Introducing Moldable Tasks in OpenMP

Pierre-Étienne Polet, Ramy Fantar, Thierry Gautier

▶ **To cite this version:**

Pierre-Étienne Polet, Ramy Fantar, Thierry Gautier. Introducing Moldable Tasks in OpenMP. IWOMP 23 - International Workshop on OpenMP, Sep 2023, Bristol, UK, United Kingdom. pp.51-65, 10.1007/978-3-031-40744-4_4 . hal-04409117

HAL Id: hal-04409117

https://hal.science/hal-04409117v1

Submitted on 22 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Introducing moldable tasks in OpenMP

Pierre-Étienne Polet[1,2], Ramy Fantar[2], and Thierry Gautier[1]

[1] Inria, CNRS, ENS de Lyon, UCBL, LIP, Lyon, France
[2] Thales DMS, 06560 Valbonne, France

Abstract. This paper introduces a new approach to handle implicit parallelism in library functions. If the library already utilizes a third-party programming model like OpenMP, it may run in parallel. Otherwise, if the library remains sequential, OpenMP directives in client code cannot be used for direct parallelization. To express implicit parallelism and, in the meanwhile, dynamically adjust the parallel degree of a task when its starts, we propose to use moldable tasks. We handle this by introducing a new construct called `taskmoldable` that generates multiples tasks from a single function call and an iteration space. For the Lapack Cholesky factorization algorithm, our `taskmoldable` directive allows simple code annotation to express parallelism between tiles and improves programmability. Performance results on a beamforming application indicates that our moldable implementation is slightly faster by 5% in mean, than a parallel execution achieved with Intel MKL.

Keywords: Moldable task · OpenMP Task · Task Dependency

1 Introduction

The task programming model promotes seamless collaboration between application programmers and library developers, ensuring functional composition regardless of their respective choices during development. A work stealing scheduler from Cilk [6] has undergone theoretical analysis to provide guarantees for expected parallel time and space. A provable OpenMP-3.0 task scheduler should inherit these same guarantees. Furthermore, task models with dependencies have been subject to analysis within the same framework [14,30], allowing for the application of similar theoretical results to the OpenMP-4.0 dependent task model.

Although theoretical results have provided satisfactory findings, they are limited to a rigid task model that lacks consideration for the physical parallelism of the target machine during task creation. However, task management, including creation and scheduling, incurs overhead that significantly affects application performance. To mitigate this, researchers have proposed solutions such as lightweight task implementations (e.g., Cilk [6] for independent tasks, Kaapi [15,7]. for data flow dependencies), task throttling [1], high-performance work queue data structures for scalability [3,18], and caching task graph construction for multiple iterations [15,33].

Parallel programs typically involve more arithmetic operations and memory accesses compared to their sequential counterparts. Although these extra overheads don't fundamentally alter the asymptotic number of operations, they do reduce practical efficiency. To mitigate these additional operations, it is crucial to align the parallelism level of the application with the hardware's degree of parallelism. It is important to note that the aforementioned solutions do not address these supplementary costs.

OpenMP efficiently adapts application parallelism to hardware through worksharing constructs. Parallel worksharing loops are automatically distributed among the threads in the current parallel region, ensuring balanced iteration space. The taskloop construct generates the right number of tasks based on the parallel region's size, limiting arithmetic overheads.

This paper proposes a new task generating construct for expressing hidden implicit parallelism in library functions. It allows to define *moldable tasks*, which can adapt their parallel degree to available resources, based on established theoretical scheduling concepts [29,20].

The following section motivates our proposal, focusing on concrete case studies. Section 3 provides detailed information about the `taskmoldable` directive and its key clauses. We then demonstrate the usage of the new directive in a classical Cholesky factorization and a beamforming application [16,28], concluding the presentation.

2 Motivation

A recently proposed linear algebra API [10] defined *batched API* to process a set of independent linear algebra subroutine calls on small matrices, with "*the aim of providing more efficient, but portable, implementations of algorithms on high-performance manycore architectures* [10]." They were present in commonly used APIs such as Nvidia cuBLAS, Magma [17] or Intel MKL.

A call to perform `batch_count` matrix multiplications on a set of input data is[3]:

```
1 gemm_batch(m,n,k,A,B,C,bc);
```

where each parameter is an array of size `bc`, the batch count, of the required parameter to call the BLAS `gemm` kernel, *i.e.* the call is equivalent to:

```
1 for (int i=0; i<bc; ++i)
2   gemm(m[i],n[i],k[i],A[i],B[i],C[i]);
```

The `gemm_batch` operation performs `bc` independent calls to the `gemm` BLAS subroutine where each `gemm` works with different parameters and data provided in the effective array parameters. Because the for loop resides inside the code body of the BLAS batch function it was not accessible to parallelize calls to `gemm_batch` using any OpenMP worksharing directives or the taskloop

[3] For simplicity, we omit some parameters such as the operations on matrices (transposition...), alpha and beta assumed to be 1, the leading dimensions or the info error parameter which are required to pass arguments to each underlying `gemm` kernel.

generating task construct. Our proposal aims at providing an OpenMP construct to expose the *implicit loop*, and its iterations, as a task generating construct.

