Fast Fluid Dynamics on the Single-chip Cloud Computer

Marco Fais, Francesco Iorio
High-Performance Computing Group
Autodesk Research
Toronto, Canada
francesco.iorio@autodesk.com

***Abstract*—Fast simulation of incompressible fluid flows is necessary for simulation-based design optimization. Traditional Computational Fluid Dynamics techniques often don't exhibit the necessary performance when used to model large systems, especially when used as the energy function in order to achieve global optimization of the system under scrutiny. This paper maps an implementation of the Stable Fluids solver for Fast Fluid Dynamics to Intel's Single-ship Cloud Computer (SCC) platform to understand its data communication patterns on a distributed system and to verify the effects of the on-die communication network on the algorithm's scalability traits.**

Keywords: Fast Fluid Dynamics (FFD), Computational Fluid Dynamics (CFD), Conjugate Gradient, Distributed Systems, Intel Single-chip Cloud Computer .

I. INTRODUCTION

Simulation of incompressible fluids is often used in conjunction with the design and analysis phases of engineering and construction projects.

Fast Fluid Dynamics (FFD) is a technique originally introduced by Foster and Metaxas [2] for computer graphics, used to simulate incompressible fluid flows using a simple and stable approach.

In recent years FFD has been applied to numerous scenarios and its validity has been independently verified by multiple groups [1]. While simulations results diverge from experimental data, the accuracy of the prediction is often sufficient to provide guidance when fast simulation turnaround is required for design optimization and emergency planning scenarios.

FFD techniques use a regular grid spatial partitioning scheme. In order to simulate very large problems the amount of memory a single system can support is often not sufficient. Aggregating collections of systems is often used to simulate large domains and this technique corresponds to employing distributed-memory architecture. Even when memory on a single system is sufficient, the number of computing cores operating in single-image SMP architectures can exhaust the total available memory bandwidth. The overall algorithms scalability can thus suffer, regardless of the amount of available parallelism the algorithms actually exhibit.

The goal of our work is to design and implement a variant of the FFD method to evaluate its scalability traits on a system that employs an on-die network and not to produce the fastest possible implementation.

II. THE SCC ARCHITECTURE AND THE RCCE COMMUNICATION LIBRARY

Intel's Single Chip Cloud system is a novel microprocessor system design based on computing "tiles" organized in a 2D grid topology [5][6]. No hardware support for cache coherence is provided, but hardware support for message passing, message routing and synchronization primitives is available.

The main message communication library available on the system is RCCE [7], and offers basic synchronous message passing and synchronization facilities.

III. STABLE FLUIDS ALGORITHM

Stable Fluids was introduced by Stam [3][4] as a fast, stable technique to solve incompressible fluid field motion; the fluid domain is decomposed in a regular voxel grid. Each voxel contains the density and the velocity at the corresponding spatial location, thus defining a vector velocity field U and a density scalar field D.

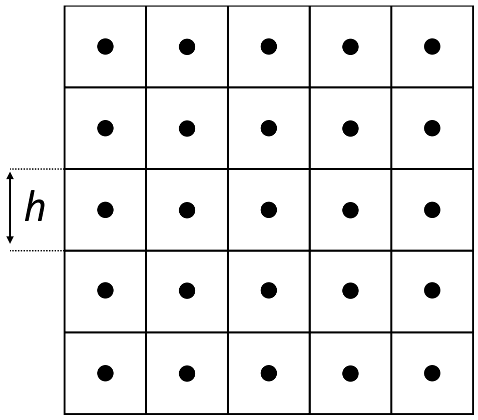

Figure 1. Regular voxel grid

To solve for a time step the simulator performs a series of sequential phases that operate on the velocity field and the density field:

Add velocity forces: compute contribution of external forces on the velocity field.

Diffuse velocity: compute the diffusion effect of the velocity field.

Advect velocity: compute the movement of velocity as affected by the velocity field itself.

Project velocity: compute the effects of mass conservation on the velocity field.

Add density sources: compute the contribution of external density sources on the density field.

Diffuse density: compute the effects of diffusion on the density field.

Advect density: compute the movement of the density field as affected by the velocity field.

IV. Simulating fluids on the SCC

Mapping the Stable Fluids solver on the SCC requires decomposing the fluid field into multiple tiled subdomains and assigning a core to each subdomain. A block partitioning scheme is the most natural solution due to the network topology the cores in the SCC architecture are organized into.

