context.

- NIC4.2: shared otherwise

We split NIC2 into two sub rules (NIC2.1 and NIC2.2) since the original sentence checks a condition and leads to two different data-sharing attributes. For coverage analysis, it is better to have separated rules for different data-sharing attributes. Similarly, NIC4 is split into NIC4.1 and NIC4.2. NIC5.1 only states what happens when function arguments use pass-by-reference. We think NIC6 does not really cover function arguments passed by values, since a function argument is different from a variable declared inside a function body. We added NIC5.2 to indicate a function argument passed by value should be private to be consistent with other rules.

Rules for Predetermined Attributes. The rules for predetermined attributes (prefix PDT) are summarized below. As with the NIC rules we perform similar rule re-organization and splitting. For example, the original PDT5 rule is related to a loop iteration variable associated with for-loops of four types of constructs. We split it into four rules: one for each construct.

- Declared in a scope inside the construct
  - PDT2: private if the variable has an automatic storage duration
  - PDT8: shared if the variable has an static storage duration
- Declared in a scope outside of the construct
  - PDT1: threadprivate if within a threadprivate directive
- Dynamic storage: PDT3 - shared if the variable has a dynamic storage duration
- Static data member: PDT4 - shared if the variable is a static data member
- If loop iteration variables are in question:
  - PDT5.1: private if in the associated for-loops of a for construct
  - PDT5.2: private if in the associated for-loops of a parallel for construct
  - PDT5.3: private if in the associated for-loops of a taskloop construct
  - PDT5.4: private if in the associated for-loops of a distribute construct
  - PDT6: linear if the loop is the only loop associated with the SIMD construct
  - PDT7: lastprivate if there are multiple loops associated with the SIMD construct
- Array section: PDT9 - firstprivate if the variable is an array section mapped within a target construct, and derived from a variable of a pointer type.

Note that unlike many NIC rules stating two choices for a condition (e.g. NIC1 and NIC6, NIC2.1 and NIC2.2), most PDT rules (e.g. PDT1, PDT5.1 through 5.4, PDT6, etc.) only state what will happen when certain conditions are met. When these conditions are not met, the decision will be deferred to a later stage using either implicitly determined rules or explicit data-sharing clauses.

Implicitly Determined Rules. We label the seven sentences for implicitly determined rules with IDs and re-organized them as follows:

- Default clause: IDT1 - for variables in a parallel, teams, task generating constructs, follow the default clause if it is present.
- In a Parallel construct: IDT2 - the variables are shared if no default clause is present.
- In a Target construct:
  - IDT4.1: variables that are not mapped are firstprivate.
  - IDT4.2: variables that are mapped, follow data-mapping attribute rules and clauses.
- Task generating construct:
  - IDT5: In an orphaned task generating construct, formal arguments passed by reference are firstprivate.
  - IDT6: A variable is shared when it is in a task generating construct without a default clause, its data sharing attribute is not determined by the above rules, and the same variable in the enclosing context is determined to be shared by all implicit tasks bound to the current team.
  - IDT7: In a task generating construct, a variable without applicable rules above is firstprivate.
- Others: IDT3 - In constructs other than task generating or target constructs (e.g. teams, simd and worksharing), these variables reference the variables with the same names that exist in the enclosing context, if no default clause is present.

Explicit Data-Sharing Clauses. Finally, there are seven clauses indicating data-sharing attributes, including **default**, **shared**, **private**, **firstprivate**, **lastprivate**, **reduction** and **linear**. We categorize them into a DSC (data-sharing clause) set (DSC01 through DSC07).

3.3 Synchronization Semantics

We define synchronization semantics as any information deciding if there is any synchronization mechanism to prevent the shared accesses to a variable from being simultaneous or not. We categorize the following OpenMP directives and clauses as relevant to synchronization, including **nowait**, **critical**, **barrier**, **taskwait**, **taskgroup**, **atomic**, **flush**, **ordered** (both clause and directive) and **depend**. They are labeled as N01 through N10. N00 is reserved to indicate that no explicit synchronization is specified.

4 Coverage Analysis and Improvements

For each semantic label, if there is a microbenchmark using the corresponding construct, clause or rule, we claim that the label is covered in our coverage analysis. For example, a microbenchmark shown in Fig. 2 covers PD12 (**parallel for**), PDT5.1 (predetermined to be private for an associated loop iteration variable) and N00 (no explicit synchronization is specified).

4.1 Analysis Methods

Some coverage information can be obtained by checking if some OpenMP keywords (such as **collapse**, **depend**, and **taskgroup**) are used in our benchmark suite. This gives us an overview of which semantic labels are covered and which are missing.

To recognize complex code patterns beyond keywords, we built a simple source analysis tool, namely CoverageAnalyzer, using the ROSE source-to-source compiler framework [7,10]. CoverageAnalyzer parses source files into Abstract Syntax Trees and finds code patterns satisfying conditions defined in data-sharing attribute rules. For example, to check if PDT8 is covered, CoverageAnalyzer tries to find all OpenMP regions first, then searches each region for locally declared variables. If the variable is not declared static, we find a match to the conditions corresponding to PDT8 and conclude that PDT8 is covered.

Sometimes we got lucky and did not have to implement condition search for all rules in CoverageAnalyzer. For example, all NIC rules require a code pattern in which a variable is referenced within an OpenMP region, but not within an OpenMP construct. This can only happen through a function call. CoverageAnalyzer finds that none of the existing programs in v1.0.1 has an OpenMP region in which a function call to user-defined functions is made. So we can safely conclude that none of NIC rules are covered.

4.2 Analysis Results

The coverage of semantic labels in each semantic category is summarized in Table 1. In the parallel category, missed constructs include **master**, **taskloop**, **teams** and their applicable combined directives. Within the shared semantic category, NIC rules have zero coverage while two data-sharing clauses (**default** and **linear**) in DSC are not covered. For PDT and IDT, uncovered rules include those involving static variables, **threadprivate**, **collapse**, **taskloop**, **distribute**, multiple loops associated with SIMD, orphaned task constructs using formal arguments passed by reference and so on. For synchronization semantics, only two out of ten relevant clauses are covered (**nowait** and **depend**).

Table 1. Coverage analysis result for v1.0.1 of DataRaceBench

	Parallel	Shared				Sync.
		NIC	PDT	IDT	DSC	
Semantic label count	30	9	12	8	7	10
Covered labels	PD1–4, 6, 8, 11, 12, 14, 15, PC02		2, 3, 5.1, 5.2, 6	2, 3, 4.1, 6	2–6	1, 10
Covered label count	11	0	5	4	5	2
Coverage ratio	36.67%	0.0%	41.67%	50.0%	71.43%	20.0%

4.3 Improving Coverage

Based on the coverage analysis results, we added 44 new microbenchmarks into DataRaceBench Version 1.2.0[2] to cover the missed semantic labels. For simplicity, we treat some combined constructs (e.g. **target simd**) as covered if their individual constructs are covered by existing microbenchmarks. For example, Fig. 5 shows a new microbenchmark program to cover NIC4.1 and NIC4.2. In the case of not referenced within a construct, a static data member should be shared, unless it is within a **threadprivate** directive. Figure 6 covers both **ordered** clause and directive. **ordered(2)**, an OpenMP 4.5 addition, also associates two loops and make their loop iteration variables private. **target teams** and **taskgroup** are covered in Figs. 7 and 8 respectively.

```
class A {
public:
   static int ctr;
   static int pctr;
#pragma omp threadprivate(pctr)
};
int A::ctr=0;
int A::pctr=0;
A a;
void foo()
{
    a.ctr++;
    a.pctr++;
}
int main()
{
#pragma omp parallel
      foo();
//...
}
```

Fig. 5. Race-yes using static data members

```
#include <stdio.h>
int a[100][100];
int main()
{
   int i, j;
#pragma omp parallel for ordered(2)
   for (i = 0; i < 100; i++)
      for (j = 0; j < 100; j++)
         {
            a[i][j] = a[i][j] + 1;
#pragma omp ordered depend(sink:i-1,j) \
               depend (sink:i,j-1)
            printf ("test␣i=%d␣j=%d\n",i,j);
#pragma omp ordered depend(source)
         }
   return 0;
}
```

Fig. 6. Race-no using ordered(2)

As a result, DataRaceBench Version 1.2.0 covers all semantic labels from each semantic group. This means that the new coverage ratios are all equal to 100%, as shown in Table 2.

Table 2. Coverage analysis result for v1.2.0 of DataRaceBench

	Parallel	Shared				Sync.
		NIC	PDT	IDT	DSC	
Semantic label count	30	9	12	8	7	10
Covered labels	All	All	All	All	All	All
Covered label count	30	9	12	8	7	10
Coverage ratio	100%	100%	100%	100%	100%	100%

[2] Available at https://github.com/LLNL/dataracebench/releases.

```
// ...
  double a[len];

  /* Initialize with some values*/
  for (i=0; i<len; i++)
    a[i]= ((double)i)/2.0;

#pragma omp target map(tofrom: a[0:len])
#pragma omp teams num_teams(2)
  {
    a[50]*=2.0;
  }
```

Fig. 7. Race-yes using target+teams

```
  int result = 0;
#pragma omp parallel
#pragma omp single
  {
#pragma omp taskgroup
#pragma omp task
      {
        sleep(3); result = 1;
      }
#pragma omp task
     result = 2;
  }
  assert (result==2);
```

Fig. 8. Race-no using taskgroup

5 Evaluation

In order to assess if the new microbenchmarks in DataRaceBench v1.2.0 are beneficial, we use them to evaluate two popular data race detection tools, Archer and Intel Inspector. Archer [6] is an OpenMP data race detector that exploits ThreadSanitizer [5] to achieve scalable happens-before tracking. It uses static analysis to reduce false positives generated by the dynamic analysis performed by ThreadSanitizer. Intel Inspector [3] is a dynamic analysis tool that detects threading and memory errors in C, C++ and Fortran codes. It supersedes Intel's Thread Checker tool [11,12], with added memory error checking. The versions of the selected tools used are listed in Table 3, with the compilers used with these tools (either to build the tools, compile the microbenchmarks, or both).

Table 3. Data race detection tools: versions and compilers

Tool	Version	Compiler
Archer	towards_tr4 branch	Clang/LLVM 4.0.1
Intel Inspector	2018 (build 522981)	Intel Compiler 18.0.1

Intel Inspector provides different levels of analysis with varying configurations. We configure Intel Inspector to use maximal time and resources in our evaluation using the command line: `inspxe-cl -collect ti3 -knob scope=extreme -knob stack-depth=16 -knob use-maximum-resources=true`.

Our testing platform is the Quartz cluster hosted at the Livermore Computing Center [4]. Each computation node of the cluster has two Intel 18-core Xeon E5-2695 v4 processors with hyper threading support. We ran each tool 5 times for each microbenchmark using 72 threads. For each run, we use ten minutes as a timeout limit to terminate potential runtime hanging.

5.1 Experiment Results

Table 4 shows our experimental results. The first column lists the file names (each with a prefix such as DRB072 as a short ID) of all the newly added microbenchmark programs. The second column indicates if the program is known to contain a data race or not (‘Y’ or ‘N’). During multiple runs for a given program, a tool may report different numbers of data races detected. So ranges of numbers (min race - max race) are given in Column 3 and 5 of the table. For example, an entry of 0–0 means that no data race was found in any run of the respective tool. An entry of 1–3 means in all five runs at least one data race was detected. If a range such as 0–4 is reported, this means a tool generated mixed results for a given program.

Column 4 and 6 (labeled as “type”) give a verdict for a tool’s result for a given program. Based on the range numbers, the result is given as true negative (TN), false positive (FP) or mixed TN and FP for a program without known data races. Similarly, a tool’s result an be true positive (TP), false negative (FN) or mixed TP and FN for a race-yes program.

In some cases, a tool may fail due to errors during compilation or runtime steps. We mark the result as compile-time segmentation fault (CSF), unsupported feature by a compiler (CUN), runtime segmentation fault (RSF) or runtime timeout (RTO). If any error happens, we try to investigate log files to identify any valid true or false positives. Negative reports are ignored since a negative test report with errors is inconclusive. For example, a tool may trigger a runtime timeout and generate partial logs with identified data races, which should be counted. Table 5 summarizes the numbers of positive, negative and unknown (marked as not available or N/A) results based on the information in Table 4.

The results show that new benchmark programs generate new insights for the two tools. Archer did not report any false positives or false negatives in the experiments. However, 13 programs triggered the tool to have some compile-time or runtime errors. Five of these error happened because the version of Clang does not support the OpenMP 4.5 features used in DRB094, DRB095, DRB096, DRB100 and DRB112 (marked as CUN). Another five errors are compiler segmentation faults raised by a phase called InstrumentParallel, for DRB085, DRB086, DRB087, DRB091 and DRB102 (marked as CSF). Runtime segmentation faults happened for DRB097 and DRB116 (marked as RSF). A runtime timeout (RTO) happened with DRB106. The tool generated partial results with true positives for DRB106. We are actively working with the Archer developers to address these issues in their latest development branch.

