Investigating Memory Operations Performance in the StarPU Runtime

Lucas Leandro Nesi, Lucas Mello Schnorr
Graduate Program in Computer Science (PPGC/UFRGS), Porto Alegre, Brazil

***Abstract*—Programming parallel applications for heterogeneous HPC platforms is much simpler using the task-based programming paradigm. The applications are modeled as a directed acyclic graph (DAG) of tasks. The simplicity exists because a runtime takes care of all activities usually carried out by the application developer, such as task mapping, process deployment, and load balancing. Besides task scheduling, the runtime is also responsible for handling memory management operations i.e. copying the necessary data to the location where a given task is schedule to execute. Correctly interleaving such memory operations with computation is crucial to achieve high performance, so a worker never wait for data. Poor memory management may be caused by bad scheduling choices or lack of appropriate applications' hints about its tasks dependencies. In this paper, we investigate the CPU-GPU memory management of the StarPU runtime, a well-known task-based middleware for HPC applications. Our results include the identification of poor data handling management when the GPU memory is saturated, ultimately leading to low application performance. Besides fixing the performance issue, we present the design of novel graphical strategies that were fundamental to identify the problem. Our experiments using the dense tiled-based Cholesky factorization show how our fix lead to performance gains of 66% and better scalability for larger input sizes.**

I. Introduction

A challenge found in High Performance Computing (HPC) area is the complexity of programming applications. Especially ones that can intelligently use all the resources' computational power in heterogeneous platforms. The task-based programming paradigm presents some benefits on this matter. It transfers some responsibilities (computation to resource mapping, data management, and communication), that previously relied on the programmers, into a runtime. The task-based applications use a Direct Acyclic Graph (DAG) of tasks as the main structure to schedule them into resources, considering the dependencies and data transfers. Among alternatives like Cilk [1] and Xkaapi [2], StarPU [3] is one example of a runtime using this paradigm. It permits the use of distinct tasks' implementations (CPU, GPU), has different tasks schedulers, and automatically transfers data between resources.

Similar to any other HPC approach, the performance analysis of task-based parallel applications is laborious due to its inherent stochastic nature. StarPU can collect execution's traces that describe the behavior of the application. Different tools can use these traces to help in the performance analysis; one example is the StarVZ workflow [4]. It uses consolidated data science tools and `R` to create trace's visualizations. Various aspects can be studied to improve the application performance, including scheduling decisions, memory transfers, and overall application behavior. In this paper, we focus on the analysis of the memory operations performance, an aspect that has a small presence on performance tools.

The contributions of this paper are the following. **(a)** We include in the StarPU runtime system extra trace information about the memory management activity. It mainly consists of blocks identification on operations, additional attributes on memory chunks, and data coherency states. **(b)** We extended the StarVZ workflow, adding new visual elements, to enable a comprehensive performance analysis over the runtime data management module. For this, we use the previously discussed new traces in StarPU and all available memory management data. **(c)** We present the methodology used with these new features in a real use case application, a dense linear algebra solver called Chameleon. We show how that led to a StarPU software inefficiency discovery, and compare the application performance after our proposed correction patch.

The paper is structured as follows. Section II provides basic concepts on the StarPU runtime system and the application we have used in our experiments: the dense linear algebra Cholesky factorization as implemented by Chameleon/MORSE. Section III presents related work on the visualization of memory management and task-based applications. We also briefly compare ourselves against the state-of-the-art. Section IV introduces the methodology we have devised to investigate memory operations performance in the StarPU runtime, employing modern data science and visualization tools. Section V detail the experiments conducted with the task-based Cholesky factorization, including how we have identified a StarPU software flaw, and the performance increase after the flaw has been corrected. Finally, Section VI concludes this paper with future work.

II. Background Concepts

To explore heterogeneous environments, StarPU allows the tasks to be implemented for different resources, like CPUs, CUDA GPUs, and OpenCL devices. It employs different scheduling heuristics to allocate tasks to resources. Classical heuristics are the `LWS` (local work stealing) and the `DM` (deque model); moreover, more sophisticated schedulers consider additional information. One example is the `DMDA` (deque model data aware); that uses data about the task's dependencies and estimated transfer time to take its decisions [5]. StarPU is responsible for transferring data between resources. For controlling the presence and the coherence of the memory, StarPU attributes a entity to each different resource memory called

memory node. It uses a `modified`, `shared` or `invalid` (MSI) memory system where each data block can assume one of the three states on each memory node [3].

