Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite

Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, Thierry Gautier

Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite

Philippe Virouleau[1], Pierrick Brunet[1], François Broquedis[4], Nathalie Furmento[2], Samuel Thibault [3], Olivier Aumage[1], and Thierry Gautier[1]

[1]INRIA, [2]CNRS, [3]University of Bordeaux, [4]Grenoble Institute of Technology
MOAIS and RUNTIME Teams, Computer Science Laboratories of Grenoble and Bordeaux, France

`firstname.lastname@inria.fr`
`thierry.gautier@inrialpes.fr`

Abstract. The recent introduction of task dependencies in the OpenMP specification provides new ways of synchronizing tasks. Application programmers can now describe the data a task will read as input and write as output, letting the runtime system resolve fine-grain dependencies between tasks to decide which task should execute next. Such an approach should scale better than the excessive global synchronization found in most OpenMP 3.0 applications. As promising as it looks however, any new feature needs proper evaluation to encourage application programmers to embrace it. This paper introduces the KASTORS benchmark suite designed to evaluate OpenMP tasks dependencies. We modified state-of-the-art OpenMP 3.0 benchmarks and data-flow parallel linear algebra kernels to make use of tasks dependencies. Learning from this experience, we propose extensions to the current OpenMP specification to improve the expressiveness of dependencies. We eventually evaluate both the GCC/libGOMP and the CLANG/libIOMP implementations of OpenMP 4.0 on our KASTORS suite, demonstrating the interest of task dependencies compared to taskwait-based approaches.

Keywords: *OpenMP, task dependencies, benchmarks, runtime systems, KASTORS*

1 Introduction

HPC architectures evolved so rapidly that it is now common to build shared-memory configurations with several dozens of cores. The recent appearance of technologies such as the Intel Xeon Phi co-processor makes affordable configurations with thousands of cores a not-so-far reality. Efficiently programming such large-scale platforms requires to express more and more fine-grain parallelism.

Standard parallel programming environments such as OpenMP have evolved to address this requirement, introducing new ways of designing highly parallel programs. Extending OpenMP to support task parallelism stands as a first step to improve the scalability of OpenMP applications on large-scale platforms. Indeed, task parallelism usually comes with lower runtime-related overhead than thread-based approaches, allowing OpenMP programmers to create a large amount of tasks at low cost. Task parallelism also promotes the runtime system to a central role, as having more units of work to execute requires smarter scheduling decisions and load balancing capabilities.

OpenMP was recently extended to support task dependencies. Instead of explicitly synchronizing all the tasks of a parallel region at once, the application programmer can now specify a list of variables a task will read as input or write as output instead. This information is transmitted to the task scheduling runtime system. The runtime then marks a task as ready for execution only once all its dependencies have been resolved. Dependencies therefore provide a way to define finer synchronizations between tasks, able to scale better than global synchronizations on large-scale platforms. Dependencies also give the runtime system more options to efficiently schedule tasks, as these become ready for execution as soon as the data they access has been updated.

As promising as it looks however, any new feature needs proper evaluation to encourage application programmers to embrace it. While several compilers and runtime systems are now beginning to support OpenMP 4.0 task dependencies, no benchmark suite currently exists to evaluate their respective benefits and compare them to traditional task parallelism.

This paper highlights two major contributions. We first introduce a new benchmark suite to experiment with OpenMP 4.0 task dependencies. We present performance results for both the GCC/libGOMP and the CLANG[1]/libIOMP compilers and their runtime systems, comparing kernels involving either dependent or independent tasks. Secondly, we comment on the issues we met while implementing these benchmarks along the lines of current 4.0 revision of the OpenMP specification. Building on this experience, we contribute some extension proposals to the existing OpenMP specification, to improve the expressiveness of the task dependency support.

The remainder of this paper is organized as follows. Section 2 describes the task dependency programming model in OpenMP 4.0. It then analyzes the strategies adopted by GCC/libGOMP and CLANG/libIOMP to implement this model. Section 3 introduces the KASTORS benchmark suite we have designed to evaluate OpenMP 4.0's task model implementations. Section 4 presents the performance results of KASTORS using two different hardware configurations. We identify and discuss practical issues with the current OpenMP specification, and we propose extensions in section 5 to address these issues. We finally present some related works in section 6 before concluding.

