HeSP: A Simulation Framework for Solving the Task Scheduling-Partitioning Problem on Heterogeneous Architectures

Antón Rey, Francisco D. Igual(✉), and Manuel Prieto-Matías

Dept. Arquitectura de Computadores Y Automática,
Universidad Complutense de Madrid, Madrid, Spain
{anrey,figual,mpmatias}@ucm.es

Abstract. In this paper we describe HeSP, a complete simulation framework to study a general task scheduling-partitioning problem on heterogeneous architectures, which treats recursive task partitioning and scheduling decisions on equal footing. Considering recursive partitioning as an additional degree of freedom, tasks can be dynamically partitioned or merged at runtime for each available processor type, exposing additional or reduced degrees of parallelism as needed. Our simulations reveal that, for a specific class of dense linear algebra algorithms taken as a driving example, simultaneous decisions on task scheduling and partitioning yield significant performance gains on two different heterogeneous platforms: a highly heterogeneous CPU-GPU system and a low-power asymmetric big.LITTLE ARM platform. The insights extracted from the framework can be further applied to actual runtime task schedulers in order to improve performance on current or future architectures and for different task-parallel codes.

1 Introduction and Motivation

Task-parallel programming models have emerged as an appealing solution in order to tackle the programmability problem on both homogeneous and heterogeneous platforms. These efforts aim at reducing user intervention to manage data dependences, task allocation and data transfer management by delegating those tasks to underlying runtime task schedulers. However, the ever-increasing heterogeneity in current (and future) architectures has dramatically aggravated the challenge for runtime developers; as more types of computing resources are available, it becomes more difficult to concurrently exploit them in order to optimize co-operative parallel implementations. One of the main conceptual problems lies on how to optimally (and possibly dynamically) partition a task into sub-tasks (that is, solving a *task partitioning problem*), and how to efficiently schedule them to the most convenient resource among those available in order to maximize performance (that is, solving a *task scheduling problem*).

In this paper, we present HeSP (*Heterogeneous Scheduler-Partitioner*), a simulation framework that addresses both problems in a simultaneous fashion.

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 183–195, 2016.
DOI: 10.1007/978-3-319-43659-3_14

Based on per-task and data transfers performance models, HeSP adds an additional degree of freedom to typical task scheduling policies by considering a joint task partitioning/scheduling approach. The framework proceeds by finding a set of task partitions that divides the initial workload into a number of sub-tasks with different granularity, that better fit to the underlying hardware resources at a given execution point. The approach drives to considerable performance improvements and more efficient resource utilization. We show that the new task scheduler-partitioner paradigm is of wide appeal to increase the scheduling quality on highly heterogeneous architectures, and to gain insights that can be further applied to specific task-parallel implementations, actual runtime task schedulers, and present and future heterogeneous architectures.

Runtime task schedulers are capable of managing efficient load balancing, asynchronous out-of-order task execution and handling data across separated memory spaces, abstracting these mechanisms to the programmer. Concretely, StarPU [1], OmpSs [2] or XKaapi [4], among others, offer implicit parallel programming models with transparent data dependence analysis among tasks, and support scheduling on heterogeneous processing platforms. Efficient scheduling under this task-based perspective strongly depends on the quality of the scheduling policies implemented in the runtime, and more specifically, how they address the special features of the algorithm and the underlying architecture.

These efforts usually consider the static creation and management of equally-sized tasks operating on uniform data tiles, which naturally drives to an improper load balancing among computing resources on heterogeneous architectures, given the different processing capabilities of each type of resource. As a side effect, establishing the optimal block size, even in the homogeneous target system case, is a time-consuming effort for the developer, and strongly depends on the algorithmic properties of the target implementation and the features of the underlying architecture. Although each processor type typically reaches its performance peak for substantially different task sizes, and the chosen initial granularity exposes a fixed amount of parallelism, few strategies have been developed in order to dynamically adapt task granularity to the underlying heterogeneous hardware. Focusing on dense linear algebra implementations, [8] propose a hierarchical directed acyclic graph (DAG onwards) strategy, creating a two-level DAG hierarchy on systems featuring two types of computing platforms (CPU/GPU). Similarly, [5] proposes an offline adaptation of the task grain size to the processor type and to statically assign tasks to distributed compute nodes. On the other hand, [3] proposes an alternative approach in which computing resources are aggregated as needed in order to adapt the computing capabilities to coarse grain kernels. The *Versioning* task scheduler for the OmpSs runtime [6] defines multiple implementations per task, each one targeting a different processor type, and decides at runtime where to map them based on historical runtime information.