The Chameleon/MORSE package [6] contains a series of solvers for linear algebra implemented on top of StarPU. From the set of available solvers, we adopt the task-based solver that implements the dense linear algebra Cholesky factorization because of its integration as computing phase in many HPC applications. An important factor for our work is that the Cholesky factorization algorithm uses a triangular matrix divided into blocks, and four different tasks: `dpotrf` (Cholesky Factorization), `dgemm` (Matrix Multiplication), `dsyrk` (Symmetric rank-k update), and `dtrsm` (Triangular Matrix Equation Solver). The Cholesky factorization begins applying tasks on lower coordinates blocks and iteratively computes all blocks from all coordinates. From an optimization perspective, the data management can free lower coordinates blocks rapidly because they are no longer required in future iterations.

III. Related Work

Data management views are absent from some performance analysis tools available for StarPU applications, like Temanejo[7] and StarVZ [4], that we are extending in this work. Another tool is Vite [8] that has a classical Gantt chart for the memory nodes' events; however, it is modest and consider asynchronous transfers events as states. The problem is that multiple asynchronous transfers can occur, so the states are misleading. In all cited cases, the tools focus on the tasks scheduling and resources states.

The performance analysis of task-based applications considering the data management or memory interference is present in [9], where the authors presented a task-oriented approach. They analyze the impact of different schedulers on data reuse by tasks on the same resource. Studying the influence on cache misses and other metrics when the sequence of scheduled tasks changes. Data operations focusing in data reuse is also studied by [10]. Where the authors propose a metric called Kernel Reuse Distance (KRD). Also, the authors of [11] analyze the workload for general HPC applications, they state a series of metrics, including data cache, reuse and prefetch for characterizing the applications.

Comparing to the others, our approach is oriented by resources, analyzing the data flow in the heterogeneous environment. We work on a high-level view of the application, using the runtime decisions to explore possible problems. Instead of using low level metrics and comparing them with multiple executions, we focus on the behavior characterization of one execution. We provide some visualization elements that facilitate the performance analysis and enrich our perception of task-based applications running over heterogeneous platforms.

IV. Analyzing the Data Management

Understanding the application's memory flow could lead to optimizations and overall performance improvement. We present our methodology to review the memory manager behavior and memory blocks locality at different resources. The StarPU's data management module is responsible for all actions involving the application's memory; some examples are the data allocation on different accelerators, transfers between inner/inter-node resources, and determine when a memory block can be freed to reopen space for another one. All these actions operate over a specific memory handle that is absent from the original StarPU's trace data. For gathering all the necessary information needed for our performance analysis, we proposed the following extensions to it. We first include the events' memory identification on all events with some extra information to create correlations between activities and to understand the decisions behind it. Second, we add trace events on memory's coherence update function, since it can't be precisely inherited using the current available information. These new events can be used to compute the presence of memory blocks on each memory node.

One classical way to analyze space/time information is applying Gantt charts. We use it to inspect the memory nodes events and do some extensions to it. Visualization present in Figure 1 memory states plot. Each element on the Y axis is a memory node, associated to a node ram or accelerator. The X axis is the time in microseconds. In the example of Figure 1, memory nodes 1 and 2 are for each one of the two GPUs. Memory node 0 is omitted here since lack events during the executions. Each state has a color associated with the action taken. Also, it is possible to visualize the memory blocks coordinates of each handle in the center of the state, useful if a time frame is selected. An example is available on Figure 2, where the allocating actions are over the memory blocks of coordinates 9×10 or 9×15. On the left side of the visualization, a percentage of the most present state is shown.

We propose a blocks' residency over memory nodes visualization. We trace the coherency `MSI` states update and derive the presence of each block on all memory nodes. On Figure 3 we can observe this new visualization for the memory blocks of coordinates 13×8, 13×9, 13×10, 13×11 of the input matrix. For each block, the X axis is the time divided into time intervals of 10 seconds. This interval is sufficiently large for the visualization; yet, small enough to show the application behavior evolution. At each time interval, the Y axis shows the percentage of time that this memory block was on each memory node. Because each block can be present on multiple memory nodes, the maximum residency percentage on Y is bigger than 100%. The maximum percentage is the number of memory nodes times 100, as the memory block can be present on all memory nodes with the `shared` state.

