IV International Conference on Computational Methods for Coupled Problems in Science and Engineering
COUPLED PROBLEMS 2011
M. Papadrakakis, E. Oñate and B. Schrefler (Eds)

COMPARISONS OF COUPLING STRATEGIES FOR MASSIVELY PARALLEL CONJUGATE HEAT TRANSFER WITH LARGE EDDY SIMULATION

S. Jaure*, F. Duchaine* L.Y.M.Gicquel*

*European Centre for Research and Advanced Training in Scientific Computation (CERFACS)
CERFACS, 42 Avenue Gaspard Coriolis, 31057 Toulouse Cedex 01, France
Tel : (33).(0)5.61.19.31.31 - Fax : (33).(0)5.61.19.30.00 e-mail: jaure@cerfacs.fr

Key words: Multiphysics, High Performance Computing, Coupling, Massively Parallel, Conjugate Heat Transfer

Abstract. The optimization of gas turbines is a complex multi-physic and multi-component problem that has long been based on engineer intuitions and expensive experiments or trial and error tests. Today, turbine experts commonly acknowledge that computer simulation is a very promising path for optimization, which can reduce costs and diminish the duration of the design process. Computations however remain a great challenge essentially because of the High Performance Computing (HPC) context, which is necessary for accurate estimates of real-life type of problems. Despite this difficulty, current high-fidelity computer simulations become accessible for specific components of gas turbines [5]. These stand-alone simulations and solutions now face a new challenge: to improve the quality of the results, new physics must be introduced with specific and distinct numerical models. For example, in the context of multi-component simulations, further improving the accuracy of turbine wall temperature is of limited interest if wall temperature boundary conditions are still set approximately. Dealing with multi-physics, recent studies have shown interesting results by taking into account reactive flow as well as radiative and conductive heat transfers to predict wall temperature of a helicopter combustion chamber [2, 1].

Based on the simulation of conjugate heat transfer within an industrial combustor, the current study aims at comparing different strategies of code coupling on HPC architectures. The flow solver is the Large Eddy Simulation (LES) code AVBP already ported on massively parallel architectures [5]. The conduction solver is based on the same data structure and thus has the same performances in term of parallelism. Coupling these two codes although possible requires exchanging and treating information based on two different computational grids and time evolutions. Such transfers have to be thought to maintain code scalability while maintaining numerical accuracy, thus raising communication and HPC issues: transferring data from a distributed interface to an other distributed

interface in a parallel way and on a very large number of processors is challenging and the solutions are not yet clear.

The strategies investigated in this work go from standard client/server couplers to fully distributed couplers. Altough the standard client/server couplers are easier to implement, they appear to have scalability issues which fully distributed methods do not share.

1 Introduction

Multiphysical simulation is a relatively new research field within the Computational Fluid Dynamics (CFD) community, and more generally in computational physics. There are two main strategies to perform multiphysics: on the first hand one can write an all in one solver handling all of the different physics together, on the other hand several individual solvers can be interfaced together forming a new aggregated solver. The first strategy requires developing new solvers capable of handling different physics, and thus different spacial and time scales, different types of equations,etc... Whereas the second solution has the advantage of focusing the development effort on the interface between the codes which is much easier and thus the preferred solution. The latter is called code coupling.

Generic couplers already exist (MpCCI, PALM, etc...) enabling users to interface different solvers in a simple way, producing an aggregate multiphysical solver. However the structure of these tools is generally based on one or a few sequential operators providing the services to interface the different solvers. As long as the solvers remain sequential or the applications do not require an important amount of parallelism, fairly good results can be obtained without a significant impact on the performance. However it is by design incompatible with massively parallel applications: in massively parallel applications scalability is synonym with data distribution whereas centralization is synonym with bottle neck. Thus for massively parallel cases these *sequential* "couplers " could become potential bottlenecks, specially if their is a lot of inter-code exchanges.