2 The way OpenMP specifies dependencies between tasks

The OpenMP Architecture Review Board recently introduced a new way of expressing task parallelism using OpenMP, through the task dependencies extension that comes with revision 4.0 of the specification. This section gives some insight into how programmers can use task dependencies to advantageously replace the often excessive global task synchronizations found in many OpenMP 3.0 applications. We also study the way the development teams of GCC/libGOMP and CLANG/libIOMP choose to implement this new OpenMP task model within their compilers and associated runtime systems.

2.1 Task dependencies by the example

OpenMP 4.0 introduces the **`depend`** keyword to specify the access mode of each shared variable a task will access during its execution. Access modes can be set to

[1] Intel branch with support for OpenMP: `http://clang-omp.github.io/`

Listing 1.1: LU with independent tasks

```
 1 for (k=0; k<NB; k++) {
 2   lu0(M[k*NB+k]);
 3   for (j=k+1; j<NB; j++)
 4 #pragma omp task untied shared(M)
 5     fwd(M[k*NB+k], M[k*NB+j]);
 6 
 7   for (i=k+1; i<NB; i++)
 8 #pragma omp task untied shared(M)
 9     bdiv(M[k*NB+k], M[i*NB+k]);
10 
11 #pragma omp taskwait
12 
13   for (i=k+1; i<NB; i++)
14     for (j=k+1; j<NB; j++)
15 #pragma omp task untied shared(M)
16       bmod(M[i*NB+k],
17            M[k*NB+j],
18            M[i*NB+j]);
19 #pragma omp taskwait
20 }
```

Listing 1.2: LU with task dependencies

```
 1 for (k=0; k<NB; k++) {
 2 #pragma omp task untied shared(M)\
 3     depend(inout: M[k*NB+k:BS*BS])
 4   lu0(M[k*NB+k]);
 5   for (j=k+1; j<NB; j++)
 6 #pragma omp task untied shared(M)\
 7     depend(in: M[k*NB+k:BS*BS])\
 8     depend(inout: M[k*NB+j:BS*BS])
 9     fwd(M[k*NB+k], M[k*NB+j]);
10 
11   for (i=k+1; i<NB; i++)
12 #pragma omp task untied shared(M)\
13     depend(in: M[k*NB+k:BS*BS])\
14     depend(inout: M[i*NB+k:BS*BS])
15     bdiv(M[k*NB+k], M[i*NB+k]);
16 
17   for (i=k+1; i<NB; i++)
18     for (j=k+1; j<NB; j++)
19 #pragma omp task untied shared(M)\
20    depend(in: M[i*NB+k:BS*BS])\
21    depend(in: M[k*NB+j:BS*BS])\
22    depend(inout: M[i*NB+j:BS*BS])
23     bmod(M[i*NB+k],M[k*NB+j],M[i*NB+j]);
24 }
```

either **`in`**, **`out`** or **`inout`** whether the corresponding variable is respectively read as input, written as output or both read and written by the considered task. This information is then processed by the underlying runtime system to decide whether a task is ready for execution or should first wait for the completion of other ones.

Listing 1.1 shows the implementation of a LU factorization. It is inspired by the SparseLU kernel from the BOTS [5] benchmark suite, which generates OpenMP independent tasks. The matrix being factorized is divided into a set of smaller sub-matrices on which are applied three computation kernels, called **`fwd`**, **`bdiv`** and **`bmod`**. At iteration *k*, the update of sub-matrix *(i,j)* by the **`bmod`** function requires an update of both sub-matrices *(k,j)* and *(i,k)* from the **`fwd`** and **`bdiv`** functions respectively. For each iteration, we make sure no **`bmod`** task starts executing before tasks **`fwd`** and **`bdiv`** have completed using the broad range, explicit **`taskwait`** synchronization keyword on line 11. This keyword indiscriminately waits for the completion of every task created by the parallel section so far.

While respecting the algorithm semantics, this solution limits the potential parallelism that can be generated out of such an application. Listing 1.2 shows the same algorithm implemented with OpenMP task dependencies. Instead of waiting for the termination of *all* previous tasks before executing **`bmod`** tasks, we specify dependencies to make sure **`bmod`** tasks can execute as soon as the data they access has been updated by the corresponding **`fwd`** and **`bdiv`** tasks. Running this **`depend`** version of the program leads to the creation of a dependency graph. When an OpenMP thread turns idle, the runtime system browses this graph to decide which task should be executed next, according to tasks' current state and resolved dependencies.

