Efficient synchronization for stencil computations using dynamic task graphs

Zubair Wadood Bhattia, Roel Wuytsb,c,a, Pascal Costanzab,d, Davy Preuveneersa, Yolande Berbersa

Abstract

Executing stencil computations constitutes a significant portion of execution time for many numerical simulations running on high performance computing systems. Most parallel implementations of these stencil operations suffer from a substantial synchronization overhead. Furthermore, with the rapidly increasing number of cores these synchronization costs keep rising. This paper presents a novel approach for reducing the synchronization overhead of stencil computations by leveraging dynamic task graphs to avoid global barriers and minimizing spin-waiting, and exploiting basic properties of stencil operations to optimize the execution and memory management. Our experiments show a reduction in synchronization overhead by at least a factor four when compared to state-of-the-art stencil compilers like Pochoir and Patus.

Keywords: Scientific computing, parallel programming, model of computation, stencil computations, synchronization;

1. Introduction and problem statement

Because of their relevance in high-performance computing, there is an ongoing interest in optimizing stencil computations, in particular for current multicore hardware. At the forefront of these effort are stencil compilers such as Pochoir[1] and Patus[2] that take a stencil description (written in a domain specific language) and generate optimized multithreaded code. While these state-of-the-art stencil compilers and auto-tuning frameworks are very effective in increasing the arithmetic intensity, our results show that their parallel versions require quite a lot of synchronization. With the number of hardware cores continuously increasing the cost of synchronization is becoming higher and higher. Therefore it has become evermore important to minimize the synchronization cost.

Figure 1 shows an often-used but naive parallel implementation of a 2D five-point stencil that uses parallel_for. The top loop cannot be parallelized using parallel_for due to the dependencies of successive iterations, whereas, the iterations of the second loop (line 2) are independent of each other. Therefore, it is possible to safely parallelize them, as shown in figure 1. However, this causes over-synchronization by placing an all-to-all synchronization barrier in each iteration of the top loop.

*Corresponding author. E-mail address: zubairwadood.bhatti@cs.kuleuven.be
This paper proposes a novel approach that minimizes synchronization overhead for stencil computations to result in faster execution. We model stencil computations as dynamic task graphs and execute them with a workstealing scheduler. By construction our approach avoids global barriers (synchronization is only point-to-point) and because tasks are only created when they are ready to be executed there is minimal spin-waiting. Furthermore we exploit basic properties of stencil computations for memory management and optimization of the execution of dynamic task graphs.

We have implemented our approach on top of the latest version of (Threading Building Blocks) TBB library, with its support for Flow Graphs. The TBB flow graph library has low-level infrastructure that we leverage to express and execute our dynamic task graph stencil computations. Our results show that we reduce synchronization by at least a factor of four when compared to state-of-the art stencil compilers Pochoir[1] and Patus[2]. It is important to note that our technique is complementary to the other optimizations (e.g. cache blocking) used in stencil compilers, that could therefore benefit from our approach.

The rest of the paper is structured as follows. Section 2 explains how we express stencil operations as a dynamic task graph and execute them very efficiently. Our experimental results are shown in section 3 and compared with the state-of-the art. Section 4 presents our conclusions.

2. Stencil computations as dynamic task graphs

A task graph is a directed acyclic graph where the nodes represent computational tasks and the edges represent dependencies between them. These dependencies are either data-flow dependencies or simply precedence constraints. Contrary to a simple spawn-sync model of computation where only parent-child dependencies may be expressed, task graphs allow arbitrary dependencies between any pair of tasks. Tasks become ready for execution when messages arrive at all their incoming edges. After finishing execution the task sends messages on all its outgoing edges. All tasks can be executed in parallel with each other as long as their precedence constraints are fulfilled and sufficient resources are available to execute them.

The execution schematics of a dynamic task graph (DTG) [3] are similar to a conventional task graph; however, it allows creation and deletion of tasks and edges on the fly. Dynamic task graphs cannot be scheduled statically off-line and must be scheduled dynamically at runtime. Task-graphs synchronize through point-to-point messages and do not require any barrier synchronizations. We express stencil computations as dynamic task graphs and use the known properties of stencil operations to optimize their execution and memory management.

2.1. Task creation

During initialization tasks are created for the first two timesteps to solve regions of the matrix e.g. tiles for a 2D matrix and cubes for a 3D matrix. These initial tasks then create other tasks (along with their incoming
edges) working on the same regions for the subsequent timesteps. All tasks are created after their predecessors and two timesteps in advance, as shown in figure 2. By creating tasks only two timesteps in advance we remove the possibility that a task is created after one of its predecessors has finished execution.