The application developer knows that the `gemm_batch` is equivalent to executing the above implicit iteration loop. Thus the iterations may be partitioned in a disjoint set of N intervals $I_k = [b_k, e_k[$ such that $\cup_{k=0}^{N-1} I_k = [0, bc-1]$, such that the batched `gemm` calls could be rewritten to:

```
1 for (int i=0; i<N; ++i)
2   gemm_batch(m+b_k[i],n+b_k[i],k+b_k[i],A+b_k[i],B+b_k[i],
3     C+b_k[i], e_k[i]-b_k[i]);
```

Therefore, the `gemm_batch` operation can be viewed as a list homomorphism [5]. Given a list l, a concatenation operator $\#$, and a function f assumed to be a list homomorphism, we can transform the call $f(l1\#l2)$ into calls to $f(l1) \oplus f(l2)$, where $\oplus$ is a reduction operator. In our case, the function f represents the structured block outlined following the OpenMP directive.

Overview of moldable task. The two main issues are defining the implicit loop iteration space (*e.g.* `bc`) and passing the effective parameters to the subcalls. We propose annotating the code with a new directive **`taskmoldable`** used to inform that the following structured block has an implicit loop to partition. The size of the iteration is specified by the clause **`batch_count`** and the transformation of effective parameters to the parameters of the subsequences calls is specified by the clause **`access`**.

```
1 #pragma omp taskmoldable access(linear: m,n,k,A,B,C) \
2                          batch_count(bc)
3 gemm_batch(m,n,k,A,B,C,bc);
```

In this example, the transformation is of kind **`linear`**, that is a default mapping function, that applies $M_X : i \to X + i$ to any variable X listed in the clause `access` to pass the parameter on the partition i. We called such transformation a *mapping function*. It could be defined by the user.

`syrk`: a list homomorphism with reduction. The proposed clause can be applied to list homomorphisms that involve reduction. Figure 1 illustrates a common call to the `syrk` subroutine in the left-looking Cholesky factorization [25], as seen in the Lapack netlib `potrf`. `syrk` computes $T = T - A \times A^T$ with T symmetric. The moldable task has dependence types `in` on `A` and `inout` on `T`. The computation $T = T - A \times A^T$ is equal to $T = T - \sum_i A_i \times A_i^T$ where A_i is the i-th tile of size $nb \times nb$ (except the last tile) as depicted in Fig. 1.

The moldable task in Fig. 1 expresses the fact that the call to `syrk` is a list homomorphism with respect to matrix A_i starting at position `A+i*nb` from `A`: This is an access `strided{nb}` with our proposal. Matrix `T` is fully accessed by all calls to `syrk`. It was possible to keep the original `inout` dependence-type, but to keep the possibility to reorder the accumulation depending on the predecessor tasks releasing the matrix bloc A_i the clause specifies that the dependence type on T expressed by generated task is `mutexinoutset`. Thus by

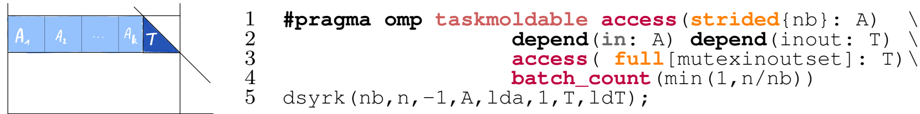


Fig. 1: Left: SYRK $T = T - A \times A^T$ in left looking version of the Cholesky factorization. Right: annotation with `taskmoldable`.

accumulating the sequence of `syrk` calls on each A_i to the matrice T, we obtain the result.

The code that generates the explicit tasks from Fig. 1 is equivalent to the following, where `bc` is the batch count generated from the clause `batch_count`.

```
1 for (int i=0; i< bc; ++i)
2   #pragma omp task depend(in: A+i*nb) depend(mutexinoutset: T)
3   dsyrk(nb,n,-1,A+i*nb,lda,1,T,ldT);
```

The `taskmoldable` directive enables the expression of internal parallelism within library functions, including batched or non-BLAS subroutines. By reducing programming efforts, code annotations can result in highly parallel task-based programs. The next section focuses on presenting the `taskmoldable` directive and its associated clauses, which facilitates the extraction of more parallelism. Section 4 provides detailed accounts of three comprehensive case studies: the application of `taskmoldable` to matrix-matrix multiplication (`gemm`), the sequential Lapack left-looking Cholesky factorization [25] and a beamforming application [16,28].