Task dependency support comes with several benefits. First, task dependencies involve decentralized, selective synchronization operations that should scale better than the broad-range taskwait-based approaches. In some situations, this way of programming unlocks more valid execution scenarios than explicitly synchronized tasks, which provides the runtime system with many more valid task schedules to choose from. For example, many instances of the **fwd**, **bdiv** and **bmod** computations can legally run concurrently with the version of the LU factorization expressing task dependencies. On the contrary, this level of concurrence is not possible with the **taskwait** version because the lack of accurate dependency information leads to an over-conservative synchronization scheme. As an added benefit, information about task dependencies also enables the runtime system to optimize further, such as improving task and data placement on NUMA systems.

In expressing dependencies, the programmer needs to strike the right balance however. Indeed, dependencies expressed too coarsely might limit the amount of available parallelism. On the other hand, defining fine-grain dependencies may increase runtime-related overheads, depending on the way the underlying runtime system keeps track of the variables set inside a **depend** clause and the method it uses to enforce dependencies.

2.2 Maturity of compiler support

Both the GCC (4.9) and CLANG[2] compilers now support most of the functionalities of OpenMP 4.0. The GNU libGOMP runtime system is responsible for executing OpenMP applications compiled with GCC, while the CLANG compiler generates calls to the Intel libIOMP library. Please note that support for OpenMP 4.0 dependent tasks is still very recent in these compilers at the time of this writing. GCC 4.9 was released on April 22, 2014. The CLANG branch with Intel OpenMP support is under active development. However, even though both compilers are still maturing their support for OpenMP dependent tasks, the tests that we conducted and that we present in Section 4 show that the OpenMP dependent task support in these compilers already favorably compares to the legacy independent task support.

Compiler support for the "depend" clause The GCC 4.9 compiler stores the entire list of dependencies into a **void**** array. This array is passed as argument to the `GOMP_task` function call generated out of a **#pragma omp task** directive. It contains the addresses of all the variables referenced in the **depend** clause.

The Clang 3.4 compiler also generates a list out of the **depend** clause, and passes it to the `__kmpc_omp_task_with_deps` runtime function. This structure stores the addresses of all the variables described inside the **depend** clause, as well the length and flags of associated dependencies.

Runtime support for task dependencies The libGOMP library that comes with GCC 4.9 uses two data structures to manage task dependencies. The first data structure is the **depend** variable list generated by the compiler for the newly created task,

[2] In this article, we designate as "CLANG" the branch developed and maintained by Intel to integrate OpenMP support into the CLANG compiler. This branch is available there: `http://clang-omp.github.io/`.

as mentioned above. The second data structure is a hashtable located in the parent of the created task, which keeps track of all the pending dependencies between all the already created child tasks of the parent task. Upon creating the new child task, the runtime walks the list of **`depend`** variables and looks every variable up in the pending dependency table. If any unresolved dependency is found, the task creation enter the *deferred path*. Otherwise, the task is fully instantiated immediately.

The libIOMP used with Clang 3.4 has a very similar approach. Each parent task maintains a hashtable containing the pending dependencies for its children tasks. If a newly created task is found clear of any pending dependency, the `__kmpc_omp_task_with_deps` runtime function immediately instantiates it by calling the `__kmpc_omp_task` function. Otherwise, the new task is added as a successor to all tasks from which it expects some data, and the `__kmpc_omp_task_with_deps` function returns a code indicating that the new task has not yet been queued. The waiting new task will subsequently be woken up upon completion of its last predecessor.

3 The KASTORS suite overview

We designed the KASTORS benchmark suite to evaluate implementations of the OpenMP dependent task paradigm, introduced as part of the OpenMP 4.0 specification [11]. This section introduces the different benchmarks and describes how we extended them to express task dependencies.

Cholesky and QR decompositions from PLASMA The PLASMA [9] library developed at ICL/UTK provides a large number of key linear algebra algorithms optimized for multi-core architectures. Several implementations of each algorithm are available, using either static or dynamic scheduling. Dynamic scheduled algorithms are built on top of the QUARK [12] runtime system, which uses a data-flow dependency model to schedule tasks. The two algorithms we selected are a Cholesky decomposition and a QR decomposition, respectively known as DPOTRF and DGEQRF in PLASMA, which all operate on double precision floating point matrices (**`double`** type).

