Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

LBM and SPH Scalability Using Task-based Programming

Jan Christian Meyer[a,*], Tufan Arslan[a], Jørgen Valstad[b], Janusa Ragunathan[b], Nick Brown[c], Luis Cebamanos[c]

[a]High Performance Computing section, IT Dept, NTNU
[b]Department of Computer Science, NTNU
[c]Edinburgh Parallel Computing Centre (EPCC)

Abstract

Computational Fluid Dynamics encompasses a great variety of numerical approaches that approximate solutions to the Navier-Stokes equations, which generally describe the movements of viscous fluid substances. While the objectives of these approaches are to capture related physical phenomena, the details of different methods lend them to particular classes of problems, and scalable solutions are important to a large range of scientific and engineering applications. In this paper, we investigate the practical scalability of two proxy applications that are made to recreate the essential performance characteristics of Lattice-Boltzmann Methods (LBM) and Smoothed Particle Hydrodyamics (SPH), using the former to simulate the formation of vortices resulting from sustained, laminar flow, and the latter to simulate violent free surface flows without a mesh. The differing scalability properties of these methods suggest different designs and programming methods in order to exploit extreme scale computing platforms. In particular, we investigate implementations that enable the use of task-based programming constructs, which have received attention in recent years as a means of enabling improved parallel scalability by relaxing the synchronization requirements of classical, bulk-synchronous execution that both LBM and SPH simulations exemplify. We find that suitable adaptations of the central data structures suggest that scalable LBM performance can be improved by tasking constructs in situations that are determined by an appropriate match between the input problem and the platform's performance characteristics. This suggests an adaptive scheme to identify and select the highest performing implementation at program initialization. The SPH implementation admits a substantial performance gain by partitioning the physical domain into a greater number of independent tasks than the number of participating processors, but its performance remains dependent on a powerful node architecture to support conventional SMP workloads, suggesting that further algorithmic improvements beyond the benefits of task programming are required to make it a strong candidate for exascale computing.

1. Introduction

CFD problems admit a great variety of both numerical and technical solutions, each suited to different problem classes, parallelization strategies and performance characteristics of the computing platform. LBM [3] and SPH [5] both present numerical solutions that admit highly parallel solutions, but the complexity of their programmatic implementation details create challenges with adapting them to emerging computer platforms, as design decisions embedded in a complete application program can result in unanticipated performance constraints on emerging platforms. In order to address this portability issue, we investigate the performance characteristics of two *proxy applications*, which are simultaneously developed to make it simple to experiment with programming alternatives, while retaining sufficient detail to simulate known physical effects in a CFD context.

Both computational methods share the trait that the limited operational intensity of their computational kernels make them memory bound, so efficient execution becomes highly dependent on the interplay between memory hierarchy levels and interconnect technology. Because of this, we primarily focus on applying different decomposition techniques to partition the physical domain for parallel execution.

Furthermore, both methods develop integrals over long sequences of finite time steps, creating a *bulk-synchronous*[8] execution pattern featuring periodic, global synchronizations. As global synchronization operations are inherently tied to interconnect latency, scalable implementations depend on effective use of latency hiding techniques[7]. Task-based programming presents a category of programming models that aim to relax synchronization requirements by exposing an abundance of parallel work units that can be scheduled according

* Corresponding author: e-mail. Jan.Christian.Meyer@ntnu.no tel. +47 73590345 March 21, 2019

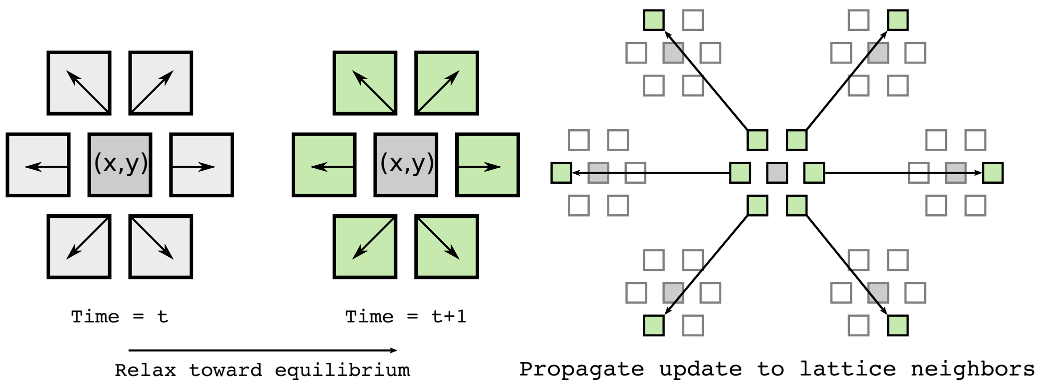

Fig. 1: (a) D2Q6 collision stage ; (b) D2Q6 streaming stage

to sequences that follow from their data dependencies, rather than requiring frequent, collective barriers at constant points in the program control flow.

In this paper, we quantify the impact of adapting the proxy applications to partition their problem domains both in classical, bulk-synchronous modes, and with methods that account for problem-dependent, non-uniform distributions of data that reduce the effects of tightly synchronized execution. Because task-based approaches incur an additional run-time penalty for scheduling the parallel work units, they can not be expected to provide great benefit for computational loads that are equally distributed between all parallel units. We simulate a set of input cases that have been selected in order to cover several different work distributions, in order to produce a neutral comparison.

The rest of this paper is structured as follows. Section 2. describes the origin of the proxy applications, and motivation for studying their performance. Section 3. describes the computation of the LBM proxy application, and its initial design. Section 4. describes the computation of the SPH proxy application, and its initial design. Section 5. describes the adaptations we apply to the two designs in order to decouple parallel tasks from the dimensions of the executing platform. Section 6. presents our experimental results, and discusses their implications for scalable implementation strategies. Section 7. concludes the study, and identifies directions for future research.

