2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)

A Dynamic Task-based D3Q19 Lattice-Boltzmann Method for Heterogeneous Architectures

João V. F. Lima[†], Gabriel Freytag[*], Vinicius Garcia Pinto[*], Claudio Schepke[‡], Philippe O. A. Navaux[*]
[*]*Universidade Federal do Rio Grande do Sul, Brazil*
[†]*Universidade Federal de Santa Maria, Brazil*
[‡]*Universidade Federal do Pampa, Campus Alegrete, Brazil*
jvlima@inf.ufsm.br, gfreytag@inf.ufsm.br, vgpinto@inf.ufrgs.br,
claudioschepke@unipampa.edu.br, navaux@inf.ufrgs.br

***Abstract*—Nowadays computing platforms expose a significant number of heterogeneous processing units such as multicore processors and accelerators. The task-based programming model has been a *de facto* standard model for such architectures since its model simplifies programming by unfolding parallelism at runtime based on data-flow dependencies between tasks. Many studies have proposed parallel strategies over heterogeneous platforms with accelerators. However, to the best of our knowledge, no dynamic task-based strategy of the Lattice-Boltzmann Method (LBM) has been proposed to exploit CPU+GPU computing nodes. In this paper, we present a dynamic task-based D3Q19 LBM implementation using three runtime systems for heterogeneous architectures: OmpSs, StarPU, and XKaapi. We detail our implementations and compare performance over two heterogeneous platforms. Experimental results demonstrate that our task-based approach attained up to 8.8 of speedup over an OpenMP parallel loop version.**

1. Introduction

Nowadays computing platforms expose a significant number of heterogeneous processing units such as multicore processors and accelerators. Large-scale applications from the industry usually require mixing different parallelization paradigms to exploit such machines at their full potential. Designing such parallel applications that support multiple paradigms in a portable and efficient way is challenging.

Several libraries or languages are designed to improve programming productivity. A parallel construct, such as parallel loops from OpenMP, enables to identify potential concurrent instruction sets, an approach commonly used in numerical applications. The main drawback of a *for each* construct is the addition of strong synchronization points to enforce the completion of a set of independent tasks, with the associated memory update, before the execution of a new task set. This addition of artificial synchronization points limits the performance of several algorithms as the classical block matrix factorizations (Cholesky, LU, QR) where parallelism between tasks across the outer iterations exists [1].

On the other hand, the task-based model simplifies programming by unfolding parallelism based on data-flow dependencies between tasks. Runtime systems with support for data-flow programming are nowadays *de facto* standard for parallel linear algebra libraries on multi-cores [2]. Besides, they are suitable for heterogeneous architectures since it considers heterogeneity concerning computing power and disjoint address spaces [3], [4], [5].

The Lattice-Boltzmann Method (LBM) is an iterative numerical method to model and simulate fluid dynamics properties, where space, time and velocity are discrete. Depending on the number of dimensions and propagation directions, an LBM problem may demand high memory and processing power and requires significant computation power to be simulated in a computationally acceptable time. Many parallel implementations of LBM over various HPC systems can be found in literature such as MPI [6], OpenMP [7], hybrid MPI/OpenMP [8], MPI/CUDA [9], hybrid MPI/OpenMP/CUDA [10]. Although some implementations use OpenMP tasks [11], most research is devoted to OpenMP *parallel loop* solutions. However, to the best of our knowledge, no dynamic task-based strategy of LBM has been proposed to exploit CPU+GPU computing nodes.

In this paper, we present a dynamic task-based D3Q19 LBM implementation for heterogeneous architectures using three runtime systems: OmpSs, StarPU, and XKaapi. We detail our implementations and compare performance over heterogeneous platforms equipped with 2 GPUs. The contributions of this paper are:

- We present a task-based D3Q19 LBM algorithm with 3D space partitioning for heterogeneous architectures;
- We evaluated our algorithm on top of three runtime systems: OmpSs, StarPU, and XKaapi;
- Experimental results demonstrate that a task-based approach is able to exploit heterogeneous CPU+GPU platforms efficiently with up to 8.8 speedup over a OpenMP parallel loop version.

The remainder of the paper is organized as follows. Section 2 presents the related works on LBM and task-based algorithms. Section 3 gives an overview of the D3Q19 LBM

2377-5750/19/$31.00 ©2019 IEEE
DOI 10.1109/PDP.2019.00023

CPS Conference Publishing Services

method. Section 4 describes the runtime systems used in our experiments. We detail our task-based LBM algorithm in Section 5. Section 6 presents the experimental hardware and methodology used. Our experimental results are presented in Section 7. Finally, Section 8 and Section 9, respectively, present the discussion and conclude the paper.

2. Related Work

Most research on parallel LBM methods is devoted to traditional multicore architectures with OpenMP. In [12] the authors evaluate the performance of a parallel LBM in a multicore processor. In [13] the authors compare the performance of the OpenMP LBM implementation against the performance of an implementation based on a functional language called SequenceL. The authors in [14] describe the LBM-IB library for shared-memory architectures and compared the performance of an OpenMP LBM over a Pthreads based implementation, both cube-based implementations.

