Compas'2020 : Parallélisme / Architecture/ Système/ Temps Réel
MILC - Lyon, France, du 30 juin au 3 juillet 2020

Automatic task-based parallelization of C++ applications by source-to-source transformations

Garip Kusoglu, Berenger Bramas, Stephane Genaud

CAMUS Inria Nancy – Grand Est, Strasbourg University, ICPS ICube,
300 bd Sébastien Brant, 67412 Illkirch - France
Garip.Kusoglu@inria.fr

Résumé

Currently, multi/many-core CPUs are considered standard in most types of computers including, mobile phones, PCs or supercomputers. However, the parallelization of applications as well as refactoring/design of applications for efficient hardware usage remains restricted to experts who have advanced technical knowledge and who can invest time tuning their software. In this context, the compilation community has proposed different methods for automatic parallelization, but their focus is traditionally on loops and nested loops with the support of polyhedral techniques. In this study, we propose a new approach to transform sequential C++ source code into a task-based parallel one by inserting annotations. We explain the different mechanisms we used to create tasks at each function/method call, and how we can limit the number of tasks. Our method can be implemented on top of the OpenMP 4.0 standard. It is compiler-independent and can rely on external well-optimized OpenMP libraries. Finally, we provide preliminary performance results that illustrate the potential of our method.

Mots-clés : Task-based, Compilation, Parallelization, Object-oriented.

1 Introduction

High-performance computing (HPC) is crucial for making advances and discoveries in numerous domains [28]. While computers and supercomputers are becoming more powerful, they are also becoming increasingly more complex and heterogeneous. The efficient utilization and programmability of such systems are ongoing research topics and remain reserved for experts who master all the technological layers and who can fine-tune their target applications to specific hardware. Indeed, using CPUs or accelerators at an efficiency close to their theoretical peak performance is a tedious challenge especially for many-core CPUs like the Intel KNL or the Sunway SW26010 [22], that make up the Sunway TaihuLight cluster, the fastest supercomputer in the world from 2016 to 2018.

This complexity has been defined as the programmability wall [14]. To overcome this challenge and provide optimized applications to non-experts, the research community has covered multiple facets to provide methods and tools to benefit from the power of modern hardware. On the one hand, the HPC community has proposed several parallelization paradigms, such as the task-based methods. This paradigm has gained popularity because it allows parallelizing with an abstraction of the hardware by delegating task distribution and load balancing to dynamic schedulers. It is a convenient solution that works well on both homogeneous and heterogeneous computing nodes. It has proven its potential on numerous computational applications [2, 3, 13, 29, 24, 13, 1, 4, 25]. However, this method requires a high degree of expertise and significant programming efforts.

On the other hand, the compilation community has been creating automatic parallelization and optimization methods focusing mainly on loops with the support of the polyhedral mathematical foundation [16]. In practice, these methods work well if the time-consuming parts of code

are isolated and centralized in loops that can be parallelized with a fork-join strategy. However, many applications are not adapted because they use abstraction mechanisms, have small loops, use indirection or have loops that cannot be analyzed statically. Therefore, its impact on real applications and the adoption by non-experts remain limited.
In this context, we propose a method that bridges the gap between automatic parallelization and task-based method. Our method consists in transforming a C++ source code by inserting annotations to obtain a parallel application, as an HPC expert developer would do.
The contributions of the current study are :

— introduce our compiler APAC and present the mechanisms to apply the task-and -dependency method in automatic parallelization for C++ ;
— describe different strategies to generate the OpenMP source code to control the number of tasks ;
— provide a performance study that illustrates the benefit of our method.

The paper is organized as follows. In Section 2, we provide the background concerning the task-based method and a state of the art on automatic parallelization. Then, in Section 3, we describe our method and the different mechanisms in action. Section 4 includes the performance study. Finally, we conclude in Section 5 and provide an extensive discussion in Section 6.

2 Background

2.1 Task-based programming (sequential task flow)

The HPC community has proposed and promoted the task-based method for parallelizing scientific applications. With this method, an application is transformed into a graph where each node is a task and each edge a dependency. Several runtime systems have been conceived on top of this paradigm [17, 26, 20, 10, 31, 7, 12], each with its features and interface [30]. Underneath, they implement the paradigm using a task-graph (also called task-dependency graph) where the nodes of the graph represent operations while the edges represent the dependencies between them. The graph materializes either within the application itself, e.g. the application uses a graph data structure to model its logic, or it materializes implicitly at runtime within a task runtime system as it is the case with the sequential task flow (STF) model. In the STF model, a sequential code is annotated or split into tasks that are submitted to the runtime system by a single thread. Together with the tasks, the thread provides the parameters and the type of data accesses - "read" or "(read-)write" - allowing the runtime system to generate the dependencies and to ensure that the parallel execution remains equivalent to the sequential one.

