On the parallelization of a finite volume program with OOPar

Philippe R. B. Devloo[1], Tiago L. D. Forti[2], and Edimar C. Rylo[3]

[1] FEC UNICAMP, Campinas - SP, Brazil,
phil@fec.unicamp.br
[2] fortiago@fec.unicamp.br
[3] ecrylo@fec.unicamp.br

Abstract. This paper describes the parallelization of a finite volume code using the object oriented programming environment OOPar. The original finite volume code is a complex CFD code which includes a large number of time solvers and iterative solvers. Rather than parallizing each solver separately, a strategy was adopted where only core operations are parallelized on a sub-structured mesh. This approach has the advantage that in a single step all solvers are parallelized, thus minimizing the impact on the existing code structure. OOPar is an object oriented environment which introduces a new paradigm in parallel computing, dividing the execution of the program in distributed tasks which act on distributed objects. OOPar allowed to define the execution of flux operators as a set of tasks which act on distributed data. Having the program structured using tasks and data allowed to gradually debug the parallel code: first all tasks were executed on the current processor, allowing to debug the operation of the tasks without transmitting the data from one processor to another; second the tasks were distributed over the pool of available processors, allowing to verify the consistency of transmission of data. Both serial code and parallel implementation coexist. This allowed to verify the results of the parallel code with the results of the serial code at each step of the debugging process. At this initial step of parallelization of the finite volume code, validation was emphasized. The parallel code developed produces identical results as the serial code. Any deviation is detected instantly.

1 Introduction

This paper describes the parallelization of a finite volume [8, 5, 3, 9, 2] code using OOPar [1, 7]. It is the first case of application of OOPar in a complex code. The original code is a complex tridimensional CFD code which includes a large numbers of time solvers and iterative solvers. Some functionalities of this code can be enumerate:

– Simulations of compressible Navier-Stokes equations with turbulence models: $k\epsilon$, SST and $Spalart - Allmaras$ models
– Transient and steady-state solutions

- Explicit (Runge-Kutta and Euler's) time solvers
- Implicit residuals solved with Dual Time Step and Newton's methods
- Consistent tangent matrix to Newton's method, computed in some cases using automatic differentiation
- Block diagonal preconditioner
- Multigrid preconditioner
- Structured and unstructured meshes with hexahedron, pyramid, tetrahedron and prism elements
- Adaptive mesh refinement

The goal of this work is to parallelize that code. Rather than parallelizing each solver separately, a strategy was adopted where only the core operations are parallelized on a sub-structured mesh.

2 Finite volume computation

The algorithm of finite volume consists of a loop over time steps. For each time step a time integrator is called to perform it. These time integrators consist basically of flux computations and some volume operators to obtain the next solution. For example a very simple explicit Euler's time integrator performs the following computation

$$W^{n+1} = W^n + \Delta t \, F(W^n)$$

where W^i is the state solution at the time step i, Δt is the time step and F is the flux vector which includes convective, dissipative, viscous and turbulent fluxes.

The finite volume computation is separated in two types of operations: volume operations and fluxes.

Operations that operate directly on cells are called volume operations. Examples of this class of operation are the multiplication of cell flux value by time step, dot product of two state vectors, multiplication of two state vectors term by term, the summation of two data etc. Fluxes are evaluated between two neighbouring cells.

The code is implemented in C++ using the object oriented philosophy. The interface of these two types of operations (flux and volume operations) is defined by two abstract classes: a flux base class and and a volume operator base class. All fluxes and volume operators in the code are derived objects of these two base classes.

The main entry point of the flux base class is the method ContributeFlux. This base class also define the interface method ComputeFlux which is implemented by the derived fluxes.

Any flux computation is performed calling the method ContributeFlux. This method collects data from cells, aligns them and call the method ComputeFlux. The method ComputeFlux is implemented on the derived object and it performs the specific flux computation.

The main entry point of the volume operator base class is the method ExecuteOperation. The interface method is OperateData which is implemented by derived volume operators.

Any volume operation is performed calling the method ExecuteOperation which calls OperateData from the derived volume operator.

These two base objects and theirs derived objects represent the core of that CFD code. Figure 1 represents these objects.