3 A new directive: taskmoldable

As the taskloop directive [31,22], the **taskmoldable** directive is a *task generating construct*. It enhances the functionality of the taskloop directive by allowing the capturing of (implicit) parallel loops within any structured block through user annotations and parameter passing rules to generated tasks.

3.1 General structure

The general structure of the directive is the following:

```
1 #pragma omp taskmoldable batch_count(<counter-list>) \
2     access( <data-mapping> [{args}] [<dependence-type>] : \
3         <list-item> )\
4     depend( <dependence-type> : <list-item> )\
5     <data-sharing attribute>\
6     num_tasks( <integer-list> ) | grainsize( grain-size-list)
7 {<structured block>}
```

When a thread encounters a `taskmoldable` construct, it creates an explicit task that partitions the implicit iterations defined by `batch_count` into chunks, each of which is assigned to an explicit task for parallel execution. Each chunk has an identifier from 0 to the maximal number of chunks - 1. The size of the chunk is computed before creating the explicit task. The data environment of

each generated task is created according to the data-sharing attribute clauses on the `taskmoldable` construct, per-data environment ICVs.

The clause `access` is used to translate the variables to be passed to each explicit task. The effect is as if each variable in the list-item appearing in the structured block is rewritten by applying the data mapping function on the chunk id. The `data-mapping` is either a predefined identifier: `linear`, `strided{<integer expr>}` or `full`; or a user-defined identifier. Section 3.3 presents the data mapping function. Optionally, the clause `access` can specify the dependence-type of the generated tasks expressed on the variable.

Clause `batch_count` accepts a list of integers that are associated to an implicit nested iteration loops. For instance, `batch_count`($C_0, C_1, ..., C_{k-1}$) is associated with the implicit nested loops generating the tasks as illustrated in the following code. As for the taskloop directive, clauses `num_tasks` and `grainsize` limit the number of tasks generated at runtime. For `taskmoldable` directive, their parameters are a list of values applied on each loop of the nest.

3.2 Compilation

The compiler rewrites the `taskmoldable` directive to a code equivalent to the following skeleton:

```
1  #pragma omp task depend( weak-dependency-type: <list-item> )
2  {
3    _kmpc_omp_taskmoldable_size( C_0, .., C_{k-1},
4        num_tasks, grainsize, S_0, .., S_{k-1} );
5    for (int i_0=0; i_0<C_0; i_0+=S_0)
6      for (int i_1=0; i_1<C_1;i_1+=S_1)
7        ...
8          for (int i_{k-1}=0; i_{k-1}<C_{k-1}; i_{k-1}+=S_{k-1})
9          #pragma omp task depend( <inherited> )
10         {<structured block. Variables of the 'access' list-item
11           have been replaced by the mapping function called with
12           (i_0,i_1,...,i_{k-1},S_0,S_1,...,S_{k-1}) as effective parameters>}
13 }
```

The moldable task is created with dependencies using the weak variant [24,14] of the dependency type used in the `depend` clauses: e.g. a `depend( inout: A )` in the `taskmoldable` definition is translated to `depend(weak-inout: A)`. The objective is to postpone real dependencies on the child tasks (because those are making real memory accesses and computation) rather than to the moldable task which only creates tasks.

Then, the task calls the runtime function `_kmpc_omp_taskmoldable_size` to compute the size of the tasks in each dimension S_0, S_1 ..., S_{k-1} from the sizes of the `batch_count` clause (dimension C_0, C_1, ..., C_{k-1}) and the values passed in clause `num_tasks` or `grainsize`.

3.3 Data mapping functions

A data mapping function is associated with an item using the `access` clause of the `taskmoldable` directive:

```
access( mapping_id [{<args>}] [<dependence-type>]: list-item )
```

The optional `dependence-type` argument is presented in the next section dealing with the expression of dependencies on generated tasks. The runtime defines the subset of the initial workload for each task. It provides a tuple `start` = $(i_0, i_1, ..., i_{k-1})$ that defines the beginning of the sub-iteration space it should process. The `access` clauses provide information to get the right data for each task. To do so the user provides a function called *mapping_id* where the declaration is defined as follows:

```
F(item,batch_count,start,args...)
```

which is used to replace `items` from `item-list` each time it appear in the `structured block`.

We propose three basic mappings, `strided{args}` that take as argument a stride on each dimension, `linear` that assumes the data are linearly spaced and `full` that assume all task work on the same data. They are defined as follows:

strided(A,bc,start,strides) -> A + $\sum_{u=0}^{|bc|-1} i_u$ * strides[u]
linear(A,bc,start) -> A + $\sum_{u=0}^{|bc|-1} i_u * \prod_{v=0}^{u-1}$ bc[v]
full(A,bc,start) -> A

Items from `list-item` are expected to be pointers of types that allow pointer arithmetic.

