Performance Analysis and Optimization of the FFTXlib on the Intel Knights Landing Architecture

Michael Wagner*, Victor López*†, Julián Morillo*†, Carlo Cavazzoni‡, Fabio Affinito‡, Judit Giménez*† and Jesús Labarta*†
* Barcelona Supercomputing Center (BSC), Barcelona, Spain
Email: michael.wagner@bsc.es
† Universitat Politècnica de Catalunya (UPC), Barcelona Spain
‡ Cineca, Casalecchio di Reno (BO), Italy

Abstract—**In this paper, we address the decreasing performance of the FFTXlib, the Fast Fourier Transformation (FFT) kernel of Quantum ESPRESSO, when scaling to a full KNL node. An increased performance in the FFTXlib will likewise increase the performance of the entire Quantum ESPRESSO code one of the most used plane-wave DFT codes in the community of material science. Our approach focuses on, first, overlapping computation and communication and, second, decreasing resource contention for higher compute efficiency. In order to achieve this we use the OmpSs programming model based on task dependencies. We allow overlapping of computation and communication by converting all steps of the FFT into tasks following a flow dependency. In the same way, we decrease resource contention by converting each FFT into an individual task that can be scheduled asynchronously. In both cases, multiple FFTs can be computed in parallel. The task-based optimizations are implemented in the FFTXlib and show up to 10 % runtime reduction on the already highly optimized version. Since the task scheduling is done dynamically during execution by the parallel runtime, not statically by the user, it also frees the user from finding the ideal parallel configuration himself.**

I. Introduction

Utilizing the enormous computation resources of current high performance computing (HPC) systems is a challenging and complex endeavor that requires consideration of parallel execution, network, system topology, and hardware accelerators as well as a variety of different parallel programming models such as message passing (MPI), threading and tasking (OpenMP), one-sided communication (PGAS), and architecture specific models to incorporate hardware accelerators such as GPUs. In addition, new architectures such as Intel's Knights Landing (KNL) urge for the adaption of existing, already optimized software to take advantage of their specific features and capacities. To facilitate the continuing optimization and adaption of software, performance analysis tools assist developers not only in identifying performance issues within their applications but also in understanding how new architectures and concepts influence their parallel behavior.

In this paper, we share our experience in combining the efforts of three research groups – scientific application development, performance analysis, and programming model development – to increase the performance of the FFTXlib on the Intel Knights Landing architecture. FFTXlib is the standalone miniapp that represents the Fast Fourier Transformation (FFT) kernel of Quantum ESPRESSO, one of the most used plane-wave DFT codes in the community of material science. The miniapp allows analyzing the impact of the parallelization parameters and their performance and is a easy-to-use tool for co-design and benchmarking of novel architectures like the KNL. The FFT kernel implements a layered MPI communication with FFT task groups to split the cost of collective communication operations to balance the impact on the performance (see Section II). However, the complexity in the many parallelization layers can be hard to manage, which was one reason for the development of the miniapp.

We analyzed the scaling behavior of the FFTXlib on the KNL architecture and uncovered two performance issues, namely, increasing communication costs and low computation efficiency. Based on the analysis we propose a new approach based on task dependencies with the OmpSs programming model that enables the concurrent execution of multiple FFTs. We target the increasing communication costs with overlapping computation and communication by converting all steps of the FFT into tasks following a flow dependency. We target the decreasing computation efficiency by decreasing resource contention, which is done by converting each FFT into an individual task that can be scheduled asynchronously, i.e. compute phases of high resource requirements can be overlapped with phases of low resource requirements.

We implement both versions in the FFTXlib. This allows comparing the performance of the new method to the existing code. Next to an increase in performance, we anticipate to remove some of the complexity of the multiple communication layers by providing a method whose performance is less affected by the specific setup. Since we make all optimizations available in the FFTXlib miniapp, we expect that the performance and usability gains are transferable to the entire Quantum ESPRESSO code.

The remainder of the work is organized as follows. In Section II we present the FFTXlib and the involved tools and libraries. In Section III we analyze the FFTXlib on the Intel Knights Landing architecture and discuss the main performance issues. In Section IV we present optimization approaches using the OmpSs programming model based on task dependencies and evaluate these optimizations in Section V. Finally, we draw conclusion in Section VI.

II. Background

In this section we provide an introduction to the FFTXlib and discuss the reasoning for the used tools and libraries to better follow the analysis and optimization techniques presented in the following sections.

