OpenMP and StarPU Abreast: the Impact of Runtime in Task-Based Block QR Factorization Performance

Marcelo Cogo Miletto, Lucas Mello Schnorr

Graduate Program in Computer Science (PPGC/UFRGS), Porto Alegre, Brazil

{marcelo.miletto, schnorr}@inf.ufrgs.br

Abstract. *Directed Acyclic Graph (DAG) is a high-level abstraction to describe the activities of parallel applications. A DAG contains tasks (nodes) and dependencies (edges) in the task-based programming paradigm. Application performance depends on the choices of the runtime system. Our work intends to evaluate and compare the performance of three different runtime systems, GCC/libgomp, LLVM/libomp, and StarPU for a task-based dense block QR factorization. The obtained results show that while GCC/libgomp achieves up to 5.4% better performance in the best case, it has scalability problems for fine-grain problems with large DAGs. LLVM/libomp and StarPU are more scalable, and StarPU is much faster in task creation and submission than the other runtimes.*

1. Introduction

The path followed by the microprocessor industry to enable high-performance computing in nowadays machines had lead to the popularization of multicore architectures. To explore the performance provided by this type of architecture, one must consider the task-based programming model among the different parallel programming paradigms. The task-based paradigm permits to organize the application workload as a Directed Acyclic Graph (DAG), being a powerful idea that helps to port performance over different sophisticated computing platforms by using a runtime system to schedule complex workloads to many cores [Dongarra et al. 2017]. The nodes of the DAG represent computational tasks, and the edges express dependencies among them. The execution can be done in parallel, respecting the dependencies present in the graph.

There are many libraries and language extensions to represent task-oriented applications. StarPU [Augonnet et al. 2011], OpenMP [Dagum and Menon 1998], and TBB [Willhalm and Popovici 2008] are examples of these libraries. These tasking models are in constant evolution, OpenMP, for example, now enables more flexibility in the representation of tasks due to the implementation of more complex task-based directives. The StarPU library, which is designed specifically for task-based parallelism, counts with a dynamic runtime system which helps applications to perform over heterogeneous. Runtime systems help to reduce the complexity of programming by taking care of some operations like task scheduling, possibly using some strategy to favor load balancing and data locality. The runtime knows the DAG structure and decides which tasks can execute at a given moment. Also, it needs to specify where the task will run, associating it to a processing unit. The runtime takes these decisions in execution time, and they are different for two executions of the same program since the runtime decisions are stochastic, depending on several factors such the system resources availability at a given moment.

The ease offered by the runtime decisions represents extra computation cost for the application, possibly adding overhead into its execution time. The objective of this work is to compare and measure the impact that different runtimes have on the same task-based application. We rely on tracing techniques to support our findings. The experiments in this work will be conducted using the GCC/libgomp, the LLVM/libomp, and the StarPU runtimes. We chose these three runtimes because they are non-commercial and open source. OpenMP runtimes are widespread, and we include StarPU because our research group has some involvement in its development. **Our main contributions** are as follows. First, we evaluate the performance of the three runtimes for the same task-based application using experiments involving different computational environments and workloads. Second, we analyze application traces to highlight the runtime characteristics in terms of task management, looking at task creation/submission overhead, and the load balancing by terms of the idleness of each thread. Moreover, third, we identify a problem in the task creation/submission for GCC/libgomp where the overhead for it was much higher than for other runtimes. We detail these contributions in the context of a block QR factorization we have implemented for the three runtimes. We use dense QR factorization because this type of application is suitable for the task-based paradigm and capable of outstanding performance in modern architectures. Comparing to related work, ours include tracing techniques for performance analysis to all the three assessed runtimes.

Section 2 presents the application used and an overview of task-based programming and on tracing parallel application behavior. Section 3 presents the experimental methodology. Section 4 discusses the obtained results. Section 5 presents related work, and limitations of our work. Section 6 concludes the paper and presents future work. The companion material of this work is publicly available at `https://zenodo.org/record/3436264`.

2. Background on QR Factorization, Task-based Paralelism and Tracing

This section introduces an overview of what are the application characteristics and how to represent it as a task-based application using different programming interfaces. Then, we present three different strategies to trace the application behavior, one for each runtime.