In comparison, Intel Inspector reported mixed results (TN FP) for DRB096, a program using **taskloop** combined with **collapse**(2) to cover PDT 5.3. In only one out of the five runs, the tool reported a write-to-write race for loop iteration variables. The tool also generated two false positives (FP) for DRB105 and DRB107. DRB105 is a classic task implementation of Fibonacci number generation using **taskwait**. The tool reported a write-to-write data race for the line of i=fib(n-1); For DRB107 (shown in Fig. 8 using **taskgroup**), the tool

Table 4. Evaluation report (column R: whether a program contains a data race)

Microbenchmark Program	R	Data Race Detection Tools			
		Archer		Intel Inspector	
		min race - max race	type	min race - max race	type
DRB073-doall2-orig-yes.c	Y	84 - 92	TP	2 - 2	TP
DRB074-flush-orig-yes.c	Y	1 - 3	TP	1 - 1	TP
DRB075-getthreadnum-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB076-flush-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB077-single-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB078-taskdep2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB079-taskdep3-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB080-func-arg-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB081-func-arg-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB082-declared-in-func-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB083-declared-in-func-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB084-threadprivatemissing-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB085-threadprivate-orig-no.c	N	-	CSF	0 - 0	TN
DRB086-static-data-member-orig-yes.cpp	Y	-	CSF	1 - 1	TP
DRB087-static-data-member2-orig-yes.cpp	Y	-	CSF	1 - 1	TP
DRB088-dynamic-storage-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB089-dynamic-storage2-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB090-static-local-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB091-threadprivate2-orig-no.c	N	-	CSF	0 - 0	TN
DRB092-threadprivatemissing2-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB093-doall2-collapse-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB094-doall2-ordered-orig-no.c	N	-	CUN	0 - 0	RTO
DRB095-doall2-taskloop-orig-yes.c	Y	-	CUN	2 - 2	TP
DRB096-doall2-taskloop-collapse-orig-no.c	N	-	CUN	0 - 4	FP TN
DRB097-target-teams-distribute-orig-no.c	N	0 - 0	RSF	0 - 0	TN
DRB098-simd2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB099-targetparallelfor2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB100-task-reference-orig-no.cpp	N	-	CUN	0 - 0	TN
DRB101-task-value-orig-no.cpp	N	0 - 0	TN	0 - 0	TN
DRB102-copyprivate-orig-no.c	N	-	CSF	0 - 0	TN
DRB103-master-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB104-nowait-barrier-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB105-taskwait-orig-no.c	N	0 - 0	TN	3 - 4	FP
DRB106-taskwaitmissing-orig-yes.c	Y	35 - 48	RTO TP	4 - 6	TP
DRB107-taskgroup-orig-no.c	N	0 - 0	TN	1 - 1	FP
DRB108-atomic-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB109-orderedmissing-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB110-ordered-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB111-linearmissing-orig-yes.c	Y	73 - 85	TP	1 - 2	TP
DRB112-linear-orig-no.c	N	-	CUN	0 - 0	TN
DRB113-default-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB114-if-orig-yes.c	Y	42 - 48	TP	1 - 1	TP
DRB115-forsimd-orig-yes.c	Y	44 - 47	TP	1 - 1	TP
DRB116-target-teams-orig-yes.c	Y	0 - 0	RSF	1 - 1	TP

Table 5. The numbers of positive, negative and unknown results of the tools

Tool	Race:Yes				Race:No			
	TP	TP/FN	FN	N/A	TN	TN/FP	FP	N/A
Archer	15	0	0	4	17	0	0	8
Intel Inspector	19	0	0	0	21	1	2	1

reported two tasks writing to **result** causing a data race. DRB094 (shown in Fig. 6) caused a runtime timeout error (hanging) for Intel Inspector. In this program, the 2nd loop is associated with **ordered**(2) so its loop interaction variable should be private according to PDT5.1. Making j explicitly private will fix the hanging. We have reported these issues to Intel.

6 Conclusion

In this paper, we presented a semantics-driven approach to analyzing and improving DataRaceBench's coverage of the OpenMP standard. We focused on three semantic categories (parallel, shared and synchronization) and labeled a set of relevant OpenMP language constructs, clauses and rules for coverage analysis. The application of our approach resulted in adding 44 new microbenchmarks which significantly increased DataRaceBench's coverage. Finally, the new microbenchmarks were used to re-evaluate two data race detection tools: Intel Inspector and Archer. While these two tools performed almost equally well in our original evaluation [9], the new microbenchmarks reveal that Intel Inspector outperforms Archer in terms of supporting more microbenchmarks without any errors. However, there is still room for improvements for Intel Inspector when analyzing programs using **taskloop**, **taskwait** or **taskgroup**.

In addition, as an unexpected side effect of extracting semantics from the OpenMP 4.5 standard, we found a misuse of the term "construct". **declare simd** is called a construct while it is a non-executable declarative directive and an OpenMP construct must be an executable directive. We have reported this issue to the OpenMP language committee. Another discovery is that the data-sharing attribute rules in OpenMP are surprisingly difficult to understand. We had to reorganize these rules, split some of them, and made previously hidden rules explicit to extract semantic labels. We suggest to the OpenMP language committee to improve the clarity of the rules and define an official algorithm.

In the future, we plan to explore semantics related to variables and read-/write accesses. We also want to increase DataRaceBench's coverage of OpenMP runtime library routines and environment variables. In the domain of scientific computing, only a few computational patterns are covered in DataRaceBench, such as stencil and matrix multiplication. Adding more representative numerical computation patterns with and without data races may also be beneficial.

Acknowledgment. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, Lawrence Livermore National Security, LLC, via projects LDRD 17-ERD-023. IM release number LLNL-CONF-750770.

References

1. SPEC's Benchmarks (1995). http://www.spec.org/benchmarks.html
2. Helgrind (2000). http://valgrind.org/docs/manual/hg-manual.html

3. Intel Inspector 2017 (2017). https://software.intel.com/en-us/intel-inspector-xe
4. Livermore Computing Quartz system (2017). https://hpc.llnl.gov/hardware/platforms/Quartz
5. Threadsanitizer (2017). https://github.com/google/sanitizers
6. Atzeni, S., et al.: Archer: effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium, pp. 53–62. IEEE (2016)
7. Davis, K., Quinlan, D.: ROSE: an optimizing preprocessor for the object-oriented overture framework. http://www.c3.lanl.gov/ROSE/
8. Dongarra, J.J., Luszczek, P., Petitet, A.: The linpack benchmark: past, present and future. Concurr. Comput.: Pract. Exp. **15**(9), 803–820 (2003)
9. Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I.: DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 11. ACM (2017)
10. Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R.: A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 15–28. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13217-9_2
11. Petersen, P., Shah, S.: OpenMP support in the Intel® thread checker. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 1–12. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45009-2_1
12. Sack, P., Bliss, B.E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and efficient filtering for the Intel thread checker race detector. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability. pp. 34–41. ACM (2006)
13. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997). https://doi.org/10.1145/265924.265927

Tasking Evaluations

On the Impact of OpenMP Task Granularity

Thierry Gautier(✉), Christian Perez, and Jérôme Richard

Univ. Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude Bernard Lyon 1, LIP, 69007 Lyon, France

thierry.gautier@inrialpes.fr, {christian.perez,jerome.richard}@inria.fr

Abstract. Tasks are a good support for composition. During the development of a high-level component model for HPC, we have experimented to manage parallelism from components using OpenMP tasks. Since version 4-0, the standard proposes a model with dependent tasks that seems very attractive because it enables the description of dependencies between tasks generated by different components without breaking maintainability constraints such as separation of concerns. The paper presents our feedback on using OpenMP in our context. We discover that our main issues are a too coarse task granularity for our expected performance on classical OpenMP runtimes, and a harmful task throttling heuristic counter-productive for our applications. We present a completion time breakdown of task management in the Intel OpenMP runtime and propose extensions evaluated on a testbed application coming from the Gysela application in plasma physics.

Keywords: Task granularity · Reordering · Cache reuse
Component model

1 Introduction

Tasks have been incorporated in OpenMP-3.0 in November 2008. This initial model only considers *independent* tasks, such as provided by the famous Cilk [13] parallel programming environment. In July 2013, OpenMP-4.0 integrates a *dependent*-task model. This model enable computing complex schedules that favor, for instance, data reuse among tasks.

One of our main testbed application extracted from the Gysela application [17] has been parallelized using dependent tasks. Preliminary experiments have shown that a hand-coded version of the code can greatly improve performances due to a better use of caches, but at the expense of code maintainability and, also with a loss of performance portability caused by hard-coded scheduling decisions. This paper reports our mitigated experience on delegating all task scheduling concerns to the OpenMP runtime. Our issues mainly come from the required fine-grain task granularity since reusing in-cache data is expected in our application.

The algorithmic structure of the testbed application is the following: the working set is decomposed by planes, each plane is sub-divided in regions (such as line groups) where a chain of k tasks operate on it. The $k-1$ first tasks of each chain are independent, while the last task of each performs a per-plane stencil computation. Thus, tasks working on different planes are independent. Figure 4 illustrates it. At the end of each iteration, a final task operates on all regions of the same plane. Because the graph structure is quite simple, at the beginning of this work we were very confident to delegate the all task scheduling concerns to an OpenMP runtime.

Depending on the size of the working set and the hardware, only some regions or few planes could be fit into the shared cache. Two problems occur. First, the task creation iterates over all the first tasks of all chains, then over all the second tasks and so forth, which sequentially iterates several times over all the working set with causing $O(k)$ evictions. Second, the scheduling heuristics of the tested OpenMP runtimes (an Intel-based, LLVM and a GNU runtime) are not designed for constructive cache sharing. For instance, the Intel runtime relies on a work-stealing scheduler where working threads tend to have disjoint working sets. Constructive cache-sharing schedules have been studied since long time [7,10].

These two problems are strongly connected. The order of the task creation could not be easily chosen due to software engineering constraints. In our application, a high-level assembly of components [5] enforces the order and we do not want to violate the separation of concern by analyzing[1] memory access patterns arising from tasks submitted by different components. Moreover, even if we reschedule tasks in order to provide an efficient sequential execution, there is no guarantee that the OpenMP task scheduler will exploit it for constructive cache sharing.

Considering the scheduling performance guarantee as the most prominent issue, preliminary experiments of our application using OpenMP tasks enable us to locate four performance critical issues:

overhead: The task implementation has a significant overhead that limits scalability. In our case, it cannot be easily amortized by computation because of the fine granularity.

concurrency: The task creation is slowed down as the number of threads increases.

harmful heuristic: The task throttling [12] may improve performance. But a naive static heuristic is implemented on several OpenMP runtimes and it has been proved highly counter-productive. When present, the scheduler could not be clairvoyant on the future of the computation because almost all tasks are serialized.

task scheduling: Even if the task throttling is disabled, the default scheduling strategy between thread sharing cache favor, as discussed above, breadth-first execution where cores tend to have disjoint working sets.

In the case of coarse grain applications, the task creation overhead and concurrency issues are amortized by the computation [6]. The first three issues may

[1] Such analysis may be complex if made statically.

be overcome at the expense of a dedicated and optimized implementation: our experimental results with libKOMP, an extended version of the LLVM OpenMP runtime, illustrates the gains in term of performances. Nevertheless, the last issue is about scheduling where the best solution often relies on the application pattern. In this paper, we propose a two steps solution where submitted tasks are reordered cooperatively with the scheduler. This also points out one of the missing feature in OpenMP standard: the capability to specify specialized scheduling strategies for a set of tasks.

2 Background: OpenMP Tasks Management

This section deals with the LLVM OpenMP runtime [2] tag release 5.0 as currently developed by LLVM team[2]. It also compares some of the key design choices with those implemented in the GNU OpenMP libGOMP [1] coming with GCC 6 series.

2.1 Implementation of the OpenMP Task Model

The OpenMP task model enables the creation of tasks with dependencies in a simple way as sketched in the next listing.

```
#pragma omp task depend(inout: a, b) depend(out: c) depend(in: d)
    < code >
```

The encountering thread of the OpenMP task directive creates a task that could be performed asynchronously to the caller. A task execution corresponds to an execution of <`code`>. Data sharing attributes describe how the task data environment is built from the environment of the encountering thread.