2.2 OpenMP

OpenMP (Open Multi-Processing) [15] is an API that was originally dedicated to the programming of shared memory multiprocessing and which is now extended to heterogeneous computing. It is designed for C/C++ and Fortran and is supported by most compilers, such as GNU (gcc/g++), Clang, Intel compilers, etc. To use OpenMP, one simply needs to annotate a source code using a *pragma*. Its portability and simplicity make it a de facto standard in the development of HPC scientific applications. For legacy reasons, it is expected that the source code would still be valid even if the compiler does not support OpenMP and omits the *pragma*. OpenMP supports the task-based method since version 4 [11] by letting the programmer indicate the data accesses for each task. OpenMP supports two synchronization mechanisms to wait for the completion of tasks. The first approach is to declare a *taskgroup* scope. A thread generates some tasks inside the *taskgroup* and will leave the *taskgroup* scope only once all the inserted tasks and their descendants will have completed their execution. The second approach is the *taskwait* statement that a thread can pass only when all the tasks it has inserted (but not necessarily their descendants) are terminated.

2.3 Automatic parallelization related work

Automatic parallelization has been investigated since the '90s to help programmers using multicore CPUs [8]. State-of-the-art systems mostly focus on the automatic parallelization of affine loops [9, 27, 18] by considering that the workload is concentrated in loop-blocks on native data types. For instance, the AutoPar-Clava compiler [6, 5] uses OpenMP annotations to parallelize C code. Meanwhile, automatic task-based parallelization has also been investigated but with less investment. The OoOJava compiler [21] offers a semi-automatic parallelization process. When using OoOJava, it is required that programmers add annotations in their source code to indicate which portions of the code should be considered as tasks. The compiler is in charge of finding dependencies and ensuring the sequential coherency, which relieves the programmers from a significant effort. Similarly, the @PT compiler [23] allows parallelizing an application by inserting few annotations inside JAVA codes. Both OoOJava and @PT appear efficient and provide a way for the programmer to give information that the compiler could not know otherwise. However, they still require programmers to modify their code to make it parallel.
The JPar compiler [19] is very similar to what we want to achieve. This project consists of a Java compiler that analyzes the AST to find the data accesses and to infer task boundaries. It enables only those tasks that contain a sufficient number of instructions in their bodies. However, the compiler is designed for JAVA, which is not among the languages privileged by the high-performance community. It also has many differences with the C++ language and those, such as the garbage collector, impact the mechanisms used for gaining automatic parallelization. This compiler uses the "future" concept to model the dependencies whereas we want to use the sequential task-based programming model.
Some attempts have also been made to use a high-level language to automatically generate task-based C++ code, such as NT^2 (Numerical Template Toolbox) a specific DSL similar to Matlab's language [32]. However, the specificity of the language and its lack of standardization limit its adoption by end-users and requires complete re-writing of existing software.

3 Source-to-source transformations for automatic task-based parallelization

3.1 Principles

APAC is a C++ source-to-source compiler based on clang-tools v8 and the LibTooling library from the Clang subproject of the LLVM framework. It takes as input standard C++ source code and generates C++ source code with annotations to enable task-based parallelism. The LibTooling library makes it convenient to apply source code modifications and transformations. Therefore, in the current implementation, we have decided to avoid working on the Clang AST. Instead, we use LibTooling's rewriter feature to commit changes directly to the source code buffer.

3.2 Task creation and data access detection

APAC applies two main transformations to the input code. It encapsulates the body of each function into an OpenMP *taskgroup*, and each method/function call into an OpenMP *task* statement. However, the calls to methods/functions from the standard library are not encapsulated into tasks, and APAC avoid adding a *taskgroup* when a function does not create any task. With this approach, most calls become asynchronous but a function will return only when all the tasks it has created (and their sub-tasks) have completed.