HeSP extends the aforementioned efforts by exploring the global impact of *arbitrary* degrees of task granularity on an arbitrary heterogeneous platform,

adapting task sizes not only to the individual processor capabilities, but also to the current degree of available parallelism dictated by a specific algorithm.

1.1 A Motivating Example: Tiled Cholesky Factorization

Let us expose a motivating and illustrative example of the actual problems related with equally-sized task partitioning on heterogeneous platforms. The blocked Cholesky factorization decomposes an $n \times n$ symmetric positive definite matrix A stored by $s \times s$ blocks of dimension $b \times b$ each, into $A = LL^T$ where L is a lower triangular matrix. At runtime, the outer loop in the code depicted in Fig. 1 that calculates the Cholesky factorization divides the operation into a number of sub-tasks that, when executed under a task-parallel paradigm, generate a task DAG as that shown in Fig. 2(a). In the task DAG, nodes correspond to different tasks, and edges denote data dependencies between them.

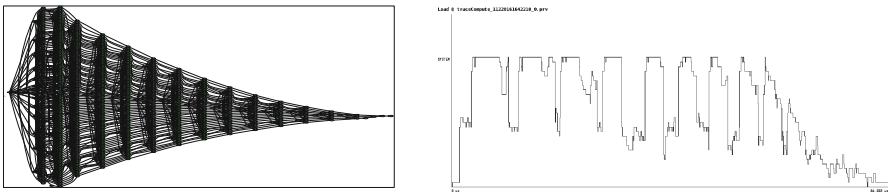
```
void cholesky (double *A[s][s], int b, int s) {
  for (int k = 0; k < s; k++) {
     chol (A[k][k], b, b);                  // Cholesky factor. (diag. block)

     for (int j = k + 1; j < s; j++)
        trsm (A[k][k], A[k][j], b, b);      // Triangular solve

     for (int i = k + 1; i < s; i++) {
        for (int j = i + 1; j < s; j++)
           gemm (A[k][i], A[k][j],          // Matrix multiplication
                 A[i][j], b, b);

        syrk (A[k][i], A[i][i], b, b);      // Symmetric rank-b update
} } }
```

Fig. 1. C implementation of the blocked Cholesky factorization.



(a) Task DAG. (b) Compute load trace.

Fig. 2. (a) Task DAG in which the computation evolves from left to right, and (b) compute load trace generated by the Cholesky factorization in Fig. 1, for a problem size $n = 16,384$, and block size $b = 1,024$.

The Cholesky factorization is an appealing example for our purposes: it exhibits different sub-task types (CHOL, SYRK, GEMM and TRSM) and complex

data dependences among them, and it features different degrees of parallelism as the factorization evolves. Consider, for example, how the DAG depicted in Fig. 2(a) reduces the potential parallelism (that is, the number of tasks that can be potentially executed in parallel, typically related with the width of the DAG) at the first stages of the factorization, and (in a much larger extent) at the last stages. This is usually translated into processor load patterns like that shown in Fig. 2(b), that represents a timeline of an execution of the Cholesky factorization on a highly heterogeneous platform, composed by 28 Intel Xeon cores and 3 different GPUs. The plot represents the number of active processors as the execution proceeds. Areas with reduced load are usually due to load imbalance. Note that this phenomenon can be motivated by two different factors: different processing capabilities of each processor type, and lack of potential parallelism on specific stages of the execution. The first can be alleviated by *scheduling* heuristics (e.g. mapping tasks in the critical path to fast processors), but the second is inherent to the algorithm, and can be alleviated by dynamic task *partitioning* in order to expose additional parallelism at runtime.