2. Background and Motivation

HPC applications inherently require interdisciplinary collaborations, with application experts providing the knowledge of the application domain, and HPC experts providing knowledge of large scale computing platforms. Performance representative proxy applications are research vessels to support such collaborations, and address the challenges associated with adapting full-scale application programs to novel programming models and architectures. By implementing particular solutions of a simplified problem instance, their intention is to remain short and simple to modify, yet capture the critical performance properties of a complete application, so that exploratory studies can be carried out prior to making expensive design decisions that determine the evolution of more elaborate solutions.

The proxy applications in this study have been developed in collaboration with the Dept. of Naval Architect and Marine Technology at Piri Reis University[6] as part of investigating effective fuel tank designs, but the methods themselves are in wide use, and applicable to a range of scientific and engineering problems. As a source of potential future Exascale applications, CFD modeling is a central research area: it is represented in the PRACE Unified European Applications Benchmark Suite (UEABS) [2] by both the Code_Saturne and NEMO programs, and fluid flow simulation was identified as important to the EoCoE European Centre of Excellence at the Exascale workshop organized by PRACE-5IP WP7.2 in June 2017.

3. The LBM Proxy Application

3.1. Computational Requirement

LBM computations model fluid motion as variations in density across a lattice of evenly spaced points, conventionally denoted in the form $DnQm$, where n represents the number of dimensions, and m represents the degree of connectivity between neighboring points. Density updates are calculated in *collision* and *streaming* stages of a time step. The collision stage estimates density updates at each point according to a function that tends towards equilibrium, while the streaming stage consists only of data movement, as updated density values propagate between neighboring points. The proxy application in this paper uses a $D2Q6$ lattice, corresponding to a flat, hexagonal mesh. Our form of the $D2Q6$ collision step is derived from the $D2Q7$ formula given by Chen *et al.*[3], by cancelling the free parameter α that controls the speed of sound in the fluid medium.

Fig. 1 illustrates the collision and streaming stages for one lattice point in this configuration: note that data access during the collision stage is restricted to values that are entirely local to each point, making the computational kernel trivially data-parallel. The streaming stage involves dependencies on neighboring points in a similar manner to a stencil application, except that neighbor elements are the destinations of data transfers, as opposed to sources used for read access. In order to treat each point in parallel during both stages, we use two buffers for the 6 densities of each point, corresponding to their present values, and updates for the next time step. It is both possible to merge the two stages into a single kernel that simultaneously computes updates and propagates them, and to remove the need for the additional buffer by carrying out the streaming stages as a sequence of pairwise exchanges. Our proxy application maintains separate stages and buffers for the sake of simplifying performance analysis, as it isolates computational requirements from data movement costs.

The operational intensity of the collision kernel is limited, as the 6-directional relaxation operation requires between 222 and 246 floating point operations per lattice point, and the lattice point consists of 16 values and an integer tag. For double-precision numbers, this results in the interval $[2.13, 2.37]\frac{FLOP}{byte}$, so a simple Roofline model[9] for any modern HPC system suggests that its behavior is memory bound. The streaming stage further exacerbates this, because it consists of copy operations only. There are several extensions to the application that can raise its operational intensity: more intricate lattice structures, simulating a mixture of multiple different fluids, or fluids in different phases all create additional arithmetic in the collision stage. In this study, we restrict our investigations to $D2Q6$ lattices with a single fluid for simplicity, and because one purpose of the application is to demonstrate the performance issues of LBM as clearly as possible.

3.2. Communication Requirement

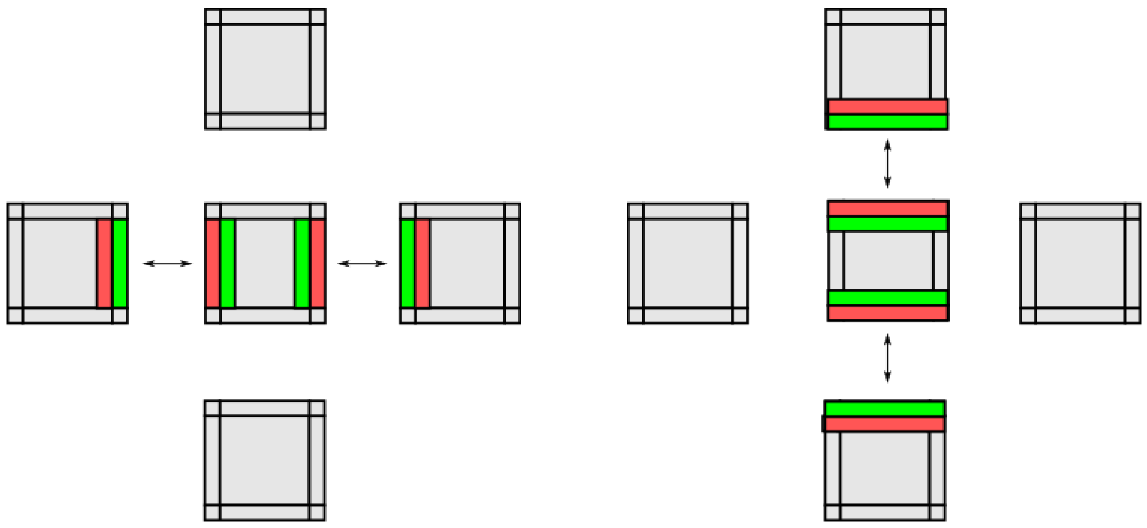

Fig. 2: (a) Horizontal exchange ; (b) Vertical exchange including halo points

The bulk-synchronous, data parallel execution pattern of the application lends itself to well-known domain decomposition techniques for rectangular arrays. The similarity to stencil applications suggests that 2D lattices can be partitioned into rectangular sections augmented with a halo region of 1 neighbor point, provided that a border exchange is carried out each iteration, between the collision and streaming stages. Our default implementation of the border exchange operation eliminates the need for separately exchanging corner points, by dividing the exchange into horizontal and vertical exchanges, as illustrated in Fig. 2.

3.3. Physical Model Problems

We apply the proxy application to three different physical phenomena, *Poiseuille* flow, turbulent flow in the wake of a *Cylinder*, and the formation of *Moffatt vortices*. The primary function of these problems is to validate that the proxy application correctly represents a working LBM solver, by verifying that it produces physical effects that are known to appear under particular conditions, illustrated in Fig. 3.