2.1. Block QR Factorization

Given a matrix A of size $m \times n$, the QR factorization transform it in the two factors Q and R, where Q is an orthogonal matrix having the same size of A, and R is an upper triangular matrix of size $m \times m$ with nonzero diagonal elements. This factorization method is used to solve a system of linear equations that arise from many applications and plays a vital role in solving the linear least-squares problem. There are many algorithms for the QR factorization [Golub and Van Loan 2012] like the Gram-Schmidt orthogonalization [Schmidt 1908], Givens rotations [Givens 1958], and the Householder reflections [Householder 1958]. The use of the latter is a common choice as there are high-performance implementations [Lopez 2015] using blocks. The blocked version groups Householder transformations and execute them using matrix-matrix operations (Level-3 BLAS), achieving high performance with good cache utilization [Buttari et al. 2008]. By executing matrix-oriented operations (blocks) instead of rows and columns, the factorization makes those block operations more suitable for exploring parallelism. Indeed, the strategy treats the matrix factorization as a factorization of $nb \times nb$ submatrices, where nb

is the number of blocks for a square matrix. Computing many blocks of the matrix at the same time make parallelism possible.

The implementation of block QR relies on four LAPACK [Anderson et al. 1999] routines that factorize and update the blocks. They are organized in iterations as represented by the pseudocode in Algorithm 1. This algorithm follows a set of steps where all operations get repeated in smaller submatrices of the problem called the trailing submatrices, defined by the outermost loop. The LAPACK routines, along with the data they use, are as follows. `DGEQRT` factorizes a diagonal matrix block from the input, producing three matrices: R, V and T. R is the upper triangular matrix, and V is the lower triangular matrix that holds the Householder reflectors, both R and V overwrite the factored matrix block. Finally, T is stored separately, storing the accumulated Householder reflectors using the compact WY technique. `DLARFB` applies the reflectors calculated by `DGEQRT` to blocks to the right of the diagonal one, using V with the reflectors along with the matrix T. `DTPQRT` computes the QR factorization of a submatrix by combining the R factor calculated by `DGEQRT` or previous calls of `DTPQRT` on blocks below the diagonal. The routine updated the R factor of the diagonal block, generating the matrix V with the reflectors and T from the accumulation of the reflectors, storing it separately. Finally, `DTPMQRT` uses the reflectors V and the matrix T calculated by `DTPQRT` and to update blocks to the right of the blocks factorized by `DTPQRT`.

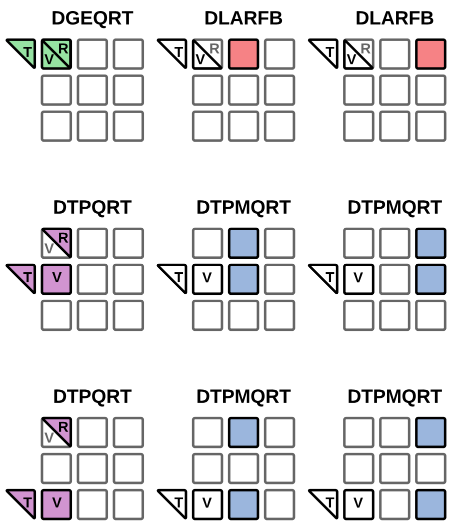
Algorithm 1: The Block Householder QR Factorization with LAPACK routines.

```
Data: Matrix A of size $mb \times nb$
for $(k = 1, 2, \dots, min(mb, nb))$ do
    $DGEQRT(A_{kk}, T_{kk})$;
    for $(j = k + 1, k + 2, \dots, nb)$ do
        $DLARFB(A_{kj}, V_{kk}, T_{kk})$;
    for $(i = k + 1, k + 2, \dots, mb)$ do
        $DTPQRT(R_{kk}, A_{ik}, T_{ik})$;
        for $(j = k + 1, k + 2, \dots, nb)$ do
            $DTPMQRT(A_{kj}, A_{ij}, V_{ik}, T_{ik})$;
```

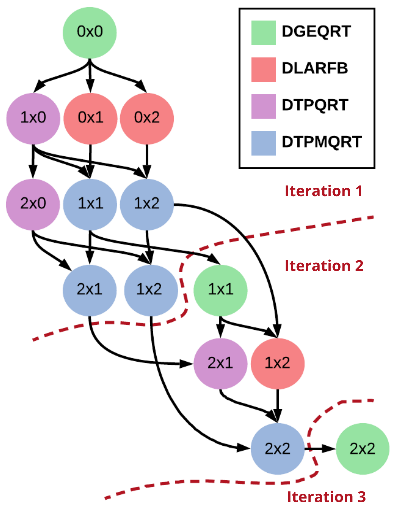
The steps of the first outer-loop iteration of the algorithm are depicted in Figure 1a for a matrix of 3x3 blocks. The first step ($DGEQRT$) is in the top-left. Remaining steps are depicted from left to right starting at the top. Filled blocks represent data writes on the block in that step, and the highlighted block areas mean that data is being read. By looking at the figure, it is possible to perceive that some steps can be done in parallel as the data access pattern does not generate conflicts. For example, after the $DGEQRT$ routine factorizes the first matrix block, the $DLARFB$ reads the lower part V, and $DTPQRT$ reads and writes in the upper part R, so these steps can be done in parallel.