The compiler and the runtime are responsible for the management of task internal data structures. For instance, the Intel and Clang compilers generate a pair of runtime calls [2] to `__kmpc_omp_task_alloc` and `__kmpc_omp_task`, for independent tasks, or `__kmpc_omp_task_with_deps` if the task directive includes depend clauses. The previous listing is translated to the following pattern (missing parameters are not important here) where two main function calls are marked in bold:

```
kmp_int32 outlined_function(kmp_int32 gtid, void* taskdata)
{  ... < code > ... }

kmp_tasking_flags_t flag = ...;
kmp_task_t* newtask = __kmpc_omp_task_alloc( ..., ..., &flags,
    size_of_task, size_of_shared, outlined_function, ...);
kmp_depend_info_t dep_list[4] = {..., ..., ..., ...};
__kmpc_omp_task_with_deps( ..., ...,  newtask, 4, dep_list, ..., ...);
```

The GNU compiler and libGOMP runtime merge these two calls [1] at the expense of recopying parts of the task data generated by the compiler on the C stack:

[2] https://openmp.llvm.org, http://llvm.org/git/openmp.git. The LLVM runtime has been forked from Intel public source and it is fully compatible with GCC, ICC and Clang compilers.

```
void outlined_function(void* taskdata)
{ ... }

<opaque type> taskdata = { .... };
void* dep_list[5] = { 4, 3, a, b, c, d }
GOMP_task( outlined_function, taskdata, fcopy_taskdata, datasize,
    ..., ..., flags, dep_list, ...);
```

Many other OpenMP runtimes follow the same approach: the compiler generates the code of the outlined function with correct copies or data sharing (according to the specified data sharing rules). Then, the runtime allocates an internal task descriptor, copies the fields, computes the dependencies and then pushes the task to various scheduling queue(s). The next section focuses on this internal data structure and algorithms used to build correct dependencies. Their choices explain the observed overhead or limitations.

Table 1. Main characteristics of libGOMP and libOMP. The sizes are in bytes and the task descriptors take into account structures for managing dependencies.

	Size of task descriptors	Dependencies	Task throttling threshold	Queues	Scheduler
libGOMP		Hash table + lock per team	64× number of threads	Per task (children), task group and teams	Multiple lists scheduling
libOMP	424	Hash table + lock per dependencies	256 tasks per queue	One per thread	Work stealing

2.2 Internal Data Structures and Algorithms to Manage Dependencies

GNU libGOMP and LLVM/Intel libOMP runtimes have made very different implementation choices as summed up in Table 1. The main difference between libGOMP and libOMP comes from the locking strategy to ensure coherent computation of dependencies: in libGOMP, exclusive accesses are guaranteed by a lock associated to the team data structure, while in libOMP, there is one lock per task. This explains scalability issues of libGOMP when the task granularity is too small [21,27].

The OpenMP dependent-task model is based on defining *dependence-type* of a list of memory references in the clause `depend`. The runtime should keep track of the previous accesses made on memory regions described by the array sections of the depend clause. Up to now, the standard restricts the usage to avoid the overlap of two array sections. This make the computation of dependencies much simpler in a sequence of tasks by identifying an array section to its base array. Indeed, runtimes can only store the last dependency into an associative table to retrieve it from a pointer. The task creation consists in the following steps:

Allocation of the internal task descriptor. libGOMP relies on the `malloc` function of the C library. The LLVM and Intel runtimes implement a thread-local heap allocator.

Initialization of the task descriptor fields. Once allocated, the runtime initializes a data structure, copy the ICVs, and update a counter to detect termination.

Checking dependencies. This step consists in adding the newly created task into the list of successors from all its predecessor. In the two runtimes, the scheme is almost the same: for each pointer identifying the array section, the runtime looks into a hash table to retrieve the last dependencies of the array section.

Enqueue. If a task is detected as ready for execution, then it is enqueued into runtime queues. The LLVM and Intel runtimes push a task into a queue owned by the running thread. GNU libGOMP enqueues the task into several queues: the child queue of its parent, the queue of the task group if it exists, and the queue of the team that stores all the ready tasks.

This high-level view masks the way the Intel, LLVM and GNU runtimes manage concurrency. The steps ‘Allocation’ and ‘Initialization’ are mostly involving local updates of data structures. They do not require locking mechanisms for exclusive accesses. Checking dependencies is the most complex operation of the task creation since predecessors of a task may finish while the task is being checked. The design of GNU libGOMP is such that all modifications related to dependencies are mutually exclusive by using a global lock associated with the team. This is the main scalability problem of libGOMP. The LLVM and Intel runtimes enable more concurrency between insertions and suppressions of dependencies. To manage the modification of data structures, they use a lock per *dependency node* attached to each task. Because concurrent accesses are more frequent, the thread generating tasks is slowed down: new tasks are created and enqueued at a low throughput compared to a sequential task creation.

2.3 Task Throttling

The term ‘task throttling’ refers to all kind of heuristics [4,12]. It enables the runtime to serialize tasks in order to reduce the inherent overhead of task creation. Sophisticated strategies have been designed and experimented [12] which dynamically profiles the application tasks to produce good decisions. In the LLVM and Intel libOMP or GNU libGOMP threads throttle task creations are based on static thresholds: when there is more than `256` tasks per queue in LLVM libOMP; and when there is more than `64 * omp_get_num_threads()` pending tasks in libGOMP.

These heuristics can efficiently reduce the overhead of task creation (see next section). However, these heuristics are not well suited, and even harmful, for some classes of applications [18], such as our. There is a huge gap between these research results and heuristics found in those OpenMP runtimes. Moreover, the scheduling decision could not be adapted during runtime.

3 Performance Evaluation and Extension of the LLVM Runtime

Experiments have been made on a quad-socket server with 4 NUMA nodes. Each NUMA node holds a 24-core Intel Xeon E7-8890v4 CPU for a total of 96 cores. The goal of the experimentations is to evaluate the capacity of fine-grained OpenMP tasks to be a building block to improve reuse of data in shared caches. We restrict all our experimentation on one NUMA node with up to 24 cores.

We make use of the LLVM libOMP version from http://llvm.org/git/openmp.git, branch `release_50`. The source code of the LLVM runtime has been instrumented to precisely measure the clock cycles for basic operations for the OpenMP task management in libOMP. We use the *time stamp counter* (`rdtsc`) that is incremented at constant rate on the platform.

3.1 Completion Time Breakdown of OpenMP Tasks Management

The LLVM OpenMP runtime libOMP has been instrumented to measure the delay for each the different steps in the task creation as presented in Sect. 2.2. In order to limit the overhead, we insert calls to get the real time stamp counter a the begin and the end of each of these steps. Delays are cumulated per thread and a final summation is computed at the end of the program to avoid overhead due to concurrent update. It impacts six functions, including the initialization of finalization of the library to dump the values.

Figure 1 reports results for the BOTS [11] benchmarks with only independent tasks. Figure 2 reports results on the Jacobi and SparseLU benchmarks of the KASTORS suite [27]. They compare two versions of the same code: one with independent tasks and the second with dependent tasks.

Each measure is the average cycles per operation over 30 runs. In all figures, we present the number of cycles for the following internal operations: `alloc` is the allocation and initialization of the data fields for the internal task descriptor; `atomic` is extracted from `alloc` and refer to a piece of code that update concurrent object by atomic instruction; finally, `enqueue` is the operation of inserting the descriptor into a scheduler queue. On the benchmark with dependent tasks, `check deps` is the operation of checking and adding the dependencies between tasks and `release deps` is the operation of releasing successors of the ended tasks. The sum of all these operations captures the code between a task submission and its insertion in scheduler queues.

For the independent task benchmarks, the serialization of all submitted tasks on the case of 1-core execution shows that a task throttling heuristic can reduce the overhead of task management. The task initialization cost increases slowly as the number of cores grows: parts of the initialization make use of atomic operations for which the cost depends on the number of concurrent data accesses. The enqueue operation is stable mostly when the number of cores is greater than 1, except for Uts [18] which is a search algorithm working on very large unbalanced trees: the concurrency on each queue of libOMP is exacerbated.

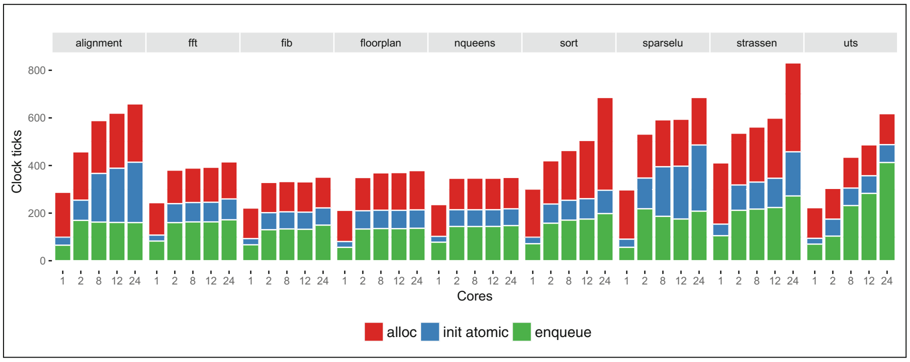


Fig. 1. Completion time breakdown of OpenMP independent tasks in libOMP on BOTS.

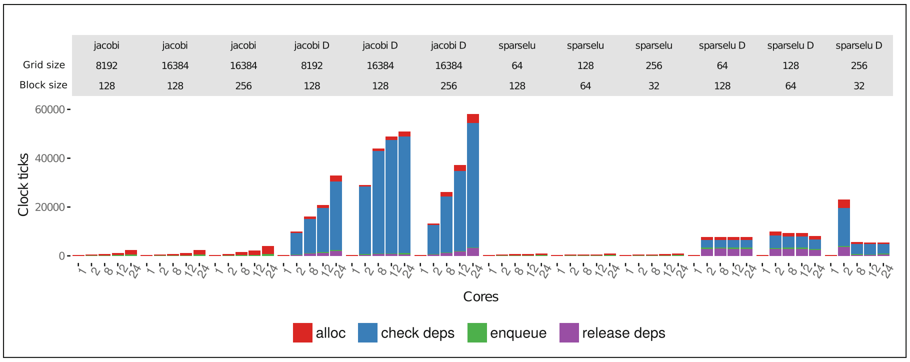


Fig. 2. Comparison of the completion time breakdown between OpenMP-3.0 tasks and OpenMP-4.0 dependent tasks on the Jacobi and SparseLU benchmark from the KASTORS. The suffix 'D' denotes the dependent task version of the code.

For the KASTORS benchmark, except for Jacobi, the global behavior is similar to SparseLU in Fig. 2. On average, the cost of task creation is about 10 times bigger than for independent task. Most of the cost comes from checking dependencies. Next comes the release of dependencies to activate successors when tasks are finished.

Jacobi is a 2D stencil. The grid size is either 8192 or 16384 and the block size is 128 or 256. The application is memory bound and the tasks are very fine-grained (about 5×10^5 clock ticks). Concurrent data structures are under pressure because workers end their tasks quickly. It explains the big increase in task creation cost (Fig. 2 jacobi_taskdep for all the inputs) wherein the generating thread run in quasi-concurrence with one of the $P - 1$ other threads.

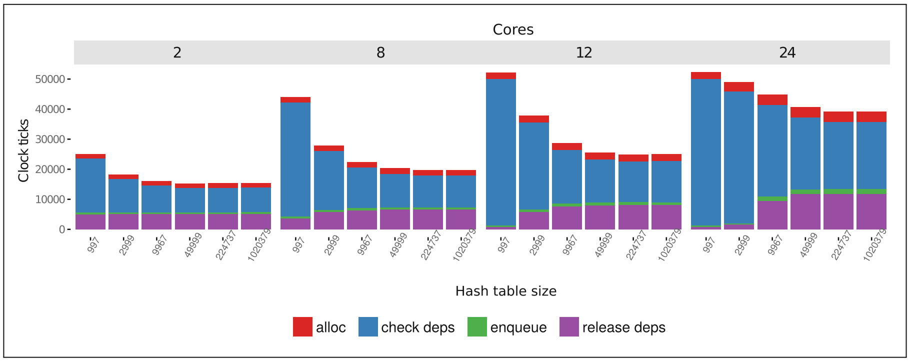


Fig. 3. Completion time breakdown of the jacobi_taskdep benchmark for different hash-table sizes (x axis). The standard hash-table size in libOMP is 997. The groups refer to the number of cores used.

3.2 Impact of the Task Serialization

In the LLVM libOMP, the task queues are bounded to 256. When the queue is full, the task throttling forces the serialization of the newly created tasks. Such a situation arises when the generating thread creates tasks faster than the worker threads can consume them. Increasing the queue size may impact the scheduling order of the tasks. For instance, in jacobi_taskdep [27], the generating thread creates first a set of independent tasks to copy an old data version in the new data version, then it creates tasks making stencil computation from an old data version to produce a new version. Tasks of the second set depend on tasks of the first set. In this case, the task throttling may block generation of tasks of the second set: the worker thread may not activate the successor tasks because they are not yet submitted!

On jacobi_taskdep and on the smallest grid (8192, blocksize = 128), we observe between 15% to 25% of gains for a range of a number of cores without task serialization (a queue of size 2^{16} is large enough). For a grid of size of 16384 and with the same block size (generating 4 times more tasks), the gain ranges from 2% on 24 cores to 19% on 2 cores (15% on 8 cores) with a small standard deviation.