When creating a task, the compiler makes explicit the data-dependencies through the `depend` clause. First, it qualifies the parameters passed in the call either as *read* or *write* variables. For that purpose, the following rule is applied : if the parameter type (callee side) is *const* or passed by value, then the depend is *in*. Otherwise, the parameter type is a non-*const* reference/pointer and the depend clause is *inout*. The argument type (caller side) does not matter.
We provide in Code 1a an example of parallelization with APAC. At the beginning of the *main* function, we create the parallel section at line 20. Then, a task is created at line 26 that includes

the type of data access.
Second, if the call returns a result assigned to a variable, this variable is also added to the *inout* list. A particular case is when this statement is used to declare and initialize at the same time the variable getting the call result. In that case we need to split the statement into its declaration and initialization. If the declaration qualifies the variable as *const*, this property must be removed. This is illustrated in Code 2a which is transformed into Code 2b.

3.3 Dependency expression based on previous tasks' results

A function/method call may pass an element of an array as an argument where the index of the element is the output of a previous task. For instance, if a task read-write *i* and the next task accesses *a[i]*, we must make sure that when the *depend* clause is evaluated the value of *i* is known. Therefore, in this situation, we put a *taskwait* before the second task, which ensures a correct dependency evaluation at the cost of constraining the degree of parallelism.

3.4 Scope local declaration of variables

The sequential code may contain declarations of variables inside a scope or in the body of an *if/else*, *switch*, *while* or *for* statements. These variables are destroyed at the end of the scope where they are declared. If such a declaration is simply an alias (pointer or reference) to a variable declared outside the scope, nothing has to be done and the variable will be passed as *firstprivate* to the task. Otherwise, the compiler creates a variable in the heap and uses a reference on the pointed element inside the task. The allocated element is deleted by an extra task at the end of the corresponding scope. By doing so, we ensure that variables are alive in the tasks and only destroyed when no other task wants to use it. For example, see line 2 of Code 3b, which was originally Code 3a.

3.5 Synchronizations to enforce coherency

We currently transform only the calls into tasks. Therefore, the use of variables directly inside the function, between calls, are not protected by the task mechanism. This is why accessing variables requires enforcing access coherency. For instance, if a variable is accessed by a task and is also directly accessed later in the same *taskgroup*, we need to put a *taskwait* if both accesses cannot be done concurrently (which happens when both accesses are in *write*, or one in *read* and the other in *write*). An example of this situation is given in Code 4b.

3.6 Management of returned value after task-encapsulation

A function/method can have multiple *return* statements in its body. These *return* statements cannot simply be put in the tasks as we have to make sure that the correct value is returned when we transform the source code. To address this problem, we put a single *return* statement after the *taskgroup* and a goto-label just before closing the *taskgroups*, as shown in Code 5b. Then, each original *return* is transformed into a *taskwait*, followed by the assignment of the only variable that will be returned, and a jump to the goto-label at the end of the *taskgroups*.

3.7 Limiting the number of tasks

The files generated by APAC will have *taskgroup* inside each function/method and tasks at each call. However, each task implies an overhead and thus the number of tasks should be controlled. Ideally, the granularity should be taken into account to avoid creating tiny tasks, but we do not support this feature currently. To control the number of tasks we use two different strategies that will set a boolean variable at the beginning of each *taskgroup*. This boolean is passed to each task to decide if it is inserted and made parallel or if it is executed directly by the thread that manages *taskgroups*. We must use the same boolean for all the tasks in a *taskgroup* because it will be undefined behavior to have only some of the tasks in parallel and some other computed by the thread that manages the *taskgroups*.

3.7.1 Maximum number of existing tasks

This strategy sets the activation boolean to true if the number of existing tasks is lower than a given number at the creation of the *taskgroups*. We have to maintain a global counter, which

is incremented before each task (in the *taskgroup* and just before the task statement) and decremented at the end of each task (inside the task). If the activation boolean is set to false the global counter remains unchanged in the current *taskgroups*.

3.7.2 Maximum parallel depth

This strategy allows us to control the call level above which the parallelism becomes disabled. The main function is considered as level zero and the activation boolean is set to true if the current depth is lower than a given number. To implement this mechanism, we use a thread private counter that will allow transferring the depth between *taskgroups*, function calls, and tasks. More precisely, at the beginning of the *taskgroup*, the thread in charge copies its private counter into a local variable. Then, it passes this local variable as *firstprivate* to the tasks of the *taskgroups*. Finally, when a thread starts a task, it copies the *firstprivate* counter + 1 into its private one and uses it in the *taskgroup* it may create in the current task. We use a thread private counter such that each branch of the tree has its own counter and allows each thread to know its current depth.