2.2. Task-Based Parallelism for Dense block QR Factorization

In the task-based programming paradigm, a Directed Acyclic Graph (DAG) describes parallel applications, being composed of tasks (nodes) and dependencies among them (edges). Tasks can be scheduled concurrently by a runtime system respecting the restrictions imposed by dependencies and priorities from the application developer. Even with



(a) Steps of the first iteration.



(b) DAG for 3x3 blocks.

Figure 1. Visual representations of the block QR Householder Algorithm.

programmer hints, performance depends mostly on the actions of the runtime system, responsible by mapping tasks to workers efficiently. Distinct runtimes may have a better or worse performance than others for the same application. The task-based block QR factorization of Algorithm 1 defines a DAG where tasks (nodes) represent the LAPACK operations over some block, and the edges are the data dependencies between them. Figure 1b shows the resultant DAG for a matrix that was divided into $mb \times nb$ blocks where $mb = nb = 3$. The numbers inside the circles represent the matrix block that the task modifies, and the dotted red lines separate the iterations of the algorithm, Figure 1a represents the first iteration with its nine tasks.

OpenMP enables the programmer to determine explicitely data dependencies and priorities for the OpenMP tasks. Algorithm 2 shows an example with the modes that a specific memory region should be accessed using the `inout` and `out` constructs. A new task is created for every time that a `task` clause is reached in an OpenMP program. The created task will be responsible for executing the block of code that is below this construct, having some characteristics like the data dependencies and task priority. The task dependency is only fulfilled with the termination of the predecessor task. The OpenMP depend construct allows the programmer to define a range of the data as a dependency, as shown in the algorithm with `depend(out:T[0:len]` construct parameter.

Algorithm 2: Creating a DGEQRT task with OpenMP.

```
#pragma omp task depend(inout:A[k:k2]) depend(out:T[0:len])
    DGEQRT(&A[k], T, block_size);
```

StarPU is a task-based library equipped with different scheduling algorithms, some HEFT-based fitted for heterogeneous architectures (i.e., CPU and GPUs). Tasks are implemented with codelets that encapsulate specialized functions written for specific architectures (i.e, CUDA, OpenCL, OpenMP). The StarPU runtime system takes care of scheduling and executing those codelets. StarPU data handles and access modes set the dependencies over the codelet-defined tasks. A data handle maps the address of any data structure and the user can specify the modes that this piece of data will be used. Those modes are read, write, and read-write which are analogous to the `in`, `out` and `inout` OpenMP constructs. Algorithm 3 depicts, for a `DGEQRT` task, how the programmer submits a task using the `starpu_task_insert` function. As parameters, the function receives a codelet, the data handles, and the access modes that the task works on. For the `DGEQRT` task, we pass three data items representing the upper and lower part of the diagonal block kb and the matrix T. Other parameters such as block dimension can be passed using the `STARPU_VALUE`. The `NULL` indicates that there are no more parameters to this task.

Algorithm 3: Creating a DGEQRT task using StarPU.

```
starpu_task_insert(dgeqrt_codelet,
  STARPU_RW, blocks[kb_lower], STARPU_RW, blocks[kb_upper],
  STARPU_W, handle_T, NULL);
```

2.3. Tracing Task-Aware Application Behavior

Tracing techniques can be used to register specific events of the task-based program while the application executes. We need to identify the begin, end, and the worker of the execution of each task. The structural properties of the DAG must also be obtained, knowing a task location in the DAG and its dependencies. We detail below how such data is extracted from the runtimes with different tracing strategies. The **OpenMP Tools Interface** (OMPT) [Eichenberger et al. 2013] allows runtime task tracking with minimal overhead, using user-defined callbacks. The runtime triggers notifications whenever something has occurred. Captured events give details about what a specific thread is doing, such as task execution, barrier entry and exit, and other relevant events. Unfortunately, a complete OMPT support is lacking from current versions of GCC/libgomp and LLVM/libomp. Nevertheless, the LLVM/libomp implementation provides the necessary OpenMP thread and task behavior. Because the GCC/libgomp compiler lacks OMPT for tasks, we employ different methods to trace the two OpenMP runtimes. So, while we adopt an OMPT tracer we have implemented for LLVM/libomp, we employ Score-P tool [Knüpfer et al. 2012] for GCC/libgomp. Score-P contains a tool called Opari2, which is a source to source compiler that modifies the application source code prior to the compilation, automatically instrumenting the code to record specific events such as task creation and execution. **StarPU** generates application traces in a very straightforward way with FxT traces, a binary format that contains data about all events that occurred during the program execution. This trace file is converted to a Paje trace file format [de Oliveira Stein et al. 2010] using the `starpu_fxt_tool`, and exploited with the StarVZ framework [Garcia Pinto et al. 2018] to obtain data in a CSV format.