Other studies have target parallel LBM on accelerators. In [15], the authors developed an optimized LBM implementation for distributed GPUs. In [16] they study two 2D refinement methods, one based on Multi-Domain and one based on Irregular meshing, and target two strategies, GPU and CPU+GPU, on each refinement method. In [17] the authors proposed a hybrid MPI/OpenMP LBM with 2D partitioning and several optimizations over a CPU+MIC architecture.

Some authors report task-based implementations of LBM without data dependencies. Ye et al. [18] parallelized the flow simulation of the Entropic Lattice-Boltzmann Method (ELBM) for the D3Q19 model in a heterogeneous CPU+GPU platform with OpenMP and CUDA, designing a task-level parallelism mechanism based in a task queue. In [7] the authors implemented a task-level pipeline of the D2Q9 LBM model using the Intel Threading Building Blocks (TBB), which outperformed an OpenMP implementation. Meadows and Ishikawa [11] developed two task-based implementations from an OpenMP+MPI optimized implementation of the Lattice Quantum Chromodynamics (QCD) method. One of the implementations was developed with OpenMP tasks and the other was developed with hand-coded tasks. Both task-based implementations outperformed the original implementation.

Some studies on scientific applications have achieved performance gains through task-based algorithms with data dependencies. In [19] the authors present a parallel task-based application to simulate the propagation of seismic waves (Ondes3D) with StarPU over heterogeneous architectures. In [20] the study presented a task-based parallelization of the Fast Multipolar Method (FMM) and showed that the tasks and dependencies introduced in the last version of the OpenMP API may allow performance improvement. Finally, in [21] the authors presented a benchmark for hybrid data flows and shared memory architectures showing that the data-flow models based in data dependency graphs provide greater flexibility than task dependency graphs, which in turn provide better programmability and performance.

3. Lattice-Boltzmann Method

The Lattice-Boltzmann Method (LBM) is a numerical method for fluid flow simulations and fluid physics modeling. It is frequently adopted as an alternative technique for computational simulations of Fluid Dynamics instead of using discrete Navier-Stokes equations solvers [6] or other conventional numerical schemes based on discretizations of macroscopic continuum equations [22].

In the LBM, space, time and velocity of the particles are considered discrete. A lattice is formed by discrete points, each one with a fixed number of discrete displacement directions and at each iteration, particles realize a space displacement among the lattice points, enabling simulations of physical properties of fluid flows in a simple way.

In this paper, we consider a three-dimensional lattice structure with 18 propagation directions, as shown in Fig. 1. As the propagation can be null, one more propagation direction is added to the eighteen directions, resulting in a D3Q19 structure.

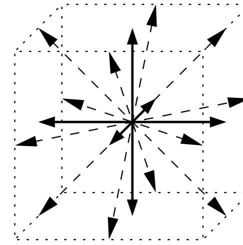

Figure 1. D3Q19 lattice geometry.

The possible directions, velocity and ω weight are defined by [6]:

- A static point at coordinate $(0, 0, 0)$, where the particle has zero velocity. The value of ω_i in this case is $1/3$.
- Six nearest directions $(-1, 0, 0)$, $(+1, 0, 0)$, $(0, -1, 0)$, $(0, +1, 0)$, $(0, 0, -1)$ and $(0, 0, +1)$, with unity velocity and $\omega_i = 1/18$.
- Twelve diagonal line neighbors $(1, 1, 0)$, $(-1, 1, 0)$, $(1, -1, 0)$, $(-1, -1, 0)$, $(1, 0, 1)$, $(-1, 0, 1)$, $(1, 0, -1)$, $(-1, 0, -1)$, $(0, 1, 1)$, $(0, -1, 1)$, $(0, 1, -1)$ and $(0, -1, -1)$, with velocity $\sqrt{2}$ and $\omega_i = 1/36$.

To deal with the collisions against the boundaries, a mechanism called Bounce-back is used [6]. It consists in the inversion of the speed vectors directions each time that a collision occurs against static points in the boundary. This method prevents the forces leaving, returning them to the fluid.

4. Task-based Runtime Systems

Task parallelism is considered a well-suited programming model for heterogeneous architectures since parallelism is explicit and detection of synchronizations is implicit due to data dependencies between tasks. A task-based

algorithm unfolds a directed acyclic graph (DAG) or a data-flow graph (DFG) whenever dependencies between tasks and data are considered [23]. Besides, this model favors fine granularity and asynchronism that are essential in order to exploit parallelism, reduce idleness and improve scalability in modern architectures.

One of the key features in a "heterogeneous-ready" runtime is task multi-versioning that requires two versions of the same code: one for CPU and one for GPU. One of the first papers to mention this concept [24] associates a signature to tasks that must be respected by all implementations. This signature includes task parameters and their access modes (read or write). Therefore, there is a clear separation between a task definition and its architecture specific implementations.