Data block (tile) size is a crucial parameter in task-parallel executions, as it ultimately determines the amount of available parallelism, and the efficiency of each individual task execution. In Fig. 1, block size is determined by `b`; note that, typically, larger block sizes usually imply higher performance per individual task, and smaller block sizes tend to expose higher degrees of parallelism, which naturally drives to better processor occupation. In addition, different block (task) sizes are desired for different architectures, and even for different problem dimensions in the same architecture.

Altogether, these observations motivate the exploration of new techniques that explore the impact of *heterogeneous* or *non-uniform* task partitioning on the performance and resource occupation of heterogeneous architectures. In the following, we introduce HeSP, a complete framework that supports the definition of complex heterogeneous architectures and simulates simultaneous task-scheduling and task-partitioning schemes that alleviate the aforementioned problems.

2 HeSP: Heterogeneous Scheduler-Partitioner

HeSP is a simulation framework that approximately solves the task scheduling-partitioning problem targeting heterogeneous architectures. At a glance, the input to this problem is *(1)* a hardware platform description where several finite-size memory spaces are connected according to a certain network topology, together with a (possibly heterogeneous) set of processors associated with them; and *(2)* a task to be computed in that platform. A solution to this problem consists of *(1)* a set of tasks –presumably with different granularity–, related by arbitrary data dependences and equivalent to the input task, and *(2)* a task-to-processor mapping. The objective function is typically performance maximization, although energy consumption minimization is also supported by HeSP.

2.1 Features of the Scheduling-Partitioning Simulation Framework

Besides supporting recursive task partitioning, HeSP is designed to be a realistic framework that simulates not only current heterogeneous architectures, but also state-of-the-art scheduling and data management policies on task-parallel executions. In the following, we introduce its features in detail.

Task and Data Scheduling Heuristics. HeSP implements different heuristics for task-to-processor assignments. *Random* (R-P) and *Fastest* (F-P) processor selection policies consider such processor choices among idle processors at the task release time. The *Earliest Idle Time* (EIT-P) and *Earliest Finish Time* (EFT-P) policies select the processor becoming idle first, and the processor finishing first if that task is assigned to it, respectively. EFT-P estimates the finishing time accounting for eventual data transfers if needed. Task scheduling order is specified by choosing between *First-come, first-served* (FCFS) or *Priority-List* (PL) choices. In PL, a priority list is built by sorting tasks by their critical times in decreasing order. Critical times are computed by averaging task processing time for all processors, and propagating them throughout the task DAG by a backflow algorithm. The combination of *Priority-List* and EFT-P heuristics is practically identical to the well-known *HEFT* scheduling algorithm [7].

When several independent memory spaces are present, HeSP considers data movement for scheduling decisions, considering individual memory spaces of each accelerator as software caches of a main memory space, typically tied to CPUs. Common caching policies like *write-through* (WT), *write-back* (WB) or *write-around* (WA) are used. When a task is about to be scheduled to a processor, the required data transfers are issued from the source memory space to the memory space the processor is tied to using prefetching schemes.

Performance and Data Transfer Models. HeSP estimates computing or transfer times relying on analytical models extracted a priori for each task/data type and size mapped to any existing processor/interconnect in the system. These estimations are required when making both scheduling and partitioning decisions, jointly or in an isolated fashion. The quality of these models will ultimately determine the accuracy of the simulated scheduling results.

Recursive Task Partitioners. Task *partitioners*, specified for each task type willing to be partitioned, are just blocked algorithms (see, for example, Fig. 1 for the specific case of the Cholesky factorization) with an input parameter that specifies the data granularity/degree of parallelism of the following partition. On a partitioner invocation, the corresponding emergent sub-tasks are managed by HeSP by introducing them in the respective task DAG which the partitioned task belongs to. In Fig. 3, starting from initial CHOL task –Cholesky factorization–, it is illustrated how three successive task partitions –corresponding to respective CHOL, TRSM and SYRK blocked algorithms– affect the prior task DAG, and the corresponding data partitions they induce.