Proof: Let $X_t$ be a task that processes the region $X$ in timestep $t$, $P_{X_t}$ and $S_{X_t}$ are the sets of its immediate predecessors and successors respectively. $Create(X_t)$, $Start(X_t)$ and $Finish(X_t)$ are the creation, starting and finishing times of the task $X_t$. For any stencil of a constant shape and order $S_{X_t} \equiv P_{X_t+2}$. If the task $X_t$ creates the task $X_{t+2}$, $Create(X_{t+2}) < Finish(X_t)$. $VT \in S_{X_t}$, $Start(T) > Finish(X_t)$. Therefore, $Create(X_{t+2}) < Start(T)$, $VT \in S_{X_t}$, thus, $Create(X_{t+2}) < Finish(T)$, $VT \in P_{X_{t+2}}$.

2.2. Scheduling

Each processor runs an Intel Threading Building Blocks (TBB) work-stealing thread. Each thread has a pool (work queue) of processes waiting for execution. A thread executes processes from its own work queue as long as it is not empty. When the work queue of a thread is empty, it randomly steals tasks from the pools of other threads. A task becomes ready for execution after it has received messages on all its incoming edges. When a task becomes ready for execution it is spawned as a TBB task in a work queue. For example, when the task $X_t$ in figure 2 finishes execution it sends a message to all of its successors ($W_{t+1}, X_{t+1}, Y_{t+1}$). Now the task $Y_{t+1}$ has a message on all its incoming edges, therefore it becomes ready for execution and is spawned as a TBB task on a work queue. We stress that a task only becomes ready after all of its predecessors have finished and sent their message; therefore, no synchronization is required during its execution.

2.3. Memory allocation and deallocation

We have taken care to preserve data locality, even in the presence of TBB’s work-stealing scheduler and its “Breadth-First Theft and Depth-First Work” strategy [4]. TBB’s approach can result in the loss of the data locality property when a TBB child task gets stolen by a thread executing on a different socket. If this happens it is important to reallocate the region to the new processor, essentially letting the data follow the task and reestablish data locality. For any stencil operation the results produced by the task $X_t$ are no longer required after start of execution of the task $X_{t+2}$, because $S_{X_t} \equiv P_{X_{t+2}}$. If $X_t$ and $X_{t+2} \in X_t$ execute on the same processor, $X_{t+2}$ simply overwrites the results of $X_t$ (data locality is kept), whereas, if they execute on different processors $X_{t+2}$ deletes the results of $X_t$ and locally allocates space for its own results (reestablishing data locality).
3. Results and discussion

We implemented a five point stencil for a 2D Laplacian heat equation shown in figure 1 as a dynamic task graph using the Intel TBB library [4]. The performance of our implementation is compared to the state-of-the-art stencil compiler Pochoir[1] and the auto-tuning framework Patus[2] along with a naive approach using a parallel for loop. Two different experiments were performed using different domain sizes and processors.

We used the Intel® VTune™ profiler locks-and-waits analysis to measure the synchronization overhead. Figure 3 presents a comparison of the synchronization overhead of the naive, Pochoir and DTGs version for the two experiments. The Pochoir version requires less synchronization compared to the naive approach and Patus because the hyper-trapezoidal decomposition used in Pochoir allows the computation of several timesteps without synchronization. Moreover, the synchronization occurs hierarchically within groups of zoids; therefore, this is not an all-to-all synchronization. However, the DTG version requires on average four times less synchronization than Pochoir for both experimental settings.

![Figure 3. Comparison of synchronization, showing a 4X+ reduction in synchronization costs for both experiments. (1) On a domain of 400x400 double precision floats for 800 timesteps, using an Intel® Xeon® E7-4870 processor and (2) a domain size of 4000x4000 and 200 timesteps on a dual socket Intel® Xeon® X5660 system.](image)

4. Conclusion

This paper presents an approach to implement stencil computations as dynamic task graphs in order to minimize synchronization overhead. The approach we propose avoids global barriers (synchronization is only point-to-point) and because tasks are only created when they are ready to be executed there is minimal spin-waiting. Our experiments show that this significantly reduces the synchronization overhead when compared to the state-of-the-art stencil compiler and auto-tuning framework (Pochoir and Patus).

Acknowledgements

This research is partially funded by the Research Fund KU Leuven, Intel and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). The experiments were performed at the Intel ExaScience Lab, Belgium.

References