3. Comparison Methodology and Experimental Details

This section describes the hardware and software environment used in the experiments, and also the strategies for generating application trace. We use four hosts with varying hardware characteristics, as depicted in Table 1. For each platform, we detail the name as the platform identification, the amount/type of CPU, Cache, and RAMs.

Environment Configuration: We have compared the GCC/libgomp, LLVM/libomp, and StarPU runtimes. All of them use the same four basic linear algebra operations (see

Table 1. Description of the machines utilized for the experiments.

Name	CPU	L1/L2/L3	RAM
tupi	1 × 8 Xeon E5 2620V4 2,1 GHz	32KB/256KB/20MB	64 GB DDR4
orion	2 × 6 Xeon E5 2640V2 2,3 GHz	32KB/256KB/15MB	48 GB DDR3
draco	2 × 8 Xeon E5 2630V2 2,5 GHz	32KB/256KB/20MB	64 GB DDR3
hype	2 × 10 Xeon E5 2650V3 2,3 GHz	32KB/256KB/25MB	128 GB DDR4

Algorithm 1) as implemented by the same 3.8.0 LAPACK library, built on top of the BLAS library. The GCC/libgomp option represents the GCC 7.4.0 version, which implements the OpenMP 4.5 version with the libgomp runtime. The LLVM/libomp represents the libomp runtime [LLVM 2015] (also adhering to the OpenMP 4.5 specification) implemented in the LLVM/CLANG compiler, concentrating efforts of the Intel runtime. Finally, the StarPU option represents the 1.3 release of this runtime using the default Locality Work Stealing (LWS) scheduler. Subsection 2.3 discusses the tools we employ to collect data from each runtime. GCC/libgomp traced with Score-P 5.0; LLVM/libomp traced with OMPT 4.5 callback specifications with a driver we have implemented; and StarPU, with FxT 0.3.5. In this scenario, an essential factor to consider when tracing applications is the intrusion generated by the event recording. Through experiments, we stated that the tracing did not change the makespan tendency that the results without tracing presented, thus guaranteeing that the application behavior remains close to its natural behavior.

Design of Experiments: To assess the impact of each runtime in each of the platforms detailed in Figure 1 separately, we consider two factors for the QR factorization: matrix size (problem size), block size. The levels of these factors increase by powers of two. Matrix size range from 1024 to 32768 (six levels), while block size range from 32 to 1024 (six levels). Since some combinations of matrix and block size lead to substantial execution time and reduced resource utilization, we define a partial design. For example, not all block sizes combine with all matrices, for matrix 16384 we saw that with 128 as block size there was already a high number of tasks, and considerable execution time, so no further combinations with smaller sizes were made. In smaller matrix cases, we also not considered combinations that did not provide enough parallelism. In each machine, the working threads of the different versions were bound to the same cores. For each experimental combination, we replicated the experiment 30 times for matrices up to size 8192, and ten replications for the rest. We consider the execution times to respect a gaussian distribution, and standard errors are computed by assuming a CI of 99.7%.

4. Performance Evaluation Results and Comparison

We verify the similarity between the two implemented block QR versions (OpenMP and StarPU) looking at the execution trace and numerical results for a small case. We present an overview of the makespans as an initial comparison between the runtimes. Then, we employ application traces to investigate idleness of each thread/core and to look into runtime task management behavior, identifying meaningful performance issues.

Small-scale comparison of scheduling and numerical results: We compare the OpenMP and StarPU versions of the QR factorization we have implemented. The comparison includes a minimal example with a square matrix of size 1024 and a block size of 256. The purpose of this small-scale experiment is to assess the task scheduling and numerical results produced by each version. Figure 2 shows task execution location as a function of time. Colors indicate the type of the task, and numbers within each rectangle represent the unique identification following the order of the loop that creates them. Both runtimes lead to similar execution times, with slight variations on task scheduling. We automatically compared the numerical results of each version against the sequential execution, with no difference considering a precision of four decimal places.

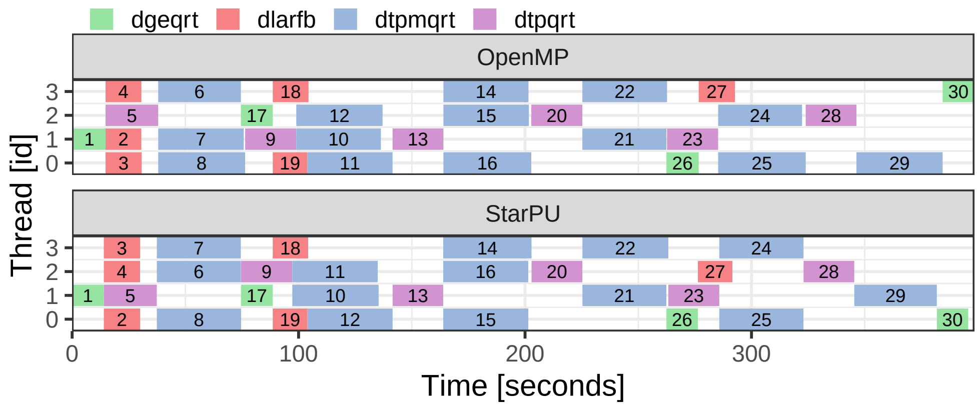


Figure 2. Task-scheduling comparison of the two implemented codes.