Listing 1.3: Dynamic algorithm pattern

```
wrapper_algorithm_dynamic_call(...) {
  // sequential work
  for (...)
    QUARK_Insert_Task(
         wrapper_blas_function,
         packed_parameters);
  // sequential work
  for (...)
    QUARK_Insert_Task(
        wrapper_another_blas_function,
        packed_parameters);
  // sequential work
}
```

Listing 1.4: OpenMP algorithm pattern

```
algorithm_call(...) {
    // sequential work
    for (...)
#pragma omp task depend(inout:array
     [...])
        blas_function(...);
    // sequential work
    for (...)
#pragma omp task depend(inout:array
     [...])
        another_blas_function(...);
    // sequential work
}
```

The initial implementation uses multiple levels of wrappers, packing and unpacking parameters at each level, which affects the readability of algorithms and can be error prone. Listings 1.3 and 1.4 show the initial dynamic version and the transformations we made for the OpenMP 4.0 port respectively. On the original version the `wrapper_blas_function` performs parameters unpacking before calling the actual BLAS/LAPACK routines it is built on. The OpenMP 4.0 modification led to the removal of multiple wrapper levels, thus improving code readability and maintainability, and removing the need for such parameter management.

Poisson2D This algorithm solves the Poisson equation on the unit square [0,1]x[0,1], which is divided into a grid of NxN evenly-spaced points. This benchmark relies on a 5-point 2D stencil computational kernel that is repeatedly applied until convergence is detected. We implemented two main blocked versions of this kernel, using either independent tasks and tasks with dependencies.

SparseLU This benchmark computes the LU decomposition of a sparse matrix. We modified the original BOTS implementation to express task dependencies like described on listings 1.1 and 1.2, except only non-NULL blocks are updated to adapt the traditional LU decomposition to sparse matrices.

Strassen The Strassen algorithm uses matrix decompositions to compute the multiplication of large dense matrices. Similarly to SparseLU, we modified the BOTS implementation to add parallelism to the addition part of the algorithm and express task dependencies instead of using taskwait-based synchronizations.

4 Performance Evaluation

All experiments were performed with the libGOMP library distributed with GCC 4.9 (git-mirror commit 6ed3847ffd0), and the libIOMP library distributed with clang-omp 3.4 (llvm commit 233b1e3f034, clang-omp commit 7580e521e51f).We conducted our experiments on two different NUMA configurations.

The first one holds 8 AMD Magny Cours processors for a total of 48 cores. Each core has access to 64 KB of L1 cache, 512 KB of L2 cache. Both L1 and L2 caches are private, while L3 cache is shared between the 6 cores of a processor. This configuration provides a total of 256 GB (32 GB per NUMA node) of main memory. We will refer to this configuration as **AMD48**.

The second one holds 4 Intel Xeon E5-4620 processors for a total of 32 cores. Each core has access to 64 KB of L1 cache, 256 KB of L2 cache. Both L1 and L2 caches are private, while L3 cache is shared between the 8 cores of a processor. This configuration provides a total of 380 GB (95 GB per NUMA node) of main memory. We will refer to this configuration as **INTEL32**.

Plasma The DPOTRF and DGEQRF algorithms from Plasma rely on the BLAS library. We used different versions of optimized BLAS depending on the machine: ATLAS 3.10.1 was used on INTEL32, and ATLAS 3.8.4 was used on AMD48. Measurements