Note that any task can be partitioned again as long as its dependent data blocks can be divided consistently, so an extremely hierarchical task DAG can

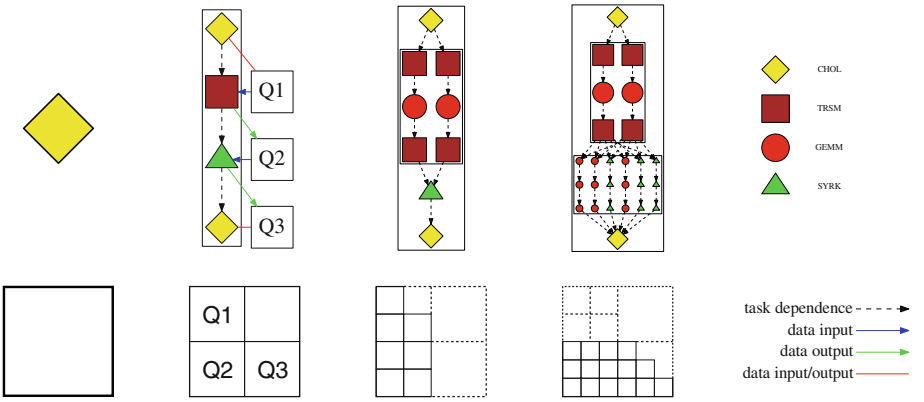


Fig. 3. Three successive task partitions and corresponding partitioned data blocks. Tasks and their related data dependences –Q1, Q2 and Q3 quadrants– are shown only for the first partition for the sake of clarity. (Color figure online)

be constructed by recursively partition its tasks. A *task cluster* is a set of tasks generated from a single task partitioning, being the source task their *parent*. We refer to the *DAG/graph depth* to indicate the maximum number of nested task clusters, and *DAG/graph width* as the maximum number of tasks that can be run in parallel. For instance, in the four task DAGs in Fig. 3, the corresponding depths are 0, 1, 2 and 2, and their widths are 1, 1, 2 and 6. Dependences between tasks, shown as dashed arrows, represent RaW, WaR and WaW constraints.

Recursive Data Partitioning and Data Coherence Management. New tasks generated after a partition reference to finer-grained input and output data dependences, which are partitions of the initial data block(s) of the parent task (see Fig. 3). HeSP implements validate/invalidate mechanisms to ensure data coherency among different memory spaces while handling asynchronous memory transfers. Since recursive task partitions induce corresponding recursive data block partitions, the existing partitioned data blocks are organized in a directed acyclic graph structure (data DAG onwards) in which nodes represent data blocks and directed links represent nesting relations between them; for example, $A \longrightarrow B$ means B is fully contained in A and A is bigger than B.

Armed with the data DAG, validations and invalidations are propagated by top-bottom and bottom-top mechanisms throughout this graph to maintain coherence. For instance, to ensure that a task can start its computation and store the result in an output block OB, not only the block OB must be invalidated on the remaining memory spaces in which the block might be present, but the hypothetical data block partitions contained in OB and the bigger blocks in which OB might be contained must be invalidated as well. Similarly, after a certain task has finished its computation updating OB, both OB and all the blocks within OB must be validated in the memory space corresponding to the processor assigned to that task.

In general, these data block partitions induced by task partitions form tree structures. However, it is possible to have a pair of blocks which intersect partially, nested within a common bigger parent block. This case shows up when two partitions of non-divisible grain sizes are applied to the same data block (for example, quadrant $Q2$ in Fig. 3). In this case, a new data block descriptor which refers to its intersection is introduced in the data DAG as a common child node of two intersecting blocks (see Fig. 4). With this mechanism, together with the validate/invalidate propagation mechanisms, data coherency is ensured for all possible data partitions and hierarchical data graphs.