Makespan comparison for different platforms/runtimes and block sizes: Figure 3 depicts an overview of the makespan for combinations of machines and runtimes for the matrix size 16384, by varying the block sizes and by consequence the number of tasks. This workload is a representative case to summarize differences on the execution time of other factors (platforms, runtimes). The vertical bars represent the average execution time (makespan) as a function of the number of tasks. As shown in the left facets of Figure 3, the choice of the matrix size 16384 enable us to confirm that fewer tasks (1496) lead to a lack of parallelism opportunities. As we increase the number of tasks (to 11440), performance improves but ultimately lead to more unsatisfactory performance because of the overhead of too many tasks (89440). The right facet of Figure 3 illustrates the peculiar case of GCC/libgomp that stands out. There is an enormous overhead when compared to other runtimes.

From this overview, we select two scenarios worth of further investigation. The first is that the configuration with 11440 tasks (block size of 512 for matrix size 16384) leads to better performance for all machines, with GCC/libgomp the best overall. The second is the case with 707264 tasks (block size 128), where all runtimes present a considerable overhead. GCC/libgomp presents a more massive overhead than others. In what follows, we investigate these cases using detailed application traces.

Quantify Idle Time: We define idle time as the sum of time not computing tasks from the total execution time. We compute per-worker idle time to explain the two interesting scenarios. Figure 4 shows the distribution (using boxplots) of workers idle time as a function of the runtime type, in two scenarios: the left facet shows the case with block size 512 (total of 11440 tasks), while the right depicts the case with block size 128 (707264 tasks). The case with fewer tasks (left facet) indicates that GCC/libgomp presents a

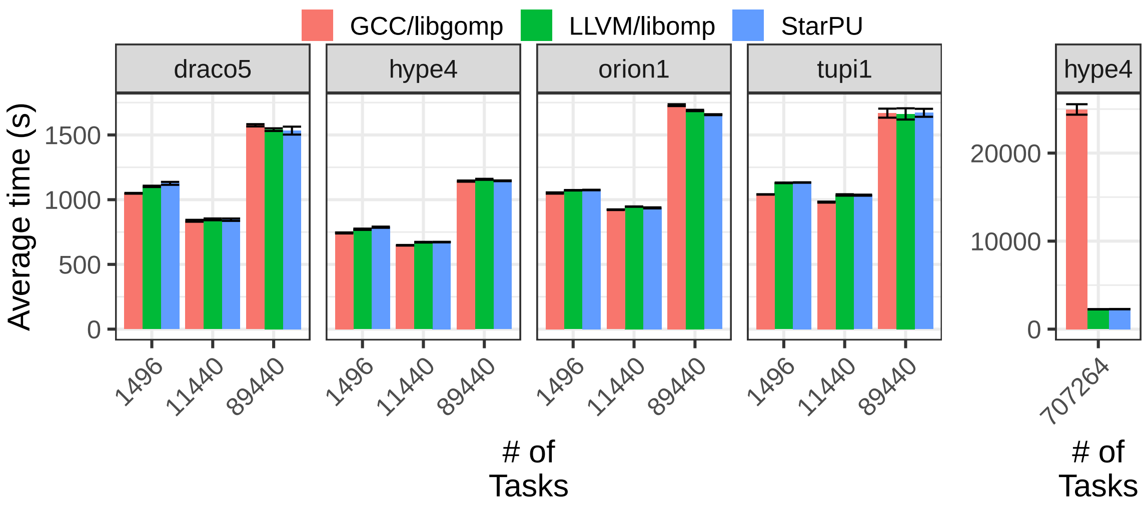


Figure 3. The average makespan for each machine using the 16384 matrix size, while varying the block size from 1024 to 256 (left facets). The same matrix with 128 block size for the `hype4` machine (right facet).

slightly reduced idle time per worker and a lower median, while the other runtimes have a higher idle percentage. The LLVM/libomp runtime has a clearly defined group of threads like GCC/libgomp, but with higher idle time. Differently, StarPU workers demonstrate a larger span, presenting values below and above the other two runtimes. With more tasks (right), GCC/libgomp presents a strange behavior: worker zero (the outlier in the bottom) barely stopped working in contrast with all the others that have been idle almost 90% of the time. The other two runtimes equally distribute the work among the workers. Because in both cases, the DAG has many parallelism opportunities, idle time is a result of the runtime choices, leading to unbalanced workload per worker or additional overhead for tasks management. Application makespan is most impacted, explaining why the GCC/libgomp has better performance (647s) for the block 512 cases when compared to LLVM/libomp (671s) and StarPU (672s), but much worse execution time with more tasks.

