Applying StarPU runtime system to scientific applications: Experiences and lessons learned

Georgios Tzanos, Vineet Soni, Charles Prouveur, Matthieu Haefele, Stavroula Zouzoula, Lazaros Papadopoulos, Samuel Thibault, Nicolas Vandenbergen, Dirk Pleiter, Dimitrios Soudris

Applying StarPU runtime system to scientific applications: Experiences and lessons learned

Georgios Tzanos
National Technical University of Athens,
Athens, Greece
giorgostzanos@microlab.ntua.gr

Vineet Soni, Charles Prouveur
Maison de la Simulation,
CEA, CNRS, France
{vineet.soni, charles.prouveur}@cea.fr

Matthieu Haefele
Université de Pau et des Pays de l'Adour,
Pau, France
matthieu.haefele@univ-pau.fr

Stavroula Zouzoula
Linköpings universitet,
Linköping, Sweden
stavroula.zouzoula@liu.se

Lazaros Papadopoulos
National Technical University of Athens,
Athens, Greece
lpapadop@microlab.ntua.gr

Samuel Thibault
LaBRI, Université de Bordeaux
Bordeaux, France
samuel.thibault@labri.fr

Nicolas Vandenbergen
JSC, Forschungszentrum Juelich,
Juelich, Germany
n.vandenbergen@fz-juelich.de

Dirk Pleiter
JSC, Forschungszentrum Juelich,
Juelich, Germany
d.pleiter@fz-juelich.de

Dimitrios Soudris
National Technical University of Athens,
Athens, Greece
dsoudris@microlab.ntua.gr

***Abstract*—Task-based runtime systems are adopted by application developers for their valuable features including flexibility of execution and optimized resource management. However, the use of such advanced programming models in complex HPC applications often requires significant training time and programming effort. In this work, we share experiences and lessons learned from the use of StarPU in three independent projects of various complexity. We reach conclusions, with respect to training, programming effort, and existing challenges, that are useful to the communities of application developers, as well as to the developers of runtime systems. Finally, we suggest extensions to the runtime systems beneficial to application developers.**

***Index Terms*—StarPU, task-based programming models, HPC**

I. Introduction

Task-based programming models have been proposed to address challenges imposed by the complexity of modern parallel and heterogeneous computing architectures. Since programming paradigms at lower abstraction levels (such as thread-based) cannot effectively address the resource management challenges, programming paradigms at higher level, such as task-based models, are increasingly adopted by application developers.

Available runtime systems that leverage the task-based programming model include PARSEC [1], QUARK [2], StarPU [3], and StarSs [4]. They offer features such as resource allocation and scheduling policies of tasks in heterogeneous systems, including CPU and accelerators. Scheduling decisions are usually made at runtime, based on user-defined constraints, application requirements and availability of resources. These high-level programming models are a modular alternative to the heavy tuning of the software for each specific underlying hardware architecture, which, although it may provide optimal performance, often requires huge programming effort. Instead, they offer a flexible and portable application implementation.

Task-based programming models have been successfully applied to applications from various domains [5]. However, the exploitation of the advantages offered by the runtime systems, is not always straightforward. Building complex HPC applications on top of task-based runtime systems, often requires extended changes in code structure and many algorithmic modifications. More specifically, expressing the application algorithms as a graph of tasks such that the dependencies between them are minimized which in turn maximizes parallelism expression, is often very challenging and requires significant development effort. Apart from the development effort, the training time is also a critical factor for the adoption of the runtime systems leveraging task-based programming. Finally, the overhead imposed by the runtime resource management mechanisms is another factor that determines their effectiveness in each context.

There are several works that evaluate task-based programming models applied either to a few microbenchmarks or to entire benchmark suites. However, the evaluation is mainly performed in terms of performance and scalability, often by comparing taskified applications against other programming models. For example, the performance and the scalability of task-based programming models such as HPX, Cilk++ and OmpSs are presented and often compared against PThreads and OpenMP in several studies [6][7][8][9].

Only few works discuss aspects beyond performance and consider programmability and programming effort issues. As an example, in a recent work, the lines of code between OmpSs

and OpenMP application versions are compared, as an indicator of the readability and compact level of application source code [7]. However, since the runtime systems are widely adopted in the HPC community due to their benefits in terms of flexibility and portability, aspects beyond performance, such as training and programming effort required should also be investigated. Indeed, the effort needed to leverage task-based runtime system advantages, as well as lessons learned by using them are valuable to the communities of HPC application developers.