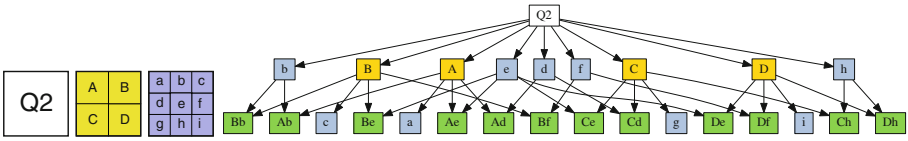


Fig. 4. A data block (Q2 quadrant) can be simultaneously divided according to different tilings (yellow and blue tilings, corresponding to TRSM and SYRK task partitions in Fig. 3). Additional data block descriptors (green) are constructed to represent partial overlaps between not nested data blocks. (Color figure online)

Iterative Solver. HeSP solves the scheduling-partitioning problem by iteratively searching for those hierarchical task DAGs which best fit to an heterogeneous processing platform—according to performance optimization—given a specific combination of the mentioned scheduling heuristics—processor selection heuristic and task ordering. A schedule stage is followed by a partition stage for each iteration, being the number of iterations a user-defined parameter.

At the partition stage, HeSP chooses a candidate task to be partitioned or a candidate task cluster to be merged back/repartitioned with a different granularity. A global analysis of the schedule-partition done in the previous iteration can provide useful information—i.e., bottleneck identification, number of idle resources, or too fine grained tasks— to help the iterative process to converge towards a better overall schedule-partition. This is the reason why we chose an iterative approach as the first implementation of HeSP instead of a *local-scoped* constructive approach, in which scheduling or partitioning decisions are made at every task arrival to the scheduling queue.

The partition procedure is based on two stages: *(1)*, task selection to build the candidate list, and *(2)*, sampling type to choose the final candidate. For *(1)*, HeSP implements three different policies: *All*, *CP* and *Shallow*. *All* selects all tasks of the previous step, *CP* selects only tasks belonging to the critical path, and *Shallow* selects those tasks whose depth (that is, number of task clusters that contain it) is minimal. All existing task clusters are candidates to be merged back or repartitioned. For each added candidate, a positive score is computed by subtracting the current cost delay by an estimated cost after its eventual partition or merge, being this estimation based on the available parallelism at its

scheduling time of the previous step. For each candidate whose data dependences have a characteristic size d, a partition parameter $p \in (0, 1]$ is chosen such that new tasks created after the eventual partition will depend on data blocks of size $b = p \times d$[1]. The more available parallelism is exposed, the smaller p is set in order to generate a higher amount of parallel finer-grained tasks.

In the second stage, a final selection among all candidates is done according to *Hard* or *Soft* procedures. In *Hard*, the candidate with the maximum score is chosen; in *Soft*, the candidate is randomly selected such that the selection probability equals the score divided by the sum of all scores.

3 Performance Results on Heterogeneous Architectures

In the following, we will feed HeSP with data describing two different heterogeneous architectures: BUJARUELO, a highly heterogeneous CPU-GPU architecture, composed by 28 Intel Xeon-E5 2695v3 cores running at 2.3 GHz, 2 GeForce GTX980 GPU and 1 GTX950; and ODROID, a low-power asymmetric ARM architecture with two types of processors: 4 slow Cortex-A7 and 4 fast Cortex-A15, running at 800 and 1300 MHz respectively. NVIDIA CUBLAS/CUSOLVER v7.5 and Intel MKL v11.3 were used to extract task performance models on BUJARUELO, and BLIS v0.9.1 was used on ODROID.

3.1 Framework Validation and Evaluation of Scheduling Heuristics

The goal of the first set of experiments is twofold: first, to validate the results extracted from HeSP by comparing them with an equivalent execution using a real task scheduler; second, to illustrate the impact on performance of several scheduling policies in HeSP when using *homogeneous* or *uniform* task partitions.

Each point in the OmpSs line in Fig. 5 (left) corresponds to the best scheduling performance out of 20 OmpSs executions for each grain size. These 20 trials were set to let OmpSs *Versioning* scheduler [6] improve itself by gathering enough task execution delay samples for each task type/size and processor. The other two curves—HESP-REPLICA-PM and HESP-REPLICA-RD—denote the performance attained by HeSP when applying the same task-to-processor mapping extracted from the best OmpSs trial, using our performance models and the real OmpSs task delays, respectively, for each uniform tiling.

Differences in performance between HESP-REPLICA-RD and OMPSS points are a measure of the OmpSs runtime overhead while the differences between HESP-REPLICA-PM and HESP-REPLICA-RD are mainly due to the accuracy of our performance models and possible differences between own OmpSs task delay instrumentation module and the instrumentation we used to extract our performance models. Summarizing, the differences between the replicated schedules are small enough and easily explainable to assert the validity of the following results. In general, our observations reveal a qualitative matching between real and simulated workloads for all problem sizes, with deviations that can be easily explained and do not usually affect the quality of the observations.