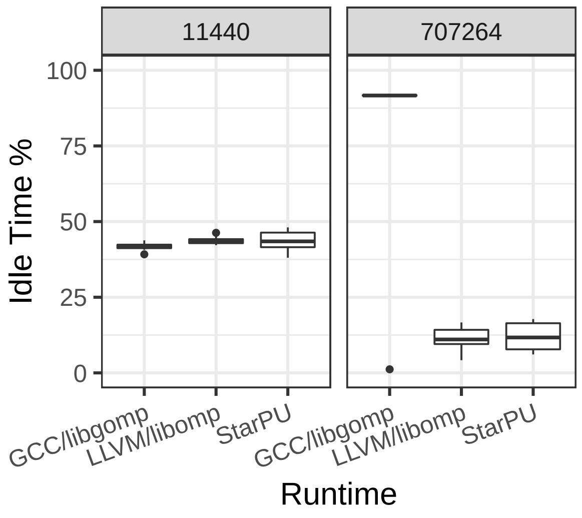
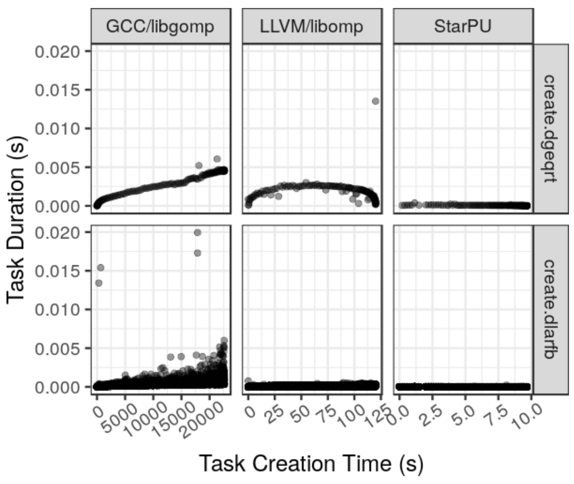
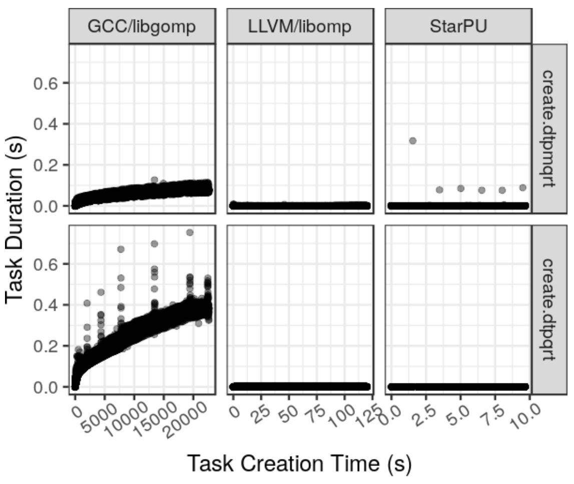


Figure 4. Idleness for the 128 and 512 block size cases in hype4 machine.

Task Creation Overhead: To better understand the GCC/libgomp runtime behavior with too many tasks (the 128 block size with a matrix size of 16384), we measure the time spent on every single task creation in each runtime. We then compare the two OpenMP runtimes task creation duration along the execution time, as depicted in Figure 5. Task creation duration for `DGEQRT` and `DLARFB` tasks are shown in the left (Figure 5a). The GCC/libgomp runtime presents increasing task creation duration as the execution evolves. The LLVM/libomp also presents this behavior, but attenuated, and, for the `DGEQRT` task, task duration decreases at the end of task submission. StarPU is the best in task creation time and submission for both types of tasks. Figure 5b depicts the task creation and submission time for `DTPMQRT` and `DTPQRT` tasks. Again, the GCC/libgomp presents a sustained increase in task creation time as the execution evolves, while both LLVM/libomp and StarPU keep relatively stable task creation times. As a conclusion, StarPU exhibits better performance overall: the whole task submission process took only 10 seconds while the OpenMP solution (LLVM/libomp) took 120 seconds to complete.



(a) `DGEQRT` and `DLARFB`.



(b) `DTPMQRT` and `DTPQRT`.

Figure 5. Task creation duration along execution time.