The domain decomposition operation is performed upon starting the simulator. In the current implementation the subdomains' locations and sizes do not change after initialization. The partitions are implicit: system software provides to each core its own network row and column index in the grid, and the total number of cores present in each row and column. Every core can therefore directly compute the size (number of rows and columns) and the origin of its own subdomain.

The effect of this partitioning scheme is that cores that are physical neighbors in the mesh topology operate on adjacent subdomains. This is important for optimizing overall communication latencies, as a significant amount of data dependencies refer to neighboring subdomains.

As a result, almost all communication happens between cores that are direct neighbors in the SCC mesh network. Fig. 1 shows how a part of the domain is mapped onto cores at indexes (0, 0), (0, 1), (1, 0), (1, 1) on the SCC.

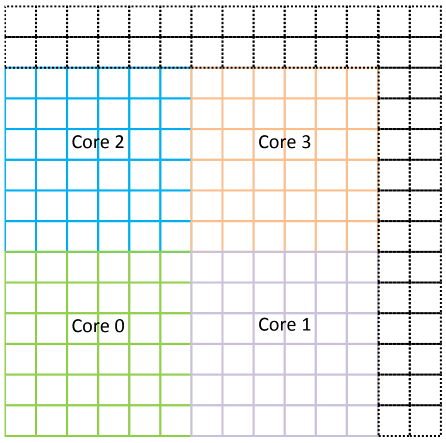

Figure 2. Domain decomposition

For efficiency reasons the fluid domain data is organized in memory in an Array of Structures layout. In this way it is possible to maximize spatial and temporal coherence in the different phases of the algorithm, and hopefully reduce performance degradation due to stalls in the memory hierarchy. Almost all data structures are evenly partitioned among the cores involved in the execution of the solver, the only exceptions are the structures containing details about the voxel type of the other data grids: in the current implementation, these data structures are replicated on each core, as they are involved in handling internal and external boundary conditions.

We implemented the solver as a C++ class library, using template constructs to facilitate changing of the basic data type of the simulation; varying the data type between different levels of precision obviously affects the overall simulation precision, performance and memory usage.

In the next subsections we analyze the individual phases that are performed in solving for a time step in the simulation.

A. *Add forces/sources phase*

Adding forces to the velocity field and the density field is a trivially parallel operation that doesn't require any data communication across subdomain boundaries; considering F as a vector field of external forces and S an external scalar field of density sources, this phase can be expressed as follows:

The formula for velocity is:

$$U_{i,j}^{n+1} = U_{i,j}^{n} + dt\, F_{i,j}$$

The formula for density is:

$$D_{i,j}^{n+1} = D_{i,j}^{n} + dt\, S_{i,j}$$

B. *Diffusion phase*

The diffusion phase computes the effect of diffusion on velocity and density in the voxel grid, which involves solving a system of linear equations. In this system *h* represents the size of a voxel as shown in Figure 1. ν represents the fluid viscosity constant, and κ represents the density diffusion constant.

The formula for velocity is:

$$U_{i,j}^{n+1} - \nu\, dt \frac{\left(U_{i-1,j}^{n+1} + U_{i+1,j}^{n+1} + U_{i,j-1}^{n+1} + U_{i,j+1}^{n+1} - 4\,U_{i,j}^{n+1}\right)}{h^2} = U_{i,j}^{n}$$

The formula for density is:

$$D_{i,j}^{n+1} - \kappa\, dt \frac{\left(D_{i-1,j}^{n+1} + D_{i+1,j}^{n+1} + D_{i,j-1}^{n+1} + D_{i,j+1}^{n+1} - 4\,D_{i,j}^{n+1}\right)}{h^2} = D_{i,j}^{n}$$

Our implementation uses the Conjugate Gradient method to solve the linear systems due to its ability to handle internal boundaries. Solving the linear systems results in a strictly data-parallel 5-point stencil data access pattern. Due to the predictable nature of the data access the communication requirements are all statically known. For this reason we can perform all the required data exchanges concurrently at the beginning of the phase then proceed to compute the voxels that do not require subdomain boundary values. At the end, we process the boundary voxels and as a result, completely overlap the data communication of all the cores.

C. *Advection phase*

The purpose of the Advection phase is to move both density and velocity along the velocity field.

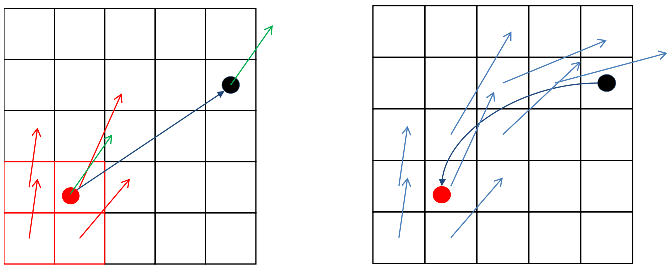

Figure 3. Advection phase backtracking and interpolation

In this phase *h* represents the size of a voxel as shown in Figure 1, Δt represents the time step, *Interp* represents a 2D linear interpolation function.