[1] A task cluster is a candidate to be merged if $p = 1$ or repartitioned if $p < 1$.

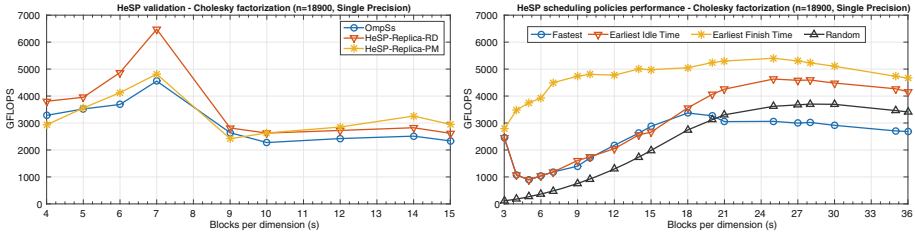


Fig. 5. Left: Comparison between OmpSs and their replicated schedules. Right: Comparison between different scheduling policies and block sizes in HeSP.

To introduce the context where our *heterogeneous* or *non-uniform* partitioning approach takes place and its potential benefits, Fig. 5 (right) reports the performance obtained by running HeSP simulations using different scheduling policies for different uniform task partitions. Some facts are remarkable: first, the optimal tile size does not only depend on the underlying architecture and problem size, but also on the selected scheduling policy; second, for every policy, performance curves follow a similar pattern, exhibiting a peak performance in a trade-off tile size that best balances potential parallelism and optimal individual task performance; third, differences in performance are relevant depending on the selected scheduling policy, being even more dramatic for large tile sizes; this gives a clue on the potential benefits that will be obtained by using a non-uniform partitioning scheme, as exposed next.

3.2 Impact of Non-uniform Partitioning on Performance

In the following, we illustrate the main performance improvements obtained with HeSP using *All/Soft* configuration for task partitioning selection. Table 1 reports performance values on BUJARUELO and ODROID using the best uniform and non-uniform partitions found by HeSP for different scheduling policies[2], together with additional metrics that clarify many of the concepts exposed hereafter, including average processor load, optimal/average block size and task DAG depth. The first point to notice is the overall improvement attained for all non-uniform task partitions found by HeSP and the overall reduction in the optimal average block size on non-uniform partitions.

Note the direct relation between the average processor occupancy and the improvements of the non-uniform partitions. For example, EIT-P with uniform partitioning yields high processor occupancies (between 91 % and 98.5 %), so the potential benefit expected from additional extracted parallelism is poor, ranging between 0.76 % and 2.02 %. Contrary, uniform partitions on EFT-P schedules yield better performance than EIT-P ones while still leaving more room for potential parallelism. Although the quality of EFT-P schedules could actually leave little room for additional improvements, the greater processor availability

[2] In all cases, we use WB as the caching mechanism.

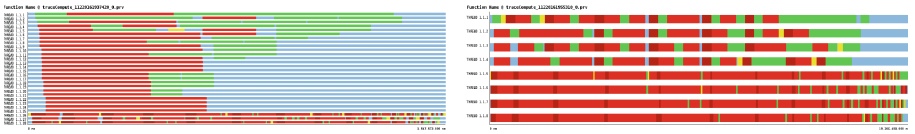
Table 1. Performance comparison for BUJARUELO and ODROID.