The goal of this study is building a scalable conjugate heat transfer simulation by coupling a CFD code with a thermal conduction simulation using two CERFACS codes: AVBP for the LES and AVTP for the thermal conduction. The scalability of AVBP has already been studied [9, 5]. Even though the demonstrator is built using those codes, all the concepts presented here remain general. Because LES is used for the fluid domain, the codes must be more tightly coupled than if RANS was used. Discussing the usage of RANS over LES for coupling applications is not the goal of this paper. Though as said in [3], RANS accuracy is more limited by closure models than by mesh resolution, meaning that RANS is not necessarily able to take advantage of high computing power, unlike LES.

In this multi-code coupling envirnomment there are two phases: the initialization phase and the run-time phase. During the initialization phase the different codes are to be

connected to each other, this implies performing geometric searches on the partitioned geometry. Then follows the run-time phase, which corresponds to the actual physical computation with the inter-code information exchanges.

In the first part of this work we will demonstrate how a sequential coupler can become a bottleneck for a massively parallel multiphysical application, either during the initialization or the run-time of the simulation. Then we will propose and assess a methodology designed for massively parallel applications.

2 Sequential coupling and massively parallel applications

The main service provided by a coupler is to enable codes to communicate at their interfaces. Here an "interface" stands for a portion of the geometrical domain which is shared between two solvers: for conjugate heat transfer it is just surfaces, but for other types of applications such as radiative coupling the "coupling interface" can be the entire domain. Because different codes operate on different physics, they may have different meshes, different processor counts and thus different domain partitioning. In order to communicate physical fields from a solver to an other, the data must be projected from the source code's interface to the destination code's interface. Though this operation is a simple interpolation process, in massively parallel codes (especially for CFD codes), the source and destination interfaces are distributed on different sets of the processors, making the task more complex.

How the geometry is distributed depends on the type of code, whether structured or unstructured, and on the partitioning algorithm (recursive inertial bisection, recursive coordinate bisection, etc...). To remain general we will assume that both coupled codes are unstructured and we do not make any assumption on the partitioning algorithm used. All communications are assumed to be performed using an Application Programming Interface (API) such as Message Passing Interface (MPI), which is the actual standard for inter process communications in computational physics. The conclusions could be generalized to other network type mediums, e.g., ordinary IP network. However assessing coupling strategies performed with slow mediums like temporary files is not considered here.

The most straight forward method to solve this two code conjugate heat transfer problem, demonstrated by Duchaine et al. [2] and then reused by Amaya, Poitou et al. [1], is to gather the entire distributed interface on to a service processor, perform the remapping or interpolation and then scatter this interface information on to the destination solver's interface. This follows a many-to-one and one-to-many communication scheme (Fig. 1). In this type of scheme the entire stress is focused on the service processor:

- receiving data from all the processors can lead to network collisions and thus degraded message passing performance.

- enough memory to gather the entire interface on the service processor is mandatory, and in some cases may not be possible.

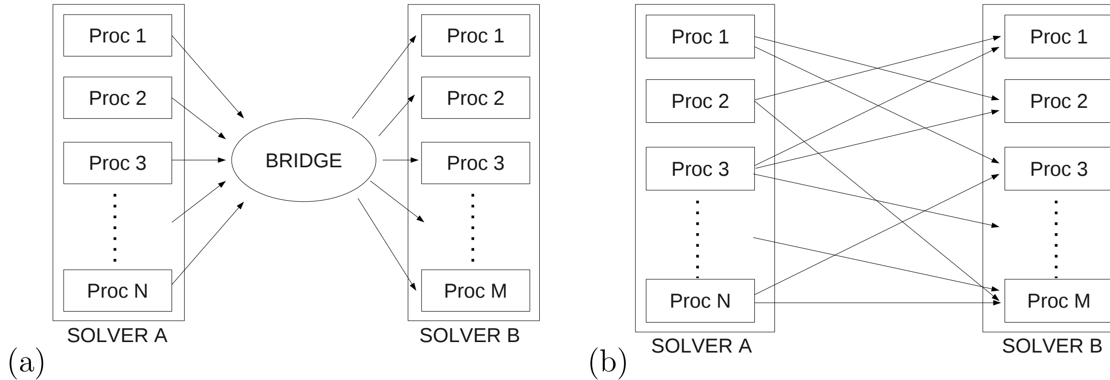


Figure 1: (a) Centralized Communication scheme (CCS), (b) Direct Communication scheme(DCS)

- the service processor has to perform all the interpolation process in a sequential way.

