LBM-HPC an Open-Source Tool for Fluid Simulations. Case Study: Unified Parallel C (UPC-PGAS)

Pedro Valero-Lara
CFD & Computational Technology Unit
Basque Center for Applied Mathematics (BCAM)
Bibao, Spain
pvalero@bcamath.org

Johan Jansson
CFD & Computational Technology Unit and
Computational Technology Laboratory
Basque Center for Applied Mathematics (BCAM) and
KTH Royal Institute of Technology
Bilbao, Spain and Stockholm, Sweden
jjansson@bcamath.org, jjan@csc.kth.se

***Abstract*—The main motivation of this work is the evaluation of the Unified Parallel C (UPC) model, for Boltzmann-fluid simulations. UPC is one of the current models in the so-called Partitioned Global Address Space paradigm. This paradigm attempts to increase the simplicity of codes and achieve a better efficiency and scalability. Two different UPC-based implementations, explicit and implicit, are presented and evaluated. We compare the fundamental features of our UPC implementations with other parallel programming model, MPI-OpenMP. In particular each of the major steps of any LBM code, i.e., Boundary Conditions, Communication, and LBM solver, are analyzed.**

***Keywords*-Lattice-Boltzmann Method; Parallel Computing; CLUSTER; Partitioned Global Address Space (PGAS); Unified Parallel C (UPC);**

I. Introduction

In order to increase the efficiency and scalability of distributed-memory programming environments, many attempts have been made over the past several years. One such programming enviroment consists of the Partitioned Global Address Space (PGAS) [1] model. This model provides a shared memory view of distributed memory systems, hiding the most important aspects regarding parallel programming by facilitating the implementation of parallel codes and providing better efficiency and scalability. Recently, some PGAS languages have arisen, such as Coarray Fortran, Unified Parallel C [2], the Global address space Programming Interface (GPI), among others.

The Lattice Boltzmann Method (LBM) is a clever discretization of the Boltzmann equation [3]. Due to the particular features of LBM, it has been adapted to numerous parallel computer architectures, [4], [5]. Also, given the growing popularity of LBM, multiple tools have arisen which have consolidated this method into academia and industry. In particular, we have considered the open-source LBM-HPC framework [6].

A recent work which covers a subject closely related to the present contribution is the one by [7], where they analyze the performance of matrix-vector multiplication and LBM via GPI and MPI-OpenMP programming models. This study shows that the performance achieved by both approaches is similar. Here, we will focus on another PGAS-based model, UPC. Also, we study two different UPC approaches, explicit and implicit. Besides, the three parts of any LBM code, boundary conditions (BCs), communication and LBM solver, are studied individually.

This paper is structured as follows: Section II briefly describes the UPC framework and the different implementations. In Section III, a performance study is carried out to compare the computational efficiency of each implementation; finally, in Section IV some conclusions are outlined.

II. Implementation

Unified Parallel C (UPC) is an extension of C, see [2] for a complete review. It focuses on providing low-latency, high speed communication routines for large scale systems.

Next, we briefly introduce the main features of the UPC programming model. In UPC, there are two thread monitoring expressions, `THREADS`, a macro gives the total number of threads, and, `MYTHREAD`, an integer expression gives the index of the current thread whose range is `0 ...THREADS-1`. To declare a global array accessible by all threads, it must include the tag `shared`. Although data is accessible by every thread, every part of the data has an "affinity". The affinity can be controlled by using the block-size qualifier.

```
shared type variable;
shared [block_size] type variable [index];
shared [ ] type variable [index];
shared [*] type variable [index];
```
(1)

When there is no block-size qualifier (default), elements are distributed such that the ith element has affinity to thread `i%THREADS`. If block-size is specified, the ith element has affinity to thread `(i/block_size)%(THREADS×block_size)`. This is an efficient approach for exploiting

locality. All elements have affinity to thread 0, considering the `[ ]` qualifier. Finally, `[*]` qualifier is commonly used for multidimensional arrays. The elements, `variable[i]`, have affinity to `i%THREADS`, regardless of other dimensions.

Parallelization in UPC is carried out from the use of `MYTHREAD` and `THREADS` (explicit), or the parallel loop statement, `upc_forall` (implicit). The usage of `upc_forall` is equivalent to a normal for loop, except that it automatically divides the loops among the threads according to a fourth parameter (affinity):

```
upc_forall(initial;test;increment;affinity)
```

The affinity parameter can be either a pointer, or an integer expression. The integer expression distributes the loop iterations over the set of threads with respect to (`affinity%THREADS`). This expression identifies the id of the UPC thread, which computes that iteration. If it is a pointer, threads will execute those iterations in which the address referenced by the affinity parameter is stored into local memory.