A. The FFTXlib Miniapp

Quantum ESPRESSO [1] is one of the most used codes based on plane-wave DFT in the community of material science and it has been optimized for a very large number of HPC architectures in the recent years. The parallel structure of Quantum ESPRESSO is mainly based on several layers of MPI communicators, plus a finer grain OpenMP parallelization. The parallelization strategy relies on the symmetries of the solution of the Schroedinger equation permitting to treat independently many physical quantities (Kohn-Sham states, G-vectors, Brillouin indexes, etc.). In addition, some ad-hoc parallelizations on data structure permit to squeeze the performances on the most intensive computational kernels such as the linear algebra and the parallel FFT. In particular, the interplay between the distribution of the Kohn-Sham states and the distribution of the parallel FFT data structures (i.e. task groups [2]) allows to tune the pattern of collective communications. In this sense, the distribution of FFT task groups can permit to split the cost of *MPI_Alltoall* communications in different parts, balancing the impact on the performance. On the other side, the advantage of such complexity in the many parallelization layers is quite hard to manage because of the high number of tuning parameters. This is one of the reasons that led to the development of a miniapp containing only the FFT kernel. With this miniapp it is possible to analyze the impact of the parallelization parameters and their performance. Moreover, this miniapp can be also considered as a simple tool for a future activity of co-design and benchmarking of novel architectures.

The FFTXlib [3], [4] permits to study in depth the behavior of the FFT kernel in Quantum ESPRESSO according to the *knobs* of the parallelization. In plane-waves DFT codes, the cut-off of the kinetic energy imposes a constraint to the number of G-vectors, i.e. the number of vectors on which the many-body wave function is expanded. As a consequence, when a transformation from the reciprocal to the real space is needed, the domain on which the FFT acts is shaped as a sphere rather than a 3D cube. This feature reflects in the need of a redistribution of data among the MPI processes in order to balance the workload. After this redistribution is accomplished, the most efficient procedure for the parallel 3D FFT is by decomposing it in a 1D and a 2D FFT. This shows how the whole FFT is quite communication intensive rather than computationally intensive (typically in DFT simulations, the FFT grid is not huge).

The usage of task groups (introduced by [2]) permits to redistribute the G-vectors upon which the FFT acts. Task groups were introduced in order to prevent cases in which the number of processes can be larger than the number of FFT planes to process. An analogous situation is cared by the ortho-groups in Quantum ESPRESSO, where only a sub-communicator takes on charge the the diagonalization workload. Because of the task groups distribution, in order to perform a whole FFT transformation, a further MPI_Alltoall communication is required.

The FFTXlib miniapp reproduces the FFT kernel needed when an operator diagonal in real space should be applied to the wave functions. Since the wave functions are expressed in the reciprocal space, first a forward transformation is applied, then the potential is applied and, finally, a backward transformation is performed.

When the FFT task group parallelization is switched on, each MPI process has only a subset of the G-vectors for a given set of Kohn-Sham states. For the computation of the FFT, then, the G-vectors should be redistributed with a MPI_Alltoall inside the task groups. After this has been done, the FFT can be computed for the whole wavefunction. The structure of the code with task groups is depicted in Figure 1.

```
DO I = 1, NB, NTG
  CALL pack NTG bands
  CALL multi-band FW-FFT along Z
  CALL multi-band Scatter
  CALL multi-band FW-FFT along XY
  CALL VOFR
  CALL multi-band BW-FFT along XY
  CALL multi-band Scatter
  CALL multi-band BW-FFT along Z
  CALL unpack NTG bands
END DO
```

Fig. 1. Mockup of the FFT kernel in the FFTXlib with task groups.

Using this schema, the communication is split in two parts. The first one is taking place in the pack/unpackK routines. Here the G-vectors are redistributed among the processes belonging to the different task groups. The second one is taking place in the scatter between the 1D and 2D FFT. In this function the data is scattered from the 1D pencils to the 2D planes with an MPI_Alltoall. It is important to stress that the second communication takes place only within the task groups.

In order to better understand the mechanisms of task groups, we examine two extreme cases. First, the number of task groups is equal to one, i.e. task groups are switched off. In this case, the G-vectors are distributed among all the processes. The pack/unpack does not redistribute the G-vectors, so all the cost is shifted to the scatter routine that will involve all the processes and will be much more time-consuming. The opposite is the case where the number of task groups is equal to the number of MPI processes. In this case, each process will perform a single FFT within its set of G-vectors. Hence, the cost of the Scatter will be minimal and much more time will be consumed during the execution of the pack/unpack subroutines. All the options between these two extreme cases should be benchmarked. In order to do so, FFTXlib is a practical tool that does not require the whole execution of a DFT simulation and permits to extract performance data with the help of analysis tools.

B. Performance Tools and Libraries

While there are many of the common performance monitors and analyzers available on KNL, not all of them are equally suited. The proprietary tools of Intel (Vtune, Adviser) can be particularly helpful for analyzing the aspects of the single-thread performance on KNL such as vectorization. For the analysis of the parallel behavior the tools can be categorized mainly in two groups. On the one hand, there are tools based on periodic sampling like HPCToolkit, Openspeedshop or VTune. These tools intercept the application periodically to record the current status of the application, e.g. call stack information and hardware performance counters. The collected information is typically accumulated to profiles that represent a statistical summary of high-level application characteristics like the approximated time spent in each function. On the other hand, tools like Score-P use source code instrumentation to trigger runtime events, e.g. entering and leaving a function, sending and receiving a message, to gather detailed information and store them separately in application traces. Event-based tracing tools capture each process/thread and each runtime activity individually and record the exact communication behavior by intercepting the MPI runtime, thus, allowing a detailed analysis of the parallel behavior up to an entire replay or simulation of the application execution. While tools using instrumentation provide much more detail than sampling tools, they may introduce higher overhead, especially, for the function call interception, which on KNL is more costly because of the lower frequency.