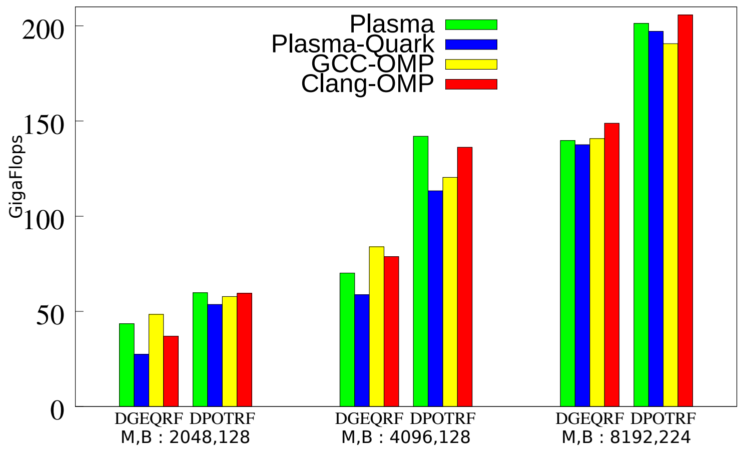

Fig. 1: Plasma on AMD48

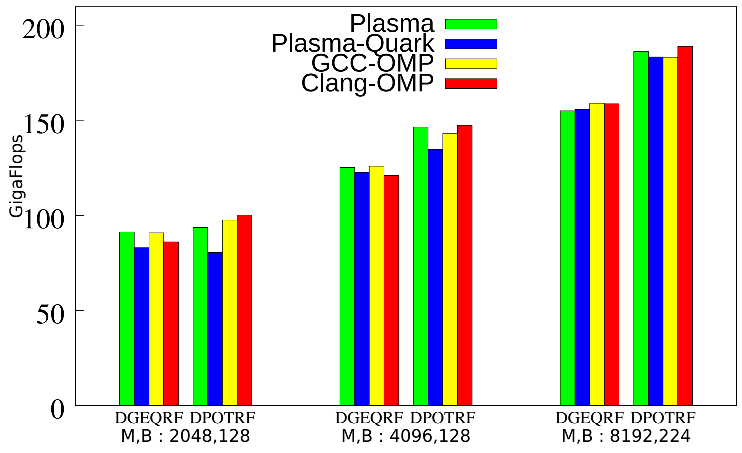

Fig. 2: Plasma on INTEL32

were conducted using the maximum number of CPUs for each machine. The performance results are expressed in gigaflops (the higher the better), from an average of 10 runs, with an average standard deviation of 2 Gflops.

We compared four versions of these algorithms: The original PLASMA implementation with static scheduling (Plasma), the dynamic scheduling implementation on top of Quark runtime (Plasma-Quark) and two KASTORS OpenMP versions compiled with GCC/libGOMP (GCC-OMP) and CLANG/libIOMP (Clang-OMP), respectively. Each version was run on the following matrix size (M) / block size (B) couples: (2048 / 128), (4096 / 128), (8192 / 224).

Table 1: Sequential Poisson2D, Strassen and SparseLU execution time (s).

Platform	Compiler	Poisson2D		SparseLU		Strassen	
		8192	16384	128x64	64x128	4096	8192
AMD48	GCC	5.28	21.34	97.74	94.36	38.35	269.93
AMD48	CLANG	5.42	21.66	99.34	95.08	32.92	234.16
INTEL32	GCC	1.64	6.53	77.88	71.97	25.32	180.11
INTEL32	CLANG	1.77	6.79	76.74	71.58	20.12	142.54

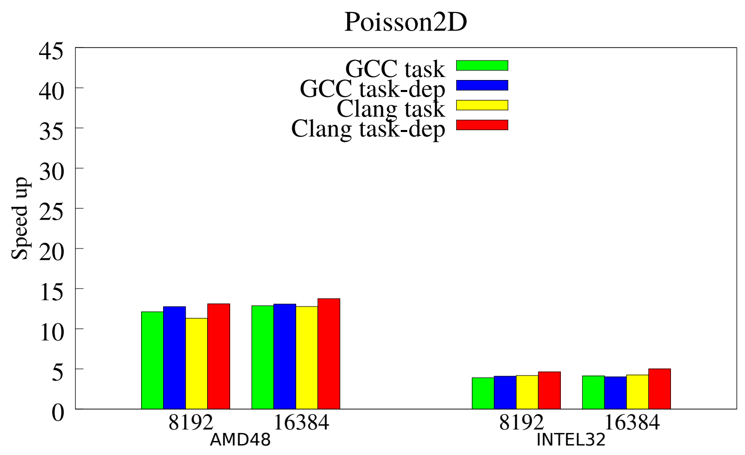

Fig. 3: Poisson 2D

For both algorithms the results are positive: On both machines the OpenMP versions compete with the original versions. In several cases, the Clang-OMP version even leads by a slight margin. The overall good results of the original static version can be explained by the fact that it does not have to pay the overhead of task creation. One can also notice that QUARK-based versions always are slightly slower than both CLANG and GCC implementations for small matrix sizes, which leads to the conclusion that the libGOMP and libIOMP runtimes induce less overhead and provide a better handling of fine-grain tasks than QUARK. The conclusion is encouraging, as we were able to get similar or better performance results using a portable OpenMP-based approach than with a specifically designed runtime.