BUJARUELO (32,768 × 32,768 Cholesky factorization in single precision)								
	Best Uniform			Best Found Non-uniform				
	Perf	Avg. load	Block	Perf	Improve	Avg. load	Avg	DAG
Config	(GFLOPS)	(%)	size	(GFLOPS)	(%)	(%)	block size	depth
FCFS/R-P	3453.91	75.3	1024	4189.17	21.29	82.3	991.23	2
PL/R-P	4460.30	88.4	1024	4752.43	6.55	89.4	978.33	2
FCFS/F-P	2846.78	53.4	2048	3687.93	29.55	63.6	446.52	3
PL/F-P	3381.76	68.4	2048	3614.28	6.88	66.2	1165.70	3
FCFS/EIT-P	5650.10	91.3	1024	5747.87	1.73	92.3	1002.26	2
PL/EIT-P	6096.91	93.9	1024	6206.55	1.80	95.4	1009.91	2
FCFS/EFT-P	6581.96	23.3	2048	7569.34	15.00	63.9	412.15	5
PL/EFT-P (*)	7046.87	55.9	2048	8030.50	13.96	86.9	407.41	4
ODROID (8,192 × 8,192 Cholesky factorization in double precision)								
	Best Uniform			Best Found Non-uniform				
	Perf	Avg. load	Block	Perf	Improve	Avg. load	Avg	DAG
Config	(GFLOPS)	(%)	size	(GFLOPS)	(%)	(%)	block size	depth
FCFS/R-P	3.75	63.9	512	4.87	29.9	70.8	458.89	2
PL/R-P	4.89	70.9	512	5.84	19.3	77.4	461.11	2
FCFS/F-P	7.59	69.7	512	8.10	6.74	73.7	335.80	3
PL/F-P	8.55	88.4	512	8.80	2.91	92.0	466.00	2
FCFS/EIT-P	8.46	98.5	256	8.52	0.76	99.1	255.19	2
PL/EIT-P	8.74	96.2	512	8.91	2.03	97.7	463.76	2
FCFS/EFT-P	8.77	89.6	512	8.96	2.20	96.2	301.23	3
PL/EFT-P (*)	8.84	91.4	512	9.08	2.75	99.0	352.07	3

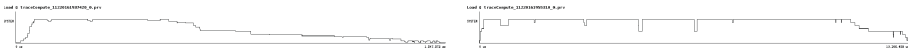
they offer permits the iterative scheduler-partitioner to find finer-grained partitions (see Fig. 6(d)), attaining remarkable net improvements for BUJARUELO (between 13.96 % and 15 %). Note also that bigger performance improvements do not only correspond with lower processor occupancies, but also with higher task DAG depths (up to 5 in BUJARUELO). This observation reinforces the importance of managing arbitrary task granularity, introduced by HeSP, extending the idea of using only two degrees of granularity for two types of processors introduced in other works [8].

This reasoning also applies when comparing the highly heterogeneous BUJARUELO with the less heterogeneous ODROID since the optimal uniform tile size seems to fit better to homogeneous platforms, yielding higher occupancies for all scheduling policies tested, hence leaving less room for non-uniform partitioning improvements. Even with those limitations, HeSP does always provide improvements in all cases.

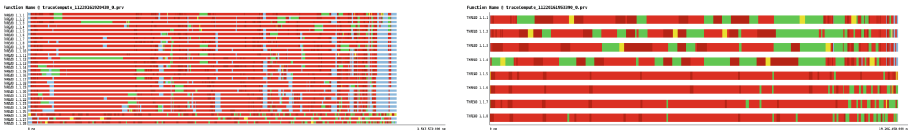
Note the even better improvements, with simpler –i.e. less deep– partitions, attained by our scheme when jointly applied with simpler schedulers –R-P/F-P– and naive FCFS task ordering. Since bad scheduling decisions exhibit a smaller worsening global impact when applied to a bigger set of smaller tasks, task partitions cooperating with a simple scheduler might alleviate its poor performance: under highly heterogeneous scenarios and available resources, it could be safer to partition a task rather than taking the risk of assigning it to the wrong processor.



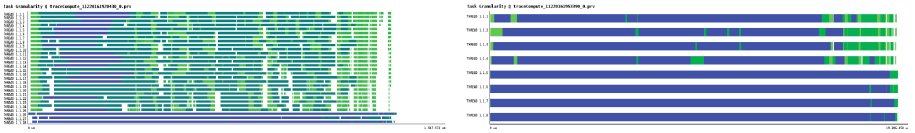
(a) Best uniform partitioning. Task scheduling.



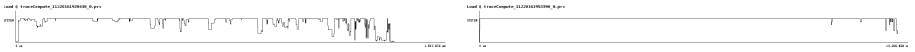
(b) Best uniform partitioning. Compute load.