For our performance measurements we choose the open source BSC tools [5] including the Extrae trace monitor [6] and the Paraver trace analyzer [7] for two main reasons. First, Extrae combines the benefits of instrumentation and sampling by intercepting the parallel runtime to provide the exact communication behavior but uses sampling instead of function instrumentation to record the application behavior in the compute phases. Thus, for our measurements we saw an average overhead of 0.6 % for the MPI runs and 2.2 % for the runs with MPI+OmpSs. Second and for this case most important, currently Extrae/Paraver are the only tools that allow to record and analyze detailed information from the OmpSs/Nanos++ runtime, which was necessary, for instance, to evaluate the task scheduling.

OmpSs [8] is a task-based programming model, based on OpenMP and StarSs, that extends new directives to support asynchronous parallelism through data dependencies and heterogeneity for multiple architectures and accelerators. OmpSs takes from OpenMP its philosophy of providing a way to, starting from a sequential program, produce a parallel version of the same by introducing annotations in the source code. These annotations do not have an explicit effect in the semantics of the program, instead, they allow the compiler to produce a parallel version of it. This characteristic feature increases the programmer's productivity, since the application can be parallelized incrementally instead of redesign it to implement a parallel version. The task construct allows the annotation of functions or structured blocks to define a task, which is considered the elementary unit of work that represents a specific instance of an executable code. The task construct implements the *in*, *out* and *inout* clauses to express data dependencies among tasks. These data dependencies are evaluated at runtime whenever a task is created to generate a dynamic task dependency graph, thus, only tasks with no predecessors in the graph will be scheduled for execution. The dynamic task scheduling during execution by the accompanying parallel runtime Nanos++ [9] is typically easier to use as well as faster than statically defined parallelism by the user.

III. Analysis

In this section we describe the analysis process and the resulting conclusions. The measurements for the analysis were performed on a local KNL test system at BSC, which allowed us to deploy and use our latest tools and libraries. The test system has a single KNL node with 68 cores at 1.4 Ghz with four-time hyper-threading.

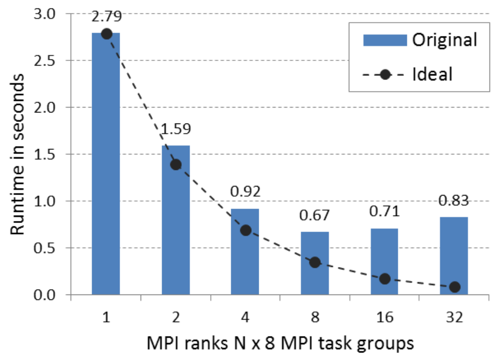

Fig. 2. Runtime of the FFT phase with increasing number of MPI ranks with the following parameters: Plane wave energy cut off: 80, lattice parameter: 20, number of bands: 128, number of task groups: 8.

Figure 2 shows the runtime of the FFT phase with an increasing number of MPI ranks ranging from 1 x 8 (ranks x FFT task groups) to 32 x 8; whereas the last two entries (16 x 8 and 32 x 8) use 2 and 4 hyper-threads per core, respectively. It can be seen that, first, the FFT phase does not scale very well with an increasing number of MPI ranks and, second, there is not benefit from using the hyper-threading; in fact the runtime is increased again.

To identify the main issues with the limited scalability we recorded traces of the different measurements with Extrae 3.4.3 in detailed tracing mode. Based on the traces we computed basic efficiency and scalability factors that allow to model the overall efficiency [10]. Table I depicts these factors for an increasing number of MPI ranks (the configuration 32 x 8 is excluded since it does not provide any additional benefit or information over 16 x 8). The basic idea of the model is combining these factors to compute derived indicators for performance issues. For instance, the global efficiency is derived by combining (multiplying) the parallel efficiency with the computation scalability; whereas the parallel efficiency itself is derived by combining (multiplying) load balance

TABLE I
EFFICIENCY AND SCALABILITY FACTORS FOR EXECUTIONS WITH 1-16 RANKS WITH 8 FFT TASK GROUPS EACH.