We also identify another source of poor performance in the computations carried out by other workers during the task creation. The GCC/libgomp fails to keep other workers busy while creating the tasks because after some time in the execution, the other workers remain idle, waiting for tasks to compute as we can see by the high idleness presented in Figure 4 for the block size 128 case, except for one worker, the worker 0.

5. Related Work and Discussion

Scheduling on task-based runtime systems is subject of constant studies. Complexity scaling of modern architectures makes scheduling even harder. Runtime systems play critical roles in this scenario. Some studies promote improvements on NUMA machines by leveraging memory-aware scheduling [Broquedis et al. 2010]. Other assess the behavior of task implementations of OpenMP in these environments [Terboven et al. 2012], discussing some essential aspects related to task creation overhead, such as different strategies to do it like the single-producer and the parallel-producer pattern. Other investigations concentrate on the balance between computational performance and energy efficiency [Broquedis et al. 2012, Agullo et al. 2017, Lima et al. 2017]. LibKomp is a runtime based on the X-Kaapi library [Gautier et al. 2013] that takes the tasking model of

OpenMP 3.0 and expand to support task dependencies [Broquedis et al. 2012]. Their comparison against other runtimes (GCC/libgomp, Intel TBB, Cilk+) uses the BOTS benchmark, concluding that achievable performances are comparable and even better for some cases. One of these cases is on applications that create a significant number of tasks. A comparison between GCC/libgomp, the Klang compiler with the StarPU runtime, and a native StarPU application has already been presented [Agullo et al. 2017], employing the Fast Multipole Method (FMM) as workload. This parallel application has tasks with very different workloads. The authors compared the performance in fork-join schemes and task-based schemes using parallel efficiency as a performance metric. For the task scheme using different numbers of threads, GCC/libgomp, Klang/StarPU, and StarPU alone performed very similarly. The GCC/libgomp was faster with only one thread, which means that its tasks are lighter than in other runtimes. They also detail differences between the StarPU and OpenMP execution models concerning the main thread behavior. In OpenMP, the main thread does both task creation and execution and is bound to a specific core. In StarPU, this thread only creates the tasks and is unbound by default. The conclusion was that it is possible to achieve a competitive performance using OpenMP tasks when compared with an optimized code natively written using the StarPU runtime system. There is also a performance/energy comparison among GCC/libgomp, LLVM/libomp, OmpSs, X-KAAPI and libkomp with different CPU governors using dense linear algebra algorithms (Cholesky, LU, and QR) as task-based applications [Lima et al. 2017]. Their findings show that there are differences in the GFlop/s rate for the tested runtimes with identical configurations. They also show that different task scheduling algorithms could impact application energy efficiency.

Discussion: Different runtimes impact the computational performance and the energy efficiency for the same application. Our work goes further intending to analyze the behavior of the chosen runtime systems for the dense block QR factorization, and deeply investigate behaviors that can degrade performance such as task management and load balancing. By executing experiments varying the problem size and the number of tasks, it is possible to capture the runtime behaviors in different scenarios. We employ tracing techniques like FxT, OMPT, and Score-P. Our work includes a comparison between OpenMP and StarPU runtimes using a dense linear algebra application, a combination that remains unexplored so far. Besides, we employ alternative yet equivalent tracing techniques to capture execution characteristics, unlike the majority of related works that used some global performance metric derived from application timestamps, or captured no measurements at all.

6. Conclusion

In this paper, we have proposed a detailed trace-based comparison of three task-based runtime systems (GCC/libgomp, LLVM/libomp, and StarPU) using our implementations of the dense block QR factorization. The runtime systems were subject to experiments from which we have identified the runtime impact on the application performance. Our results suggest that for the block QR application, GCC/libgomp runtime is incapable of sustaining a stable performance when the number of tasks is considerably high, but performs up to 5,4% better for the tupi machine when the block size is appropriate to support enough parallelism and reduced overhead. StarPU presents a very competitive performance comparing to the LLVM/libomp runtime, which is nice to add the fact that StarPU offers portability in terms of the specialized code support (MPI, CUDA, OpenMP) and dynamic scheduling based on performance models. As future directions, we can perform a more in-depth investigation on how task scheduling activities occur for each runtime system. For instance, we could investigate further the reason behind so much idle time in some scenarios, and correlate poor performance in task creation with the type of task data structures in the runtime. The main reason these areas remain unexplored is the lack of a single way to track scheduling decisions. Even if StarPU has many data regarding such actions, OpenMP runtimes are still incomplete regarding the OMPT API. Moreover, future work can explore other aspects of the task-based runtimes such as investigating how the scheduling affected the data reuse in cache hierarchy using a methodology like TaskInsight [Ceballos et al. 2017] which can explain why they achieved different idleness values. Also, we consider adding other existing runtimes to our studies like the StarPU OpenMP runtime support (SORS) which enables to generate StarPU code from the OpenMP task-based directives and the X-Kaapi/libkomp runtime.