Poisson2D The results are shown in figure 3 and the corresponding sequential times are reported in table 1. The speedup of both independent tasks and dependent tasks versions are low: Less than 14 on AMD48 and 6 on INTEL32. The application is memory bound with about the same number of arithmetic operations per load and store. The Poisson2D code is an iterative computation in which tasks are created at each time-step to update the sub domain of the initial grid. One of the important problem is that tasks are not bound to resources in order to take data locality into account. Typical scenario is that tasks between successive iterations may be performed by different threads of the same

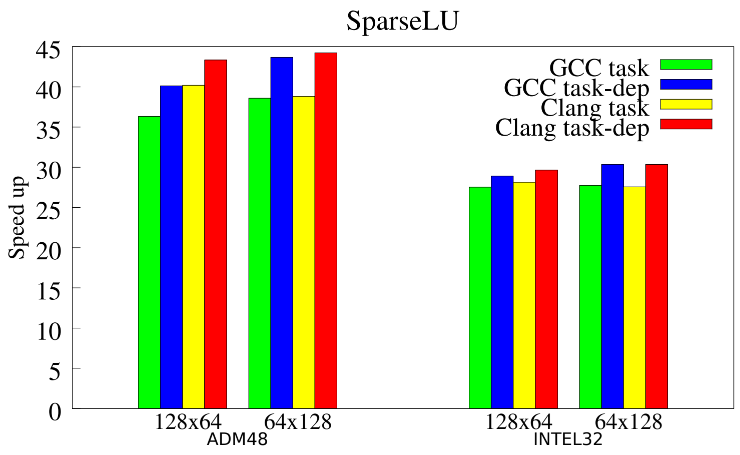

Fig. 4: SparseLU

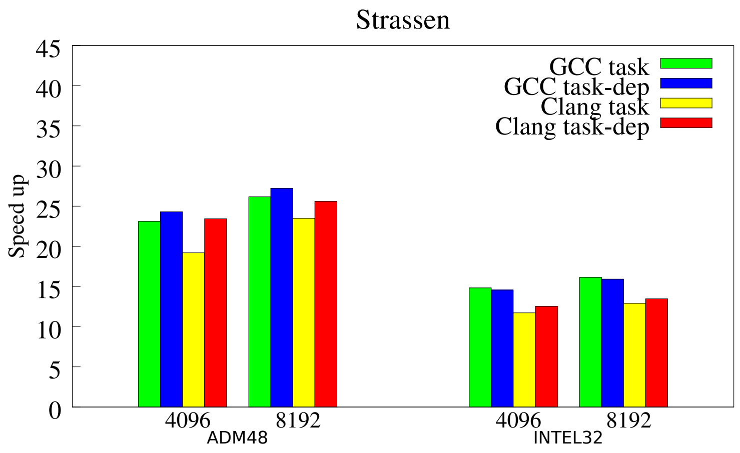

Fig. 5: Strassen

parallel region. Neither the CLANG or the GCC runtime tries to schedule tasks in order to maximize data reuse. Moreover, tasks that access to same data (due to sharing of frontiers between two sub-domains) may be better scheduled if they are mapped to cores on the same NUMA node. It is challenging for OpenMP runtime developers to take data dependencies into account in order to better schedule tasks to improve data locality.

SparseLU For this benchmark we used two matrices and sub-matrices sizes : 128x64 and 64x128. Speed up were measured from an average of 10 runs, with an average standard deviation of 0.5. Results are shown on Figure 4 and reference sequential times are listed in Table 1.

For every tested configuration, the dependent task version outperforms the independent task version for both compilers. The best speed up is achieved using the 64x128 configuration. The OpenMP 4.0 version using dependencies performs slightly better in most cases because it is able to exploit inter-iteration parallelism. The version using independent tasks cannot exploit it, because of the global synchronization steps (see listings 1.1 and 1.2). It is therefore unable to provide enough parallelism to fully benefit from the available number of cores. This confirms that using data-flow dependencies can lead to a better use of CPUs.

Strassen The Strassen matrix multiplication is typically performed in 3 steps: a matrix add, a matrix multiply and a matrix add again. The most expensive part is the multiply, but the first adds are not negligible. Thus we define dependences between these two parts. We apply a recurse cutoff of 5, and for matrix smaller than 128x128, multiplies are computed with the usual algorithm instead of Strassen. Experiments were made using two matrix/blocksize combinations: 8192/128, 16384/128. Speed up were measured from an average of 10 runs. Results are shown on Figure 5 and reference sequential times are listed on Table 1. The dependent task version very slightly outperforms the independent task version for both compilers on most test cases. We can also notice that these fine-grain dependency expressions are at the bleeding-edge of the OpenMP 4.0 implementation for both GCC and Clang, as some finer dependencies are not yet supported by these compilers.