We can also see that in this communication algorithm there are two levels of communications introducing a high latency. This scheme will be called a *Centralized Communication Scheme (CCS)*.

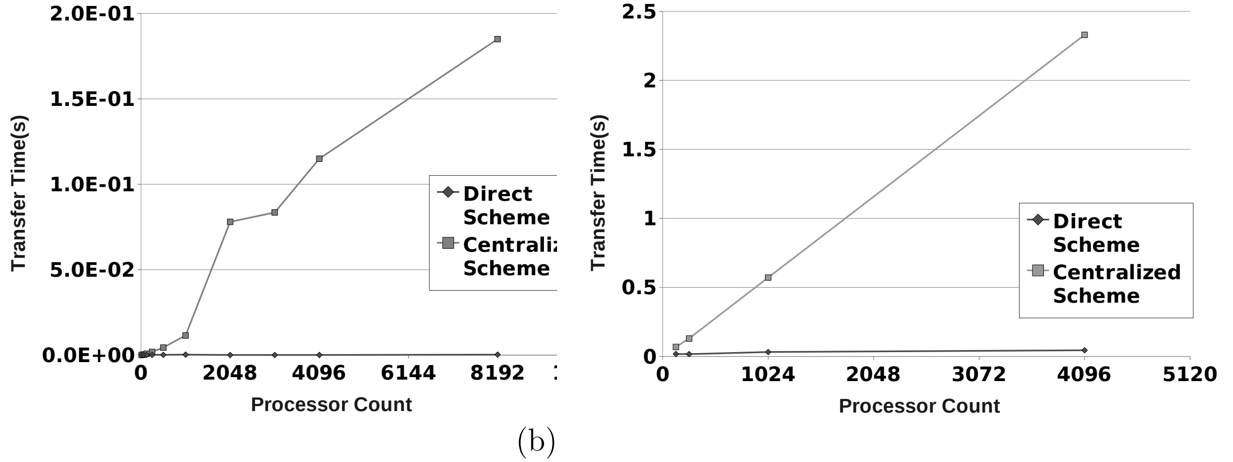
A direct communication scheme (Fig. 1) can be used to transfer the data, this however requires knowing *a priori* the adjacency information between both solver's processors. For each vertex of a given code, its dependencies, i.e. the source vertices used to interpolate the value of the given vertex, must be identified (their processor and interpolation weights).

Here we will assume this information is known, we will focus on the communication scheme, a methodology to obtain this information will be proposed in section 3. We will call this scheme a *Direct Communication Scheme (DCS)*. In DCS only one level of communication is needed, thus leading to lower latency than in CCS. Locality of data is also respected and the quantity of data processed by each worker remains small leading to low memory, communication and CPU stress. Most of all, in this scheme the stress is distributed over the entire set of processes handling the interfaces instead of centralizing it on one service processor.

2.1 Scalability evaluation of the two coupling strategies

These two schemes have been compared on a SGI-ALTIX ICE super computer. For the tests, a toy application implementing both communications schemes was timed and tests were performed for small to large processor partitions: from 64 to 8192 processors.

Figure 2 shows for two size of messages that the transfer time on the CCS scales with the number of processors used, whereas the DCS maintains almost constant timings regardless of the processor count. The differences in timings have to be evaluated in CPU cost, hence multiplied by the number of processors used. On massively parallel applications this can lead to huge differences, e.g., using the data of the "small messages" (Fig. 2) on 8192 processors performing 10000 coupling iterations the CCS consumes 4200



(a) (b)

Figure 2: Coupler transfer time : (a) small messages 1Kb, (b) large messages 100Kb

CPU hours whereas less than 10 hours for the DCS (Fig. 3).