A. MPI-OpenMP

Our hybrid MPI-OpenMP approach is a classic implementation in which the loops of the main LBM steps (stream and collision) are computed by using OpenMP, and the boundaries, among the different local blocks, are transfered between the computational nodes, by using MPI_send/receive calls. The communication is managed by using a scheme based on the use of ghost cells. The ghost cells consists of replicates of the borders of all immediate neighbors. These ghost cells are not updated locally, but provide stencil values when updating the borders of local blocks.

B. Unified Parallel C

Our first UPC implementation follows an explicit approach; it consists of using the block-size qualifier `[*]`, which distributes one square fluid-block per thread. To implement this scheme, it is necessary to use an additional dimension, whose size is equal to number of threads to be executed. (Algorithm 1).

All the major steps are carried out by accessing memory positions directly, whether local or remote (in other nodes).

Similarly to the MPI-OpenMP implementation, our Explicit-UPC approach implements a ghost-cell based scheme. It requires a higher use of memory space, however in contrast to the MPI-OpenMP implementation, our UPC implementation is not in need of using MPI-type explicit calls (MPI_Send/Recv) or data buffers for sending or receiving data.

Our second UPC implementation follows an implicit approach; this approach increases the transparency of those aspects related to parallel programming, reducing considerably the cost of the implementation with respect to the two previous approaches.

Algorithm 1 UPC explicit.

```
1: // UPC_x(_y) number of UPC threads in x(y)-direction
2: // MT = MYTHREAD
3: int main(int argc, char **argv){
4:   mid_x = MT%UPC_x;
5:   mid_y = floor(MT/UPC_x);
6:   static shared [*] double u[THREADS][Ly][Lx],v,p;
7:   static shared [*] double f1[THREADS][Ly][Lx][9], f2;
8:   t=0;
9:   while(t<It){
10: /*————————— Boundary Conditions —————————*/
11:    // Top boundary
12:    if(mid_y == 0){
13:     for(j=0; j<Lx; j++){
14:      u[MT][0][j] = uMax; v[MT][0][j] = v0; p[MT][0][j] = p0;
15:      for(z=0; z<9; z++){
16:       cu=3*(cx[z]*uMax+cy[z]*v0);
17:       f=p0*w[z]*(1.+cu+1./2.* (cu)^2) ...
18:       f1[MT][0][j][z] = f;
19:      } } ... // Other boundary conditions
20: /*——————————— Communication ———————————*/
21:    // From bottom to top
22:    if(mid_y > 0 && UPC_y > 1){
23:     for(j=0; j<Lx; j++){
24:      for(z=0; z<9; z++){
25:       f1[MT][0][j][z]=f1[MT-UPC_x][Ly-2][j][z];
26:      } } } ... // Other directions
27: /*————————————— LBM —————————————*/
28:    for(i=1; i<Ly-1; i++){
29:     for(j=1; j<Lx-1; j++){
30:      u_local=0.;v_local=0.;p_local=0.;
31:      for(z=0; z<9; z++){
32:       new_i=i-cx[z]; new_j=j-cy[z];
33:       ftmp[z]= f1[MT][new_i][new_j][z];
34:      } ... } }
35:    t++;
36:   }
37: }
```

In this case, the block-size qualifier for shared memory variables is set to `[Lx*Ly/THREADS]` by following an implicit scheme .In contrast with the previous UPC implementation, Blocks of rows are assigned to each thread. As we see later, the implicit affinity-distribution has a significant impact on performance. As a consequence of using the implicit approach is that the maximum number of UPC-threads to be executed is limited by the vertical size (number of rows) of our fluid-domain.

This approach (Algorithm 2) uses a `upc_forall` constructor with a pointer as the affinity parameter, which forces the thread which have the pointer `&u[i][0]` stored into local memory to be executed by these iterations. The communication layer is packed in the LBM solver, being totally invisible from a programmer's point of view.

Here the variables `Lx` and `Ly` refer to the whole fluid-domain, not to local domain per thread, as in the explicit approach.