4 Results

4.1 Configuration

Software and backend

APAC can use several runtime systems as backend. In the current study, we focus on OpenMP that will be in charge of managing the tasks/dependencies. It is important to understand that OpenMP is an API that can be implemented in many ways and thus performances are not portable. For instance, because we declare several layers of *taskgroups* and tasks, the libgomp used by GNU compilers constraints the parallel execution as it appears that only one thread will be involved executing tasks from a nested *taskgroups*. This is why it is mandatory to use Clang++-8 currently and its libomp OpenMP implementation to have parallelism. During the executions, we bind the threads using *OMP_PROC_BIND=true*.

Hardware

We perform the tests on two different configurations :

- PC : Intel i7-8550U CPU at 1.80GHz with caches of sizes 32/256/8192 KB, 16 GB of RAM and 4 cores.
- WS : Intel Xeon Gold 6126 CPU at 2.60GHz with caches of sizes 32/1024/19712 KB, 187 GB of RAM and 20 cores.

4.2 Test-cases

We assess our method on two test cases [1] :

- Quicksort : this application sorts an array of 100 million integers. The values are randomly generated and the application ends with a test to ensure the array is sorted. We limit the activation of the tasks after a depth of 5.
- Molecular-Dyn : this application computes interactions between 10.000 particles distributed in a grid of dimension 5^3. The simulation executes 100 iterations, where one iteration consists of the computation of interactions between closed neighbors and the displacement of the particles.

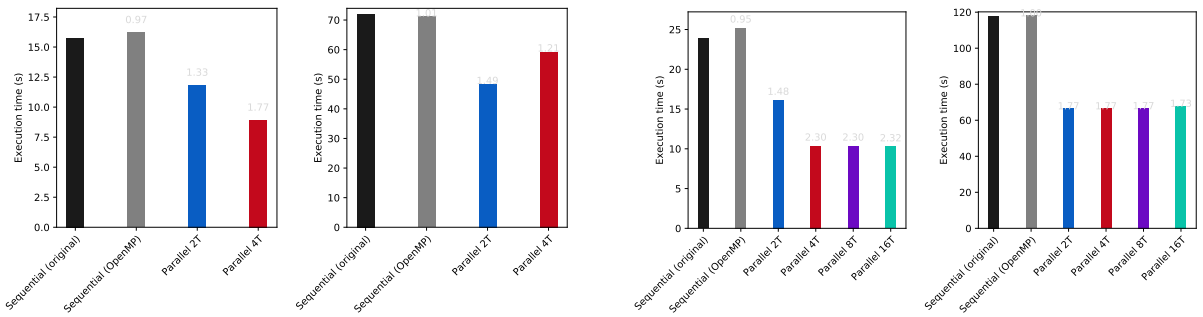
The execution time is the median of 10 executions measured with the Linux *time* command. Therefore, it includes all the overhead of the parallelization system and the portions of code that have not been parallelized.

4.3 Results

Figure 1 provides the results for both configurations and test cases. The execution time is given for the original version, the parallel version executed in sequential and the parallel version executed in parallel. We observe that for the Quicksort, Figures 1a and 1c, the parallel version

1. The original and the transformed source code are available at `https://mybox.inria.fr/d/8fdb415c6583465eb72a/`. It is not needed to create an account but a valid email will be required.

executed in sequential suffers from a small overhead. However, this overhead vanishes for the Molecular-Dyn application, Figures 1b and 1d.
Concerning the parallel executions, APAC provides a speedup for the Quicksort up to 4 threads on the PC configuration and up to 16 threads on the WS configuration. For the Molecular-Dyn application, the parallel efficiency is quickly bounded and there is no improvement with more than 2 threads on the PC and with more than 4 threads on the WS. This limitation comes from the fact that we measure the elapsed time of the whole execution and because there is an overhead that remains significant due to the code's taskification at places where execution will finally be sequential.
However, the results are promising because we can parallelize codes that were not designed to be executed in parallel and to provide a significant speedup compared to the sequential execution.