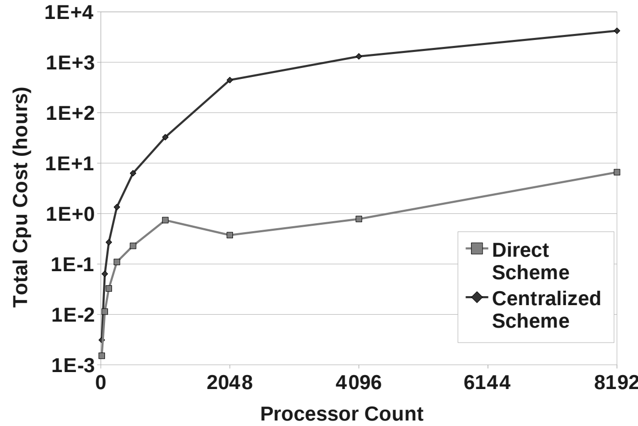


Figure 3: Coupler price in CPU Hours

The conclusion is obvious, in order to perform massively parallel coupling simulations, the communication scheme must stay direct. The result is not surprising since all scalable massively parallel codes use this communication scheme for their internal communications.

2.2 Memory requirements of the two coupling strategies

As said earlier, CCS can be limited by the amount of memory on the service processor. This is unfortunate since to take advantage of higher computing power, simulation sizes have to increase. DCS does not explicitly share that limitation. However implementing such a scheme requires being able to connect the interfaces together.

Connecting two interfaces together is fairly simple, either use a brute force nearest neighbor technique [2], or a more efficient tree based approach (Oct-Trees, balanced Kd Trees [4], etc...) if interface node count becomes high. Unfortunately these methods have a drawback: constructing trees requires memory, and such algorithms are not directly suitable for distributed geometry. The peak memory consumption criterion during the initialization phase is critical even for DCS. If the connection algorithm requires more memory locally than available on each node, no simulation can run at all. Typical examples of such constraints are some IBM Blugene machines where the available memory per core is as low as 256Mb.

3 Mesh Interface Connection

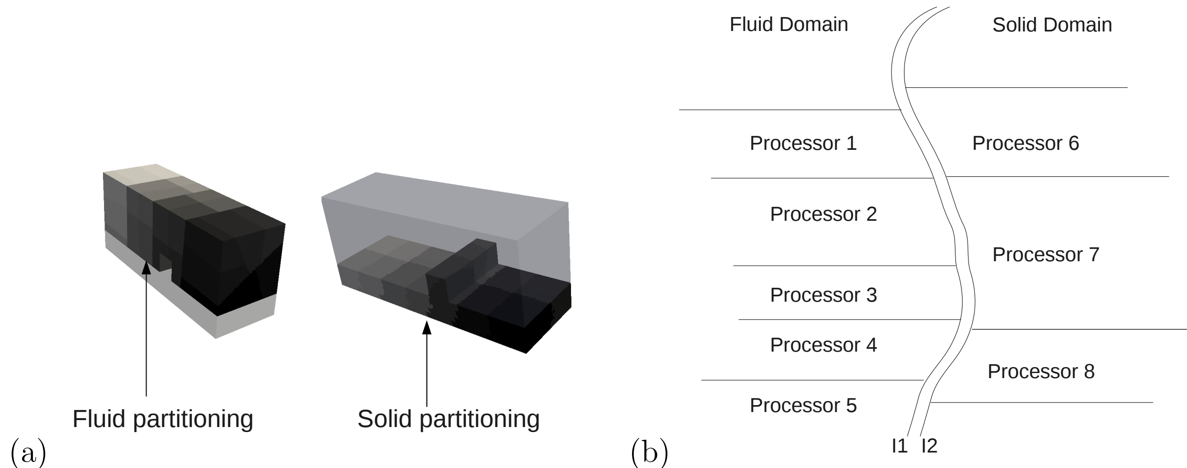


Figure 4: (a) An example of a partitioned geometry (each color group corresponds to a partition), (b) Interface partitioning

As we have seen, coupling massively parallel applications has to remain a distributed process to ensure scalability: not only during the run-time part, but also the initialization part. On one hand, distributing the work load during the run-time minimizes transfer time from a solver to an other. On the other hand, distributing the workload in the initialization improves the capacity of the coupled application to handle large simulations. Thus a fully distributed methodology to perform interface connection is proposed in the following by addressing the processors adjacency first and the interpolation coefficients in a second step.