The formula for velocity is:

$$U_{i,j}^{n+1} = Interp\left(U_{i,j}^{n}, \begin{pmatrix} i \\ j \end{pmatrix} - \frac{\Delta t}{h} U_{i,j}^{n}\right)$$

The formula for density is:

$$D_{i,j}^{n+1} = Interp\left(D_{i,j}^{n}, \begin{pmatrix} i \\ j \end{pmatrix} - \frac{\Delta t}{h} U_{i,j}^{n}\right)$$

The data access pattern of the Advection phase is unpredictable at compile time, since it is data dependent. In fact, the access pattern depends on the evolution of the velocity field. This model of computation is known as *dynamic stencil* and its efficient parallelization is generally problematic.

Currently our solution involves using an implementation of a request-response protocol that allows one core to request another core for the voxel values of a specific grid. Each core batches its requests into a queue for every other core involved.

The queue data structure is implemented as a collection of fixed size arrays. The queue is initially composed of a single array of requests per destination core. When the space in each array is exhausted it is sent to the target core and a new array is allocated, becoming the current requests storage array. We thus use a data structure that can grow dynamically to accommodate computation requirements. To optimize memory usage, a garbage collection mechanism releases unused requests arrays as required. At the end of the Advection phase, unused arrays in each queue are deallocated in a single operation.

Take for example a queue composed of four arrays, where the three extra arrays have been allocated during a previous Advection phase. If the current Advection phase uses only two arrays, the last two are deallocated at the end of the phase. This strategy is based on the assumption that changes in the velocity field are not abrupt between consecutive executions, thus generating a similar amount of requests. Since we will likely require similar sized sets for the next Advection phase, we don't release all the arrays at the end of the phase.

To compute the Advection phase on a 2D domain, then for each voxel in its local subdomain, each core first computes the global grid indices of its four neighbor voxels (eight in a 3D environment), resulting from the backtracking operation. If all the required voxels are local, the final voxel value is computed. Otherwise a new request is added to the queue of the core which owns the subdomain containing each remote voxel, and the computation of the final voxel value is deferred.

In summary, since the request-response protocol introduces some communication overhead, we use a batching strategy to minimize overhead. Core specific requests are batched and sent at the end of the local computation or when the current request array is full. A communication thread running on each core monitors incoming requests from other cores, then creates messages containing the requested data and enqueues the messages for transmission back to the requesting cores.

On each core when all the required remote data has been successfully received, all the previously deferred voxels can finally be computed.

This approach has proven to be quite efficient due to the low communication latency on the SCC mesh network. However it is important to underline that performance is highly data dependent. For example, small velocities and small time steps imply a small number of voxels with remote dependencies, with the remote voxels likely being stored in the memory of physically neighboring cores on the SCC mesh network. This results in a limited amount of communication between direct physical neighbors, minimizing both the required bandwidth and message routing distance on the mesh, in turn minimizing latency.

In a different scenario, large velocities and/or large time steps introduce large amounts of voxels with remote dependencies, which may involve communicating across larger routing distances on the mesh. This implies additional hops in the communication network, more message collisions/conflicts and in general, higher communication latency.

The implementation of our request-response protocol on the SCC required functionality not available in the RCCE library, which only supports pure send-receive communications. Our protocol requires both asynchronous message passing and data-dependent message destinations. We then extend the RCCE library with additional functions which will be discussed in section V.

D. *Projection phase*

The Projection phase corrects the velocity field to ensure conservation of mass, and involves computing the current flow gradient field and solving the Poisson equation to force the flow in and out of each voxel to be equivalent.

The current flow gradient field is easily obtained using the current velocity field, and only requires statically known communication of voxel values along borders of the subdomains. The solver then proceeds to solve the following linear system, where *P* represents the pressure field in the Poisson equation:

$$P_{i+1,j}+P_{i-1,j}+P_{i,j+1}+P_{i,j-1}-4\mathrm{P}_{i,j}=\left(U^{x}_{i+1,j}-U^{x}_{i-1,j}+U^{y}_{i,j+1}-U^{y}_{i,j-1}\right)h$$

Solving the linear system is accomplished by re-using the Conjugate Gradient method already applied during the Diffusion phase. The data access pattern is the same and we can easily overlap all data communication by using asynchronous communication functions.