(a) Quicksort PC (b) Molecular-Dyn PC (c) Quicksort WS (d) Molecular-Dyn WS

FIGURE 1 – Execution details for the two test cases on PC and WS configurations. The speedup against the sequential execution is given above each bar.

5 Conclusion

We have presented the APAC compiler conceived to automatically parallelize sequential programs. We have described the different mechanisms implemented in APAC that allow us to extract data dependencies and to produce OpenMP compliant source code. This includes mechanisms for managing the number of tasks or the depth of parallelization manually. We evaluated our approach on two classical computational problems, and demonstrated that APAC enables a significant speedup.

6 Discussion and Perspective

The path to creating a fully automatic and efficient task-based parallel system will require leveraging several challenges that we plan to address. Firstly, the limitation of the number of tasks is currently manual in APAC and must be set by the user, which requires an understanding of how the code is actually parallelized. Therefore, the final system must provide an automatic strategy. Secondly, we should evaluate the granularity of tasks statically or build an incomplete granularity model that would be completed at runtime to decide when to enable the tasks. We should also aggregate tasks that are contiguous if there are dependencies between them, aggregate tasks if the granularity of a single task seems inefficient or avoid creating tasks if there is only one task in a *taskgroup* section. Thirdly, we must detect range accesses but this cannot be done by considering each value in the range as a distinct dependency element. If a function works on an array of millions of elements, we cannot declare millions of dependencies for the task. That is why we will need to investigate if the dependency interval should be used or if macro-dependency is sufficient. Fourthly, we should transform portions of code that are not calls into tasks. Finally, we should detect potential cases that could lead to speculative execution.

Bibliographie

1. Agullo (E.), Aumage (O.), Bramas (B.), Coulaud (O.) et Pitoiset (S.). – Bridging the gap between openmp and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, n10, Oct 2017, pp. 2794–2807. 1
2. Agullo (E.), Bramas (B.), Coulaud (O.), Darve (E.), Messner (M.) et Takahashi (T.). – Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing*, vol. 36, n1, Feb 2014, pp. C66–C93. 1
3. Agullo (E.), Bramas (B.), Coulaud (O.), Darve (E.), Messner (M.) et Takahashi (T.). – Task-based fmm for heterogeneous architectures. *Concurrency and Computation : Practice and Experience*, vol. 28, n9, Dec 2015, pp. 2608–2629. 1
4. Agullo (E.), Buttari (A.), Guermouche (A.) et Lopez (F.). – Task-Based Multifrontal QR Solver for GPU-Accelerated Multicore Architectures. – In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 54–63, Dec 2015. 1
5. Arabnejad (H.), Bispo (J.), Cardoso (J. M. P.) et Barbosa (J. G.). – Source-to-source compilation targeting openmp-based automatic parallelization of c applications. *The Journal of Supercomputing*, Dec 2019. 3
6. Arabnejad (H.), Bispo (J. a.), Barbosa (J. G.) et Cardoso (J. a. M.). – Autopar-clava : An automatic parallelization source-to-source tool for c code applications. – In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, *PARMA-DITAM '18*, PARMA-DITAM '18, p. 13–19, New York, NY, USA, Jan 2018. Association for Computing Machinery. 3
7. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, vol. 23, n2, 2011, pp. 187–198. 2
8. Banerjee (U.), Eigenmann (R.), Nicolau (A.) et Padua (D. A.). – Automatic program parallelization. *Proceedings of the IEEE*, vol. 81, n2, Feb 1993, pp. 211–243. 3
9. Bastoul (C.). – Efficient code generation for automatic parallelization and optimization. – In *Proceedings of the Second International Conference on Parallel and Distributed Computing*, *ISPDC'03*, ISPDC'03, p. 23–30, USA, 2003. IEEE Computer Society. 3
10. Bauer (M.), Treichler (S.), Slaughter (E.) et Aiken (A.). – Legion : Expressing locality and independence with logical regions. – In *International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 66. IEEE Computer Society Press, 2012. 2
11. Board (O. A. R.). – Openmp application program interface, jul 2013. 2
12. Bramas (B.). – Increasing the degree of parallelism using speculative execution in task-based runtime systems. *PeerJ Computer Science*, vol. 5, mars 2019, p. e183. 2
13. Carpaye (J. M. C.), Roman (J.) et Brenner (P.). – Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping. *Journal of Computational Science*, vol. 28, 2018, pp. 439–454. 1
14. Chapman (B.). – The multicore programming challenge. – In Xu (M.), Zhan (Y.), Cao (J.) et Liu (Y.) (édité par), *Advanced Parallel Processing Technologies*, pp. 3–3, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 1
15. Chapman (B.), Jost (G.) et Pas (R. v. d.). – *Using OpenMP : portable shared memory parallel programming*. – Cambridge, The MIT Press, 2008, *Scientific and engineering computation series*. 2
16. Clauss (P.) et Loechner (V.). – Parametric analysis of polyhedral iteration spaces. *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 19, n2, 1998, pp. 179–194. 1
17. Danalis (A.), Bosilca (G.), Bouteiller (A.), Herault (T.) et Dongarra (J.). – PTG : An abstraction for unhindered parallelism. – In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, (WOLFHPC), IEEE*, pp. 21–30. IEEE Press, 2014. 2