Poiseuille flow is a velocity profile that appears during laminar fluid flow through a cross-section of a pipe; the fluid is retarded by the edges of the pipe, but flows at a velocity related to the square of the distance from the pipe wall elsewhere, giving rise to the characteristically curved velocity profile shown in Fig. 3 (a).

Turbulent flow in the wake of a cylindrical obstruction appears when a cylinder laterally obstructs a laminar flow, as vortices begin to form in its wake. A 2D cross-section of such a system is shown in Fig. 3 (b). The velocity field in the figure reflects a time step prior to the onset of turbulence, so as to clearly display the geometry of the lattice, with the cylindrical obstruction left of the image center, laminar flow from left to right, and a region of lower velocities creating wake with lower velocities directly to its right.

Moffatt vortices[4] are circular motions that form inside a wedge when a laminar flow steadily passes its opening, when its angle is sharper than 146°. Two such vortices can be seen in the velocity profile shown in Fig. 3 (c).

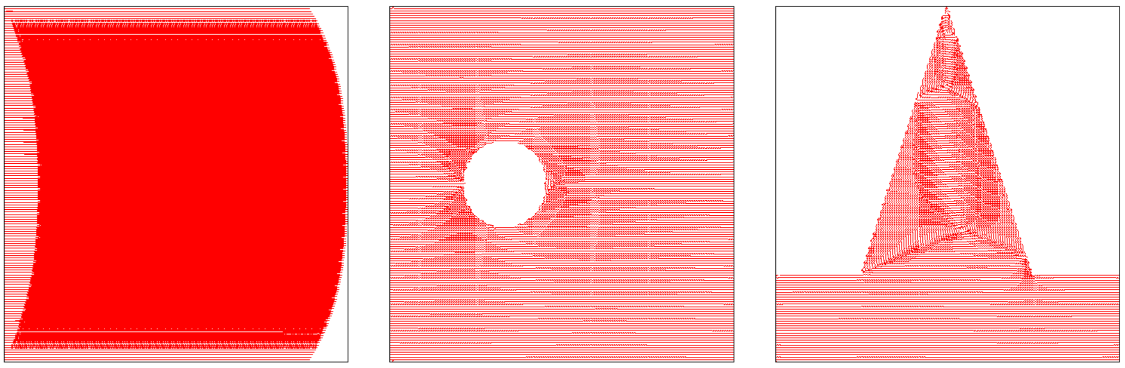

Fig. 3: (a) Poiseuille geometry; (b) Cylinder geometry; (c) Moffatt geometry

Beyond validating that the LBM proxy is sufficiently realistic to capture physical effects, these problem geometries are also selected in order to produce a range of program performance characteristics. Specifically, the presence of solids in the simulated domain creates a measure of computational load imbalance, as solid points do not require any computation. Thus, the Poiseuille flow case represents a perfectly balanced domain where every lattice point requires the solver to update it in the collision phase, the Cylinder case is similar, but with a limited size solid body in a localized area, and the Moffatt case represents a situation where the domain contains a large number of solid points distributed throughout.

4. The SPH Proxy Application

4.1. Computational Requirement

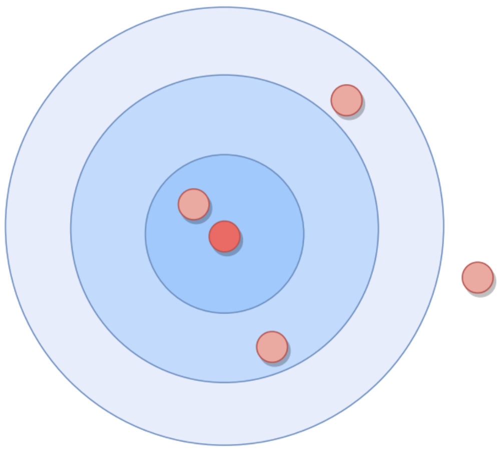

Fig. 4: Particles Near a Three-level Interaction Radius

SPH computations model fluid motion as a set of particles, described in terms of their spatial coordinates and physical properties. This method does not require the domain to be partitioned by a mesh, but instead relies on examining the spatial distance between all pairs of particles that lie within a specific interaction radius of each other, and computing the forces they interact with accordingly, to produce updated particle positions. The numerical kernel we use to approximate these forces is due to Ozbulut *et al.*[6], and discriminates between three regions that are parametric in the spatial resolution of the problem. A system with five particles and the three levels of the interaction radius are illustrated in Fig. 4.

The main bottleneck of the computation is the detection of particle pairs that are within the proximity of each other. As positions are unrestricted, and one fluid element may travel across the entire physical domain throughout execution, this becomes a search problem as common to N-body simulations, and a straightforward implementation makes $\frac{N(N-1)}{2}$ comparisons between all particle pairs. The implementation is organized using two main data structures: one is a total list of individual particles where the elements are their individual properties, and the other is a list of particle *pairs*, annotated with the data describing the strength of their interaction.

Global domain boundaries are handled as solid obstacles, so that for each particle approaching the edge of the domain, a *virtual* (or *ghost*) particle moving in the opposite direction is created for the duration of the time step, at a distance that is symmetric about the boundary. In this manner, particles are retained within the limits of the simulated system as they collide with their virtual counterparts at its edge, and no distinction needs to be made between virtual and actual particles for the purposes of the computational kernels.