Acknowledgements

We thank these projects and institutions for supporting this investigation: FAPERGS MultiGPU (16/354-8) and GreenCloud (16/488-9), the CNPq project 447311/2014-0, the CAPES/Brafitec EcoSud 182/15, the CAPES/Cofecub 899/18, the National Council for Scientific and Technological Development (CNPq), and the "Coordenação de Aperfeiçoamento de Pessoal de Nível Superior" (CAPES) - Finance Code 001.

References

[Agullo et al. 2017] Agullo, E., Aumage, O., Bramas, B., Coulaud, O., and Pitoiset, S. (2017). Bridging the gap between openmp and task-based runtime systems for the fast multipole method. *IEEE Trans. on Paral. and Distrib. Syst.*, 28(10):2794–2807.

[Anderson et al. 1999] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' guide*, volume 9. Siam.

[Augonnet et al. 2011] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.

[Broquedis et al. 2010] Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.-A., and Namyst, R. (2010). Forestgomp: an efficient openmp environment for numa architectures. *International Journal of Parallel Programming*, 38(5-6):418–439.

[Broquedis et al. 2012] Broquedis, F., Gautier, T., and Danjean, V. (2012). Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *International Workshop on OpenMP*, pages 102–115. Springer.

[Buttari et al. 2008] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2008). Parallel tiled qr factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590.

[Ceballos et al. 2017] Ceballos, G., Grass, T., Hugo, A., and Black-Schaffer, D. (2017). Taskinsight: Understanding task schedules effects on memory and performance. In *Intl. Work. on Prog. Models and Applic. for Multi. and Manycores*, pages 11–20. ACM.

[Dagum and Menon 1998] Dagum, L. and Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *Comp. in Sci. & Eng.*, 5(1):46–55.

[de Oliveira Stein et al. 2010] de Oliveira Stein, B., de Kergommeaux, J. C., and Mounié, G. (2010). Pajé trace file format. Technical report, ID-IMAG, Grenoble, France, 2002.

[Dongarra et al. 2017] Dongarra, J., Tomov, S., Luszczek, P., Kurzak, J., Gates, M., Yamazaki, I., Anzt, H., Haidar, A., and Abdelfattah, A. (2017). With extreme computing, the rules have changed. *Computing in Science & Engineering*, 19(3):52.

[Eichenberger et al. 2013] Eichenberger, A. E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., and Lorenz, D. (2013). Ompt: An openmp tools application programming interface for performance analysis. In *International Workshop on OpenMP*, pages 171–185. Springer.

[Garcia Pinto et al. 2018] Garcia Pinto, V., Mello Schnorr, L., Stanisic, L., Legrand, A., Thibault, S., and Danjean, V. (2018). A visual performance analysis framework for task-based parallel applications running on hybrid clusters. *CCPE*, 30(18):e4472.

[Gautier et al. 2013] Gautier, T., Lementec, F., Faucher, V., and Raffin, B. (2013). X-kaapi: a multi paradigm runtime for multicore architectures. In *2013 42nd International Conference on Parallel Processing*, pages 728–735. IEEE.

[Givens 1958] Givens, W. (1958). Computation of plain unitary rotations transforming a general matrix to triangular form. *J. of the Soc. for Ind. and Appl. Math.*, 6(1):26–50.

[Golub and Van Loan 2012] Golub, G. H. and Van Loan, C. F. (2012). *Matrix computations*, volume 3. JHU press.

[Householder 1958] Householder, A. S. (1958). Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342.

[Knüpfer et al. 2012] Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al. (2012). Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer.

[Lima et al. 2017] Lima, J. V., Raïs, I., Lefèvre, L., and Gautier, T. (2017). Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms. In *Intl. Symp. on Comp. Arch. and High Perf. Comp. Workshops*, pages 7–12. IEEE.

[LLVM 2015] LLVM, P. (2015). Openmp*: Support for the openmp language.

[Lopez 2015] Lopez, F. (2015). *Task-based multifrontal QR solver for heterogeneous architectures*. PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier.

[Schmidt 1908] Schmidt, E. (1908). Über die auflösung linearer gleichungen mit unendlich vielen unbekannten. *R. del Circolo Matematico di Palermo (1884-1940)*, 25(1):53–77.

[Terboven et al. 2012] Terboven, C., Schmidl, D., Cramer, T., and an Mey, D. (2012). Assessing openmp tasking implementations on numa architectures. In *Intl. Work. on OpenMP*, pages 182–195. Springer.

[Willhalm and Popovici 2008] Willhalm, T. and Popovici, N. (2008). Putting intel® threading building blocks to work. In *Proceedings of the 1st international workshop on Multicore software engineering*, pages 3–4. ACM.