3.3 Impact of the Hash Table Capacity

The hash table converts memory addresses to meta data in the libOMP procedure to compute dependencies (`__kmp_process_deps`). libOMP implements a hash table with separate chaining when keys are hashed to the same slot. When the load factor of the hash table increases, the cost of insertions becomes linear in the number of chained keys. If the number n of dependencies is high, the cost of finding a key is on average $O(n/s)$ where s is the number of slots.

By default, the number of slots in libOMP is 997 for each implicit task (which generally creates more dependent tasks). With this condition, the load factor of the hash table is near to 1: almost all insertions cause hash collisions. We experiment `jacobi_taskdep` on a grid size of 16384 with a small block size of 128 and with different sizes of the hash table. The number of dependencies to resolve is 2883584. For sizes bigger than 49999, the gain is small. The completion time on 24 cores is 3.16 s with the default value and 2.21 s with a hash-table size of 49999: the gain on the completion time reach 30%. Figure 3 reports the completion time breakdown of the internal task management. As expected, the cost of checking dependencies is reduced as the hash table is getting bigger.

4 Evaluation of the Gysela Testbed Application

The evaluated testbed application is a prototype of semi-Lagrangian 2D advection extracted from Gysela, an iterative gyro-kinetic simulation of magnetic fusion plasmas [17]. The extracted part is the most computationally intensive of the whole application and improving its performance is a major concern. The prototype makes the uses of task-based scheduling since it offers a promising approach to improve the performance of the existing code (based on OpenMP fork-join directives) through a better data locality and a finer-grained parallelism.

4.1 Overview

Being able to maintain the application is crucial since several algorithmic variants are provided and new algorithms are regularly devised. While studying this aspect is beyond the scope of this paper, it deeply impacts the evaluated code. Indeed, the prototype is split into independent computational parts called software components [23] in such a way parts can be easily replaced. Components are then assembled during a compilation process [5] that produces an OpenMP code.

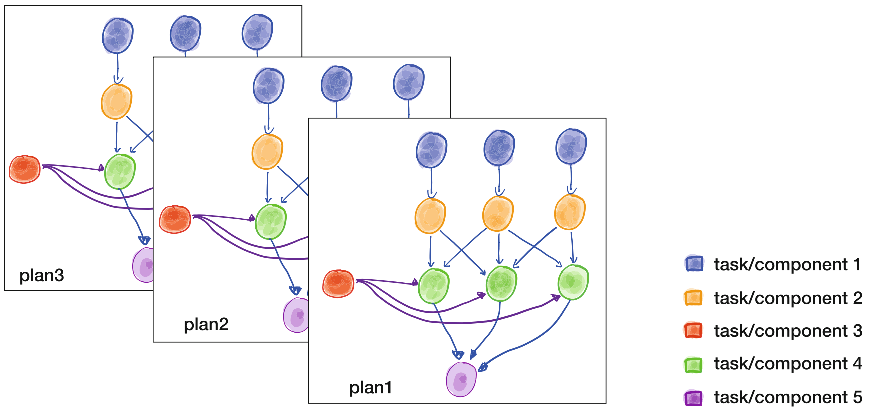


Fig. 4. Sketches of dependencies between OpenMP tasks in our testbed application. It represents tasks working on three planes. Each application level component spawns tasks for all the planes. (Color figure online)

The prototype iterates several times over 2D slices (plane) of 3D and 4D arrays. An iteration is defined by a sequence of 5 components. Each component generates a bag of independent tasks (following an SPMD approach) working on sub-parts of the planes (usually few lines). Assembling components results in adding dependencies between the generated bags of tasks. Figure 4 displays the structure of the task graph submitted to the runtime per plane of the working set. Tasks that work on different planes are totally independent. Because the graph structure is quite simple, at the beginning of this work we were very confident to delegate the all task scheduling concerns to an OpenMP runtime. However, performance issues have been identified on current OpenMP runtimes.

4.2 Task Submission

A carefully hand-written OpenMP-native implementation has been designed to study how fast it can be to use OpenMP tasks when all maintainability constraints are skipped. This implementation submits tasks by following a depth-first strategy, making use of recursive tasks (enabling parallel submission of independent tasks) and synchronization steps (enforcing runtimes to work on a sliding window of tasks). This implementation is 38% faster thanks to a better tasks scheduling and data-reuse in caches.

Although the hand-written implementation has demonstrated the feasibility in term of performance, important concerns such as code readability and separation of concerns are totally ignored. By using a HPC component model [5], the whole task graph is submitted sequentially all at once using a breadth-first strategy, as for the jacobi_taskdep benchmark. In practice, Component 1 submits a bag of many tasks, then Component 2 do the same and so forth.

It is worth noting that such a design comes from maintainability constraints. Indeed, the separation of concerns that helps to maintain components also hinders the use of a depth-first submission strategy. Moreover, it also prevents components to make assumptions on the implementation of other components, such as the dependency of submitted tasks. Since OpenMP 4.5 provides no way to submit dependent tasks in parallel, submission is doomed to stay sequential. Nevertheless, this design suffers from several sources of slowdown with both GNU libGOMP and LLVM libOMP and shared common conclusions with previous sections.

4.3 Characteristics of the Performances Drop

As for `jacobi_taskdep`, task submission becomes slower than the actual execution of tasks before they can be fine enough for the computation to fit better in caches resulting in starvation of worker threads and higher completion times. This high overhead comes from a combination of many technical factors: a small fixed-size hash table not well-suited for so many tasks, a contention of shared data structures in runtimes as tasks are being submitted while others are running.

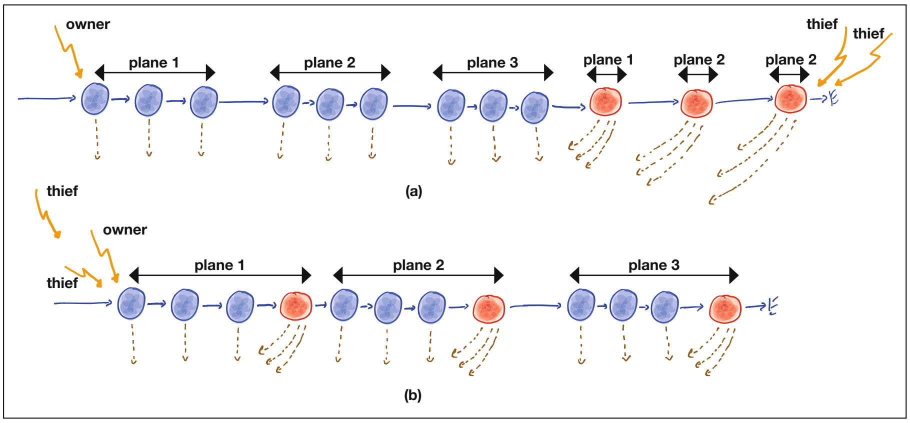


Fig. 5. Reordering strategy principle. (a) State of the ready list with the original work stealing after the task submission from components 1 (blue), 2 (hidden because tasks are dependent of the component 1) and 3 (red). See Fig. 4. (b) State of the ready list using the reordering strategy. (Color figure online)

The task throttling prevents the execution of tasks using a depth-first strategy as shown in Sect. 2.3: the submission is halted and current tasks executed before dependent tasks can be submitted. Without clairvoyance on all the computation, the execution order is close to the sequential order of task creation: this breadth-first strategy causes tasks to work simultaneously on a bigger amount of data (all the planes) resulting in poor utilization of caches.

4.4 Improving Locality Through Tasks Rescheduling

Even when the task throttling threshold is increased, the available scheduling algorithms are not able to group the execution of tasks working on the same plane although they are dependent and share data. The execution order mainly follows the submission order which turns out to be inefficient in our case. Figure 5(a) represents the submission order in the scheduler's ready list of the generating thread and the way the owner thread and thieves operate on the list during a steal operation. In work stealing, the owner (victim) and the thieves operate at the two extremities of the list to avoid any contention.

However, here, we want the cores to share data in caches. It is preferable that all threads operate on the same side of the list to favor data sharing. Thus, the LLVM libOMP function `__kmp_steal_task` has been modified to work in cooperation with functions that enqueue and dequeue tasks for the owner thread of the queue. Now, a thread enqueues new ready tasks at the same side of the list, where all other threads are working.

Keeping lists ordered as in case (a) is not enough, the ready tasks (red tasks of Fig. 4) have to be enqueued close to those working on the same plane. Thus, we have developed a fast reordering strategy of the ready list which computes

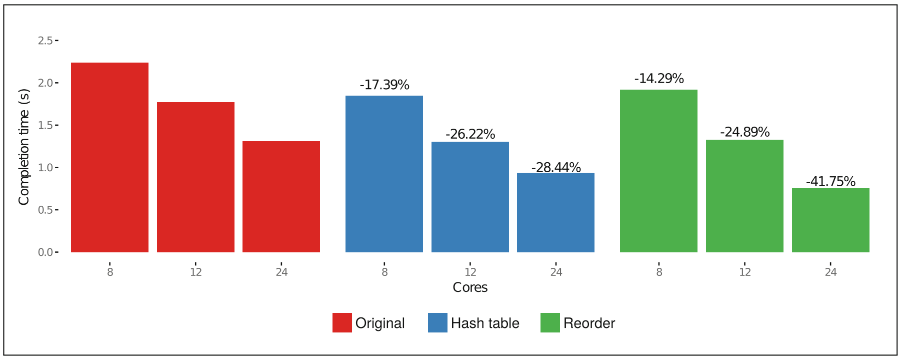


Fig. 6. Comparison of completion times for the Gysela application with different configurations.

on-line position where to insert ready tasks. This helps to favor the case (b) of Fig. 5. The heuristic is simple and well-suited for such a dependency task graph. It adds $O(1)$ instructions per dependencies.

Each task keeps the range of tasks in the ready list on which it depends. A task having no predecessor task is enqueued in the ready list and it initializes the range on itself. The algorithm which computes dependencies visits, for each newly created task, all its predecessors. During this step, the union of the range of all predecessors is incrementally computed, and the last inserted tasks in the ready list to the oldest in the union range is reordered. Due to dependencies, the ranges tend to include all the tasks. The reordering is currently stopped when the ranges become too wide.

Figure 6 reports the completion time on 8, 12 and 24 cores of the Gysela testbed application. A bigger hash-table size (132069 in place of 997) improves the performance by at least 17%. On 24 cores, the reordering achieves a performance gain of 41% over the original LLVM libOMP library.

5 Discussion

The OpenMP standard becomes predominant in the HPC runtime community. The recent integration of tasks into the standard has completely changed the way applications can describe parallel algorithms, enabling the description of more complex and finer-grained parallel computations. However, we are facing issues where OpenMP specification does not help us to guarantee performance portability. Indeed, in our case based on the decision to delegate the task management to OpenMP, the task granularity is enforced to reach high performance, while the task submission order is a consequence of the need for separation of concerns in the code. These two factors are the main sources of the issues explained in this paper: the task implementation of experimented runtime exhibits a high overhead and the submission order is not well-suited for reusing cached data. It seems a better long-term solution to improve OpenMP rather than handcrafting the generated OpenMP code.

What solutions are offered by OpenMP? Let us consider several opportunities to solve our issues.

5.1 Optimizing OpenMP Runtime Implementation

There are technical solutions for some issues presented above. The first one concerns the task throttling heuristic which is too basic in LLVM and GCC runtimes. One way is to integrate a more robust heuristic, for instance, such as in [12]. Another possible direction would be to claim that such heuristic will potentially always takes wrong decisions, as in our case, with a strong performance loss. Note that in our past work [9], thanks to a very low overhead in task creation, our implementation, without any throttling heuristic, was very competitive with the GCC or Intel implementations. We think that the task granularity is an algorithmic parameter and that OpenMP provides an explicit way to control it using the clause `if` of the task directive. Thus, in our point of view, it is better to disable any throttling heuristic in the runtime that may impact performance, even if it is in few cases such as ours.

Another important parameter which impacts performances is the cost of finding dependencies using the hash table. Preliminary results for GCC exhibits a similar behavior. The LLVM runtime has a too small hash-table that, indeed, generate a lot of hash collision. This problem should be studied and we currently integrate in the LLVM runtime a resizable hash table (the size depends on the load factor).

5.2 Parallelization of Task Submission

As described in Sect. 4.3, if the task submission is slow compared to the execution, the scheduler may never be able to activate the dependent tasks because they are not yet created: the scheduler is not clairvoyant. A straightforward idea is to make the task submission parallel. As for Gysela, a simple way would be to take into account the independence of tasks that belong to different planes. However, the component model used need to be extended to take into account the hierarchical structure of some applications such as Gysela and the high-level component assembly compiler back-end need to be changed too.

Moreover, according to the current OpenMP standard, the parallel submission is restricted to independent tasks only. The enforced constraint on the depend clause [8] is that it "*establishes dependences only between sibling tasks*", *i.e.* between tasks that are child tasks of the same task region.

Past projects have deals with a way to parallelize task submission in presence of dependencies. For instance, Athapascan-1 [14] was able to successfully parallelize the task graph submission of a stencil [22] on distributed architectures using a *postponed access mode* in order to delegate real access to data to sub tasks. More recently, a similar solution proposed for OpenMP with the use weak dependencies [20] seems very interesting if implementation scale enough with the number of submitted tasks.