4.2. Communication Requirement

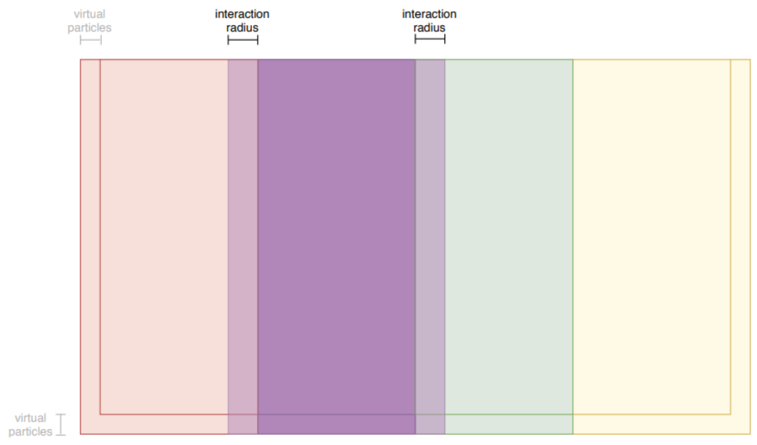

Fig. 5: Domain Decomposition and Halo Regions in the SPH Proxy Application

Distributed memory parallelism is implemented as a horizontal partitioning of the physical domain, as illustrated in Fig. 5. Ranges of x-coordinates are assigned to each MPI rank, which assumes responsibility for the particles that presently occupy its partition. This produces a fluctuating computational load which corresponds to the particle distribution in a given time step, and requires border exchanges to account for two effects: particles that pass from one subdomain to another must be *migrated* so that responsibility passes from one MPI process to the other, and particles that lie within one interaction radius of a subdomain boundary must be *mirrored* in temporary copies, so that their interactions with the particles of the neighboring rank can be correctly calculated.

This requires a border exchange sequence of several steps for each rank:

1. Count the number of particles to send left and right
2. Exchange particle counts with neighbors, and dynamically dimension temporary receive buffers
3. Serialize the particles to send
4. Exchange the contents of the send and receive buffers with neighbors
5. Deserialize received particles, and append them to the local particle list

Each time step requires two of these exchanges, once for mirrored particles, and once for particles that migrate between ranks. Migrating particles are deleted from their sender's particle list.

4.3. Physical Model Problem

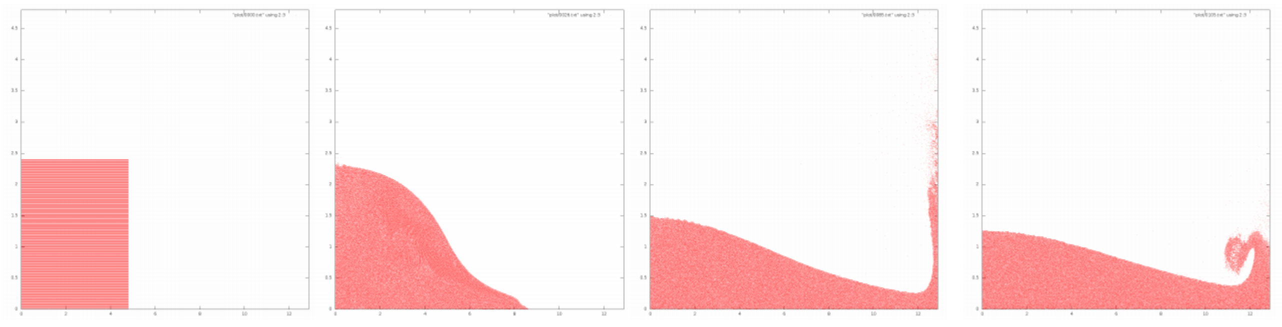

Fig. 6: Time Evolution of the Dam Break Problem, 21000 Time Steps

The SPH proxy application simulates *dam break*, which a widely used benchmark problem for CFD applications. Its initial configuration is a fluid volume that is aggregated on one side of a containing tank as if dammed up, and their free flow from this initial state imitates the effect of removing the dam. Fig. 6 shows the time evolution of the resulting, violent flow.

Beyond its role as a validation that the proxy application simulates known effects, this problem is chosen for the load imbalance created by its movement; the initial configuration exclusively concerns the left half of the domain, whereas after approximately 20.000 time steps, the particles are almost evenly distributed in the system. The choice to make the domain decomposion only along the horizontal axis reflects this behavior, as the movement of the fluid is predominantly in the horizontal direction.

5. Adaptations for Task Parallelism

Task parallelism presents a relaxation of the bulk-synchronous execution pattern in our proxy applications. The typical pattern of time integration combined with domain decomposition creates a rigid schedule where each participating process works on one partition of the problem, synchronizes, and repeats. The lock-step nature in this mode of execution creates a performance impediment, as synchronizations are bottlenecked by the slowest participant in the event of load imbalance. More flexible work schedules can be derived if the application exposes a greater number of parallel work units than there are physical resources. On the other hand, task management introduces an additional cost by compounding the workload with scheduling overheads, so effective use requires a balance between the granularity and number of exposed tasks.

In this section, we describe modifications to our proxy applications that aim to divide the parallel work into units that are independent of the number of parallel hardware resources, to admit more flexible run time decisions.

5.1. Tiling Domain Decomposition in LBM

The imbalance of the computational load in the LBM application stems from the input problem geometry, which remains fixed for the duration of execution. The work associated with a lattice point where fluid flows is constant, however, so the advantage that can be obtained is to distribute the domain so as to omit regions of solid points, where no computation is required. As the classical Cartesian domain decomposition depends more on the number of processes than on details of the input data, we decouple the partitioning from the distributed memory communication pattern by introducing a tile data structure that represents an arbitrary, rectangular subdomain, as illustrated in Fig. 7.

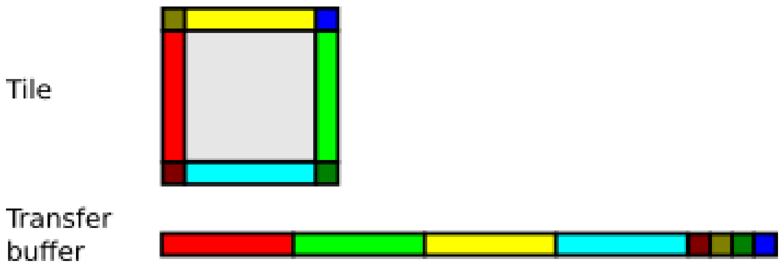

Fig. 7: LBM Tile Data Structure

By assigning the workload of each MPI process as a list of such independent tiles, and rewriting the collision and streaming kernels to work in terms of them, we gain the opportunity to balance the computation with adjustable granularity, at the cost of introducing an overhead for managing the list of tiles. It also complicates the border exchange operation, as the requirement to replicate halo points from neighboring MPI processes becomes a requriement to replicate halo points between all neighboring tiles, regardless of where they are hosted.