V. Experiments

Experiments were conducted based on preliminary tests that strongly suggested that the Cholesky Factorization, present in the dense linear algebra solver Chameleon, had performance problems, for unknown reasons, when running on certain machines. We apply the methodology of Section IV in this real case scenario to check potential problems related to memory management. First, we use a test case that has idle times problems. The input parameters are block size of

960x960 and 60x60 tiles. The machine used was `tupi`, with a `Intel Xeon CPU E5-2620, 64GB DDR4` memory ram, `2x NVIDIA GeForce GTX 1080ti`.

The Figure 1 presents, from top to bottom, the plots: **(a)** Application Workers, **(b)** Mem Nodes, **(c)** StarPU Workers, **(d)** Ready Tasks, and **(e)** Used memory for this specific condition generated by StarVZ. In **(a)** and **(c)** the Y axes are the workers, in **(b)** the memory nodes, in **(d)** the number of ready tasks, and in **(e)** the overall memory utilization in MB. In all plots, the X axis is the time in milliseconds. Each state has a different color associated with its task or action. The red vertical line, manually added, crossing all plots presents the moment where the used memory reaches a plateau with its maximum value. At that moment, it is possible to check that the GPUs have a lot of idle times and the memory nodes are doing a lot of allocating actions until the end of the application. The GPU idle times of 33% and 32%, present on the left side of the plot **(a)**, are impairing the overall application performance.

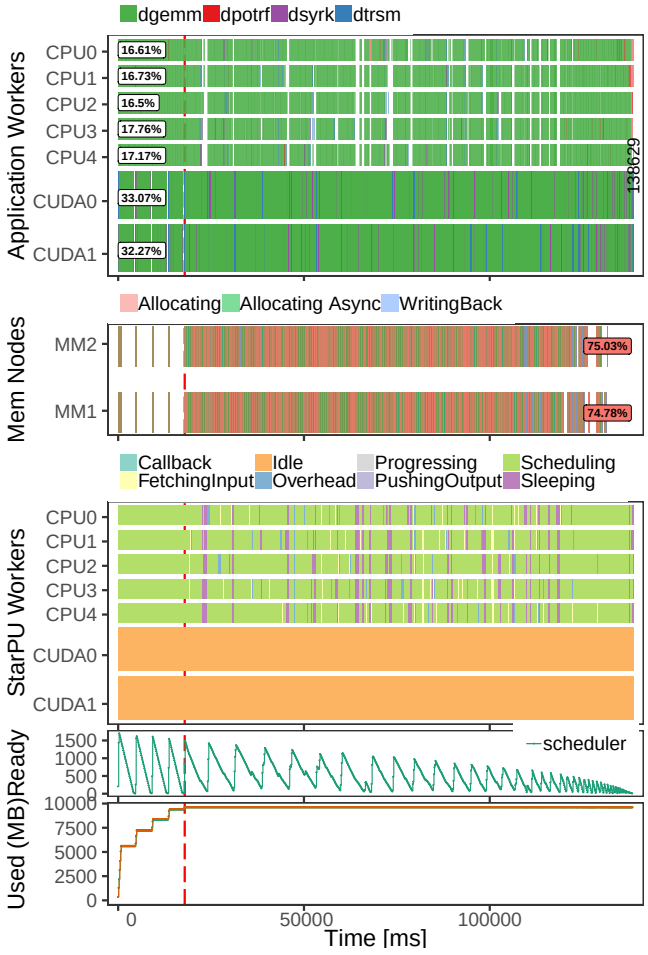

Fig. 1. Multiple Performance Analysis plots.

The possible correlation between idle times and allocation states led us to investigate memory nodes actions after the maximum memory utilization. We select a arbitrary time frame since their behavior is similar after the memory utilization peak. The Figure 2 gives a zoom on the Figure 1 **(b)** plot X axis, and shows for each action the associated memory block coordinates of the input matrix. There are many allocation's states occurring over the same memory blocks, which is a unexpected behavior (repeated allocations for the same memory block). Inspecting the StarPU source code we were able to determine that this happens if allocations fail. Using the GPU resources monitor, we checked that the GPUs are using all the memory. Our hypothesis at this point is that the devices don't have enough memory, but this shouldn't be a problem; as StarPU could free multiple memory blocks, especially those that would no longer be used by the Cholesky factorization.