In this work, instead of focusing on the evaluation of the efficiency of runtime systems in terms of performance and scalability, we focus on alternative aspects, such as the training time needed to get familiarized with a task-based runtime system and the programming effort required to apply it. Therefore, this work is a contribution to sharing experiences and lessons learned by building HPC applications on top of a task-based runtime system. It targets the communities of HPC application developers that consider the use of task-based runtime systems and take into consideration not only the expected performance gains, but also the programming effort for application code modifications. Additionally, it targets developers of task-based runtime systems, who expect feedback from application developers in order to improve their tools.

The approach followed in this work is based on the collection of experiences from three independent projects, in which an application leverages the StarPU task-based runtime system. Based on the analysis of the experiences, we provide interesting conclusions and suggest improvements for future StarPU extensions. We argue that such experiences and lessons learned are important for application developers considering the use of task-based runtime systems, but are also a valuable feedback to developers of runtime systems.

The rest of the paper is organized as follows: In Section 2 we provide a brief introduction of StarPU. In Section 3 we provide information about the three independent projects in which StarPU was applied in specific applications and present results for each one. In Section 4 we discuss findings based on cumulative results from all projects. Finally, in Section 5 we draw conclusions.

II. The StarPU Runtime System

The StarPU runtime system aims at providing optimized application execution over large clusters of heterogeneous systems, such as composed of CPUs and GPUs [10]. It uses a task-based programming paradigm which captures high-level information from the application, and allows its scheduler to be very well informed of the computation performed by the application.

A. Terminology

Although the fundamental programming interface of the StarPU runtime system is the submission of a Directed Acyclic Graph (DAG) of tasks, applications expressing their computation as a task graph would be tedious. The recommended StarPU programming interface is thus rather based on the more convenient notions of *data*, *codelet*, and *tasks*, and the actual task graph is inferred from these.

A piece of *data* can be a vector, a matrix, a tile within a matrix, a sparse matrix, a block of a block-sparse matrix, etc. It represents the unit that StarPU will manipulate, i.e. transfer to a GPU, send over the network etc. It will be given as input and output parameters to *tasks*. The application needs to explicitly register its data to StarPU before submitting tasks since StarPU will handle all the transfers to GPUs and over the network. A data can be *partitioned* into sub-data, for instance to split a matrix into tiles.

A *codelet* is a collection of functions which achieve the same computation, thus various *implementations* for the computation. For instance, an `sgemm` codelet would be composed of the pointer to the `cblas_sgemm` function for execution on CPUs and the pointer to the `cublasSgemm` function for execution on CUDA GPUs, and so on for OpenCL, FPGA, etc. or even multiple implementations for the same architecture. The codelet also records the number of parameters for the function and whether they are used as input, output, or both. In the `sgemm` example there are 3 parameters: matrix tile A and B as input, and matrix tile C as input and output. Last but not least, the codelet contains performance models for the different implementations of the computation, which allows to predict an estimation of the time taken by each implementation for a given e.g. data input/output size. This notably allows to make a balance between GPU acceleration and the cost of transferring the data to the GPU. Different performance models are available, the most commonly-used is the history-based model which is calibrated on-line during application execution.

Submitting a *task* then boils down to combining a codelet with some data: for instance, applying the `sgemm` codelet over three tiles of a matrix.

B. Sequential-Task-Flow (STF) programming paradigm

Since the codelet provides input/output information, the programming interface can *infer* the dependencies between tasks to attain a sequential consistency semantic. This yields to what we called the *Sequential-Task-Flow* programming paradigm. For instance, algorithm 1 is the tiled Cholesky factorization, expressed with the STF paradigm.

In line 2, a task is submitted that applies the POTRF codelet that will read and *write* tile $A[k][k]$. In line 4, tasks are submitted that will make the TRSM codelet *read* the same $A[k][k]$ tile. StarPU will thus automatically add a *read-after-write* dependency between the POTRF task and the TRSM tasks. Similarly, all other dependencies are inferred, resulting with building the Cholesky task graph, while the programmer is only faced with a sequential-looking source code.

C. Features provided by a runtime system

Since the runtime system has a complete vision over the data and the tasks to be computed, it is able to provide the application with a flurry of features *without further effort from*

Algorithm 1 STF tile Cholesky

```
1: for (k = 0; k <NT; k++) do
2:   starpu_task_insert(&POTRF, RW, A[k][k], 0);
3:   for (m = k+1; m <NT; m++) do
4:     starpu_task_insert(&TRSM, R, A[k][k], RW,
       A[m][k], 0);
5:   for (n = k+1; n <NT; n++) do
6:     starpu_task_insert(&SYRK, R, A[n][k], RW,
       A[n][n], 0);
7:     for (m = n+1; m <NT; m++) do
8:       starpu_task_insert(&GEMM, R, A[m][k], R,
         A[n][k], RW, A[m][n], 0);
9: starpu_task_wait_for_all();
```

the application programmer. Some of these features are listed below.