An implementation of the `linear` *mapping_id* could be the following:

```
template<T> T* linear(T* A, int* bc, int* starts, int dim_count)
{
    int pos = 0; int size = 1;
    for(int u = 0; u < dim_count; u++)
    {
        pos += starts[u];
        size *= bc[u]
    }
    return A+pos;
}
```

The runtime tries to decompose the computation into N tasks where N is either provided by the clause `num_tasks`, or by default, computed automatically: The dimensions of the split are even and computed by the runtime function `_kmpc_omp_taskmoldable_size`. Preliminary experimental results reported in section 4.3 show that our proposition of compilation of moldable tasks could be applied to heterogeneous architectures.

3.4 Data dependencies

Task generating constructs often require implicit synchronization to ensure the correctness of parallel executions. However, relaxing these synchronization requirements can enable better utilization of hardware resources. Sharing this goal, a recent proposal to enhance OpenMP, as described in [22], suggests extending the `depend` clause to include the `taskloop` construct.

The same issue appears with the `taskmoldable` directive. The main difference is that iteration loops are hidden from the annotation thus the expression of dependencies on generated task is different. Thanks to the data mapping function we are able to replace the item and the mapped item through the function, as defined in the above section. In that way, the generated task inherits the dependencies from the `depend` clause in the `taskmoldable` directive but on items rewritten by the data mapping function.

However, the programmer may optionally refine the way generated tasks declare dependencies through the `access` clause. This is illustrated in Fig. 1 where the generated tasks declare `mutexinoutset` dependencies on item `T` while the moldable task has declared dependency type `inout` on it. The interest here is to allows commutativity on the reduction operated by `syrk` in case of faster resolution of dependencies on the A_i.

We assume the availability of the weak-dependencies [24] to postpone real dependencies on the moldable task to the generated tasks. Without them, it is possible to inline the execution of the moldable task creation of the task generating tasks: at runtime, the `taskmoldable` directive is translated to the code that directly generates the explicit tasks in place of creating the task (that will generate the explicit tasks).

3.5 Implementation

We have created a customized runtime specifically designed to handle moldable tasks, allowing us to validate a prototype before integrating it into the LLVM OpenMP runtime. Thanks to our previous development work in the LLVM runtime, we have taken care to ensure an easier merge process.

The task entry point follows the code outlined in section 3.2. To create a moldable task, we utilize the runtime function `__kmpc_omp_task_moldable`, which extends `__kmpc_omp_task_withdeps` to include task dependencies and additional mapping functions. These are stored in supplementary fields of the task data structure `kmp_task_t`.

We have also incorporated support for partitioning moldable tasks between CPUs and GPUs. If the user provides a GPU version of the list homomorphism function, the runtime divides the implicit iteration between CPUs and GPUs. Initial experiments regarding this feature are reported in section 4.3. We briefly discuss how to integrate moldable tasks with targets in the following perspective.

The runtime should select a granularity for each moldable task, our implementation relies on previous executions of similar functions to compute an expected performance for each worker, then it creates one task for each worker that handles a subset of the moldable task proportional to its performance. Tasks running on GPU targets may be further split by the runtime to fit the memory constraints of the co-processor.

4 Evaluation

The two next sections illustrate our `taskmoldable` directive with two decompositions of the `gemm` matrix product, and how to produce a highly parallel

(a) Row/Column decomposition (b) Block decomposition

Fig. 2: GEMM block decomposition.

Cholesky factorization from the sequential code of the Lapack netlib library. Then we present a beamforming application with performance evaluation.

4.1 Gemm decomposition

The `strided{<optional-args>}` mapping function is well-suited when all data are stored in an array by spacing each consecutive partition with a constant stride. As an example, we will show how to generate two classical decomposition of `gemm` with the `taskmoldable` directive:

```
1 #pragma omp taskmoldable batch_count(m,n)\
2     access(strided{lda,0}:A)\
3     access(strided{0,1}:B)\
4     access(strided{ldc,1}:C)\
5     depend(inout:C) depend(in:A,B)
6 gemm( m, n, k, A, B, C );
```

The result of this decomposition, in case of even split, is presented as figure 2a. It can generate up to $m \times n$ independent tasks. In this case, the block decomposition only works with two dimensions thus the input matrices are split by row or by column but not in blocks. To allows full block decomposition we should handle dependencies between tasks and work on the third dimension of the `gemm`. The following code result in decomposition 2b. We describe dependency management in 3.4.