In order to facilitate similar handling of locally and remotely stored tiles, the tile structure is augmented with an additional layer of data replication, in the form of a transfer buffer which serializes the points from halo regions. The sequence of this serialization is shown by color coding in Fig. 7. Outbound and inbound buffers sized to the interior and exterior surfaces of a tile, respectively, and require the additional overhead of copying values from the interior and into the outbound buffer prior to border exchange, and from the inbound buffer into the halo region afterwards. While this additional step is not strictly necessary for data transfer between locally hosted tiles, it is highly beneficial for MPI communication, as send and receive operations benefit from accessing contiguous memory locations, while the tile itself is stored as a row-major array. The structure similarly regularizes local transfers, as they can be implemented using single calls to memory copy routines that are similarly optimized for contiguous access.

5.2. Cell List Domain Decomposition in SPH

The main impediment to thread-level scalability in the pair detecting method of the SPH proxy application is that it requires discovered pairs to be appended to a shared list, which is an atomic operation. Experiments

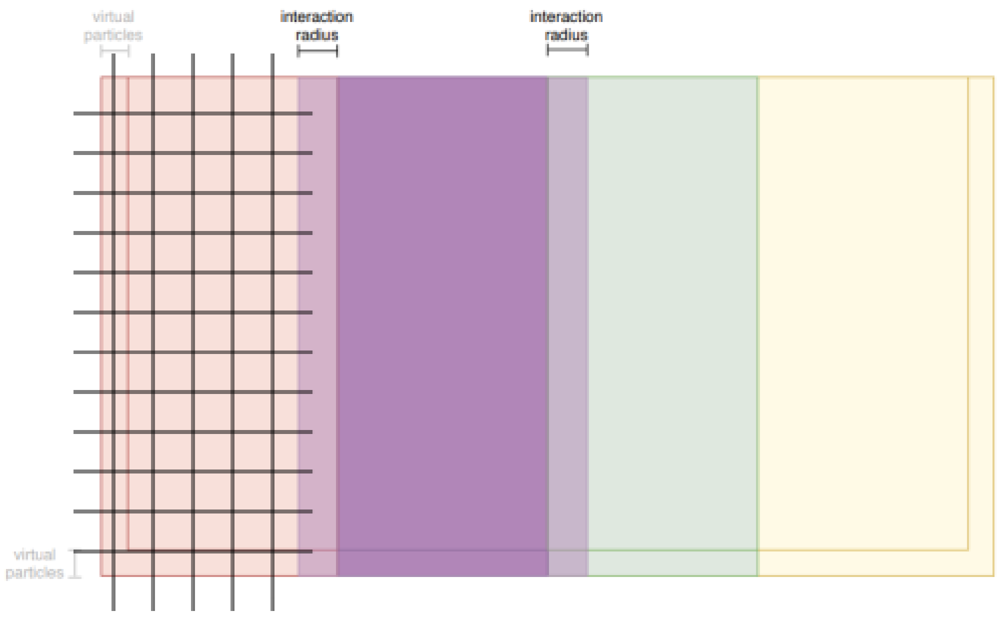

Fig. 8: SPH cell list, superimposed on leftmost rank

with increasing numbers of threads on single-node runs suggest that core utilization peaks near 4 threads, which is insufficient to make use of modern multi-core platforms with 2-digit core counts.

To mitigate this effect, we introduce a second level of domain partitioning internal to each MPI process, illustrated in Fig. 8. It consists of a grid of cells where particles inside each cell is stored in a separate list. This adds the overhead of requiring the pair detection stage to iterate over the collection of cells, but has two major benefits: the atomic list updates can be localized to each cell and merged into a final list of all pairs upon completion, and the particle count inside each cell is much lower than that of the entire subdomain, which substantially reduces the number of required comparisons, even though they remain asymptotically bounded as $O(n^2)$. Ideally, the cell size approaches a particle's interaction radius, which restricts the search for its admissible neighbors to the particles in its neighborhood of nine cells, but their dimensioning requires some care, as the subdomain size that partitions the global domain evenly among MPI processes is not necessarily a multiple of the interaction radius.

The performance improvements obtained by this optimization were found to permit full thread-level utilization of compute nodes. Furthermore, it performs categorically better than the atomic operation variant because it not only admits greater thread-level parallelism, but also reduces the magnitude of the workload algorithmically.

5.3. Task-based Programming Constructs

The adaptations discussed in this section were implemented for the purpose of admitting task-based parallelism by separating independently computable work units in the form of tiles and cells, and allow their parallel execution to be written in terms of a loop that iterates over tile/cell lists, dispatching each iteration as a work unit for a waiting thread pool. Preliminary experiments with this mechanism as implemented by the OpenMP `taskloop` construct in the Intel icc v18.0.1 toolchain indicated that while this expression produces correct results, the resulting performance was inferior to utilizing the data structures that are amenable to task programming, but using the `parallel for` worksharing construct to traverse the list.

For this reason, the performance results in the following sections are gathered using the latter approach, as the essential purpose of the experiments is to quantify the application performance implications of altering the program logic to expose a greater number of parallel work units than there are physical processing cores, and assigning them to threads at run time. The precise performance details of a given task library implementation are subject to change over time, and given a faster task dispatching mechanism, it is a minor modification to adapt the implementations we report on to make use of it.

6. Experimental Results

6.1. LBM Results

Scalability experiments were carried out on *Vilje*, which is an SGI Altix ICE X system with dual 8-core Intel Xeon E5-2670 processors, and 32GB of memory per node. It has an Infiniband FDR interconnect, which is configured in an enhanced hypercube topology, and hierarchically partitions into multiples of 18 nodes per rack unit, with four rack units per physical cabinet.