18. Ferrandi (F.), Fossati (L.), Lattuada (M.), Palermo (G.), Sciuto (D.) et Tumeo (A.). – Automatic parallelization of sequential specifications for symmetric mpsocs. – In Rettberg (A.), Zanella (M. C.), Dömer (R.), Gerstlauer (A.) et Rammig (F. J.) (édité par), *Embedded System Design : Topics, Techniques and Trends*, pp. 179–192, Boston, MA, 2007. Springer US. 3
19. Fonseca (A.) et Cabral (B.). – Controlling the granularity of automatic parallel programs. *Journal of Computational Science*, vol. 17, 2016, pp. 620 – 629. – Recent Advances in Parallel Techniques for Scientific Computing. 3
20. Gautier (T.), Lima (J. V.), Maillard (N.) et Raffin (B.). – XKaapi : A runtime system for data-flow task programming on heterogeneous architectures. – In *2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1299–1308. IEEE, 2013. 2
21. Jenista (J. C.), Eom (Y. H.) et Demsky (B.). – Ooojava : an out-of-order approach to parallel programming. – In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pp. 11–11. USENIX Association, 2010. 3
22. Liu (C.), Xie (B.), Liu (X.), Xue (W.), Yang (H.) et Liu (X.). – Towards efficient spmv on sunway manycore architectures. – In *Proceedings of the 2018 International Conference on Supercomputing*, *ICS '18*, ICS '18, p. 363–373, New York, NY, USA, 2018. Association for Computing Machinery. 1
23. Mehrabi (M.), Giacaman (N.) et Sinnen (O.). – Annotation-based parallelization of java code. – In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 775–784, May 2017. 3
24. Moustafa (S.), Kirschenmann (W.), Dupros (F.) et Aochi (H.). – Task-based programming on emerging parallel architectures for finite-differences seismic numerical kernel. – In Aldinucci (M.), Padovani (L.) et Torquati (M.) (édité par), *Euro-Par 2018 : Parallel Processing*, pp. 764–777, Cham, 2018. Springer International Publishing. 1
25. Myllykoski (M.) et Mikkelsen (C. C. K.). – Introduction to starneig–a task-based library for solving nonsymmetric eigenvalue problems. *arXiv preprint arXiv :1905.04975*, 2019. 1
26. Perez (J. M.), Badia (R. M.) et Labarta (J.). – A dependency-aware task-based programming environment for multi-core architectures. – In *2008 IEEE International Conference on Cluster Computing*, pp. 142–151. IEEE, 2008. 2
27. Ramon-Cortes (C.), Amela (R.), Ejarque (J.), Clauss (P.) et Badia (R.). – AutoParallel : A Python module for automatic parallelization and distributed execution of affine loop nests. – In *PyHPC 2018 - 8th Workshop on Python for High-Performance and Scientific Computing*, Dallas, TX, United States, novembre 2018. 3
28. Requena (S.), (GENCI) et al. – Extreme data and computing initiative, oct 2016. https ://exdci.eu/sites/default/files/public/files/d3.1_0.pdf. 1
29. Sukkari (D.), Ltaief (H.), Faverge (M.) et Keyes (D.). – Asynchronous task-based polar decomposition on single node manycore architectures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, n2, Feb 2018, pp. 312–323. 1
30. Thoman (P.), Dichev (K.), Heller (T.), Iakymchuk (R.), Aguilar (X.), Hasanov (K.), Gschwandtner (P.), Lemarinier (P.), Markidis (S.), Jordan (H.) et et al. – A taxonomy of task-based parallel programming technologies for high-performance computing. *J. Supercomput.*, vol. 74, n4, avril 2018, p. 1422–1434. 2
31. Tillenius (M.). – Superglue : A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM Journal on Scientific Computing*, vol. 37, n6, 2015, pp. C617–C642. 2
32. Tran Tan (A.), Falcou (J.), Etiemble (D.) et Kaiser (H.). – Automatic task-based code generation for high performance domain specific embedded language. *International Journal of Parallel Programming*, vol. 44, n3, Jun 2016, pp. 449–465. 3