	1 x 8	2 x 8	4 x 8	8x 8	16 x 8
Parallel efficiency	95.75 %	91.21 %	92.70 %	90.97 %	86.15 %
→ Load Balance	97.31 %	95.04 %	98.31 %	98.18 %	96.91 %
→ Communication Efficiency	98.40 %	95.97 %	94.29 %	92.66 %	88.90 %
→ Synchronization	99.56 %	98.88 %	98.09 %	97.76 %	95.81 %
→ Transfer	98.83 %	97.06 %	96.13 %	94.78 %	92.78 %
Computation Scalability	100.00 %	91.87 %	78.09 %	54.74 %	27.32 %
→ IPC Scalability	100.00 %	92.78 %	78.68 %	56.28 %	28.26 %
→ Instructions Scalability	100.00 %	99.78 %	99.62 %	99.42 %	98.88 %
Global Efficiency	95.75 %	83.80 %	72.39 %	49.79 %	23.54 %

and communication efficiency. Load balance is defined by the average time divided by the maximum time spent in computation. The communication efficiency is defined as the maximum time over all processes in computation; i.e. outside of MPI. The computation scalability is the accumulated time over all processes in computation in relation to the smallest run, in this case 1 x 8. It can be further characterized by IPC and instruction scalability, which relate the average IPC in computation and the accumulated number of instructions in computation to the smallest run, respectively. From Table I can be inferred that the main issues that limit scalability are the low computation scalability and the decreasing communication efficiency. It also shows why hyper-threading does not provide additional benefits since the average IPC is more or less cut in half when going from 8 x 8 (no hyper-threading) to 16 x 8 (two-time hyper-threading). Apart from these, the FFTXlib achieves a very good load balance and instructions scalability (i.e. parallel work load replication), which testify the high level of the previously performed optimizations. In addition, the two-layer MPI communication has shown to effectively reduce the decrease in communication efficiency [3].

Figure 3 details the detected issues in the form of timelines. Timelines show the evolving program behavior along time (horizontal axis) for each process (vertical axis). The current state of each executed process is marked by a colored rectangle, which highlight the evolution of a selected metric of the recorded application. The upper timeline depicts the entire FFT phase with the color representing the length of each compute phase on a gradient from green (short) to blue (long). This timeline presents the general behavior of the FFT phase where the 64 FFTs are executed with 8 FFTs at the same time, i.e. 8 repeating phases can be seen. On the bottom there are three timelines zoomed into the third of these 8 repeating phases. These timelines show from left to right the average IPC, the MPI calls, and the used communicators. They allow identifying the main phases of the FFT: the preparation of the Psis with very low IPC (light green, average 0.06 IPC), the packing of the group sticks which calls an MPI_Alltoallv (dark yellow in the MPI call timeline), the forward FFT along Z (second phase in IPC timeline, average 0.52 IPC), teh forward scatter which calls MPI_Alltoall (violet in the MPI call timeline), the forward FFT along XY, the inner loop, and the backward FFT along XY (all three in the central phase, average 0.77 IPC), as well as the backward scatter, backward FFT along Z, and unpacking of the group sticks (all three mirroring the forward direction).

The communicator timeline highlights well the different communication operations and partners in the two-layered MPI communication with the FFT task groups. In the packing and unpacking of the group sticks there are 8 sub-communicators with 8 neighboring ranks each performing the MPI_Alltoallv. In general, for a setup of R x T (MPI ranks x FFT task groups) there are R sub-communicators with T ranks each. In the forward and backward scatter there are again 8 sub-communicators with 8 alternating ranks each (i.e. 1, 9, 17, ...) performing the MPI_Alltoall. In general, for a setup of R x T there are T sub-communicators with R ranks each.

IV. OPTIMIZATION

Based on the analysis of the FFTXlib we implemented two different optimization strategies. Both strategies convert different parts of the code into tasks where the dependencies are handled by the OmpSs runtime environment. In particular, there is a flow dependency within each loop iteration, while the iterations itself are independent from each other.

The first optimization strategy targets the decreasing communication efficiency by trying to overlap communication with computation phases. Therefore, each step of the FFT is converted into a task with the according dependencies, which can be done by marking them with the `$omp task` pragma as shown in Figure 4. This allows the OmpSs runtime to schedule each task only based on their dependencies, not by user-defined order. After that, we enabled all threads to participate in the parallel regions by nesting tasks in *fft_scalar.FFTW.f90* [4]. To this aim, we converted the main loops in functions *cft_2xy* and *cft_2z* into OpenMP task loops. The task loops distribute the number of iterations of the loop in chops equal to a given grain size. For our implementation, we have used a grain size equal to 10 and 200 (in the case of function *cft_2z*). Finally, we included an outer task loop around the main loop.

The second optimization strategy targets the low computation scalability, in particular, the decreasing IPC. Therefore, we tried to soften the resource contention by replacing the