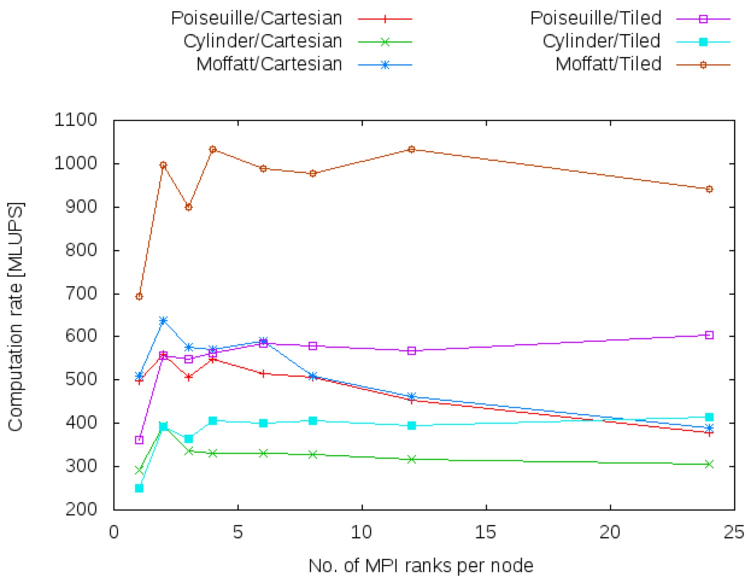

Fig. 9: LBM thread/rank balance results

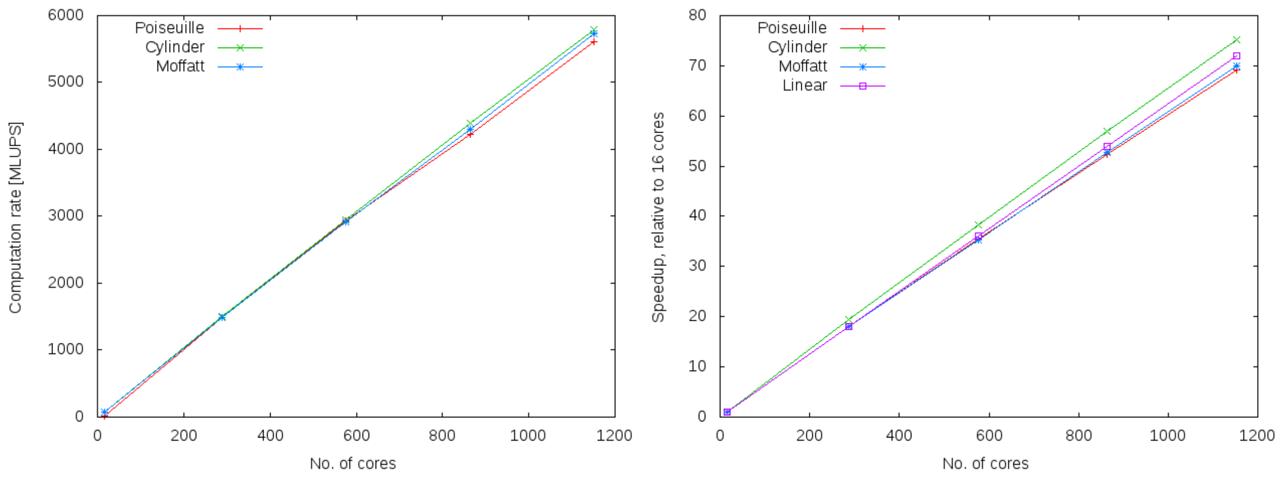

Fig. 10: Scaling results for LBM with Cartesian domain decomposition

In preparation for the scalability testing, we carried out a series of experiments on 8 nodes of a smaller, local computing cluster, featuring nodes with two 24-core Intel Xeon E5-2650 v4 processors each, Infiniband interconnect, and 128GB of memory per node. These experiments used a fixed input size of 16000×16000 lattice points on 192 cores, varying the balance of MPI processes and threads through configurations of $1, 2, 4, 6, 8, 12$ and 24 processes per node, and $\frac{24}{\#processes}$ threads per process. Results are shown in Fig. 9, with the total system computation rate measured in millions of lattice point updates per second (MLUPS).

It is interesting to note that the conventional Cartesian domain decomposition shows deteriorating performance with increasing MPI parallelism at the node level, whereas the tiling approach sustains its performance regardless of whether parallel cores are utilized for processes or threads. However, the primary result is that the majority of the performance improvement from MPI parallelism is achieved at 2 ranks per node, where each process is assigned to a physical socket. Due to this result, the scalability experiments were carried out in a 2 processes per node configuration.

As a basis for comparison, Fig. 10 shows speedup results from running LBM problem instances of 8000×8000 points for the Poiseuille and Moffatt geometries, and 8000×12000 points for the Cylinder geometry. Subfigure (a) shows total system computational rate in terms of millions of lattice point updates per second (MLUPS). Subfigure (b) shows speedup figures obtained in strong scaling mode, with 16 core (1 node) runs as the baseline of comparison, and tests of 1000 time steps using 288, 576, 864, and 1152 cores.

Performance scales nearly linearly, with slight superlinearity in the Cylinder case. The latter can be attributed to the cylinder problem consuming more memory than the other geometries, and therefore obtaining a suboptimal baseline measurement where its problem barely fits the computational node, and inflated performance improvements when this additional performance impediment is removed. We still note that the strong