Figure 4 illustrates the typical envirommemt to be dealt with in massively parallel CFD applications where the meshes are partitioned into sub-domains each processed by a different processor. This partitioning also applies to the coupling interfaces (I1,I2), as represented on Fig. 4. As the partitioning algorithm is usually not aware of the coupling process, the different distributions have no reason to match, leading to complex associ-

ations between interface processors of both solvers. As the communications are bidirectional, the coupling interface connection process is separated into two phases where each solver is either server or client.

3.1 Computing coupling interface's processors adjacency

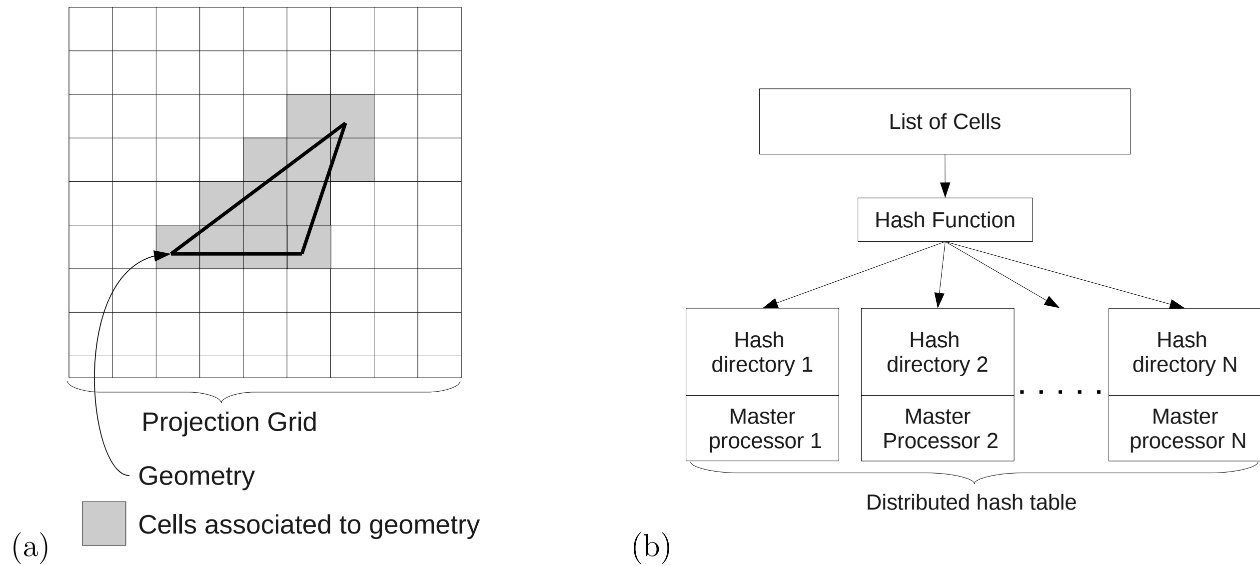


Figure 5: (a) Geometry projection to cell list,(b) Hashing of the list of cells

In order to locate the processor having parts of the interface involved in the coupling scheme, the server solver has to map it's entire geometry into a distributed database. To do so each processor of the server projects it's geometry on to a coarse uniform grid, associating 3D scalar vertices (x, y, z) to cell coordinates (i, j, k) (Fig. 5), each processor ending up with a list of cells to be mapped into a *Distributed Hash Table* (DHT). A DHT is a hash table ([7], [6]) for which the hash directories have been distributed over a network (Fig. 5). In this implementation each hash directory is assigned to a unique processor within the solver, the processors managing hash directories are called *master processors*. The *number of master processors* can be chosen between one up to the total number of processors available to the solver. Choosing a unique master gives a centralized scheme, and, if all processors are masters then the system becomes a peer-to-peer scheme. This entire process is a distributed version of a technique called spatial hashing [10] where the distributed hash table stores the associations between cells and processors. Knowing all this information, one can compute the list of processors associated to each cell (i, j, k).

Then each processor of the client solver interrogates the distributed hash table stored on the server solver (using the same projection and hashing mechanism) to obtain the list of potential processors which share a portion of the coupled interface.