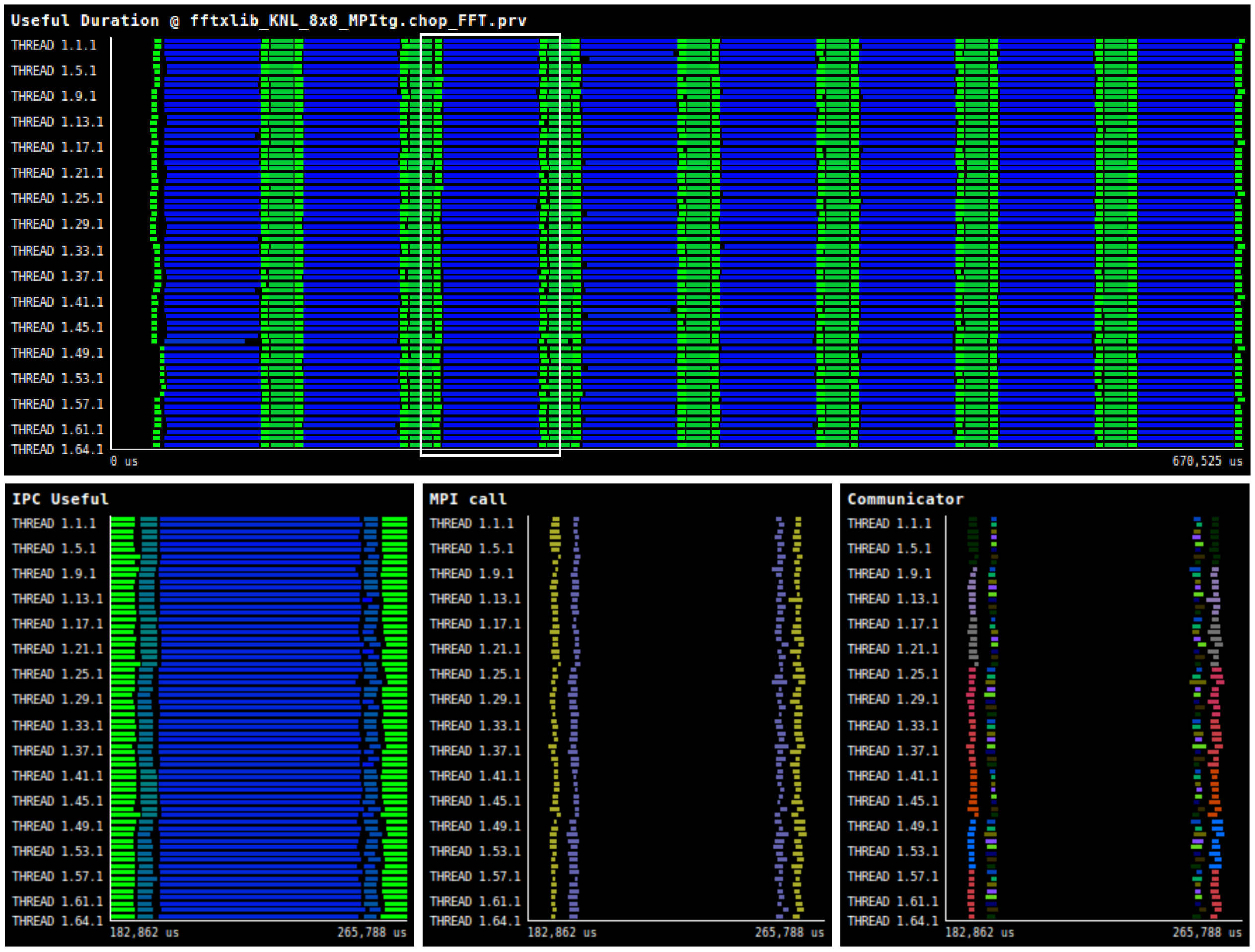

Fig. 3. Timeline showing the FFT phase (top), and a zoom into the a single sub-phase showing the average IPC, the MPI calls, and the used communicators.

```
!$omp taskloop inout(psis)
DO I = 1, NB, NTG !number of bands, number of task groups
  !$omp task in(aux) out(psis) in(dffts)
  CALL pack NTG bands
  !$omp task in(psis) in(dffts) out(aux)
  CALL multi-band FW-FFT along Z
  !$omp task inout(psis) in(dffts) inout(aux)
  CALL multi-band Scatter
  !$omp task inout(psis) in(dffts)
  CALL multi-band FW-FFT along XY
  !$omp task inout(psis) in(dffts)
  CALL VOFR
  !$omp task inout(psis) in(dffts)
  CALL multi-band BW-FFT along XY
  !$omp task inout(psis, aux) in(dffts)
  CALL multi-band Scatter
  !$omp task inout(psis) in(aux, dffts)
  CALL multi-band BW-FFT along Z
  !$omp task in(psis, dffts) out(aux)
  CALL unpack NTG bands
END DO
```

Fig. 4. Modifications to execute each step of the FFT as a task.

```
DO I = 1, NB, NTG !number of bands, number of task groups
  !$omp task default(shared) firstprivate(ipsi) &
  !$omp & private(aux, time, i, j) inout(psis) &
  !$omp & reduction(+:ncount, my_time)
  CALL pack NTG bands
  CALL multi-band FW-FFT along Z
  CALL multi-band Scatter
  CALL multi-band FW-FFT along XY
  CALL VOFR
  CALL multi-band BW-FFT along XY
  CALL multi-band Scatter
  CALL multi-band BW-FFT along Z
  CALL unpack NTG bands
  !$omp end task
END DO
!$omp taskwait
```

Fig. 5. Modifications to execute each FFT as a task.

second MPI layer (the FFT task groups). Instead of converting each step in to a single task (as for the first optimization), the approach converts each loop iteration, i.e. each FFT, into a single task (see Figure 5). Since there are no dependencies between the loop iterations each task can be scheduled without any further constraints.