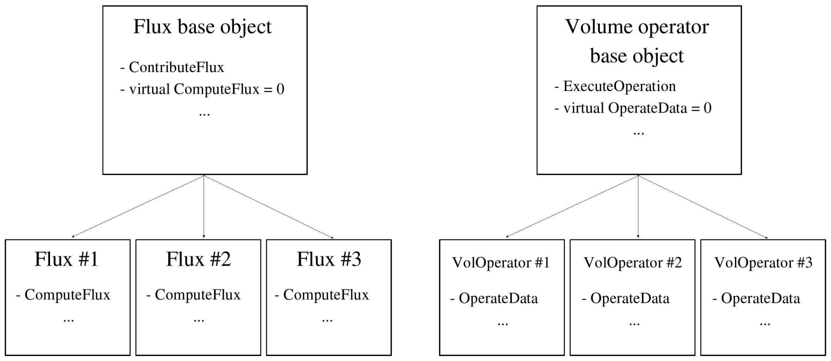

Fig. 1. Derived flux and volume operator objects

The parallelization of flux and volume operator base classes provides all derived fluxes and volume operators to be parallel. The methods ContributeFlux and ExecuteOperation were modified in the parallelization. No modification were made on the interface methods ComputeFlux and OperateData of the derived classes.

This approach has the advantage that in a single step all solvers are parallelized, thus minimizing the impact on the existing code structure and allowing the serial and parallel code to coexist.

3 OOPar

OOPar is an object oriented environment which introduces a new paradigm in parallel computing, dividing the execution of the program in distributed tasks which act on distributed objects. OOPar allowed to define the execution of flux operators as a set of tasks which act on distributed data.

3.1 Distributed objects

A distributed object is an object which reside in several processors but corresponds to one object on the main process. A distributed object is submitted to oopar which returns an objectID. This ID, global and unique, allows the user to identify the distributed object on the pool of processors. The ID is used to create tasks to operate on the distributed object. For instance, a distributed vector will have an ID for each part of it lying in a remote processor. With this array of IDs the user is able to submit tasks to operate on the vector.

A distributed object is composed of objectIDs and object versions. The version is incremented whenever the object is modified. Figure 2 illustrates a distributed object.

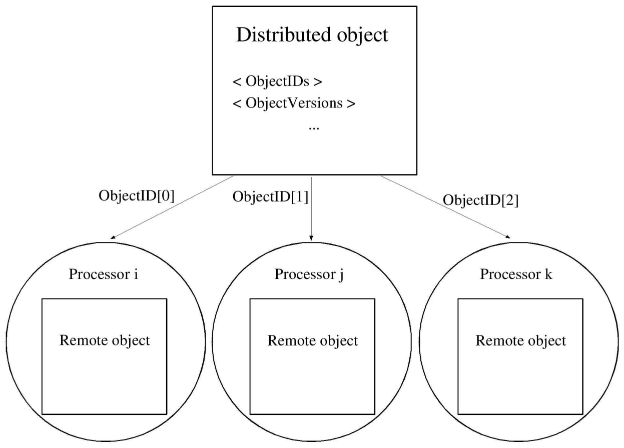

Fig. 2. Distributed object

3.2 Distributed tasks

In the C++ programming language there is a separation of data and program. A data object is distributed and can be transferred between processors but a method can not be. A task is used to perform a method on distributed objects.

A task is created basically with its dependent data and the processor of execution. When a task is created it is necessary to inform the task its dependencies. A task has objects as dependency. For instance, if a task operates on a vector, it is necessary to add the objectID of this vector, the access request (read or write access) and the required version of the vector. Figure 3 illustrates a set of tasks.

All tasks in the program must be objects derived from the class OOPTask. It is necessary to implement the method Execute on these derived tasks. This method implements the operations the task perform on the dependent data. When the method Execute is performed all objects are available to the task to be operated on the execution processor. It means the Execute method operates on objects as the serial method does.

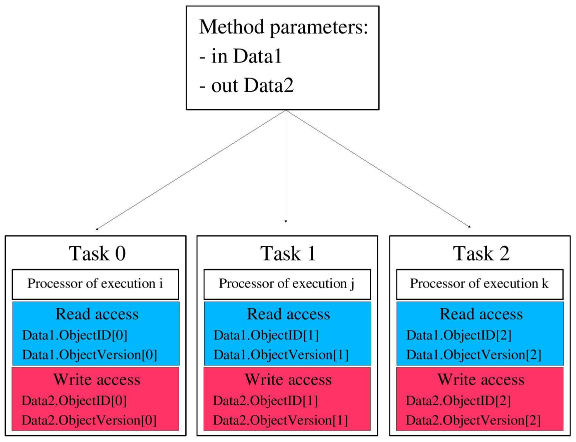

Fig. 3. Distributed tasks

In the parallelization of the CFD code the Execute method of derived tasks calls the serial method of the dependent objects. For that reason the operations of the parallel version are exactly the same of the serial version. This feature was very useful in the debugging processing when the serial and parallel execution were compared to validate the code parallelization.