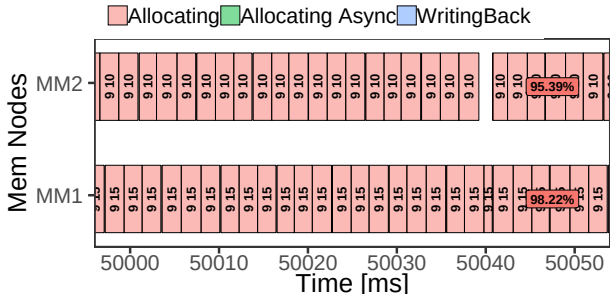

Fig. 2. Memory States on time frame of 50000 ms to 50050 ms.

We use the second visualization presented in Section IV, the blocks' residency, to understand this previosly described behavior. First, we select early used blocks, with lower coordinates, that would be more appropriated cases for being free. Since, As previously discussed, the Cholesky algorithm only uses these tiles in the earlier iterations. On Figure 3, we can see that the blocks became present in all memory nodes at some point, block 13×8 at 50s for example, and remains there until the end of the execution. The presence in all memory nodes indicates that StarPU is deciding not to free these blocks.

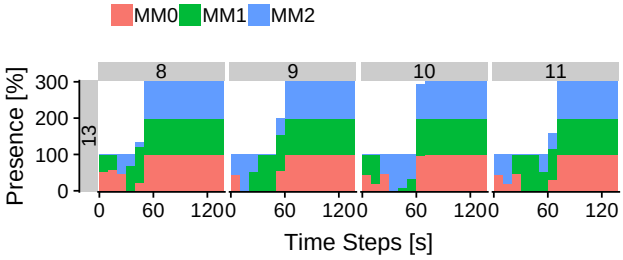

Fig. 3. Time presence (%) of the blocks (13, 8-11) in each memory node.

All these insights gave us enough information to inspect decisions directly. We use `gdb` to check StarPU's functions that free unused memory blocks. We found out that StarPU believed that it had free space on the GPUs. Also, we detected a huge difference when comparing the internal StarPU's used memory values to the ones given by the GPU resources monitor. We discovered that the CUDA function `cudaMalloc` could allocate more memory than the requested size. The function rounds the demanded memory to a device dependent page size. In the case of the `GTX 1080ti`, it is 2MB. It makes a block with 7200 KB to use 8192 KB; over-allocating

992 KB per block and 1800 MB per matrix (in the 60x60 blocks case). StarPU was wrongly calculating the used size on the resources and kept calling the expensive `cudaMalloc` function even with the GPU memory full. We then proposed a correction patch for StarPU, and compare the performance of the Cholesky application before and after it.

We executed 10 experiments for each configuration, to tackle experimental variability, using different matrices sizes and patch's versions. The input size distribution has more points around and after the memory limit when the real matrix size don't fit on GPU. Also, we use the following parameters: block size of 960x960, `DMDA` scheduler. The Figure 4 presents the performance comparison between StarPU's versions. The Y axis is the GFlops performance as reported by the Cholesky application with a 99% confidence interval. The X axis is the matrix width in cells. The gray line is the threshold where the matrix size fits on the GPU memory; considering the rounding behavior, number of blocks and the CUDA driver used memory. Two different StarPU versions are used. The red line is the **original** version (commit `be5815e`), and the blue line is our **corrected** version (commit `ca3afe9`). The **original** version has a performance drop after the memory threshold, failing from the relatively constant ~690 GFLOPs to lower values depending on the matrix size. However, the **corrected** version constantly keeps its performance on the ~690 GFLOPs mark. Demonstrating the effectiveness of our fix, keeping the program scalable as the input size increases.