We use the individual tasks to de-synchronize the computation phases. As mentioned in the previous section and as can be seen in the IPC timeline in Figure 3, there is a main compute phase with high compute intensity (high IPC) and phases with lower compute intensity. In the original version, all processes execute the computation phases more or less at the same time

TABLE II
EFFICIENCY AND SCALABILITY FACTORS FOR EXECUTIONS WITH 1-16 RANKS WITH 8 OMPSS TASKS EACH.

	1 x 8	2 x 8	4 x 8	8x 8	16 x 8
Parallel efficiency	99.13 %	95.53 %	91.67 %	83.33 %	70.47 %
→ Load Balance	99.86 %	98.25 %	95.52 %	91.81 %	90.32 %
→ Comm Efficiency	99.26 %	97.23 %	95.97 %	90.77 %	78.03 %
→ Synchronization Efficiency	100.00 %	99.84 %	99.85 %	97.52 %	92.17 %
→ Transfer Efficiency	99.26 %	97.39 %	96.11 %	93.07 %	84.66 %
Computation Scalability	100.00 %	92.56 %	81.16 %	61.36 %	37.29 %
→ IPC Scalability	100.00 %	94.04 %	84.05 %	66.14 %	42.57 %
→ Instructions Scalability	100.00 %	99.46 %	98.55 %	97.19 %	91.18 %
Global Efficiency	99.13 %	88.42 %	74.40 %	51.13 %	26.28 %

since they are statically parallelized and synchronized with the MPI collective calls. By using tasks that are scheduled dynamically by the runtime the compute phases are executed based on dependencies and resource availability. Therefore, the execution of the compute phases is de-synchronized, i.e. at any time only a subset of processes executes the main phase with high compute intensity while others execute the phases with lower compute intensity. As a result, the increasing resource contention that leads to the decrease in IPC when scaling to the full node (see Table I) can be partly absorbed.

The first optimization strategy is especially targeting large scales where the impact of the communication is very high and the computational load is relatively rather small. The second optimization is especially targeting scenarios with high computational load. For the execution on the Knights Landing test system, we chose the second version since the test node contains only 68 cores and the clock frequency of 1.4 GHz is lower than on standard CPUs, i.e. the compute performance is expected to be lower.

V. EVALUATION

In this section we evaluate the optimization using OmpSs tasks. We first discuss the general performance gain that can be achieved and then use the performance analysis tools to detail how this gains are achieved, i.e. evaluate our hypothesis about increased IPC through decreased resource contention via de-synchronization.

Figure 6 shows the runtime of the FFT phase with an increasing number of MPI ranks for the second optimization strategy with OmpSs vs. the original version (the last two entries use 2 and 4 hyper-threads per core, respectively). Thereby, the original version uses N x 8 MPI ranks, i.e. N ranks for the first MPI layer and 8 FFT task groups. The OmpSs version uses N MPI ranks and 8 threads that replace the FFT task groups. From Figure 6 it can be seen that the version using OmpSs performs the FFT phase about 7-10 % faster (not counting hyper-threading), in particular, the fastest version with OmpSs (16x8) is about 10 % faster as the fastest original version (8x8).

Table II that shows the efficiency and scalability factor for the version with OmpSs confirms that our intention to increase the computation scalability was effective. While in the original

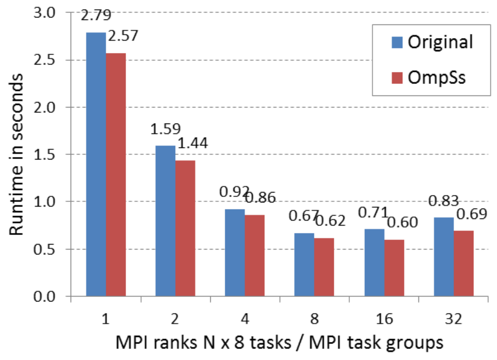

Fig. 6. Runtime of the FFT phase with increasing number of MPI ranks with the following parameters: Plane wave energy cut off: 80, lattice parameter: 20, number of bands: 128, number of task groups: 8 (orginal), 1 (OmpSs).

version the compute efficiency drops from 1.1 IPC for 1 x 8 to 0.6 IPC for 8 x 8, in the OmpSs version the compute efficiency only decreases to 0.8 IPC for the setup with 8 MPI ranks and 8 tasks each. In the case of two-time hyper-threading (16 x 8) the scheduling in OmpSs provides even slightly better results: while in the original version the compute efficiency is halved (0.3 IPC) in comparison to 8 x 8, the version with OmpSs still achieves 0.5 IPC.

As stated in the previous section, we anticipate higher compute efficiency by reducing resource contention through asynchronous execution of phases with different resource requirements, in particular, overlap phases with high compute intensity and phase with low compute intensity. To evaluate this hypothesis we recorded the runtime behavior of the original version and the OmpSs version. Figure 7 compares the two versions. It highlights the execution behavior for both in the timelines on the left side (same time scale) and the distribution of the phases with regard to IPC in the histogram on the right (same IPC scale). Both versions use 64 cores in a setup of 8 x 8.