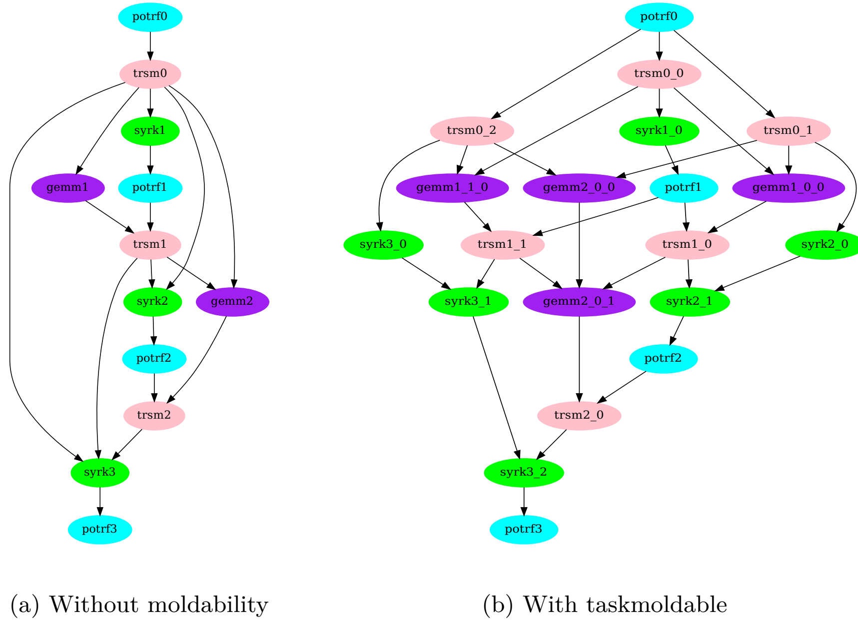
```
1 #pragma omp taskmoldable batch_count(m,n,k)\
2     access(strided{lda,0,1}:A)\
3     access(strided{0,1,ldb}:B)\
4     access(strided{ldc,1,0}[mutexinoutset]:C)\
5     depend(inout:C) depend(in:A,B)
6 gemm( m, n, k, A, B, C );
```

4.2 Lapack Cholesky factorization

The Cholesky factorization algorithm is used in signal processing algorithms such as adaptive beamforming [13], it is also commonly studied in dependency graph generation. We worked on the block left-looking version of the algorithm which is implemented in the subroutine `portf` in Lapack[4], the associated code is sketched below[5]:

[4] https://netlib.org/lapack/

[5] Code is rewritten in C to follow the guideline of the paper



(a) Without moldability (b) With taskmoldable

Fig. 3: Left looking Cholesky task graph with N/NB = 4

```
1  cholesky( N, NB, A, lda ):
2    for( j = 0; j < N; j += NB )
3    {
4      #pragma omp task depend(in:A[0][j]) depend(inout:A[j][j])
5      syrk( NB, j, -1, A[0][j], lda, 1 A[j][j], lda )
6      #pragma omp task depend(inout:A[j][j])
7      potrf( NB, A[j][j], lda )
8      if( j + NB < N )
9      {
10         #pragma omp task depend(in:A[0][j],A[0][j+NB])\
11             depend(inout:A[j][j+NB])
12         gemm( NB, N - j - NB, j, -1, A[0][j], lda, A[0][j+NB],
13             lda, 1, A[j][j+NB], lda )
14         #pragma omp task depend(in:A[j][j])\
15             depend(inout:A[j][j+NB])
16         trsm( NB, N - j - NB, 1, A[j][j], lda, A[j][j+NB], lda )
17     }
18   }
```

At each iteration, it updates a group of `NB` columns with their definitive values. A graph of tasks generated with this code is provided as Figure 3a. Whereas the code is elegant and relatively simple, it does not express a lot of parallelism.

By seeing the calls to `syrk`, `gemm` and `trsm` as moldable tasks and adapting the granularity of the split we can achieve the same level of parallelism as a right-looking implementation, the dependency graph is provided as Figure 3b. Furthermore, if we make the granularity finer, as presented in section 4.1, more parallelism can be generated on `gemm` and `trsm` function calls. The code is provided below:

```
1  cholesky( N, NB, A, lda ):
2    for( j = 0; j < N; j += NB )
3    {
4      #pragma omp taskmoldable batch_count(j) depend(in:A[0][j])
5          depend(inout:A[j][j]) access(strided{1}:A[0][j]) \
6          access(full[mutexinoutset]:A[j][j])
7      syrk( NB, j, -1, A[0][j], lda, 1 A[j][j], lda )
8      #pragma omp task depend( inout: A[j][j] )
9      potrf( NB, A[j][j], lda )
10
11     if( j + NB < N )
12     {
13         #pragma omp taskmoldable batch_count((NB,N-j-NB,j))\
14              depend(in:A,B) depend(inout:C) \
15              access(strided{0,1,lda}:A[0][j])\
16              access(strided{0,1,lda}:A[0][j+NB])\
17              access(strided{lda,1,0}[mutexinoutset]:A[j][j+NB])
18         gemm( NB, N-j-NB, j, -1, A[0][j], lda, A[0][j+NB], lda,
19                 1, A[j][j+NB], lda )
20         #pragma omp taskmoldable batch_count(N-j-NB)\
21              depend(in:A[j][j]) depend(inout:A[j][j+NB])\
22              access(strided{1}:A[j][j+NB])
23         trsm( NB, N - j - NB, 1, A[j][j], lda, A[j][j+NB], lda )
24     }
25   }
```

The key point here is that original code can be annotated with OpenMP directives[6] for parallelizing compared to restructuring algorithms to exploit parallelism between tiles [9]. This was possible thanks to advanced features such as dependencies between arrays [8,23].