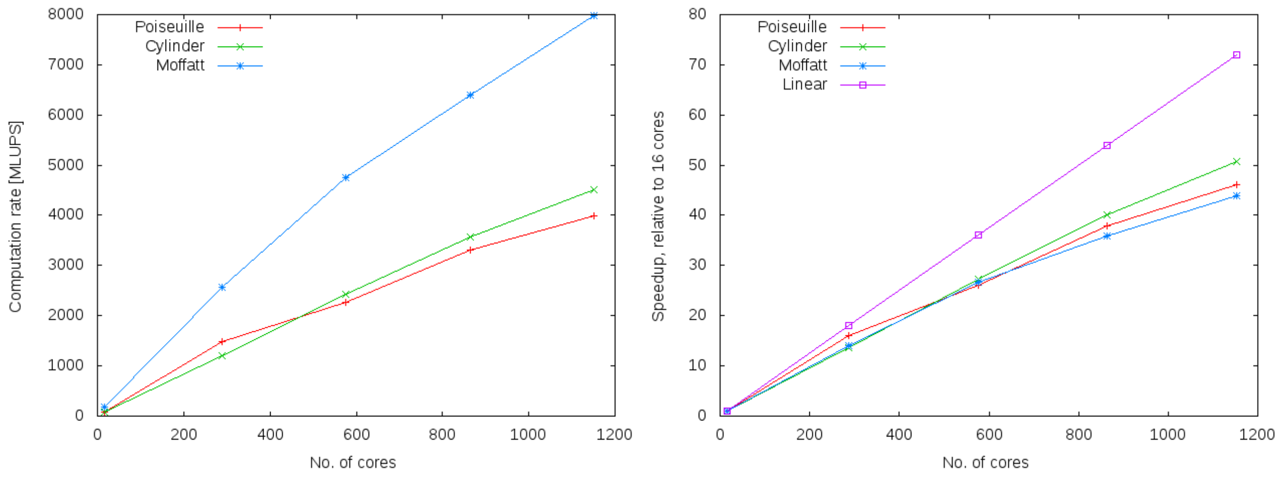

Fig. 11: Scaling results for LBM with the tiling domain decomposition

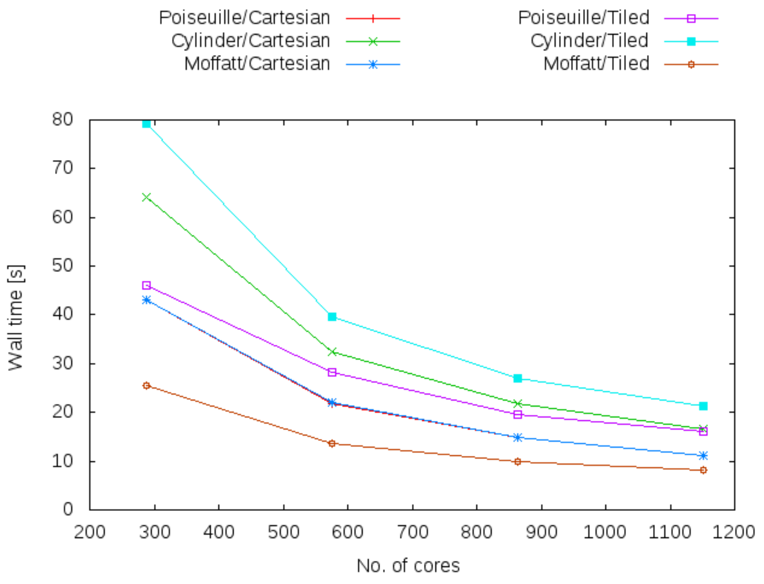

Fig. 12: Wall time comparison, 1000 iterations

scaling behavior of the LBM application is highly amenable to partitioning on a conventional 2D process grid determined by the number of processes.

Fig. 11 shows the results from a similar set of experiments, but using a tiled domain decomposition, with a uniform tile size of 250 × 250 points. The most notable effect is that the additional overhead of managing the additional memory and communication requirements of the tiling decomposition make it deviate from linear scaling much earlier than the Cartesian decomposition.

On the other hand, the Moffatt problem contains a large of solid points that can be excluded from the computation, arranged in contiguous areas. The Cartesian decomposition assigns all points to processes equally, so a fraction of both the baseline and timings at scale consists of traversing memory regions that contain no computation. Since the tiling decomposition can take adapt of the domain's shape by omitting tiles without fluid points altogether, a sufficient amount of these reaches a point where the performance penalty of managing tiles is outweighed by the reduced workload, and it is visible from the attainable computation rates that the Moffatt geometry is a case the where this is favorable, while the Poiseuille and Cylinder geometries are not.

Fig. 12 shows the absolute wall time requirements of all tested configurations in comparision to each other. This clearly illustrates the trade-off between the two decompositions. For the Poiseuille and Cylinder problems, the additional work of managing a software queue of independent subdomains and their interactions is the slower alternative, because their geometries are dominated by fluid flow, and there is little to gain by excluding solid areas.

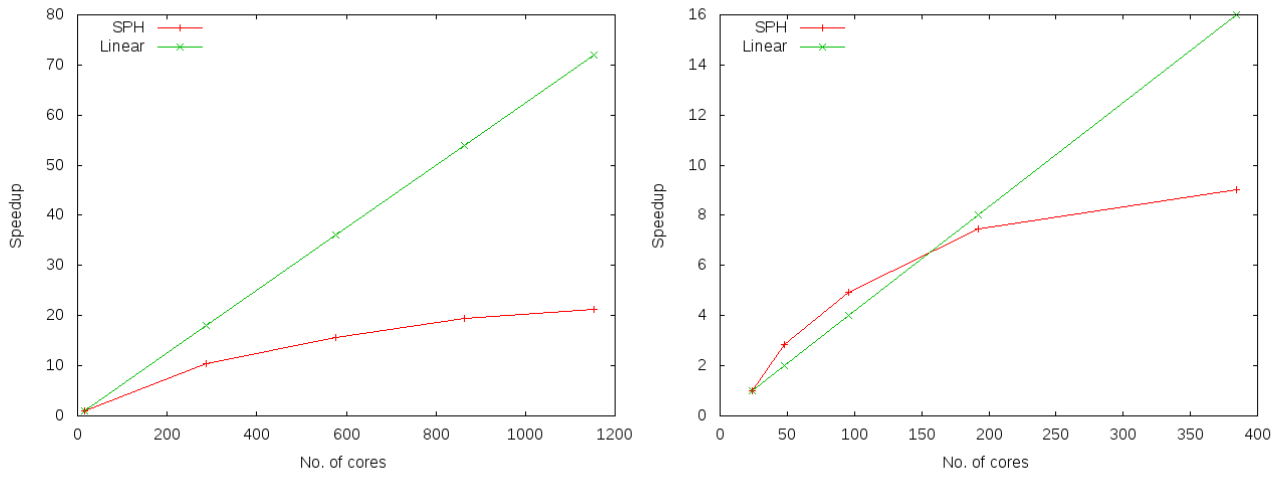

Fig. 13: (a) Vilje speedup, 16 core baseline; (b) Archer speedup, 24 core baseline

6.2. *SPH Results*

In addition to Vilje, SPH scalability was also tested on *Archer*, which is a Cray XC30 system containing two 12-core Intel Xeon E5 v2 processors per node, and 64 GB of memory per node. Its Infiniband interconnect is configured in a dragonfly topology. Simulations were run up until 20000 time steps, covering the stages of the dynamic behavior from the initial load imbalance, to an approximately even distribution of particles.