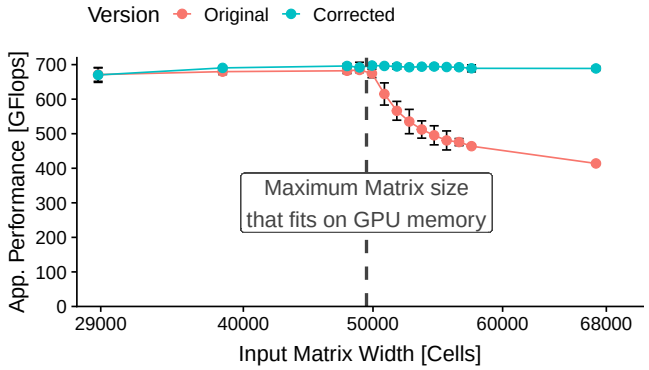

Fig. 4. Application performance (GFLOPs) before/after the patch.

VI. Conclusion

In this paper, we presented a visual performance analysis of heterogeneous task-based applications' memory management running over StarPU. We add new trace events on StarPU and use it in the new methodology incorporated into the StarVZ workflow. Since all methods rely on the runtime's features, any StarPU application can use this memory management analysis. The examination of memory management could lead to useful insights and possibly discovery of performance problems. We presented a real use case scenario using the dense linear algebra solver Chameleon. Using this methodology, we discover a performance problem (GPUs with high idle times) when the input matrix's size was higher than the GPUs' total memory.

The new visualizations allowed to verify that several slow allocations' states were occurring in the memory nodes; along with the unnecessary presence of memory blocks in limited memory resources. The problem was that StarPU computes the memory used by each resource only considering the size request for allocation; yet, in the case of CUDA GPUs, the `cudaMalloc` function can reserve more memory than the requested. In a case study, it led to a deviation of 1.8 GB between the real used memory and the runtime's registered, causing misleading decisions. Our proposed solution, to solve this StarPU's problem, resulted in a performance sustain independent of the input matrix size and if it fit in GPU's memory. For future work, we consider the analysis of the memory of other applications, trying to find optimizations on both the runtime and application.

Acknowledgements

We would like to thank Samuel Thibault and Luka Stanisic for the insights and the discussions about this work. We also thank these projects for supporting this investigation: FAPERGS GreenCloud (16/488-9), the FAPERGS MultiGPU (16/354-8), the CNPq 447311/2014-0, the CAPES/Brafitec EcoSud 182/15, and the CAPES/Cofecub 899/18.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, 1996.

[2] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IEEE Intl. Symposium on Parallel and Distributed Processing*, 2013.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Conc. and Comp.: Pract. and Exp., SI: Euro-Par 2009*, vol. 23, pp. 187–198, 2011.

[4] V. G. Pinto, L. M. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A visual performance analysis framework for task-based parallel applications running on hybrid clusters," *Concurrency and Computation: Practice and Experience*, 2018.

[5] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *16th International Conference on Parallel and Distributed Systems*, Shangai, China, Dec. 2010. [Online]. Available: https://hal.inria.fr/inria-00523937

[6] E. Agullo, G. Bosilca, B. Bramas, C. Castagnede, O. Coulaud, E. Darve, J. Dongarra, M. Faverge, N. Furmento, L. Giraud, X. Lacoste, J. Langou, H. Ltaief, M. Messner, R. Namyst, P. Ramet, T. Takahashi, S. Thibault, S. Tomov, and I. Yamazaki, "Poster: Matrices over runtime systems at exascale," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, H. Wasserman, Ed., Nov 2012.

[7] R. Keller, S. Brinkmann, J. Gracia, and C. Niethammer, "Temanejo: Debugging of thread-based task-parallel programs in starss," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 131–137.

[8] C. Kevin, F. Mathieu, and J. Johnny., "Visual trace explorer (ViTE)."

[9] G. Ceballos, T. Grass, A. Hugo, and D. Black-Schaffer, "Analyzing performance variation of task schedulers with taskinsight," *Parallel Computing*, vol. 75, pp. 11 – 27, 2018.

[10] M. Pericàs, A. Amer, K. Taura, and S. Matsuoka, "Analysis of data reuse in task-parallel runtimes," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Cham: Springer International Publishing, 2014, pp. 73–87.

[11] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov, "Characteristics of workloads used in high performance and technical computing," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ser. ICS '07. New York, NY, USA: ACM, 2007, pp. 73–82.