4.3 Case of study: beamforming

We implement a beamforming algorithm design to work on rectangular arrays of sensors using only moldable tasks, the algorithm is composed of three main parts. First sensor data are converted from the temporal domain to the frequency domain with FFT1D. Then we apply dephasing coefficients to each input to compute the beams, thanks to the shape of the array we can decompose this step in two consecutive matrix multiplications. The final step is to convert complex values to energy by computing the absolute value of each element. Due to data pattern restrictions in the used libraries, we insert a transposition step between the FFT and the matrices multiplications. The pseudo-code is provided below, the values of parameters `stride_x` are user-defined, constant and depend on the in-memory representation of each array.

```
1  beamforming():
2    #pragma omp taskmoldable batch_count(fft_count) \
3        depend(in:Sensor_t) depend(out:Sensor_f)\
4        access(strided{fft_stize}:Sensor_t,Sensor_f)
5    fft1DExecBatch( fft_size, Sensor_t, Sensor_f, fft_count )
6
7    #pragma omp taskmoldable batch_count( fft_count )\
8        depend(in:Sensor_f) depend(out:Sensor_f2)\
9        access(strided{fft_size}:Sensor_f)\
10       access(strided{1}:Sensor_f2)
11   transpose( Sensor_f, Sensor_f2 )
```

[6] Here we have presented C pragma directive - we also assume Fortran compatible directive

```
12
13    #pragma omp taskmoldable batch_count(gemm_0_count)\
14        depend(in:Sensor_f2,Dephase_x) depend(out:Pseudo_beam)\
15        access(strided{stride_S},Sensor_f2)\
16        access(strided{stride_X},Dephase_x)\
17        access(strided{stride_P},Pseudo_beam)
18    gemmStrideBatch( Sensor_f2, Dephase_x, Pseudo_beam, gemm_0_count )
19
20    #pragma omp taskmoldable batch_count(gemm_1_count)\
21        depend(in:Pseudo_beam,Dephase_y) depend(Beam)\
22        access(strided{stride_P}:Pseudo_beam)\
23        access(strided{stride_Y}:Dephase_y)\
24        access(strided{stride_B}:Beam)
25    gemmStrideBatch( Pseudo_beam, Dephase_y, Beam, gemm_1_count )
26
27    #pragma omp taskmoldable batch_count(abs_count)\
28        depend(in:Beam) depend(out:Energy)\
29        access(strided{1}:Beam,Energy)
30    abs( Beam, Energy, abs_count )
```

This code was executed on our custom runtime, it ran on a workstation with Intel Xeon 8253, 16 cores processor. It ran about 1M FFT of size 4K, 4K square GEMM of size 1024 and 256M abs values at each iteration. The implementation uses Intel MKL sequential on Intel CPU and OpenBlas on AMD ones, cuBlas and cuFFT are used for Nvidia GPUs.

Overhead of task managements : On this beamforming benchmark, at each iteration of the time step loop, the code generates 40 `taskmoldable` constructs decomposed into 4062 tasks. The number of dependencies is 103086 between all the tasks. We measure a mean creation cost of 360µs per moldable task; 4µs per task; and 140ns per dependency. Our moldable task runtime does not fit well with the analysis in [27] because the task granularity is not a free execution parameter: It is fixed by the runtime according to available resources and their performances. When more resources are used for the execution, the moldable tasks are decomposed into more finer tasks.

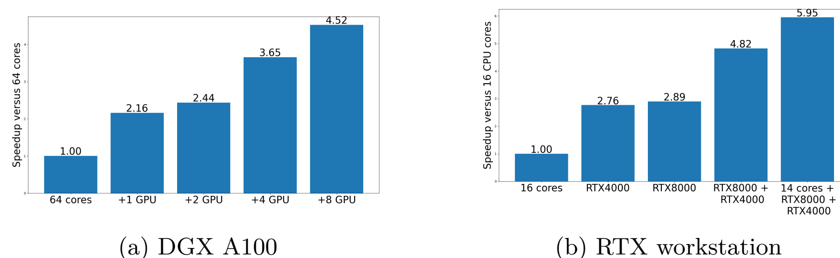
Moreover, we compare the performances of the moldable implementation using MKL sequential and a classical one using MKL parallel library. We find out that the moldable implementation is 5% faster, in mean, than the classical one, thus the overhead implied by the moldability and our runtime is negligible for this workload. Mean execution times over 100 iterations for different core count are provided in table 1.