- Maximum parallelism, since tasks are released for execution as soon as their input data are available.
- Data transfer to/from accelerators, with transfer/computation overlapping and data prefetching and eviction.
- Optimized scheduling that takes into account load balancing, task duration and data transfer delays [11].
- Out-of-core management when the data set is larger than the memory [12].
- Distribution of tasks over nodes of a cluster, with automatic data transfers [13].

The support for these features however comes with some cost which may limit the obtained performance gain. The typical overhead per task, due to runtime operations (submitting the task, handling its dependencies and data, scheduling it), is usually around one or two dozens of microseconds, which means that for the overhead to remain negligible, the amount of work per task should be well beyond 100μs worth of computation, typically 1ms or even 10ms.

III. Approach description and Application results

The selection of the applications to be included in this study was based on the following three criteria:

- Building the selected applications on top of StarPU to be meaningful. In other words, the obtained results have to be interesting to application developers, as well as to StarPU developers.
- The complexity of the application source code should vary: We selected both small linear algebra kernels, as well as large and complex scientific applications.
- To include applications developed in various programming languages which are dominant in HPC computer programming: C, C++ and FORTRAN.

In the context of this study, we define *project* as all the work undertaken by one or more engineers in an institute who familiarized themselves with StarPU, taskified an application, applied StarPU, collected evaluation results and finally provided feedback. The three projects included in this study are based on the use of StarPU in: i) A set of linear algebra kernels ii) KKRnano (scientific application that belongs to materials science domain) iii) MetalWalls (scientific application, which is a supercapacitors simulator). An overview of each project is shown in Table I. These projects were completed independently between 2018 and 2019 by different research groups. None of the engineers had prior experience with StarPU and this applies to all projects. Also, all engineers followed the guidelines prepared by StarPU developers and received guidance from expert StarPU developers when necessary [14].

TABLE I. OVERVIEW OF SELECTED PROJECTS

	Projects		
	Linear Algebra Kernels	KKRnano	MetalWalls
Original application programming model	-	OpenMP/MPI	OpenMP/MPI
Motivation	Demonstrate execution flexibility	Evaluate StarPU	Exploit StarPU features
Original app. LOC	∼200 per kernel	∼3K	∼7K
Programming Language	C	FORTRAN	F90 and C++
Institute	LIU	NTUA, Juelich	Pau University, Maison de la Simulation, Saclay

The information collected from each project is the following:

- Training time required by engineers to get familiarized with StarPU
- Programming effort required to apply StarPU in terms of lines of code and development time
- Parameters that affect the performance of the StarPU-ized version of each application and challenges to improve performance
- Features that StarPU could include as future extensions to assist application developers

A. Linear algebra kernels

We applied StarPU in a set of widely-used Linear Algebra Kernels: i) Sum of matrix row elements, ii) 2D convolution, iii) 1D Jacobi. The first kernel is the most computationally intensive part of a count-based streaming aggregation algorithm, as described in the literature [15], which is developed both for CPU and GPU. The rest of the kernels are taken from the Polybench benchmark suite (PolyBench/C 4.1 and PolyBench/GPU 1.0), also available both in CPU and GPU implementations [16]. The goal was to evaluate the flexibility of execution of such kernels in heterogeneous systems, based on StarPU scheduling decisions.

This work was conducted by a junior engineer in LIU without prior experience in StarPU and the training time required to get familiarized with StarPU was about 2 months. The engineer was not initially familiar with the Linear Algebra Kernels source code. The steps to apply StarPU to each one of the kernels were the following:

1) Taskification and StarPU implementation: The main computational part of each kernel was submitted to StarPU as a single task.
2) Enable StarPU scheduling decision-making based on the history-based performance model provided by StarPU

mechanisms. The model needed to be calibrated by executing the task at least 10 times before starting to collect results, according to StarPU instructions.

3) Execution of the application for each different input size in an heterogeneous node (CPU and GPU) and observation of the scheduling decisions taken by StarPU.

Applying the above steps in the first kernel required about 2 weeks due to limited familiarization with StarPU. However, for the second and the third kernel it required 2 days, only. The total number of StarPU-specific lines of source code needed (initializing StarPU, defining and submitting tasks, etc.) were about 35 in each kernel.