3.2 Computing interpolation coefficients

A first identification of the communication paths being available, each processor communicates with it's potential neighbors the portion of interface they could share. Whenever possible, the neighbor processor then computes interpolation coefficients (based on an efficient Kd-Tree nearest neighbor search algorithm), and affects to those coefficients a *quality of interpolation* scalar (based on the distance between the elements to interpolate). This information is then received by the interrogating processor, which selects for each interpolated element the source providing the best interpolation quality. This information is then reorganized to produce an interpolation sparse matrix which will be used whenever a solver receives data. To conclude, each processor communicates to it's neighbors the list of vertices it depends on.

3.3 Scalability and memory tests of the proposed algorithm

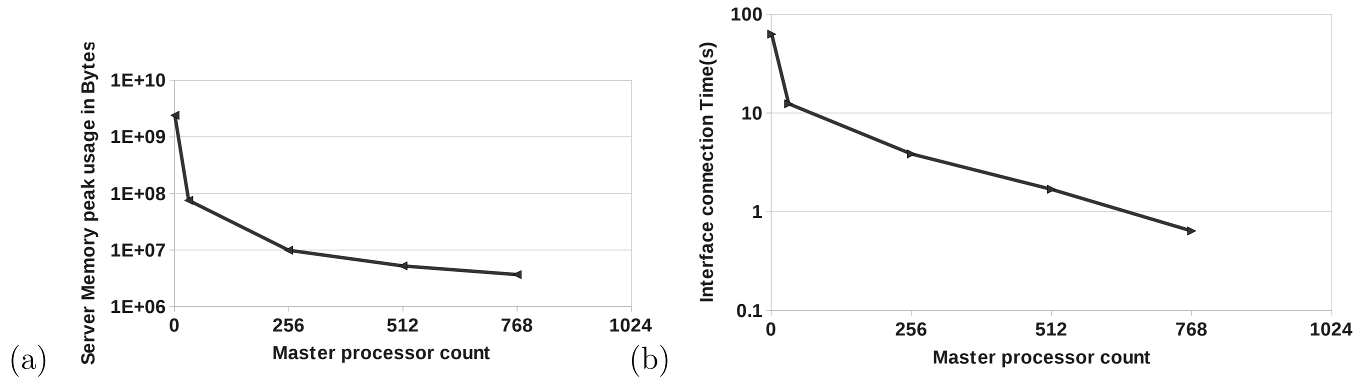


Figure 6: (a) Peak memory usage on server solver, (b) Interface connection time

As explained before, scalability must be analyzed from a general point of view, that is not only considering *CPU* stress, but also *memory* stress. The results discussed below are obtained by instrumenting the code, that is adding timers at strategic points and instrumenting the dynamic allocator.

In this first illustration, a key functionality is tested: the distributed hash table mechanism. The test is carried out in the following way:

- Two pseudo solvers are started, one will act as a server the other one as a client. The solvers do not have an equal processor count.
- A global set of nodes is distributed throughout each solver's processors.
- The server solver maps it's nodes using the distributed hash table.
- Each processor of the client solver asks the server through the distributed hash table the location of it's local nodes (several locations are possible).

After the first validation phase with small amounts of nodes, the real size tests are carried out on partitions of a SGI ALTIX ICE super computer. The results for 1024 cores partitioned into a server solver with 768 computing cores and a client with 256 computing cores exchanging the location of 100 million nodes are shown on Fig. 6. Local peak memory consumption used by the method for different count of master processors is specifically illustrated. The main goal has been achieved: by increasing the number of master processors, the memory load can be distributed over the solver's processors. An added benefit seen on Fig. 6 is a global application speed that is also increasing with the number of master processors. This is explained by the reduction of the quantity of data that each master processor has to process.

3.3.1 Full process tests: application on an industrial combustor

The entire interface processor adjacency computation and interpolation process has been tested on an industrial combustor configuration. The combustion chamber walls were coupled to a LES reacting flow in the burner. The test have been carried out on a 9.2 million cell fluid mesh involving 170 thousand coupled nodes and a 6.7 million cell solid mesh involving 226 thousand coupled nodes. To ensure full execution of the conjugate heat transfer problems, tests were executed in pure *peer-to-peer* mode, i.e., all processors are master processors. The test results have been summarized in table 1.