The general conclusion of these experiments with early implementations of the new OpenMP 4.0's data-flow paradigm is that there is little reason not to make use of it for the benchmarked codes. Under the large majority of the cases tested, the new paradigm incurs no performance regression. Moreover, the finer-grain synchronization scheme it provides is able to deliver more parallelism to feed large multicore configurations, usually leading to better performance results.

5 Extending OpenMP dependency expressiveness

5.1 Enabling reductions for OpenMP tasking: the cumulative-write mode

OpenMP supports reduction for several directives such as `section`, `parallel` and **`for`**. Since the introduction of task-based programming within OpenMP 3.0, programs increasingly use loops to generate tasks, instead of performing computations directly within an **`omp`** **`for`** loop. OpenMP 4.0's task dependencies should strengthen this tendency even further, to enable more load-balancing flexibility. Unfortunately, the `reduction` clause is not available for the **`omp task`** directive, nor any alternative mechanism, leaving programmers to build it by themselves.

The approach we propose is to extend the set of **`in`**,**`out`** and **`inout`** data access modes with a new *cumulative-write* mode **`cw`**. This new mode indicates that several tasks may contribute to a piece of data in any order (but not simultaneously). It is inspired by the reduction support implemented in runtime system Kaapi [7]. One might suggest to use the existing **`inout`** mode instead. However, the **`inout`** mode imposes a strong ordering between tasks, that proves to be unnecessary in that case. The following example illustrates this idea with a modified version of the LU decomposition code from Fig. 1.2. Using a *cumulative write* mode **`cw`** for the **`bmod`** computation (line 23) breaks the strong inter-iteration dependency that was caused by the **`inout`** mode, thus enabling more parallelism.

```
      #pragma omp task untied shared(M) \
           depend(in: M[i*NB+k:BS*BS]) \
           depend(in: M[k*MSIZE+j:BS*BS]) \
           depend(cw: M[i*NB+j:BS*BS])
      bmod(M[i*NB+k], M[k*NB+j], M[i*NB+j]);
```

On the next input or inout dependency, or at the next synchronization point (taskwait, explicit or implicit barrier), the runtime ensures that variables previously accessed using the **`cw`** mode may then be read. The reduction operator can be a pre-defined operators or a user defined reduction operator such as introduced with OpenMP-4.0. We plan to extend reduction operators to enable non-commutative operators as well (such as appends on a string or matrix multiplication) such as in [7] or [1].

5.2 Expressing dependencies on non-contiguous memory areas

OpenMP 4.0 makes it possible to express task dependencies on the slice of an array, by specifying the offset and the length of the dependent slice. These slices specifications can be chained to express dependencies on 2D-subarrays.

```
#pragma omp task depend(inout:array[offset1:length1][offset2:length2])
        blas_function(...);
```

Chained array slices are available only for constant arrays and variable length arrays, however. The lack of leading dimension information makes them impractical for pointers, which strongly limits their interest. Temporarily copying data to constant arrays is tedious and time consuming. Variable length arrays on the other hand, are passed by value (copy) from a function to another, and only integrated the C language standard since C99. However, VLA is an optional feature in C11. A possible workaround could be to cast the pointer to a variable length array or constant array, such as this:

```
double *f_ = malloc(NX * NY * sizeof(double));
double (*f)[NX][NY] = (double (*)[NX][NY])f_;
#pragma omp task depend(inout:f[offset1:length1][offset2:length2])
        blas_function(...);
free(f_);
```

This solution is not very much elegant, however. Our proposal is thus to extend the slice syntax to enable the specification of the missing leading dimension information. Following this extension, 2D sub-arrays may be specified such as this:

```
double *f = malloc(NX * NY * sizeof(double));
#pragma omp task depend(inout:f[offset1:length1:NX][offset2:length2])
        blas_function(...);
free(f);
```

6 Related work

OpenMP benchmarks To the best of our knowledge, KASTORS is the only benchmark suite focusing on OpenMP task dependencies. This section describes existing benchmarks evaluating other aspects of the OpenMP specification.