When the task is submitted, oopar put the task in queue waiting to dependencies to be satisfied. A dependency is satisfied when the object reach the requested version. Once the dependencies are satisfied the task is put on execution.

The dependency data must be available to the task on the execution processor. All communication of object data are performed by oopar before the task is put on execution. When a read access is requested to the object, oopar transfer a copy of the object. When a write access is requested the oopar transfer the object, deleting it from the original processor.

4 Sub-structuring

The original finite volume mesh is partitioned in a separated module of the program using Metis [6]. The partitioned mesh is saved in two cgns files [4], one containing the grid and other containing the initial solution.

The cgns grid file does not contain clone volumes. They will be created on the main program while the sub-meshes are read. Clone volumes are created in boundaries between sub-meshes in both sides of the boundary.

A clone volume is the representation of an actual volume lying in a different sub-mesh. Its function is to store solution values to be used in flux computations.

In the main program the sub-meshes are read one by one on processor 0 and are sent to processors where they will reside. Rather than partitioning a global mesh into sub-meshes, a symbolic representation of the global mesh is constructed while the sub-meshes are read.

The sub-meshes and the master mesh are implemented by objects of the same class. The master mesh has an array containing the sub-meshes objectIDs and another array to store their object versions.

The object mesh has a set of volumes and a set of faces as attribute. The sets of faces and volumes of each sub-mesh are represented in the master mesh, except for sets of clone volumes. Clone volumes represent actual volumes. And these actual volumes are represented in the master mesh. That representation of sets does not include the volume and face data themselves what would require a lot of memory. Only the sets are represented. It is illustrated in figure 4.

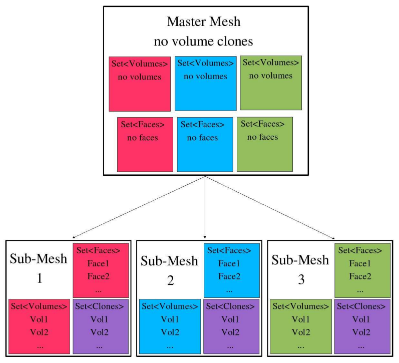
Master Mesh
no volume clones
Set<Volumes> no volumes
Set<Volumes> no volumes
Set<Volumes> no volumes
Set<Faces> no faces
Set<Faces> no faces
Set<Faces> no faces
Sub-Mesh 1
Set<Faces> Face1 Face2 ...
Set<Volumes> Vol1 Vol2 ...
Set<Clones> Vol1 Vol2 ...
Sub-Mesh 2
Set<Faces> Face1 Face2 ...
Set<Volumes> Vol1 Vol2 ...
Set<Clones> Vol1 Vol2 ...
Sub-Mesh 3
Set<Faces> Face1 Face2 ...
Set<Volumes> Vol1 Vol2 ...
Set<Clones> Vol1 Vol2 ...

Fig. 4. Master mesh representation

For each set of volume in the remote sub-meshes there is an equivalent set of volume in the master mesh, except for the sets of clone volumes. Clone volumes are not present in the master mesh because they do not correspond to actual volumes. Instead they clone an actual volume lying in another processor.

The master mesh is responsible for making interface between the original code and all remote meshes. The original code will operate on the master mesh with no difference from the serial version of the code.

Flux and volume operators are created based on the master mesh. There are a set of fluxes and volume operators that must be evaluated on the mesh. Flux and volume operators are distributed in order to have the same operations acting on each sub-mesh. The flux and volume operator objects have also the concept of master object. But differently from the mesh that is an agglomeration of remote objects, flux and volume operator objects are created and then distributed.

These master flux and volume operator objects also contain an array of objectIDs and object versions. They are responsible for creating and submitting tasks to perform the distributed computations. Each flux or volume operation computation which operate on the master mesh will spawn computations on the sub-meshes.