#core:	1	2	4	8	12	16
MKL parallel: (s)	15.9	8.7	4.4	2.3	1.7	1.3
Moldable (s):	15.3	7.8	4.0	2.0	1.7	1.2
Delta (%):	4	10	9	13	0	8

Table 1: Beamforming execution times

By adding two different RTX GPUs to the workstation, we show that the same code can scale on heterogeneous platforms, it implies to allow a task to

execute different code for each target and to handle memory with the runtime. Results of executions speed by adding GPUs are provided in Figure 4 with results on a DGXA100 server. On the RTX platform, CPU-only execution ran in 28s, it was 14.7 times faster than sequential execution and 2.89 times slower than heterogeneous execution. On the DGXA100, CPU-only execution ran in 9s, we reach a speedup of 4.5 by adding GPUs.



(a) DGX A100 (b) RTX workstation

Fig. 4: Beam-forming speedups

5 Related work

Tasks based runtimes are used to schedule tasks on the fly. There are multiple runtimes available as OpenMP [12], StarPU [4], OMPSS [11], Kaapi [15] or PaRSEC [19]. Those runtimes aim to schedule dependent tasks with a lack of knowledge about task computational cost and without knowing tasks that will be scheduled in the future. None of them offer a moldable task concept that allows the expression of functions as a set of tasks. This criterion is absent in the classification of [32]. In [2] the authors theoretically analyze the upper bound on performance using an assumption that tasks are moldable without support in StarPU used by their application. Several moldable task schedulers are proposed and analyzed in scheduling literature [29,20] with *ad hoc* simulation or experimentation without any runtime support of moldable tasks.

The OpenMP task concept exists and had been extended to provide task creation from loop structures using taskloop [31], moreover, recent contributions open the path to data dependencies between tasks from different taskloops without the need for a global synchronization [22]. Furthermore, [21] provides a structure that allows OpenMP tasks to run inner loops as worksharing constructs, and [26] extensions allow more control over the parallelism generated inside a task that calls library code that uses openMP tasks. OpenMP does not provide a syntax to exploit the implicit parallel structure of library functions.

`taskmoldable` is a task generating directive. At runtime, it creates an explicit task that postpones real dependencies to its child tasks. Athapascan-1 runtime [14] allows this passing rule with the *postponed access mode*. Similar features are recently proposed under the term *weak-dependencies* in OMPSS [24].

With the absence of weak-dependencies it is always possible to directly create child tasks with an anticipated decision to decompose the moldable task.

6 Conclusion and perspectives

The `taskmoldable` directive provides a means for users to leverage hidden parallelism in a function without sacrificing the performance of library-specific implementations or requiring an extensive restructuring of the function's internal design. Our evaluation and examples demonstrate how this directive enables the extraction of parallelism from a sequential Cholesky implementation. Furthermore, we achieve minimal overhead when handling moldable tasks in domain-specific workloads like beamforming, allowing us to compete with an MKL implementation.

Our ongoing research aims to expand the scope of our preliminary experimental results, encompassing CPUs and GPUs. This extension will enable users to annotate code, constructing performance models that guide the sizing of partitions. Additionally, we are exploring the integration of target clauses into the directive, facilitating heterogeneous computations based on moldable tasks.

Another future direction involves exploring how to express the moldability of more complex moldable code structures beyond nested loops. For instance, we aim to enable the perception of the entire Cholesky factorization as a moldable task.

Acknowledgements: Experiments presented in this paper were carried out using the Grid'5000 testbed (see https://www.grid5000.fr), supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.

References

1. Agathos, S.N., Kallimanis, N.D., Dimakopoulos, V.V.: Speeding up openmp tasking. In: European Conference on Parallel Processing (2012)
2. Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., Takahashi, T.: Task-based fmm for heterogeneous architectures. Concurrency and Computation: Practice and Experience **28**(9), 2608–2629 (2016)
3. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems **34**(2), 115–144 (2001)
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In: European Conference on Parallel Processing. pp. 863–874. Springer (2009)
5. Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design. pp. 5–42. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
6. Blumofe, R.D., Leiserson, C.E.: Space-efficient scheduling of multithreaded computations. SIAM J. Comput. **27**, 202–229 (1998)