Algorithm 2 UPC implicit.

```
1: #DEFINE CHUNK Lx*Ly/THREADS
2: int main(int argc, char **argv){
3:     static shared[CHUNK] double u[Ly][Lx],v,p;
4:     static shared[CHUNK] double f1[Ly][Lx][9],f2;
5:     t=0;
6:     while(t<It){
7: /*————————— Boundary Conditions —————————*/
8:         upc_forall(i=0; i<Ly; i++; &u[i][0]){
9:             for(j=0; j<Lx; j++){
10:            // Top boundary
11:            if(i==0){
12:                u[i][j] = uMax; v[i][j] = v0; p[i][j] = p0;
13:                for(z=0; z<9; z++){
14:                    cu=3*(cx[z]*uMax+cy[z]*v0);
15:                    f=p0*w[z]*(1.+cu+1./2.* (cu)^2) ...
16:                    f1[i][j][z] = f;
17:            } } ...// Other boundary conditions
18:        }
19: /*—————————-LBM and Communication—————————-*/
20:        upc_forall(i=1; i<Ly-1; i++; &u[i][0]){
21:            for(j=1; j<Lx-1; j++){
22:                u_local=0.; v_local=0.; p_local=0.;
23:                for(z=0; z<9; z++){
24:                    new_i=i-cx[z]; new_j=j-cy[z];
25:                    ftmp[z]=f1[new_i][new_j][z];
26:                } ...} }
27:        t++;
28:    }
29: }
```

III. Performance Analysis

To critically evaluate the performance of the different approaches implemented, next, we carry out a scaling study over Hornet, a new supercomputer system at the High Performance Computing Center Stuttgart (HLRS). It is a Cray XC40 system, based on Intel Haswell processors and Cray Aries interconnect technology. The specific features are given in Table I.

Platform	Hornet (Cray XC40)
Cabinets	21
# Compute nodes	3944
# Compute cores	94656 (24 cores per node)
# Processor	Intel Xeon CPU E5-2680 v3 (30M Cache, 2.50 GHz)
# cores/processor	12
Total compute memory	5.4 PB (128 GB per node)
Node-node interconnect	Cray Aries (Dragonfly topology)
Peak performance (TOP 500)	3786 TeraFLOPS

Table I
DETAILS OF THE EXPERIMENTAL PLATFORMS.

We analyze the strong scaling. Our LBM problem consists of a size of around 1000 million lattice-nodes. All tests have been performed using double precision. Our test-bed is a well-known and standard case for fluid simulations, the *lid-driven cavity flow* [8].

The three major steps of any LBM code over distributed memory platforms, BCs, communication, and LBM, are studied individually. As the Implicit-UPC exhibits some particular features, we include a separate subsection for its study. At the end of this section, we present an overview which includes all major steps. The number of cores considered in this study ranges from 240 to 30720.

A. Boundary Conditions

First, we analyze the performance for computing BCs (Figure 1 ■). Both, MPI-OpenMP and Explicit-UPC approaches share the same workload distribution, i.e. uniform square blocks are distributed to each of the MPI processes or UPC threads. This distribution is well-balanced for this kind of operations. However, for the Implicit-UPC implementation, a set of rows is assigned to each of the threads. It supposes an important unbalancing, as UPC threads, which compute the top and bottom regions of the fluid-domain, represent a much higher computational cost, with respect to the rest of threads.

The best behavior, in terms of strong scaling, is found in the hybrid MPI-OpenMP approach. The trend presented by the Explicit-UPC approach is similar, being the fastest for the first tests carried out (from 240 to 960 number of cores). The unbalancing, caused by the implicit affinity, provides the worst scenario for computing BCs, since increasing the number of UPC threads does not reduce the cost associated to the top and bottom boundaries, which is a consequence of the workload distribution.

B. Communication

We lack information about the performance of the Implicit-UPC implementation, as it integrates both, the LBM solver and the communication, without using explicit memory transfer calls (Figure 1 ▲).

Although the other approaches have a similar behavior, the Explicit-UPC approach exhibits better results with respect to the hybrid MPI-OpenMP approach. However, the benefit achieved in the first tests is reduced considerably by increasing the number of cores. In terms of speedup, time consumed by the memory transfers of the MPI-OpenMP approach over the Explicit-UPC approach, the benefit of using the PGAS-based implementation evolves from 11.45 (240 cores) to 3.57 (30720 cores).

C. LBM solver

Next, we study the results obtained from the hybrid MPI-OpenMP and the Explicit-UPC approaches when executing LBM. Except in the first tests (from 240 cores to 1920 cores), in which the time consumed by the Explicit-UPC approach is slightly inferior, both implementations exhibit an equivalent result for the rest of tests.

D. Implicit-UPC

The Implicit-UPC compacts LBM solver and communication, thus, to analyze the performance of this approach, we compare the results for our MPI-OpenMP approach and Explicit-UPC approach with the Implicit-UPC approach while executing both, communication and LBM. The speedup of using Implicit approach against Explicit, ranges from 1.12, when 3840 cores are used, to 2.19, when we consider 30720 cores. The speedup obtained by the Implicit-UPC against the MPI-OpenMP is even more significant, which ranges from 1.26, for 1920 cores, to 2.45, for 30720 cores. Given these results, we can assume that the overhead caused by the memory transfers into Implicit-UPC is equivalent or even smaller than in the case of Explicit-UPC.