Besides, another fundamental feature is memory management of disjoint address spaces offering an abstraction layer similar to a distributed shared memory (DSM). A runtime system needs to manage memory transfers based on data dependencies between tasks. For instance, a GPU task would require memory allocation and data transfer to the GPU memory for each parameter before execution. Throughout execution, it may require a transfer from GPU memory to main memory if the result of this task is read by another one.

Programming interfaces such as OmpSs, StarPU, and XKaapi, described below, enable executions on multi-CPU and multi-GPU systems simultaneously.

4.1. OmpSs

OmpSs [5] is a programming model to exploit task-level parallelism by OpenMP-like pragmas and a runtime system to schedule tasks while preserving dependencies. OmpSs does not rely on a library API to write a program and the user depends on the Mercurium compiler. The OmpSs runtime, called Nano++, offers different scheduling strategies, and most results are reported on a centralized scheduling strategy with data locality on multi-CPU, multi-GPU, and clusters.

Figure 2 illustrates an example of task multi-versioning on top of OmpSs. An algorithm may call the method *TaskCompute* and the runtime will identify two task versions of this method by directive *target* followed by a *task* directive with data-flow dependencies on created tasks. The clause *device* identifies the task's target (*smp* or *cuda*), in addition to clause *implements* that gives an alternative version for the specified target device.

```
1  #pragma omp target device (smp) copy_deps
2  #pragma omp task inout([BS]A)
3  void TaskCompute(double c, double* A, int BS)
4  { /* CPU implementation */ }
5
6  #pragma omp target device(cuda) implements(TaskCompute) \
7    copy_deps
8  #pragma omp task inout([BS]A)
9  void TaskComputeCUDA(double c, double* A, int BS)
10 { /* GPU implementation */ }
```

Figure 2. Example of task multi-versioning with OmpSs annotations.

4.2. StarPU

StarPU is a runtime system providing a data management facility and an unified execution model over heterogeneous architectures including GPUs and Cell BE processors [4]. StarPU programming model relies on explicit parallelism by tasks with data dependencies and a memory layer to abstract transfers among disjoint address spaces. In addition, the StarPU runtime also provides a set of scheduling policies from dynamic work balance or based on performance models.

Figure 3 illustrates an example of the StarPU API. The *starpu_codelet* structure describes a task with two implementations *compute_cpu_func* and *compute_cuda_func* for CPU and GPU, respectively, and its data dependencies. This task is instantiated and submitted inside the method *compute*. The example makes use of a data handle to register this memory pointer in StarPU that will manage its coherence between different address spaces.

```
1  static void compute_cpu_func(void *descr[], void *arg)
2  { /* CPU implementation */ }
3
4  static void compute_cuda_func(void *descr[], void *arg)
5  { /* GPU implementation */ }
6
7  static starpu_codelet cl = {
8    .where = STARPU_CPU | STARPU_CUDA,
9    .cpu_func = compute_cpu_func,
10   .cuda_func = compute_cuda_func,
11   .nbuffers = 1,
12   .modes = { STARPU_RW }
13 };
14
15 void compute(double* A, int n){
16   starpu_data_handle handle;
17   starpu_vector_data_register(&handle, 0, A, n,
18     sizeof(double));
19
20   struct starpu_task *compute_task= starpu_task_create();
21   scal_task->cl = &cl;
22   scal_task->handles[0] = handle;
23
24   int ret = starpu_task_submit(compute_task);
25   starpu_task_wait_for_all();
26   starpu_data_unregister(handle);
27 }
```

Figure 3. StarPU program example with multi-versioning.

4.3. XKaapi

The XKaapi task model [25], as in Cilk [26], enables non-blocking task creation: the caller creates the task and proceeds with the program execution. The semantic remains sequential such as XKaapi's predecessors Athapascan [23] and KAAPI [25]. XKaapi has several APIs (C, Fortran, C++) to program heterogeneous parallel architectures. In this paper, code fragments rely on the C++ API.

An XKaapi program is a sequential code complemented with annotations or runtime calls to create tasks. Parallelism is explicit, while the detection of synchronizations is

implicit: the dependencies between tasks and the memory transfers are automatically managed by the runtime. A task is a function call that returns no value except through its effective parameters. Tasks are created by calling the template function *ka::Spawn*.

The extensions to the C++ interface provide a high-level interface for multi-versioning a task implementation [24]. A task implementation for GPU (respectively CPU) is the specialization of the class *TaskBodyGPU* (respectively *TaskBodyCPU*). At least one implementation is expected per task signature (*TaskCompute* in the example). The code fragment of Figure 4 illustrates how to program a multi-version task using the C++ API. The *ka::Spawn<TaskCompute>* creates a task of type *TaskCompute*. The data type *range_1d* is an abstraction to view a memory region as a 1D array.

```
1  struct TaskCompute: public ka::Task<2>::Signature<
2    double, ka::RW<ka::range1d<double> > >{};
3
4  template<>
5  struct TaskBodyCPU<TaskCompute> {
6    void operator()( double c, ka::range1d_rw<double> A)
7    { /* CPU implementation */ }
8  };
9
10 template<>
11 struct TaskBodyGPU<TaskCompute> {
12   void operator()( double c, ka::range1d_rw<double> A)
13   { /* GPU implementation */ }
14 };
15
16 ka::range1d<double> A(pA, 256);
17 ka::Spawn<TaskCompute>()(1.0, A);
18 ka::Sync();
```

Figure 4. Example of an XKaapi C++ task. It shows a task *Signature* with its parameters and access modes, as well as a CPU and GPU implementation.