Processors		Memory peak		Connection wall clock time
AVBP	AVTP	AVBP	AVTP	
128	32	12Mb	26Mb	15.5s
256	64	9Mb	17Mb	11s
512	128	7Mb	15Mb	9s

Table 1: Full connection process test results

The primary objective which is maintaining a reasonable memory consumption has been clearly obtained. Note also that the connection timings remain relatively low and do decrease with processor count. These values do not decrease linearly with processor count: due to partitioning the processors handling the boundaries do not scale linearly with processor count. As for the transfer timings, when artificially synchronized (this would be the case for a perfect load balancing), the DCS maintained very low communication latencies: total transfer time (communication+interpolation) of the order of 1.10-4s. Since in this case, the iteration time for each code is of the order of 1s, the transfer timings can be neglected. Without artificial synchronization, because both codes execute at different speeds, they have to wait for each other during the inter-code transfers. These synchronization times depend on the load balancing. Obtaining a perfect load balancing is a difficult task, because it depends on the mesh, the solver type and the number of solver iterations between two inter-code exchanges. In the these tests, the time spent during

these inter-code synchronizations went up to 5s. The solutions to this problem are not clear yet.

4 Conclusion

This paper compares two different code coupling strategies for multi-physical massively parallel applications in CFD. Massively parallel codes in CFD obtain good scalability by distributing their geometry over large sets of processors and because each code has it's own geometry distribution, connecting coupling interfaces and transferring data from a code to an other is a fairly complex operation. Connecting interfaces and transferring data can be done by either using a centralized scheme or a fully distributed scheme. As discussed in this work, the centralized scheme has poor scalability and is incompatible with large simulations due to it's lack of memory distribution. To circumvent this major drawback a fully distributed methodology, in terms of interface connection and data transfer, has been proposed. As demonstrated here using this methodology, very large simulations can be tightly coupled on massively parallel machines without severely degrading the performances of the codes. Other fully distributed mesh connection methods must also be stated here, notably the methodology developed in the PUNDIT program [8].

Note that the load balancing issues still have to be investigated. In practice, obtaining a good load balancing between the different codes is difficult and thus a great deal of CPU power is wasted in inter-code waits which need to be reduced. Finally this work naturally leads to future large conjugate heat transfer simulations where more physical topics will be investigated, notably coupling frequency and multiphysical solution convergence.

REFERENCES

[1] J. Amaya, E. Collado, B. Cuenot, and T. Poinsot. Coupling LES, radiation and structure in gasturbine simulations. *Proceedings of the CTR Summer Program*, 2010.

[2] F. Duchaine, A. Corpron, V. Moureau, F. Nicoud, and T. Poinsot. Development and assessment of a coupled strategy for conjugate heat transfer with large eddy simulation. application to a cooled turbine blade. *International Journal of Heat and Fluid Flow*, pages 1129–1141, 2009.

[3] U.S. Departement Of Energy. The opportunities and challenges of exascale computing. pages 12–13, 2010.

[4] J. E. Goodman, J. O'Rourke, and P. Indyk. *Handbook of Discrete and Computational Geometry (2nd ed.)*, chapter Chapter 39 : Nearest neighbors in high-dimensional spaces. Chapman & Hall/CRC, 2004.

[5] N. Gourdain, L. Gicquel, G. Staffelbach, O. Vermorel, F. Duchaine, J-F Boussuge, and T. Poinsot. High performance parallel computing of flows in complex geometries: II. applications. *Computational Science and Discovery*, 2, 2009.

[6] R. Jenkins. *Dr Dobbs Journal*, 1997.

[7] Donald Knuth. *The Art of Computer Programming, volume 3, Sorting and Searching*, page 506542. 1973.

[8] J. Sitaraman, M. Floros, A. M. Wissink, and M. Potsdam. Parallel unsteady overset mesh methodology for a multi-solver paraidgm with adaptive cartesian grids. *26th AIAA Applied Aerodynamics Conference*, 2008.

[9] G. Staffelbach, J.M. Senoner, L.Y.M. Gicquel, and T. Poinsot. Large eddy simulation of combustion on massively parallel machines. *In 8th International meeting High performance computing for computational science*, pages 444–464, 2008.

[10] M. Teschnerand and B. Heidelberger. Optimized spatial hashing for collision detection of deformable objects. *Proc. VMV, Munich, Germany*, 2003.