E. Overview

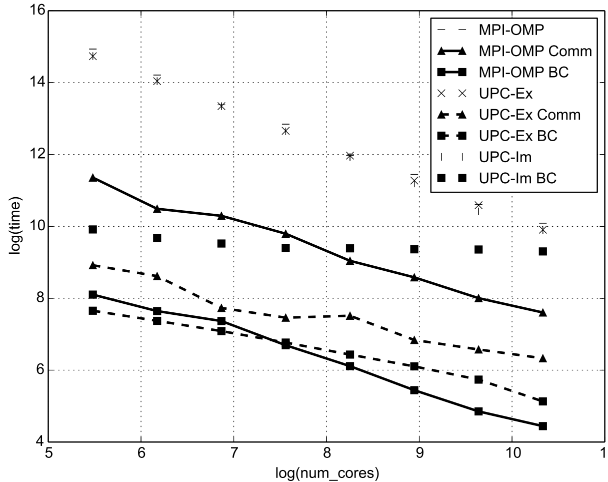

Figure 1. Overview for the hybrid MPI-OpenMP, Explicit-UPC, and Implicit-UPC approaches.

As Figure 1 illustrates, the contribution of the BCs in both, MPI-OpenMP and Explicit-UPC, is similar, being slightly inferior for MPI-OpenMP. On the other hand, the contribution of the memory communication is considerably superior in MPI-OpenMP than in Explicit-UPC. Obviously, the higher the number of cores (computational nodes) considered the higher the communication overhead. The overhead caused by the memory transfers for the MPI-OpenMP ranges from 2% to 7.5% over the total time, and from 0.3% to 1.8% for the Explicit-UPC. The dominant part in Implicit-UPC is the BCs computation; which ends up being the main bottleneck. As we have previously introduced, this is mainly caused by the workload distribution provided by the implicit model. Finally, the peak saved time for the Implicit-UPC and Explicit-UPC corresponds to 27% and 19%, respectively. Despite the great overhead caused by the BCs overhead, the Implicit-UPC is presented as the most efficient approach among the tested approaches. However, the overall speedup reached is degraded considerably, when considering 30720 cores, in which the benefit of using the Implicit-UPC is equivalent to that of using the Explicit-UPC.

IV. Conclusion

For conclusions, we highlight the benefit of using UPC against MPI-OpenMP model is mainly reflected in an important reduction of the communication overhead. Although, the Implicit-UPC performance overtakes the performance shown by the other two approaches, when considering the LBM and communication steps, the benefit reached in these two steps is degraded by the overhead caused by the BCs. Despite this, the Implicit-UPC is proven to be the fastest approach, reaching a peak benefit around 27% and 15%, in terms of execution time, with respect to the MPI-OpenMP and the Explicit-UPC counterparts, respectively.

Acknowledgements

This research was supported by the Basque Government through the BERC 2014-2017 program, by Spanish MINECO: BCAM Severo Ochoa excellence accreditation SEV-2013-0323. The author would like to thank the computing facilities of High Performance Computing Center Stuttgart (HLRS) at the University of Stuttgart.

References

[1] "Partitioned global address space." [Online]. Available: http://www.pgas.org/

[2] "Berkeley upc - unified parallel c." [Online]. Available: http://upc.lbl.gov/

[3] S. Succi, *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond (Numerical Mathematics and Scientific Computation)*, ser. Numerical mathematics and scientific computation. Oxford University Press, USA, Aug. 2001.

[4] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias, "Accelerating solid-fluid interaction using lattice-boltzmann and immersed boundary coupled simulations on heterogeneous platforms," *Procedia Computer Science*, vol. 29, no. 0, pp. 50 – 61, 2014, 2014 International Conference on Computational Science.

[5] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, "Scalable lattice boltzmann solvers for cuda gpu clusters," *Parallel Computing*, vol. 39, no. 67, pp. 259 – 270, 2013.

[6] "Lbm-hpc." [Online]. Available: http://www.bcamath.org/en/research/lines/CFDCT/software

[7] F. Shahzad, M. Wittmann, M. Kreutzer, G. H. Thomas Zeiser, and G. Wellein, "Pgas implementation of spmvm and lbm using gpi." *Proceedings of the 7th International Conforence on PGAS Programming Models (PGAS)*, 2013.

[8] M. A. Mussa, S. Abdullah, C. S. Azwadi, N. Muhamad, and K. Sopian, "Numerical simulation of lid-driven cavity flow using the lattice boltzmann method," pp. 236–240, 2008.