XKaapi runtime schedules tasks by work-stealing strategy with extensions for task multi-versioning and concurrent GPU operations [3]. The main difference between XKaapi scheduler and other runtime systems is that XKaapi computes data-flow dependencies only when an idle thread searches for a ready task. The scheduler moves the cost of computing ready tasks from the work performed by the victim during task's creations to the steal operations performed by thieves.

5. Lattice-Boltzmann Task-based Implementation

Our implementation is based on the LBM blocked partitioning described by [6]. The global lattice is divided according to the number of partitions defined at execution time for each 3D axis direction (x, y, and z). A block has information about velocity, obstacles, and buffers for neighbor transfers. Each task will operate on a block of the lattice. Figure 5 shows the task-based algorithm of our implementation. It has 23 different tasks overall, except for StarPU implementation with additional exchange tasks for each neighbor direction.

First, task *initialize* determines the initial conditions for all points in the block. Its data-flow arguments are the sub-lattice in write-only mode and initial density values in read-only. The main loop of the method follows, which creates four computational tasks and data exchange tasks at each time step. Task *redistribute* has a sub-lattice as read-write argument and calculates the macroscopic density and speed. Next, in the *propagate* task, the physical properties are redirected to neighbor points.

1: Initialize parameters
2: **for** Each block $N_x \times N_y \times N_z$ **do**
3:    Create task INITIALIZE conditions
4: **end for**
5: **for** Each time step **do**
6:    **for** Each block $N_x \times N_y \times N_z$ **do**
7:       Create task REDISTRIBUTE
8:       Create task PROPAGATE
9:       **for** Each propagation direction **do**
10:          Create task to save border velocities to buffer
11:          Create task to update border from neighbor buffer
12:       **end for**
13:       Create task BOUNCEBACK
14:       Create task RELAXATION
15:    **end for**
16: **end for**

Figure 5. The task-based algorithm of the D3Q19 LBM method.

Tasks devoted to data transfers are responsible to perform data exchanges between neighbor blocks through sub-lattice buffers. For each propagate direction, a block generates two tasks: a task to read border velocities of a specific direction from the sub-lattice and save to a buffer of its own block; and a task to update the sub-lattice borders from neighbor buffers. Figure 6 shows a 3D division of the lattice by $3 \times 3 \times 3$ along with the orthogonal buffers.

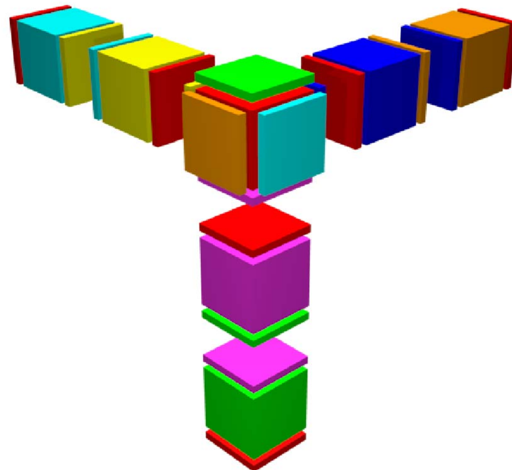

Figure 6. A 3D lattice division with orthogonal buffers.

The last two tasks of the main loop are *bounceback* and *relaxation*. Task *bounceback* deals with boundary conditions and inverts the speed directions when collissions occur against static points of the boundary. In the end, task *relaxation* emulates the shock among the particles. Both tasks receive as data-flow argument the sub-lattice of its block in read-write mode.

Figure 7 illustrates a DAG from OmpSs with two time steps and $2 \times 2 \times 2$ domain division. Tasks for data transfers were grouped for the sake of simplicity.

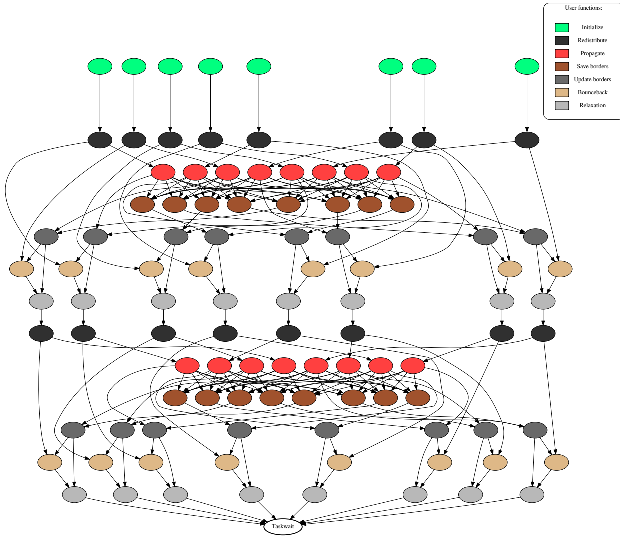

Figure 7. DAG of the task-based LBM algorithm for two timesteps.