5.3 Specialization of Task Scheduler

It is generally accepted that task scheduling depends on the targeted application. How to specialize a task scheduler for an OpenMP program? In addition to its original implementation, Cilk [13] provides guarantees on the expected performances in term of *work* and *depth* or *critical path*. What could be such a performance model for OpenMP task schedulers, even in presence of restrictions?

We propose a two steps organization of the way applications may influence the task scheduler of an OpenMP runtime. First, hints should be pass to the runtime in order to schedule a group of tasks according to a specific heuristic. Similarly to the clause `schedule` available for work-sharing loops, we expect a clause `task_schedule` for task groups and parallel directives. Such a clause enable the application to pick a specific task scheduler (among those provided) that should be preferred by the runtime, and may be defined by some expert users.

Secondly, in the same way OMPT has been defined to capture (in a portable manner) the state and the events generated by OpenMP runtimes, we expect to have access to an API (for experts) to enforce actions made by the runtime in order to have a better control over the scheduler or to redefined it.

6 Related Work

Optimizing task submission has been the subject of numerous works. In lazy approaches, the task creation is delayed until an idle resource requires tasks [16, 24]. Compilation strategies can reduce the overhead by exploiting the structure of the scheduler: for instance, the Cilk compiler generates two variants of each task (fast and slow clones) [13] in a way that move overheads out of the work and onto the critical path. However, this method, defined in Cilk as the *work-first principle*, may come at the expense of an impaired scalability. In [4], the authors have similar considerations about the generation of fast/slow clones. Orthogonal optimizations concern the optimization of the data-structure representation. The size of internal descriptors of the dependent tasks in LLVM libOMP is at most of 424 bytes per dependency, while in libKOMP, a native task descriptor is less than *64 bytes* explaining most of the speedup [9].

Swann [25] compared different methods of dependency analysis. TurboBLYSK [21] has proposed a way to cache dependencies of task graphs in order to reuse them without any overhead during the resolution. Following the work-first principle, in [15], the computations of dependencies have been moved from the work to steal operations.

A fast task creation can reduce the inactivity of worker threads. The scheduling algorithm may have a strong impact on the overall performance, such as the reorder method proposed in Sect. 4.4. A lot of scheduling heuristics in runtime systems has been proposed to improve the task locality [3,7,10] and to control the task affinity [19,26], but few of them are dealing with task reordering as presented above.

7 Conclusion

This paper has presented preliminary reports of using fine-grained tasks in OpenMP. Most of the measures and developments have been made with the LLVM OpenMP runtime supported by the LLVM group. Due to the fine granularity and preliminary experiments, we assumed that GNU libGOMP would behave the same way with at least similar overheads. The completion time breakdown analysis has focused on the task submission, especially costs related to checking dependencies and in the way to make the scheduler clairvoyant in order to reorder the on-line queue of ready tasks.

Further investigations on a wider range of applications are needed for the reordering method.

Several extensions of the Intel libOMP have been proposed and implemented. Results obtained on the Gysela prototype are satisfactory. Future works will focus on optimizing an OpenMP runtime for issues identified by such an application: support for fine-grained task. Finally, if the overhead cannot be avoided, then parallelizing the submission may be a solution.

References

1. GNU libgomp. https://gcc.gnu.org/onlinedocs/libgomp
2. Intel®OpenMP* Runtime Library (2016). https://www.openmprtl.org
3. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2000, pp. 1–12. ACM, New York (2000)
4. Agathos, S.N., Kallimanis, N.D., Dimakopoulos, V.V.: Speeding up OpenMP tasking. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 650–661. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32820-6_64
5. Aumage, O., Bigot, J., Coullon, H., Pérez, C., Richard, J.: Combining both a component model and a task-based model for HPC applications: a feasibility study on gysela. In: Proceedings of GCCGrid 2017. IEEE (2017)
6. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 63–77. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85261-2_5
7. Blelloch, G.E., Gibbons, P.B., Matias, Y.: Provably efficient scheduling for languages with fine-grained parallelism. J. ACM **46**(2), 281–321 (1999)
8. OpenMP Application Review Board: OpenMP application programming interface - version 4.5, November 2015. https://www.openmp.org
9. Broquedis, F., Gautier, T., Danjean, V.: libKOMP, an efficient OpenMP runtime system for both fork-join and data flow paradigms. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 102–115. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_8

10. Chen, S., et al.: Scheduling threads for constructive cache sharing on CMPs. In: Proceedings of SPAA 2007, pp. 105–115. ACM, New York (2007)
11. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proceedings of ICPP 2009, pp. 124–131. IEEE (2009)
12. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 36:1–36:11. IEEE Press, Piscataway (2008)
13. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. SIGPLAN Not. **33**(5), 212–223 (1998)
14. Galilée, F., Roch, J.L., Cavalheiro, G.G.H., Doreille, M.: Athapascan-1: on-line building data flow graph in a parallel language. In: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT 1998, pp. 88–95. IEEE Computer Society, Washington, DC (1998)
15. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO 2007 (2007)
16. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: implementing a fast parallel call. J. Parallel Distrib. Comput. **37**(1), 5–20 (1996)
17. Grandgirard, V., et al.: A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. Comput. Phys. Commun. **207**, 35–68 (2016)
18. Olivier, S., et al.: UTS: an unbalanced tree search benchmark. In: Almási, G., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72521-3_18
19. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Prins, J.F.: Scheduling task parallelism on multi-socket multicore systems. In: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2011, pp. 49–56. ACM, New York (2011)
20. Pérez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the integration of task nesting and dependencies in OpenMP. In: IPDPS, pp. 809–818. IEEE Computer Society (2017)
21. Podobas, A., Brorsson, M., Vlassov, V.: TurboBŁYSK: scheduling for improved data-driven task performance with fast dependency resolution. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 45–57. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_4
22. Revire, R.: Scheduling dynamic task graph on large scale architecture. Ph.D. thesis, Institut National Polytechnique de Grenoble - INPG, France, September 2004. https://tel.archives-ouvertes.fr/tel-00010909
23. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
24. Traoré, D., Roch, J.-L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel STL algorithms. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 887–897. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85451-7_95
25. Vandierendonck, H., Tzenakis, G., Nikolopoulos, D.S.: Analysis of dependence tracking algorithms for task dataflow execution. ACM TACO **10**(4), 61:1–61:24 (2013)

26. Virouleau, P., Broquedis, F., Gautier, T., Rastello, F.: Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 531–544. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_39
27. Virouleau, P., et al.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_2

Mapping OpenMP to a Distributed Tasking Runtime

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

[1] Department of Computer Science, University of Houston, Houston, TX, USA
jakemp@uh.edu

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

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

1 Introduction

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

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

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

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

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

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

2 Overview of OpenMP and HPX

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

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

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

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

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

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

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

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

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

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

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

3 Runtime Implementation Overview

3.1 OpenMP Runtime Task Management

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

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

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

3.2 The HPX Runtime

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

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

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

A future is used to coordinate shared state between two or more dependent tasks. If a task hasn't completed when a second, dependent task is being created, the latter will append the remainder of its task creation to the end of the first task. Once complete, the first task will resume creation of the second task. If it does not depend on any other inputs, then the second task will begin executing immediately. If the second task has other outstanding dependencies, the first task will pass the remainder of the second task's creation to the next task it depends on, and another task will be pulled from a work queue and executed.

4 The OMPX Implementation

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

4.1 Initialization and Parallel Regions

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

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

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

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

4.2 Worksharing Constructs and Synchronization

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

4.3 Tasking

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

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

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

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

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

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

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

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

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

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

5 Evaluation

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

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

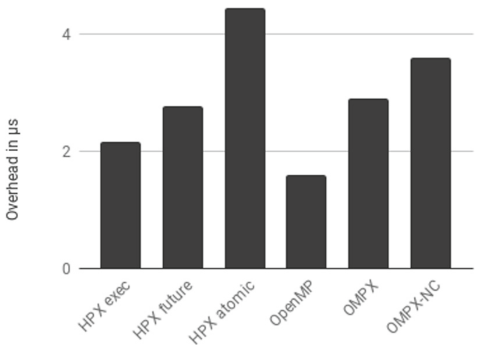


Fig. 5. Task creation overhead per task

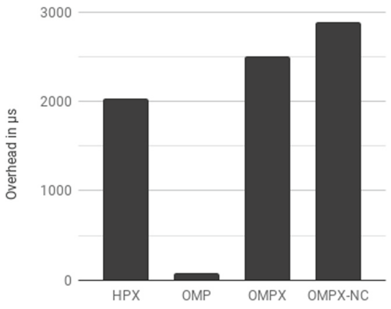


Fig. 6. EPCC barrier benchmark

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

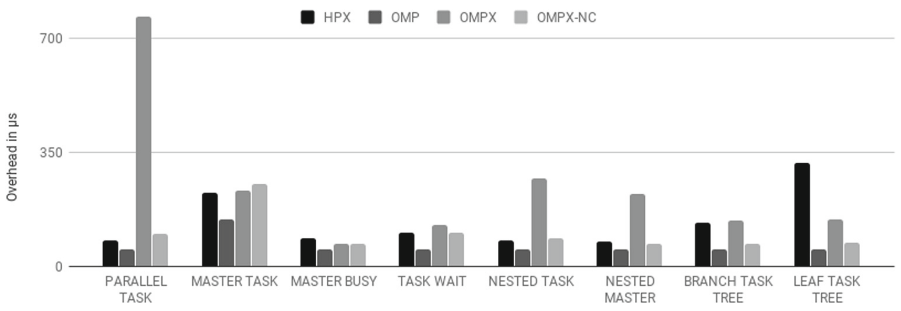


Fig. 7. EPCC Taskbench overhead

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

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

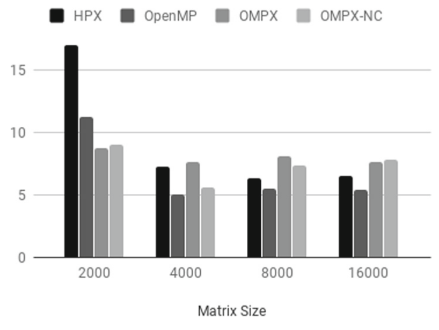


Fig. 8. Jacobi speedup

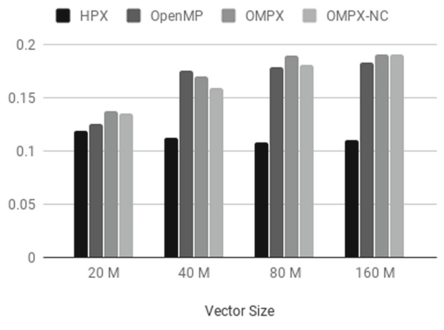


Fig. 9. Stencil speedup

We have four kernel benchmarks to evaluate the OMPX runtime: LU, Jacobi, 1D Stencil, and Nqueens. The LU decomposition benchmark divides the matrix into blocks and works on it in place, with each task writing to one block and reading from multiple. Jacobi iteratively solves the 2D heat equation for a matrix that is divided up into chunks of rows, with each task writing one chunk and reading from three. 1D stencil solves the 1D heat equation for an array which is divided into chunks that the tasks operate on, similar to Jacobi. In each of these kernels, the only synchronizations are task dependencies and a final wait after the tasks have been created. The Nqueens benchmark solves the Nqueens problem on a given board size, and has no data dependencies, only taskwait.

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

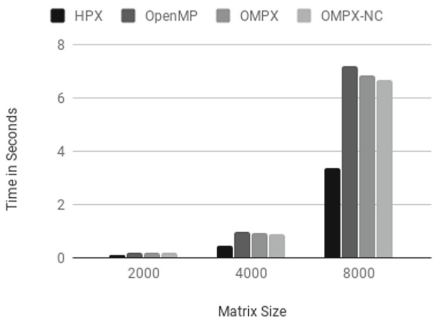


Fig. 10. LU execution time

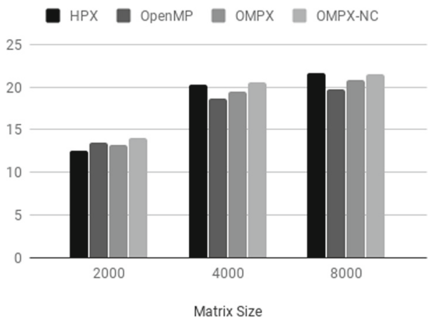


Fig. 11. LU speedup

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

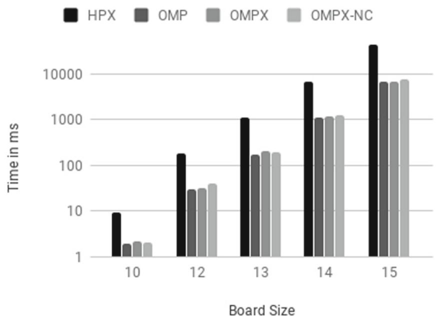


Fig. 12. Nqueens execution time

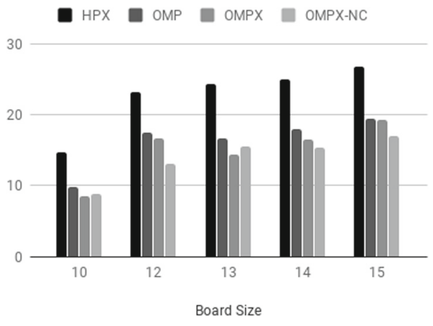


Fig. 13. Nqueens speedup

6 Related Work

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

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

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

7 Conclusions

We have introduced OMPX, an HPX based implementation of the Intel OpenMP runtime, and discussed how a multi-paradigm programming interface like OpenMP can be mapped to a purely task based library like HPX. We have used microbenchmarks and kernel applications to to compare the performance of HPX, the Intel OpenMP runtime, and two versions of OMPX. In the benchmarks where data dependencies existed, HPX and OMPX were able to leverage locality in a way that OpenMP currently does not to achieve superior performance. Additional benchmarks using larger applications would be desirable, but applications that are task dependency based are not widely available.