Appendix

```
void a_function(const int a, const int* b, int& c){
    /* This function does not perform any call */
    /* Therefore, no taskgroup is added to it */
    ... Some code ...
}
void a_function_with_call(const int a, const int* b, int& c){
    /* This function has one call */
    /* So, a taskgroup contains its body */
    #pragma omp taskgroup
    {
        /* Each call becomes a task */
        #pragma omp task depend(in:a,b) depend(inout:c) default(shared)
        {
            a_function(a,b,c);
        }
    }
}
int main(){
    /* The parallel region is declared in the main */
    #pragma omp parallel
    #pragma omp master
    #pragma omp taskgroup
    {
        int a; int *b; int c;
        /* Each call becomes a task */
        #pragma omp task depend(in:a,b) depend(inout:c) default(shared)
        {
            a_function_with_call(a,b,c);
        }
    }
    return 0;
}
```

(a) Automatic parallelization code.

Code 1 – Example of transformation - automatic parallelization. The compiler first creates the parallel region in the *main* followed by a *taskgroup*. Then, each call is put into a task, and the arguments are marked as *in* or *inout*. All the OpenMP statements of this example have been added by APAC.

```
int x;
const int y = f(x);
```

(a) Original code.

```
int x;
int y;
#pragma omp task depend(in:x) depend(inout:y) default(shared)
{
    y = f(x);
}
```

(b) Transformed code.

Code 2 – Example of transformation - Assignment. The compiler removes the *const* qualifier from *y*, then it declares the *y* before the task, and finally creates a task where the result from *f(x)* is put into *y*.

```
if(...){
    int var1;
    int& var2 = z;
    ... some code ...
}
```

(a) Original code.

```
if(...){
    int* apac_ptr_var1 = new int();
    int& var1 = *apac_ptr_var1;
    int& var2 = z;
    ... some code with tasks...
    #pragma omp task depend(inout: var1) firstprivate(apac_ptr_var1) default(shared)
    {
        delete apac_ptr_var1;
    }
}
```

(b) Transformed code.

Code 3 – Example of transformation - Declaration in scope. The compiler allocates *apac_ptr_var1* in the heap and transforms *var1* as a reference on it. The main code remains unchanged and continues to use *var1*. Finally, *apac_ptr_var1* is deleted by an extra task at the end of the scope.

```
int var;
f(var);
... some code ...
var += 1;
```

(a) Original code.

```
int var;
#pragma omp task depend(inout: var) default(shared)
{
    f(var);
}
... some code with tasks ...
#pragma omp taskwait
var += 1;
```

(b) Transformed code.

Code 4 – Example of transformation - Coherency enforcement. If a variable is used in a task and in the body of the function, the compiler inserts a synchronization/*taskwait* to make sure that the task has completed.

```
int functionWithReturn(int a, int b){
    if(a>b){
        ... some work with a ...
        return a;
    }
    else{
        ... some work with b ...
        return b;
    }
}
```

(a) Original code.

```
int functionWithReturn(int a, int b){
int apac_res;
#pragma omp taskgroup
{
    if(a>b){
        ... some work with a ...
        #pragma omp taskwait
        apac_res = a;
        goto apac_endtaskgrouplabel_functionWithReturn;
    }
    else{
        ... some work with b ...
        #pragma omp taskwait
        apac_res = b;
        goto apac_endtaskgrouplabel_functionWithReturn;
    }
apac_endtaskgrouplabel_functionWithReturn: ;
}
return apac_res;
}
```

(b) Transformed code.

Code 5 – Example of transformation - Return management. Each return statement is replaced by a *taskwait* followed by an assignment to *apac_res* and finally by a *goto* to jump at the end of the *taskgroup*. Finally, *apac_res* is returned after the *taskgroup*.