Linear algebra kernels evaluation results

The evaluation was conducted in a computing system with 2x Intel Xeon Gold 6138, 2x20 H/T cores and an NVIDIA Tesla V-100. With respect to memory specifications, the system contains 8 DDR4 DIMMs of 16GB, clocked at 2666MT/s (total 128GB RAM). The first kernel processes double-precision, while the rest process single-precision data. The scheduling criterion was to minimize the execution time, based on the StarPU history-based performance model.

The results are shown in Fig.1. It can be observed that StarPU assigned the tasks with small computational workload to CPU, while tasks with large workload to GPU. The task size ranges from 15ms up to thousands of ms, depending on the size of the input data.

B. KKRnano scientific application

KKRnano is an HPC application for performing density functional theory (DFT) computations on nanostructure systems [17]. It uses a Green function based variant of the Korringa-Kohn-Rostoker (KKR) method which can be shown to have linear asymptotic scaling behaviour, as opposed to the $O(N^3)$ scaling behaviour of classical eigensolver-based DFT solvers. This makes KKRnano particularly suitable to simulation of huge systems with hundreds of thousands of atoms on exascale systems.

The most computationally expensive part of KKRnano is the computation of the discretized and truncated Green function, which is equivalent to solving a specific huge block-sparse linear system. This system is solved using the transpose-free quasi-minimal residual solver, or TFQMR. As a typical KKRnano run spends around 90% of its compute cost on this solver, it was deemed useful to isolate KKRnano's implementation of the solver into a custom benchmarking suite.

StarPU, as an advanced programming model is evolving with various new features [18]. Upcoming features include multi-criteria scheduling, as well as fault tolerance, which are important for complex scientific applications, such as KKRnano. The goal of applying StarPU to KKRnano was to evaluate an initial implementation of StarPU in such a complex scientific application, not only in terms of functional requirements (i.e. performance, scalability), but non-functional, as well, such as programming effort and amount of code refactoring required. The training time required by a junior engineer in NTUA to get familiarized with StarPU was about 2 months. The engineer was not familiar with the original KKRnano source code.

Fig.2 shows the data flow of the original application (native TFQMR linear solver) and the application version that integrates StarPU. It is observed that each iteration consists of two steps. Each step includes a set of Sparse matrix-vector multiplications (*Spvm*) followed by a sparse-matrix *zgemm* operation (*SpGEMM*). In the original application, the *Spvm* operations are performed one after the other. Each *Spvm* operation runs in parallel using OpenMP, when it is feasible. The *SpGEMM* operation uses openMP-based parallelism, as well.

In order to apply StarPU effectively, we followed the steps below:

1) The *Spvm* operations are grouped in two StarPU tasks, as can be seen in the "StarPU-ized" TFQMR linear solver. Experiments showed that having more than two tasks would reduce performance, due to the small computational workload assigned to each one. Thus, the sequential steps of the TFQMR-solver were submitted to StarPU as two tasks (using C - Fortran interface) in order to be scheduled in a more efficient way than the purely sequential.
2) The next step was to register the *block compressed sparse row* (bcsr) matrices in StarPU, to partition them through StarPU and assign all dense matrix multiplications to a single StarPU task. Inside the StarPU task OpenMP parallelism was applied to the GEMM operations.

The purpose of this implementation was to let StarPU execute the six tasks (i.e.three tasks in each step) as a pipeline and handle data management efficiently.

A significant challenge lied in the fact that the critical parts of the application were developed in FORTRAN, which is not fully supported by StarPU. More specifically, the StarPU functions for partitioning the bcsr matrices were not available for FORTRAN, so the corresponding application parts were re-written in C. The re-writing effort was about 1500 lines of FORTRAN code converted to C and required about 1 month. The total time and programming effort in order to familiarize oneself with StarPU, implement StarPU in KKRnano and obtain results was approximately 5 months.

KKRnano evaluation results

The evaluation results are shown in Fig.3. The computing system consists of a single node with 2x Intel Xeon E5-2658A, 2x12 H/T cores and 128 GB RAM. All implementations use AVX2 (Advanced Vector Extensions 2) instructions.