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

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

References

1. Intel OpenMP* runtime. https://www.openmprtl.org/
2. Intel threading building blocks user guide. https://software.intel.com/en-us/node/506045
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03869-3_80
4. Ayguade, E., et al.: The design of OpenMP tasks. IEEE Trans. Parallel Distrib. Syst. **20**(3), 404–418 (2009)
5. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage And Analysis, p. 66. IEEE Computer Society Press (2012)
6. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 4.0
7. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for OpenMP tasks. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 271–274. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_24
8. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the new adventures of old x10. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, pp. 51–61. ACM (2011)
9. Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: PTG: an abstraction for unhindered parallelism. In: 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), pp. 21–30. IEEE (2014)
10. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Proces. Lett. **21**(02), 173–193 (2011)

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

Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime

Jannis Klinkenberg[1], Philipp Samfass[2], Christian Terboven[1], Alejandro Duran[3], Michael Klemm[3], Xavier Teruel[4], Sergi Mateo[4], Stephen L. Olivier[5](✉), and Matthias S. Müller[1]

[1] Chair for High Performance Computing, IT Center, RWTH Aachen University, Aachen, Germany
{j.klinkenberg,terboven,mueller}@itc.rwth-aachen.de

[2] Department of Informatics, Technical University of Munich, Garching, Germany
samfass@in.tum.de

[3] Intel, Santa Clara, USA
{alejandro.duran,michael.klemm}@intel.com

[4] Barcelona Supercomputing Center, Barcelona, Spain
{xavier.teruel,sergi.mateo}@bsc.es

[5] Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, USA
slolivi@sandia.gov

Abstract. In modern shared-memory NUMA systems which typically consist of two or more multi-core processor packages with local memory, affinity of data to computation is crucial for achieving high performance with an OpenMP program. OpenMP* 3.0 introduced support for task-parallel programs in 2008 and has continued to extend its applicability and expressiveness. However, the ability to support data affinity of tasks is missing. In this paper, we investigate several approaches for task-to-data affinity that combine locality-aware task distribution and task stealing. We introduce the task affinity clause that will be part of OpenMP 5.0 and provide the reasoning behind its design. Evaluation with our experimental implementation in the LLVM OpenMP runtime shows that task affinity improves execution performance up to 4.5x on an 8-socket NUMA machine and significantly reduces runtime variability of OpenMP tasks. Our results demonstrate that a variety of applications can benefit from task affinity and that the presented clause is closing the gap of task-to-data affinity in OpenMP 5.0.

Keywords: OpenMP · OpenMP tasks · Task affinity
Task scheduling · Work stealing

1 Introduction

Modern non-uniform memory access (NUMA) shared memory systems comprise several sockets with multiple cores and local memory connected via an internal fabric such as the Intel® QuickPath Interconnect [13]. Cores on a socket can access local memory with lower latency and higher bandwidth than remote memory. A proper data distribution and data locality (i.e., the alignment of computation and data accessed by it) is crucial to sustain performance and exploit the full potential of these architectures. Since the trend to increase size and complexity of such machines continues, we consider these issues to become even more important in the future.

OpenMP* is a programming paradigm that is widely used for shared memory parallelization. Tasking, which allows parallelization of irregular and recursive algorithms, was first introduced with OpenMP 3.0 [8] and applicability as well as functionality have been extended since then. Many implementations already use task stealing for automatic load balancing and hence, for improving parallel performance. However, OpenMP still lacks locality-aware task scheduling and does not provide a means to specify or consider affinity between tasks and data. Thus, dynamic mapping of tasks to threads during run time leads to variations in performance and increased execution times on NUMA architectures.

In this paper, we discuss several approaches for task-to-data affinity. By providing hints to compiler and runtime system, we guide task scheduling to reduce NUMA effects and increase sustainable memory bandwidth. We investigate various scenarios to identify applications and situations that may or may not benefit from task affinity using compute nodes from the production HPC system at RWTH Aachen University.

Our work makes the following contributions:

1. We introduce the revised `affinity` clause for the `task` construct that will be included in OpenMP 5.0 and reason about fundamental design choices.
2. We detail our experimental implementation in the LLVM OpenMP runtime and explain how NUMA-aware task distribution and stealing can be applied to influence the mapping between tasks and threads.
3. We perform an evaluation on different architectures using multiple benchmarks to demonstrate the performance for various types of applications.

The remainder of the paper is structured as follows. Section 2 provides information about related work in the area of task affinity and data locality using OpenMP. Essential design concepts to support task affinity and the revised `affinity` clause are introduced in Sect. 3. In Sect. 4, we describe our experimental implementation in the LLVM runtime. Section 5 presents an evaluation of our approach before we conclude and discuss future work in Sect. 6.

2 Related Work

Literature induces several contributions that are using information provided by OpenMP task dependencies to mitigate the above-mentioned issues arising with task parallel programs on NUMA architectures. Muddukrishna et al. [6] propose techniques for data distribution and locality-aware scheduling. They implement separate task queues for architectural locations, i.e., NUMA nodes or home caches, and use task data dependencies and weighted sums of page-wise access latencies to identify adequate target queues for tasks. Task stealing is based on the ranking of NUMA node distances. Virouleau et al. [12] also analyze data specified in the `depend` clauses. They implemented approaches for data distribution, assignment of tasks and browsing the architectural hierarchy during task stealing inside the XKAAPI runtime system and evaluated their implementation on a 192-core NUMA machine.

There are also numerous other approaches. Huang et al. [5] introduce features for explicit locality control in OpenMP. Threads are mapped to locations in a block-wise fashion. Parallel regions as well as worksharing and tasking constructs can then be executed with a defined set of locations and their threads. Thus, they enable the programmer to influence the data layout and align tasks and data. Olivier et al. [7] extended the OpenMP runtime with a set of API calls to explicitly control the locality domain on which tasks are executed. Additionally, they performed evaluations with a version that allows stealing between locality domains and a strict version that represents prescriptive assignments between tasks and locality domains.

In our previous work [10], several design issues for task-to-thread and task-to-data affinity are discussed. They propose a new clause for the `task` construct as well as solutions for `taskgroup` and `taskloop`. Experimental extensions for the `task` construct have been implemented in the Nanos++ runtime [2]. Evaluation indicates that the approaches improve performance up to 40% on a 2-socket machine with Intel® Xeon® processors. In this paper, we carry out a deeper investigation on task-to-data affinity. We present an extended `affinity` clause and various scheduling strategies. Although data specified for affinity might also be used as dependences, the essential concepts are not the same. Compared to prior work, our work targets a clear separation between task dependences and affinity.

3 Task Affinity Support in OpenMP

In this section, we describe the essential concepts of our solution as well as the requirements and limitations associated with it. Further, we present the revised clause and design choices for it.

Looking at task-to-data affinity, the approach is that a programmer specifies data accessed by the tasks. This is then treated as a recommendation for the OpenMP implementation to execute the task on a resource that is close to the location of the data. Specifying data on the task construct implies that the data has to be accessible at the time of task creation. However, it provides an architectural independent abstraction level and relieves the programmer from thinking about explicit mappings (i.e., task-to-thread affinity) by placing the burden on the runtime system. The runtime system is responsible for identifying the physical location of the data references which might be an expensive operation. Futher, it has to find a suitable thread close to that location to execute the task, ideally on the same NUMA node.

As mentioned earlier, many OpenMP runtimes implement task stealing to achieve a better load balance. Thus, information provided in task affinity clauses should not be prescriptive but rather present hints to the runtime in order to guide task scheduling, establish better data locality and at the same time maintain the ability for quick load balancing.

With OpenMP 5.0, users will have the possibility to express task affinity using the following clause:

```
#pragma omp task affinity(list)
```

In contrast to our previous work, it is now possible to specify multiple data references and affinity clauses on a single task construct. Additionally, it offers the freedom to express affinity to an entire array section or a dynamic number of list items via iterators. Due to the fact that currently just task-to-data affinity will be supported in OpenMP, there will not be any affinity type identifier as we previously proposed [10]. However, if support for other kinds of task affinity is complemented in future specifications, type identifiers can be introduced. In that case, task-to-data affinity represents the default case for clauses without identifier.

This paper compares and evaluates two fundamental approaches for task-to-data affinity, which we refer to as modes. The first, here denoted by *domain* mode, demands that tasks are executed close to the original location where the data has been allocated. Another approach, denoted by *temporal* mode, aims to execute the task at the last location where a task that used the same data was executed. For the latter one, some kind of book keeping is required to keep track of the last place. In addition, if tasks are stolen, that information has to be updated.

While memory-bound applications might profit from referring to the original location of the data, the *temporal* mode might be well suited for compute-bound programs that rely on cache locality.

4 Implementation in LLVM

We developed an experimental implementation in the LLVM OpenMP runtime[1]. Since it is compatible with the Intel compiler and also shipped with clang, which is open source and available on many clusters, it has the potential to have an impact on a large community.

In 2013, Intel donated a large portion of their OpenMP runtime to the LLVM project. Due to the high compatibility, the Intel runtime internally works and performs similarly. To avoid compiler modifications we emulated the affinity clause with an API call implemented in our runtime. Benchmarks used for the evaluation call the API function directly before the task construct. Although the

[1] Our implementation is based on a LLVM development version for OpenMP 5.0 from September 2017 and is available at https://github.com/jklinkenberg/openmp/tree/task-affinity.

specification supports multiple affinity clauses and list items, we are currently only considering the first data reference that has been specified.

Our approach consists of two main components, a NUMA-aware task distribution and task stealing, which are detailed in Sects. 4.2 and 4.3 respectively.

4.1 Characteristics: LLVM Runtime vs Libgomp

Before presenting our NUMA-aware extensions, some default characteristics of LLVM and libgomp [4] (GCC's OpenMP runtime library) with regard to tasking are reviewed.

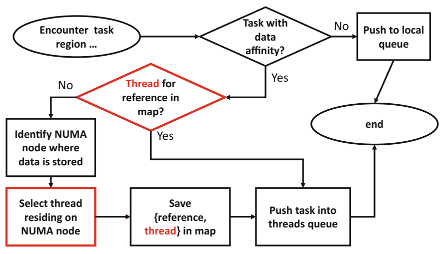
LLVM employs one task queue per OpenMP thread in the team. A thread that is encountering a task construct will by default create a task and push it to its local task queue. If a thread is idle, e.g., in a `taskwait` or `barrier`, it will start a defined sequence of actions. First, it will check its local queue for work. If this queue is empty, it will start stealing tasks from random thread queues until all work is done. If a thread has stolen and executed a task, the process starts from the beginning since the stolen task might have created child tasks that now reside in the local task queue again. Creating and processing tasks with multiple threads reduces the idle time and since they are mostly working on the local queue there is only little performance degradation due to lock contention. The design is also well suited for nested constructs if parent and child tasks work on the same data. On the contrary, unequal distribution of tasks leads to an increased overhead for task stealing and load balancing. Programs might also suffer from that design if tasks working on remote data reside in a local queue that could run faster in a different location.

In contrast to LLVM, libgomp implements a single central task queue. Thus, libgomp does not suffer from poor distribution of tasks. However, a single queue leads to a higher scheduling overhead because locks have to be acquired by each thread for each enqueue and dequeue operation to avoid data races.

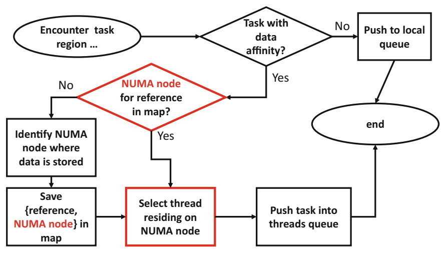
4.2 NUMA-Aware Task Distribution

In our implementation, we change the behavior of just pushing tasks to the local task queue. For tasks with the affinity clause the runtime first has to determine the physical location of the data reference. Usually physical data allocation of the OS is done in pages of, e.g., 4 KB. Analyzing multiple data references within a page leads to the same result. Hence, this choice is an adequate granularity for our approach. We use *move_pages* from libnuma [1] to identify the physical location. In general, *move_pages* allows explicit migration of individual pages to other NUMA nodes. However, if the target NUMA node is not specified it will not move the page but instead return the node where the corresponding page currently resides.