(c) Best non-uniform partitioning. Task scheduling.



(d) Best non-uniform partitioning. Task granularity.



(e) Best non-uniform partitioning. Compute load.

Fig. 6. Execution traces for the blocked Cholesky factorization on BUJARUELO (left column, $n = 32,768$) and ODROID (right column, $n = 8,192$), using PL/EFT-P. For each case, traces are adjusted to fit the longest execution. In the task scheduling traces, colors correspond to the legend in Fig. 3. (Color figure online)

Figure 6 reports execution traces for the best-performing configurations observed for both architectures[3] (marked in Table 1 with an asterisk), using uniform and non-uniform task partitioning setups. In the traces, each line corresponds to a different processor. In BUJARUELO, (25 CPUs on top, 3 GPUs on bottom), observe the amount of idle times (marked in light blue) in the early and last stages of computation; see how the corresponding best non-uniform schedule fills those gaps by exposing extra parallelism through task partitioning. Concretely, observing the task granularity trace, in which granularity is reported as a gradient (from light green for small tasks to dark blue for large tasks), it is possible to conclude that HeSP is able to refine task granularity only on those stages in which processor occupancy is scarce, improving global performance.

[3] Detailed trace generation is supported by HeSP using Paraver (http://www.bsc.es/computer-sciences/performance-tools/paraver).

The increase in compute load can be observed by comparing the corresponding uniform and non-uniform compute load traces (Figs. 6(b) and (e)).

Similar qualitative results are observed for ODROID, filling the gaps that arise in the same stages on slow cores (top four lines in the trace) with finer-grained tasks. In this case, as was observed in Table 1, the opportunities for improvement are more reduced, but overall performance is also increased by our scheme.

4 Conclusion

In this paper we have presented the HeSP framework and its internal mechanisms towards joint scheduling/partitioning tasks on heterogeneous architectures. Insights reveal that important performance benefits and improved processor loads can be extracted from the framework for a family of scheduling policies. The extracted insights for the Cholesky factorization can be applied to other irregular task-parallel implementations, or to arbitrary heterogeneous architectures.

The static iterative implementation of HeSP has shown to be useful to explore the practical performance bounds of a scheduling-partitioning problem, and it naturally paves the road towards a constructive implementation, in which local information is applied on a per-task basis. This approach can be applied directly on actual task schedulers (e.g. OmpSs) or programming models, in order to introduce in them the recursive task partitioning as an additional degree of freedom. Future work also includes the exploration of more sophisticated scheduling techniques attending not only performance optimization, but also energy consumption on different architectures.

Acknowledgements. This work is funded by project TIN 2015-65277-R (MINECO/FEDER).

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. CC: PE **23**(2), 187–198 (2011)
2. Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguade, E., Labarta, J.: Productive programming of GPU clusters with OmpSs. In: IPDPS 2012, pp. 557–568, May 2012
3. Cojean, T., Guermouche, A., Hugo, A., Namyst, R., Wacrenier, P.: Resource aggregation in task-based applications over accelerator-based multicore machines. Technical report, INRIA (2015)
4. Gautier, T., Ferreira Lima, J.V., Maillard, N., Raffin, B.: XKaapi: a runtime system for data-flow task programming on heterogeneous architectures. In: 27th IPDPS, Boston, May 2013
5. Haidar, A., YarKhan, A., Chongxiao, C., Luszczek, P., Tomov, S., Dongarra, J.: Flexible linear algebra development and scheduling with Cholesky factorization. HPCC **2015**, 861–864 (2015)

6. Planas, J., Badia, R.M., Ayguade, E., Labarta, J.: Self-adaptive OmpSs tasks in heterogeneous environments. In: IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), pp. 138–149, May 2013
7. Topcuoglu, H., Hariri, S., Min-You, Wu: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel and Distrib. Syst. **13**(3), 260–274 (2002)
8. Wei, W., Bouteiller, A., Bosilca, G., Faverge, M., Dongarra, J.: Hierarchical dag scheduling for hybrid distributed systems. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 156–165, May 2015