Fig. 13 shows parallel speedup figures obtained in the strong scaling mode on Vilje and Archer, with a linear curve plotted for comparison. Configurations of 16, 288, 576, 864, and 1152 cores are shown for the former, and 24, 48, 96, 144, and 192 cores are shown for the latter. The results were obtained with a simulation of 7381 fluid particles. The problem size is chosen to admit a 16 core baseline measurement, which restricts it to problem sizes that fit the amount of memory available on one node.

An interesting point is that in Fig. 13 (b), there is an initial benefit of distributing the problem on several nodes which allows each part to better utilize higher levels of the memory hierarchy, before speedup levels off and approaches its strong scaling limit. The more general observation is that upscaling the problem strongly depends on a large enough data set: the addition of more particles might let the speedup curve remain closer to the linear line for larger systems, but its characteristic shape remains indicative that the computation is bound by memory access and communication. Apart from the evidence that SPH benefits greatly from node architectures that support traditional SMP performance using complex compute cores, the strong scaling mode of the comparison also contributes to the scalability limitations visible in Fig. 13. This is due to a hard-coded problem instance that was embedded in the proxy application for verification purposes during development. Amdahl's law[1] predicts that this limits the size of applicable parallel platforms, but the effect is not inherent to SPH. Refinements to the spatial resolution would increase the number of simulated particles while decreasing the physical size of each, and produce more accurate simulations in exchange for additional parallel work. Preliminary work on a weak scaling comparison suggests that this shifts the scale at which performance reaches diminishing returns, but systematic experiments with its development are beyond the scope of this paper, because it would require a non-trivial extension and validation of the proxy application to make the resolution of its problem a parameter.

The wall time measurements presented in Fig. 14 compare the absolute performance of our two platforms. It is evident that Archer nodes feature superior memory technology, as comparable times to solution are attainable at $\frac{1}{6}$ of the system size.

7. Conclusions and Future Work

In this paper, we have examined the attainable performance of two CFD proxy applications featuring different numerical approaches, and examined the impact of implementing them using data structures suitable for task-based programming. For the LBM application, we have found that the overhead associated with maintaining and assigning task lists can be amortized when the fluid distribution in the simulated domain is sufficiently sparse, but the technique does not improve application performance at scale in cases where the domain consists mostly of fluid flow. For the SPH application, we found that augmenting the domain decomposition with partitioning into more fine-grained, independent cells yields an algorithmic improvement which substantially improves run time, but scalability quickly reaches diminishing returns depending on the number of particles that can be fit on the system.

From our results, we can recommend approaches to upscaling these computations:

- The LBM would benefit from an initial analysis of the domain, to adaptively select the optimal partitioning scheme according to the underlying architecture.

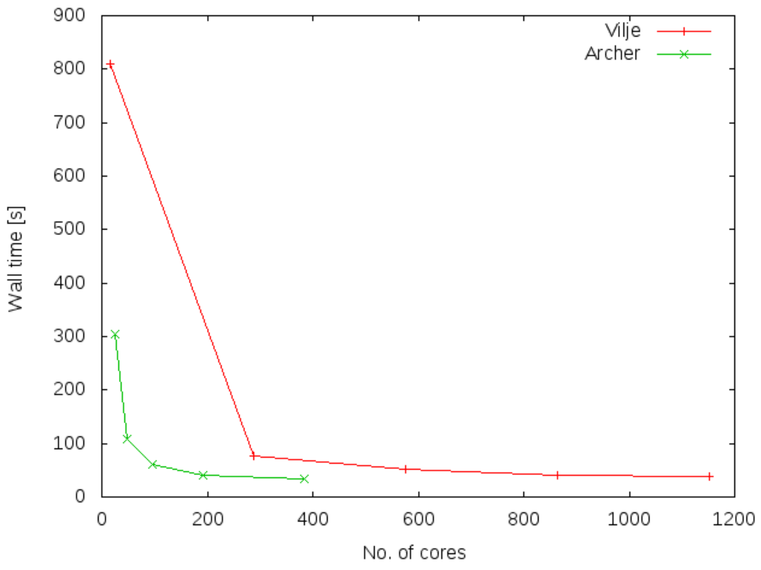

Fig. 14: SPH wall time comparison, 20000 time steps

- The SPH approach presently benefits from complex compute nodes with deep memory hierarchies, and will require further algorithmic improvements to become a viable candidate for exploiting future exascale platforms.

Interesting directions for future work are to find methods to accurately identify input data properties that govern the most favorable LBM decomposition, and experimenting with further reductions of work involved in the neighbor-finding stage of the SPH application, such as re-using the particle distribution from previous time steps in incremental updates.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 730913. We have also been granted access to ARCHER, the UK national supercomputer, to do this work via the INTERTWinE FET-HPC project.

References

1. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the 1967 Spring Joint Computer Conference (AFPIS'67)*, pages 483–485, 1967.
2. Mark Bull. Unified european applications benchmark suite. *PRACE-2IP Deliverable D7.4*, 2013.
3. S. Chen, G. D. Doolen, and K. G. Eggert. Lattice-Boltzmann fluid dynamics; a versatile tool for multi-phase and other complicated flows. *Los Alamos Science*, 22:98–109, 1994.
4. H. K. Moffatt. Viscous and resistive eddies near a sharp corner. *Journal of Fluid Mechanics*, 18(1):1–18, 1964.
5. J. J. Monaghan. Simulating free surface flows with sph. *Journal of Computational Physics*, 110(2):399–406.
6. M. Ozbulut, M. Yildiz, and O. Goren. A numerical investigation into the correction algorithms for SPH method in modeling violent free surface flows. *International Journal of Mechanical Sciences*, 79:56–65, 2014.
7. David A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, 2004.
8. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
9. Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.