In the second step, a thread residing on that NUMA node or close to it must be selected by scanning the place list, and the task is pushed to its queue. To accomplish that we implemented the temporal and domain mode discussed in Sect. 3. Although book keeping is only required in *temporal* mode, we also use it


Encounter task region ...
Task with data affinity?
No
Push to local queue
Yes
Thread for reference in map?
No
Yes
end
Identify NUMA node where data is stored
Select thread residing on NUMA node
Save {reference, thread} in map
Push task into threads queue

(a) Temporal mode


Encounter task region ...
Task with data affinity?
No
Push to local queue
Yes
NUMA node for reference in map?
No
Yes
end
Identify NUMA node where data is stored
Save {reference, NUMA node} in map
Select thread residing on NUMA node
Push task into threads queue

(b) Domain mode

Fig. 1. Flow charts describing the process of the task distribution phase.

for performance improvements in *domain* mode. In *temporal* mode, the physical location has to be identified only once for a data reference. An entry containing the reference and the selected thread is added to the map. If a task is stolen by another thread, the entry is updated accordingly. For the next tasks using the same data, the current value in the map is used as a target. In *domain* mode, the NUMA node for the reference is saved, focusing on the original allocation location. With a map, this identification is needed only once, and the information can be reused for the next task specifying affinity to the same data. In *domain* mode, the thread selection is performed for every task again. Flow charts describing the process are illustrated in Fig. 1. Map entries remain for the complete application run and will be cleared afterwards.

To select threads inside a NUMA node we implemented three different strategies, a *random* selection of a thread on the node, selection in a *round robin* fashion and a strategy selecting the thread with the *lowest* queue size. In total, combining each mode with each strategy results in 6 different versions.

4.3 NUMA-Aware Task Stealing

To further improve our solution we adapted the task-stealing process of Sect. 4.1. We stick with the behavior to favor tasks in the local queue before stealing. However, before stealing from a random victim of the complete task team, a thread should prefer stealing from a thread residing on the same NUMA node. Thus, data locality can be improved but we still maintain load balancing between NUMA nodes.

5 Evaluation

In this section, we evaluate the performance of our experimental task affinity implementation and compare it with the same LLVM runtime without any extensions as baseline for all tests. In some parts of our analysis, we additionally compare it with GCC's libgomp and the Intel OpenMP runtime. For assessing the performance we use kernels of the following benchmarks.

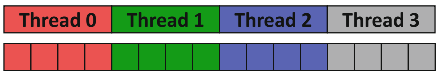
STREAM Synthetic Benchmark: Typically, STREAM is used to measure the sustainable cache or memory bandwidth on a given architecture. It consists of four different kernels that are executed multiple times per application run. For our research, we created a special version of the STREAM benchmark that utilizes OpenMP tasks for the kernels instead of a worksharing construct to represent memory bound, task-parallel applications and to determine an upper bound for improvement that can be obtained with task-to-data affinity. The size of each double array is 2^{31} elements, which matches 16 GB per array. Each kernel is repeated 10 times. To create enough tasks for distribution, each kernel is split into n_{tasks} OpenMP tasks, where n_{tasks} has been set to `OMP_NUM_THREADS` $* 20$.

Merge Sort: We included a task parallel merge sort from the Barcelona OpenMP Task Suite (BOTS) [3] as a representative for recursive divide and conquer algorithms. The vector size to be sorted was set to 2^{31} integer values.

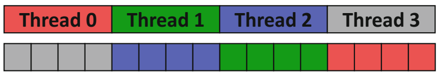
CG Solver with SPMXV: A common way to parallelize sparse matrix vector multiplication (SPMXV) is distributing the rows of the matrix among the participating threads using a worksharing construct. Depending on the sparsity pattern of the matrix, the SPMXV might suffer from severe load imbalance. One mitigation is to use a dynamic or guided schedule or fixed size chunking. However, in our scenario we use OpenMP tasks for that purpose and evaluate if and how task affinity can improve the performance of applications that suffer from natural load imbalances. Similar to STREAM, each chunk of rows is split into 20 separate tasks. Finally, we compare the tasking versions with a basic worksharing version of the solver. As input we use a sparse matrix with 235 million non-zero entries from an institutional application at RWTH Aachen University.

Health: The Health benchmark, also derived from the Barcelona OpenMP Task Suite, runs a simulation for a national health system. Due to its hierarchical nature, Health represents a divide and conquer algorithm with a tree-based data structure, which is able to exploit locality when traversing the tree. As input file we use *large.input* but set the number of cities per level (branching factor) to 48. In the data initialization, we evenly distribute the top level cities across NUMA nodes. By increasing the branching factor we also increase the work load and memory footprint of the application. Thus, effects can also be observed on larger machines with multiple NUMA nodes.

All tests are conducted on two different architectures that are part of the production HPC system at RWTH Aachen University. The first system is a two-socket NUMA machine equipped with Intel® Xeon® E5-2650v4 (codename "Broadwell") processors with 24 cores in total running at 2.2 GHz and 128 GB memory. In order to evaluate the impact of task affinity on systems with higher amount of NUMA nodes we also perform tests on an 8-socket NUMA machine consisting of Intel® Xeon® E7-8860v4 (code name "Broadwell") processors running at 2.2 GHz with 144 cores in total and 1 TB memory. Both systems run CentOS* 7.4 (RHLE) with Linux* kernel version 3.10.0–693.17.1, installed patches for Spectre/Meltdown and address space layout randomization (ASLR) set to 1.



(a) Parallel creator: Task creation and data initialization follow the same pattern.



(b) Parallel creator inverted: Due to flipped chunks threads create tasks that are working on remote data.

Fig. 2. Parallel task creation schemes.

In order to ensure reliability and reproducibility we apply the same environment and basic configuration to all application runs. Each program is compiled with the Intel® Composer XE for C/C++ version 18.0.1 and `-O3` optimization. libgomp versions are compiled with GCC 7.2.0. We use aligned memory allocation for data used in the benchmarks and distribute it across NUMA nodes relying on the first touch policy of the operating system and `#pragma omp parallel for schedule(static)`. Additionally, we disable Transparent Huge Pages (THP) in the Linux* kernel. To avoid movement or migration of data that has been distributed across NUMA nodes we turn off automatic NUMA balancing OpenMP threads are pinned to physical cores and equally distributed across the machine by setting OpenMP environment variables `OMP_PLACES=cores` and `OMP_PROC_BIND=spread`. Currently, several parts in the LLVM runtime take advantage of the assumption that tasks are just pushed to local queues which no longer holds for our approaches and might lead to deadlock-like situations. Solving that issue requires a higher implementation effort that is not part of this research. To avoid problems we disabled constraints during task stealing by setting `KMP_TASK_STEALING_CONSTRAINT=0`.

5.1 Preliminary Analysis with STREAM

Since the tasking version of STREAM is easy to understand and balanced, we decided to use it for addressing the following research questions:

How Does the Location Where Tasks Are Created Affect the Performance of an Application and How Can Task Affinity Help? The current implementation of the adapted STREAM benchmark allows multiple execution scenarios. Besides running with or without task affinity, one can select how to create OpenMP tasks for the kernels. With the first option, denoted by *single creator*, only the master thread is creating tasks but all threads in the team participate in the work. This option is comparable to the existing `taskloop` construct nested inside a `master` region, except that `taskloop` is currently not providing support for affinity specifications. In option two, denoted by *parallel creator*, each OpenMP thread is responsible for creating tasks corresponding to its own chunk that actually follows the same pattern than the data initialization does. An example is illustrated in Fig. 2a. We also created a third option, a worst case scenario denoted by *parallel creator inverted*, to analyze the impact if tasks are created in a different location than the data they are using. Here, each OpenMP thread

Table 1. STREAM: speedup compared to baseline LLVM version.

	Single creator	Parallel creator	Parallel creator inverted
domain.lowest	4.510	0.997	8.664
domain.rand	4.521	0.980	8.512
domain.round	4.501	0.981	8.516
temporal.lowest	3.695	0.980	1.398
temporal.rand	3.672	0.951	1.365
temporal.round	3.661	0.934	1.378

is still responsible for creating tasks but for a different chunk. To achieve that we flipped the order of chunks as demonstrated in Fig. 2b. Experimental results on our 8-socket machine with 64 OpenMP threads are shown in Fig. 3. Each option is executed 15 times without task affinity as well as with every task affinity combination presented in Sect. 4.2. As a reference point, we also added the result for application runs with the default STREAM benchmark.

As expected, the LLVM and Intel runtimes are reporting similar performance results and are already performing quite well in *parallel creator* versions because task creation and data distribution follow the same pattern. Thus, they are very close to the solution achieved with the default STREAM. Obviously, in such situations, using task affinity is not beneficial. We see slightly higher execution times for parallel task affinity versions caused by overhead for memory location tracking and sophisticated task scheduling.

On the other hand, there are noticeable performance differences in scenarios where tasks are not (always) created at the correct place. For the *single creator* scenario, random task stealing in the LLVM and Intel runtime prevents data locality in many cases. Looking at *parallel creator inverted*, threads start working on the local task queue first, that solely holds tasks accessing remote data, before trying to steal from others. In both situations, task affinity can mitigate these issues by distributing tasks to threads on the correct NUMA node. Table 1 displays speedup achieved for each setup compared to the baseline LLVM version. Compared to *domain* mode, versions with *temporal* mode are reporting poor performance caused by task stealing from a different domain. Due to the map update that happens after a task has been stolen, the next task accessing the same data will be pushed to the thread on the new NUMA node. Consequently, the task accesses the data remotely. Especially, memory-bound applications can profit from using *domain* mode whereas the *temporal* mode might provide better results for compute-bound applications where the same data is repeatedly accessed by multiple tasks.

Tests with libgomp, which employs a central task queue, illustrate that it is severely suffering in the *single creator* setting. Due to its design, relative performance for the *parallel creator* version is even worse compared to LLVM and Intel. On the other hand, inverting the chunks does not lead to a serious performance degradation in libgomp. However, ultimately, we observe that our task affinity implementation is outperforming libgomp in most cases.

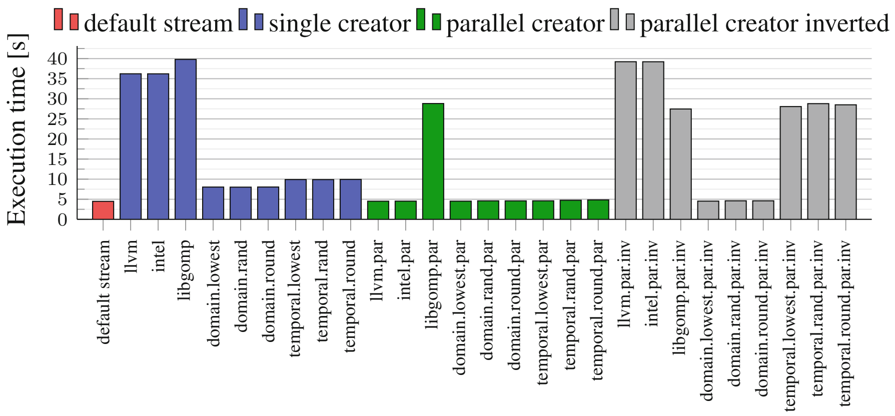


Fig. 3. Execution time (median of 15 runs) for STREAM benchmark runs with and without task affinity on an 8-socket NUMA machine using 64 OpenMP threads.

How Does NUMA-Aware Task Stealing Affect the Performance of Task Affinity? Distributing OpenMP tasks to adequate threads close to the data is the first step to establish data locality. However, the mapping between tasks and threads is not enforced to maintain load balancing. If threads are idle, task stealing is still expected to kick in. Especially in *single creator* setups or applications that exhibit an irregular task creation scheme, that might cause some disturbance. Although distributing a task to a thread on the correct NUMA node, it sometimes happens that another under-utilized thread, residing on a different NUMA node, steals the task before it is started at the desired thread. Consequently, this leads to remote memory accesses and higher execution time. To mitigate this issue we use NUMA-aware task stealing to prefer stealing from threads residing on the same node before stealing from any other threads. To investigate the influence on task affinity we perform *single creator* runs with NUMA-aware task stealing enabled and disabled. Figure 4 shows the speedup compared to the baseline for both versions. Although most performance gain stems from proper task distribution we observe that runs with NUMA-aware task stealing outperform those without and obtain an improvement of about 7% to 13%. Hence, it is used for further tests.