The Barcelona OpenMP Tasks Suite [5] evaluates the addition of tasking to the OpenMP 3.0 specification, comparing several ways of generating tasks out of small computing kernels. We adapted two of these kernels, *SparseLU* and *Strassen*, to express task dependencies, and we added the modified versions to our KASTORS suite.

Older OpenMP benchmark suites such as PARSEC [3], SPECOMP [10] and Rodinia [4] could be extended to benefit from task parallelism, as well as the NAS Parallel Benchmark suite [2] (NPB). NPB was originally introduced to evaluate the performance of parallel supercomputers. It was later extended to provide OpenMP 2.5 compatible versions of most kernels. We consider modifying some of them to exploit task parallelism, especially the "multi-zone" ones [8] involving nested parallelism.

Compilers and runtime systems supporting task dependencies KAAPI [7][3], StarPU [1][4], Quark [12] and OMPSs [6] are libraries that support task dependencies in a different context than OpenMP. We present them here focusing on the different access modes they propose, and the way application programmers can provide task dependencies using these libraries.

The KAAPI and StarPU runtime systems were designed to improve the performance of task-based parallel applications on large-scale heterogeneous platforms. Unlike the current OpenMP specification, KAAPI and StarPU support the expression of dependencies on multi-dimensional arrays. StarPU provides ways of splitting a data with user-defined filters that can help to express dependencies on sub-matrices, for example.

The Quark runtime system that comes with the PLASMA library is responsible for executing tasks created out of BLAS operators in a dynamic way. Unlike KAAPI and StarPU, Quark only considers unidimensional arrays, but comes with an original `scratch` access mode to reuse thread-specific temporary data.

OMPSs [6] is a programming model inspired by OpenMP with specific directives to support task dependencies and heterogeneity. The OMPSs programming model includes the `concurrent` clause to express that tasks will perform reductions on the listed variables. Similar access-modes also exist in KAAPI and StarPU.

7 Conclusion

The introduction of task dependencies in the revision 4.0 of the OpenMP specification provides application programmers with a new, powerful way of describing synchronizations in task-based parallel applications. OpenMP compilers and runtime systems will play a key role in the adoption of such a new feature by the community, as they are responsible for tracking and resolving variable dependencies before executing tasks.

[3] `http://kaapi.gforge.inria.fr`
[4] `http://starpu.gforge.inria.fr`

The performance of OpenMP applications expressing task dependencies will be closely related to how efficiently compilers and runtime systems implement this new feature.

We introduced a new benchmark suite, called KASTORS, composed with small kernels ported on the OpenMP dependent task model. We then compared their performance with taskwait-based versions of the same kernels for both the GCC/libGOMP and the CLANG/libIOMP compilers and runtime systems. While the support for task dependencies in these compilers is still very recent, experiments run on two different hardware configurations show that the versions with task dependencies offer performances comparable and sometimes better to taskwait-based versions on most kernels. The results obtained by rewriting some PLASMA kernels to use OpenMP task dependencies also demonstrated that this model even outperforms dedicated data-flow solutions in some situations, while improving portability.

In the near future, we plan to extend the KASTORS suite with new benchmarks coming from either modified versions of existing benchmarks or small kernels inspired from real-life scientific applications. Based on CLANG, we are also developing a source-to-source compiler that generates direct calls to the KAAPI or StarPU runtime systems out of C/C++ OpenMP programs. We are currently working on releasing this compiler with a full support of the extensions proposed in section 5.

Acknowledgments

This work has been partially supported by the INRIA ADT K'STAR Project, the ANR-11-BS02-013 HPAC Project and the IRSES2011-295217 HPC-GA Project.

References

1. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.
2. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Report RNR-94-007, Department of Mathematics and Computer Science, Emory University, March 1994.
3. C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
4. S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, Dec 2010.
5. A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 124–131. IEEE, 2009.
6. A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
7. T. Gautier, X. Besseron, and L. Pigeon. Kaapi: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO'07*, 2007.

8. H. Jin and R. F. V. der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. In *IPDPS*. IEEE Computer Society, 2004.
9. J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra. *Multithreading in the PLASMA Library*, pages 119–141. Chapman and Hall/CRC, 2013.
10. M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran. Spec omp2012 – an application benchmark suite for parallel systems using openmp. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 223–236, Berlin, Heidelberg, 2012. Springer-Verlag.
11. OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013.
12. A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.