The scalability results of Fig.3 show the execution time for 690 iterations of the TFQMR solver for three different implementations. The pure OpenMP is the original implementation of the KKRnano solver. The StarPU-based implementation with 6 StarPU tasks is described earlier (StarPU-6-tasks). We notice that the execution time, compared to the OpenMP implementation after 15 cores, increases between

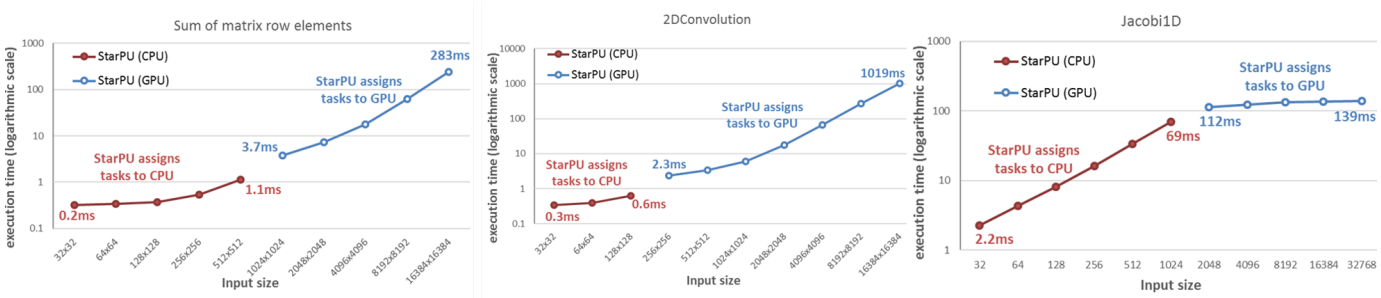

Fig. 1. Execution time of a set of linear algebra kernels using StarPU.

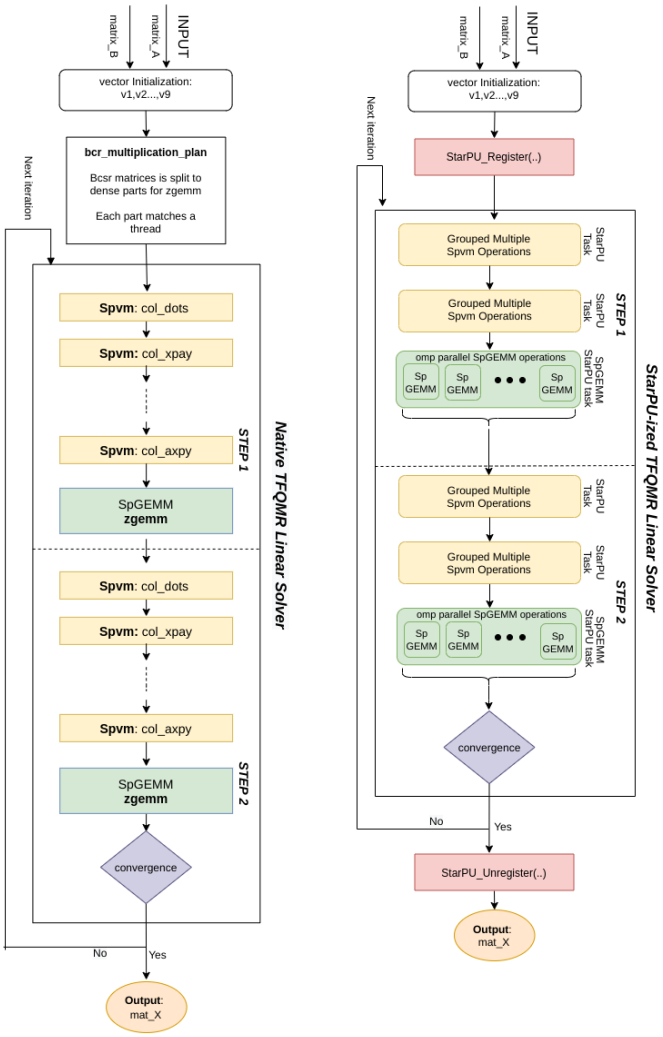

Fig. 2. Data-flow of the native and the StarPU-ized KKRnano versions.

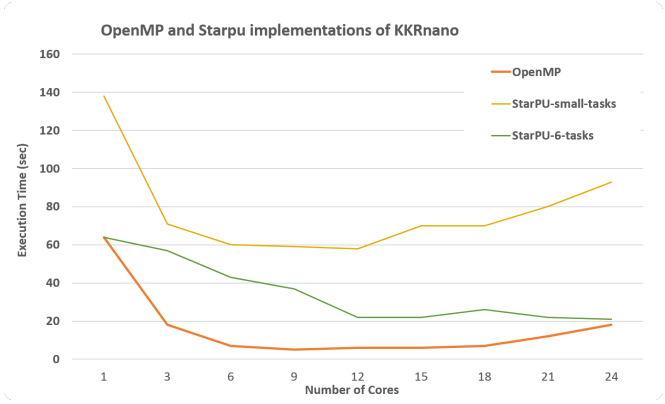

Fig. 3. Execution time for 693 iterations of TFQMR solver of KKRnano of original and StarPU-ized implementations.