Is Task Affinity Able to Reduce the Run Time Variability of Task Executions? Complexity and execution time of the STREAM kernels differ which makes it hard to distinguish between real variations and those caused by different complexity. Therefore, we just focus on the Triad kernel for this analysis. We run a *single creator* scenario with the same configuration as before and record the execution time for each individual task. Figure 5 shows the distribution of execution times. The LLVM baseline version has a substantially higher spread and median which leads to the fact that several tasks are suffering from remote memory accesses. On the contrary, all versions with task affinity, although having some

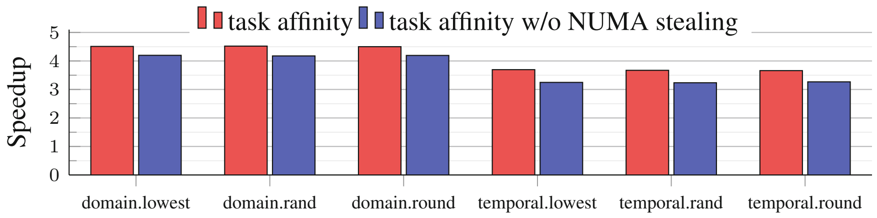


Fig. 4. Speedup for STREAM *single creator* runs on an 8-socket NUMA machine using 64 threads with NUMA-aware task stealing enabled and disabled compared to baseline LLVM runtime.

outliers caused by task stealing from a remote NUMA node, show a significant reduction of variability. Measuring remote data volume with Likwid [9,11] confirms that assumption. The baseline version is accessing 69% of all data remotely whereas for task affinity versions it is just about 13% to 17%.

What is the Most Promising Thread Selection Strategy? Results achieved with the balanced STREAM indicate that there is no thread selection strategy that is clearly outperforming the others. However, we notice a slightly better performance with strategy *lowest* for both, *domain* and *temporal* mode. For more complex or irregular benchmarks we expect *lowest* to obtain a fair load balance within a NUMA node at the expense of a negligible overhead to determine the queue with the lowest size. Therefore, *lowest* is considered the most promising strategy and used for further analysis in this section.

5.2 Overall Performance and Scalability

In this section, we evaluate the overall performance for the previously mentioned benchmark kernels with regard to scalability and speedup, i.e., the relative execution time compared to the baseline LLVM runtime. We execute each application with and without task affinity and vary the number of threads per NUMA node starting from one thread per NUMA node up to using all cores of the given architecture. The STREAM tasking version runs with a *single creator* scenario. Similar to STREAM, the SPMXV kernel in CG can also be executed with a single or parallel creators. As pointed out in Sect. 5.1, using task affinity is not profitable for a balanced *parallel creator* scenario that follows the same or a similar pattern than the data initialization. For SPMXV we selected a *parallel creator* scheme to investigate whether it makes sense to apply task affinity if an application exhibits natural imbalances. Additionally, we compare the results to a version that uses a worksharing construct for the kernel. Figure 6 presents absolute execution time as well as relative execution time compared to the baseline for each benchmark executed on our 2-socket and 8-socket NUMA machines.

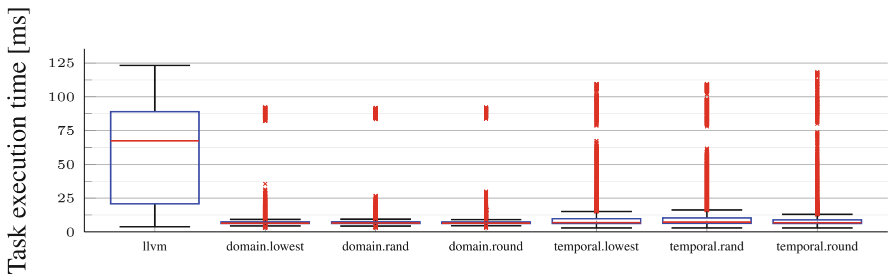


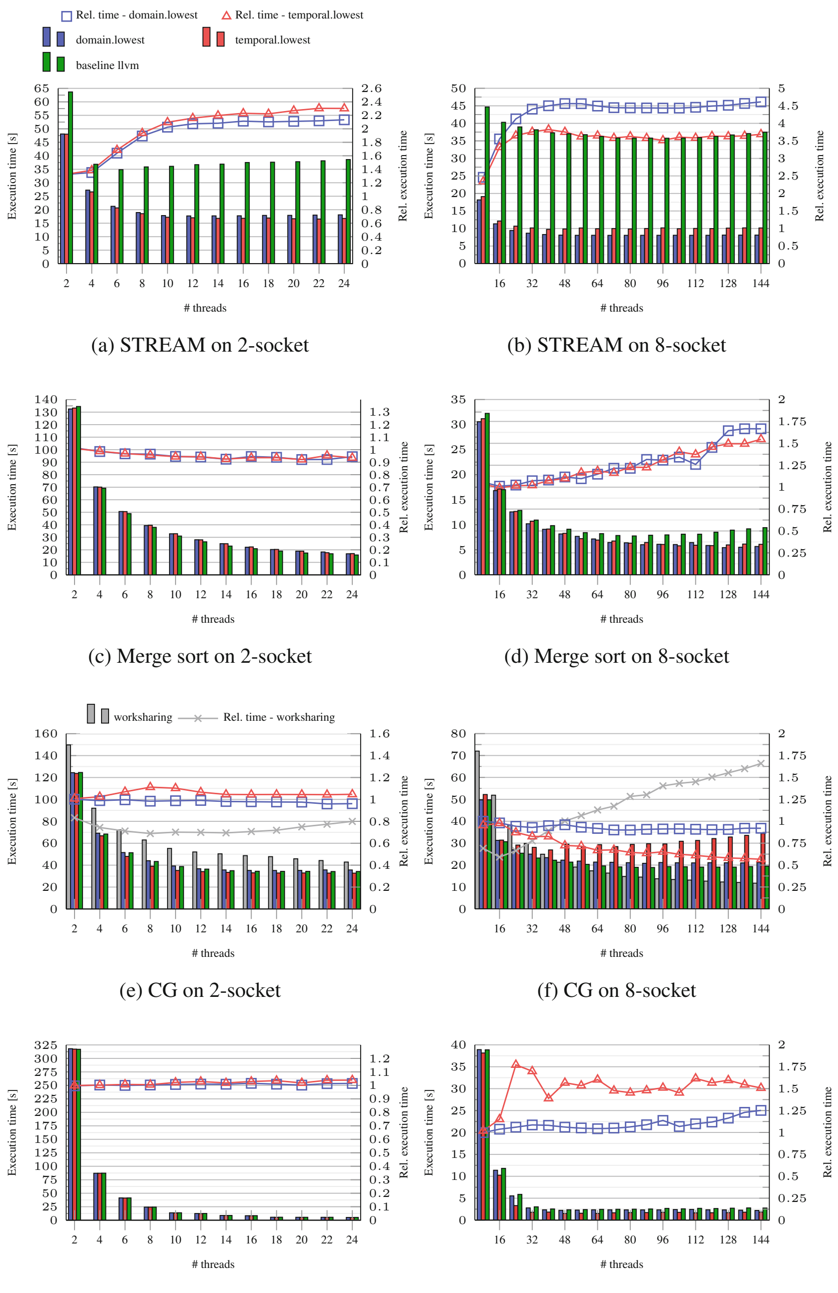
Fig. 5. Variability of individual task execution times for the Triad kernel using an array size of 2^{31}. Each box plot is based on execution times of 12,800 individual tasks.

STREAM stops scaling with 6 to 8 threads per socket. While the baseline is clearly suffering from remote memory accesses, affinity versions can saturate more bandwidth by mostly working on local data and improve performance up to 4.5x. On 8 sockets *temporal* mode is more prone to task stealing from a remote NUMA node whereas the chance to steal remote tasks decreases with two sockets and *temporal* performs better.

On the contrary, improvement for merge sort is moderate, especially when using a smaller number of cores. Nevertheless, by increasing the core count and putting more pressure on the memory subsystem we were able to achieve a performance gain of about 1.6x on 8 sockets. Remote memory accesses prevent further scaling of the baseline version at about 80 threads. Although parallel efficiency of affinity versions decreases with higher number of threads, there is still some improvement up to 128 threads. On two sockets NUMA effects are much weaker and using task affinity does not pay off.

Results for SPMXV indicate that, although dealing with imbalance, creating tasks close to the data in LLVM is already performing well. Overhead for affinity with *domain* mode does not result in a high degradation but also does not provide any advantage whereas *temporal* mode is not able to compete on 8 sockets, especially at higher core count. Further, we observe that tasking beats the worksharing version when using a smaller amount of cores. By increasing the number of cores, data chunks and the impact of load imbalance decrease and overhead of tasking versions becomes visible. Surprisingly, temporal mode performs slightly better than LLVM on two sockets.

For Health, a slight improvement can be achieved on two sockets. Since NUMA effects are much stronger on 8 sockets and Health profits from temporal locality, it can obtain a speedup of about 1.6x by using the *temporal* mode.



(a) STREAM on 2-socket

(b) STREAM on 8-socket

(c) Merge sort on 2-socket

(d) Merge sort on 8-socket

(e) CG on 2-socket

(f) CG on 8-socket

(g) Health on 2-socket

(h) Health on 8-socket

Fig. 6. Scalability results for benchmarks on a 2-socket and 8-socket NUMA machine: Absolute execution time and relative execution time compared to baseline LLVM. For these plots we use the median of 10 application runs.

6 Conclusion and Future Work

In this paper, we evaluated the language extensions to express task-to-data affinity that will be part of OpenMP 5.0. The revised `affinity` clause for the `task` construct and its design choices have been described. Further, we presented two fundamental approaches for task-to-data affinity. The first targets *temporal* locality by placing the task on the thread where the last task has been executed that used the same data. The second approach, denoted by *domain* mode, focuses on the the original location where the data has been allocated and distributes the task to a thread close to that location, ideally on the same NUMA node. Additionally, several strategies for choosing a thread within a NUMA node have been presented and evaluated.

We created an experimental implementation in the LLVM OpenMP runtime consisting of a NUMA-aware task distribution (i.e., the approaches mentioned above) and NUMA-aware task stealing to further increase data locality by preferring to steal from a thread within the same NUMA node. To assess the performance of our approach and to identify situations and applications that can benefit from task affinity we use several benchmark kernels representing various application types and two architectures with different numbers of NUMA nodes. Results show that a significant speedup can be obtained with *domain* mode for memory-bound applications with single task creator schemes or if tasks are created at a thread which is not close to the data. Applications with a parallel task creator scheme that follows the same pattern as the data initialization do not profit from using task affinity. Irregular and recursive applications, especially when relying on cache locality, may benefit from *temporal* mode on architectures with multiple NUMA nodes.

For future work we plan to investigate ways to enable the selection of modes presented in this paper by, e.g., adding a clause to the `parallel` or `taskgroup` construct. Another plan is to extend the `taskloop` construct to allow affinity expressions based on the iterator variable that are then internally passed to the created tasks. Since our approach is currently limited to a single data reference, we consequently plan to evaluate strategies that deal with multiple affinity specifications and list items.

Acknowledgements. 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 Energys 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.

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology through TIN2015-65316-P project and by the Generalitat de Catalunya (contract 2017-SGR-1414).

Some of the experiments were performed with computing resources granted by JARA-HPC from RWTH Aachen University under project jara0001. Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under grant numbers 01IH16004B (Project Chameleon).

References

1. libnuma. http://man7.org/linux/man-pages/man3/numa.3.html. Accessed 23 Apr 2018
2. Nanos++ runtime. https://github.com/bsc-pm/nanox. Accessed 26 Apr 2018
3. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: 2009 International Conference on Parallel Processing, pp. 124–131, September 2009
4. GNU: GOMP An OpenMP implementation for GCC. https://gcc.gnu.org/projects/gomp/. Accessed 16 Apr 2018
5. Huang, L., Jin, H., Yi, L., Chapman, B.M.: Enabling locality-aware computations in OpenMP. Sci. Program. **18**(3–4), 169–181 (2010)
6. Muddukrishna, A., Jonsson, P.A., Brorsson, M.: Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors. Sci. Program. **2015**, 5:1–5:16 (2015)
7. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 65:1–65:12. IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
8. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 3.0, May 2008. http://www.openmp.org/
9. Röhl, T., Eitzinger, J., Hager, G., Wellein, G.: LIKWID monitoring stack: a flexible framework enabling job specific performance monitoring for the masses. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 781–784, September 2017

10. Terboven, C., et al.: Approaches for task affinity in OpenMP. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 102–115. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45550-1_8
11. Treibig, J., Hager, G., Wellein, G.: LIKWID: a lightweight performance-oriented tool suite for x86 Multicore environments. In: Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW 2010, pp. 207–216. IEEE Computer Society, Washington, DC (2010)
12. Virouleau, P., Broquedis, F., Gautier, T., Rastello, F.: Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 531–544. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_39
13. Ziakas, D., Baum, A., Maddox, R.A., Safranek, R.J.: Intel QuickPath interconnect architectural features supporting scalable system architectures. In: 2010 18th IEEE Symposium on High Performance Interconnects, pp. 1–6, August 2010

Author Index