In the original version (top, left) all phases are executed synchronously as can be seen by the structured blocks of phases with high IPC (blue) and low IPC (green) across the processes. Thus, phases with high resource demands are executed simultaneously on all processes, which leads to the

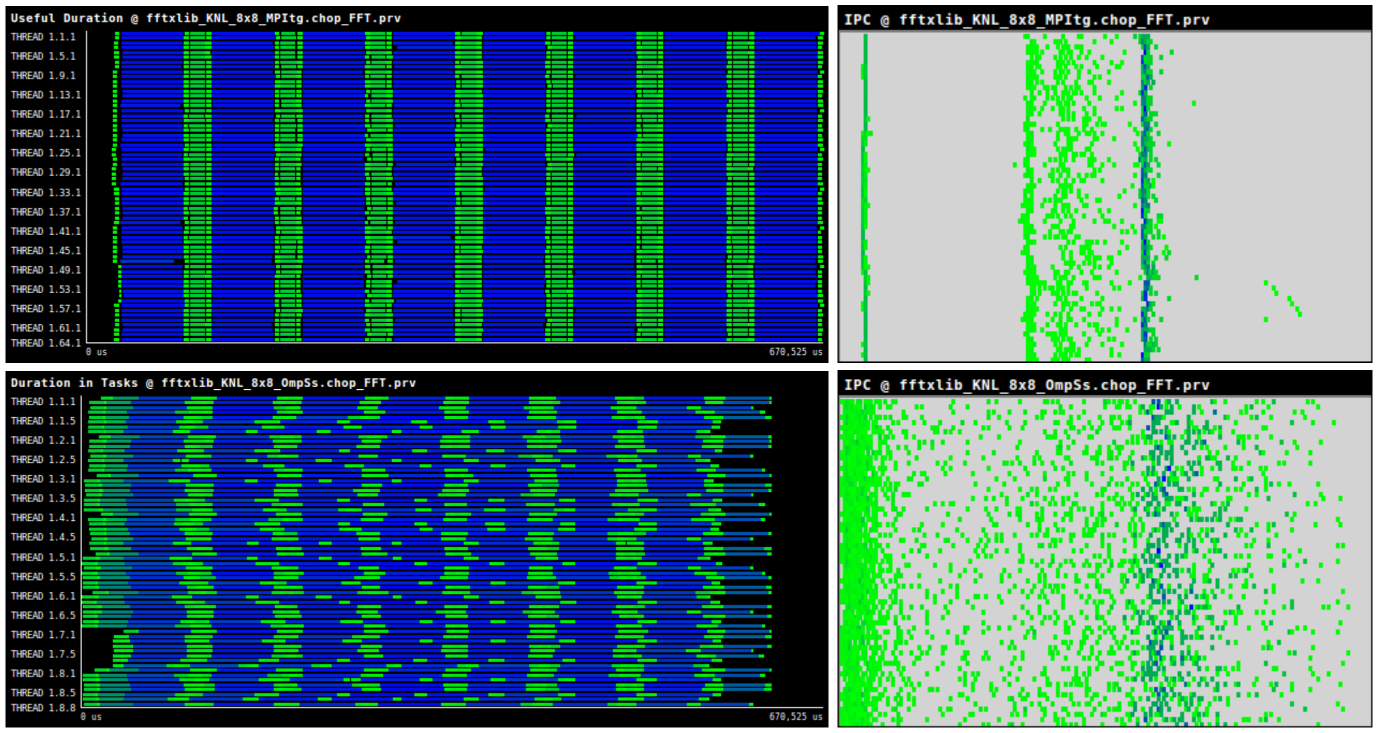

Fig. 7. The effects of de-synchronizing of compute phases with tasks: original version with 8 x 8 (top) vs. OmpSs 8 x 8 (bottom). Left: timeline showing the execution of the compute phases; right: histogram with the distribution of IPC.

detected resource contention and decreasing IPC with more active cores on the node. In the same way, phases with low resource demands are executed simultaneously and, thus, available resources are not fully utilized. In the OmpSs version (bottom, left) the phase are executed asynchronously. Thus, phases with high resource demands are executed at the same time as phases with low resource demands.

The resulting performance with regards to IPC can be observed in the histograms on the right side of Figure 7. The histograms show the distribution of IPC among the different phases, whereas each compute phase is categorized by process (vertical axis), IPC (horizontal axis), and duration (color gradient from green to blue), e.g. the blue dots represent the blue phases from the timeline on the left. The further on the right a point is located, the higher the IPC. Phases on the same process with similar IPC are grouped together and their time is accumulated. The upper histogram of the original version shows three vertical point clouds, whereas the one on the right sight with mainly blue dots shows the behavior of the main compute phase with the highest resource demands. As each process executes these phases more or less at the same time, all resources are fairly shared and all phases achieve more or less the same IPC.