x1.1 and x3. This overhead is attributed to the relatively small StarPU tasks generated for performing *SpGEMM* operations. Extensive profiling shows that the size of each one of these tasks in terms of computing time is between 0.18 - 0.5ms. Therefore, although the *Spvm* operations are about 2 times faster in the StarPU version of KKRnano (Spvm task size is about 10msec), the *SpGEMM* operations are about 3-4 times slower. This conclusion is also supported by the results of another StarPU-based implementation of KKRnano (StarPU-small-tasks), in which, instead of using 6 StarPU tasks, we assigned each SpGEMM operation to a single StarPU task. So, instead of using OpenMP parallelization, we applied task-based parallelization to the GEMM operations. However, results show that the StarPU-6-tasks implemenation significantly outperforms the StarPU-small-tasks, due to the very small task workload of the latter.

Enabling more features of StarPU, such as the seamless scheduling between CPU and GPU was also investigated. However, due to the sparsity of the matrix, the arithmetic intensity is too small and the time needed for data transfer is expected to exceed the computing time. Finally, one of the features that would help application developers managing legacy scientific codes is the extended support of FORTRAN by StarPU. Indeed, as stated above, in the KKRnano case the conversion of FORTRAN functions into C required significant programming effort.

C. MetalWalls scientific application

MetalWalls is a classical molecular dynamics code able to simulate electrochemical systems and in particular supercapacitors with an accuracy that put this numerical tool in a world leading position [19]. The original production application is a pure MPI F90 code made of more than 20k lines. A 7K lines mini-app has been extracted from it and focuses on the bulk of the computations: a matrix free conjugate gradient that finds the charge distribution on the electrodes given a bulk configuration such that a constant potential on the electrodes is kept. This set of computing kernels are representative of the application as kernels that have been stripped down retain the same algorithms and data structures.

The motivation to use StarPU is threefold. i) The task-based programming model allows to expose more parallelism. Three physics quantities need to be computed in the conjugate gradient and they can be evaluated concurrently as they do not have any dependencies between them. In the original MPI code, these three quantities are computed in a sequence whereas with tasks, they are computed concurrently. ii) Tasks allow for a greater degree of freedom in exploiting the parallelism in the code especially overcoming the load-imbalance issues. iii) Tasks also enable the use of heterogeneous architecture more efficiently with intelligent scheduling. Since StarPU has the ability to handle heterogeneous architecture with feature-rich scheduling techniques and sophisticated data-handling, it enables a powerful framework for a future effort on porting the application on GPU.

The implementation of StarPU in Metallwals was performed by two senior engineers from CNRS, who have a deep understanding of the original application, but were not familiar with StarPU. As a whole, it took two weeks to get trained on StarPU and another two weeks to implement it in Metalwalls with some tuning and performance evaluation. This relatively short time is mainly explained by the fact that the StarPU implementation has been derived from an OpenMP-Task implementation. Hence, the key concepts of taskification and major refactorings were already assimilated and realised. In total, approximately 650 lines of StarPU specific code were added to the mini-app.

The process of implementing StarPU in the MetalWalls mini-application was the following: Since the mini-application computes data with an algorithm by block in all three of its main compute kernels, the obvious candidate to create tasks was based on these blocks. In order to avoid the data-race condition, the only memory overhead is a temporary vector created with the size of the original vector $\times$ the number of StarPU workers. This essentially allows to create all tasks from every kernel at once and they are then executed concurrently when they are scheduled by the runtime system. At the end of each conjugate-gradient iteration, the contribution of all tasks is accumulated into the original vector. As a result, only a single barrier/synchronisation point is needed for the entire computation of a conjugate gradient iteration. Here, although GPU computing is not exploited in this work, it can be implemented later in the future to take advantage of the advanced StarPU task scheduling policies for heterogeneous computing.

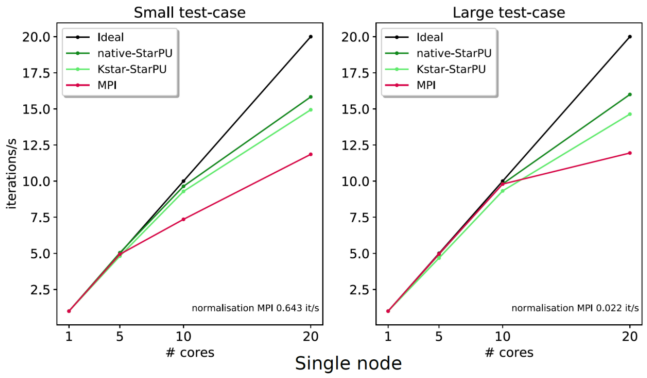

Fig. 4. Speedup comparison of StarPU tasks and MPI for small and large test-cases on a single node.