V. RCCE EXTENSION

Due to the data-dependent and unpredictable nature of the data access pattern in the Advection phase, the basic RCCE library provided by the SCC SDK is not suitable. The RCCE API does not contain functionality to efficiently listen to incoming messages which can arrive at any time from any core. It also lacks support for asynchronous communication, which is fundamental to implement our request-response protocol.

Some other communication libraries have been developed since the SCC architecture has been released, iRCCE [11] and RCKMPI [10] are the most popular. The former is an extension to the RCCE library while the latter is an implementation of the MPI standard for the SCC.

iRCCE is a promising library, as it adds non-blocking point-to-point communication capabilities to RCCE and introduces a new, smarter version of the “send/receive” functions. This alternative communication scheme is referred to as “pipelined”. It splits the Message Passing Buffer (MPB) into two chunks, allowing both the sender and the receiver to access the MPB at the same time, in parallel. While the new features introduced by the iRCCE extension are useful in the context of our work, they are still not sufficient for our purposes. In particular it is not possible to efficiently receive a message without knowing the sender in advance, and mixing of non-blocking communication requests with blocking collective operations is not supported.

In our computation we often need to compute the norm of a vector partitioned among all the cores’ address spaces. Without mixing point-to-point communication requests with collective operations, we would require a barrier every time we need to compute a norm. Moreover, the pushing mechanism used by iRCCE to allow the communication to progress leads to a more complicated and less portable application code. One of our purposes is to write the algorithm in a way that minimizes the effort required to port the code to different distributed memory architectures, a cluster, for example. For this reason we decided to isolate the architecture-dependent aspects of the communications in a separate thread that emulates a communication controller, for example a DMA engine, or a hardware thread in a Simultaneous Multithreading system.

Using a dedicated thread for communication management introduces a small amount of overhead due to the context switches between the computation thread and the communication thread. However this solution is more flexible, because the communication management thread waking pattern (and hence the context switch frequency) is configurable. An additional advantage is that the application code is cleaner, as the calls to the functions that allow communication progress is not interleaved with the algorithm code.

RCKMPI is one of several implementations of the MPI standard [8] developed for the SCC architecture, derived from the MPICH2 implementation [9]. Its main advantage is that many parallel applications programmers are familiar with MPI and a parallel application written with RCKMPI only needs to be recompiled with an MPI implementation to be ported to a variety of distributed systems. However, RCKMPI is affected by some of the issues already discussed: in particular the need for a receiver to statically know the rank of the sender and the size of the message.

For these reasons we implement our own extension of the basic RCCE library, reusing most pre-existing data structures to support asynchronous communication and a request-response protocol. Our extension uses an interface similar to the standard TCP/IP “select” function, and introduces a non-blocking operation to quickly identify incoming messages and operate on them.

Our “select” function takes an array of chars as input, which will be filled with the ranks of the cores that are requesting to initiate a communication. Upon completion, our function returns the number of valid entries in the array. Our “select” is based on custom variants of the low-level RCCE “send_general” and “receive_general” primitives.

Our new “send” adds a header to the message containing the type of the message and its size in bytes, so that the new “receive” does not require the size of the message as a parameter.

The type of the message is an additional one-byte field that can be used by the sender program to mark the content of the message, so that the receiver program can perform different tasks according to this information.

We allocate two new sets of communication flags in the MPB, that are used for signaling by all the new functions (“select”, size-agnostic “send” and “receive”). This way we can handle both point-to-point and collective communication requests without signaling conflicts. The new flags allocated on the MPB reduce the size of the largest data chunk transferable by 48 bytes (using flags of 1 byte), but we consider this trade-off acceptable.

VI. RESULTS

We tested our solver on domains of different sizes. For each experiment we incremented the size of the domain proportionally to the number of cores involved, which provided a good measure of the impact of communications on the overall performance.

For each domain size, the domain partitions were assigned to neighboring cores in the mesh network by using the logical layout provided by the RCCE library. This ensured neighboring logical domain partitions were assigned to physically adjacent cores.

While our experiment provided a test of both the processor cores and the on-chip mesh communication network, initially we only used the default frequencies for the processor cores and communication mesh.

The focus of the tests was not absolute performance but an analysis of the scalability traits.