As mentioned, clone volumes are created in both sides of boundaries between sub-meshes. The process of computing a flux becomes:

- update clone values if necessary
- create tasks to perform flux computation on all sub-meshes

Flux tasks call the serial code to perform actual flux computation.

Volume operations are evaluated only on actual volumes. Tasks are created to perform the volume operator on remote processors.

5 Synchronization points

As described in section 2 the algorithm of the serial code involves a loop over time steps and the call of a time integrator to perform each time step computation.

After each time step a convergence check is performed to verify if the steady state solution is achieved. It is done in the CFD code by evaluating the norm of two different solutions and comparing this result with a convergence parameter ϵ. It can be stated as

$$\| W^i - W^{i-1} \| < \epsilon$$

for a given definition of norm.

The parallel implementation of the CFD code has few points of synchronization. These are the points where norms or dot products are computed. It is performed to check the problem convergence and on iterative linear solvers as GMRES where dot products are computed.

Fluxes and volume operators have no synchronization points. It is due to the version control of oopar which allows the user to create lots of tasks based only on the version of dependent objects and leaving to oopar the responsibility of transferring data and the responsibility of granting access to dependent objects. Synchronization is performed by the dependency of tasks on data objects with specific versions. The program does not take care about it. It is performed by oopar.

6 Implemented tasks on the CFD code

The master execution acts on the remote objects through tasks derived from OOPTask.

A task is responsible for executing a method on a remote processor. Since a method act on a data object, it is necessary to guarantee the data will be available in the remote processor with the appropriate version. Only when the data reaches the correct version oopar transfers the data to the requesting processor. OOPar takes care by itself of the whole process of transferring data between processors.

6.1 Flux and volume operator tasks

In order to parallelize the CFD code a number of tasks were derived from OOP-Task. In these tasks the Execute method call the serial method on this data performing the same computation of the serial code.

Only the code of flux and volume operator method had to be modified to support the parallelization. The methods ContributeFlux and ExecuteOperation are the interface of all fluxes and volume operator computations. Figure 1 shows these methods. These methods initialize the necessary data structure and call the abstract method ComputeFlux or OperateData that are implemented on derived objects. Only the methods ContributeFlux and ExecuteOperation were modified in the parallelization.

If the flux or volume operator is master (it means it is distributed) then tasks are created to compute on remote objects. If the objects on which the flux or volume operator operates are not distributed (serial code or remote objects on parallel execution) the serial code is evaluated.

6.2 Sending objects

Beside evaluating methods on remote processors, tasks may be used to transfer objects from one machine to another. In the CFD code, after each sub-mesh is read from file, it is necessary to send these meshes to the processors where they will reside.

In order to transfer these sub-meshes it is created a number of tasks that perform no computation. The sub-mesh is added as a write dependent data object of the task. It leads oopar to transfer the sub-mesh object to the processor of execution of the task. Because it is a write access request the object will be transferred to the processor of destination. As the task performs no computation the task is finished after the transferring. When the task is finished it is released the access to the sub-mesh object for other tasks.

The sub-mesh object will stay on the processor it was sent until another task request a write access on a different processor. That request will happen on the CFD code only in the ending in order to save the state solution obtained.

Once the sub-mesh is transferred another sub-mesh may be read. This process of read and send the sub-meshes aims to not have all sub-meshes in the current processor at same time. If it happened it could be prohibitive due to the excessive memory requirements.

6.3 Bringing objects and synchronization points

Another need of parallel codes is to implement synchronization points. With oopar it is performed by the OOPWaitTask object. It is used in the code to compute norms and dot products.

The process consists of computing the dot product, for instance, at each processor separately. And an OOPDouble object is used to create a global object to accumulate the dot product of each remote object, resulting in the the dot product of the distributed objects.

In this example OOPWaitTask was used to make a synchronization point waiting for the final result of the OOPDouble. Wait task can also be used to bring any object from its processor to the current processor. It is used, for instance, to save solution. Each remote solution vector is transferred to the current processor to save the solution.

7 Debugging process

Having the program structured using tasks and data allowed to gradually debug the parallel code. In the first test a single partition was adopted. Once it was validated the program was tested running with more partitions. The process of debugging these tests followed the same steps as follow.

Firstly all tasks were executed on the current processor, allowing to debug the operation of the tasks without transmitting data from one processor to another.