6. Tools and Methods

This section details the hardware configuration, the runtime systems, and methodology used in our experiments.

6.1. Evaluation Platform

We target an experimental platform enhanced with two GPU accelerators called CHIFFLET.

CHIFFLET is composed of two NUMA nodes with one Intel Xeon E5-2680 v4 (Broadwell) processor each (total 2 processors) and 14 cores per processors (28 cores total) running at 2.4 GHz or 3.3 GHz with Turbo Boost, and 768 GB of main memory. It is enhanced with 2 NVIDIA GTX 1080 Ti GPUs (Pascal architecture) of 3,584 CUDA cores running at 1.58 GHz each (7,168 CUDA cores total) with 12 GB GDDR5X of main memory per GPU. We used as software environment the GCC compiler 6.3.0 and NVIDIA CUDA 9.1.

The compilation flags were `-O3`, `-march=native`, `-mavx`, and `-arch=sm_60`.

6.2. Software Description

We evaluated the following configurations of each runtime system:

- OmpSs version 17.12 with two scheduler algorithms: default *breadth-first* (bf) that implements a global ready queue, and *work-first* (wf) that has a local ready queue per thread;
- StarPU version 1.2.4 with two scheduling strategies: the Deque Model Data Aware (DMDA) scheduler that maps tasks to workers using a history-based performance model, and Work Stealing (WS) scheduler.
- XKaapi version `bd0b1bf31`[1] with work stealing scheduler [3];.

All three runtime systems dedicate one CPU thread to manage a GPU worker. In the context of our experimental platform, an execution with all processing units in Chifflet should have 26 CPU workers and 2 GPU workers.

6.3. Methodology

All executions were composed of two steps: the first allocates memory for blocks and its sub-lattices and reads the obstacle file; the second step is the computation. We report execution time only from the computation step. Each result is a mean of 30 executions with the same initial conditions and obstacles. The results with StarPU had a warm-up phase of 1 run, which were not included in the mean values, to calibrate its history-based performance model. The 99% confidence interval is represented on the graphs by a vertical line around the mean values.

All experiments have three-dimensional lattice structures with a rectangular obstacle placed in the canal at the first third of the x-axis, as illustrated in Figure 8. In addition, we fixed the number of time steps at 200 iterations.

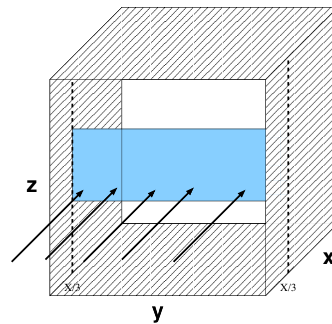

Figure 8. Rectangular obstacle of the experiments.

The number of partitions was chosen according to the best execution time for values 2, 4, and 8 in all 3D axis. Lattice sizes from $256 \times 256 \times 256$ had a partition value of 8 but OmpSs due to a runtime error.

1. Available at: http://kaapi.gforge.inria.fr

7. Experimental Results

The goal of our experiments is to evaluate the performance of our dynamic task-based LBM algorithm over a heterogeneous platform enhanced with GPUs. Our objectives are:

- Evaluate performance of our algorithm over a heterogeneous platform in terms of execution time;
- Assess the impact of different runtime systems on a task-based algorithm that demands high memory footprint and processing power;
- Analyse the performance gain over a CPU-only strategy with OpenMP parallel loops at each computational phase.

Figure 9 shows performance results of the LBM task-based algorithm increasing the lattice size and varying the number of GPUs. The StarPU version attained better performance in all cases, followed by XKaapi and OmpSs. StarPU scheduler based on Work Stealing outperformed DMDA by up to 18.91% with size $512 \times 512 \times 512$ over 2 GPUs. Nonetheless, StarPU had significant standard variation on both scheduling strategies, impacting the confidence interval. The maximum standard deviation of StarPU was 24.94% with DMDA scheduler at $448 \times 448 \times 448$ over 2 GPUs.

XKaapi had better results than OmpSs, but did not scale with the addition of GPUs. OmpSs with work-first scheduler showed performance results similar to XKaapi on 2 GPUs. The scheduling strategies of OmpSs had similar results for CPU-only cases; but work-first attained performance gains of 20.78% with 1 GPU and 32.11% with 2 GPUs over CPU-only results of OmpSS using work-first and input size $512 \times 512 \times 512$. The breadth-first scheduler of OmpSs did not perform as expected and did not scale with additional GPUs. StarPU WS scheduler reduced execution time by up to 4% with 1 GPU and 29.87% with 2 GPUs over CPU-only results of StarPU with the same scheduler and input size $512 \times 512 \times 512$.