TABLE 1. EXECUTION TIMES FOR ONE TIME STEP OF SIMULATION

Domain Size	Cores	Time (seconds)
1024 X 1024	1 X 1	116.76
2048 X 2048	2 X 2	142.63
4096 X 4096	4 X 4	153.00
6144 X 6144	6 X 6	153.45
8192 X 6144	8 X 6	153.85

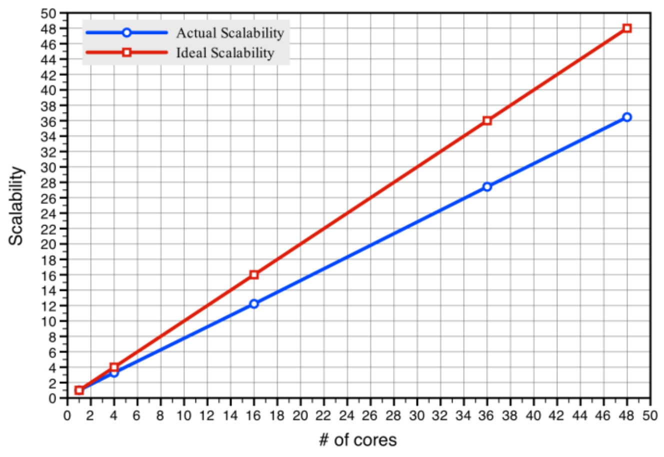

Figure 4. Scalability results

Table 1 reports the execution times for solving one simulation time step using single precision floating point as the basic data type. Fig. 1 represents the actual scalability of the current implementation of the solver on the SCC and compares it with the ideal scalability curve.

The reference single-core solver used for obtaining the baseline timing does not contain any form of communication. The multi-core distributed solver thus introduces a certain amount of overhead even in its minimal 2x2 cores implementation.

The results demonstrate that while communication indeed introduces overhead, the overall scalability traits of the algorithm are good. The overhead is constant beyond 4x4 cores. As a result the solver exhibits a constant execution time for larger domains, up to the maximum size tested. The memory used approached the upper limit of the SCC system used for our tests.

VII. CONCLUSION AND FUTURE WORK

The approach chosen in our implementation exhibited fairly good scalability, with the experimental results being quite promising. We plan to continue the work with the introduction of additional optimizations for performance, communication and synchronization.

This paper focused on simulations performed on 2D domains, but work is already underway on an extension of the solver to 3D domains. The performance optimization work will concentrate on the improvement of memory access, additional exploitation of asynchronous data transfer, and better exploitation of temporal coherence, especially in the Advection phase.

Variations of the cores and mesh frequencies will also be evaluated to understand their effects on power usage, and to find the optimal frequencies that allow the fastest algorithm performance while minimizing power usage. The chosen domain partitioning layout is expected to benefit this experiment by minimizing the average distance messages need to travel on the mesh network.

ACKNOWLEDGMENT

The authors would like to acknowledge Dr. Jos Stam for the precious collaboration and guidance on the algorithm details, and Alex Tessier for his insightful comments.

REFERENCES

[1] W. Zuo and Q. Chen, “Validation of fast fluid dynamics for room airflow ”, IBPSA Building Simulation 2007, Beijing, September 2007

[2] N. Foster and D. Metaxas, “Realistic animation of liquids”, Graphical Models and Image Processing, volume 58, number 5, 1996, pp.471-483

[3] J. Stam, “Stable fluids”, In SIGGRAPH 99 Conference Proceedings, Annual Conference Series, August 1999, pp.121-128

[4] J. Stam, “Real-time fluid dynamics for games”, Proceedings of the Game Developer Corner, March 2003

[5] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, T. Mattson, “A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS”, Proceedings of the International Solid-State Circuits Conference, Feb 2010

[6] T. G. Mattson, R. F. Van Der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, “The 48-core SCC Processor: the programmer's view”, Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, p.1-11, November 13-19, 2010

[7] T. Mattson, R. Van Der Wijngaart, “RCCE: a small library for many-more communication”, Intel Corporation, May 2010, Software 1.0-release

[8] Message Passing Interface Forum, “MPI: a message passing interface standard”, High-Performance Computing Center Stuttgart (HLRS), September 2009, Version 2.2

[9] “MPICH2”, Internet: http://www.mcs.anl.gov/research/projects/mpich2, [June 20, 2011]

[10] I. A. Comprés Urena, “RCKMPI user manual”, Internet: http://communities.intel.com/docs/DOC-6628, January 2011

[11] C. Clauss, S. Lankes, J. Galowicz, T. Bemmerl, “iRCCE: a non-blocking communication extension to the RCCE communication library for the Intel Single-chip Cloud Computer”, Internet: http://communities.intel.com/docs/DOC-6003, February 2011