In the OmpSs version the IPC of the phases is much more scattered. While generally such an imbalance is undesirable in a synchronized execution since it leads to wait times at the synchronization points, for a dynamic scheduling the imbalance has a much lower effect. In this, case the imbalance was actually the target of our optimization since the imbalance increases the average IPC. As can be seen, nearly all blue phases achieve a higher IPC as in the original version. The degree to which the IPC is increased for each individual phase depends on how many others core are executing the same phase at the same time. Blue phases that are close to the value of the original version are phases that coincide with blue phases on nearly all cores. Blue phase with higher IPC coincide with green phases on many other cores. While the behavior of the asynchronous scheduling seems to be more “chaotic”, the figure highlights clearly that the dynamic scheduling by the parallel runtime is more efficient than the static, user-defined scheduling with the FFT task groups. As a result the average IPC for these phases is increased from about 0.75 to 0.85 IPC.

In addition to the previous gain, the OmpSs version can gain an additional runtime reduction from two-times hyper-threading of about 3 %. This is among others due to the fact that the tasking approach is much more flexible in resource scheduling than the static FFT task groups, which allows integrating the additional resources, i.e. the four extra cores and the two-time hyper-threading, more efficiently. Moreover, since the scheduling is done dynamically during execution by the parallel runtime, not statically by the user, it frees the user from finding the ideal parallel configuration himself. This is particularly interesting for further platforms and heterogeneous systems because the runtime is capable of seamlessly scheduling the tasks on different architectures or accelerators.

VI. Conclusions and Future Work

In this paper, we address the decreasing performance of the FFTXlib, the Fast Fourier Transformation (FFT) kernel of Quantum ESPRESSO, when scaling to a full KNL node. We analyze the FFTXlib with the performance tools Extrae and Paraver to understand the behavior of the FFTXlib on the KNL architecture and highlight the two main performance issues that arise when scaling two a full KNL node. The first issue is the increasing communication cost for the use of the collective communication operations. The second issue is the decreasing computation efficiency caused by a decreasing IPC, where we identified resource contention as the main contributor.

We present two approaches that use tasks based on the OmpSs parallel programming model. The first approach targets the increasing communication costs by overlapping computation and communication. The second approach targets the decreasing computation efficiency by softening resource contention, which is done by asynchronously scheduled tasks, so, compute phases of high resource requirements can be overlapped with phases of low resource requirements.

We implement both approaches in the FFTXlib and evaluate the results of the second approach on the Knights Landing test system. We choose the second version since the test node contains only 68 cores and the clock frequency of 1.4 GHz is lower than on standard CPUs, i.e. issues in computation are more critical than issues in communication. The tasks based optimization proofs to soften resource contention and increases the IPC about 10 % in the main compute phase. As a result, the FFT phase shows up to 10 % runtime reduction on the already highly optimized version. An increased performance in the FFTXlib will likewise increase the performance of the entire Quantum ESPRESSO code one of the most used plane-wave DFT codes in the community of material science. Furthermore, since the task scheduling is done dynamically during execution, not statically within the code, it relieves some of the complexity of the two-layered MPI communication from the user and shifts it to the parallel runtime.

We continue analyzing and optimizing the FFTXlib, in particular, we try to combine the approaches to overlap communication and computation with asynchronously scheduled tasks. For this we also investigate the effects of automatically overlapping computation and communication using MPI communication within OmpSs tasks [11].

VII. Acknowledgments

We gratefully acknowledge the support of the MaX and POP projects, which have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 676598 and 676553, respectively.

References

[1] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. Dal Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, and R. M. Wentzcovitch, "Quantum espresso: a modular and open-source software project for quantum simulations of materials," *Journal of Physics: Condensed Matter*, vol. 21, no. 39, p. 395502 (19pp), 2009. [Online]. Available: http://www.quantum-espresso.org

[2] C. Bekas, A. Curioni, and W. Andreoni, "New Scalability Frontiers in *Ab Initio* Electronic Structure Calculations Using the BG/L Supercomputer," in *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA 2006, Umeå, Sweden, June 18-21, 2006, Revised Selected Papers*, 2006, pp. 1026–1035.

[3] F. Affinito and C. Cavazzoni, "FFT Data Distribution in Plane-waves DFT Codes. A Case Study from Quantum ESPRESSO," in *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, Edinburgh, United Kingdom, September 25-28, 2016*, 2016, p. 212.

[4] F. Affinito, C. Cavazzoni, and S. de Gironcoli, "FFTXlib." [Online]. Available: https://github.com/fabioaffinito/FFTXlib

[5] "BSC Tools." [Online]. Available: https://tools.bsc.es

[6] "Extrae instrumentation package." [Online]. Available: https://tools.bsc.es

[7] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *Transputer and occam Developments*, pp. 17–32, 1995. [Online]. Available: https://tools.bsc.es

[8] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[9] "Nanos++ RTL." [Online]. Available: http://pm.bsc.es/projects/nanox

[10] C. Rosas, J. Giménez, and J. Labarta, "Scalability Prediction for Fundamental Performance Factors," *Supercomputing Frontiers and Innovations*, vol. 1, no. 2, 2014.

[11] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 5–16.