Figure 10 illustrates speedup results of our task-based algorithm with 2 GPUs over an OpenMP parallel loop version of LBM executed with 28 CPU threads. Our approach obtained speedup over OpenMP for all input sizes. Speedup at input size $512 \times 512 \times 512$ over OpenMP was 3.65 by OmpSs breadth-first, 5.42 by OmpSs work-first, 5.25 by XKaapi, 7.19 by StarPU DMDA, and 8.87 by StarPU WS.

In order to analyse the impact of scheduling strategies, Figure 11 illustrates a Gantt diagram obtained from StarPU comparing both DMDA and WS strategies for 10 iterations and input size $512 \times 512 \times 512$. DMDA had poor efficiency on CPUs and they were frequently idle, with maximum occupancy of 62% on a CPU thread. On the other hand, WS efficiently distributed tasks over CPUs and GPUs with a CPU occupancy of at least 93%.

8. Discussion

Our experimental results provide evidence that our dynamic task-based approach with LBM can efficiently exploit heterogeneous architectures. The algorithm obtained speedup over an OpenMP parallel loop version in all cases.

Two runtime features proved to be essential on our implementation of the D3Q19 LBM model: software cache and scheduling over multiple architectures. First, the software cache manages disjoint address spaces and is able to evict memory when GPU memory is full. Second, scheduling over CPUs and GPUs enables the execution of tasks on both worker types simultaneously in order to compute a common problem. The speedup analysis demonstrated that a CPU+GPU approach was more efficient than a multi-threaded strategy.

We can note that different scheduling strategies clearly impact performance although we implemented the same algorithm. The StarPU WS strategy outperformed its DMDA history-based performance model due to bad decisions of DMDA algorithm for large workloads [27] such as the D3Q19 LBM. XKaapi and OmpSs work-first strategies had similar results due to the work stealing scheduler that was able to harness CPU workers efficiently. Despite the high-level programming model of OmpSs, our experiments suggest that its scheduling strategy (breadth-first) was not efficient.

In the context of programming models, OmpSs has an elegant approach compared to StarPU and XKaapi with incremental parallelism through a set of compiler directives and library routines. A naive code analysis using the Lines of Code Without Comments (NLOC) metric shows that OmpSs implementation has 2438 NLOC, which is less than StarPU (3992 NLOC) and XKaapi (3442 NLOC). Still, these implementations significantly surpass NLOC metrics of OpenMP (707 NLOC).

9. Conclusion

In this paper, we present a dynamic task-based D3Q19 LBM implementation using three runtime systems for heterogeneous architectures: OmpSs, StarPU, and XKaapi. Experimental results demonstrate that a task-based approach is able to exploit heterogeneous platforms efficiently. Our algorithm obtained up to 8.8 speedup over an OpenMP parallel loop version.

Future works include more experimental evaluations on performance for large lattice sizes, analysis of energy consumption, and algorithm extensions to distributed platforms.

Acknowledgment

Experimental results presented in this paper were carried out in part using the Grid'5000 testbed, being developed under the Inria ALADDIN development action with support from CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 15–44, 2010.

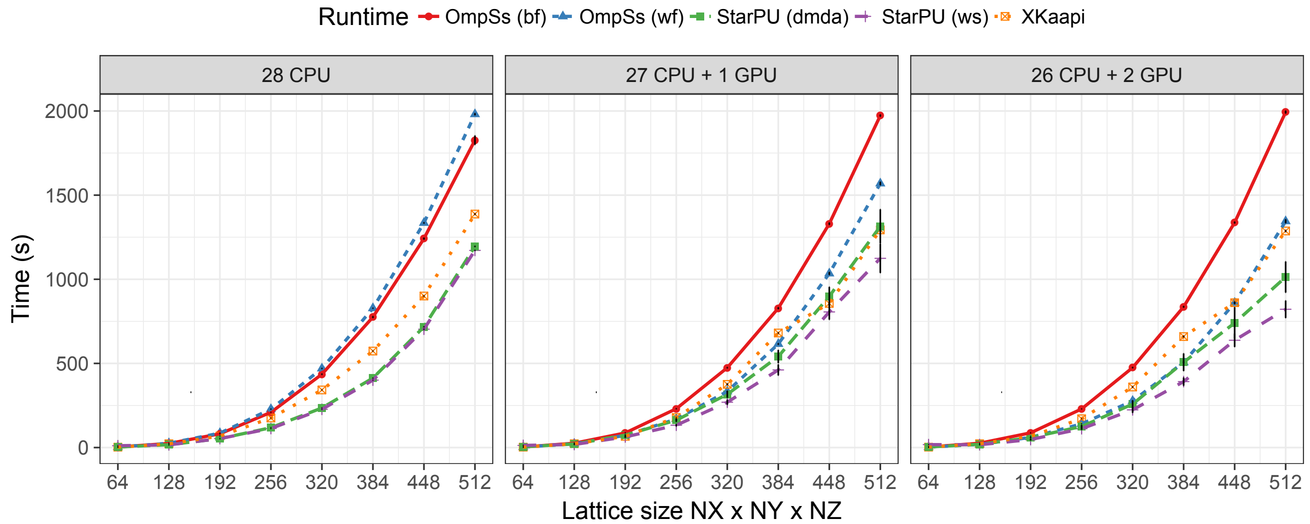

Figure 9. Execution time of LBM over all CPUs (left), 1 GPU (center) and 2 GPUs (right) on Chifflet.