Secondly the tasks were distributed over the pool of available processors, allowing to verify the consistency of transmission of data.

Both serial code and parallel implementation coexist. This allowed to verify the results of the parallel code with the results of the serial code at each step of the debugging process. Instead of having empty master mesh we kept it as original and also performed the sub-structuring process. In the figure 4 it is shown that the master mesh contains all the set of volumes. As described in section 4 the master mesh does not contain any volume or face data. Only the sets of remote objects are represented but these sets are empty. In the debugging process these sets are filled with the volume and face data from the remote objects. With that process the master mesh contains the structure of the master object necessary to the parallel execution and also the data necessary to the serial execution. The process of filling the master mesh with the face and volume data from the remote objects was implemented with an #ifdef directive. Compiling the code with the directive TEST_PARALLEL produces the debugging code.

We had both serial and parallel objects on same time. It allowed us to check the consistency of the parallel execution at any time by comparing it with the serial execution. Computing serial methods and creating parallel tasks we could compare both results. Any deviation was detected instantly.

8 Performance evaluation

At this point only validation tests were performed. It is necessary to evaluate the performance of the parallel code. This work is still remaining and some tests are foreseen.

1. The first proposed test aims to compare the serial to parallel execution with only one sub-mesh. The serial code computation is based on methods and the parallel code is based on tasks. The process of creating and management of tasks has certainly a cost. Evaluate that cost is the goal of this test.

2. A second test consists of running the parallel code with n sub-meshes in one machine. It is expected with this test to evaluate the computational cost of additional tasks and communication tasks. With n sub-meshes clone volumes are created and communication between the clone and its actual volume is necessary. This test can be performed on single or multiple threads.
3. The third test consists of running the parallel code on several machines to evaluate the communication cost between machines. With several sub-meshes residing in different machines the impact of the communication between processors can be evaluated. Is is also possible to evaluate the global performance of the parallel code.

9 Conclusions

The parallelization of the CFD code demonstrates that OOPar is a very useful tool on the development of parallel codes.

It was possible to parallelize a complex code with small impact on the structure of the original code with serial and parallel version of the code coexisting.

The parallelization was based on tasks and distributed objects. The oopar control version of data transfers to oopar the responsibility of managing the synchronization points. It is also responsibility of oopar to transfer data between processors.

The coexistence of the serial and parallel code was important in the debugging process allowing to compare both executions at any point we wanted. The parallel code developed produces identical results as the serial code.

References

1. P.R.B. Devloo, F.A.M. Menezes, and E.C. Silva. OOPAR : The development of an environment for parallel computing using the object oriented programming philosophy. In B.H.V. Topping, editor, *Advances in Computational Structures Technology*, pages 151–156, 10 Saxe-Coburg Place, Edinburgh, EH3 5BR, UK, 1996. CIVIL-COMP Press.
2. Chris Ding and Yun He. A ghost cell expansion method for reducing communications in solving pde problems. In *SC2001 - International Conference for High Performance Computing and Communications*, 2001.
3. Jean-Frédéric Gerbeau, Nathalie Glinsky-Olivier, and Bernard Larrouturou. Semi-implicit roe-type fluxes for low-mach numbers flows. Rapport de recherche 3132, INRIA, Mars 1997.
4. CGNS Group. *CFD General Notation System - Overview and Entry-Level Document.* CGNS Group, January 2002.
5. A. Jameson, W. Schimidt, and E. Turkel. Numerical solution of the euler equations by finite volume methods using runge-kutta time stepping schemes. In *Proceedings of the AIAA 14th Fluid and Plasma Dynamics Conference*, 1981.
6. George Karypis and Vipin Kumar. METIS – *A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings os sparse matrices.* University of Minnesota, Dept. of Computer Science, sep 1998.

7. Gustavo C. Longhin and Philippe R. B. Devloo. Parallelization of a scientific code using oopar. In *IBERIAN LATIN-AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING*, volume XXIV. ABMEC, 2003.
8. Clovis Maliska. *Transferencia de Calor e Mecânica dos Fluidos Computacional.* Livros Técnicos e Científicos Editora S.A., Rio de Janeiro. Brasil, 1995.
9. T. E. Tezduyar. Cfd methods for three-dimensional computation of complex flow problems. *Journal of Wind Engineering and Industrial Aerodynamics*, 81, 1999.