7. Broquedis, F., Gautier, T., Danjean, V.: Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In: Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World. p. 102–115. IWOMP'12, Springer-Verlag, Berlin, Heidelberg (2012)
8. Bueno, J., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Implementing ompss support for regions of data in architectures with multiple address spaces. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. pp. 359–368. ICS '13, Association for Computing Machinery, New York, NY, USA (2013)
9. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing **35**(1), 38–53 (2009)
10. Dongarra, J., Duff, I., Gates, M., Haidar, A., Hammarling, S., Higham, N.J., Hogg, J., Valero-Lara, P., Relton, S.D., Tomov, S., et al.: A proposed api for batched basic linear algebra subprograms (2016)
11. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel processing letters **21**(02), 173–193 (2011)
12. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of openmp task scheduling strategies. In: International Workshop on OpenMP. pp. 100–110. Springer (2008)
13. Fuhrmann, D.R., San Antonio, G.: Transmit beamforming for mimo radar systems using partial signal correlation. In: Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004. vol. 1, pp. 295–299. IEEE (2004)
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. p. 88. PACT '98, IEEE Computer Society, USA (1998)
15. Gautier, T., Besseron, X., Pigeon, L.: Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation. p. 15–23. PASCO '07, Association for Computing Machinery, New York, NY, USA (2007)
16. Guerreiro, A.M.G., Neto, A.D.D., Lisboa, F.: Beamforming applied to an adaptive planar array. In: Proceedings RAWCON 98. 1998 IEEE Radio and Wireless Conference (Cat. No. 98EX194). pp. 209–212. IEEE (1998)
17. Haidar, A., Dong, T., Tomov, S., Luszczek, P., Dongarra, J.: Framework for batched and gpu-resident factorization algorithms to block householder transformations. In: ISC High Performance. Springer, Springer, Frankfurt, Germany (07-2015 2015)
18. Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing. p. 280–289. Association for Computing Machinery, New York, NY, USA (2002)
19. Hoque, R., Herault, T., Bosilca, G., Dongarra, J.: Dynamic task discovery in parsec: A data-flow task-based runtime. In: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems. ScalA '17, Association for Computing Machinery, New York, NY, USA (2017)
20. Marchal, L., Simon, B., Sinnen, O., Vivien, F.: Malleable task-graph scheduling with a practical speed-up model. IEEE Transactions on Parallel and Distributed Systems **29**(6), 1357–1370 (2018)
21. Maronas, M., Sala, K., Mateo, S., Ayguade, E., Beltran, V.: Worksharing tasks: An efficient way to exploit irregular and fine-grained loop parallelism. In: 2019

IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC). pp. 383–394. IEEE (2019)
22. Maroñas, M., Teruel, X., Beltran, V.: Openmp taskloop dependences. In: OpenMP: Enabling Massive Node-Level Parallelism: 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14–16, 2021, Proceedings 17. pp. 50–64. Springer (2021)
23. Paek, Y., Hoeflinger, J., Padua, D.: Efficient and precise array access analysis. ACM Trans. Program. Lang. Syst. **24**(1), 65–109 (jan 2002)
24. Perez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the integration of task nesting and dependencies in openmp. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 809–818 (2017)
25. Rothberg, E.E., Gupta, A.: An evaluation of left-looking, right-looking and multifrontal approaches to sparse cholesky factorization on hierarchical-memory machines. Int. J. High Speed Comput. **5**, 537–593 (1991)
26. Scogland, T.R., Sunderland, D., Olivier, S.L., Hollman, D.S., Evans, N., de Supinski, B.R.: Making openmp ready for c++ executors. In: OpenMP: Conquering the Full Hardware Spectrum: 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11–13, 2019, Proceedings 15. pp. 320–332. Springer (2019)
27. Slaughter, E., Wu, W., Fu, Y., Brandenburg, L., Garcia, N., Kautz, W., Marx, E., Morris, K.S., et al.: Task bench: A parameterized benchmark for evaluating parallel runtime performance. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15. IEEE (2020)
28. Somasundaram, S.D.: Wideband robust capon beamforming for passive sonar. IEEE Journal of Oceanic Engineering **38**(2), 308–322 (2012)
29. Sun, H., Elghazi, R., Gainaru, A., Aupy, G., Raghavan, P.: Scheduling parallel tasks under multiple resources: List scheduling vs. pack scheduling. In: 2018 IEEE International Parallel and Distributed Processing Symposium. pp. 194–203 (2018)
30. Tchiboukdjian, M., Gast, N., Trystram, D.: Decentralized list scheduling. Annals of Operations Research **207**(1), 237–259 (2013)
31. Teruel, X., Klemm, M., Li, K., Martorell, X., Olivier, S.L., Terboven, C.: A proposal for task-generating loops in openmp. In: OpenMP in the Era of Low Power Devices and Accelerators: 9th International Workshop on OpenMP, IWOMP, Canberra, ACT, Australia, September 16-18, 2013. Springer (2013)
32. Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., et al.: A taxonomy of task-based parallel programming technologies for high-performance computing. J. Supercomput. **74**(4), 1422–1434 (apr 2018)
33. Yu, C., Royuela, S., Quiñones, E.: Enhancing openmp tasking model: Performance and portability. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) OpenMP: Enabling Massive Node-Level Parallelism. pp. 35–49. Springer International Publishing, Cham (2021)