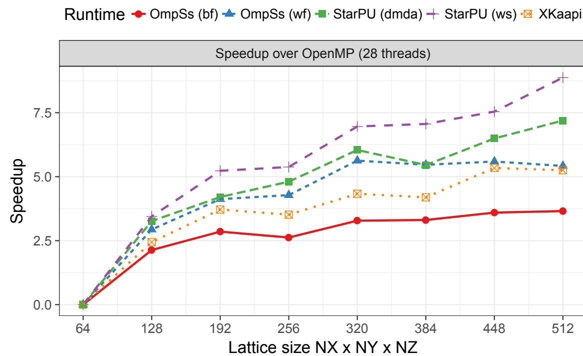

Figure 10. Speedup results of the task-based LBM algorithm with 26 CPUs plus 2 GPUs over OpenMP (28 threads).

[2] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, Jan. 2009.

[3] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 1299–1308.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[5] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of gpu clusters with ompss," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 557–568.

[6] C. Schepke, N. Maillard, and P. O. A. Navaux, "Parallel lattice boltzmann method with blocked partitioning," *International Journal of Parallel Programming*, vol. 37, no. 6, pp. 593–611, 2009.

[7] H. Başağaoğlu, J. R. Harwell, H. Nguyen, and S. Succi, "Enhanced computational performance of the lattice Boltzmann model for simulating micron- and submicron-size particle flows and non-Newtonian fluid flows," *Computer Physics Communications*, vol. 213, pp. 64–71, 2017.

[8] J. R. Clausen, D. A. Reasor, and C. K. Aidun, "Parallel performance of a lattice-Boltzmann/finite element cellular blood flow solver on the IBM Blue Gene/P architecture," *Computer Physics Communications*, vol. 181, no. 6, pp. 1013–1020, 2010.

[9] J. Kraus, M. Pivanti, S. F. Schifano, R. Tripiccione, and M. Zanella, "Benchmarking GPUs with a parallel Lattice-Boltzmann code," *Proceedings - Symposium on Computer Architecture and High Performance Computing*, pp. 160–167, 2013.

[10] C. Feichtinger, J. Habich, H. Köstler, U. Rüde, and T. Aoki, "Performance modeling and analysis of heterogeneous lattice boltzmann simulations on cpu–gpu clusters," *Parallel Computing*, vol. 46, pp. 1 – 13, 2015.

[11] L. Meadows and K.-i. Ishikawa, "OpenMP Tasking and MPI in a Lattice QCD Benchmark," in *International Workshop on OpenMP*. Springer, 2017, pp. 77–91.

[12] W. Zhou, Y. Yan, X. Liu, and B. Liu, "Lattice boltzmann parallel simulation of microflow dynamics over structured surfaces," *Advances in Engineering Software*, vol. 107, pp. 51–58, 2017.

[13] H. Başağaoğlu, J. Blount, J. Blount, B. Nelson, S. Succi, P. M. Westhart, and J. R. Harwell, "Computational performance of SequenceL coding of the lattice Boltzmann method for multi-particle flow simulations," *Computer Physics Communications*, vol. 213, pp. 92–99, 2017.

[14] P. Nagar, F. Song, L. Zhu, and L. Lin, "LBM-IB: A parallel library to solve 3D fluid-structure interaction problems on manycore systems," *Proceedings of the International Conference on Parallel Processing*, vol. 2015-December, pp. 51–60, 2015.

[15] E. Calore, A. Gabbana, J. Kraus, E. Pellegrini, S. F. Schifano, and R. Tripiccione, "Massively parallel lattice–boltzmann codes on large gpu clusters," *Parallel Computing*, vol. 58, pp. 1–24, 2016.

[16] P. Valero-Lara and J. Jansson, "Heterogeneous cpu+gpu approaches for mesh refinement over lattice-boltzmann simulations," *Concurrency and Computation: Practice and Experience*, 2016.

[17] P. Tang, A. Song, Z. Liu, and W. Zhang, "An Implementation and Optimization of Lattice Boltzmann Method Based on the Multi-Node CPU+MIC Heterogeneous Architecture," *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, no. 1, pp. 315–320, 2016.