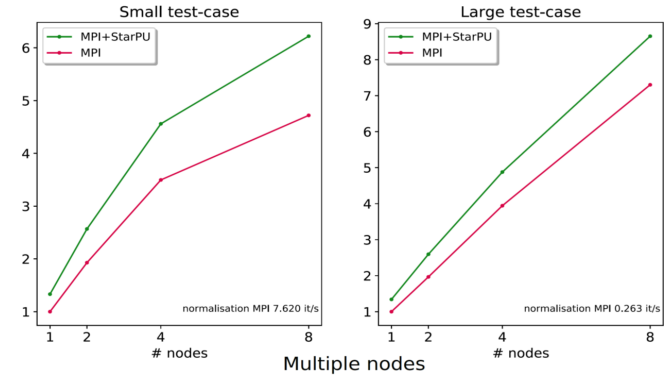

Fig. 5. Speedup comparison of StarPU tasks and MPI for small and large test-cases on multiple nodes.

MetalWalls evaluation results

The performance evaluation was carried out on Intel Haswell dual-socket nodes comprised of 10 cores processor per socket and with the GNU 9.2.0 compilers. Two test cases (small and large) were used whereby each kernel is used in different proportions to evaluate the effectiveness of the StarPU task performance. The multi-node tests were conducted with 1 MPI process per socket while the StarPU workers were bound to the cores analogous to *compact* for OpenMP threads.

The single-node and multi-node performance of StarPU tasks in comparison to the original MPI implementation are given in Fig.4 and Fig.5, respectively. As can be seen, the StarPU tasks clearly outperform the original MPI implementation in all cases with an increase in performance going from 18 to 33%. However, its performance drops noticeably while going from a single socket to the complete node. This could be attributed to memory affinity issues due to NUMA effects between the two sockets. Considering the multi-node scalability, one can see that the small test-case already reaches its scalability limit due to its small size, while the large test-

TABLE II. CUMULATIVE RESULTS

	Projects		
	Linear Algebra Kernels	KKRnano	MetalWalls
Training time	2 months	2 months	2 weeks
Developing time	1st kernel: 2weeks 2nd-3rd: 2 days	3 months	2 weeks
StarPU-specific LOC	∼35	∼700	∼650
Task size	15ms or more	10ms(Spvm)˜ 0.34ms(SpGEMM)	0.37ms - 1ms
StarPU extensions suggested	-	Extended FORTRAN support	Better support of StarPU and OpenMP combination

case continues to scale almost perfectly.

One of the key factors to consider in task programming is the size of tasks. It is fine-tuned in the current implementation based on the execution time of each block for all kernels. With the help of StarPU runtime overhead data available in the StarPU handbook, it was chosen to increase the block size from 8×8 to 128×128 to increase the task-size from $1.44 - 3.83\mu s$ to about $370 - 980\mu s$. More importantly, since the size of tasks is determined based on the time taken by each task, it varies significantly from one processor to another. Choosing the right task-size that performs well on every machine is difficult and in the Metalwalls case, the too small task size is the limiting factor for further scalability.

StarPU allows to combine OpenMP simply by enclosing the OpenMP parallel construct within `starpu_pause()` and `starpu_resume()` function calls within the StarPU region. But, upon mixing OpenMP and StarPU, the results were not promising. It is yet to be investigated whether it is due to the combined overhead cost of both techniques or something else. Element-wise dependency for vectors can be imposed only by manually partitioning the vectors into segments/elements and applying dependency over those segments. The possibility to express dependency inside a vector could reduce some code refactoring.

IV. Discussion

Table II summarizes the main findings. The *training time* required for engineers without prior knowledge of StarPU ranges from two weeks to months, depending on the extent by which they are familiar with the original application. The training time consists of understanding how to taskify applications efficiently so as to take advantage of the StarPU features and how to apply the StarPU API. The *development time* (i.e. the time required to taskify the application and apply StarPU), also depends on the complexity of the application. For example, in the KKRnano case, since the engineer was not familiar with the KKRnano application, significant amount of time was required to understand how to taskify the application and implement StarPU as efficiently as possible. On the other hand, in the linear algebra kernels project, although the engineer was not familiar with the original code, the low complexity of the code made the process of taskification and implementing StarPU much faster. This is also reflected in the number of StarPU-specific LOC.