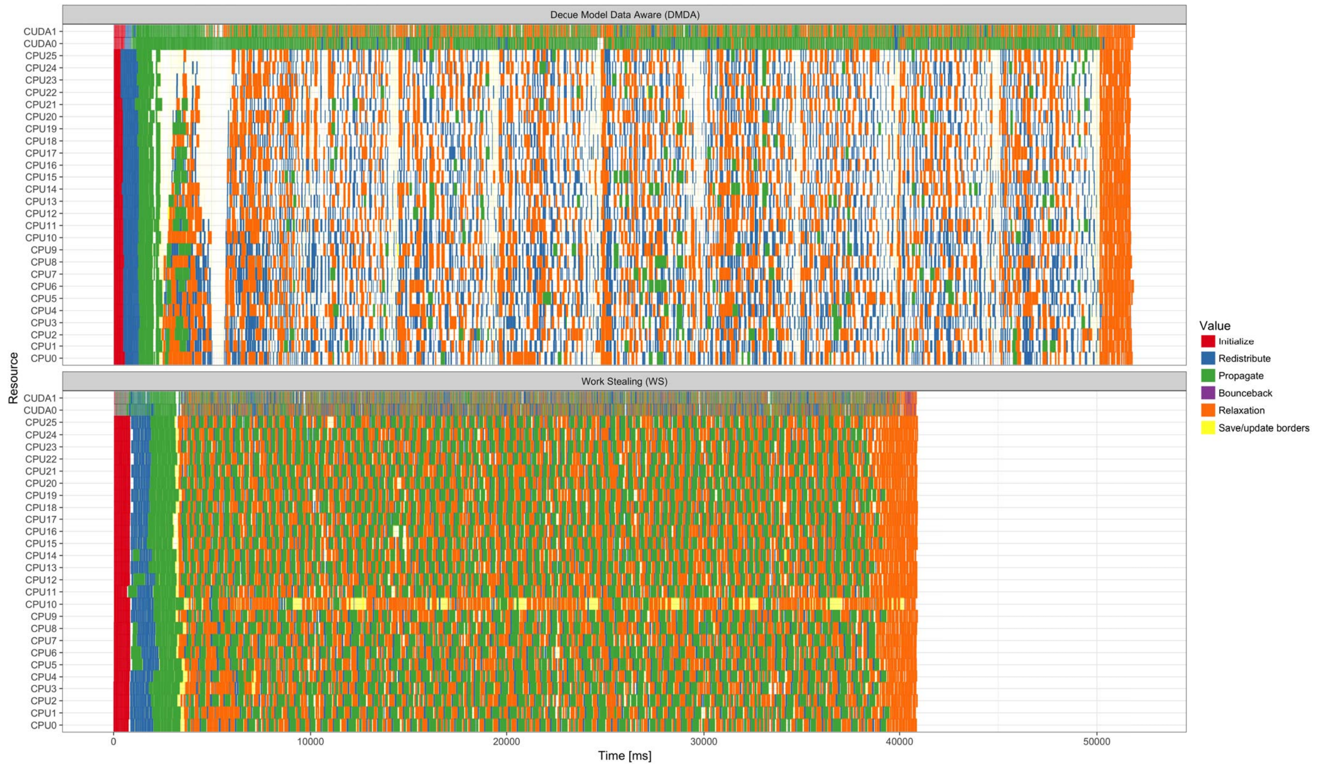

Figure 11. StarPU gantt chart from LBM using DMDA (top) and Work Stealing (bottom) scheduling with lattice size 512x512x512 and running the first 10 iterations.

[18] Y. Ye, K. Li, Y. Wang, and T. Deng, "Parallel computation of entropic lattice boltzmann method on hybrid CPU-GPU accelerated system," *Computers and Fluids*, vol. 110, pp. 114–121, 2015.

[19] V. Martínez, D. Michéa, F. Dupros, O. Aumage, S. Thibault, H. Aochi, and P. O. A. Navaux, "Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system," in *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2015, pp. 1–8.

[20] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset, "Bridging the gap between openmp and task-based runtime systems for the fast multipole method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2794–2807, 2017.

[21] V. Gajinov, S. Stipić, I. Erić, O. S. Unsal, E. Ayguadé, and A. Cristal, "Dash: A benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models," in *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 2014, p. 4.

[22] S. Chen and G. D. Doolen, "Lattice Boltzmann Method for Fluid Flows," *Annual Review of Fluid Mechanics*, vol. 30, no. 1, pp. 329–364, 1998.

[23] F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille, "Athapascan-1: On-line building data flow graph in a parallel language," in *Proc. of PACT'98*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 88–95.

[24] E. Hermann, B. Raffin, F. c. Faure, T. Gautier, and J. Allard, "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations," in *Proc. of the 2010 Euro-Par*, vol. 6272. Springer, 2010, pp. 235–246.

[25] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proc. of PASCO'07*. London, Canada: ACM, 2007.

[26] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[27] V. G. Pinto, L. Stanisic, A. Legrand, L. M. Schnorr, S. Thibault, and V. Danjean, "Analyzing dynamic task-based applications on hybrid platforms: An agile scripting approach," in *2016 Third Workshop on Visual Performance Analysis (VPA)*, Nov 2016, pp. 17–24.