A difficult but critical process is the identification of an optimal *task granularity*. Indeed, the task granularity is an utmost important parameter for performance. It is worth estimating up front how long the tasks will take, and expect having to group several pieces of computation into the same task to amortize the runtime overhead. For the Linear Algebra Kernels, the task-size is 15ms or higher, therefore the StarPU overhead is negligible. However, in the KKRNano cases, the relatively low task size increases the StarPU implementation performance overhead. Also, the fact that the right task size is platform-specific as stated in the MetalWalls analysis of results, imposes even more challenges to application developers.

A task-based runtime system is beneficial to easily obtain optimized execution on GPUs and distribution over the network with little effort from the programmer beyond migrating the source code to a task-based paradigm, as demonstrated in the Linear Algebra Kernels case. An advanced task-based runtime system is not beneficial for applications with little task size imbalance and that aim only for CPU execution without network use.

Additional StarPU features, such as better FORTRAN support would be very beneficial for developers, since the conversion of FORTRAN code to C is often very time consuming. Thus, modern advanced runtime systems would be much more attractive to developers that maintain legacy scientific code. Additionally, effectively combining StarPU with widely used programming interfaces, such as OpenMP, would allow developers to achieve the required performance for their applications with increased flexibility.

Finally, although this study is about StarPU, *similar conclusions can be drawn when using other task-based runtime systems*, such as StarSs [4] and HPX[9]. Even if the precise overhead will vary according to the runtime implementation, the same concerns are still applicable.

V. Conclusions

This work is a contribution to the communities of application developers that are considering implementing advanced task-based runtime systems, such as StarPU, within their applications. It focuses on the implementation effort and lessons learned from its use. Based on the analysis of three projects, we identified parameters that affect the training, development time, as well as the performance of the StarPU implementations compared to the corresponding native applications, such

as the optimal task size. Finally, we pointed out FORTRAN support and OpenMP within StarPU support as important future StarPU extensions.

ACKNOWLEDGEMENT

This work has received funding from the EU's Horizon 2020 research and innovation programme, under grant agreement No. 801015 (EXA2PRO, www.exa2pro.eu).

REFERENCES

[1] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[2] A. YarKhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, University of Tenessee, 2012.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par*, ser. Lecture Notes in Computer Science, vol. 5704. Delft, The Netherlands: Springer, Aug. 2009, pp. 863–874.

[4] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[5] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-Based FMM for Multicore Architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. 66–93, 2014.

[6] A. Podobas, M. Brorsson, and K.-F. Faxén, "A comparison of some recent task-based parallel programming models," in *MULTIPROG*, 2010.

[7] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero, "Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite," *ACM TACO*, vol. 12, no. 4, pp. 1–22, 2015.

[8] A. Leist and A. Gilman, "A comparative analysis of parallel programming models for c++," in *ICCGI*, 2014.

[9] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *PGAS*, 2014, pp. 1–11.

[10] "Starpu main webpage," http://starpu.gforge.inria.fr, 2020.

[11] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *ICPADS*, Shangai, China, Dec. 2010.

[12] A. AlOnazi, H. Ltaief, D. Keyes, I. Said, and S. Thibault, "Asynchronous Task-Based Execution of the Reverse Time Migration for the Oil and Gas Industry," in *CLUSTER*. Albuquerque, United States: IEEE, Sep. 2019, pp. 1–11. [Online]. Available: https://hal.inria.fr/hal-02403109

[13] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[14] "Starpu exa2pro tutorial," https://starpu.gitlabpages.inria.fr/tutorials/2019-05-EXA2PRO/, 2019.

[15] L. Papadopoulos, D. Soudris, I. Walulya, and P. Tsigas, "Customization methodology for implementation of streaming aggregation in embedded systems," *Journal of Systems Architecture*, vol. 66, pp. 48–60, 2016.

[16] "Polybench benchmark suite," http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2019.

[17] A. Thiess, R. Zeller, M. Bolten, P. H. Dederichs, and S. Blügel, "Massively parallel density functional calculations for thousands of atoms: KKRnano," *Phys. Rev. B*, vol. 85, p. 235103, Jun 2012.

[18] D. Soudris, L. Papadopoulos, C. W. Kessler, D. D. Kehagias, A. Papadopoulos, P. Seferlis, A. Chatzigeorgiou *et al.*, "Exa2pro programming environment: architecture and applications," in *Proceedings of SAMOS*, 2018, pp. 202–209.

[19] C. Merlet, B. Rotenberg, P. A. Madden, P.-L. Taberna, P. Simon, Y. Gogotsi, and M. Salanne, "On the molecular origin of supercapacitance in nanoporous carbon electrodes," *Nature materials*, vol. 11, no. 4, pp. 306–310, 2012.